SAT Solver verification

By Filip Marić

May 28, 2015

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 backtrack-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
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
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 ?case
    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 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 \land 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 \land uniq l2 \land set l1 \cap set l2 = {})
  (is \(?lhs = ?rhs\))
  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 \((\text{butlast } l)\)
  proof (cases l = [])
    case True
    thus \(?thesis\)
    using assms
    by simp
  next
case False
  hence \(l = \text{butlast } l @ [\text{last } l]\)
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 uniqHeadTailSet:
  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
shows \( \exists \ a_1 \ a_2. \ a_1 \in \text{set } l \land \ a_2 \in \text{set } l \land \ a_1 \neq \ a_2 \)

proof -
  let \(?a_1 = l ! 0\)
  let \(?a_2 = l ! 1\)
  have \(?a_1 \in \text{set } l\) using nth-mem[of 0 l]
  using assms by simp
  moreover
  have \(?a_2 \in \text{set } l\) using nth-mem[of 1 l]
  using assms by simp
  moreover
  have \(?a_1 \neq \ ?a_2\)
  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 \textit{firstPos} - first position of an element

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

\textbf{primrec} \texttt{firstPos :: 'a => 'a list => nat}
\textbf{where}

\begin{itemize}
  \item \texttt{firstPos a [] = 0 |}
  \item \texttt{firstPos a (h # t) = (if a = h then 0 else 1 + (firstPos a t))}
\end{itemize}

\textbf{lemma} \texttt{firstPosEqualZero:}
\hspace{1em} shows \((firstPos a (m # M') = 0) = (a = m)\)
\hspace{1em} by (induct M') auto

\textbf{lemma} \texttt{firstPosLeLength:}
\hspace{1em} assumes \(a \in \text{set } l\)
\hspace{1em} shows \(\text{firstPos a l} < \text{length l}\)
\hspace{1em} using assms
\hspace{1em} by (induct l) auto

\textbf{lemma} \texttt{firstPosAppend:}
\hspace{1em} assumes \(a \in \text{set } l\)
\hspace{1em} shows \(\text{firstPos a l} = \text{firstPos a (l @ l')}\)
\hspace{1em} using assms
by (induct l) auto

lemma firstPosAppendNonMemberFirstMemberSecond:
  assumes a ∉ set l1 and a ∈ set l2
  shows firstPos a (l1 @ l2) = length l1 + firstPos a l2
using assms
by (induct l1) auto

lemma firstPosDomainForElements:
  shows (0 ≤ firstPos a l ∧ firstPos a l < length l) = (a ∈ set l) (is ?lhs = ?rhs)
by (induct l) auto

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

note \( ** = \text{this} \)
moreover
{
  assume \( Q \): \( a \neq m \ b \neq m \)
  from \( Q \ (a \in \text{set} \ (m \# l')) \)
  have \( a \in \text{set} \ l' \)
    by simp
  from \( Q \ (b \in \text{set} \ (m \# l')) \)
  have \( b \in \text{set} \ l' \)
    by simp
  from \( a \in \text{set} \ l' \ b \in \text{set} \ l' \ \text{Cons} \)
  have \( \text{firstPos} \ a \ l' = \text{firstPos} \ b \ l' \)
    by \( (\text{simp split: split-if-asn}) \)
  with Cons
  have \( ?\text{case} \)
    by \( (\text{simp split: split-if-asn}) \)
}

note \( *** = \text{this} \)
moreover
{
  have \( a = m \lor b = m \lor a \neq m \land b \neq m \)
    by auto
}
ultimately
show \( ?\text{thesis} \)
proof \( (\text{cases} \ a = m) \)
  case True
  thus \( ?\text{thesis} \)
    by \( (\text{rule} \ *) \)
next
  case False
thus \(?thesis\)
proof (cases b = m)
  case True
  thus \(?thesis\)
  by (rule **) 
next 
  case False
  with \(<a \neq m>\) show \(?thesis\)
  by (rule ***)
qed 
qed 
qed simp
}
thus \(?thesis\)
by auto
qed

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

1.5 \(\textit{precedes} - \text{ordering relation induced by} \ \textit{firstPos}\)

definition precedes :: \(\textit{a} \Rightarrow \textit{a} \Rightarrow \textit{list} \Rightarrow \textit{bool}\)
where
precedes \(a\ b\ l\ == (a \in\ \textit{set\ l} \land\ b \in\ \textit{set\ l} \land\ \textit{firstPos\ a\ l} <\ \textit{firstPos\ b\ l})\)

lemma noElementsPrecedesFirstElement:
  assumes a \# b
  shows \(\neg\ \textit{precedes\ a\ b\ (b\ #\ list)}\)
proof -
  
  assume precedes \(a\ b\ (b\ #\ list)\)
  hence \(a \in\ \textit{set\ (b\ #\ list)}\ \textit{firstPos\ a\ (b\ #\ list)} <\ 0\)
  unfolding precedes-def
  by (auto split: split-if-asm)
  hence \(\textit{firstPos\ a\ (b\ #\ list)} = 0\)
  by auto
  with \(<a \neq b>\)
  have \(\textit{False}\)
  using \(\textit{firstPosEqualZero}[a\ b\ list]\)
  by simp
  
  thus \(?thesis\)
  by auto
Qed
lemma lastPrecedesNoElement:
assumes uniq l
shows \( \neg (\exists \ a. \ a \neq \text{last} \ l \land \text{precedes} \ (\text{last} \ l) \ a \ l) \)
proof-
{ 
assume \( \neg \ ?\text{thesis} \)
then obtain \( a \)
  where \( \text{precedes} \ (\text{last} \ l) \ a \ l \ a \neq \text{last} \ l \)
  by auto
hence \( a \in \text{set} \ l \text{ last} \ l \in \text{set} \ l \text{ firstPos} (\text{last} \ l) \ l \leq \text{firstPos} a \ l \)
  unfolding \precedes\text{-def}
  by auto
hence \( \text{length} \ l - 1 \leq \text{firstPos} a \ l \)
  using \text{firstPosLast}[of \ l \ \text{last} \ l]
  using \( (\text{uniq} \ l) \)
  by force
hence \( \text{firstPos} a \ l = \text{length} \ l - 1 \)
  using \text{firstPosAppend}[of \ a \ l \ \text{last} \ l]
  using \( (a \in \text{set} \ l) \)
  by auto
hence \( a = \text{last} \ l \)
  using \text{firstPosLast}[of \ l \ \text{last} \ l]
  using \( (a \in \text{set} \ l) \ \text{last} \ l \in \text{set} \ l \)
  using \( (\text{uniq} \ l) \)
  using \text{firstPosEqual}[of \ a \ l \ \text{last} \ l]
  by force
with \( (a \neq \text{last} \ l) \)
have \( \text{False} \)
  by simp
}
thus \?thesis
  by auto
qed

lemma precedesAppend:
assumes \text{precedes} \ a \ b \ l
shows \text{precedes} \ a \ b \ (\text{l @} \ l')
proof-
from \( \text{precedes} \ a \ b \ b \)
have \( a \in \text{set} \ l \ b \in \text{set} \ l \text{ firstPos} a \ l \leq \text{firstPos} b \ l \)
  unfolding \precedes\text{-def}
  by (auto split: split-if-asm)
thus \?thesis
  using \text{firstPosAppend}[of \ a \ l \ l']
  using \text{firstPosAppend}[of \ b \ l \ l']
  unfolding \precedes\text{-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 $\text{precedes } a \ b \ (l_1 \oplus l_2)$
proof
  from $(a \in \text{set } l_1)$
  have $\text{firstPos } a \ l_1 < \text{length } l_1$
    using $\text{firstPosLeLength [of } a \ l_1]\$
    by simp
  moreover
  from $(a \in \text{set } l_1)$
  have $\text{firstPos } a \ (l_1 \oplus l_2) = \text{firstPos } a \ l_1$
    using $\text{firstPosAppend [of } a \ l_1 l_2]\$
    by simp
  moreover
  from $(b \notin \text{set } l_1)$ $(b \in \text{set } l_2)$
  have $\text{firstPos } b \ (l_1 \oplus l_2) = \text{length } l_1 + \text{firstPos } b \ l_2$
    by (rule $\text{firstPosAppendNonMemberFirstMemberSecond}\$
  moreover
  have $\text{firstPos } b \ l_2 \geq 0$
    by auto
  ultimately
  show $\text{thesis}$
    unfolding $\text{precedes-def}$
    using $(a \in \text{set } l_1)$ $(b \in \text{set } l_2)$
    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 \texttt{assms}
  have \texttt{firstPos a l = firstPos b l}
    unfolding \texttt{precedes-def}
  by \texttt{auto}
  thus \texttt{thesis}
    using \texttt{firstPosEqual[of a b]}\[1\]
    using \texttt{assms}
    by \texttt{simp}
qed

lemma \texttt{precedesTotalOrder}: 
  assumes \( a \in \text{set l} \) and \( b \in \text{set l} \)
  shows \( a=b \lor \text{precedes a b l} \lor \text{precedes b a l} \)
  using \texttt{assms}
  unfolding \texttt{precedes-def}
  by \texttt{auto}

lemma \texttt{precedesMap}: 
  assumes \( \text{precedes a b list} \) and \( \forall x y. x \neq y \longrightarrow f x \neq f y \)
  shows \( \text{precedes} (f a) (f b) (\text{map f list}) \)
  using \texttt{assms}
  proof \texttt{(induct list)}
    case \texttt{(Cons l list')} 
    { 
      assume \( a = l \)
      have \( b \in \text{set l} \)
        proof --
          from \( a = l \)
          have \( \text{firstPos} (f a) (\text{map f} (l \# \text{list'})) = 0 \)
            using \texttt{firstPosEqualZero[of f a f l map f list']}\[2\]
            by \texttt{simp}
          moreover
          from \( \text{precedes} a b (l \# \text{list'}) \)
          have \( b \in \text{set} (l \# \text{list'}) \)
            unfolding \texttt{precedes-def}
            by \texttt{simp}
          hence \( f b \in \text{set} (\text{map f} (l \# \text{list'})) \)
            by \texttt{auto}
          moreover
          hence \( \text{firstPos} (f b) (\text{map f} (l \# \text{list'})) \geq 0 \)
            by \texttt{auto}
          ultimately
          show \( \texttt{thesis} \)
            using \( (a = l; f b \in \text{set} (\text{map f} (l \# \text{list'}))) \)
            unfolding \texttt{precedes-def}
            by \texttt{simp}
    }
    qed
moreover

\{ 
  assume \( b = l \) 
  with \((\text{precedes} \ a \ b \ (l \ # \ \text{list}'))\) 
  have \( a = l \) 
  \quad \text{using} \ \text{noElementsPrecedesFirstElement}[\text{of} \ a \ \text{list}'] 
  \quad \text{by} \ \text{auto} 
  \quad \text{from} \ (a = b) \ (b = l) 
  have \ ?\text{case} 
  \quad \text{unfolding} \ \text{precedes-def} 
  \quad \text{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 \( f \ a \neq f \ l \ f \ b \neq f \ l \) 
  \quad \text{by} \ \text{auto} 
  \quad \text{from} \ (\text{precedes} \ a \ b \ (l \ # \ \text{list}')) 
  \quad \text{have} \ b \in \text{set}(l \ # \ \text{list}') \ a \in \text{set}(l \ # \ \text{list}') \ \text{firstPos} \ a \ (l \ # \ \text{list}') \leq \ \text{firstPos} \ b \ (l \ # \ \text{list}') 
  \quad \text{unfolding} \ \text{precedes-def} 
  \quad \text{by} \ \text{auto} 
  \quad \text{with} \ (a \neq l) \ (b \neq l) 
  \quad \text{have} \ a \in \text{set} \text{ list}' \ b \in \text{set} \text{ list}' \ \text{firstPos} \ a \ \text{list}' \leq \ \text{firstPos} \ b \ \text{list}' 
  \quad \text{by} \ \text{auto} 
  \quad \text{hence} \ \text{precedes} \ a \ b \ \text{list}' 
  \quad \text{unfolding} \ \text{precedes-def} 
  \quad \text{by} \ \text{simp} 
  \quad \text{with} \ \text{Cons} 
  \quad \text{have} \ \text{precedes} \ (f \ a) \ (f \ b) \ (\text{map} \ f \ \text{list}') 
  \quad \text{by} \ \text{simp} 
  \quad \text{with} \ (f \ a \neq f \ b) \ (f \ b \neq f \ b) 
  \quad \text{have} \ ?\text{case} 
  \quad \text{unfolding} \ \text{precedes-def} 
  \quad \text{by} \ \text{auto} 
\}

ultimately 
show \ ?\text{case} 
  \quad \text{by} \ \text{auto} 

next 
  case \ \text{Nil} 
  \quad \text{thus} \ ?\text{case} 
  \quad \text{unfolding} \ \text{precedes-def} 
  \quad \text{by} \ \text{simp} 
\qed 

lemma \ \text{precedesFilter}:
assumes \( \text{precedes} \ a \ b \ \text{and} \ f \ a \ \text{and} \ f \ b \)
shows \( \text{precedes} \ a \ b \ (\text{filter} \ f \ \text{list}) \)
using \( \text{assms} \)
proof (induct \( \text{list} \))
case (\text{Cons} \ l \ \text{list}')
show \( ?\text{case} \)
proof -
  from \( \text{precedes} \ a \ b \ (l \ # \ \text{list}') \)
  have \( a \in \text{set}(l \ # \ \text{list}') \ b \in \text{set}(l \ # \ \text{list}') \ \text{firstPos} \ a \ (l \ # \ \text{list}') \leq \ \text{firstPos} \ b \ (l \ # \ \text{list}') \)
  unfolding \( \text{precedes-def} \)
  by auto
  from \( f \ a \ \langle a \in \text{set}(l \ # \ \text{list}') \rangle \)
  have \( a \in \text{set}(\text{filter} \ f \ (l \ # \ \text{list}')) \)
  by auto
moreover
  from \( f \ b \ \langle b \in \text{set}(l \ # \ \text{list}') \rangle \)
  have \( b \in \text{set}(\text{filter} \ f \ (l \ # \ \text{list}')) \)
  by auto
moreover
have \( \text{firstPos} \ a \ (\text{filter} \ f \ (l \ # \ \text{list}')) \leq \ \text{firstPos} \ b \ (\text{filter} \ f \ (l \ # \ \text{list}')) \)
proof -
  \{
    assume \( a = l \)
    with \( f \ a \) \( \langle a \in \text{set}(l \ # \ \text{list}') \rangle \)
    have \( \text{firstPos} \ a \ (\text{filter} \ f \ (l \ # \ \text{list}')) = 0 \)
    by auto
    with \( \langle b \in \text{set} \ (\text{filter} \ f \ (l \ # \ \text{list}')) \rangle \)
    have \( ?\text{thesis} \)
    by auto
  
moreover
  \{
    assume \( b = l \)
    with \( \text{precedes} \ a \ b \ (l \ # \ \text{list}') \)
    have \( a = b \)
    using \( \text{noElementsPrecedesFirstElement[of} \ a \ b \ \text{list}'] \)
    by auto
    hence \( ?\text{thesis} \)
    by \((\text{simp add: precedesReflexivity})\)
  
moreover
  \{
    assume \( a \neq l \ b \neq l \)
    with \( \text{precedes} \ a \ b \ (l \ # \ \text{list}') \)
    have \( \text{firstPos} \ a \ \text{list}' \leq \ \text{firstPos} \ b \ \text{list}' \)
    unfolding \( \text{precedes-def} \)
    by auto
moreover
from \( a \neq b \) \( \langle a \in \text{set} (l \# \text{list}') \rangle \)
have \( a \in \text{set list}' \)
  by simp
moreover
from \( b \neq l \) \( \langle b \in \text{set} (l \# \text{list}') \rangle \)
have \( b \in \text{set list}' \)
  by simp
ultimately
have \( \text{precedes a b list}' \)
  unfolding precedes-def
  by simp
with \( \langle f a \rangle \langle f b \rangle \text{Cons}(1) \)
have \( \text{precedes a b (filter f list')} \)
  by simp
with \( \langle a \neq l \rangle \langle b \neq l \rangle \)
have \(?thesis\)
  unfolding precedes-def
  by auto
}
ultimately
show \(?thesis\)
  by blast
qed
ultimately
show \(?thesis\)
  unfolding precedes-def
  by simp
qed

definition
precedesOrder list == \{(a, b). precedes a b list \land a \neq b\}

lemma transPrecedesOrder:
trans (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 precedesTransitivity[of x y list z]
  using firstPosEqual[of y list z]
  unfolding precedes-def
  by auto
}
thus \(?thesis\)
  unfolding trans-def
  unfolding precedesOrder-def
  by blast

qed

simp
lemma wellFoundedPrecedesOrder:
  shows $\text{wf}(\text{precedesOrder list})$
unfolding $\text{wf-eq-minimal}$
proof -
  show $\forall Q. a : Q \rightarrow (\exists a\text{Min} \in Q. \forall a'. (a', a\text{Min}) \in \text{precedesOrder list} \rightarrow a' \notin Q)$
proof -
  { fix $a : 'a$ and $Q : 'a$ set
    assume $a \in Q$
    let $?\text{listQ} = \text{filter}(\lambda x. x \in Q)$ list
    have $\exists a\text{Min} \in Q. \forall a'. (a', a\text{Min}) \in \text{precedesOrder list} \rightarrow a' \notin Q$
    proof (cases $?\text{listQ} = []$)
      case True
      let $?a\text{Min} = a$
      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 $a \in \text{set list}$
        unfolding $\text{precedesOrder-def}$
        unfolding $\text{precedes-def}$
        by simp
        with $(a \in Q)$
        have $a \in \text{set $?\text{listQ}}$
        by (induct list) auto
        with $(?\text{listQ} = [])$
        have False
        by simp
        hence $a' \notin Q$
        by simp
      }
      thus $?\text{thesis}$
      by simp
    qed
    with $(a \in Q)$ obtain $a\text{Min}$ where $a\text{Min} \in Q \forall a'. (a', a\text{Min}) \in \text{precedesOrder list} \rightarrow a' \notin Q$
    by auto
    thus $?\text{thesis}$
    by auto
  next
    case False
    let $?a\text{Min} = \text{hd} ?\text{listQ}$
from False
have ?aMin ∈ Q
by (induct list) auto
have ∀ a′. (a′, ?aMin) ∈ precedesOrder list → a′ /∈ Q
proof
  fix a′
  { assume (a′, ?aMin) ∈ precedesOrder list
    hence a′ ∈ set list precedes a′ ?aMin list a′ ≠ ?aMin
      unfolding precedesOrder-def
      unfolding precedes-def
      by auto
    have a′ /∈ Q
      proof−
      { assume a′ ∈ Q
        with : ?aMin ∈ Q (precedes a′ ?aMin list)
        have precedes a′ ?aMin ?listQ
          using precedesFilter[of a′ ?aMin list λ x. x ∈ Q]
          by blast
        from (a′ ≠ ?aMin)
        have ¬ precedes a′ (hd ?listQ) (hd ?listQ # tl ?listQ)
          by (rule noElementsPrecedesFirstElement)
        with False (precedes a′ ?aMin ?listQ)
        have False
          by auto
      }
      thus ?thesis
      by auto
    qed
  } thus (a′, ?aMin) ∈ precedesOrder list → a′ /∈ Q
    by simp
  qed
with (?aMin ∈ Q)
show ?thesis
  ..
  qed
qed

1.6 isPrefix - prefixes of list.

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

definition
  isPrefix :: 'a list => 'a list => bool
where isPrefix p t = (∃ 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 (simp add: firstPos_def)
  with ⟨a ∈ set p, b ∈ set p⟩
  show ?thesis
  using firstPosAppend[of a p s]
  by simp
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 (simp add: firstPos_def)
  with ⟨a ∈ set p, b ∈ set p⟩
  show ?thesis
  using firstPosAppend[of a p s]
  by simp
qed
by (auto split: split-if-asm)

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

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

  from \(\text{firstPos} \ a \ l = \text{firstPos} \ a \ p\), \(\text{firstPos} \ b \ l = \text{firstPos} \ b \ p\)
    \(\text{firstPos} \ a \ l \leq \text{firstPos} \ b \ l\),
  \(a \in \text{set} \ p\), \(b \in \text{set} \ p\)
  show \(?\text{thesis}\)
    unfolding \(\text{precedes-def}\)
  by simp
next
    case False
  from \(a \notin \text{set} \ p\), \(a \in \text{set} \ l\), \(p \qdot s = l\)
  have \(a \in \text{set} \ s\)
    by auto
  with \(a \notin \text{set} \ p\), \(p \qdot s = l\)
  have \(\text{firstPos} \ a \ l = \text{length} \ p + \text{firstPos} \ a \ s\)
    using \(\text{firstPosAppendNonMemberFirstMemberSecond} \, [\text{of} \, a \ p \ s]\)
  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 \(\text{firstPos} \ b \ l = \text{firstPos} \ b \ p\), \(\text{firstPos} \ a \ l \leq \text{firstPos} \ b \ l\)
  by simp
qed
qed

1.7 list-diff - the set difference operation on two lists.

primrec list-diff :: \('a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list\)
where
list-diff \(x\) \([]\) = \(x\) |
list-diff \(x\) \((y\#ys)\) = 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 ∈ 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 ∈ 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 ∈ set a ∧ x /∈ set b) = (x ∈ set (list-diff a b))
by (induct a) auto

lemma listDiffDoubleRemoveAll:
  assumes x ∈ 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 /∈ set a
  shows list-diff a b = list-diff a (removeAll x b)
using assms
proof (induct b arbitrary: a)
  case (Cons y b')
  from ⟨x /∈ set a⟩
  have x /∈ set (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 /∈ set (removeAll y a)⟩
    by auto
next
  case True
  thus ?thesis
    using Cons (1) [of removeAll y a]
    using ⟨x /∈ set a⟩
    using ⟨x /∈ set (removeAll y a)⟩
    by auto
qed simp

lemma listDiffMap:
  assumes ∀ x y. x ≠ y → f x ≠ f y
  shows map f (list-diff a b) = list-diff (map f a) (map f b)
using assms
by (induct b arbitrary: a) (auto simp add: removeAll-map)

1.8 remdups - removing duplicates

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

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

lemma remdupsAppendSet:
  shows set (remdups (a @ b)) = set (remdups a @ remdups (list-diff b a))
proof (induct a)
  case Nil
thus ?case
  by auto
next
  case (Cons x a')
  thus ?case
  proof (cases x ∈ set a')
    case True
    thus ?thesis
    using Cons
    using listDiffDoubleRemoveAll[of x a' b]
    by simp
  next
    case False
    thus ?thesis
    proof (cases x ∈ set b)
      case True
      show ?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 /∈ set a'⟩
        by auto
        also have ... = set (x # remdups a' @ remdups (list-diff removeAll x b) a')
        by auto
        also have ... = set (x # remdups a' @ remdups (removeAll x (list-diff b a')))
        by simp
        also have ... = set (remdups a' @ x # remdups (removeAll x (list-diff b a')))
        by simp
        also have ... = set (remdups a' @ x # removeAll x (remdups (list-diff b a')))
        by (simp only: remdupsRemoveAllCommute)
        also have ... = set (remdups a') ∪ set (x # removeAll x (remdups (list-diff b a')))
        by simp
        also have ... = set (remdups a') ∪ {x} ∪ set (removeAll x (remdups (list-diff b a')))
        by auto
        also have ... = set (remdups a') ∪ set (remdups (list-diff b a'))
      proof
        from ⟨x /∈ set a'⟩ ⟨x ∈ set b⟩
        have x ∈ set (list-diff b a')
        using listDiffIff[of x b a']
        by simp
        hence x ∈ set (remdups (list-diff b a'))
by auto
thus ?thesis
by auto
qed
also have ... = set (remdups (a' @ b))
  using Cons(1)
  by simp
also have ... = set (remdups ((x ≠ a') @ b))
  using (x ∈ 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 ∉ set a')
    by auto
    also have ... = set (x ≠ remdups a' @ remdups (list-diff (removeAll x b) a'))
      by auto
    also have ... = set (x ≠ remdups a' @ remdups (list-diff b a'))
      using (x ∉ set b)
      by auto
    also have ... = set (remdups ((x ≠ a') @ b))
      by auto
finally show ?thesis
      by simp
qed
qed
qed
lemma remdupsAppendMultiSet:
  shows multi-of (remdups (a @ b)) = multi-of (remdups a @ remdups (list-diff b a))
proof (induct a)
case Nil
  thus ?case
  by auto
next
case (Cons x a')
thus ?case
proof (cases x ∈ set a')
  case True
  thus ?thesis
  proof
  case (cases x ∈ set a')
  thus ?thesis
  using Cons
  using listDiffDoubleRemoveAll[of x a' b]
  by simp
next
  case False
  thus ?thesis
proof (cases x ∈ set b)
  case True
  show ?thesis
  proof
  have multiset-of (remdups (x # a') @ remdups (list-diff b (x # a')))
  proof
  have remdups (x # a') = x # remdups a'
  using x ∈ set a'
  by auto
  thus ?thesis
  by simp
  qed
  also have ... = multiset-of (remdups a' @ remdups (list-diff b a'))
  by auto
  also have ... = multiset-of (remdups a' @ x # remdups (removeAll x (list-diff b a')))
  by simp
  also have ... = multiset-of (remdups a' @ x # remdups (list-diff b a'))
  proof (simp add: union-assoc)
  also have ... = multiset-of (remdups a' @ x # removeAll x (remdups (list-diff b a')))
  by (simp only: remdupsRemoveAllCommuteg)
  also have ... = multiset-of (remdups a') + multiset-of (x # removeAll x (remdups (list-diff b a')))
  by simp
  also have ... = multiset-of (remdups a') + {#x#} + multiset-of (removeAll x (remdups (list-diff b a')))
  proof
  from x ∉ set a' x ∈ set b
  have x ∈ set (list-diff b a')
  using listDiffIff[of x b a']
  by simp
  qed
  proof
  have x ∈ set (list-diff b a')
  using listDiffIff[of x b a']
  by simp
  qed
hence $x \in \text{set } (\text{remdups } (\text{list-diff } b \ a'))$
  by auto
thus ?thesis
  using removeAll-multiset[of \text{remdups } (\text{list-diff } b \ a') \ x]
  by (simp add: union-assoc)
qed

also have ... = \text{multiset-of } (\text{remdups } (a' \ @ \ b))
  using Cons(1)
  by simp
also have ... = \text{multiset-of } (\text{remdups } ((x \ # a') \ @ \ b))
  using ($x \in \text{set } b$)
  by simp
finally show ?thesis
  by simp
qed

next
  case False
  thus ?thesis
  proof
    have \text{multiset-of } (\text{remdups } (x \ # a') \ @ \ \text{remdups } (\text{list-diff } b \ (x \ # a')))
      = \text{multiset-of } (x \ # \text{remdups } a' \ @ \ \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 ?thesis
        by simp
    qed
  also have ... = \text{multiset-of } (\text{remdups } (a' \ @ \ \text{remdups } (\text{list-diff } (\text{removeAll } x \ b) \ a')))
    by auto
  also have ... = \text{multiset-of } (\text{remdups } (x \ # \text{remdups } a' \ @ \ \text{remdups } (\text{list-diff } b \ a')))
    using ($x \notin \text{set } b$)
    using \text{removeAll-id[of } x \ b]
    by simp
  also have ... = \{\#x\} \ + \text{multiset-of } (\text{remdups } (a' \ @ \ b))
    using Cons(1)
    by (simp add: union-commute)
  also have ... = \text{multiset-of } (\text{remdups } ((x \ # a') \ @ \ b))
    using ($x \notin \text{set } a'$ \ $x \notin \text{set } b$)
    by (auto simp add: union-commute)
finally show ?thesis
  by simp
qed

qed


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 a3 + b2]
by simp

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' {##} {#x#}]
  using multisetUnionLeMono1[of r multiset-of (list-diff a' b) multiset-of a' {#x#}]
  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
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 ⊤
    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 ⊤
      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

*consts* member :: 'a ⇒ 'b ⇒ bool (infixl el 55)

*defs* (overloaded)

literalElClause-def [simp]: ((literal::Literal) el (clause::Clause)) == literal ∈ set clause  
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

\textbf{primrec}
\begin{align*}
\textit{var} & : \text{Literal} \Rightarrow \text{Variable} \\
\text{where} & \begin{align*}
\text{var} (\textit{Pos} v) &= v \\
\text{var} (\textit{Neg} v) &= v
\end{align*}
\end{align*}

Set of variables of a given clause, formula or valuation

\textbf{primrec}
\begin{align*}
\textit{varsClause} & : (\text{Literal list}) \Rightarrow (\text{Variable set}) \\
\text{where} & \begin{align*}
\text{varsClause} [] &= \{\} \\
\text{varsClause} (\text{literal} \# \text{list}) &= \{\text{var literal}\} \cup (\text{varsClause list})
\end{align*}
\end{align*}

\textbf{primrec}
\begin{align*}
\textit{varsFormula} & : \text{Formula} \Rightarrow (\text{Variable set}) \\
\text{where} & \begin{align*}
\text{varsFormula} [] &= \{\} \\
\text{varsFormula} (\text{clause} \# \text{formula}) &= (\text{varsClause clause}) \cup (\text{varsFormula formula})
\end{align*}
\end{align*}

\textbf{consts \textit{vars} :: 'a \Rightarrow Variable set}
\textbf{defs \textbf{(overloaded)}}
\begin{align*}
\textit{vars-def-clause} & \ [\textbf{simp}]: \textit{vars} (\text{clause}::\text{Clause}) == \text{varsClause clause} \\
\textit{vars-def-formula} & \ [\textbf{simp}]: \textit{vars} (\text{formula}::\text{Formula}) == \text{varsFormula formula} \\
\textit{vars-def-set} & \ [\textbf{simp}]: \textit{vars} (\textit{s}::\text{Literal set}) == \{\forall \exists \ l. \ l \in s \land \text{var} \ l = vbl\}
\end{align*}

\textbf{lemma clauseContainsItsLiteralsVariable:}
\begin{align*}
\textbf{fixes} & \ \textit{literal} :: \text{Literal} \ \text{and} \ \textit{clause} :: \text{Clause} \\
\textbf{assumes} & \ \textit{literal el clause} \\
\textbf{shows} & \ \text{var literal} \in \text{vars clause} \\
\textbf{using} & \ \text{assms} \\
\textbf{by} \ (\text{induct clause}) \ \text{auto}
\end{align*}

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

qed

lemma formulaContainsItsClausesVariables:
  fixes clause :: Clause and formula :: Formula
  assumes clause el formula
  shows vars clause ⊆ vars formula
  using assms
  by (induct formula) auto

lemma varsAppendFormulae:
  fixes formula1 :: Formula and formula2 :: Formula
  shows vars (formula1 @ formula2) = vars formula1 ∪ vars formula2
  by (induct formula1) auto

lemma varsAppendClauses:
  fixes clause1 :: Clause and clause2 :: Clause
  shows vars (clause1 @ clause2) = vars clause1 ∪ vars clause2
  by (induct clause1) auto

lemma varsRemoveLiteral:
  fixes literal :: Literal and clause :: Clause
  shows vars (removeAll literal clause) ⊆ vars clause
  by (induct clause) auto

lemma varsRemoveLiteralSuperset:
  fixes literal :: Literal and clause :: Clause
  shows vars clause − {var literal} ⊆ vars (removeAll literal clause)
  by (induct clause) auto

lemma varsRemoveAllClause:
  fixes clause :: Clause and formula :: Formula
  shows vars (removeAll clause formula) ⊆ vars formula
  by (induct formula) auto

lemma varsRemoveAllClauseSuperset:
fixes clause :: Clause and formula :: Formula
shows vars formula = vars clause \subseteq vars (removeAll clause formula)
by (induct formula) auto

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

lemma varInFormulaVars:
fixes variable :: Variable and formula :: Formula
shows variable \in vars formula = (\exists literal. literal el formula \land var literal = variable) (is ?lhs formula = ?rhs formula)
proof (induct formula)
case Nil
  show ?case
  by simp

next
case (Cons clause formula)
  show ?case
  proof
    assume P: \?lhs (clause # formula)
    thus \?rhs (clause # formula)
    proof (cases variable \in vars clause)
      case True 
        with varInClauseVars
        have \exists literal. literal el clause \land var literal = variable
        by simp
        thus ?thesis
        by auto
      next
      case False 
        with P 
        have variable \in vars formula
        by simp
        with Cons 
        show ?thesis
        by auto
      qed

next
  assume \?rhs (clause # formula)
  then obtain l
    where lEl: l el clause # formula and varL: var l = variable
    by auto
  from lEl formulaContainsItsLiteralsVariable [of l clause # formula]
  have var l \in vars (clause # formula)
    by auto

34
with varL
show ?lhs (clause # formula)
  by simp
qed
qed

lemma varsSubsetFormula:
  fixes F :: Formula and F' :: Formula
  assumes ∀ c::Clause. c el F → c el 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
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 oppositeIsDifferentFromLiteral [simp]:
fixes literal :: Literal
shows \(\text{opposite literal} \neq \text{literal}\)
by (induct literal) auto

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

lemma literalsWithSameVariableAreEqualOrOpposite:
fixes literal1 :: Literal and 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 literal1)
case Pos
note Pos1 = this
show \(?\text{thesis}\)
proof (cases literal2)
case Pos
with \(?\text{lhs}\) Pos1 show \(?\text{thesis}\)
by simp
next
case Neg
with \(?\text{lhs}\) Pos1 show \(?\text{thesis}\)
by simp
qed
next
case Neg
note Neg1 = this
show \(?\text{thesis}\)
proof (cases literal2)
case Pos
with \(?\text{lhs}\) Neg1 show \(?\text{thesis}\)
by simp
next
case Neg

36
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**

\[
\text{oppositeLiteralList} :: \text{Literal list} \Rightarrow \text{Literal list}
\]

where

\[
\text{oppositeLiteralList} \; \text{clause} = \text{map opposite} \; \text{clause}
\]

**lemma** literalElListIffOppositeLiteralElOppositeLiteralList:

fixes \( \text{literal} :: \text{Literal} \) and \( \text{literalList} :: \text{Literal list} \)

shows \( \text{literal el literalList } = (\text{opposite literal}) \; \text{el} \; (\text{oppositeLiteralList literalList}) \)

**unfolding** oppositeLiteralList-def

**proof** (induct literalList)

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

**next**

- **case** (Cons \( l \) \( \text{literalList} \'))
  - **show** ?case
    - **proof** (cases \( l = \text{literal} \))
      - **case** True
        - thus ?thesis
          - by simp
      - **next**
        - **case** False
          - thus ?thesis
            - by auto
      - qed
    - qed

**lemma** oppositeLiteralListIdempotency [simp]:

fixes \( \text{literalList} :: \text{Literal list} \)

shows \( \text{oppositeLiteralList} \; (\text{oppositeLiteralList} \; \text{literalList}) = \text{literalList} \)

**unfolding** oppositeLiteralList-def

by (induct \( \text{literalList} \)) auto

**lemma** 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
shows vars valuation1 ⊆ vars valuation2
using assms
proof (induct valuation1)
case Nil
show ?case
by simp
next
case (Cons literal valuation)
note caseCons = this
hence literal el valuation2
by auto
with valuationContainsItsLiteralsVariable [of literal valuation2]
have var literal ∈ vars valuation2
with caseCons
show ?case
by simp
qed

lemma varsAppendValuation:
fixes valuation1 :: Valuation and valuation2 :: Valuation
shows vars (valuation1 @ valuation2) = vars valuation1 ∪ vars valuation2
by (induct valuation1) auto

lemma varsPrefixValuation:
fixes valuation1 :: Valuation and valuation2 :: Valuation
assumes isPrefix valuation1 valuation2
shows vars valuation1 ⊆ vars valuation2
proof
from assms
have set valuation1 ⊆ 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 ⇒ Valuation ⇒ 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 ⇒ Valuation ⇒ 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 el valuation
      by simp
    thus ?thesis
      using valuationContainsItsLiteralsVariable[of literal valuation]
      by simp
  next
    assume literalFalse literal valuation
    hence opposite literal el 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')
  end

40
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 literalFalse l 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 ≠ []
  shows ¬ clauseFalse clause []
  using assms
  by (induct clause) auto

lemma valuationContainsItsFalseClausesVariables:
  fixes clause::Clause and valuation::Valuation
  assumes clauseFalse clause valuation
  shows vars clause ⊆ vars valuation

proof
  fix v::Variable
  assume v ∈ vars clause
  hence ∃ l. var l = v ∧ 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 \ \in\ \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 \Rightarrow Valuation \Rightarrow bool
where
  formulaTrue [] valuation = True
| formulaTrue (clause # formula) valuation = (clauseTrue clause valuation \land formulaTrue formula valuation)

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

primrec
  formulaFalse :: Formula \Rightarrow Valuation \Rightarrow bool
where
  formulaFalse [] valuation = False
| formulaFalse (clause # formula) valuation = (clauseFalse clause valuation \lor formulaFalse formula valuation)

lemma formulaTrueIffAllClausesAreTrue:
  fixes formula :: Formula and valuation :: Valuation
  shows formulaTrue formula valuation = (\forall\ \text{clause} \ \text{el} \ \text{formula} \ \rightarrow\ \text{clauseTrue clause valuation})
  by (induct formula) auto

lemma formulaFalseIffContainsFalseClause:
  fixes formula :: Formula and valuation :: Valuation
  shows formulaFalse formula valuation = (\exists\ \text{clause} \ \text{el} \ \text{formula} \ \land\ \text{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 \( f1 :: \text{Formula} \) and \( f2 :: \text{Formula} \) and \( \text{valuation :: Valuation} \)
  shows \( \text{formulaTrue} \ (f1 @ f2) \ \text{valuation} = \text{formulaTrue} \ (f2 @ f1) \ \text{valuation} \)
  by (auto simp add: formulaTrueIffAllClausesAreTrue)

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

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

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

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

lemma formulaTrueAppendValuation:
\[\text{fixes } \text{formula} :: \text{Formula} \textbf{and } \text{valuation} :: \text{Valuation} \textbf{and } \text{valuation}' \]
\[\text{:: } \text{Valuation} \]
\[\text{assumes } \text{formulaTrue } \text{formula valuation} \]
\[\text{shows } \text{formulaTrue } \text{formula} \ (\text{valuation @ valuation}') \]
\[\text{using assms} \]
\[\text{by (induct formula) } (\text{auto simp add: clauseTrueAppendValuation}) \]

\begin{description}
\item[lemma \text{formulaFalseAppendValuation}:]
\[\text{fixes } \text{formula} :: \text{Formula} \textbf{and } \text{valuation} :: \text{Valuation} \textbf{and } \text{valuation}' \]
\[\text{:: } \text{Valuation} \]
\[\text{assumes } \text{formulaFalse } \text{formula valuation} \]
\[\text{shows } \text{formulaFalse } \text{formula} \ (\text{valuation @ valuation}') \]
\[\text{using assms} \]
\[\text{by (induct formula) } (\text{auto simp add: clauseFalseAppendValuation}) \]
\end{description}

\begin{description}
\item[lemma \text{trueFormulaWithSingleLiteralClause}:]
\[\text{fixes } \text{formula} :: \text{Formula} \textbf{and } \text{literal} :: \text{Literal} \textbf{and } \text{valuation} :: \text{Valuation} \]
\[\text{assumes } \text{formulaTrue } \text{(removeAll [literal] formula)} \ (\text{valuation @ [literal]}) \]
\[\text{shows } \text{formulaTrue } \text{formula} \ (\text{valuation @ [literal]}) \]
\[\text{proof} - \]
\[\{ \]
\[\text{fix } \text{clause} :: \text{Clause} \]
\[\text{assume } \text{clause} \text{ el formula} \]
\[\text{with assms} \]
\[\text{have } \text{clauseTrue } \text{clause} \ (\text{valuation @ [literal]}) \]
\[\text{proof } (\text{cases clause = [literal]}) \]
\[\text{case True} \]
\[\text{thus } \text{thesis} \]
\[\text{by simp} \]
\[\text{next} \]
\[\text{case False} \]
\[\text{with } \text{clause el formula} \]
\[\text{have } \text{clause el (removeAll [literal] formula)} \]
\[\text{by simp} \]
\[\text{with } \text{(formulaTrue (removeAll [literal] formula)} \ (\text{valuation @ [literal]}); \]
\[\text{show } \text{thesis} \]
\[\text{by } (\text{simp add: formulaTrueIffAllClausesAreTrue}) \]
\[\text{qed} \]
\[\} \]
\[\text{thus } \text{thesis} \]
\[\text{by } (\text{simp add: formulaTrueIffAllClausesAreTrue}) \]
\[\text{qed} \]
\end{description}
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 val2form 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 \lor inconsistent valuation2 \lor (∃ 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
qed
thus \( ?\thesis \)
  by simp
qed

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

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

lemma inconsistentPrefix:
  assumes isPrefix \( valuation1 \) \( valuation2 \) and inconsistent \( valuation1 \)
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 var literal ∈ vars formula 
  by (rule formulaContainsItsLiteralsVariable) 
with totalFormula
have var literal ∈ vars valuation 
  by auto
thus thesis using variableDefinedImplesLiteralDefined [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 ∨ 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 ∨ 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: ¬
    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 \in vars valuation and totalRemoveAll: 
    total valuation (vars (removeAll [literal] formula))
  shows total valuation (vars formula)
proof –
  have vars formula – vars [literal] \subseteq 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 \Rightarrow 'a \Rightarrow bool
defs (overloaded)
  modelFormula-def [simp]: model valuation (formula::Formula) == consistent valuation \land (formulaTrue formula valuation)
  modelClause-def [simp]: model valuation (clause::Clause) == consistent valuation \land (clauseTrue clause valuation)

Checks if a formula has a model
definition satisfiable :: Formula \Rightarrow bool
where
  satisfiable formula == \exists valuation. model valuation formula

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

lemma satisfiableSubset:
  fixes formula0 :: Formula and formula :: Formula
  assumes subset: \forall (clause::Clause). clause el formula0 \longrightarrow clause el formula
  shows satisfiable formula \longrightarrow satisfiable formula0
proof
  assume satisfiable formula
  show satisfiable formula0
    proof –
      from (satisfiable formula) obtain valuation :: Valuation
  qed
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
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 \in 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 \in vars valuation)
  have consistent (valuation @ [literal])
  proof (cases inconsistent (valuation @ [literal]))
    case True
    hence inconsistent valuation \or inconsistent [literal] \or (\exists l. literalTrue l valuation \and literalFalse l [literal])
      by (rule inconsistentAppend)
  qed
with ⟨consistent valuation⟩
have ∃ l. literalTrue l valuation ∧ literalFalse l [literal]
by auto
hence literalFalse literal valuation
by auto
hence var (opposite literal) ∈ (vars valuation)
  using valuationContainsItsLiteralsVariable [of opposite literal valuation]
  by simp
with ⟨var literal / vars valuation⟩
have False
by simp
thus ?thesis ..
qed simp
ultimately
show ?thesis
by auto
qed

2.2.9 Tautological clauses

lemma tautologyNotFalse:
  fixes clause :: Clause and valuation :: Valuation
  assumes clauseTautology clause consistent valuation
  shows ¬ clauseFalse clause valuation
  using assms
    clauseTautologyCharacterization[of clause]
    clauseFalseIffAllLiteralsAreFalse[of clause valuation]
    inconsistentCharacterization
  by auto

lemma tautologyInTotalValuation:
  assumes
    clauseTautology clause
    vars clause ⊆ vars valuation
  shows clauseTrue clause valuation
  proof −
    from ⟨clauseTautology clause⟩
    obtain literal
      where literal el clause opposite literal el clause
      by (auto simp add: clauseTautologyCharacterization)
    hence var literal ∈ vars clause
      using clauseContainsItsLiteralsVariable[of literal clause]
      using clauseContainsItsLiteralsVariable[of opposite literal clause]
      by simp
    hence var literal ∈ vars valuation
      using ⟨vars clause ⊆ vars valuation⟩
by auto

hence \((\text{literalTrue} \; \text{literal valuation} \lor \text{literalFalse} \; \text{literal valuation})\)

using `\text{varInClauseVars}[\text{of var literal valuation}]`

using `\text{varInClauseVars}[\text{of var (opposite literal) valuation}]`

using `\text{literalsWithSameVariableAreEqualOrOpposite}`

by auto

thus \(\text{thesis}\)

using \((\text{literal el clause}) \; (\text{opposite literal el clause})\)

by `(auto simp add: \text{clauseTrueIffContainsTrueLiteral})`

qed

lemma `\text{modelAppendTautology}`:

assumes

```text
\text{model valuation F clauseTautology c} \\
\text{vars valuation \supseteq vars F \cup vars c}
```

shows

```text
\text{model valuation (F @ [c])}
```

using `assms`

using `\text{tautologyInTotalValuation}[\text{of c valuation}]`

by `(auto simp add: \text{formulaTrueAppend})`

lemma `\text{satisfiableAppendTautology}`:

assumes

```text
\text{satisfiable F clauseTautology c}
```

shows

```text
\text{satisfiable (F @ [c])}
```

proof

```text
from \((\text{clauseTautology c})\)
obtain \(l\)

where \(l \; \text{el c} \; \text{opposite l el c}\)

by `(auto simp add: \text{clauseTautologyCharacterization})`

from \((\text{satisfiable F})\)

obtain valuation

where consistent valuation \text{formulaTrue} F valuation

unfolding `\text{satisfiable-def}`

by auto

show \(\text{thesis}\)

proof (cases var \(l \in \text{vars valuation}\))

```text
\text{case True}
```

hence \(\text{literalTrue} \; l \; \text{valuation} \lor \text{literalFalse} \; l \; \text{valuation}\)

using `\text{varInClauseVars}[\text{of var l valuation}]`

by `(auto simp add: \text{literalsWithSameVariableAreEqualOrOpposite})`

hence \(\text{clauseTrue} \; c \; \text{valuation}\)

using \((d \; \text{el c}) \; (\text{opposite d el c})\)

by `(auto simp add: \text{clauseTrueIffContainsTrueLiteral})`

thus \(\text{thesis}\)

using \((\text{consistent valuation}) \; (\text{formulaTrue F valuation})\)

unfolding `\text{satisfiable-def}`

by `(auto simp add: \text{formulaTrueIffAllClausesAreTrue})`

56
next
  case False
  let ?valuation' = valuation @ [l]
  have model ?valuation' F
    using (var l \notin vars valuation)
    using (formulaTrue F valuation) (consistent valuation)
    using modelExpand[of valuation F l]
    by simp
  moreover
  have formulaTrue [c] ?valuation'
    using (l \in 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

57
lemma satisfiable AppendTautologicalFormula:
assumes
  satisfiable F \forall c. c \in F' \rightarrow clauseTautology c
shows
  satisfiable (F \oplus F')
using assms
proof (induct F')
case Nil
  thus ?case
    by simp
next
case (Cons c F'')
  hence satisfiable (F \oplus F'')
    by simp
  thus ?case
    using Cons(3)
    using satisfiableAppendTautology[of F \oplus F'' c]
    unfolding satisfiable-def
    by (simp add: formulaTrueIffAllClausesAreTrue)
qed

lemma satisfiableFilterTautologies:
shows satisfiable F = satisfiable (filter (% c. \neg clauseTautology c) F)
proof (induct F)
case Nil
  thus ?case
    by simp
next
case (Cons c F')
  let ?filt = \lambda F. filter (% c. \neg clauseTautology c) F
  let ?filt' = \lambda F. filter (% c. clauseTautology c) F
  show ?case
    proof
      assume satisfiable (?filt (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 ?thesis
  using satisfiableAppendTautology[of F' c']
  using (clauseTautology c')
  unfolding satisfiable-def
  by (auto simp add: formulaTrueIffAllClausesAreTrue)
next
case False
  hence ?filter (c' ≠ F') = c' ≠ ?filter F'
    by auto
  hence satisfiable (c' ≠ ?filter F')
    using (satisfiable (?filter (c' ≠ F'))):
    by simp
moreover
  have ∀ c. c ∈ ?filter F' → clauseTautology c
    by simp
ultimately
  have satisfiable ((c' ≠ ?filter F') @ ?filter F')
    using satisfiableAppendTautologicalFormula[of c' ≠ ?filter F' ?filter F']
    by (simp (no-asms-use))
thus ?thesis
  unfolding satisfiable-def
  by (auto simp add: formulaTrueIffAllClausesAreTrue)
qed
qed

lemma modelFilterTautologies:
assumes
  model valuation (filter (% c. ¬ clauseTautology c) F)
  vars F ⊆ vars valuation
shows model valuation F
using assms
proof (induct F)
case Nil
  thus ?case
    by simp
next
case (Cons c' F')
  let ?filter = λ F. filter (% c. ¬ clauseTautology c) F
  let ?filter' = λ F. filter (% c. clauseTautology c) F
  show ?case
    proof (cases clauseTautology c')
      case True
        thus ?thesis
          using Cons
          using tautologyInTheTotalValuation[of c' valuation]
        by auto
next
  case False
  hence \( \neq {\text{filt}} (c' \# F') = c' \# \neq {\text{filt}} F' \)
    by auto
  hence model valuation \( (c' \# \neq {\text{filt}} F') \)
    using (model valuation \( (\neq {\text{filt}} (c' \# F')) \))
    by simp
  moreover
  have \( \forall c. c el \neq {\text{filt}}' F' \rightarrow \text{clauseTautology } c \)
    by simp
  moreover
  have \( \forall c. \text{vars } ((c' \# \neq {\text{filt}} F') \# \neq {\text{filt}}' F') \subseteq \text{vars valuation} \)
    using \( \text{varsSubsetFormula } (\neq {\text{filt}} F' F') \)
    using \( \text{varsSubsetFormula } (\neq {\text{filt}}' F' F') \)
    using \( \text{varsAppendFormulae } (c' \# \neq {\text{filt}} F' \neq {\text{filt}}' F') \)
    using Cons(3)
    using \( \text{formulaContainsItsClausesVariables } (\neq {\text{filt}} F') \)
    by auto
  ultimately
  have model valuation \( ((c' \# \neq {\text{filt}} F') \# \neq {\text{filt}}' F') \)
    using \( \text{modelAppendTautologicalFormula } (\neq {\text{filt}} F') \)
    \( \neq {\text{filt}}' F' \)
    using \( \text{varsAppendFormulae } (c' \# \neq {\text{filt}} F' \neq {\text{filt}}' F') \)
    by (simp (no_asm-use)) (blast)
  thus \( \text{thesis} \)
    using \( \text{formulaTrueAppend } (\neq {\text{filt}} F' \neq {\text{filt}}' F' \text{ valuation}) \)
    using \( \text{formulaTrueIffAllClausesAreTrue } (\neq {\text{filt}} F' \text{ valuation}) \)
    using \( \text{formulaTrueIffAllClausesAreTrue } (\neq {\text{filt}}' F' \text{ valuation}) \)
    by auto
  qed
qed

2.2.10 Entailment

Formula entails literal if it is true in all its models

definition \text{formulaEntailsLiteral } :: \text{Formula } \Rightarrow \text{Literal } \Rightarrow \text{bool} where
\text{formulaEntailsLiteral } \text{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 \text{clauseEntailsLiteral } :: \text{Clause } \Rightarrow \text{Literal } \Rightarrow \text{bool} where
\text{clauseEntailsLiteral } \text{clause literal } ==
\( \forall (\text{valuation}::\text{Valuation}). \text{model valuation clause } \rightarrow \text{literalTrue literal valuation} \)
Formula entails clause if it is true in all its models

**definition** `formulaEntailsClause` :: `Formula ⇒ Clause ⇒ bool`

**where**

`formulaEntailsClause formula clause` ==
`∀ (valuation::Valuation). model valuation formula → model valuation clause`

Formula entails valuation if it entails its every literal

**definition** `formulaEntailsValuation` :: `Formula ⇒ Valuation ⇒ bool`

**where**

`formulaEntailsValuation formula valuation` ==
`∀ literal. literal el valuation → formulaEntailsLiteral formula literal`

Formula entails formula if it is true in all its models

**definition** `formulaEntailsFormula` :: `Formula ⇒ Formula ⇒ bool`

**where**

`formulaEntailsFormula-def`: `formulaEntailsFormula formula formula'` ==
`∀ (valuation::Valuation). model valuation formula → model valuation formula'`

**lemma** `singleLiteralClausesEntailItsLiteral`:

**fixes** `clause` :: `Clause` and `literal` :: `Literal`

**assumes** `length clause = 1` and `literal el clause`

**shows** `clauseEntailsLiteral clause literal`

**proof**

from `assms` have `onlyLiteral`: `∀ l. l el clause → l = literal`
using `lengthOneImpliesOnlyElement[of clause literal]` by `simp`

{ fix `valuation` :: `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` :: `Clause` and `formula` :: `Formula` and `literal` :: `Literal`

**assumes** `clause el formula` and `clauseEntailsLiteral clause literal`

**shows** `formulaEntailsLiteral formula literal`

**proof**

{ fix `valuation` :: `Valuation`

assume \( \text{modelFormula} : \text{model valuation formula} \)

with \( \langle \text{clause el formula} \rangle \)
have \( \text{clauseTrue clause valuation} \)
  by (simp add: \( \text{formulaTrueIffAllClausesAreTrue} \))
with \( \text{modelFormula} \) \( \langle \text{clauseEntailsLiteral clause literal} \rangle \)
have \( \text{literalTrue literal valuation} \)
  by (auto simp add: \( \text{clauseEntailsLiteral-def} \))
}\)
thus \( ?\text{thesis} \)
  by (simp add: \( \text{formulaEntailsLiteral-def} \))
qed

lemma \( \text{formulaEntailsLiteralAppend} \):
  fixes \( \text{formula} :: \text{Formula and formula'} :: \text{Formula and literal :: Literal} \)
  assumes \( \text{formulaEntailsLiteral formula literal} \)
  shows \( \text{formulaEntailsLiteral (formula @ formula')} \) literal
proof –
{ 
  fix valuation :: Valuation
  assume \( \text{modelFF'} : \text{model valuation (formula @ formula')} \)
  
  hence \( \text{formulaTrue formula valuation} \)
    by (simp add: \( \text{formulaTrueAppend} \))
  with \( \text{modelFF'} \) \( \langle \text{formulaEntailsLiteral formula literal} \rangle \)
  have \( \text{literalTrue literal valuation} \)
    by (simp add: \( \text{formulaEntailsLiteral-def} \))
}\)
thus \( ?\text{thesis} \)
  by (simp add: \( \text{formulaEntailsLiteral-def} \))
qed

lemma \( \text{formulaEntailsLiteralSubset} \):
  fixes \( \text{formula} :: \text{Formula and formula'} :: \text{Formula and literal :: Literal} \)
  assumes \( \text{formulaEntailsLiteral formula literal} \) \( \forall (c :: \text{Clause}). \) \( c \text{ el formula} \rightarrow c \text{ el formula'} \)
  shows \( \text{formulaEntailsLiteral formula'} \) literal
proof –
{ 
  fix valuation :: Valuation
  assume \( \text{modelF'} : \text{model valuation formula'} \)
  with \( \forall (c :: \text{Clause}). \) \( c \text{ el formula} \rightarrow c \text{ el formula'} \)
  have \( \text{formulaTrue formula valuation} \)
    by (auto simp add: \( \text{formulaTrueIffAllClausesAreTrue} \))
  with \( \text{modelF'} \) \( \langle \text{formulaEntailsLiteral formula literal} \rangle \)
  have \( \text{literalTrue literal valuation} \)
    by (simp add: \( \text{formulaEntailsLiteral-def} \))
}\)

62
thus ?thesis
by (simp add:formulaEntailsLiteral-def)
qed

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

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

lemma formulaEntailsItsClauses:
  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' = (\forall clause :: Clause.
    clause el formula' \longrightarrow formulaEntailsClause formula clause)
  (is \?lhs = \?rhs)
proof
  assume \?lhs
  show \?rhs
  proof
    fix clause :: Clause
    show clause el formula' \longrightarrow 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 \?rhs
        have clauseTrue clause valuation
          by (simp add: formulaTrueIffAllClausesAreTrue)
      }
      thus \?thesis
      by (simp add: formulaEntailsClause-def)
    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
    }
  }

65
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 \( \text{formula} :: \) Formula and \( \text{valuation} :: \) Valuation
assumes \( \text{formula} \text{False} \) formula valuation and \( \text{formula} \text{Entails} \text{Valuation} \) formula valuation
shows \( \neg \) satisfiable formula

proof —
from \( \text{formula} \text{False} \) 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 \( \text{clause} \) el formula obtain literal :: Literal
  where literal el clause and literalTrue literal valuation’
  by (auto simp add:formulaTrueIffAllClausesAreTrue clauseTrueIf-
  fContainsTrueLiteral)
  with \( \text{clauseFalse} \) clause valuation
  have literalFalse literal valuation
  by (auto simp add:clauseFalseIffAllLiteralsAreFalse)
  with \( \text{formula} \text{Entails} \text{Valuation} \) 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 \( \text{literalTrue} \) literal valuation’ \( \text{literalFalse} \) literal valuation’
  modelV’
  have False
    by (simp add:inconsistentCharacterization)
  }
  thus ?thesis
  by (auto simp add:satisfiable-def)
qed

lemma formulaFalseInEntailedOrPureValuationIsUnsatisfiable:
fixes \( \text{formula} :: \) Formula and \( \text{valuation} :: \) Valuation
assumes \( \text{formula} \text{False} \) formula valuation and
\( \forall \text{literal’}. \text{literal'} \) el valuation \( \longrightarrow \) formulaEntailsLiteral formula lit-
eral’ \( \lor \) \( \neg \) opposite literal’ el formula
shows \( \neg \) satisfiable formula

proof —
from \( \text{formula} \text{False} \) 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 \( \text{clause} \) el formula obtain literal :: Literal
  where literal el clause and literalTrue literal valuation’

67
by (auto simp add: formulaTrueIffAllClausesAreTrue clauseTrueIfContainsTrueLiteral)
with ⟨clauseFalse clause valuation⟩
have literalFalse literal valuation
  by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
with ⟨\forall literal'. literal' el valuation \rightarrow formulaEntailsLiteral formula literal' \lor \neg opposite literal' el formula⟩
have formulaEntailsLiteral formula (opposite literal) \lor \neg literal el formula
  by auto
moreover
  { assume formulaEntailsLiteral formula (opposite literal) 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)
  }
moreover
  { assume \neg literal el formula
    with ⟨clause el formula⟩ ⟨literal el clause⟩
    have False
      by (simp add: literalElFormulaCharacterization)
  }
ultimately
have False
  by auto
} thus \textit{thesis}
  by (auto simp add: satisfiable-def)
qed

lemma unsatisfiableFormulaWithSingleLiteralClause:
  fixes formula :: Formula \and literal :: Literal
  assumes \neg satisfiable 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 \\in \ vars valuation)
      case True
\{
  \textbf{assume} \textit{literalsTrue literal valuation}
  \with model valuation (removeAll \{\textit{literals} \} \textit{formula}).
  \textbf{have} model valuation \textit{formula}
  \by (auto simp add: formulaTrueIffAllClausesAreTrue)
  \with \textit{\neg satisfiable formula}
  \textbf{have} False
  \by (auto simp add:satisfiable-def)
\}
\with \textit{True}
\textbf{show} \textit{?thesis}
\using variableDefinedImpliesLiteralDefined [of \textit{literals valuation}]
\by auto

\textbf{next}
\textbf{case} False
\with model valuation (removeAll \{\textit{literals} \} \textit{formula})
\textbf{have} \textit{model} (valuation @ \{\textit{literals} \}) (removeAll \{\textit{literals} \} \textit{formula})
\by (rule modelExpand)
\textbf{hence}
\textit{formulaTrue} (removeAll \{\textit{literals} \} \textit{formula}) (valuation @ \{\textit{literals} \})
\and \textit{consistent} (valuation @ \{\textit{literals} \})
\by auto
\from \textit{formulaTrue} (removeAll \{\textit{literals} \} \textit{formula}) (valuation @ \{\textit{literals} \})
\textbf{have} \textit{formulaTrue} \textit{formula} (valuation @ \{\textit{literals} \})
\by (rule trueFormulaWithSingleLiteralClause)
\with \textit{consistent} (valuation @ \{\textit{literals} \})
\textbf{have} \textit{model} (valuation @ \{\textit{literals} \} \textit{formula})
\by simp
\with \textit{\neg satisfiable formula}
\textbf{have} False
\by (auto simp add:satisfiable-def)
\textbf{thus} \textit{?thesis} ..
\textbf{qed}
\}
\textbf{thus} \textit{?thesis}
\by (simp add:formulaEntailsLiteral-def)
\textbf{qed}

\textbf{lemma} unsatisfiableFormulaWithSingleLiteralClauses:
\textbf{fixes} \textit{F} :: Formula \and \textit{c} :: Clause
\textbf{assumes} \textit{\neg satisfiable} \ (\textit{F} @ val2form \ (\textit{oppositeLiteralList} \textit{c})) \textit{\neg clauseTautology c}
\textbf{shows} \textit{formulaEntailsClause F c}

\textbf{proof}–
\{
\textbf{fix} \textit{v} :: Valuation
\textbf{assume} \textit{model} \textit{v F}
\textbf{with} \textit{\neg satisfiable} \ (\textit{F} @ val2form \ (\textit{oppositeLiteralList} \textit{c}))
\}
have \( \neg \text{formulaTrue} (\text{val2form} (\text{oppositeLiteralList} c)) \) \( v \)

unfolding satisfiable-def
by (auto simp add: formulaTrueAppend)

have clauseTrue c v
proof (cases \( \exists \ l. \ l \in c \land (\text{literalTrue} l v) \))
  case True
  thus ?thesis
  using clauseTrueIffContainsTrueLiteral
  by simp
next
  case False
  let \( ?v' = v @ (\text{oppositeLiteralList} c) \)

have \( \neg \text{inconsistent} (\text{oppositeLiteralList} c) \)
proof -
{  
  assume \( \neg ?\text{thesis} \)
  then obtain \( l :\text{Literal} \)
  where \( l \in (\text{oppositeLiteralList} c) \) \( \text{opposite} l \in (\text{oppositeLiteralList} c) \)
  using inconsistentCharacterization \[ of \text{oppositeLiteralList} c \]
  by auto
  hence \( (\text{opposite} l) \in c \) \( \text{opposite} l \in c \)
  using literalElListIffOppositeLiteralElOppositeLiteralList \[ of \text{opposite} l \in c \]
  by auto
  with \( \neg \text{clauseTautology} c \)
  have False
  by simp
}
thus ?thesis
by auto
qed

with \( \text{False} (\text{model} v F) \)

have consistent \( ?v' \)
using inconsistentAppend \[ of v \text{oppositeLiteralList} c \]
unfolding consistent-def
using literalElListIffOppositeLiteralElOppositeLiteralList
by auto

moreover
from \( \text{model} v F \)

have formulaTrue F \( ?v' \)
using formulaTrueAppendValuation
by simp

70
moreover
have formulaTrue (val2form (oppositeLiteralList c)) ?v'
  using val2formFormulaTrue[of oppositeLiteralList c v @ oppositeLiteralList c]
  by simp
ultimately
have model ?v' (F @ val2form (oppositeLiteralList c))
  by (simp add: formulaTrueAppend)
with (~ satisfiable (F @ val2form (oppositeLiteralList c)));
have False
  unfolding satisfiable-def
  by auto
thus ~thesis
  by simp
qed
}
thus ~thesis
  unfolding formulaEntailsClause-def
  by simp
qed

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

lemma val2formIsEntailed:
  shows formulaEntailsValuation (F' @ val2form valuation @ F'') valuation
proof--
  { fix l::Literal
    assume l el valuation
    hence [l] el val2form valuation
      by (induct valuation) (auto)
2.2.11 Equivalency

Formulas are equivalent if they have same models.

definition equivalentFormulae :: Formula ⇒ Formula ⇒ bool
where
equivalentFormulae formula1 formula2 == ∀ (valuation::Valuation). model valuation formula1 = model valuation formula2

lemma equivalentFormulaeIffEntailEachOther:
fixes formula1 :: Formula and formula2 :: Formula
shows equivalentFormulae formula1 formula2 = (formulaEntailsFormula formula1 formula2 ∧ formulaEntailsFormula formula2 formula1)
by (auto simp add: formulaEntailsFormula-def equivalentFormulae-def)

lemma equivalentFormulaeReflexivity:
fixes formula :: Formula
shows equivalentFormulae formula formula
unfolding equivalentFormulae-def
by auto

lemma equivalentFormulaeSymmetry:
fixes formula1 :: Formula and formula2 :: Formula
shows equivalentFormulae formula1 formula2 = equivalentFormulae formula2 formula1
unfolding equivalentFormulae-def
by auto

lemma equivalentFormulaeTransitivity:

fixes \( \text{formula1} :: \text{Formula} \) and \( \text{formula2} :: \text{Formula} \) and \( \text{formula3} :: \text{Formula} \)

assumes equivalentFormulae \( \text{formula1} \) \( \text{formula2} \) and equivalentFormulae \( \text{formula2} \) \( \text{formula3} \)

shows equivalentFormulae \( \text{formula1} \) \( \text{formula3} \)

using assms

unfolding equivalentFormulae-def

by auto

lemma equivalentFormulaeAppend:
  fixes \( \text{formula1} :: \text{Formula} \) and \( \text{formula1}' :: \text{Formula} \) and \( \text{formula2} :: \text{Formula} \)
  assumes equivalentFormulae \( \text{formula1} \) \( \text{formula1}' \)
  shows equivalentFormulae \( \text{formula1}@\text{formula2} \) \( \text{formula1}'@\text{formula2} \)
  using assms
  unfolding equivalentFormulae-def

lemma satisfiableEquivalent:
  fixes \( \text{formula1} :: \text{Formula} \) and \( \text{formula2} :: \text{Formula} \)
  assumes equivalentFormulae \( \text{formula1} \) \( \text{formula2} \)
  shows satisfiable \( \text{formula1} = \text{satisfiable} \text{formula2} \)
  using assms
  unfolding equivalentFormulae-def
  unfolding satisfiable-def

by auto

lemma satisfiableEquivalentAppend:
  fixes \( \text{formula1} :: \text{Formula} \) and \( \text{formula1}' :: \text{Formula} \) and \( \text{formula2} :: \text{Formula} \)
  assumes equivalentFormulae \( \text{formula1} \) \( \text{formula1}' \) and satisfiable \( \text{formula1}@\text{formula2} \)
  shows satisfiable \( \text{formula1}'@\text{formula2} \)
  using assms

proof —
  from \langle satisfiable \( \text{formula1}@\text{formula2} \) \rangle obtain valuation::Valuation where consistent valuation formulaTrue formula1 valuation for-
  mulaTrue formula2 valuation
  unfolding satisfiable-def
  by (auto simp add: formulaTrueAppend)

  from \langle equivalentFormulae \( \text{formula1} \) \( \text{formula1}' \) \( \text{consistent valuation} \langle formulaTrue formula1 valuation \rangle \langle formulaTrue formula1 valuation \rangle \langle consistent valuation \langle formulaTrue formula1 valuation \rangle \langle formulaTrue formula1 valuation \rangle \langle formulaTrue formula2 valuation \rangle \rangle

  unfolding equivalentFormulae-def
  by auto

  show \langle \text{thesis} \rangle
  using \langle consistent valuation \rangle \langle formulaTrue formula1 valuation \rangle \langle formulaTrue formula2 valuation \rangle

by auto
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 (simp add: formulaTrueAppend)
  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 (simp add: formulaTrueAppend)
  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
  qed

lemma entailsLiteralRelplacePartWithEquivalent:
  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 \( F :: \text{Formula} \) and \( F' :: \text{Formula} \) and \( v :: \text{Valuation} \)
assumes equivalentFormulae \((F1 @ \text{val2form } v @ F2) \ F'\)
shows equivalentFormulae \((F1 @ \text{val2form } v @ [\text{removeFalseLiterals} c v] @ F2) \ (F' @ [\cdot])\)
(is equivalentFormulae ?lhs ?rhs)

unfolding equivalentFormulae-def

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

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

  from \( \langle \text{consistent } v' \rangle \langle \text{formulaTrue } F' \ v' \rangle \langle \text{equivalentFormulae } (F1 @ \text{val2form } v @ F2) \ F' \rangle \)
  have model \( v' \ (F1 @ \text{val2form } v @ F2) \)
  unfolding equivalentFormulae-def
  by auto
moreover
have clauseTrue \((\text{removeFalseLiterals } c v) \ v'\)
proof
  from \( \langle \text{clauseTrue } c v' \rangle \)
  obtain \( l :: \text{Literal} \)

77
where \( l \in c \) and \( \text{literalTrue } l v' \)

by (auto simp add: clauseTrueIffContainsTrueLiteral)

have \(~ \text{literalFalse } l v\)

proof –

{ 
  assume \(~ \text{thesis}\)
  hence opposite \( l \in v \)
    by simp
  with \( \text{model } v' (F1 \circ val2form v \circ F2); \)
  have opposite \( l \in v' \)
    using \( \text{val2formFormulaTrue[of } v' \)\]
    by auto (simp add: formulaTrueAppend)
  with \( \text{literalTrue } l v' \) \( \text{consistent } v' \)
  have False
    by (simp add: inconsistentCharacterization)
}

thus \text{thesis} 
by auto

qed

with \( l \in c \)

have \( l \in \text{removeFalseLiterals } c v \)

unfolding \text{removeFalseLiterals-def}\n
by simp

with \( \text{literalTrue } l v' \)

show \text{thesis} 
by (auto simp add: clauseTrueIffContainsTrueLiteral)

qed

ultimately

show \( \text{model } v' ?lhs \)
by (simp add: formulaTrueAppend)

qed

qed

lemma \text{falseAndDuplicateLiteralsCanBeRemoved}:

assumes \text{equivalentFormulae } (F1 \circ val2form v \circ F2) F'

shows \text{equivalentFormulae } (F1 \circ val2form v \circ [\text{remdups (removeFalseLiterals } c v)] \circ F2) (F' \circ [c])

(is \text{equivalentFormulae } ?lhs ?rhs)

proof –

from \text{equivalentFormulae } (F1 \circ val2form v \circ F2) F'

have \text{equivalentFormulae } (F1 \circ val2form v \circ [\text{removeFalseLiterals } c v] \circ F2) (F' \circ [c])

using \text{falseLiteralsCanBeRemoved}

by simp

have \text{equivalentFormulae } [\text{remdups (removeFalseLiterals } c v)] [\text{removeFalseLiterals } c v]

using \text{removeDuplicateLiteralsEquivalentClause}

by simp

78
hence equivalentFormulae (F1 @ val2form v @ [remdup (removeFalseLiterals c v)]) @ F2)
  (F1 @ val2form v @ [removeFalseLiterals c v] @ F2)
using replaceEquivalentByEquivalent
[of [remdup (removeFalseLiterals c v)] [removeFalseLiterals c v]
F1 @ val2form v F2]
by auto
thus ?thesis
using (equivalentFormulae (F1 @ val2form v @ [removeFalseLiterals c v] @ F2)
(F' @ [c]))
using equivalentFormulaeTransitivity[of
(F1 @ val2form v @ [remdup (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)
  hence consistent v' and formulaTrue (F @ val2form v) v'
  by auto
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')
  have literalTrue l v'
    using val2formFormulaTrue[of v v']
    using formulaTrueAppend[of F val2form v]
    by simp
thus thesis
  using \( \{ \ell \in c \} \)
  by (auto simp add: clauseTrueIffContainsTrueLiteral)
qed
ultimately
show model v' \((F' @ [c])\)
  by (simp add: formulaTrueAppend)
next
assume model v' \((F' @ [c])\)
thus model v' \((F @ \text{val2form } v)\)
  using (equivalentFormulae \((F @ \text{val2form } v)\ \ F'\))
  unfolding equivalentFormulae-def
  using formulaTrueAppend[of F' [c] v']
  by auto
qed
qed

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

80
by simp
hence \text{formulaEntailsLiteral } F \text{ (opposite } l)\)
using \text{formulaEntailsValuation } F \text{ valuation}
unfolding \text{formulaEntailsValuation-def}
by simp
hence \text{literalFalse } l \text{ valuation'}
using \text{consistent valuation'}
using \text{formulaTrue } F \text{ valuation'}
unfolding \text{formulaEntailsLiteral-def}
by simp
with \text{literalTrue } l \text{ valuation'}
\text{consistent valuation'}
have False
by (simp add: inconsistentCharacterization)
}\)
thus \text{thesis}
by auto
qed ultimately
have \text{model valuation'} (\text{list-diff clause (oppositeLiteralList valuation)})
using \text{consistent valuation'}
using listDiffIff[of l clause oppositeLiteralList valuation]
by (auto simp add: clauseTrueIffContainsTrueLiteral)
}\)
thus \text{thesis}
unfolding \text{formulaEntailsClause-def}
by simp
qed

2.2.13 Resolution
definition resolve \text{clause1} \text{clause2} \text{literal} == \text{removeAll literal clause1 @ removeAll (opposite literal) clause2}

lemma resolventIsEntailed:
fixes \text{clause1} :: \text{Clause and clause2 :: Clause and literal :: Literal}
shows \text{formulaEntailsClause [clause1, clause2] (resolve clause1 clause2 literal)}
proof -
{ 
  fix valuation :: Valuation
  assume model valuation [clause1, clause2]
  from (model valuation [clause1, clause2]) obtain l1 :: Literal
  where \text{lit } l1 @ \text{clause1 and literalTrue } l1 \text{ valuation}
  by (auto simp add: formulaTrueIffAllClausesAreTrue clauseTrueIf-
  FContainsTrueLiteral)
  from (model valuation [clause1, clause2]) obtain l2 :: Literal
  where \text{lit } l2 @ \text{clause2 and literalTrue } l2 \text{ valuation}
  by (auto simp add: formulaTrueIffAllClausesAreTrue clauseTrueIf-

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 l1 ≠ 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
by (simp add: formulaEntailsClause-def)
from ⟨clauseTrue clause1 valuation⟩ ⟨clauseTrue clause2 valuation⟩
⟨consistent valuation⟩
have clauseTrue (resolve clause1 clause2 literal) valuation
  using resolventIsEntailed
  by (auto simp add: formulaEntailsClause-def)
with ⟨consistent valuation⟩
have model valuation (resolve clause1 clause2 literal)
  by simp
}
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) ∨ 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)
2.2.14 Unit clauses

Clause is unit in a valuation if all its literals but one are false, and that one is undefined.

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 formulaEn-tailsClause 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
qed
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 ⟨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 (valuation @ [l])
  using assms
  unfolding isUnitClause-def
  by auto

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

using assms
unfolding isUnitClause-def
by (auto simp add: inconsistentCharacterization)

lemma unitBecomesFalse:
assumes
isUnitClause uClause uLiteral valuation
shows
clauseFalse uClause (valuation @ [opposite uLiteral])
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
isReason :: Clause \Rightarrow Literal \Rightarrow Valuation \Rightarrow bool
where
(isReason clause literal valuation) ==
(literal el clause) \land
(clauseFalse (removeAll literal clause) valuation) \land
(\forall literal'. literal' el (removeAll literal clause)
\rightarrow precedes (opposite literal') literal valuation \land opposite literal' \neq literal)

lemma isReasonAppend:
fixeds 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 literal'. literal' el (removeAll literal clause) \rightarrow precedes (opposite literal') literal valuation \land opposite literal' \neq literal (is ?precedes valuation))
unfolding isReason-def
by auto
moreover
from (?false valuation)
have ?false (valuation @ valuation')
by (rule clauseFalseAppendValuation)
moreover
from (?precedes valuation)
have ?precedes (valuation @ valuation')
  by (simp add: precedesAppend)
ultimately
show ?thesis
  unfolding isReason-def
by auto
qed

lemma isUnitClauseIsReason:
  fixes uClause :: Clause and uLiteral :: Literal and valuation :: Valuation
  assumes isUnitClause uClause uLiteral valuation uLiteral el valuation'
  shows isReason uClause uLiteral (valuation @ valuation')
proof -
from assms
  have uLiteral el uClause and ~ literalTrue uLiteral valuation and ~ literalFalse uLiteral valuation
  and A literal. literal el uClause A literal # uLiteral --> literalFalse
  literal valuation
    unfolding isUnitClause-def
    by auto
  hence clauseFalse (removeAll uLiteral uClause) valuation
    by (simp add: clauseFalseIffAllLiteralsAreFalse)
  hence clauseFalse (removeAll uLiteral uClause) (valuation @ valuation')
    by (simp add: clauseFalseAppendValuation)
moreover
have A literal'. literal' el (removeAll uLiteral uClause) -->
  precedes (opposite literal') uLiteral (valuation @ valuation') A
  (opposite literal') # uLiteral
proof -
  { fix literal' :: Literal
    assume literal' el (removeAll uLiteral uClause)
    with (ClauseFalse (removeAll uLiteral uClause) valuation)
    have literalFalse literal' valuation
      by (simp add: clauseFalseIffAllLiteralsAreFalse)
    with (~ literalTrue uLiteral valuation: (~ literalFalse uLiteral valuation):
      have precedes (opposite literal') uLiteral (valuation @ valuation')
        A (opposite literal') # uLiteral
        using (uLiteral el valuation')
        using precedesMemberHeadMemberTail [of opposite literal'] valuation uLiteral 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 \( \text{prefix :: Valuation and valuation :: Valuation and clause :: Clause and literal :: Literal} \)
  assumes
  literal el \( \text{prefix} \) and
  isPrefix \( \text{prefix valuation} \) and
  isReason \( \text{clause literal valuation} \)
  shows
  isReason \( \text{clause literal prefix} \)
proof --
  from \( \langle \text{isReason clause literal valuation} \rangle \)
  have
    literal el \( \text{clause} \) and
    clauseFalse (removeAll literal clause) valuation (is \( ?\text{false valuation} \))
and
    \( \forall \text{literal'}. \text{literal'} el (\text{removeAll literal clause}) \rightarrow \)
    precedes (opposite literal') literal valuation \( \land \) opposite literal'
  \( \neq \) literal (is \( ?\text{precedes valuation} \))
  unfolding isReason-def
  by auto
  {
    fix \( \text{literal'} :: \text{Literal} \)
    assume \( \text{literal'} el (\text{removeAll literal clause}) \)
    with \( \langle ?\text{precedes valuation} \rangle \)
    have precedes (opposite literal') literal valuation (opposite literal')
      \( \neq \) literal
      by auto
      with \( \langle \text{literal el prefix} \rangle \langle \text{isPrefix prefix valuation} \rangle \)
      have precedes (opposite literal') literal prefix \( \land \) (opposite literal')
      \( \neq \) literal
      using laterInPrefixRetainsPrecedes [of prefix valuation opposite literal' literal]
      by auto
    }
    note * = this
    hence \( ?\text{precedes prefix} \)
      by auto
    moreover
    have \( ?\text{false prefix} \)
    proof --
      {
fix literal' :: Literal
assume literal' el (removeAll literal clause)
from :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 ?thesis
  by (auto simp add:clauseFalseIffAllLiteralsAreFalse)
qed ultimately
show ?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` :: Literal ⇒ Literal list ⇒ Valuation ⇒ bool
  where
  `isLastAssertedLiteral` literal clause valuation ==
    literal el clause ∧
    literalTrue literal valuation ∧
    (∀ 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` :: Literal list ⇒ Valuation ⇒ Literal
  where
  `getLastAssertedLiteral` clause valuation ==
    last (filter (λ l::Literal. l el clause) valuation)

lemma `getLastAssertedLiteralCharacterization`:
  assumes
    clauseFalse clause valuation
clause ≠ []
  uniq valuation
  shows
    `isLastAssertedLiteral` (`getLastAssertedLiteral` (oppositeLiteralList clause) valuation) (oppositeLiteralList clause) valuation
proof
let \( ?\text{oppc} = \text{oppositeLiteralList} \text{clause} \)
let \( ?l = \text{getLastAssertedLiteral} ?\text{oppc} \text{valuation} \)
let \( ?f = \text{filter} (\lambda l. l \in ?\text{oppc}) \text{valuation} \)

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

have \( \forall l:\text{Literal}. l \in ?\text{oppc} \rightarrow l \in \text{valuation} \)
proof
fix \( l:\text{Literal} \)
show \( l \in ?\text{oppc} \rightarrow l \in \text{valuation} \)
proof
assume \( l \in ?\text{oppc} \)
then obtain \( l':\text{Literal} \)
where \( l' \in ?\text{oppc} \)
by force

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

lemma \( \text{lastAssertedLiteralIsUniq}\):
fixes \( \text{literal :: Literal and } \text{literal'} :: \text{Literal and } \text{literalList :: Literal list and } \text{valuation :: Valuation}\)
assumes
\( \text{lastL}: \text{isLastAssertedLiteral literal literalList valuation } \text{and} \)
\( \text{lastL'}: \text{isLastAssertedLiteral } \text{literal'} \text{ literalList valuation} \)
shows \( \text{literal } = \text{ literal'} \)
using assms
proof
from \( \text{lastL}\) have *:
\( \text{literal el literalList } \)
\( \forall \ l. \ l \text{ el literalList } \land \ l \neq \text{ literal } \rightarrow \neg \text{precedes literal } l \) valuation
\( \text{ultimately}\)
show \( \text{isLastAssertedLiteral literalList valuation}\)
using assms
proof

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
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 ⟨ l ≠ literal ⟩
      show ?thesis
        unfolding precedes-def
        by simp
    next
      case True
      from ⟨ ¬ literalTrue literal valuation ⟩ ⟨ literalTrue literal [literal] ⟩
      ⟨ literalTrue l valuation ⟩
      ⟨ literal l valuation ⟩
      have precedes l literal (valuation @ [literal])
        using precedesModuleHeadMemberTail[of l valuation literal [literal]]
        by auto
      with ⟨ l ≠ 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)

qed
end
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 \in set M
  shows element x \in 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 elementsEmptyIfTrailEmpty [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 \neq []
  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) \subseteq set (elements M)
using assms
unfolding isPrefix-def by auto

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

lemma [simp]:
  assumes (e, d) ∈ set M
  shows e ∈ set (elements M)
  using assms
  by (induct M) auto

lemma uniqImpliesExclusiveTrueOrFalse:
  assumes (e, d) ∈ set M and uniq (elements M)
  shows ¬ (e, ¬ d) ∈ set M
  using assms
proof (induct M)
  case (Cons m M')
  { assume (e, d) = m
    hence (e, ¬ d) ≠ m
      by auto
    from ⟨(e, d) = m⟩ ⟨uniq (elements (m # M'))⟩
    have ¬ (e, d) ∈ set M'
      by (auto simp add: uniqAppendIff)
    with Cons
    have ?case
      by (auto split: split-if-asm)
  }
moreover
  { assume (e, ¬ d) = m
    hence (e, d) ≠ m
    by auto
  }

96
by auto
from \((e, \neg d) = m\) \langle \text{uniq (elements (m \# M'))} \rangle
have \(\neg (e, \neg d) \in \text{set } M'\)
  by (auto simp add: uniqAppendIff)
with Cons
have ?case
  by (auto split: split-if-asm)
} moreover
{
  assume \((e, d) \neq m\) \((e, \neg d) \neq m\)
from \((e, d) \neq m\) \langle (e, d) \in \text{set (m \# M')} \rangle
have \((e, d) \in \text{set } M'\)
  by simp
with \langle \text{uniq (elements (m \# M'))} \rangle
have \(\neg (e, \neg d) \in \text{set } M'\)
  by simp
with \langle (e, \neg d) \neq m \rangle
have ?case
  by simp
}
moreover
{
  have \((e, d) = m \lor (e, \neg d) = m \lor (e, d) \neq m \land (e, \neg d) \neq m\)
    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 \((m \in \text{set (markedElements } M)) = ((m, \text{True}) \in \text{set } M)\)
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**—

from \( m \in \text{set} \ (\text{markedElements} \ M) \) have \( (m, \text{True}) \in \text{set} \ M \)
by (simp add: markedElementIsMarkedTrue)
with \( \text{uniq} \ (\text{elements} \ M) \) \( m \in \text{set} \ (\text{elements} \ p) \)
have \( (m, \text{True}) \in \text{set} \ p \)

**proof**—

\{ Assume \( (m, \text{False}) \in \text{set} \ p \) with \( \text{isPrefix} \ p \ M \),
have \( (m, \text{False}) \in \text{set} \ M \)
using prefixIsSubset[of \ p \ M]
by auto
with \( ((m, \text{True}) \in \text{set} \ M) \) \( \text{uniq} \ (\text{elements} \ M) \)
have \( \text{False} \)
using uniqImpliesExclusiveTrueOrFalse[of \ m \ True \ M]
by simp \}
with \( m \in \text{set} \ (\text{elements} \ p) \)
show \(?\text{thesis} \)
using eitherMarkedOrNotMarked[of \ m \ p]
by auto
qed
thus \(?\text{thesis} \)
using 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**

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

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

**lemma** prefixBeforeElement e t = take (firstPos e (elements t)) t
by (induct t) auto

99
Elements of the trail before the first occurrence of a given element - including it

```plaintext
primrec
prefixToElement :: 'a ⇒ 'a Trail ⇒ 'a Trail
where
  prefixToElement e [] = []
| prefixToElement e (h#t) =
    (if (element h) = e then
      [h]
    else
      (h # (prefixToElement e t))
)

lemma prefixToElement e t = take ((firstPos e (elements t)) + 1) t
by (induct t) auto
```

```plaintext
lemma isPrefixPrefixToElement:
  shows isPrefix (prefixToElement e t) t
unfolding isPrefix-def
by (induct t) auto
```

```plaintext
lemma isPrefixPrefixBeforeElement:
  shows isPrefix (prefixBeforeElement e t) t
unfolding isPrefix-def
by (induct t) auto
```

```plaintext
lemma prefixToElementContainsTrailElement:
  assumes e ∈ set (elements M)
  shows e ∈ set (elements (prefixToElement e M))
using assms
by (induct M) auto
```

```plaintext
lemma prefixBeforeElementDoesNotContainTrailElement:
  assumes e ∈ set (elements M)
  shows e /∈ set (elements (prefixBeforeElement e M))
using assms
by (induct M) auto
```

```plaintext
lemma prefixToElementAppend:
  shows prefixToElement e (M1 @ M2) =
    (if e ∈ set (elements M1) then
      prefixToElement e M1
    else
      M1 @ prefixToElement e M2
    )
by (induct M1) auto
```
lemma prefixToElementToPrefixElement:
  assumes isPrefix p M \AND e \in \text{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'
        by auto
      moreover
      from False have prefixToElement e (a \# p') = a \# prefixToElement e p'
        by simp
      moreover
      from False (e \in \text{set (elements (a \# p'))}) have e \in \text{set (elements p')}
        by simp
      have ? s . (p' @ s = p' @ s)
        by simp
      from (e \in \text{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
    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
  using 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 pre-cede it

definition elementLevel :: '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 markedElementIsMemberOfItsMarkedElementsTo[of M (e, True)]
by simp
hence markedElementsTo e M ≠ []
by auto
thus ?thesis
unfolding elementLevel-def
using length-greater-0-cone[of markedElementsTo e M]
by arith
qed

lemma elementLevelAppend:
assumes a ∈ set (elements M)
shows elementLevel a M = elementLevel a (M @ M′)
using assms
unfolding elementLevel-def
by (simp add: markedElementsToAppend)

lemma elementLevelPrecedesLeq:
assumes
precedes a b (elements M)
shows elementLevel a M ≤ elementLevel b M
using assms
proof (induct M)
case (Cons m M′)
{
assume a = element m
hence ?case
unfolding elementLevel-def
unfolding markedElementsTo-def
by simp
}
moreover
{
assume b = element m
{
assume a ≠ b
hence ¬ precedes a b (b # (elements M′))
by (rule noElementsPrecedesFirstElement)
with (b = element m) (precedes a b (elements (m # M′)));
have False
by simp
}
hence a = b
by auto
hence ?case
by simp
moreover
{
  assume \( a \neq \text{element } m \neq \text{element } m \)
  moreover
  from \( \langle \text{precedes } a \ b \ (\text{elements } (m \neq M')) \rangle \)
  have \( a \in \text{set } (\text{elements } (m \neq M')) \) \( b \in \text{set } (\text{elements } (m \neq M')) \)
    unfolding precedes-def
    by (auto split: split-if-asm)
  from \( \langle a \neq \text{element } m \rangle \langle a \in \text{set } (\text{elements } (m \neq M')) \rangle \)
  have \( a \in \text{set } (\text{elements } M') \)
    by simp
  moreover
  from \( \langle b \neq \text{element } m \rangle \langle b \in \text{set } (\text{elements } (m \neq M')) \rangle \)
  have \( b \in \text{set } (\text{elements } M') \)
    by simp
  ultimately
  have \( \text{elementLevel } a \ M' \leq \text{elementLevel } b \ M' \)
    using Cons
    unfolding precedes-def
    by auto
  hence \( ?\text{case} \)
    using \( \langle a \neq \text{element } m \rangle \langle b \neq \text{element } m \rangle \)
    unfolding elementLevel-def
    unfolding markedElementsTo-def
    by auto
}
ultimately
show \( ?\text{case} \)
  by auto
next
case Nil
thus \( ?\text{case} \)
  unfolding precedes-def
  by simp
qed

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

lemma \text{differentMarkedElementsHaveDifferentLevels}: 
assumes
\( \text{uniq} \; (\text{elements} \; M) \) \text{ and } 
\( a \in \text{set} \; (\text{markedElements} \; M) \) \text{ and } 
\( b \in \text{set} \; (\text{markedElements} \; M) \) \text{ and } 
\( a \neq b \)
shows \( \text{elementLevel} \; a \; M \neq \text{elementLevel} \; b \; M \)
proof –
from \( (a \in \text{set} \; (\text{markedElements} \; M)) \)
have \( a \in \text{set} \; (\text{elements} \; M) \)
by (simp add: \( \text{markedElementsAreElements} \))
moreover
from \( (b \in \text{set} \; (\text{markedElements} \; M)) \)
have \( b \in \text{set} \; (\text{elements} \; M) \)
by (simp add: \( \text{markedElementsAreElements} \))
ultimately
have \( \text{precedes} \; a \; b \; (\text{elements} \; M) \lor \text{precedes} \; b \; a \; (\text{elements} \; M) \)
using \( (a \neq b) \)
using \( \text{precedesTotalOrder}[\text{of} \; a \; \text{elements} \; M \; b] \)
by simp
moreover
\( \{ \)
assume \( \text{precedes} \; a \; b \; (\text{elements} \; M) \)
with \( \text{assms} \)
have \( ?\text{thesis} \)
using \( \text{elementLevelPrecedesMarkedElementLt}[\text{of} \; M \; a \; b] \)
by auto
\}
moreover
\{
\textbf{assume} \ precedes \ b \ a \ (\text{elements} \ M)
\textbf{with} \ \text{assms}
\textbf{have} \ \text {?thesis}
\textbf{using} \ element\text{Level}\text{PrecedesMarkedElementLt}\ [\text{of} \ M \ b \ a]
\textbf{by} \ auto
\}
ultimately
\textbf{show} \ \text {?thesis}
\textbf{by} \ auto
\textbf{qed}

\section{3.7 Current trail level}

Current level is the number of marked elements in the trail

\textbf{definition}
\textit{currentLevel} :: \texttt{`a Trail }\Rightarrow\texttt{ nat}
\textbf{where}
\textit{currentLevel} \ t = \text{length} \ (\text{markedElements} \ t)

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

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

\textbf{lemma} \ element\text{Level}\text{LeqCurrentLevel}:
\textbf{shows} \ element\text{Level} \ a \ M \leq \text{currentLevel} \ M
\textbf{proof}--
\textbf{have} \ isPrefix \ (prefix\text{ToElement} \ a \ M) \ M
\textbf{using} \ isPrefix\text{PrefixToElement}[\text{of} \ a \ M]
.\textbf{then} \ \textbf{obtain} \ s
\textbf{where} \ prefix\text{ToElement} \ a \ M @ s = M
\textbf{unfolding} \ isPrefix-def
\textbf{by} \ auto
\textbf{hence} \ M = \text{prefixToElement} \ a \ M @ s
\textbf{by} \ (\text{rule sym})
\textbf{hence} \ \text{currentLevel} \ M = \text{currentLevel} \ (\text{prefixToElement} \ a \ M @ s)
\textbf{by} \ \text{simp}
hence \( \text{currentLevel}\ M = \text{length} (\text{markedElements} (\text{prefixToElement} a\ M)) + \text{length} (\text{markedElements} s) \)

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

lemma elementOnCurrentLevel:
  assumes \( a \notin \text{set} (\text{elements}\ M) \)
  shows \( \text{elementLevel}\ a\ (M \@ [(a, d)]) = \text{currentLevel}\ (M \@ [(a, d)]) \)
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
\((\text{prefixToLevel-aux} \[
\]
l cl) = \[
\]
| (\text{prefixToLevel-aux} (h\#t) l cl) =
  (if (\text{marked}\ h) \text{then}
  (if (cl \geq l) \text{then} \[
  \]
    else \[
  \]
    (h \# (\text{prefixToLevel-aux} t l (cl+1))))
  else
  (h \# (\text{prefixToLevel-aux} t l cl))
  )

definition
prefixToLevel :: \nat \Rightarrow\) \('a Trail\)
where
prefixToLevel-def: (prefixToLevel l t) == (prefixToLevel-aux l t l 0)

lemma isPrefixPrefixToLevel-aux:
  shows \( \exists\ s.\ \text{prefixToLevel-aux} t l i \@ s = t \)
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
by simp

lemma currentLevelPrefixToLevel-aux:
  assumes l ≥ i
  shows currentLevel (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 currentLevel-def
      by simp }
moreover
  { assume marked m i < l
    hence ?case
      using Cons [of i+1]
      unfolding currentLevel-def
      by simp }
moreover
  { assume ¬ marked m
    hence ?case
      using Cons
      unfolding currentLevel-def
      by simp }
ultimately
show ?case
  using ⟨i <= l⟩
  by auto
next
case Nil
  thus ?case
  unfolding currentLevel-def
  by simp
qed

lemma currentLevelPrefixToLevel:
  shows currentLevel (prefixToLevel level M) ≤ level
using currentLevelPrefixToLevel-aux[of 0 level M]
unfolding prefixToLevel-def
by simp

lemma currentLevelPrefixToLevelEq-aux:
  assumes l ≥ i currentLevel M >= l - i
shows currentLevel (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 currentLevel-def
      by simp
  }
moreover
  {
    assume marked m i < l
    hence ?case
      using Cons(1) [of i+1]
      using Cons(3)
      unfolding currentLevel-def
      by simp
  }
moreover
  {
    assume ¬ marked m
    hence ?case
      using Cons unfolding currentLevel-def
      by simp
  }
ultimately
  show ?case
    using ⟨i <= l⟩
    by auto
next
  case Nil
  thus ?case
    unfolding currentLevel-def
    by simp
qed

lemma currentLevelPrefixToLevelEq:
assumes
  level ≤ currentLevel M
shows
  currentLevel (prefixToLevel level M) = level
using assms
unfolding prefixToLevel-def
using currentLevelPrefixToLevelEq-aux[of 0 level M]
by simp

lemma prefixToLevel-auxIncreaseAuxilaryCounter:
assumes \( k \geq i \)
shows \( \text{prefixToLevel-aux} \ M \ l \ i = \text{prefixToLevel-aux} \ M \ (l + (k - i)) \)

using \( \text{assms} \)

**proof** (induct \( M \) arbitrary: \( i \ k \))

  case (\( \text{Cons} \ m \ M' \))
  {
  assume \( \neg \text{marked} \ m \)
  hence \( ?\text{case} \)
  using \( \text{Cons}(1)[\text{of} \ i \ k] \ \text{Cons}(2) \)
  by simp
  }

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

moreover
{
  assume \( i < l \) \( \text{marked} \ m \)
  hence \( ?\text{case} \)
  using \( \text{Cons}(1)[\text{of} \ i+1 \ k+1] \ \text{Cons}(2) \)
  by simp
  }

ultimately
show \( ?\text{case} \)
  by (auto split: split-if-asm)

qed simp

**lemma** \( \text{isPrefixPrefixToLevel-auxLowerLevel} \):

assumes \( i \leq j \)
shows \( \text{isPrefix} \ (\text{prefixToLevel-aux} \ M \ i \ k) \ (\text{prefixToLevel-aux} \ M \ j \ k) \)
using \( \text{assms} \)
by (induct \( M \) arbitrary: \( k \)) (auto simp add:isPrefix-def)

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

assumes \( \text{level} < \text{level}' \)
shows \( \text{isPrefix} \ (\text{prefixToLevel} \ \text{level} \ M) \ (\text{prefixToLevel} \ \text{level}' \ M) \)
using \( \text{assms} \)
unfolding \( \text{prefixToLevel-def} \)
using \( \text{isPrefixPrefixToLevel-auxLowerLevel}[\text{of} \ \text{level} \ \text{level}' \ M \ 0] \)
by simp

**lemma** \( \text{prefixToLevel-auxPrefixToLevel-auxHigherLevel} \):

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

lemma prefixToLevelPrefixToLevelHigherLevel:
  assumes level ≤ 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 ≥ 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 ¬ marked a
    hence ?case
    using Cons(1)[of i] (i ≤ l) (l − i < currentLevel (a # a'))
    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] (i ≤ l) (l − i < currentLevel (a # a'))
    unfolding currentLevel-def
    by simp
  }
  ultimately
  show ?case
  using (i ≤ l)
  by auto
next
case Nil
thus ?case
lemma \texttt{prefixToLevelAppend-aux2}:
\begin{itemize}
\item \textbf{assumes}\, \,
\begin{align*}
i \leq l \text{ and } \textit{currentLevel} a + i & \leq l
\end{align*}
\item \textbf{shows}\, \,
\begin{align*}
\text{prefixToLevel-aux} (a @ b) l i = a @ \text{prefixToLevel-aux} b l (i + (\textit{currentLevel} a))
\end{align*}
\end{itemize}
\textbf{using} \texttt{assms}
\begin{proof} \texttt{(induct a arbitrary; i)} \end{proof}
\begin{itemize}
\item \textbf{case} \texttt{(Cons a a')} \begin{itemize}
\item \texttt{assume} \neg \texttt{marked} a
\item \texttt{hence} \texttt{?case}
\item \texttt{using Cons}
\item \texttt{unfolding currentLevel-def}
\item \texttt{by simp}
\end{itemize}
\end{itemize}
\begin{itemize}
\item \texttt{moreover} \begin{itemize}
\item \texttt{assume} \texttt{marked} a l = i
\item \texttt{hence} \texttt{?case}
\item \texttt{using} \langle (\textit{currentLevel} (a # a')) + i \leq b \rangle
\item \texttt{unfolding currentLevel-def}
\item \texttt{by simp}
\end{itemize}
\end{itemize}
\begin{itemize}
\item \texttt{moreover} \begin{itemize}
\item \texttt{assume} \texttt{marked} a l > i
\item \texttt{hence} \texttt{prefixToLevel-aux} (a' @ b) l (i + 1) = a' @ \text{prefixToLevel-aux} b l (i + 1 + \textit{currentLevel} a')
\item \texttt{using Cons(1) [of i + 1 | (currentLevel (a # a')) + i \leq b]}
\item \texttt{unfolding currentLevel-def}
\item \texttt{by simp}
\end{itemize}
\end{itemize}
\begin{itemize}
\item \texttt{moreover} \begin{itemize}
\item \texttt{have} i + 1 + \texttt{length} (\texttt{markedElements} a') = i + (1 + \texttt{length} (\texttt{markedElements} a'))
\item \texttt{by simp}
\end{itemize}
\end{itemize}
\begin{itemize}
\item \texttt{ultimately} \begin{itemize}
\item \texttt{have} \texttt{?case}
\item \texttt{using} \langle \texttt{marked} a \rangle \langle l > i \rangle
\item \texttt{unfolding currentLevel-def}
\item \texttt{by simp}
\end{itemize}
\end{itemize}
\begin{itemize}
\item \texttt{ultimately} \begin{itemize}
\item \texttt{show} \texttt{?case}
\item \texttt{using} \langle l \geq i \rangle
\end{itemize}
\end{itemize}
by auto

next
  case Nil
  thus \( ?\text{case} \)
      unfolding \( \text{currentLevel-def} \)
      by simp

qed

lemma \( \text{prefixToLevelAppend} \):
  shows \( \text{prefixToLevel level (a @ b)} = \)
  (if level < \( \text{currentLevel a} \) then
    \( \text{prefixToLevel level a} \)
  else
    \( a @ \text{prefixToLevel-aux b level (currentLevel a)} \)
)

proof (cases level < \( \text{currentLevel a} \) )
  case True
  thus \( ?\text{thesis} \)
      unfolding \( \text{prefixToLevel-def} \)
      using \( \text{prefixToLevelAppend-aux1[of 0 level a]} \)
      by simp

next
  case False
  thus \( ?\text{thesis} \)
      unfolding \( \text{prefixToLevel-def} \)
      using \( \text{prefixToLevelAppend-aux2[of 0 level a]} \)
      by simp

qed

lemma \( \text{isProperPrefixPrefixToLevel} \):
  assumes level < \( \text{currentLevel t} \)
  shows \( \exists s. (\text{prefixToLevel level t}) @ s = t \land s \neq [] \land (\text{marked (hd s)}) \)

proof

  have \( \text{isPrefix (prefixToLevel level t)} t \)
     by (simp add: isPrefixPrefixToLevel)
  then obtain \( s ::'a \text{ Trail} \)
        where \( (\text{prefixToLevel level t}) @ s = t \)
        unfolding isPrefix-def
        by auto

  moreover
  have \( s \neq [] \)

proof

  { \( \text{assume s = []} \)
      with \( (\text{prefixToLevel level t}) @ s = t; \)
      have \( \text{prefixToLevel level t} = t \)
         by simp
      hence \( \text{currentLevel (prefixToLevel level t)} \leq \text{level} \)

using \texttt{currentLevelPrefixToLevel} of level \texttt{t}

by \texttt{simp}

with \texttt{(prefixToLevel level t = t)} have \texttt{currentLevel t \leq level}

by \texttt{simp}

with \texttt{(level < currentLevel t)} have \texttt{False}

by \texttt{simp}

\}

thus \texttt{?thesis}

by \texttt{auto}

\}

qed

moreover

have \texttt{marked (hd s)}

proof –

\{

assume \texttt{\neg marked (hd s)}

have \texttt{currentLevel (prefixToLevel level t) \leq level}

by \texttt{(simp add: currentLevelPrefixToLevel)}

from \texttt{: s \neq [] have s = [hd s] @ (tl s)}

by \texttt{simp}

with \texttt{(prefixToLevel level t) @ s = t} have \texttt{t = (prefixToLevel level t) @ [hd s] @ (tl s)}

by \texttt{simp}

hence \texttt{(prefixToLevel level t) = (prefixToLevel level ((prefixToLevel level t) @ [hd s] @(tl s)))}

by \texttt{simp}

also

with \texttt{(currentLevel (prefixToLevel level t) \leq level)}

have \texttt{... = (prefixToLevel level t) @ (prefixToLevel-aux ([hd s] @ (tl s)) level (currentLevel (prefixToLevel level t)))}

by \texttt{(auto simp add: prefixToLevelAppend)}

also

have \texttt{... = (prefixToLevel level t) @ (hd s) \# prefixToLevel-aux (tl s) level (currentLevel (prefixToLevel level t))}

proof –

from \texttt{: currentLevel (prefixToLevel level t) \leq level \neg marked (hd s)}

have \texttt{prefixToLevel-aux ([hd s] @ (tl s)) level (currentLevel (prefixToLevel level t)) = (hd s) \# prefixToLevel-aux (tl s) level (currentLevel (prefixToLevel level t))}

by \texttt{simp}

thus \texttt{?thesis}

by \texttt{simp}

\}

qed

ultimately

have \texttt{(prefixToLevel level t) = (prefixToLevel level t) @ (hd s) \# prefixToLevel-aux (tl s) level (currentLevel (prefixToLevel level t))}

by \texttt{simp}

117
hence False  
  by auto  
}  
thus ?thesis  
  by auto  
qed  
ultimately  
show ?thesis  
  by auto  
qed

lemma prefixToLevelElementsElementLevel:  
  assumes  
  e ∈ set (elements (prefixToLevel level M))  
  shows  
  elementLevel e M ≤ level  
proof –  
  have elementLevel e (prefixToLevel level M) ≤ currentLevel (prefixToLevel level M)  
    by (simp add: elementLevelLeqCurrentLevel)  
  moreover  
  hence currentLevel (prefixToLevel level M) ≤ level  
    using currentLevelPrefixToLevel[of level M]  
    by simp  
  ultimately have elementLevel e (prefixToLevel level M) ≤ level  
    by simp  
  moreover  
  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 (e ∈ set (elements (prefixToLevel level M))):  
  have elementLevel e (prefixToLevel level M) = elementLevel e M  
    using elementLevelAppend [of e prefixToLevel level M s]  
    by simp  
  ultimately  
  show ?thesis  
    by simp  
qed

lemma elementLevelLtLevelImpliesMemberPrefixToLevel-aux:  
  assumes  
  e ∈ set(elements M) and  
  elementLevel e M + i ≤ level and  
  i ≤ level  
  shows  
  e ∈ set (elements (prefixToLevel-aux M level i))
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))
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-startIrellevant:
\(\forall y. y \neq [] \implies \text{levelsCounter-aux} a (x @ y) = (x @ \text{levelsCounter-aux} a y)\)
by (induct a) (auto simp add: butlastAppend)

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

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

**lemma levelsCounter-auxIncreasesFirst:**
\(\forall m n l1 l2. \text{levelsCounter-aux} a (m \# l1) = n \# l2 \implies m \leq n\)
**proof** (induct a)
\begin{cases}
    \text{case Nil}
    \{
        \text{fix } m::\text{nat} \text{ and } n::\text{nat} \text{ and } l1::\text{nat list} \text{ and } l2::\text{nat list}
        \text{assume } \text{levelsCounter-aux} [] (m \# l1) = n \# l2
        \text{hence } m = n
        \text{by simp}
    \}
    \text{thus } ?\text{case}
    \text{by simp}
\end{cases}
\text{next}
\text{case } (\text{Cons } a \text{ list})
\begin{cases}
    \text{fix } m::\text{nat} \text{ and } n::\text{nat} \text{ and } l1::\text{nat list} \text{ and } l2::\text{nat list}
    \text{assume } \text{levelsCounter-aux} (a \# \text{list}) (m \# l1) = n \# l2
    \text{have } m \leq n
    \text{proof } (\text{cases } \text{marked } a)
    \text{case True}
    \text{with } \text{levelsCounter-aux} (a \# \text{list}) (m \# l1) = n \# l2
    \text{have } \text{levelsCounter-aux list} (m \# l1 @ [\text{Suc } 0]) = n \# l2
    \text{by simp}
    \text{with } \text{Cons}
    \text{show } ?\text{thesis}
    \text{by auto}
    \text{next}
    \text{case False}
    \text{show } ?\text{thesis}
    \text{proof } (\text{cases } l1 = [])
    \text{case True}
    \text{with } \text{(marked } a) \text{ levelsCounter-aux} (a \# \text{list}) (m \# l1) = n \# l2)
    \text{have } \text{levelsCounter-aux list} [\text{Suc } m] = n \# l2
    \text{by simp}
    \text{with } \text{Cons}
    \text{have } \text{Suc } m \leq n
\end{cases}
thesis by auto
thus thesis
by simp
next
case False
with (~ marked a: levelsCounter-aux (a # list) (m # l1) = n 
# l2)
known levelsCounter-aux list (m # butlast l1 @ [Suc (last l1)])
= n # l2
by simp
with Cons
show thesis
by auto
qed
qed

lemma levelsCounterPrefix:
assumes (isPrefix p a)
shows ? rest. rest ≠ [] ∧ levelsCounter a = butlast (levelsCounter p) @ rest ∧ last (levelsCounter p) ≤ hd rest
proof-
from assms
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 \ldots = levelsCounter-aux s (levelsCounter-aux p [0])
  unfolding levelsCounter-def
  by (simp add: levelsCounter-auxSuffixContinues)
also
have \ldots = levelsCounter-aux (tl s) ((levelsCounter-aux p [0]) @
[1])
proof-
  from \langle s \neq [] \rangle 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 \langle marked (hd s) \rangle show ?thesis
  by simp
qed
also
have \ldots = levelsCounter-aux p [0] @ (levelsCounter-aux (tl s)
[1])
  by (simp add: levelsCounter-aux-startIrellevant)
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])) \wedge
  (last (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 \langle p @ s = a \rangle have levelsCounter a = levelsCounter
  (p @ s)
  by simp
also
have \ldots = levelsCounter-aux s (levelsCounter-aux p [0])
  unfolding levelsCounter-def
  by (simp add: levelsCounter-auxSuffixContinues)
also
have \ldots = levelsCounter-aux (tl s) ((butlast (levelsCounter-aux
p [0])) @ [Suc (last (levelsCounter-aux p [0]))])
  proof-
  from \langle s \neq [] \rangle have s = hd s \# tl s
  by simp

124
then have levelsCounter-aux \textit{s (levelsCounter-aux p [0])} =
levelsCounter-aux (hd s \# tl s) (levelsCounter-aux p [0])
  by simp
with \(\sim\) marked (hd s):
  show \(?\)thesis
  by simp
qed
also have \(\ldots = \text{butlast (levelsCounter-aux p [0]) @ (levelsCounter-aux}
(tl s) [Suc (last (levelsCounter-aux p [0])))])
  by (simp add: levelsCounter-aux-startIrrelevant)
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])))]) \(\geq\) Suc (last (levelsCounter-aux p [0]))
  proof
    have (levelsCounter-aux (tl s) [Suc (last (levelsCounter-aux
p [0])))]) \(\neq\) []
      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-conv[of (levelsCounter-aux (tl s) [Suc (last
(levelsCounter-aux p [0])))])]
        by auto
      hence h \(\geq\) 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
    qed
  qed
ultimately
  have levelsCounter a = butlast (levelsCounter-aux p) @
(levelsCounter-aux (tl s) [Suc (last (levelsCounter-aux
p [0])))]) \^ last (levelsCounter a) \(\leq\) 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
qed
lemma `levelsCounterPrefixToLevel`:
assumes `p = prefixToLevel 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 p @ (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.

**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.

**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 \( (F_0::\text{Formula}) (F::\text{Formula}) == \text{equivalentFormulae} F_0 F \)

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

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

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

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 \ l. \ l \text{ el} \ (\text{elements} M) \land \neg \text{literal el} \text{decisions} M \rightarrow \exists \text{clause}. \text{formulaEntailsClause} F \text{ clause} \land \text{isReason} \text{ clause literal} (\text{elements} M)
\shows \forall \ l. \ l \text{ el} \text{elements} M \rightarrow \text{formulaEntailsLiteral} (F @ \text{val2form} (\text{decisionsTo} l M)) l
\proof
\{
\fix l :: \text{Literal}
\assume l \text{ el} \text{elements} M
\with \text{assms}
\have \text{formulaEntailsLiteral} (F @ \text{val2form} (\text{decisionsTo} l M)) l
\by \text{simp}
\have \text{isPrefix} (\text{decisionsTo} l M) (\text{decisions} M)
\by (\text{simp add: markedElementsToArePrefixOfMarkedElements})
\then \obtain s :: \text{Valuation}
}
where \((\text{decisionsTo } l \ M)@s = (\text{decisions } M)\)

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

by auto

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

by (rule sym)

with \((\text{formulaEntailsLiteral } (F @ \text{val2form } (\text{decisionsTo } l \ M)))\) l

have \((\text{formulaEntailsLiteral } (F @ \text{val2form } (\text{decisions } M)))\) l

using formulaEntailsLiteralAppend [of \(F \ @ \text{val2form } (\text{decisionsTo } l \ M)\) \(l \ \text{val2form } s\)]

by (auto simp add: formulaEntailsLiteralAppend val2formAppend)

thus \(?\text{thesis}\)

by simp

qed

lemma InvariantImpliedLiteralsAndElementsEntailLiteralThenDecision-sEntailLiteral:

fixes \(M :: \text{LiteralTrail} \ and \ F :: \text{Formula} \ and \ \text{literal} :: \text{Literal}\)

assumes \(\text{InvariantImpliedLiterals } F \ M \ and \ \text{formulaEntailsLiteral } (F \ @ \text{val2form } (\text{elements } M)))\) literal

shows \(\text{formulaEntailsLiteral } (F \ @ \text{val2form } (\text{decisions } M)))\) literal

proof –

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

  assume \(\text{model valuation } (F \ @ \text{val2form } (\text{decisions } M))\)

  hence \(\text{formulaTrue } F \ \text{valuation} \ \text{and} \ \text{formulaTrue } (\text{val2form } (\text{decisions } M))\) \ \text{val2form} \ \text{valuation} \ \text{and} \ \text{consistent valuation}

  by (auto simp add: formulaTrueAppend)

\}

  fix \(l :: \text{Literal}\)

  assume \(\text{l el } (\text{elements } M)\)

  from \(\text{InvariantImpliedLiterals } F \ M\)

  have \(\forall \ l. \ \text{l el } (\text{elements } M) \implies \text{formulaEntailsLiteral } (F \ @ \text{val2form } (\text{decisions } M)))\) l

  by (simp add: InvariantImpliedLiteralsWeakerVariant InvariantImpliedLiterals-def)

  with \(\text{l el } (\text{elements } M)\)

  have \(\text{formulaEntailsLiteral } (F \ @ \text{val2form } (\text{decisions } M)))\) l

  by simp

  with \(\text{model valuation } (F \ @ \text{val2form } (\text{decisions } M))\)

  have \(\text{literalTrue } l \ \text{valuation}\)

  by (simp add: formulaEntailsLiteral-def)

\}

hence \(\text{formulaTrue } (\text{val2form } (\text{elements } M)))\) \ \text{valuation}

by (simp add: val2formFormulaTrue)

with \(\text{formulaTrue } F \ \text{valuation} \ \langle \text{consistent valuation} \rangle\)

have \(\text{model valuation } (F \ @ \text{val2form } (\text{elements } M))\)

by (auto simp add: formulaTrueAppend)

with \(\text{formulaEntailsLiteral } (F \ @ \text{val2form } (\text{elements } M)))\) literal

have \(\text{literalTrue } l \ \text{valuation}\)
by \{ simp add: \textit{formulaEntailsLiteral-def} \}

thus \textit{thesis}
  by (simp add: \textit{formulaEntailsLiteral-def} )

qed

\textbf{lemma InvariantImpliedLiteralsAndFormulaFalseThenFormulaAndDecisionsAreNotSatisfiable:}

\texttt{fixes } M :: \texttt{LiteralTrail} \texttt{ and } F :: \texttt{Formula}
\texttt{ assumes InvariantImpliedLiterals F M and formulaFalse F (elements M)}
\texttt{ shows } \neg \texttt{satisfiable (F @ val2form (decisions M))}

proof –
\texttt{ from } (formulaFalse F (elements M));
\texttt{ have formulaFalse (F @ val2form (decisions M)) (elements M)}
\quad \texttt{by (simp add: formulaFalseAppend)}

moreover
\texttt{ from } (InvariantImpliedLiterals F M);
\texttt{ have formulaEntailsValuation (F @ val2form (decisions M)) (elements M)}
\quad \texttt{unfolding formulaEntailsValuation-def}
\quad \texttt{unfolding InvariantImpliedLiterals-def}
\quad \texttt{using InvariantImpliedLiteralsWeakerVariant[of M F]}
\quad \texttt{by simp}

ultimately
\texttt{ show \textit{thesis}}
\quad \texttt{using formulaFalseInEntailedValuationIsUnsatisfiable [of F @ val2form (decisions M) elements M]}
\quad \texttt{by simp}

qed

\textbf{lemma InvariantImpliedLiteralsHoldsForPrefix:}

\texttt{fixes } M :: \texttt{LiteralTrail} \texttt{ and prefix :: LiteralTrail and } F :: \texttt{Formula}
\texttt{ assumes InvariantImpliedLiterals F M and isPrefix prefix M}
\texttt{ shows InvariantImpliedLiterals F prefix}

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

\texttt{ from * (isPrefix prefix M)}
\texttt{ have l \texttt{ el elements M} }
\quad \texttt{unfolding isPrefix-def}
\quad \texttt{by auto}

\texttt{ from * and (isPrefix prefix M)}
\texttt{ have decisionsTo l prefix = decisionsTo l M}
\quad \texttt{using markedElementsToPrefixElement [of prefix M l]}
\quad \texttt{by simp}

\}
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)
      ⟨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 } \text{el} \text{ elements } p) \land \neg \text{literal } \text{el} \text{ decisions } p) \land \text{isPrefix } p \text{ M} \rangle

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

have \(\exists \text{ clause. } \text{formulaEntailsClause } F \text{ clause} \land \text{isReason clause literal (elements } p)\)
by auto

} thus thesis

unfolding InvariantReasonClauses-def
by auto

qed

lemma InvariantReasonClausesHoldsForPrefixElements:

fixes \(F::\text{Formula} \text{ and } M::\text{LiteralTrail} \text{ and } p::\text{LiteralTrail}\)

assumes InvariantReasonClauses \(F \text{ p and isPrefix } p \text{ M and}
\text{literal el (elements } p) \land \neg \text{literal el decisions } M\)

shows \(\exists \text{ clause. } \text{formulaEntailsClause } F \text{ clause} \land \text{isReason clause literal (elements } M)\)

proof –
from \((\text{isPrefix } p \text{ M}) \land \neg \text{literal el (decisions } M)\)

have \(\neg \text{literal el (decisions } p)\)
using markedElementsPrefixAreMarkedElementsTrail[of p M literal]
by auto

from \((\text{InvariantReasonClauses } F \text{ p (literal el (elements } p)) \land \neg \text{literal el (decisions } p)\) obtain clause :: \text{Clause}
where \text{formulaEntailsClause } F \text{ clause isReason clause literal (elements } p)\)

unfolding InvariantReasonClauses-def
by auto

with \((\text{isPrefix } p \text{ M})\)

have isReason clause literal (elements M)
using isReasonAppend [of clause literal elements p]
by (auto simp add: isPrefix-def)
with \(\text{formulaEntailsClause } F \text{ clause}\)

show thesis
by auto

qed

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** $\text{InvariantUniq } M$ and $\text{var literal } \notin \text{vars (elements } M)$ and $M' = M@[(\text{literal}, \text{True})]$ shows $\text{InvariantUniq } M'$

**proof** –

from $\langle \text{InvariantUniq } M \rangle$

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

unfolding $\text{InvariantUniq-def}$.

{  
  assume $\neg \text{uniq (elements } M')$
  with $\langle \text{uniq (elements } M') \rangle \langle M' = M@[(\text{literal, True})] \rangle$
  have $\text{literal el (elements } M)$
  using $\text{uniqButlastNotUniqListImpliesLastMemButlast [of elements } M' \rangle$
  by auto
  hence $\text{var literal } \in \text{vars (elements } M)$
  using $\text{valuationContainsItsLiteralsVariable [of literal elements } M \rangle$
  by simp
  with $\langle \text{var literal } \notin \text{vars (elements } M) \rangle$
  have $\text{False}$
  by simp
}

thus $\langle \text{thesis} \rangle$

unfolding $\text{InvariantUniq-def}$

by auto

qed

**Lemma InvariantImpliedLiteralsAfterDecide:**

**fixes** $F :: \text{Formula}$ and $M :: \text{LiteralTrail}$ and $\text{literal} :: \text{Literal}$ and $M' :: \text{LiteralTrail}$

**assumes** $\text{InvariantImpliedLiterals } F \text{ } M$ and $\text{var literal } \notin \text{vars (elements } M)$ and $M' = M@[(\text{literal, True})]$ shows $\text{InvariantImpliedLiterals } F \text{ } M'$

**proof** –

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

lemma \textbf{InvariantVarsMAfterDecide}:
fixes \( F :: \text{Formula } \text{and } F0 :: \text{Formula } \text{and } M :: \text{LiteralTrail } \text{and } \)
literal :: \text{Literal } \text{and } M' :: \text{LiteralTrail}
assumes \( \text{InvariantVarsM } M \ F0 \ Vbl \text{ and } \)
var \( \text{literal} \in Vbl \text{ and } \)

136
\[M' = M @ [(\text{literal, True}])\]
shows \(\text{InvariantVarsM M'} F0 Vbl\)

**proof** –
from \(\langle \text{InvariantVarsM M F0 Vbl} \rangle\)
have \(\text{vars (elements } M) \subseteq \text{vars } F0 \cup Vbl\)
  by \((\text{simp only:InvariantVarsM-def})\)
from \(\langle M' = M @ [(\text{literal, True}])\rangle\)
have \(\text{vars (elements } M') = \text{vars (elements } (M @ [(\text{literal, True}])])\)
  by simp
also have \(\ldots = \text{vars (elements } M @ [\text{literal}]\)
  by simp
also have \(\ldots = \text{vars (elements } M) \cup \text{vars [literal]}\)
  using \(\text{varsAppendClauses [of elements } M [\text{literal}]\]
  by simp
finally
show \(?\text{thesis}\)
  using \(\langle \text{vars (elements } M) \subseteq (\text{vars } F0) \cup Vbl \rangle\) \(\langle \text{var literal } \in Vbl\rangle\)
  unfolding \(\text{InvariantVarsM-def}\)
  by auto
qed

**lemma** \(\text{InvariantConsistentAfterDecide}:\)
fixes \(M :: \text{LiteralTrail} \) and \(\text{literal :: Literal} \) and \(M' :: \text{LiteralTrail} \)
assumes \(\text{InvariantConsistent } M \) and
\(\text{var literal } \notin \text{vars (elements } M) \) and
\(M' = M @ [(\text{literal, True}])\)
shows \(\text{InvariantConsistent } M'\)

**proof** –
from \(\langle \text{InvariantConsistent } M \rangle\)
have \(\text{consistent (elements } M)\)
  unfolding \(\text{InvariantConsistent-def}\)
  
  
  \{  
  assume \(\text{inconsistent (elements } M')\)
  with \(\langle M' = M @ [(\text{literal, True}])\rangle\)
  have \(\text{inconsistent (elements } M) \vee \text{inconsistent [literal]} \vee (\exists \ l. \ \text{literalTrue } l \ (\text{elements } M) \wedge \text{literalFalse } l [\text{literal}])\)
  using \(\text{inconsistentAppend [of elements } M [\text{literal}]\]
  by simp
  with \(\langle \text{consistent (elements } M)\rangle\) \(\langle \text{obtain } l :: \text{Literal}\)
  where \(\text{literalTrue } l \ (\text{elements } M) \) and \(\text{literalFalse } l [\text{literal}]\)
  by auto
  hence \(\text{(opposite } l) = \text{literal}\)
  by auto
  hence \(\text{var literal } = \text{var } l\)
  by auto
  with \(\langle \text{literalTrue } l \ (\text{elements } M)\rangle\)
  have \(\text{var } l \in \text{vars (elements } M)\)
  using \(\text{valuationContainsItsLiteralsVariable [of } l \text{ elements } M]\)

137
by simp
with (var literal = var l; var literal \notin vars elements M)
have False
  by simp
}
thus ?thesis
  unfolding InvariantConsistent-def
  by auto
qed

lemma InvariantReasonClausesAfterDecide:
  fixes F :: Formula and M :: LiteralTrail and M' :: LiteralTrail
  assumes InvariantReasonClauses F M and InvariantUniq M and
          M' = M @ [(l literal, True)]
  shows InvariantReasonClauses F M'
proof -
  { fix literal' :: Literal
  assume literal' el elements M' and \neg literal' el decisions M'

    have \exists clause. formulaEntailsClause F clause \land isReason clause literal' (elements M')
      proof (cases literal' el elements M)
      case True
        with assms \neg literal' el decisions M' obtain clause::Clause
          where formulaEntailsClause F clause \land 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 (M' = M @ [(l literal, True)]) (literal' el elements M')
        have literal = literal'
          by (simp split: split-if-asm)
        with (M' = M @ [(l literal, True)])
          have literal' el decisions M'
            using markedElementIsMarkedTrue[of literal M']
              by simp
        with \neg literal' el decisions M'
        have False
          by simp
        thus ?thesis
          by simp
      qed
    } thus ?thesis
  138
unfolding InvariantReasonClauses-def by auto

lemma InvariantCFalseAfterDecide:
  fixes conflictFlag::bool and M::LiteralTrail and C::Clause
  assumes InvariantCFalse conflictFlag M C and \( M' = M \circ[(\text{literal, True})]\)
  shows InvariantCFalse conflictFlag M' C
  unfolding InvariantCFalse-def
proof
  assume conflictFlag
  show clauseFalse C (elements M')
  proof
    from \( \langle \text{InvariantCFalse conflictFlag M C} \rangle \)
    have conflictFlag \rightarrow clauseFalse C (elements M)
    unfolding InvariantCFalse-def.
    with \( \langle \text{conflictFlag} \rangle \)
    have clauseFalse C (elements M)
      by simp
    with \( \langle M' = M \circ[(\text{literal, True})]\rangle \)
    show \( ?\text{thesis} \)
      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
  \( M' = M \circ[(\text{uLiteral, False})]\)
  shows formulaEntailsLiteral (F @ val2form (decisionsTo uLiteral M')) uLiteral
proof
  have decisionsTo uLiteral M' = decisions M
  proof
    from \( \langle \text{isUnitClause uClause uLiteral (elements M)} \rangle \)
    have \( \neg \text{uLiteral el (elements M)} \)
      by (simp add: isUnitClause-def)
    with \( \langle M' = M \circ[(\text{uLiteral, False})]\rangle \)
    show \( ?\text{thesis} \)
      using markedElementsToAppend[uLiteral M [(\text{uLiteral, False})]]
      unfolding markedElementsTo-def
      by simp
  qed

139
moreover
from \(\text{formulaEntailsClause} \ F \ \text{uClause} \rangle \ \langle \text{isUnitClause} \ \text{uClause} \ \text{uLiteral} \ \langle \text{elements} \ M \rangle\)
have \(\text{formulaEntailsLiteral} \ (F @ \text{val2form} \ \langle \text{elements} \ M \rangle) \ \text{uLiteral}\)
using \(\text{unitLiteralIsEntailed} \ [\text{of} \ \text{uClause} \ \text{uLiteral} \ \langle \text{elements} \ M \ F \rangle]\)
by simp
with \(\text{InvariantImpliedLiterals} \ F \ M\)
have \(\text{formulaEntailsLiteral} \ (F @ \text{val2form} \ \langle \text{decisions} \ M \rangle) \ \text{uLiteral}\)
by (simp add: \(\text{InvariantImpliedLiteralsAndElementsEntailLiteralThenDecisionsEntailLiteral}\))
ultimately
show \(?\text{thesis}\)
by simp
qed

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

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

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

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

lemma InvariantConsistentAfterUnitPropagate:
  fixes \(M \:: \text{LiteralTrail}\) and \(F \:: \text{Formula}\) and \(M' \:: \text{LiteralTrail}\) 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
  \(M' = M \@ [(\text{uLiteral}, False)]\)
  shows InvariantConsistent \(M'\)
proof (−)
  from \(\langle \text{InvariantConsistent } M\rangle\)
  have \(\text{consistent } (\text{elements } M)\)
    unfolding InvariantConsistent-def
  .
  from \(\langle \text{isUnitClause } u\text{Clause} u\text{Literal} \ (\text{elements } M)\rangle\)
  have \(\neg \text{literalFalse } u\text{Literal} \ (\text{elements } M)\)
unfolding isUnitClause-def
by simp
{
  assume inconsistent (elements M’)
  with :M’ = M @ [(uLiteral, False)]
  have inconsistent (elements M) ∨ inconsistent [unitLiteral] ∨ (∃
  l. literalTrue l (elements M) ∧ 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) : ¬ 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 @ [(uLiteral, False)]
  shows InvariantUniq M’
proof –
  from (InvariantUniq M)
  have uniq (elements M)
      unfolding InvariantUniq-def
  .
  moreover
  from (isUnitClause uClause uLiteral (elements M))
  have ~ literalTrue uLiteral (elements M)
      unfolding isUnitClause-def
  by simp
  ultimately
  show ?thesis
      using :M’ = M @ [(uLiteral, False)] uniqAppendElement[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' obtain 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
qed

lemma InvariantCFalseAfterUnitPropagate:
  fixes M :: LiteralTrail and F :: Formula and M' :: LiteralTrail and
\( u\text{Clause} :: \text{Clause} \ \text{and} \ u\text{Literal} :: \text{Literal} \)

assumes InvariantCFalse conflictFlag M C and 
\( M' = M @ \{[u\text{Literal}, \text{False}]\} \)

shows InvariantCFalse conflictFlag M' C

proof –
from \( \langle \text{InvariantCFalse conflictFlag M C} \rangle \)
have \( \ast :: \text{conflictFlag} \rightarrow \text{clauseFalse} \ C \ (\text{elements} \ M) \)

unfolding InvariantCFalse-def

\{ 
assume conflictFlag
with \( \langle M' = M @ \{[u\text{Literal}, \text{False}]\} \rangle \ast \)
have clauseFalse C (elements M')
  by (simp add: clauseFalseAppendValuation)
\}

thus \( \ast \text{thesis} \)

unfolding InvariantCFalse-def
by simp

qed

Backtrack transition rule.

lemma InvariantImpliedLiteralsAfterBacktrack:
fixes \( F :: \text{Formula} \ \text{and} \ M :: \text{LiteralTrail} \)

assumes InvariantImpliedLiterals F M and InvariantUniq M and
InvariantConsistent M and 
\( \text{decisions} \ M \neq [] \ \text{and} \ \text{formulaFalse} \ F \ (\text{elements} \ M) \)
\( M' = \langle \text{prefixBeforeLastDecision} \ M \rangle @ \{[\text{opposite} \ (\text{lastDecision} \ M), \text{False}]\} \)

shows InvariantImpliedLiterals F M'

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

\{ 
fix \( l' :: \text{Literal} \)
assume \( l' \ \text{el} \ (\text{elements} \ M') \)
let \( \text{?p} = \langle \text{prefixBeforeLastDecision} \ M \rangle \)
let \( \text{?l} = \text{lastDecision} \ M \)
have formulaEntailsLiteral (\( F @ \text{val2form} \ \langle \text{decisionsTo} \ l' \ M' \rangle \)) \( l' \)
  proof (cases \( l' \ \text{el} \ (\text{elements} \ ?p) \))
  case True
  with \( \langle \text{isPrefix} \ ?p \ M \rangle \)
  have \( l' \ \text{el} \ (\text{elements} \ M) \)
    using prefixElementsAreTrailElements[of \ ?p \ M]
    by auto
  with \( \langle \text{InvariantImpliedLiterals} F M \rangle \)
  have formulaEntailsLiteral (\( F @ \text{val2form} \ \langle \text{decisionsTo} \ l' \ M' \rangle \)) \( l' \)
    unfolding InvariantImpliedLiterals-def
    by simp
\}
moreover
from \( M' = ?p @ [(\text{opposite } ?l, \text{False})] \) True \((\text{isPrefix } ?p M)

have (decisionsTo \( l' M' \)) = (decisionsTo \( l' M \))

using prefixToElementToPrefixElement[of \( ?p M l' \)]

unfolding markedElementsTo-def
by (auto simp add: prefixToElementAppend)

ultimately
show \( ?\text{thesis} \)

by auto

next

case False
with \( l' \in (\text{elements } M') \) and \( M' = ?p @ [(\text{opposite } ?l, \text{False})] \)

have \( ?l = (\text{opposite } l') \)
by (auto split: split-if-asm)

hence \( l' = (\text{opposite } ?l) \)

by simp

from :InvariantUniq \( M \) and \( (\text{markedElements } M \neq []) \)

have (decisionsTo \( ?l M \)) = (decisions \( M \))

unfolding InvariantUniq-def

using markedElementsToLastMarkedAreAllMarkedElements
by auto

moreover
from \( \text{decisions } M \neq [] \),

have \( ?l \in (\text{elements } M) \)

by (simp add: lastMarkedIsMarkedElement markedElementsAreElements)

with \( \text{InvariantConsistent } M \),

have \( \neg (\text{opposite } ?l) \in (\text{elements } M) \)

unfolding InvariantConsistent-def
by (simp add: inconsistentCharacterization)

with \( \text{isPrefix } ?p M \)

have \( \neg (\text{opposite } ?l) \in (\text{elements } ?p) \)

using prefixElementsAreTrailElements[of \( ?p M \)]

by auto

with \( M' = ?p @ [(\text{opposite } ?l, \text{False})] \):

have decisionsTo \( (\text{opposite } ?l) M' = \text{decisions } ?p \)

using markedElementsToAppend [of opposite \( ?l ?p [(\text{opposite } ?l, \text{False})])]\]

unfolding markedElementsTo-def
by simp

moreover
from :InvariantUniq \( M \) \( \text{decisions } M \neq [] \)

have \( \neg ?l \in (\text{elements } ?p) \)

unfolding InvariantUniq-def

using lastMarkedNotInPrefixBeforeLastMarked[of \( M \)]

by simp

hence \( \neg ?l \in (\text{decisions } ?p) \)

by (auto simp add: markedElementsAreElements)
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
$M' = (\text{prefixBeforeLastDecision } M) \circledast [(\text{opposite } (\text{lastDecision } M), False)]$
shows $\text{InvariantConsistent } M'$
proof–
  from $\langle \text{decisions } M \neq [] \rangle \langle \text{InvariantUniq } M \rangle$
  have $\neg \text{lastDecision } M \in \text{elements (prefixBeforeLastDecision } M )$
      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 $\text{isPrefixPrefixBeforeLastMarked[of } M \rangle$
      using $\text{isPrefixElements[of prefixBeforeLastDecision } M \rangle$
      using $\text{consistentPrefix[of elements } (\text{prefixBeforeLastDecision } M)$
      elements $M \rangle$
      by simp
ultimately
show $\neg \text{thesis}$
unfolding $\text{InvariantConsistent-def}$
  using $\langle M' = (\text{prefixBeforeLastDecision } M) \circledast [(\text{opposite } (\text{lastDecision } M), False)] \rangle$
  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
$M' = (\text{prefixBeforeLastDecision } M) \circledast [(\text{opposite } (\text{lastDecision } M), False)]$
shows $\text{InvariantUniq } M'$
proof–
  from $\langle \text{InvariantUniq } M \rangle$
  have $\text{uniq } (\text{elements } (\text{prefixBeforeLastDecision } M))$
      unfolding $\text{InvariantUniq-def}$
      using $\text{isPrefixPrefixBeforeLastMarked[of } M \rangle$
      using $\text{isPrefixElements[of prefixBeforeLastDecision } M \rangle$
      using $\text{uniqListImpliesUniqPrefix}$
      by simp
moreover
from $\langle \text{decisions } M \neq [] \rangle$
  have $\text{lastDecision } M \in (\text{elements } M)$
      using $\text{lastMarkedIsMarkedElement[of } M \rangle$
      using $\text{markedElementsAreElements[of lastDecision } M \rangle$
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

148
ultimately
show thesis
  using assms
  using varsAppendValuation[of elements (prefixBeforeLastDecision M) [opposite (lastDecision M)]]
  unfolding InvariantVarsM-def
  by auto
qed

Backjump transition rule.

lemma InvariantImpliedLiteralsAfterBackjump:
  fixes F::Formula and M::LiteralTrail and p::LiteralTrail and bClause::Clause
  and bLiteral::Literal
  assumes InvariantImpliedLiterals F M and
  isPrefix p M and formulaEntailsClause F bClause and isUnitClause
  bClause bLiteral and
  M' = p @ [(bLiteral, False)]
  shows InvariantImpliedLiterals F M'
proof -
  from ⟨InvariantImpliedLiterals F M⟩ (isPrefix p M)
  have InvariantImpliedLiterals F p
  using InvariantImpliedLiteralsHoldsForPrefix [of F M p]
  by simp

  with assms
  show thesis
    using InvariantImpliedLiteralsAfterUnitPropagate [of F p bClause]
    bLiteral M'
    by simp
qed

lemma InvariantVarsMAfterBackjump:
  fixes F::Formula and M::LiteralTrail and p::LiteralTrail and bClause::Clause
  and bLiteral::Literal
  assumes InvariantVarsM M F0 Vbl and
  isPrefix p M and var bLiteral ∈ vars F0 ∪ Vbl and
  M' = p @ [(bLiteral, False)]
  shows InvariantVarsM M' F0 Vbl
proof -
  from ⟨InvariantVarsM M F0 Vbl⟩
  have vars (elements M) ⊆ vars F0 ∪ Vbl
    unfolding InvariantVarsM-def
  .
  moreover
  from ⟨isPrefix p M⟩
  have vars (elements p) ⊆ vars (elements M)
    using varsPrefixValuation [of elements p elements M]
    by (simp add: isPrefixElements)
  ultimately


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 ?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 \( \text{InvariantConsistent } M \)
  have consistent (\( \text{elements } M \))
    unfolding InvariantConsistent-def
    .
    with \( \text{isPrefix } p \) \( M \)
    have consistent (\( \text{elements } p \))
    using consistentPrefix [of \( \text{elements } p \) \( \text{elements } M \)]
    by (simp add: isPrefixElements)
    
    with assms
    show ?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 \( \text{InvariantUniq } M \)
  have uniq (\( \text{elements } M \))
    unfolding InvariantUniq-def
    .
    with \( \text{isPrefix } p \) \( M \)
    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 \((\text{elements } p)\) and
formulaEntailsClause F bClause and
\(M’ = p @ [(bLiteral, False)]\)
shows InvariantReasonClauses F \(M’\)
proof
from \((\text{InvariantReasonClauses } F \ M) \ (\text{InvariantUniq } M) \ (\text{isPrefix } p \ M)\)
have InvariantReasonClauses F p
by \((\text{rule InvariantReasonClausesHoldsForPrefix})\)
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 \((\text{InvariantImpliedLiterals } F \ M)\)
h ave \(*: \forall l. l \in (\text{elements } M) \rightarrow \text{formulaEntailsLiteral} (F @ val2form (decisionsTo l M)) \ l\)
unfolding InvariantImpliedLiterals-def
.
{
fix literal :: Literal
assume literal \in (\text{elements } M)
with *
have \(\text{formulaEntailsLiteral} (F @ val2form (decisionsTo literal M))\)
literal
}
by simp

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

\textbf{proof}

\begin{itemize}
\item \textbf{have} \ \forall \ \text{clause::Clause.} \ \text{clause el} \ (F @ \text{val2form} \ (\text{decisionsTo literal} \ M)) \ \rightarrow \ \text{clause el} \ (F @ [C] @ \text{val2form} \ (\text{decisionsTo literal} \ M))
\end{itemize}

\textbf{proof}

\begin{itemize}
\item \textbf{fix} \ \text{clause :: Clause}
\item \textbf{have} \ \text{clause el} \ (F @ \text{val2form} \ (\text{decisionsTo literal} \ M)) \ \rightarrow \ \text{clause el} \ (F @ [C] @ \text{val2form} \ (\text{decisionsTo literal} \ M))
\item \textbf{proof}
\item \textbf{assume} \ \text{clause el} \ (F @ \text{val2form} \ (\text{decisionsTo literal} \ M))
\item \textbf{thus} \ \text{clause el} \ (F @ [C] @ \text{val2form} \ (\text{decisionsTo literal} \ M))
\item \textbf{by} \ \text{auto}
\item \textbf{qed}
\end{itemize}

\textbf{thus} \ ?\text{thesis}

\textbf{by} \ \text{auto}

\textbf{qed}

\textbf{with} \ \langle \ \text{formulaEntailsLiteral} \ (F @ \text{val2form} \ (\text{decisionsTo literal} \ M)) \ \text{literal} \rangle

\textbf{show} \ ?\text{thesis}

\textbf{by} \ \langle \ \text{rule} \ \text{formulaEntailsLiteralSubset} \rangle

\textbf{qed}

\textbf{lemma} \ \text{InvariantReasonClausesAfterLearn}:

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

\textbf{assumes} \ \text{InvariantReasonClauses} \ \text{F} \ \text{M and}

\text{formulaEntailsClause} \ \text{F} \ \text{C and}
\text{F'} = \text{F} @ [C]

\textbf{shows} \ \text{InvariantReasonClauses} \ \text{F'} \ \text{M}

\textbf{proof}

\begin{itemize}
\item \textbf{fix} \ \text{literal :: Literal}
\item \textbf{assume} \ \text{literal el elements M \ \land \ \neg \ \text{literal el decisions M}}
\item \textbf{with} \ \langle \ \text{InvariantReasonClauses} \ \text{F} \ \text{M} ; \ \text{obtain} \ \text{clause::Clause} \ \text{where} \ \text{formulaEntailsClause} \ \text{F} \ \text{clause isReason clause literal (elements M)} \rangle
\item \textbf{unfolding} \ \text{InvariantReasonClauses-def}
\item \textbf{by} \ \text{auto}
\item \textbf{from} \ \langle \text{formulaEntailsClause} \ \text{F} \ \text{clause} \rangle \ \langle \text{F'} = \text{F} @ [C] \rangle
\end{itemize}

152
have formulaEntailsClause F’ clause
  by (simp add:formulaEntailsClauseAppend)
with isReason clause literal (elements M)
have ∃ clause, formulaEntailsClause F’ clause ∧ isReason clause literal (elements M)
  by auto
} thus ?thesis
unfolding InvariantReasonClauses-def
by simp
qed

lemma InvariantVarsFAfterLearn:
  fixes F0 :: Formula and F :: Formula and F’ :: Formula and C :: Clause
  assumes InvariantVarsF F F0 Vbl and
  vars C ⊆ (vars F0) ⊔ 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 :: Formula and F :: Formula and F’ :: Formula and C :: Clause
  assumes InvariantEquivalent F0 F and
  formulaEntailsClause F C and
  F’ = F @ [C]
  shows InvariantEquivalent F0 F’
proof–
from ⟨InvariantEquivalent F0 F⟩
have equivalentFormulae F0 F
  unfolding InvariantEquivalent-def
  .
with ⟨formulaEntailsClause F C; ⟨F’ = F @ [C]⟩
have equivalentFormulae F0 ⟨F @ [C]⟩
  unfolding extendEquivalentFormulaWithEntailedClause [of F0 F C]
  by simp
thus ?thesis
unfolding InvariantEquivalent-def
using ⟨F’ = F @ [C]⟩
by simp
qed

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

\textit{Explain} transition rule.

\textbf{lemma} \texttt{InvariantCFalseAfterExplain}: 
\begin{itemize}
  \item\texttt{fixes conflictFlag::bool and M::LiteralTrail and C::Clause and literal :: Literal}
  \item assumes \texttt{InvariantCFalse \ conflictFlag \ M \ C} and 
  opposite literal el C and \texttt{isReason \ reason \ literal (elements M) and } 
  \( C' = \text{resolve} \ C \ \texttt{reason (opposite literal)} \) 
  \item shows \texttt{InvariantCFalse \ conflictFlag \ M \ C'}
\end{itemize}
unfolding \texttt{InvariantCFalse-def} 
\textbf{proof}
\begin{itemize}
  \item assume \texttt{conflictFlag}
  \item with \texttt{InvariantCFalse \ conflictFlag \ M \ C;}
  \item have \texttt{clauseFalse \ C (elements M)} 
  unfolding \texttt{InvariantCFalse-def} 
  by simp 
  \item hence \texttt{clauseFalse (removeAll (opposite literal) C) (elements M)} 
  by (\texttt{simp add: clauseFalseIffAllLiteralsAreFalse})
  \item moreover 
  from \texttt{isReason \ reason \ literal (elements M)}
  \item have \texttt{clauseFalse (removeAll literal reason) (elements M)} 
  unfolding \texttt{isReason-def} 
  by simp 
  \item ultimately 
  show \texttt{clauseFalse \ C' (elements M)}
  using \texttt{C' = resolve \ C \ \texttt{reason (opposite literal)}}
  resolveFalseClauses [of opposite literal C elements M \texttt{reason}] 
  by simp
\end{itemize}
\textit{qed}

\textbf{lemma} \texttt{InvariantCEntailedAfterExplain}: 
\begin{itemize}
  \item\texttt{fixes conflictFlag::bool and M::LiteralTrail and C::Clause and literal :: Literal and reason :: Clause}
  \item assumes \texttt{InvariantCEntailed \ conflictFlag \ F \ C} and 
  \texttt{formulaEntailsClause \ F \ reason and } C' = \texttt{(resolve} \ C \texttt{ reason (opposite l))}
  \item shows \texttt{InvariantCEntailed \ conflictFlag \ F \ C'}
\end{itemize}
unfolding \texttt{InvariantCEntailed-def} 
\textbf{proof}
\begin{itemize}
  \item assume \texttt{conflictFlag}
  \item with \texttt{InvariantCEntailed \ conflictFlag \ F \ C;}
  \item have \texttt{formulaEntailsClause \ F \ C}
  unfolding \texttt{InvariantCEntailed-def}
\end{itemize}
Conflict transition rule.

**Lemma invariantCFalseAfterConflict:**

```ml
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:**

```ml
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 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:**

```ml
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 (InvariantEquivalence F0 F)
  show ?thesis
    unfolding InvariantEquivalence-def
    by (simp add:satisfiableEquivalent)
qed

lemma unsatReportExtensiveExplain:
  fixes F :: Formula and M :: LiteralTrail and F0 :: Formula and 
                   C :: Clause and conflictFlag :: bool
  assumes InvariantEquivalence 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:formulaUnsatIffImpliesEmptyClause)
  with (InvariantEquivalence F0 F)
  show ?thesis
    unfolding InvariantEquivalence-def
    by (simp add:satisfiableEquivalent)
qed

SAT Report

lemma satReport:
  fixes F0 :: Formula and F :: Formula and M::LiteralTrail
assumes \( \text{vars} F_0 \subseteq \text{Vbl} \) and \( \text{InvariantVarsF} F F_0 \text{Vbl} \) and \( \text{InvariantConsistent} M \) and \( \text{InvariantEquivalent} F_0 F \) and
\( \neg \text{formulaFalse} F (\text{elements} M) \) and \( \text{vars} (\text{elements} M) \supseteq \text{Vbl} \)
shows \( \text{model} (\text{elements} M) F_0 \)
proof
from \( \langle \text{InvariantConsistent} M \rangle \)
have consistent (\text{elements} M)
  unfolding \( \text{InvariantConsistent-def} \) .
moreover
from \( \langle \text{InvariantVarsF} F F_0 \text{Vbl} \rangle \)
have \( \text{vars} F \subseteq \text{vars} F_0 \cup \text{Vbl} \)
  unfolding \( \text{InvariantVarsF-def} \) .
with \( \langle \text{vars} F_0 \subseteq \text{Vbl} \rangle \)
have \( \text{vars} F \subseteq \text{Vbl} \)
  by auto
with \( \text{vars} \) (\text{elements} M) \supseteq \text{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 \((\text{simp add:totalValuationForFormulaDefinesItsValue})\)
with \( \langle \neg \text{formulaFalse} F \rangle \) (\text{elements} M)
have \( \text{formulaTrue} F \) (\text{elements} M)
  by simp
ultimately
have \( \text{model} \) (\text{elements} M) F
  by simp
with \( \langle \text{InvariantEquivalent} F_0 F \rangle \)
show \?thesis
  unfolding \( \text{InvariantEquivalent-def} \)
  unfolding \( \text{equivalentFormulae-def} \)
  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 than the level of its last asserted literal

definition
\( \text{isUIP} \ l \ c \ M \ == \\
\langle \text{isLastAssertedLiteral} \ (\text{opposite} l) \ (\text{oppositeLiteralList} c) (\text{elements} M) \rangle \land \\
(\forall \ l'. \ l' \in c \land l' \neq l \rightarrow \text{elementLevel} \ (\text{opposite} l') M < \text{elementLevel} \ (\text{opposite} l) M) \)

157
**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 then levels of all its other literals.

**definition**

\[
\text{isBackjumpLevel} \; \text{level} \; l \; c \; M \equiv \\
\text{isLastAssertedLiteral} \; (\text{opposite} \; l) \; (\text{oppositeLiteralList} \; c)(\text{elements} \; M) \land \\
0 \leq \text{level} \land \text{level} < \text{elementLevel} \; (\text{opposite} \; l) \; M \land \\
(\forall \; l', l' \; \text{el} \; c \land l' \neq l \rightarrow \text{elementLevel} \; (\text{opposite} \; l') \; M \leq \text{level})
\]

**lemma** lastAssertedLiteralHasHighestElementLevel:

*fixes* literal :: Literal and clause :: Clause and M :: LiteralTrail
*assumes* isLastAssertedLiteral literal clause (elements M) and uniq (elements M)
*shows* \( \forall \; l', l' \; \text{el} \; \text{clause} \land l' \; \text{el} \; \text{elements} \; M \rightarrow \text{elementLevel} \; l' \; M \leq \text{elementLevel} \; l \; M \)

*proof* –

\[
\{ \\
\text{fix} \; l' :: \text{Literal} \land \\
\text{assume} \; l' \; \text{el} \; \text{clause} \land l' \; \text{el} \; \text{elements} \; M \land \\
\text{hence} \; \text{elementLevel} \; l' \; M \leq \text{elementLevel} \; l \; M \\
\text{proof} \; (\text{cases} \; l' = \text{literal}) \\
\text{case} \; \text{True} \\
\text{thus} \; \text{thesis} \; \text{by simp} \\
\text{next} \\
\text{case} \; \text{False} \\
\text{from} \; \text{isLastAssertedLiteral} \; \text{literal} \; \text{clause} \; (\text{elements} \; M) \land \\
\text{have} \; \text{literalTrue} \; \text{literal} \; (\text{elements} \; M) \\
\land \; \forall \; l, l \; \text{el} \; \text{clause} \land l \neq \text{literal} \rightarrow \neg \; \text{precedes} \; \text{literal} \; l \; (\text{elements} \; M) \\
\text{by} \; (\text{auto} \; \text{simp add:isLastAssertedLiteral-def}) \\
\text{with} \; (l' \; \text{el} \; \text{clause}) \; \text{False} \\
\text{have} \; \neg \; \text{precedes} \; \text{literal} \; l' \; (\text{elements} \; M) \\
\text{by} \; \text{simp} \\
\text{with} \; \text{False} \; (l' \; \text{el} \; (\text{elements} \; M)) \; (\text{literalTrue} \; \text{literal} \; (\text{elements} \; M)) \\
\text{have} \; \text{precedes} \; l' \; \text{literal} \; (\text{elements} \; M) \\
\text{using} \; \text{precedesTotalOrder} \; [\text{of} \; l' \; \text{elements} \; M \; \text{literal}] \\
\text{by} \; \text{simp} \\
\text{with} \; \text{uniq} \; (\text{elements} \; M) \\
\text{show} \; \text{thesis} \\
\text{using} \; \text{elementLevelPrecedesLeq} \; [\text{of} \; l' \; \text{literal} \; M] \\
\text{by} \; \text{auto} \\
\text{qed} \\
\} \\
\text{thus} \; \text{thesis} \; \text{by simp} \\
\text{qed}
\]
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 and
set C = {l}
shows
  isBackjumpLevel 0 l C M
```

**Proof—**

```plaintext
have ∀ l'. l' ∈ C ∧ l' ≠ l → elementLevel (opposite l') M ≤ 0
  
proof−
  
  { 
    fix l'::Literal
    assume l' ∈ 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)⟩
show ?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
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)))
}
lemma isLastAssertedLiteral (opposite ll) (removeAll (opposite l) (oppositeLiteralList C)) (elements M):
  have elementLevel (opposite l') M <= elementLevel (opposite ll) M
  using lastAssertedLiteralHasHighestElementLevel[of opposite ll removeAll (opposite l) (oppositeLiteralList C) M]
  by auto

  thus ?thesis
  by simp
qed

moreover
from ⟨literalTrue (opposite ll) (elements M)⟩
have elementLevel (opposite ll) M >= 0
by simp

moreover
from ⟨(opposite ll) el (removeAll (opposite l) (oppositeLiteralList C))⟩
have ll el C and ll ≠ l
  using literalElListIffOppositeLiteralElOppositeLiteralList[of ll C]
  by auto
from ⟨isUIP l C M⟩
have ∀ l'. l' el C ∧ l' ≠ l → elementLevel (opposite l') M < elementLevel (opposite l) M
  unfolding isUIP-def
  by simp
  with ⟨ll el C⟩ ⟨ll ≠ b⟩
have elementLevel (opposite ll) M < elementLevel (opposite l) M
  by simp
ultimately
show ?thesis
  using ⟨isLastAssertedLiteral (opposite l) (oppositeLiteralList C) (elements M)⟩
  unfolding isBackjumpLevel-def
  by simp
qed

if UIP is reached then there exists correct backjump level.

lemma isUIPExistsBackjumpLevel:
  fixes M :: LiteralTrail and c :: Clause and l :: Literal
  assumes
    clauseFalse c (elements M) and
    isUIP l c M and
    uniq (elements M) and
    elementLevel (opposite l) M > 0
  shows
    ∃ level. (isBackjumpLevel level l c M)
from \langle \text{isUIP} \ l \ c \ M \rangle

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

  unfolding \text{isUIP-def}
  by simp

show \ ?thesis
proof (cases set \ c = \{\ l\})
  case True
  with (\text{elementLevel} (\text{opposite} \ l) \ M > 0) (\text{isLastAssertedLiteral} (\text{opposite} \ l) (\text{oppositeLiteralList} \ c) (\text{elements} \ M))
  have \text{isBackjumpLevel} 0 \ l \ c \ M
    using \text{backjumpLevelZero[of} \ l \ c \ M]\n    by auto
  thus \ ?thesis
    by auto

next
  case False
  have \ \exists \ \text{literal}. \ \text{isLastAssertedLiteral} \ \text{literal} (\text{removeAll} (\text{opposite} \ l) (\text{oppositeLiteralList} \ c)) (\text{elements} \ M)
  proof-
    let \ ?\ ll = \text{getLastAssertedLiteral} (\text{oppositeLiteralList} (\text{removeAll} \ l \ c)) (\text{elements} \ M)
    from \langle \text{clauseFalse} \ c \ (\text{elements} \ M) \rangle
    have \text{clauseFalse} (\text{removeAll} \ l \ c) (\text{elements} \ M)
      by (simp add: \text{clauseFalseRemove})
    moreover
    have \text{removeAll} \ l \ c \neq \ []
    proof-
      have (set \ c \subseteq \{\ l\} \cup (\text{removeAll} \ l \ c))
        by auto
    from \langle \text{isLastAssertedLiteral} (\text{opposite} \ l) (\text{oppositeLiteralList} \ c) (\text{elements} \ M) \rangle
    have (\text{opposite} \ l) \ \text{el} \ \text{oppositeLiteralList} \ c
      unfolding \text{isLastAssertedLiteral-def}
      by simp
    hence \ l \ \text{el} \ c
      using \text{literalElListIffOppositeLiteralElOppositeLiteralList[of} \ l \ c]\n    by simp
    hence \ l \ \text{in} \ \text{set} \ c
      by simp
    { assume \ \neg \ ?\thesis
      hence set (\text{removeAll} \ l \ c) = \ {}
        by simp
    with (set \ c \subseteq \{\ l\} \cup (\text{removeAll} \ l \ c))
    have set \ c \subseteq \{\ l\} \n
162
by simp
with \( l \in \text{set } c \):
have \( \text{set } c = \{ l \} \)
  by auto
with False
have False
  by simp

} thus ?thesis
  by auto
qed
ultimately
have \( \text{isLastAssertedLiteral } \exists \text{ll (oppositeLiteralList (removeAll l c)) (elements } M \)\)
  using \( \text{uniq (elements } M \)\):
  using \( \text{getLastAssertedLiteralCharacterization } \text{of removeAll l c elements } M \)
  by simp
hence \( \text{isLastAssertedLiteral } \exists \text{ll (removeAll (opposite l) (oppositeLiteralList c)) (elements } M \)\)
  using \( \text{oppositeLiteralListRemove } \text{of l c} \)
  by simp
thus ?thesis
  by auto
qed
then obtain ll::Literal where \( \text{isLastAssertedLiteral } \exists l (\text{removeAll (opposite l) (oppositeLiteralList c)) (elements } M \)
  by auto

with \( \text{uniq (elements } M \)\): \( \text{clauseFalse c (elements } M \)\): \( \text{isUIP l c M} \)
have \( \text{isBackjumpLevel } \text{elementLevel ll M l c M} \)
  using \( \text{backjumpLevelLastLast } \text{of l c M opposite ll} \)
  by auto
thus ?thesis
  by auto
qed
qed

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

lemma isBackjumpLevelEnsuresIsUnitInPrefix:
  fixes \( M :: \text{LiteralTrail and conflictFlag :: bool and c :: Clause and l :: Literal} \)
  assumes consistent (elements } M \) and \( \text{uniq (elements } M \)\) and \( \text{clauseFalse c (elements } M \)\) and \( \text{isBackjumpLevel level l c M} \)
  shows isUnitClause c l (elements (prefixToLevel level M))
proof –
  from \( \text{isBackjumpLevel level l c M} \)
have isLastAssertedLiteral \((\text{opposite } l) (\text{oppositeLiteralList } c) (\text{elements } M)\)
\[0 \leq \text{level} < \text{elementLevel} (\text{opposite } l) M \text{ and}\]
\[\forall l', l' \in c \land l' \neq l \longrightarrow \text{elementLevel} (\text{opposite } l') M \leq \text{level}\]

unfolding isBackjumpLevel-def
by auto

from \(\text{isLastAssertedLiteral} (\text{opposite } l) (\text{oppositeLiteralList } c) (\text{elements } M)\)
have \(l \in c\) literalTrue \((\text{opposite } l) (\text{elements } M)\)
using isLastAssertedCharacterization [of \(l \in c\) elements \(M\)]
by auto

have \(\neg\) literalFalse \(l\) (elements \((\text{prefixToLevel } \text{level } M)\))
using \(\text{level} < \text{elementLevel} (\text{opposite } l) M \land 0 \leq \text{level}\) (uniq \(\text{elements } M\))
by (simp add: literalNotInEarlierLevelsThanItsLevel)
moreover
have \(\neg\) literalTrue \(l\) (elements \((\text{prefixToLevel } \text{level } M)\))
proof –
from \(\text{consistent } (\text{elements } M)\) \(\neg\) literalTrue \((\text{opposite } l) (\text{elements } M)\)
have \(\neg\) literalFalse \((\text{opposite } l) (\text{elements } M)\)
by (auto simp add: inconsistentCharacterization)
thus ?thesis
using isPrefixPrefixToLevel\[\text{of } \text{level } M\]
prefixElementsAreTrailElements\[\text{of } \text{prefixToLevel } \text{level } M M\]
unfolding prefixToLevel-def
by auto

qed
moreover
have \(\forall l', l' \in c \land l' \neq l \longrightarrow \text{literalFalse} l' (\text{elements } (\text{prefixToLevel } \text{level } M))\)
proof –
\{
fix \(l'::\text{Literal}\)
assume \(l' \in c l' \neq l\)
from \((l' \in c)\) \(\text{clauseFalse } c\) (elements \(M\))
have literalFalse \(l'\) (elements \(M\))
by (simp add: clauseFalseIffAllLiteralsAreFalse)

have literalFalse \(l'\) (elements \((\text{prefixToLevel } \text{level } M)\))
proof –
from \((l' \in c)\) \(l' \neq l\)
have elementLevel \((\text{opposite } l') M \leq \text{level}\)
using *
by auto
thus \( ?\text{thesis} \)
\hspace{1em} using \((\text{literalFalse} \ l') (\text{elements} \ M)\)
\hspace{1em} \((0 <= \text{level})\)
\hspace{1em} \(\text{elementLevelLtLevelImpliesMemberPrefixToLevel}[\text{of opposite} \ l']\)
\hspace{1em} \(M \ \text{level}\)
\hspace{1em} by \(\text{simp}\)

\hspace{1em} \(\text{qed}\)
\hspace{1em} \text{thus} \ ?\text{thesis}
\hspace{1em} by \(\text{auto}\)

\hspace{1em} \(\text{qed}\)
\hspace{1em} \text{ultimately}
\hspace{1em} \text{show} \ ?\text{thesis}
\hspace{1em} using \((\ l \ e \ c)\)
\hspace{1em} unfolding \(\text{isUnitClause-def}\)
\hspace{1em} by \(\text{simp}\)

\hspace{1em} \(\text{qed}\)

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.

definition
\(\text{isMinimalBackjumpLevel} \ \text{level} \ l \ c \ M \ ==\)
\(\text{isBackjumpLevel} \ \text{level} \ l \ c \ M \ \&\)
\((\text{if set} \ c \ \neq \ {l}) \ \text{then}\)
\((\exists \ ll. \ ll \ e \ c \ \& \ \text{elementLevel} (\text{opposite} \ ll) \ M = \text{level})\)
\((\text{else})\)
\(\text{level} = 0\)

lemma \(\text{isMinimalBackjumpLevelCharacterization}:\)
assumes
\(\text{isUIP} \ l \ c \ M\)
\(\text{clauseFalse} \ c \ (\text{elements} \ M)\)
\(\text{uniq} \ (\text{elements} \ M)\)
shows
\(\text{isMinimalBackjumpLevel} \ \text{level} \ l \ c \ M =\)
\(\text{(isBackjumpLevel} \ \text{level} \ l \ c \ M \ \&\)
\((\forall \ \text{level'} \ . \ \text{level'} < \text{level} \ \rightarrow \ \neg \text{isBackjumpLevel} \ \text{level'} \ l \ c \ M)\)) \(\text{(is}\)
\(\text{lhs} = \text{rhs})\)
proof
assume \(\text{lhs}\)
show \(\text{rhs}\)
proof \((\text{cases} \ set \ c = \ {l})\)
case \(\text{True}\)
thus \(\text{thesis}\)
using \(\text{lhs}\)
unfolding \(\text{isMinimalBackjumpLevel-def}\)
by \(\text{auto}\)
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 \neq 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) \lneq 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 \neq []
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

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 \(\neg (\exists \ l'. \ isUnitClause c \ l' \ (elements \ (prefixToLevel \ level' \ M)))\)
  proof
  from ⟨isMinimalBackjumpLevel level l c M⟩
  have isUnitClause c l (elements (prefixToLevel level M))
    using assms
    unfolding isBackjumpLevelEnsuresIsUnitInPrefix[of M c level l]
    by simp
  hence \(\neg \ literalFalse l \ (elements \ (prefixToLevel \ level M))\)
    unfolding isUnitClause-def
    by auto
  hence \(\neg \ literalFalse l \ (elements M) \lor \ elementLevel \ (opposite \ l) \ M > level\)
    using elementLevelLtLevelImpliesMemberPrefixToLevel[of \ M \ level]
    using elementLevelLtLevelImpliesMemberPrefixToLevel[of opposite l M level]
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 level]
    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 \((l \neq ll \rightarrow ll el c)\)
  using \((\neg \text{literalFalse} ll \rightarrow \text{elements} (\text{prefixToLevel} level' M))\)
  using \((\text{isUnitClause} c l' \rightarrow \text{elements} (\text{prefixToLevel} level' M))\)
  unfolding isUnitClause-def
  by auto
next
case False
have \(l el c\)
  using \((\text{isMinimalBackjumpLevel} level l c M)\)
  unfolding isMinimalBackjumpLevel-def
  unfolding isBackjumpLevel-def
  unfolding isLastAssertedLiteral-def
  using literalElListIffOppositeLiteralElOppositeLiteralList[of \(l\) \(c\)]
  by simp
thus \(?thesis\)
  using False
  using \((\neg \text{literalFalse} l \rightarrow \text{elements} (\text{prefixToLevel} level' M))\)
  using \((\text{isUnitClause} c l' \rightarrow \text{elements} (\text{prefixToLevel} level' M))\)
  unfolding isUnitClause-def
  by auto
qed

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

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

lemma allDecisionsThenUIP:
  fixes \(M :: \text{LiteralTrail}\) and \(c :: \text{Clause}\)
  assumes \((\text{uniq} (\text{elements} M))\) and
  \(\forall l', l' el c \rightarrow (\text{opposite} l') el (\text{decisions} M)\)
  \(\text{isLastAssertedLiteral} \ (\text{opposite} l) \ (\text{oppositeLiteralList} c \ (\text{elements} M))\)
  shows isUIP \(l c M\)
proof
from `isLastAssertedLiteral (opposite l) (oppositeLiteralList c) (elements M)`
  have \( l \in c \) (opposite l) el (elements M)
  and \(*: \forall l'. l' el (oppositeLiteralList c) \land l' \neq l \rightarrow \neg precedes (opposite l) l' (elements M)\)
  unfolding isLastAssertedLiteral_def
  using literalElListIffOppositeLiteralElOppositeLiteralList
  by auto
with \( \forall l', l' el c \rightarrow (opposite l') el (decisions M)\):
  have (opposite l) el (decisions M)
  by simp
{  
  fix \( l' :: \) Literal
  assume \( l' el c l' \neq l \)
  hence (opposite l') el (oppositeLiteralList c) and (opposite l') \neq opposite l
  using literalElListIffOppositeLiteralElOppositeLiteralList[of l' c]
  by auto
  with *
  have \( \neg precedes (opposite l) (opposite l') (elements M)\)
  by simp
from \( l' el c \land l el c \rightarrow (opposite l) el (decisions M)\):
  have (opposite l') el (decisions M)
  by auto
  hence (opposite l') el (elements M)
  by (simp add:markedElementsAreElements)
from \( (opposite l) el (elements M) \land (opposite l') el (elements M) \land l' \neq l \)
  (\( \neg precedes (opposite l) (opposite l') (elements M)\)
  have precedes (opposite l') (opposite l) (elements M)
  using precedesTotalOrder[of opposite l elements M opposite l']
  by simp
  with \( \forall uniq (elements M)\):
  have elementLevel (opposite l') M \leq elementLevel (opposite l) M
  by (auto simp add:elementLevelPrecedesLeq)
moreover
from \( \forall uniq (elements M)\):
  \( (opposite l) el (decisions M) \land (opposite l') el (decisions M) \land l' \neq l \)
  have elementLevel (opposite l) M \neq 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 \textit{thesis}

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

\textbf{unfolding}\quad \text{isUIP-def}

by simp

\textbf{qed}

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

\textbf{lemma}\quad \textit{lastDecisionThenUIP}: \hspace{1cm}

\textbf{fixes}\quad M :: \text{LiteralTrail and } c :: \text{Clause}

\textbf{assumes}\quad (uniq (elements } M \text{)) \quad \text{and} \quad (\text{opposite } l \text{) el (decisions } M \text{)}

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

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

\text{shows}\quad \text{isUIP } l \text{ c } M

\text{proof} –

\textbf{from}\quad \text{isLastAssertedLiteral (opposite } l \text{) (oppositeLiteralList } c \text{) (elements } M \text{)}

\textbf{have}\quad l \text{ el c (opposite } l \text{) el (elements } M \text{)}

\text{and}\quad \ast: \forall l'. \text{ l' el (oppositeLiteralList } c \text{)} \land l' \neq \text{ opposite } l \rightarrow \neg \text{precedes (opposite } l \text{)} \text{ l' (elements } M \text{)}

\textbf{unfolding}\quad \text{isLastAssertedLiteral-def}

\textbf{using}\quad \text{literalElListIffOppositeLiteralElOppositeLiteralList}

by auto

\{
\textbf{fix}\quad l' :: \text{Literal}

\textbf{assume}\quad l' \text{ el c l' } \neq l

\textbf{hence}\quad \text{opposite } l' \text{ el (oppositeLiteralList } c \text{) and opposite } l' \neq \text{ opposite } l

\textbf{using}\quad \text{literalElListIffOppositeLiteralElOppositeLiteralList[of l' c]}

by auto

\textbf{with}\quad \ast

\textbf{have}\quad \neg \text{precedes (opposite } l \text{) (opposite } l' \text{) (elements } M \text{)}

by simp

\textbf{have}\quad (\text{opposite } l') \text{ el (elements } M \text{)}

\textbf{using}\quad (l' \text{ el c) (clauseFalse } c \text{ (elements } M \text{)})

by (simp add: clauseFalseIffAllLiteralsAreFalse)

\textbf{from}\quad (\text{opposite } l \text{) el (elements } M \text{)}\rightarrow (\text{opposite } l' \text{) el (elements } M \text{)}

\langle l' \neq l \rangle

\langle \neg \text{precedes (opposite } l \text{) (opposite } l' \text{) (elements } M \text{)}

\textbf{have}\quad \text{precedes (opposite } l' \text{) (opposite } l \text{) (elements } M \text{)}

\textbf{using}\quad \text{precedesTotalOrder [of opposite } l \text{ elements } M \text{ opposite } l' \rangle

by simp

172
hence \( \text{elementLevel}(\text{opposite } l') M < \text{elementLevel}(\text{opposite } l) M \)

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

opposite \( l \)

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

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

using \( l' \neq l \)

by simp

\}

thus \( ?\text{thesis} \)

using \( \langle \text{isLastAssertedLiteral}(\text{opposite } l) \text{ (oppositeLiteralList } c) \rangle \)

(\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.

lemma \( \text{allDecisionsThenExistsBackjumpLevel}: \)

fixes \( M :: \text{LiteralTrail} \) and \( c :: \text{Clause} \)

assumes \( \langle \text{uniq}(\text{elements } M) \rangle \) and

\( \forall l', l' \text{ el } c \rightarrow (\text{opposite } l') \text{ el } (\text{decisions } M) \)

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

shows \( \exists \text{ level}. \ (\text{isBackjumpLevel level } l \ c M) \)

proof --

from \( \text{assms} \)

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

using \( \text{allDecisionsThenUIP} \)

by simp

moreover

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

have \( l \text{ el } c \)

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

using \( \text{literalElListIffOppositeLiteralElOppositeLiteralList} \)

by simp

with \( \forall l', l' \text{ el } c \rightarrow (\text{opposite } l') \text{ el } (\text{decisions } M) \)

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

by simp

hence \( \text{elementLevel}(\text{opposite } l) M > 0 \)

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

\( \text{elementLevelMarkedGeq1}[\text{of } M \text{ opposite } l] \)

by auto

moreover

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

proof --

{ 

fix \( l'::\text{Literal} \)

assume \( l' \text{ el } c \)
with \( \forall l', l \to (\text{opposite } l') \to (\text{decisions } M) \)
have \((\text{opposite } l') \to (\text{decisions } M)\)
by simp
hence \(\text{literalFalse } l' \to (\text{elements } M)\)
using \(\text{markedElementsAreElements}\)
by simp
}
thus \(?\text{thesis}\)
using \(\text{clauseFalseIffAllLiteralsAreFalse}\)
by simp
qed
ultimately
show \(?\text{thesis}\)
using \(\text{uniq } (\text{elements } M)\)
using \(\text{isUIPExistsBackjumpLevel}\)
by simp
qed

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

\textbf{lemma} \textit{explainApplicableToEachNonDecision}:

\textbf{fixes} \(F :: \text{Formula}\) \textbf{and} \(M :: \text{LiteralTrail}\) \textbf{and} \(\text{conflictFlag} :: \text{bool}\)
\textbf{and} \(C :: \text{Clause}\) \textbf{and} \(\text{literal} :: \text{Literal}\)
\textbf{assumes} \(\text{InvariantReasonClauses } F \text{ and } \text{InvariantCFalse } \text{conflictFlag } M \text{ and } C\)
\text{conflictFlag} = \text{True} \text{ and } (\text{opposite literal el } C \text{ and } \neg \text{literal el } (\text{decisions } M))
\textbf{shows} \(\exists \text{ clause. } \text{formulaEntailsClause } F \text{ clause } \land \text{isReason clause literal } (\text{elements } M)\)
\textbf{proof}–
from \((\text{conflictFlag} = \text{True}) \langle \text{InvariantCFalse } \text{conflictFlag } M \text{ and } C \rangle\)
have \(\text{clauseFalse } C \langle \text{elements } M \rangle\)
unfolding \(\text{InvariantCFalse-def}\)
by simp
with \(\text{opposite literal el } C\)
have \(\text{literalTrue } \text{literal } (\text{elements } M)\)
by \((\text{auto simp add:clauseFalseIffAllLiteralsAreFalse})\)
with \(\neg \text{literal el } (\text{decisions } M) :: \langle \text{InvariantReasonClauses } F \text{ and } M \rangle\)
show \(?\text{thesis}\)
unfolding \(\text{InvariantReasonClauses-def}\)
by \text{auto}
qed

\section{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 `∀ x. (x, x) ∉ r`
- **Shows** `acyclic r`

**Proof** (by `acyclicI`)

- Assume `∃ x. (x, x) ∈ r^+`
- Then obtain `x` where `(x, x) ∈ r^+`
  - by `auto`
- Moreover
  - from `(trans r)`
  - have `r^+ = r`
    - by `(rule trancl-id)`
- Ultimately
  - have `(x, x) ∈ r`
    - by `simp`
  - with `∀ x. (x, x) ∉ r`
    - have `False`
      - by `simp`
- Thus `∀ x. (x, x) ∉ r^+`
  - by `auto`

**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`:

- `decisionLess = {(l1::('a*bool), l2::('a*bool)). isDecision l1 ∧ ¬ isDecision l2}`

**Definition** `lexLess`:

- `lexLess = {(M1::'a Trail, M2::'a Trail). (M2, M1) ∈ 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 ≠ []`
- **Shows** `(a @ b, a) ∈ lexLess`

**Proof**

- From `(b ≠ [])`
- Have `∃ aa list. b = aa ≠ list`
  - by `(simp add: neq-Nil-conv)`
- Then obtain `aa::'a × bool and list :: 'a Trail`
where $b = aa \# \text{list}$

by auto

thus $\exists \text{thesis}$

unfolding $\text{lexLess-def}$

unfolding $\text{lexord-def}$

by simp

qed

lemma $\text{lexLessBackjump}$:

assumes $p = \text{prefixToLevel level a and level \geq 0 and level < currentLevel a}$

shows $(p \oplus [(x, \text{False})], a) \in \text{lexLess}$

proof-

from $\text{assms}$

have $\exists \text{rest. prefixToLevel level a \oplus rest = a \land rest \neq [] \land isDecision (hd \text{rest})}$

using $\text{isProperPrefixPrefixToLevel}$

by auto

with $(p = \text{prefixToLevel level a})$

obtain $\text{rest}$

where $p \oplus \text{rest} = a \land \text{rest} \neq [] \land \text{isDecision (hd \text{rest})}$

by auto

thus $\exists \text{thesis}$

unfolding $\text{lexLess-def}$

using $\text{lexord-append-left-rightI[of hd \text{rest} (x, False) decisionLess p tl \text{rest} []]}$

unfolding $\text{decisionLess-def}$

by simp

qed

lemma $\text{lexLessBacktrack}$:

assumes $p = \text{prefixBeforeLastDecision a decisions a \neq []}$

shows $(p \oplus [(x, \text{False})], a) \in \text{lexLess}$

using $\text{assms}$

using $\text{prefixBeforeLastMarkedIsPrefixBeforeLastLevel[of a]}$

using $\text{lexLessBackjump[of p currentLevel a - 1 a]}$

unfolding $\text{currentLevel-def}$

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 $\text{transDecisionLess}$:

shows $\text{trans decisionLess}$

proof-

{ fix $x::('a\text{bool})$ and $y::('a\text{bool})$ and $z::('a\text{bool})$

assume $(x, y) \in \text{decisionLess}$

hence $\neg \text{isDecision y}$

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

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

\textbf{lemma} \textit{irreflexiveDecisionLess}: \\
\textbf{shows} \((x, x) \not\in \text{decisionLess}\) \\
unfolding \textit{decisionLess-def} \\
by simp

\textbf{lemma} \textit{irreflexiveLexLess}: \\
\textbf{shows} \((x, x) \not\in \text{lexLess}\) \\
using \textit{lexord-irreflexive[of decisionLess x] irreflexiveDecisionLess} \\
unfolding \textit{lexLess-def} \\
by \textit{auto}

\textbf{lemma} \textit{acyclicLexLess}: \\
\textbf{shows} \(\text{acyclic lexLess}\) \\
\textbf{proof} \textit{(rule transIrreflexiveOrderingIsAcyclic)}
show \textit{trans \(\text{lexLess}\)}
using \textit{trans\(\text{lexLess}\)}.

\begin{verbatim}
show \(\forall \ x. \ (x, x) \notin \text{lexLess}\)
using \textit{irreflexive\(\text{lexLess}\)}
by \textit{auto}
\end{verbatim}

\textbf{qed}

The \(\text{lexLess}\) ordering is not well-founded. In order to get a well-founded ordering, we restrict the \(\text{lexLess}\) ordering to consistent and uniq trails with fixed variable set.

\begin{verbatim}
definition \textit{lexLessRestricted} \((\text{Vbl}::\text{Variable set})\) \(\equiv\) \{\((M1, M2)\).
\(\text{vars} (\text{elements} M1) \subseteq \text{Vbl} \land \text{consistent} (\text{elements} M1) \land \text{uniq} (\text{elements} M1) \land \text{vars} (\text{elements} M2) \subseteq \text{Vbl} \land \text{consistent} (\text{elements} M2) \land \text{uniq} (\text{elements} M2) \land \((M1, M2) \in \text{lexLess})\} \end{verbatim}

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

\textbf{lemma} \textit{finiteVarsClause}:
\begin{verbatim}
fixes \(c::\text{Clause}\)
shows \(\text{finite (vars} c)\)
by \(\text{(induct} c)\) \text{auto} 
\end{verbatim}

\textbf{lemma} \textit{finiteVarsFormula}:
\begin{verbatim}
fixes \(F::\text{Formula}\)
shows \(\text{finite (vars} F)\)
\textbf{proof} \(\text{(induct} F)\)
\textbf{case} \((\text{Cons} \ c \ F)\)
thus \(\text{?case}\)
\textbf{using} \textit{finiteVarsClause[of} \ c\]\textbf{by simp}\n\textbf{qed simp}\n\end{verbatim}

\textbf{lemma} \textit{finiteListDecompose}:
\begin{verbatim}
shows \(\text{finite \{a, b\}. \ l = a \# b\}\}
\textbf{proof} \(\text{(induct} l)\)
\textbf{case} \(\text{Nil}\)
thus \(\text{?case}\)
\textbf{by simp}\n\textbf{next}\n\textbf{case} \((\text{Cons} \ x \ l')\)
thus \(\text{?case}\)
\textbf{proof}\n\textbf{let} \(\textit{S} \ l = \{(a, b). \ l = a \# b\}\)
\textbf{let} \(\textit{S}' \ x \ l' = \{(a', b). \ a' = \[] \land b = (x \# l') \lor \ldots\}
\textbf{have} \(\textit{S} (x \# l') = \textit{S}' \ x \ l'\)
\textbf{proof} \\
\end{verbatim}

178
show \( ?S (x \# l') \subseteq ?S' x l' \)
proof
fix \( k \)
assume \( k \in ?S (x \# l') \)
then obtain \( a \) and \( b \)
where \( k = (a, b) x \# l' = a \oplus b \)
by auto
then obtain \( a' \) where \( a' = x \# a \)
by auto
from \( \langle k = (a, b), (x \# l' = a \oplus b) \rangle \)
show \( k \in ?S' x l' \)
using SimpleLevi[of \( a \) \( b \) \( x \) \( l' \)]
by auto
qed
next
show \( ?S' x l' \subseteq ?S (x \# l') \)
proof
fix \( k \)
assume \( k \in ?S' x l' \)
then obtain \( a' \) and \( b \) where
\[
\begin{align*}
& k = (a', b) a' = [] \land b = x \# l' \lor (\exists a . a' = x \# a \land (a, b) \in ?S l') \\
& \text{by } auto
\end{align*}
\]
moreover
\{
assume \( a' = [] b = x \# l' \)
with \( \langle k = (a', b) \rangle \)
have \( k \in ?S (x \# l') \)
by simp
\}
moreover
\{
assume \( \exists a . a' = x \# a \land (a, b) \in ?S l' \)
then obtain \( a \) where
\[
\begin{align*}
& a' = x \# a \land (a, b) \in ?S l' \\
& \text{by } auto
\end{align*}
\]
with \( \langle k = (a', b) \rangle \)
have \( k \in ?S (x \# l') \)
by auto
\}
ultimately
show \( k \in ?S (x \# l') \)
by auto
qed
qed
moreover
have \( ?S' x l' = \\
\{ (a', b) . a' = [] \land b = x \# l' \} \cup \{ (a', b) . \exists a . a' = x \# a \land (a, b) \in ?S l' \} \)
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' \} = \exists h ' \{ (a, b). l' = a \oplus 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
qed

lemma finiteListDecomposeSet:
fixes L :: 'a list set 
assumes finite L 
shows finite \{ (a, b). \exists l. l \in L \land l = a \oplus b \} 
proof-
  have \{ (a, b). \exists l. l \in L \land l = a \oplus b \} = ( \bigcup l \in L. \{ (a, b). l = a \oplus b \}) 
  by auto
moreover
have finite ( \bigcup l \in L. \{ (a, b). l = a \oplus 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 \oplus b \}
  by (rule finiteListDecompose)
qed
ultimately
show ?thesis 
by simp
qed

lemma finiteUniqAndConsistentTrailsWithGivenVariableSet:
fixes V :: Variable set 
assumes finite V
shows finite \{\{M::LiteralTrail\}. vars (elements M) = V \land uniq (elements M) \land consistent (elements M)\}
(is finite (?trails V))

using assms
proof induct
  case empty
  thus ?case
  proof –
    have ?trails \{\} = \{M. M = []\} (is ?lhs = ?rhs)
    proof
      show ?lhs \subseteq ?rhs
      proof
        fix M::LiteralTrail
        assume M \in ?lhs
        hence M = []
          by (induct M) auto
        thus M \in ?rhs
          by simp
        qed
      qed
    next
    show ?rhs \subseteq ?lhs
    proof
      fix M::LiteralTrail
      assume M \in ?rhs
      hence M = []
        by simp
      thus M \in ?lhs
        by (induct M) auto
      qed
    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::LiteralTrail). \exists M’ l d M”.
      M = M’ @ [(l, d)] @ M” \land
      M’ @ M” \in (?trails V’) \land
      l \in \{Pos v, Neg v\} \land
      d \in \{True, False\}\}
    have ?trails (insert v V’) = ?trails’ V’
      (is ?lhs = ?rhs)
    proof
show \( \text{lhs} \subseteq \text{rhs} \)

proof
  fix \( M::\text{LiteralTrail} \)
  assume \( M \in \text{lhs} \)
  hence vars (elements \( M \)) = insert \( v \) \( V' \) uniq (elements \( M \))
  consistent (elements \( M \))
       by auto
  hence \( v \in \text{vars} (\text{elements} \ M) \)
       by simp
  hence \( \exists \ l. \ l \in \text{elements} \ M \land \text{var} \ l = v \)
       by (induct \( M \)) auto
  then obtain \( l \) where \( l \in \text{elements} \ M \) \text{var} \( l = v \)
       by auto
  hence \( \exists \ M' \ M'' \ d. \ M = M' \circ \[(l, d)\] \circ M'' \)
proof (induct \( M \))
  case (Cons \( m \) \( M1 \))
  thus \(?\)case
  proof (cases \( l = (\text{element} \ m) \))
    case True
    then obtain \( d \) where \( m = (l, d) \)
           using eitherMarkedOrNotMarkedElement[of \( m \)]
           by auto
    hence \( m \# M1 = [] \circ [(l, d)] \circ M1 \)
           by simp
    then obtain \( M' \ M'' \ d \) \text{where} \( m \# M1 = M' \circ [(l, d)] \circ M'' \)
M''
      ...
    thus \(?\)thesis
           by auto
next
  case False
  with \( (l \in \text{elements} \ (m \# M1)) \)
  have \( l \in \text{elements} \ M1 \)
           by simp
  with Cons(1) \( \text{var} \ l = v \)
  obtain \( M1' \ M'' \ d \) \text{where} \( M1 = M1' \circ [(l, d)] \circ M'' \)
           by auto
  hence \( m \# M1 = (m \# M1') \circ [(l, d)] \circ M'' \)
           by simp
  then obtain \( M' \ M'' \ d \) \text{where} \( m \# M1 = M' \circ [(l, d)] \circ M'' \)
M''
      ...
    thus \(?\)thesis
           by auto
  qed
  qed simp
  then obtain \( M' \ M'' \ d \) \text{where} \( M = M' \circ [(l, d)] \circ M'' \)
           by auto
  moreover
from ⟨\text{\texttt{var} } l = v⟩

have \(l : \{\text{Pos } v, \text{Neg } v\}\)
  by (cases \(l\)) auto

moreover
have \(*: \text{\texttt{vars}} \ \text{(elements } (M' @ M'')) = \text{\texttt{vars}} \ \text{(elements } M') \cup \text{\texttt{vars}} \ \text{(elements } M'')\)
  using \text{\texttt{varsAppendClauses}}[\text{\texttt{of elements } M' \text{ elements } M''}]
  by simp
from \(M = M' @ [(l, d)] @ M''\) (\text{\texttt{var} } l = v)

have \(**: \text{\texttt{vars}} \ \text{(elements } M) = \text{\texttt{vars}} \ \text{(elements } M') \cup \{v\} \cup \text{\texttt{vars}} \ \text{(elements } M'')\)
  using \text{\texttt{varsAppendClauses}}[\text{\texttt{of elements } M' \text{ elements } ([\text{\texttt{l, d}]} \ @ \text{\texttt{M''}})}]
  using \text{\texttt{varsAppendClauses}}[\text{\texttt{of elements } [(l, d)] \text{ elements } M'']}]
  by simp

have \(***: \text{\texttt{vars}} \ \text{(elements } M) = \text{\texttt{vars}} \ \text{(elements } (M' @ M'')) \cup \{v\}\)
  using \(* *\)
  by simp

have \(M' @ M'' \in (\text{\texttt{?trails } V})\)

proof–

from \(\text{\texttt{uniq}} \ \text{(elements } M)\) \(\text{\texttt{\{M = M' @ [(l, d)] @ M''\}}}\)

have \(\text{\texttt{uniq}} \ \text{(elements } (M' @ M''))\)
  by (auto iff: \text{\texttt{uniqAppendIf}})

moreover
have \(\text{\texttt{consistent}} \ \text{(elements } (M' @ M''))\)

proof–

\{\n  assume \(~\text{\texttt{consistent}} \ \text{(elements } (M' @ M''))\)

  then obtain \(l'\) \text{\texttt{where}} \text{\texttt{literalTrue}} \ l' \ (\text{\texttt{elements } (M' @ M''))}

  literalFalse \ l' \ (\text{\texttt{elements } (M' @ M''))}

  by (auto simp add: \text{\texttt{inconsistentCharacterization}})

  with \(M = M' @ [(l, d)] @ M''\)

  have \text{\texttt{literalTrue}} \ l' \ (\text{\texttt{elements } M}) \text{\texttt{literalFalse}} \ l' \ (\text{\texttt{elements } M})

  by auto

  hence \(~\text{\texttt{consistent}} \ \text{(elements } M)\)

  by (auto simp add: \text{\texttt{inconsistentCharacterization}})

  with \(\text{\texttt{consistent}} \ \text{(elements } M)\)

  have \text{\texttt{False}}

  by simp

\}

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

qed

moreover

have \(v \notin \text{\texttt{vars}} \ \text{(elements } (M' @ M''))\)

proof–

\{
assume \( v \in \text{vars}(\text{elements}(M' \oplus M'')) \)

with *

have \( v \in \text{vars}(M') \lor v \in \text{vars}(M'') \)

by simp

moreover

\{
\begin{align*}
\text{assume } & v \in (\text{vars}(M')) \\
\text{hence } & \exists \ l. \ \text{var } = v \land l \in \text{elements } M' \\
& \text{by (induct } M' \text{ auto)} \\
\text{then obtain } & l' \text{ where } \text{var } l' = v \land l' \in \text{elements } M' \\
& \text{by auto} \\
\text{from } & (\text{var } l = v) \land (\text{var } l' = v) \\
\text{have } & l = l' \lor \text{opposite } l = l' \\
& \text{using literalsWithSameVariableAreEqualOrOpposite[of } l l'] \\
& \text{by simp} \\
\text{moreover}
\end{align*}
\}

\{
\begin{align*}
\text{assume } & l = l' \\
\text{with } & (l' \in \text{elements } M') \land (M = M' \oplus [(l, d)] \oplus M'') \\
\text{have } & \neg \text{uniq}(\text{elements } M) \\
& \text{by (auto iff: uniqAppendIff)} \\
\text{with } & (\text{uniq } \text{elements } M) \\
\text{have } & \text{False} \\
& \text{by simp} \\
\text{moreover}
\end{align*}
\}

moreover

\{
\begin{align*}
\text{assume } & \text{opposite } l = l' \\
\text{have } & \neg \text{consistent } (\text{elements } M) \\
\text{proof–} \\
\text{from } & (l' \in \text{elements } M') \land (M = M' \oplus [(l, d)] \oplus M'') \\
\text{have } & \text{literalTrue } l' (\text{elements } M) \\
& \text{by simp} \\
\text{moreover}
\end{align*}
\}

\{
\begin{align*}
\text{from } & l' \in \text{elements } M' \land \text{opposite } l = l' \land (M = M' \oplus [\{(l, d)\}] \oplus M'') \\
\text{have } & \text{literalFalse } l' (\text{elements } M) \\
& \text{by simp} \\
\text{ultimately} \\
\text{show } & \text{thesis} \\
& \text{by (auto simp add: inconsistentCharacterization)} \\
\text{qed} \\
\text{with } & (\text{consistent } \text{elements } M) \\
\text{have } & \text{False} \\
& \text{by simp} \\
\text{ultimately}
\end{align*}
\}

have \text{False}
by auto

moreover
{
  assume \( v \in (\text{vars } (\text{elements } M'')) \)
  hence \( \exists \ l . \ \text{var } l = v \land l \text{ el elements } M'' \)
    by (induct \( M'' \)) auto
  then obtain \( l' \) where var \( l' = v \ l' \text{ el } (\text{elements } M'') \)
    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 \( \langle l' \text{ el elements } M'' \rangle \ \langle M = M' @ [(l, d)] @ M'' \rangle \)
    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 \( \langle l' \text{ el elements } M'' \rangle \ \langle M = M' @ [(l, d)] @ M'' \rangle \)
      have literalTrue \( l' \) (\( \text{elements } M \))
        by simp
      moreover
      from \( \langle l' \text{ el elements } M'' \rangle \ \langle \text{opposite } l = l' \rangle \ \langle M = M' \)
        @ [(l, d)] @ M'' \)
      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
  \[ v \notin \text{vars}(\text{elements}(M' \@ M'')) \]
  \[ \text{vars}(\text{elements } M) = \text{insert } v \ V' \]
  \[ \lnot v \in V' \]
  have \[ \text{vars}(\text{elements}(M' \@ M'')) = V' \]
    by (auto simp del: \text{vars-def-clause})
ultimately
  show ?thesis
    by simp
qed

ultimately
  show \( M \in \text{?rhs} \)
    by auto
qed

next
  show \( \text{?rhs} \subseteq \text{?lhs} \)
  proof
    fix \( M :: \text{LiteralTrail} \)
    assume \( M \in \text{?rhs} \)
    then obtain \( M' \ M'' \ l \ d \)
      where
        \( M = M' \@ ([l, d]) \@ M'' \)
        \[ \text{vars}(\text{elements }(M' \@ M'')) = V' \]
        uniq (\text{elements } (M' \@ M'')) consistent \(\text{elements } (M' \@ M'')\)
        \( l \in \{ \text{Pos } v, \text{Neg } v \} \)
        by auto
    from \( l \in \{ \text{Pos } v, \text{Neg } v \} \)
    have \( \text{var } l = v \)
      by auto
    have \( \ast: \text{vars}(\text{elements } (M' \@ M'')) = \text{vars}(\text{elements } M') \cup \text{vars}(\text{elements } M'') \)
      using \text{vars-append-clauses}[\text{of elements } M' \text{ elements } M'']
      by simp
    from \( \text{var } l = v \) \( M = M' \@ ([l, d]) \@ M'' \)
    have \( \ast\ast: \text{vars}(\text{elements } M) = \text{vars}(\text{elements } M') \cup \{ v \} \cup \text{vars}(\text{elements } M'') \)
      using \text{vars-append-clauses}[\text{of elements } M' \text{ elements } ([l, d]) \@ M'']
      using \text{vars-append-clauses}[\text{of elements } ([l, d]) \text{ elements } M'']
      by simp
    from \( \ast\ast \ast: \text{vars}(\text{elements } (M' \@ M'')) = V' \)
    have \( \text{vars}(\text{elements } M) = \text{insert } v \ V' \)
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 = M' @ [(l, d)] @ M''⟩ ; uniq (\elements (M' @ M''))
  (\neg \literalTrue l (\elements M'')) ; (\neg \literalFalse l (\elements M''))
  by (auto iff: uniqAppendIff)
moreover
have consistent (\elements M)
proof-
{    assume \neg consistent (\elements M)
then obtain \l' where \literalTrue \l' (\elements M) \literalFalse \l' (\elements M)
    by (auto simp add: inconsistentCharacterization)
have False
proof (cases \l' = l)
  case True
  with ⟨\literalFalse \l' (\elements M)⟩ ⟨M = M' @ [(l, d)] @ M''⟩
  have \literalFalse \l' (\elements (M' @ M''))
    using oppositeIsDifferentFromLiteral[of l]
    by (auto split: split-if-asm)
  with (\neg \literalFalse l (\elements M')) ; \l' = l
  show ?thesis
}
by auto

next
case False
  with \langle\text{literateTrue } l' \ (\text{elements } M)\rangle \langle M = M' @ [(l, d)] @ M'' \rangle
  have \text{literateTrue } l' \ (\text{elements } (M' @ M''))
    by (auto split: split-if-asm)
  with \langle\text{consistent} \ (\text{elements } (M' @ M''))\rangle
  have \neg \text{literalFalse } l' \ (\text{elements } (M' @ M''))
    by (auto simp add: inconsistentCharacterization)
  with \langle\text{literalFalse } l' \ (\text{elements } M)\rangle \langle M = M' @ [(l, d)] @ M'' \rangle
  have opposite \( l' = l \)
    by (auto split: split-if-asm)
  with \langle\text{var } l = v\rangle
  have var \( l' = v \)
    by auto
  with \langle\text{literateTrue } l' \ (\text{elements } (M' @ M''))\rangle \langle\text{vars} \ (\text{elements } (M' @ M'')) = V'\rangle
  have \( v \in V' \)
    using valuationContainsItsLiteralsVariable[of \( l' \) elements (\text{}(M' @ M''))]
    by simp
  qed
\}

thus \?thesis
  by auto

qed

ultimately

show \( M \in \mathcal{L}_\text{hs} \)
  by auto

qed

qed

moreover

let \( \mathcal{F} = \lambda ((M', M''), l, d), M' @ [(l, d)] @ M'' \)

let \( \mathcal{Mset} = \{(M', M''), M' @ M'' \in \mathcal{Trails} \ V'\} \)

let \( \mathcal{VSet} = \{\text{Pos } v, \text{Neg } v\} \)

let \( \mathcal{DSet} = \{\text{True}, \text{False}\} \)

have \( \mathcal{Trails}' V' = \mathcal{F} \cdot (\mathcal{Mset} \times \mathcal{VSet} \times \mathcal{DSet}) \ (\text{is } \mathcal{Lhs} = \mathcal{Rhs}) \)

proof
  show \( \mathcal{Lhs} \subseteq \mathcal{Rhs} \)
  proof
    fix \( M :: \text{LiteralTrail} \)
    assume \( M \in \mathcal{Lhs} \)
    then obtain \( M' M'' l d \)
    where \( P: M = M' @ [(l, d)] @ M'' \)
    \( M' @ M'' \in (\mathcal{Trails} V') \)
\[ l \in \{ \text{Pos} \ v, \text{Neg} \ v \} \ d \in \{ \text{True}, \text{False} \} \]

- **by auto**
  - **show** \( M \in ?\text{rhs} \)
  - **proof**
    - **from** \( P \)
      - **show** \( M = ?f ((M', M''), l, d) \)
        - **by** `simp`
    - **next**
      - **from** \( P \)
        - **show** \( ((M', M''), l, d) \in ?Mset \times ?lSet \times ?dSet \)
          - **by auto**
  - **qed**
  - **qed**

- **next**
  - **show** \( ?\text{rhs} \subseteq ?\text{lhs} \)
  - **proof**
    - **fix** \( M::\text{LiteralTrail} \)
      - **assume** \( M \in ?\text{rhs} \)
      - **then obtain** \( p \ l \ d \text{ where } P: M = ?f (p, l, d) \) \( p \in ?Mset \ l \in ?lSet \ d \in ?dSet \)
        - **by auto**
      - **from** \( p \in ?Mset \)
        - **obtain** \( M' M'' \text{ where } M' @ M'' \in ?\text{trails} V' \)
          - **by auto**
        - **thus** \( M \in ?\text{lhs} \)
          - **using** \( P \)
          - **by auto**
    - **qed**
    - **qed**
  - **moreover**
    - **have** \( ?\text{Mset} = \{(M', M''). \exists \ l. \ l \in ?\text{trails} V' \land l = M' @ M''\} \)
      - **by auto**
    - **hence** finite \( ?\text{Mset} \)
      - **using** `insert(3)`
      - **using** `finiteListDecomposeSet[of ?trails V']`
        - **by simp**
    - **ultimately**
      - **show** \( \text{?thesis} \)
        - **by auto**
    - **qed**
    - **qed**

**lemma** `finiteUniqAndConsistentTrailsWithGivenVariableSuperset`:

  **fixes** \( V :: \text{Variable set} \)
  **assumes** `finite V`
  **shows** `finite \( \{ (M::\text{LiteralTrail}). \text{vars} (\text{elements} M) \subseteq V \land \text{uniq} (\text{elements} M) \land \text{consistent} (\text{elements} M) \} \) (\text{is finite} \ ?\text{trails} V) \)`
  **proof**
    - **have** \( \{ M. \text{vars} (\text{elements} M) \subseteq V \land \text{uniq} (\text{elements} M) \land \text{consistent} \}

189
\[\{\text{elements } M\}\} = \\
\{(\bigcup v \in \text{Pow } V.\{M. \text{vars (elements } M) = v \land \text{uniq (elements } M)\\n\land \text{consistent (elements } M)\})\}
\]
by auto
moreover
have finite \((\bigcup v \in \text{Pow } V.\{M. \text{vars (elements } M) = v \land \text{uniq (elements } M)\\n\land \text{consistent (elements } M)\})\)
proof (rule finite-UN-I)
from \(\text{finite } V\)
show finite \((\text{Pow } V)\)
by simp
next
fix \(v\)
assume \(v \in \text{Pow } V\)
with \(\text{finite } V\)
have finite \(v\)
by (auto simp add: finite-subset)
thus finite \(\{M. \text{vars (elements } M) = v \land \text{uniq (elements } M)\\n\land \text{consistent (elements } M)\}\)
using finiteUniQAndConsistentTrailsWithGivenVariableSuper-
set[of \(v\)]
by simp
qed
ultimately
show \(?\)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-uf)
show finite \((\text{lexLessRestricted } Vbl)\)
proof
let \(?X = \{((M1, M2).\\n\text{consistent (elements } M1) \land \text{uniq (elements } M1) \land \text{vars (elements } M1) \subseteq Vbl \land \\
\text{consistent (elements } M2) \land \text{uniq (elements } M2) \land \text{vars (elements } M2) \subseteq Vbl\}\)
let \(?Y = \{M. \text{vars (elements } M) \subseteq Vbl \land \text{uniq (elements } M) \land \\
\text{consistent (elements } M)\}\)
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
\[\text{by } \text{auto} \]
\[\text{qed} \]
\[\text{qed} \]

\(\text{lexLessRestricted}\) is also transitive.

\textbf{lemma} \(\text{transLexLessRestricted}:\)
\begin{itemize}
\item shows \(\text{trans}(\text{lexLessRestricted} \ Vbl)\)
\end{itemize}
\textbf{proof}\begin{itemize}
\item \{\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 \(\text{lexLessRestricted-def}\)
\item using \(\text{translexLess}\)
\item unfolding \(\text{trans-def}\)
\item by \text{auto}\end{itemize}\}
\item thus \(?\text{thesis}\)
\item unfolding \(\text{trans-def}\)
\item by \text{blast}\end{itemize}
\textbf{qed}

\subsection{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}
\(\text{multLess} \ M = \text{inv-image} \ (\text{mult} \ (\text{precedesOrder} \ (\text{elements} \ M))) \ (\lambda \ x. \ \text{multiset-of} \ (\text{remdups} \ (\text{oppositeLiteralList} \ x)))\)

The following lemma will help prove that application of the \textit{Explain} DPLL transition rule decreases the conflict clause in the \textit{multLess} ordering.

\textbf{lemma} \(\text{multLessResolve}:\)
\begin{itemize}
\item assumes \(\text{opposite} \ l \ \text{el} \ C\) and \(\text{isReason} \ reason \ l \ (\text{elements} \ M)\)
\item shows \((\text{resolve} \ C \ \text{reason} (\text{opposite} \ l), C) \in \text{multLess} \ M\)
\end{itemize}
\textbf{proof}\begin{itemize}
\item let \(?X = \text{multiset-of} \ (\text{remdups} \ (\text{oppositeLiteralList} \ C))\)
\item let \(?Y = \text{multiset-of} \ (\text{remdups} \ (\text{oppositeLiteralList} \ (\text{resolve} \ C \ \text{reason} (\text{opposite} \ l))))\)
\item let \(\text{ord} = \text{precedesOrder} \ (\text{elements} \ M)\)
\end{itemize}
have $(?Y, ?X) \in (\text{mult} ?\text{ord})$

proof –
  let $?Z = \text{multiset-of} \ (\text{remdups} \ (\text{oppositeLiteralList} \ (\text{removeAll} \ (\text{opposite} l \ C))))$
  let $?W = \text{multiset-of} \ (\text{remdups} \ (\text{oppositeLiteralList} \ (\text{removeAll} \ l \ (\text{list-diff} \ \text{reason} \ C))))$
  let $?a = l$
  from $(\text{opposite} l) \ \text{el} \ C$
  have $?X = ?Z + \{#?a#\}$
  using $\text{removeAll-multiset}$
  using $\text{oppositeLiteralListRemove}$
  using $\text{literalElListIffOppositeLiteralElOppositeLiteralList}$
  by auto

moreover
  have $?Y = ?Z + ?W$
  proof –
    have $\text{list-diff} \ (\text{oppositeLiteralList} \ (\text{removeAll} \ l \ \text{reason})) \ (\text{oppositeLiteralList} \ (\text{removeAll} \ (\text{opposite} l \ C))) = \text{oppositeLiteralList} \ (\text{removeAll} \ l \ (\text{list-diff} \ \text{reason} \ C))$
    proof –
      from $(\text{isReason} \ \text{reason} \ l \ (\text{elements} \ M))$
      have $\text{opposite} l \ \notin \ \text{set} \ \text{removeAll} \ l \ \text{reason}$
      unfolding $\text{isReason-def}$
      by auto
    hence $\text{list-diff} \ (\text{removeAll} \ l \ \text{reason}) \ (\text{removeAll} \ (\text{opposite} l \ C)) = \text{list-diff} \ (\text{removeAll} \ l \ \text{reason}) \ C$
      using $\text{listDiffRemoveAllNonMember}$
      by simp
      thus $?\text{thesis}$
      unfolding $\text{oppositeLiteralList-def}$
      using $\text{listDiffMap}$
      by auto
    qed
    thus $?\text{thesis}$
    unfolding $\text{resolve-def}$
    using $\text{rempdupsAppendMultiSet}$
    by auto
  qed
  moreover
  have $\forall \ b. \ b :\# \ ?W \longrightarrow (b, \ ?a) \in ?\text{ord}$
  proof –
    {\ fx \ b
assume $b : \# ?W$

hence opposite $b \in \text{set} \ (\text{removeAll} \ l \ \text{reason})$

proof -

from ⟨$b : \# ?W$⟩

have $b \in \text{remdup} \ (\text{oppositeLiteralList} \ (\text{removeAll} \ l \ (\text{list-diff reason} \ C)))$

by (auto simp add: set-count-greater-0)

hence opposite $b \in \text{removeAll} \ l \ (\text{list-diff reason} \ C)$

using literalElListIffOppositeLiteralElOppositeLiteralList[of opposite $b \in \text{removeAll} \ l \ (\text{list-diff reason} \ C)]$

by auto

hence opposite $b \in \text{list-diff} \ (\text{removeAll} \ l \ \text{reason}) \ C$

by simp

thus $?thesis$

using listDiffIff[of opposite $b \in \text{removeAll} \ l \ \text{reason} \ C]$

by simp

qed

with ⟨isReason reason $l \ (\text{elements} \ M)⟩$

have precedes $b \ l \ (\text{elements} \ M) \ b \neq l$

unfolding isReason-def

unfolding precedes-def

by auto

hence $(b, ?a) \in ?ord$

unfolding precedesOrder-def

by simp

} 

thus $?thesis$

by auto

qed

ultimately

have $\exists \ a \ M0 \ K. \ ?X = M0 + \{\#a\#\} \wedge \ ?Y = M0 + K \wedge (\forall b. \ b :\# K \rightarrow (b, a) \in \ ?ord)$

by auto

thus $?thesis$

unfolding mult1-def

by auto

qed

hence $(?Y, ?X) \in (\text{mult1} \ ?ord)^+$

by simp

thus $?thesis$

unfolding multLess-def

unfolding mult-def

unfolding inv-image-def

by auto

qed

lemma multLessListDiff:

assumes $(a, b) \in \text{multLess} \ M$


194
shows
\((\text{list-diff } a, b) \in \text{multLess } M\)

proof—
\begin{itemize}
  \item let \(?pOrd = \text{precedesOrder (elements } M)\)
  \item let \(?f = \lambda l. \text{remdups (map opposite } l)\)
  \item have \(\text{trans } ?pOrd\)
    using \(\text{transPrecedesOrder[of elements } M]\)
    by simp
  \item have \((\text{multiset-of } (?f a), \text{multiset-of } (?f b)) \in \text{mult } ?pOrd\)
    using \(\text{assms}\)
    unfolding \text{multLess-def}
    unfolding \text{oppositeLiteralList-def}
    by simp
  \item moreover
    \item have \(\text{multiset-le } (\text{multiset-of } (\text{list-diff } (?f a) (?f x)))\)
    \((\text{multiset-of } (?f a))\)
    \(?pOrd\)
    using \(\text{trans } ?pOrd\)
    using \(\text{multisetLeListDiff[of } ?pOrd ?f a ?f x]\)
    by simp
  \item ultimately
    \item have \((\text{multiset-of } (\text{list-diff } (?f a) (?f x)), \text{multiset-of } (?f b)) \in \text{mult } ?pOrd\)
    unfolding \text{multiset-le-def}
    unfolding \text{mult-def}
    by auto
  \end{itemize}
  thus \(?thesis\)
  unfolding \text{multLess-def}
  unfolding \text{oppositeLiteralList-def}
  by \(\text{(simp add: listDiffMap remdupsListDiff)}\)

qed

lemma \text{multLessRemdups}:\n
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—
\begin{itemize}
  \item fix \(l\)
    \item have \(\text{remdups (map opposite } l) = \text{remdups (map opposite (remdups } l)\))\)
      by \(\text{(induct } l)\) auto
  \end{itemize}
  thus \(?thesis\)
using assms
unfolding multLess-def
unfolding oppositeLiteralList-def
by simp
qed

Now we show that \textit{multLess} is well-founded.

\textbf{lemma} \textit{wfMultLess}:
\textbf{shows} \textit{wf} (\textit{multLess} \textit{M})
\textbf{proof}−
  \textbf{have} \textit{wf} (\textit{precedesOrder} (\textit{elements} \textit{M}))
  \textbf{by} (simp add: wellFoundedPrecedesOrder)
  \textbf{hence} \textit{wf} (\textit{mult} (\textit{precedesOrder} (\textit{elements} \textit{M})))
  \textbf{by} (simp add: wf-mult)
  \textbf{thus} ?thesis
    unfolding multLess-def
    using wf-inv-image[of (\textit{mult} (\textit{precedesOrder} (\textit{elements} \textit{M})))]
    by auto
\textbf{qed}

\subsection{ConflictFlag ordering}

A trivial ordering on Booleans. It will be used for the \textit{Conflict} transition rule.

\textbf{definition} \textit{boolLess} = \{(\textit{True}, \textit{False})\}

We show that it is well-founded

\textbf{lemma} \textit{transBoolLess}:
\textbf{shows} \textit{trans} \textit{boolLess}
\textbf{proof}−
  \{  
    \textbf{fix} \textit{x}::\textit{bool} \textbf{and} \textit{y}::\textit{bool} \textbf{and} \textit{z}::\textit{bool}
    \textbf{assume} (\textit{x}, \textit{y}) \in \textit{boolLess}
    \textbf{hence} \textit{x} = \textit{True} \textit{y} = \textit{False}
      unfolding boolLess-def
      \textbf{by} auto
    \textbf{assume} (\textit{y}, \textit{z}) \in \textit{boolLess}
    \textbf{hence} \textit{y} = \textit{True} \textit{z} = \textit{False}
      unfolding boolLess-def
      \textbf{by} auto
    from (\textit{y} = \textit{False}) \cdot (\textit{y} = \textit{True})
    \textbf{have} False
      \textbf{by} simp
    \textbf{hence} (\textit{x}, \textit{z}) \in \textit{boolLess}
      \textbf{by} simp
  \}
  \textbf{thus} ?thesis
unfolding \texttt{trans-def}
by \texttt{blast}

\texttt{qed}

\texttt{lemma \texttt{wfBoolLess}:}
\texttt{shows \texttt{wf boolLess}}
\texttt{proof (rule finite-acyclic-wf)}
\texttt{show finite boolLess}
\texttt{unfolding boolLess-def}
\texttt{by simp}
\texttt{next}
\texttt{have boolLess$^+ =$ boolLess}
\texttt{using \texttt{transBoolLess}}
\texttt{by simp}
\texttt{thus acyclic boolLess}
\texttt{unfolding boolLess-def}
\texttt{unfolding acyclic-def}
\texttt{by auto}

\texttt{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 \textit{Learn} transtion
rule.

\texttt{definition learnLess (C::Clause) == \{((F1::Formula), (F2::Formula)).
C el F1 \land \neg C el F2\}}

We show that it is well founded

\texttt{lemma \texttt{wfLearnLess}:}
\texttt{fixes C::Clause}
\texttt{shows \texttt{wf (learnLess C)}}
\texttt{unfolding \texttt{wf-eq-minimal}}
\texttt{proof--}
\texttt{show \forall Q F. F \in Q \rightarrow (\exists Fmin\in Q. \forall F'. (F', Fmin) \in learnLess
C \rightarrow F' \notin Q)}
\texttt{proof--}
\texttt{fix F::Formula and Q::Formula set}
\texttt{assume F \in Q}
\texttt{have \exists Fmin\in Q. \forall F'. (F', Fmin) \in learnLess C \rightarrow F' \notin Q}
\texttt{proof (cases \exists Fc \in Q. C el Fc)}
\texttt{case True}
\texttt{then obtain Fc where Fc \in Q C el Fc}
\texttt{by auto}
\texttt{have \forall F'. (F', Fc) \in learnLess C \rightarrow F' \notin Q}
\texttt{proof}
\texttt{fix F'}
show \((F', Fc) \in \text{learnLess } C \rightarrow F' \notin Q\)
proof
assume \((F', Fc) \in \text{learnLess } C\)

hence \(\neg C \text{ el } Fc\)
unfolding \text{learnLess-def}
by \text{auto}
with \((C \text{ el } Fc)\) have \(\text{False}\)
by \text{simp}
thus \(F' \notin Q\)
by \text{simp}
qed

qed

with \((Fc \in Q)\)
show ?thesis
by \text{auto}

next

case \text{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 \(C \text{ el } F'\)
unfolding \text{learnLess-def}
by \text{simp}
with \text{False}
show \(F' \notin Q\)
by \text{auto}
qed

qed

with \((F \in Q)\)
show ?thesis
by \text{auto}
qed

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

qed

4.4.5 Properties of well-founded relations.

lemma \text{wellFoundedEmbed}: 

\text{fixes } rel :: ('a \times 'a) \text{ set and rel'} :: ('a \times 'a) \text{ set}

\text{assumes } \forall x y. (x, y) \in rel \rightarrow (x, y) \in \text{rel'} \text{ and } \text{wf rel'}

\text{shows } \text{wf rel}

unfolding \text{wf-eq-minimal}

proof—
\[ \forall Q \; x \in Q \rightarrow (\exists z_{\text{min}} \in Q, \forall z. (z, z_{\text{min}}) \in \text{rel} \rightarrow z \notin Q) \]

proof -
\{ 
  fix x::'a and Q::'a set
  assume x \in Q
  have \exists z_{\text{min}} \in Q, \forall z. (z, z_{\text{min}}) \in \text{rel} \rightarrow z \notin Q
  proof -
  from (wf \text{rel}') \langle x \in Q \rangle
  obtain z_{\text{min}}::'a
    where z_{\text{min}} \in Q \text{ and } \forall z. (z, z_{\text{min}}) \in \text{rel'} \rightarrow z \notin Q
  unfolding \text{wf-eq-minimal}
  by auto
  \{ 
    fix z::'a
    assume (z, z_{\text{min}}) \in \text{rel}
    have z \notin Q
    proof -
    from \forall x y. (x, y) \in \text{rel} \rightarrow (x, y) \in \text{rel'} \langle (z, z_{\text{min}}) \in \text{rel} \rangle
    have (z, z_{\text{min}}) \in \text{rel'}
      by simp
    with \forall z. (z, z_{\text{min}}) \in \text{rel'} \rightarrow z \notin Q
    show \text{?thesis}
      by simp
    qed
  } 
  with \langle z_{\text{min}} \in Q \rangle
  show \text{?thesis}
    by auto
  qed
\}
thus \text{?thesis}
by auto
qed
qed

end

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.

**record State =**

\[ getM :: \text{LiteralTrail} \]

Procedure checks the satisfiability of the formula F0 which does not change during the solving process. An external parameter is the set `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

**definition**

\[ \text{appliedDecide} :: \text{State} \Rightarrow \text{State} \Rightarrow \text{Variable set} \Rightarrow \text{bool} \]

**where**

\[ \text{appliedDecide stateA stateB decisionVars} == \]

\[ \exists l. \]

\[ (\text{var} l) \in \text{decisionVars} \land \]

\[ \neg l \in \text{elements} (\text{getM stateA}) \land \]

\[ \neg \text{opposite} l \in \text{elements} (\text{getM stateA}) \land \]

\[ \text{getM stateB} = \text{getM stateA} @ \{(l, \text{True})\} \]

**definition**

\[ \text{applicableDecide} :: \text{State} \Rightarrow \text{Variable set} \Rightarrow \text{bool} \]

**where**

\[ \text{applicableDecide state decisionVars} == \exists \text{state}'. \text{appliedDecide state decisionVars} \]

**definition**

\[ \text{appliedUnitPropagate} :: \text{State} \Rightarrow \text{State} \Rightarrow \text{Formula} \Rightarrow \text{bool} \]

**where**

\[ \text{appliedUnitPropagate stateA stateB F0} == \]

\[ \exists (\text{uc}::\text{Clause}) (\text{ul}::\text{Literal}). \]

\[ \text{uc el F0} \land \]

\[ \text{isUnitClause uc ul (elements (getM stateA))} \land \]

\[ \text{getM stateB} = \text{getM stateA} @ \{(\text{ul}, \text{False})\} \]

**definition**

\[ \text{applicableUnitPropagate} :: \text{State} \Rightarrow \text{Formula} \Rightarrow \text{bool} \]

**where**

\[ \text{applicableUnitPropagate state F0} == \exists \text{state}'. \text{appliedUnitPropagate state state'} F0 \]

**definition**

\[ \text{appliedBacktrack} :: \text{State} \Rightarrow \text{State} \Rightarrow \text{Formula} \Rightarrow \text{bool} \]

**where**
appliedBacktrack stateA stateB F0 ==
    formulaFalse F0 (elements (getM stateA)) ∧
    decisions (getM stateA) ≠ [] ∧

    getM stateB = prefixBeforeLastDecision (getM stateA) ⊕ [opposite
    (lastDecision (getM stateA)), False]

definition
    applicableBacktrack :: State ⇒ Formula ⇒ bool
    where
    applicableBacktrack state F0 == ∃ state'. appliedBacktrack state state'
        F0

Solving starts with the empty trail.

definition
    isInitialState :: State ⇒ Formula ⇒ bool
    where
    isInitialState state F0 ==
        getM state = []

Transitions are performed only by using one of the three given rules.

definition
    transition stateA stateB F0 decisionVars ==
        appliedDecide stateA stateB decisionVars ∨
        appliedUnitPropagate stateA stateB F0 ∨
        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 ⇒ Formula ⇒ Variable set ⇒ bool
    where
    isFinalState state F0 decisionVars == ¬ (∃ state'. transition state
        stateF0 decisionVars)

The following several lemmas give conditions for applicability of different rules.

lemma applicableDecideCharacterization:
    fixes stateA::State
    shows applicableDecide stateA decisionVars =
(∃ l.
    (var l) ∈ decisionVars ∧
    ¬ l el (elements (getM stateA)) ∧
    ¬ opposite l el (elements (getM stateA)))
(is ?lhs = ?rhs)

proof
assume ?rhs
then obtain l where
  ∗: (var l) ∈ decisionVars ¬ l el (elements (getM stateA)) ¬ opposite
  l el (elements (getM stateA))
  unfolding applicableDecide-def
  by auto
let ?stateB = stateA( getM := (getM stateA) @ [(l, True)] )
from ∗ have appliedDecide stateA ?stateB decisionVars
  unfolding appliedDecide-def
  by auto
thus ?lhs
  unfolding applicableDecide-def
  by auto
next
assume ?lhs
then obtain stateB l
  where (var l) ∈ decisionVars ¬ l el (elements (getM stateA))
    ¬ opposite l el (elements (getM stateA))
  unfolding applicableDecide-def
  unfolding applicableDecide-def
  by auto
thus ?rhs
  by auto
qed

lemma applicableUnitPropagateCharacterization:
fixes stateA::State and F0::Formula
shows applicableUnitPropagate stateA F0 =
  (∃ (uc::Clause) (ul::Literal).
    uc el F0 ∧
    isUnitClause uc ul (elements (getM stateA)))
(is ?lhs = ?rhs)

proof
assume ?rhs
then obtain ul uc
  where ∗: uc el 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
  unfolding appliedUnitPropagate-def
  by auto
thus ?lhs

202
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)) \land
decisions (getM stateA) \neq []) (is ?lhs = ?rhs)
proof
  assume ?rhs
  hence *: formulaFalse F0 (elements (getM stateA)) decisions (getM stateA) \neq []
    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) \neq []
    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 invariantsHoldInState stateA F0 decisionVars \rangle 
have 
  InvariantImpliedLiterals F0 (getM stateA) and 
  InvariantVarsM (getM stateA) F0 decisionVars and 
  InvariantConsistent (getM stateA) and 
  InvariantUniq (getM stateA) 
  unfolding invariantsHoldInState-def 
  by auto 

{ 
  assume appliedDecide stateA stateB decisionVars 
  then obtain l::Literal where 
  \( \forall \text{var } l \in \text{decisionVars} \) 
  \( \neg \text{literalTrue } l \) (elements (getM stateA)) 
  \( \neg \text{literalFalse } l \) (elements (getM stateA)) 
  getM stateB = getM stateA @ [(l, True)] 
  unfolding appliedDecide-def 
  by auto 

  from \( \neg \text{literalTrue } l \) (elements (getM stateA)) \( \neg \text{literalFalse } l \) (elements (getM stateA)) 
  have \( \forall \text{var } l \notin \text{vars} \) (elements (getM stateA)) 
  using variableDefinedImpliesLiteralDefined[of l elements (getM stateA)] 
  by simp 

  have InvariantImpliedLiterals F0 (getM stateB) 
  using 
  \langle getM stateB = getM stateA @ [(l, True)] \rangle 
  \langle InvariantImpliedLiterals F0 (getM stateA) \rangle 
  \langle InvariantUniq (getM stateA) \rangle 
  \langle \forall \text{var } l \notin \text{vars} \) (elements (getM stateA)) \rangle 
  InvariantImpliedLiteralsAfterDecide[of F0 getM stateA l getM stateB] 
  by simp 
  moreover 
  have InvariantVarsM (getM stateB) F0 decisionVars 
  using \langle getM stateB = getM stateA @ [(l, True)] \rangle 
  \langle InvariantVarsM (getM stateA) F0 decisionVars \rangle 
  \langle \forall \text{var } l \in \text{decisionVars} \rangle 
  InvariantVarsMAfterDecide[of getM stateA F0 decisionVars l getM stateB] 
  by simp 
  moreover 
  have InvariantConsistent (getM stateB) 
  using \langle getM stateB = getM stateA @ [(l, True)] \rangle 
  \langle InvariantConsistent (getM stateA) \rangle 
  \langle \forall \text{var } l \notin \text{vars} \) (elements (getM stateA)) \rangle 
  InvariantConsistentAfterDecide[of getM stateA l getM stateB] 
  by simp
moreover
have InvariantUniq \{ \text{getM stateB} \}
  using \{ \text{getM stateB} = \text{getM stateA} \oplus [(l, True)] \}
  \langle \text{InvariantUniq (getM stateA)} \rangle
  \langle \text{var l \notin vars (elements (getM stateA))} \rangle
  \langle \text{InvariantUniqAfterDecide[of getM stateA l getM stateB]} \rangle
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 \oplus [(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 \{ \text{getM stateB} \}
  using
  \langle \text{InvariantImpliedLiterals F0 (getM stateA)} \rangle
  \langle \text{formulaEntailsClause F0 uc} \rangle
  \langle \text{isUnitClause uc ul (elements (getM stateA))} \rangle
  \langle \text{getM stateB = getM stateA \oplus [(ul, False)]} \rangle
  \langle \text{InvariantImpliedLiteralsAfterUnitPropagate[of F0 getM stateA uc ul getM stateB]} \rangle
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 \{ \text{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
moreover
have InvariantVarsM (getM stateB) F0 decisionVars
using (InvariantVarsM (getM stateA) F0 decisionVars)
  (decisions (getM stateA) ≠ []);
  getM stateB = prefixBeforeLastDecision (getM stateA) @
  [(opposite (lastDecision (getM stateA)), False)];
InvariantVarsMAfterBacktrack[of getM stateA F0 decisionVars
getM stateB]
  by simp
moreover
have InvariantConsistent (getM stateB)
using (InvariantConsistent (getM stateA));
  (InvariantUniq (getM stateA));
  (decisions (getM stateA) ≠ []);
  (getM stateB = prefixBeforeLastDecision (getM stateA) @
  [(opposite (lastDecision (getM stateA)), False)];
InvariantConsistentAfterBacktrack[of getM stateA getM stateB]
  by simp
ultimately
have ?thesis
  unfolding invariantsHoldInState-def
  by auto
}
ultimately
show ?thesis
  using (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
  (stateA, stateB) ∈ transitionRelation F0 decisionVars
  shows invariantsHoldInState stateB F0 decisionVars
using assms
using transitionsPreserveInvariants
using rtrancl-induct[of stateA stateB
  {(stateA, stateB). transition stateA stateB F0 decisionVars} λ x.
208
lemma invariantsHoldInValidRunsFromInitialState:
  fixes F0 :: Formula and decisionVars :: Variable set
  assumes isInitialState state0 F0
  and (state0, state) ∈ transitionRelation F0 decisionVars
  shows invariantsHoldInState state F0 decisionVars
proof
  from ⟨isInitialState state0 F0⟩
  have invariantsHoldInState state0 F0 decisionVars
  by (simp add: invariantsHoldInInitialState)
  with assms
  show ?thesis
  using invariantsHoldInValidRuns [of state0 F0 decisionVars state]
  by simp
qed

In the following text we will show that there are two kinds of states:

1. UNSAT states where formulaFalse F0 (elements (getM state)) and decisions (getM state) = []
2. SAT states where ¬ formulaFalse F0 (elements (getM state)) and decisionVars ⊆ 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.

5.3 Soundness

theorem soundnessForUNSAT:
  fixes F0 :: Formula and decisionVars :: Variable set and state0 :: State
  assumes isInitialState state0 F0 and
  (state0, state) ∈ transitionRelation F0 decisionVars
  formulaFalse F0 (elements (getM state))
  decisions (getM state) = []
  shows ¬ satisfiable F0
proof
from \(\text{isInitialState state0 } F0\) : \((\text{state0, state}) \in \text{transitionRelation } F0 \text{ decisionVars}\)
have \(\text{invariantsHoldInState state } F0 \text{ decisionVars}\)
using \(\text{invariantsHoldInValidRunsFromInitialState}\)
by simp
hence \(\text{InvariantImpliedLiterals } F0 \text{ (getM state)}\)
unfolding \(\text{invariantsHoldInState-def}\)
by auto
with \(\text{formulaFalse } F0 \text{ (elements (getM state))}\)
\(\langle \text{decisions (getM state) = []} \rangle\)
show \(?thesis\)
using \(\text{unsatReport}[\text{of } F0 \text{ getM state } F0]\)
unfolding \(\text{InvariantEquivalent-def equivalentFormulae-def}\)
by simp
qed

theorem soundnessForSAT:
fixes \(F0 :: \text{Formula and decisionVars :: Variable set and state0 :: State and state :: State}\)
assumes \(\text{vars } F0 \subseteq \text{decisionVars and}\)
\(\text{isInitialState state0 } F0 \text{ and}\)
\((\text{state0, state}) \in \text{transitionRelation } F0 \text{ decisionVars}\)
\(\neg \text{formulaFalse } F0 \text{ (elements (getM state))}\)
\(\text{vars } (\text{elements (getM state)}) \supseteq \text{decisionVars}\)
shows \(\text{model } (\text{elements (getM state)}) F0\)

proof
from \(\text{isInitialState state0 } F0\) : \((\text{state0, state}) \in \text{transitionRelation } F0 \text{ decisionVars}\)
have \(\text{invariantsHoldInState state } F0 \text{ decisionVars}\)
using \(\text{invariantsHoldInValidRunsFromInitialState}\)
by simp
hence \(\text{InvariantConsistent } \text{(getM state)}\)
unfolding \(\text{invariantsHoldInState-def}\)
by auto
with assms
show \(?thesis\)
using \(\text{satReport}[\text{of } F0 \text{ decisionVars } F0 \text{ getM state}]\)
unfolding \(\text{InvariantEquivalent-def equivalentFormulae-def}\)
\(\text{InvariantVarsF-def}\)
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.

\textbf{definition terminationLess} ($F0::\text{Formula}\ decisionVars == \{((\text{stateA::State}), (\text{stateB::State})), (\text{getM stateA, getM stateB}) \in \text{lexLessRestricted} (\text{vars } F0 \cup \text{decisionVars})\}$

We want to show that every valid transition decreases a state with respect to the constructed termination ordering. Therefore, we show that \textit{Decide, UnitPropagate} and \textit{Backtrack} rule decrease the trail with respect to the restricted trail ordering. Invariants ensure that trails are indeed \textit{uniq, consistent} and with finite variable sets.

\textbf{lemma trailIsDecreasedByDecidedUnitPropagateAndBacktrack:}
\begin{itemize}
  \item \textbf{fixes stateA::State and stateB::State}
  \item \textbf{assumes invariantsHoldInState stateA F0 decisionVars and appliedDecide stateA stateB decisionVars \lor appliedUnitPropagate stateA stateB F0 \lor appliedBacktrack stateA stateB F0}
  \item \textbf{shows (getM stateB, getM stateA) \in \text{lexLessRestricted} (\text{vars } F0 \cup \text{decisionVars})}
\end{itemize}

\textbf{proof –}
\begin{itemize}
  \item \textbf{from \langle\text{appliedDecide stateA stateB decisionVars} \lor \text{appliedUnitPropagate stateA stateB F0} \lor \text{appliedBacktrack stateA stateB F0}\rangle \langle\text{invariantsHoldInState stateA F0 decisionVars}\rangle}
  \item \textbf{have \text{invariantsHoldInState stateB F0 decisionVars} using transitionsPreserveInvariants}
  \item \textbf{unfolding \text{transition-def} by auto}
  \item \textbf{from \langle\text{invariantsHoldInState stateA F0 decisionVars}\rangle}
  \item \textbf{have *: uniq (elements (getM stateA)) consistent (elements (getM stateA)) vars (elements (getM stateA)) \subseteq \text{vars } F0 \cup \text{decisionVars} unfolding \text{invariantsHoldInState-def}}
  \item \textbf{unfolding \text{InvariantVarsM-def}}
  \item \textbf{unfolding \text{InvariantConsistent-def}}
  \item \textbf{unfolding \text{InvariantUniq-def} by auto}
  \item \textbf{from \langle\text{invariantsHoldInState stateB F0 decisionVars}\rangle}
  \item \textbf{have **: uniq (elements (getM stateB)) consistent (elements (getM stateB)) vars (elements (getM stateB)) \subseteq \text{vars } F0 \cup \text{decisionVars} unfolding \text{invariantsHoldInState-def}}
  \item \textbf{unfolding \text{InvariantVarsM-def}}
  \item \textbf{unfolding \text{InvariantConsistent-def}}
\end{itemize}
unfolding InvariantUniq-def
by auto

\{ 
\begin{align*}
\text{assume} & \; \text{appliedDecide stateA stateB decisionVars} \\
\text{hence} & \; (\text{getM stateB, getM stateA}) \in \text{lexLess} \\
\text{unfolding} & \; \text{appliedDecide-def} \\
& \; (\text{auto simp add: lexLessAppend}) \\
\text{with} & \; * * * \\
\text{have} & \; ((\text{getM stateB}, (\text{getM stateA})) \in \text{lexLessRestricted (vars F0} \\
\quad & \; \cup \text{ decisionVars)}) \\
& \; \text{unfolding lexLessRestricted-def} \\
& \; (\text{by auto}) \\
\end{align*}
\}

moreover
\{ 
\begin{align*}
\text{assume} & \; \text{appliedUnitPropagate stateA stateB F0} \\
\text{hence} & \; (\text{getM stateB, getM stateA}) \in \text{lexLess} \\
\text{unfolding} & \; \text{appliedUnitPropagate-def} \\
& \; (\text{auto simp add: lexLessAppend}) \\
\text{with} & \; * * * \\
\text{have} & \; (\text{getM stateB, getM stateA}) \in \text{lexLessRestricted (vars F0} \\
\quad & \; \cup \text{ decisionVars}) \\
& \; \text{unfolding lexLessRestricted-def} \\
& \; (\text{by auto}) \\
\end{align*}
\}

moreover
\{ 
\begin{align*}
\text{assume} & \; \text{appliedBacktrack stateA stateB F0} \\
\text{hence} & \; \text{formulaFalse F0 (elements (getM stateA))} \\
& \; \text{decisions (getM stateA) \neq []} \\
& \; \text{getM stateB = prefixBeforeLastDecision (getM stateA) @ [(opposite} \\
\quad & \; (lastDecision (getM stateA)), False)]} \\
\text{unfolding} & \; \text{appliedBacktrack-def} \\
& \; (\text{by auto}) \\
\text{hence} & \; (\text{getM stateB, getM stateA}) \in \text{lexLess} \\
\text{using} & \; \text{decisions (getM stateA) \neq []:} \\
& \; \text{(getM stateB = prefixBeforeLastDecision (getM stateA) @} \\
\quad & \; [(opposite (lastDecision (getM stateA)), False)]} \\
& \; (\text{by (simp add: lexLessBacktrack}) \\
\text{with} & \; * * * \\
\text{have} & \; (\text{getM stateB, getM stateA}) \in \text{lexLessRestricted (vars F0} \\
\quad & \; \cup \text{ decisionVars}) \\
& \; \text{unfolding lexLessRestricted-def} \\
& \; (\text{by auto}) \\
\end{align*}
\}

ultimately
\begin{align*}
\text{show} & \; \text{?thesis} \\
\text{using} & \; \text{assms}
\end{align*}
by auto
qed

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 trailIsDecreasedByDecidedUnitPropagateAndBacktrack
    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−
  show ∀ Q state. state ∈ Q −→ (∃ stateMin ∈ Q. ∀ state’. (state’, stateMin) ∈ terminationLess 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) ∈ 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

Using the termination ordering we show that the transition relation is well founded on states reachable from initial state.

**Theorem** \(\text{wfTransitionRelation} :\)

**Fixes** \(\text{decisionVars} :: \text{Variable set} \text{ and } \text{F0} :: \text{Formula} \text{ and } \text{state0} :: \text{State} \)

**Assumes** finite \(\text{decisionVars} \text{ and } \text{isInitialState state0 F0} \)

**Shows** \(\text{wf } \{(\text{stateB, stateA}). \text{transition stateA stateB F0 decisionVars} \land \text{transition stateB stateA F0 decisionVars}\}\)

**Proof**–

let \(\text{?rel} = \{(\text{stateB, stateA}). \text{transition stateA stateB F0 decisionVars} \land \text{transition stateB stateA F0 decisionVars}\}\)

let \(\text{?rel'} = \text{terminationLess F0 decisionVars}\)

have \(\forall x y. (x, y) \in \text{?rel} \rightarrow (x, y) \in \text{?rel'}\)

**Proof**–


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 ∃ state. (state0, state) ∈ transitionRelation F0 decisionVars ∧ isFinalState state F0 decisionVars
proof –
  {  
    assume ¬ thesis
    let ?Q = { (state. (state0, state) ∈ transitionRelation F0 decisionVars) }
    let ?rel = { (stateB, stateA). (state0, stateA) ∈ transitionRelation F0 decisionVars ∧ transition stateA stateB F0 decisionVars }  
    have state0 ∈ ?Q
      unfolding transitionRelation-def
      by simp
    hence ∃ state. state ∈ ?Q
      by auto
  from assms
  have wf ?rel
    using wfTransitionRelation[of decisionVars state0 F0]
by auto
hence ∀ Q. (∃ x. x ∈ Q) → (∃ stateMin ∈ Q. ∀ state. (state, stateMin)) ∈ ?rel → state ∉ Q)
  unfoldingwf-eq-minimal
  by simp
hence (∃ x. x ∈ ?Q) → (∃ stateMin ∈ ?Q. ∀ state. (state, stateMin)) ∈ ?rel → state ∉ ?Q
  by rule
with (∃ state. state ∈ ?Q)
  have ?stateMin ∈ ?Q. ∀ state. (state, stateMin) ∈ ?rel → state ∉ ?Q
    by simp
  then obtain stateMin
    where stateMin ∈ ?Q and ∀ state. (state, stateMin) ∈ ?rel → state ∉ ?Q
    by auto
from (stateMin ∈ ?Q)
  have (state0, stateMin) ∈ transitionRelation F0 decisionVars
    by simp
  with (¬ thesis)
  have ¬ isFinalState stateMin F0 decisionVars
    by simp
  then obtain state′::State
    where transition stateMin state′ F0 decisionVars
    unfolding isFinalState-def
    by auto
  have (state′, stateMin) ∈ ?rel
    using (state0, stateMin) ∈ transitionRelation F0 decisionVars
      (transition stateMin state′ F0 decisionVars)
    by simp
  with (∀ state. (state, stateMin) ∈ ?rel → state ∉ ?Q)
  have state′ ∉ ?Q
    by force
moreover
from (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
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 form 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** $\neg (\exists \, Q :: (\text{State set}). \exists \, \text{state0} \in Q. \, \text{isInitialState state0} \, F_0 \land$
  
  $\forall \, \text{state} \in Q. \, (\exists \, \text{state}' \in Q. \, \text{transition state state'} \, F_0 \, \text{decisionVars}) )$

**proof**

- \{ assume $\neg$ ?thesis then obtain $Q :: \text{State set}$ and $\text{state0} :: \text{State}$ where $\text{isInitialState state0} \, F_0 \land \text{state0} \in Q$
  
  $\forall \, \text{state} \in Q. \, (\exists \, \text{state}' \in Q. \, \text{transition state state'} \, F_0 \, \text{decisionVars})$

  by auto

- let $\text{rel} = \{(\text{stateB}, \text{stateA}). \, (\text{state0}, \text{stateA}) \in \text{transitionRelation} \, F_0 \, \text{decisionVars} \land$

  $\text{transition stateA stateB} \, F_0 \, \text{decisionVars}\}$

  from $\langle \text{finite \, decisionVars} \rangle \langle \text{isInitialState state0} \, F_0 \rangle$

  have $\text{wf} \, \text{rel}$ using $\text{wfTransitionRelation}$

  by simp

  hence $\text{wfmin}: \forall \, Q \, x. \, x \in Q \longrightarrow$

  $(\exists \, z \in Q. \, \forall \, y. \, (y, z) \in \text{rel} \longrightarrow y \notin Q)$

  unfolding $\text{wf-eq-minimal}$

  by simp

- let $\text{Q} = \{ \text{state} \in Q. \, (\text{state0}, \text{state}) \in \text{transitionRelation} \, F_0 \, \text{decisionVars} \}$

  from $\langle \text{state0} \in Q \rangle$

  have $\text{state0} \in \text{?Q}$

  unfolding $\text{transitionRelation-def}$

  by simp

  with $\text{wfmin}$

  obtain $\text{stateMin} :: \text{State}$

  where $\text{stateMin} \in \text{?Q} \, \forall \, y. \, (y, \text{stateMin}) \in \text{rel} \longrightarrow y \notin \text{?Q}$

  apply (erule-tac $x=?Q$ in allE)

  by auto

- from $\langle \text{stateMin} \in \text{?Q} \rangle$

  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'} \text{ F0 decisionVars}))

obtain state'::State
 where state' \in Q \text{ transition stateMin state}' \text{ F0 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) \langle (state0, stateMin) \in \text{ transitionRelation F0 decisionVars} \rangle
(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 \text{ 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 applicableDecide state decisionVars
shows vars (elements (getM state)) \supseteq decisionVars

proof
fix \ l :: Variable
let \ l = Pos x
assume \ x \in decisionVars
hence var \ l = x and var \ l \in decisionVars and var (opposite \ l) \in decisionVars
by auto
with (\neg applicableDecide state decisionVars)
have literalTrue \ l (elements (getM state)) \lor literalFalse \ l (elements (getM state))
unfolding applicableDecideCharacterization
by force
with ⟨var ?l = x⟩
show \( x \in \text{vars} (\text{elements} (\text{getM state})) \)
using valuationContainsItsLiteralsVariable[\( \text{of } ?l \text{ elements} (\text{getM state}) \)]
using valuationContainsItsLiteralsVariable[\( \text{of opposite } ?l \text{ elements} (\text{getM state}) \)]
by auto
qed

lemma finalConflictingState:
fixes state :: State
assumes
\( \neg \) applicableBacktrack state \( F_0 \) and
\( \neg \) formulaFalse \( F_0 \) (\( \text{elements} (\text{getM state}) \))
shows
\( \text{decisions} (\text{getM state}) = \emptyset \)
using assms
using applicableBacktrackCharacterization
by auto

lemma finalStateCharacterizationLemma:
fixes state :: State
assumes
\( \neg \) applicableDecide state decisionVars and
\( \neg \) applicableBacktrack state \( F_0 \)
shows
\( (\neg \text{formulaFalse } F_0 (\text{elements} (\text{getM state})) \land \text{vars} (\text{elements} (\text{getM state})) \supseteq \text{decisionVars}) \lor \)
\( (\text{formulaFalse } F_0 (\text{elements} (\text{getM state})) \land \text{decisions} (\text{getM state}) = \emptyset) \)
proof (cases formulaFalse \( F_0 \) (\( \text{elements} (\text{getM state}) \)))
case True
hence \( \text{decisions} (\text{getM state}) = \emptyset \)
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 \( \text{thesis} \)
by simp
qed

**Theorem** finalStateCharacterization:
fixes \( F_0 :: \text{Formula} \) and \( \text{decisionVars :: Variable set} \) and \( \text{state0 :: State} \)
assumes
\( \text{isInitialState \, state0 \, F_0 \, and} \)
\( (\text{state0, \, state}) \in \text{transitionRelation \, F_0 \, decisionVars} \) and
\( \text{isFinalState \, state \, F_0 \, decisionVars} \)
shows
\( (\neg \text{formulaFalse \, F_0 \, (elements \, (\text{getM \, state}) \wedge \text{vars} \, (\text{elements} \, (\text{getM \, state}))) \supseteq \text{decisionVars}) \vee \)
\( (\text{formulaFalse \, F_0 \, (elements \, (\text{getM \, state})) \wedge \text{decisions} \, (\text{getM \, state}) = \boxed{}}) \)
proof–
from \( \langle \text{isFinalState \, state \, F_0 \, decisionVars} \rangle \)
have **:
\( \neg \text{applicableBacktrack \, state \, F_0} \)
\( \neg \text{applicableDecide \, state \, decisionVars} \)
unfolding finalStateNonApplicable
by auto
thus \( \text{thesis} \)
using finalStateCharacterizationLemma[of state decisionVars]
by simp
qed

Completeness theorems are easy consequences of this characterization and soundness.

**Theorem** completenessForSAT:
fixes \( F_0 :: \text{Formula} \) and \( \text{decisionVars :: Variable set} \) and \( \text{state0 :: State} \)
assumes
\( \text{satisfiable \, F_0} \) and
\( \text{isInitialState \, state0 \, F_0 \, and} \)
\( (\text{state0, \, state}) \in \text{transitionRelation \, F_0 \, decisionVars} \) and
\( \text{isFinalState \, state \, F_0 \, decisionVars} \)
shows
\( \neg \text{formulaFalse \, F_0 \, (elements \, (\text{getM \, state})) \wedge \text{vars} \, (elements \, (\text{getM \, state})) \supseteq \text{decisionVars} \)
proof–
from \( \text{assms} \)
have *: \( \neg \text{formulaFalse } F_0 \, (\text{elements } (\text{getM } \text{state})) \land \text{vars } (\text{elements } (\text{getM } \text{state})) \supseteq \text{decisionVars} \lor \) 
\( (\text{formulaFalse } F_0 \, (\text{elements } (\text{getM } \text{state})) \land \text{decisions } (\text{getM } \text{state}) = \{\}) \) 
using finalStateCharacterization[af \text{state0 } F_0 \text{ state } \text{decisionVars}]
by auto
{
  assume formulaFalse F_0 \, (\text{elements } (\text{getM } \text{state}))
  with *
  have formulaFalse F_0 \, (\text{elements } (\text{getM } \text{state})) \, \text{decisions } (\text{getM } \text{state}) = \{}
  by auto
  with assms
  have \( \neg \) satisfiable F_0
  using soundnessForUNSAT
  by simp
  with (satisfiable F_0)
  have False
  by simp
}
with * show ?thesis
by auto
qed

theorem completenessForUNSAT:
fixes \( F_0 :: \text{Formula} \) and \( \text{decisionVars :: Variable set} \) and \( \text{state0 :: State} \) and \( \text{state :: State} \)
assumes
\( \text{vars } F_0 \subseteq \text{decisionVars} \) and
\( \neg \) satisfiable F_0 and

isInitialState state0 F_0 and
\( (\text{state0}, \text{state}) \in \text{transitionRelation } F_0 \, \text{decisionVars} \) and
isFinalState state F_0 decisionVars

shows
\( \text{formulaFalse } F_0 \, (\text{elements } (\text{getM } \text{state})) \land \text{decisions } (\text{getM } \text{state}) = \{\} \)

proof–
from assms
have *:
(\( \neg \text{formulaFalse } F_0 \, (\text{elements } (\text{getM } \text{state})) \land \text{vars } (\text{elements } (\text{getM } \text{state})) \supseteq \text{decisionVars} \lor \) 
(\( \text{formulaFalse } F_0 \, (\text{elements } (\text{getM } \text{state})) \land \text{decisions } (\text{getM } \text{state}) = \{\}) \) 
using finalStateCharacterization[af \text{state0 } F_0 \text{ state } \text{decisionVars}]
by auto

222
{ assume ¬ formulaFalse F0 (elements (getM state)) with *
  have ¬ formulaFalse F0 (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 F0 decisionVars and
  isFinalState state F0 decisionVars

  shows satisfiable F0 = (¬ formulaFalse F0 (elements (getM state)))

using assms
using completenessForUNSAT[of F0 decisionVars state0 state]
using completenessForSAT[of F0 state0 state decisionVars]
by auto

end

6 Transition system of Nieuwenhuis, Oliveras and Tinelli.

theory NieuwenhuisOliverasTinelli
imports SatSolverVerification
begin

This theory formalizes the transition rule system given by Nieuwen-
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 appliedUnitPropagate :: State ⇒ State ⇒ bool
where
appliedUnitPropagate 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′. appliedUnitPropagate state state′

definition appliedBackjump :: State ⇒ State ⇒ bool
where
appliedBackjump stateA stateB ==
  ∃ bc bl level.
  isUnitClause bc bl (elements (prefixToLevel level (getM stateA)))
\[\text{applicableBackjump} :: \text{State} \Rightarrow \text{bool} \]
\[\text{where} \]
\[\text{applicableBackjump state} == \exists \text{ state}' . \text{appliedBackjump state state}'\]

\[\text{definition} \]
\[\text{appliedLearn} :: \text{State} \Rightarrow \text{State} \Rightarrow \text{bool} \]
\[\text{where} \]
\[\text{appliedLearn stateA stateB} == \exists c. \]
\[\text{formulaEntailsClause} (\text{getF stateA}) c \land \]
\[\text{vars c} \subseteq \text{vars (getF stateA)} \cup \text{vars (elements (getM stateA))} \land \]
\[\text{getF stateB} = \text{getF stateA} \land \]
\[\text{getM stateB} = \text{prefixToLevel level (getM stateA)} @ [(\text{bl, False})]\]

\[\text{Solving starts with the initial formula and the empty trail.} \]

\[\text{definition} \]
\[\text{isInitialState} :: \text{State} \Rightarrow \text{Formula} \Rightarrow \text{bool} \]
\[\text{where} \]
\[\text{isInitialState state F0} == \text{getF state} = F0 \land \]
\[\text{getM state} = []\]

Transitions are preformed only by using given rules.

\[\text{definition} \]
\[\text{transition stateA stateB decisionVars} == \]
\[\text{appliedDecide stateA stateB decisionVars} \lor \]
\[\text{appliedUnitPropagate stateA stateB} \lor \]
\[\text{appliedLearn stateA stateB} \lor \]
\[\text{appliedBackjump stateA stateB}\]

Transition relation is obtained by applying transition rules iter-
atatively. It is defined using a reflexive-transitive closure.

**definition**

\[
\text{transitionRelation} \; \text{decisionVars} \;== \;\{(\text{stateA, stateB}). \text{transition} \;\text{stateA stateB decisionVars}\}\]

Final state is one in which no rules apply

**definition**

\[
isFinalState :: \;\text{State} \;\Rightarrow \;\text{Variable set} \;\Rightarrow \;\text{bool}
\]

where

\[
isFinalState \;\text{state} \;\text{decisionVars} \;== \;\neg \;\exists \;\text{state'}, \;\text{transition} \;\text{state state'} \;\text{decisionVars}
\]

The following several lemmas establish conditions for applicability of different rules.

**lemma** applicableDecideCharacterization:

**fixes** stateA::State

**shows** applicableDecide stateA decisionVars =

\[
(\exists \;\text{stateB}. \;\{(\text{var} \;\text{l}). \;\text{var} \;\text{l} \;\in \;\text{decisionVars} \land \neg \;\text{l el (elements (getM stateA))} \land \neg \;\text{opposite} \;\text{l el (elements (getM stateA))})\;

(\text{is} \;\text{?lhs} = \;\text{?rhs})
\]

**proof**

**assume** \(\text{?rhs}\)

then obtain \(\text{l}\) where

\*: \(\text{var} \;\text{l} \;\in \;\text{decisionVars} \;\neg \;\text{l el (elements (getM stateA))} \;\neg \;\text{opposite} \;\text{l el (elements (getM stateA))}\)

unfolding applicableDecide-def

by auto

let \(\text{?stateB = stateA@(getM := (getM stateA))@[(l, True)]}\)

from * have applicableDecide stateA ?stateB decisionVars

unfolding applicableDecide-def

by auto

thus \(\text{?lhs}\)

unfolding applicableDecide-def

by auto

next

**assume** \(\text{?lhs}\)

then obtain \(\text{stateB}\) \(\text{l}\)

where \(\text{var} \;\text{l} \;\in \;\text{decisionVars} \;\neg \;\text{l el (elements (getM stateA))} \;\neg \;\text{opposite} \;\text{l el (elements (getM stateA))}\)

unfolding applicableDecide-def

unfolding applicableDecide-def

by auto

thus \(\text{?rhs}\)

by auto

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 appliedUnitPropagate stateA ?stateB
    unfolding appliedUnitPropagate-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 appliedUnitPropagate-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
\[\text{var } \text{bl} \in \text{vars (getF stateA)} \cup \text{vars (elements (getM stateA))}\]
\[0 \leq \text{level level} < (\text{currentLevel (getM stateA)})\]
\[\text{unfolding applicableBackjump-def}\]
\[\text{by auto}\]
\[\text{let } ?\text{stateB} = \text{stateA}[\text{getM} := \text{prefixToLevel level (getM stateA)}] @ [(\text{bl}, \text{False})]\]
\[\text{from * have appliedBackjump stateA ?stateB}\]
\[\text{unfolding appliedBackjump-def}\]
\[\text{by auto}\]
\[\text{thus } ?\text{lhs}\]
\[\text{unfolding applicableBackjump-def}\]
\[\text{by auto}\]
next
\[\text{assume } ?\text{lhs}\]
\[\text{then obtain stateB}\]
\[\text{where appliedBackjump stateA stateB}\]
\[\text{unfolding applicableBackjump-def}\]
\[\text{by auto}\]
\[\text{then obtain } bc \text{ bl level}\]
\[\text{where isUnitClause bc bl (elements (prefixToLevel level (getM stateA)))}\]
\[\text{formulaEntailsClause (getF stateA) bc}\]
\[\text{var } bl \in \text{vars (getF stateA)} \cup \text{vars (elements (getM stateA))}\]
\[\text{getF stateB = getF stateA}\]
\[\text{getM stateB = prefixToLevel level (getM stateA)} @ [(bl, False)]\]
\[0 \leq \text{level level} < (\text{currentLevel (getM stateA)})\]
\[\text{unfolding appliedBackjump-def}\]
\[\text{by auto}\]
\[\text{thus } ?\text{rhs}\]
\[\text{by auto}\]
\[\text{qed}\]

\textbf{lemma applicableLearnCharacterization}:
\[\text{fixes stateA::State}\]
\[\text{shows applicableLearn stateA =}\]
\[\exists c. \text{formulaEntailsClause (getF stateA) c} \land\]
\[\text{vars c \subseteq vars (getF stateA)} \cup \text{vars (elements (getM stateA))}\]
\[\text{(is } ?\text{lhs = } ?\text{rhs)}\]

\textbf{proof}
\[\text{assume } ?\text{rhs}\]
\[\text{then obtain } c \text{ where}\]
\[*: \text{formulaEntailsClause (getF stateA) c}\]
\[\text{vars c \subseteq vars (getF stateA)} \cup \text{vars (elements (getM stateA))}\]
\[\text{unfolding applicableLearn-def}\]
\[\text{by auto}\]
\[\text{let } ?\text{stateB} = \text{stateA}[\text{getF} := \text{getF stateA} @ [c]]\]
\[\text{from * have appliedLearn stateA ?stateB}\]
\[\text{unfolding appliedLearn-def}\]
\[\text{by auto}\]
thus \( ?!hs \)

unfolding applicableLearn-def
by auto

next
assume \( ?!hs \)
then obtain \( c \) stateB
where
\[
\text{formulaEntailsClause} \ (\text{getF stateA}) \ c
\]
\[
\text{vars} \ c \subseteq \text{vars} \ (\text{getF stateA}) \cup \text{vars} \ (\text{elements} \ (\text{getM stateA}))
\]
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 =
\[
(\neg \text{applicableDecide state decisionVars} \land
\neg \text{applicableUnitPropagate state} \land
\neg \text{applicableBackjump state} \land
\neg \text{applicableLearn state})
\]
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 \( \Rightarrow \) Formula \( \Rightarrow \) Variable set \( \Rightarrow \) bool
where
invariantsHoldInState state F0 decisionVars ==
InvariantImpliedLiterals (getF state) (getM state) \land
InvariantVarsM (getM state) F0 decisionVars \land
InvariantVarsF (getF state) F0 decisionVars \land
InvariantConsistent (getM state) \land
InvariantUniq (getM state) \land
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))
      getM stateB = getM stateA @ [(l, True)]
      getF 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
}
have \textbf{InvariantImpliedLiterals} (\texttt{getF stateB}) (\texttt{getM stateB})

using \texttt{getF stateB = getF stateA}
\langle \texttt{getM stateB = getM stateA @ [(l, True)]} \rangle
\langle \textbf{InvariantImpliedLiterals} (\texttt{getF stateA}) (\texttt{getM stateA}) \rangle
\langle \textbf{InvariantUniq} (\texttt{getM stateA}) \rangle
\langle \texttt{\var l \notin \vars (elements (\texttt{getM stateA}))} \rangle
\textbf{InvariantImpliedLiteralsAfterDecide}[\texttt{of getF stateA getM stateA l getM stateB}]

by \texttt{simp}

moreover

have \textbf{InvariantVarsM} (\texttt{getM stateB}) \texttt{F0 decisionVars}

using \texttt{(getM stateB = getM stateA @ [(l, True)])}
\langle \textbf{InvariantVarsM} (\texttt{getM stateA}) \texttt{F0 decisionVars} \rangle
\langle \texttt{\var l \in decisionVars} \rangle
\textbf{InvariantVarsMAfterDecide}[\texttt{of getM stateA F0 decisionVars l getM stateB}]

by \texttt{simp}

moreover

have \textbf{InvariantVarsF} (\texttt{getF stateB}) \texttt{F0 decisionVars}

using \texttt{(getF stateB = getF stateA)}
\langle \textbf{InvariantVarsF} (\texttt{getF stateA}) \texttt{F0 decisionVars} \rangle

by \texttt{simp}

moreover

have \textbf{InvariantConsistent} (\texttt{getM stateB})

using \texttt{(getM stateB = getM stateA @ [(l, True)])}
\langle \textbf{InvariantConsistent} (\texttt{getM stateA}) \rangle
\langle \texttt{\var l \notin \vars (elements (\texttt{getM stateA}))} \rangle
\textbf{InvariantConsistentAfterDecide}[\texttt{of getM stateA l getM stateB}]

by \texttt{simp}

moreover

have \textbf{InvariantUniq} (\texttt{getM stateB})

using \texttt{(getM stateB = getM stateA @ [(l, True)])}
\langle \textbf{InvariantUniq} (\texttt{getM stateA}) \rangle
\langle \texttt{\var l \notin \vars (elements (\texttt{getM stateA}))} \rangle
\textbf{InvariantUniqAfterDecide}[\texttt{of getM stateA l getM stateB}]

by \texttt{simp}

moreover

have \textbf{InvariantEquivalent} \texttt{F0} (\texttt{getF stateB})

using \texttt{(getF stateB = getF stateA)}
\langle \textbf{InvariantEquivalent} \texttt{F0} (\texttt{getF stateA}) \rangle

by \texttt{simp}

ultimately

have \texttt{thesis}

unfolding \texttt{invariantsHoldInState-def}

by \texttt{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 ⟨al 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
  using formulaContainsItsLiteralsVariable [of ul getF stateA]
  unfolding InvariantVarsF-def
  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⟩
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
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 getF stateB]
    by simp
moreover
  have InvariantVarsM (getM stateB) F0 decisionVars
    using

\[
\langle \text{InvariantVarsM} \ (\text{getM stateA}) \ F0 \ \text{decisionVars} \rangle \\
\langle \text{getM stateB} = \text{getM stateA} \rangle
\]

by simpl

moreover

from \(\text{vars c} \subseteq \text{vars (getF stateA)} \cup \text{vars (elements (getM stateA))}\)

\[
\langle \text{InvariantVarsM} \ (\text{getM stateA}) \ F0 \ \text{decisionVars} \rangle \\
\langle \text{InvariantVarsF} \ (\text{getF stateA}) \ F0 \ \text{decisionVars} \rangle
\]

have \(\text{vars c} \subseteq \text{vars F0} \cup \text{decisionVars}\)

unfolding \text{InvariantVarsM-def}

unfolding \text{InvariantVarsF-def}

by auto

hence \(\text{InvariantVarsF} \ (\text{getF stateB}) \ F0 \ \text{decisionVars}\)

using \(\langle \text{InvariantVarsF} \ (\text{getF stateA}) \ F0 \ \text{decisionVars} \rangle\)

\(\langle \text{getF stateB} = \text{getF stateA} @ [c] \rangle\)

using \text{varsAppendFormulae} \ [\text{of getF stateA} [c]]

unfolding \text{InvariantVarsF-def}

by simpl

moreover

have \(\text{InvariantConsistent} \ (\text{getM stateB})\)

using \(\langle \text{InvariantConsistent} \ (\text{getM stateA}) \rangle\)

\(\langle \text{getM stateB} = \text{getM stateA} \rangle\)

by simp

moreover

have \(\text{InvariantUniq} \ (\text{getM stateB})\)

using \(\langle \text{InvariantUniq} \ (\text{getM stateA}) \rangle\)

\(\langle \text{getM stateB} = \text{getM stateA} \rangle\)

by simp

moreover

have \(\text{InvariantEquivalent} \ F0 \ (\text{getF stateB})\)

using \(\langle \text{InvariantEquivalent} \ F0 \ (\text{getF stateA}) \rangle\)

\(\langle \text{formulaEntailsClause} \ (\text{getF stateA}) \ c \rangle\)

\(\langle \text{getF stateB} = \text{getF stateA} @ [c] \rangle\)

\(\langle \text{InvariantEquivalentAfterLearn} [\text{of F0 getF stateA c getF stateB}] \rangle\)

by simp

ultimately

have \(\text{?thesis}\)

unfolding \text{invariantsHoldInState-def}

by simp

}

moreover

{ 
  assume \text{appliedBackjump} \text{stateA stateB}
  then obtain \text{bc::Clause and bl::Literal and level::nat}
    where
      \text{isUnitClause} bc bl (\text{elements (prefixToLevel level (getM stateA))})
      \text{formulaEntailsClause} (\text{getF stateA}) bc
      var bl \in \text{vars (getF stateA)} \cup \text{vars (elements (getM stateA))}
      \text{getF stateB} = \text{getF stateA}
}

234
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
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[of getM stateA prefixToLevel level (getM stateA) bc bl]
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⟩ λ x. invariantsHoldInState x F0 decisionVars]
unfolding transitionRelation-def
by auto

lemma invariantsHoldInValidRunsFromInitialState:
  fixes F0 :: Formula and decisionVars :: Variable set
  assumes initialState 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. **UNSAT** states where \(\text{formulaFalse } F_0 \text{ (elements (getM state))} \) and \(\text{decisions (getM state)} = \[]\).
2. **SAT** states where \(\neg \text{formulaFalse } F_0 \text{ (elements (getM state))} \) and \(\text{decisionVars} \subseteq \text{vars (elements (getM state))}\).

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.

### 6.3 Soundness

**theorem** soundnessForUNSAT:

\[\begin{align*}
\text{fixes } & F_0 :: \text{Formula and decisionVars :: Variable set and state0 :: State and state :: State} \\
\text{assumes } & \text{isInitialState state0 F0 and (state0, state) } \in \text{transitionRelation decisionVars} \\
\text{shows } & \neg \text{satisfiable F0}
\end{align*}\]

**proof** –

\[\begin{align*}
\text{from } & \text{(isInitialState state0 F0: (state0, state) } \in \text{transitionRelation decisionVars)} \\
\text{have } & \text{invariantsHoldInState state F0 decisionVars} \\
\text{using } & \text{invariantsHoldInValidRunsFromInitialState} \\
\text{by } & \text{simp} \\
\text{hence } & \text{InvariantImpliedLiterals (getF state) (getM state) InvariantE-equivalent F0 (getF state)} \\
\text{unfolding } & \text{invariantsHoldInState-def} \\
\text{by } & \text{auto}
\end{align*}\]
with \( \text{formulaFalse} \ (\text{getF state}) \ (\text{elements} \ (\text{getM state})) \);
\( \text{decisions} \ (\text{getM state}) = [] \);

show \?thesis
using unsatReport[of getF state getM state F0]
by simp

qed

\textbf{theorem soundnessForSAT}:
fixes \( F0 :: \text{Formula} \) and \( \text{decisionVars} :: \text{Variable set} \) and \( \text{state0} :: \text{State} \) and \( \text{state} :: \text{State} \)
assumes
\( \text{vars} \ F0 \subseteq \text{decisionVars} \) and
\( \text{isInitialState} \ \text{state0} \ F0 \ and \ (\text{state0}, \text{state}) \in \text{transitionRelation} \ \text{decisionVars} \)

\( \neg \text{formulaFalse} \ (\text{getF state}) \ (\text{elements} \ (\text{getM state})) \)
\( \text{vars} \ (\text{elements} \ (\text{getM state})) \supseteq \text{decisionVars} \)
shows
\( \text{model} \ (\text{elements} \ (\text{getM state})) \ F0 \)

proof–
from \( \text{isInitialState} \ \text{state0} \ F0 \) and \( ((\text{state0}, \text{state}) \in \text{transitionRelation} \ \text{decisionVars}) \)

\begin{align*}
\text{have} & \quad \text{invariantsHoldInState} \ \text{state0} \ F0 \ \text{decisionVars} \\
\text{using} & \quad \text{invariantsHoldInValidRunsFromInitialState} \\
\text{by} & \quad \text{simp} \\
\text{hence} & \quad \text{InvariantConsistent} \ (\text{getM state}) \\
& \quad \text{InvariantEquivalent} \ F0 \ (\text{getF state}) \\
& \quad \text{InvariantVarsF} \ (\text{getF state}) \ F0 \ \text{decisionVars} \\
\text{unfolding} & \quad \text{invariantsHoldInState-def} \\
\text{by} & \quad \text{auto} \\
\text{with} & \quad \text{assms} \\
\text{show} & \quad \?thesis \\
\text{using} & \quad \text{satReport[of} \ F0 \ \text{decisionVars} \ \text{getF state} \ \text{getM state]} \\
\text{by} & \quad \text{simp} \\
\text{qed}
\end{align*}

\textbf{6.4 Termination}

This system is terminating, but only under assumption that there is no infinite derivation consisting only of applications of rule \textit{Learn}. We will formalize this condition by requiring that there there exists an ordering \textit{learnL} on the formulae that is well-founded such that the state is decreased with each application of the \textit{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.


definition lexLessState F0 decisionVars == \{((stateA::State), (stateB::State)).

\( (\text{getM stateA}, \text{getM stateB}) \in \text{lexLessRestricted} \) \\
\( \{ \text{vars F0} \cup \text{decisionVars} \} \) \\
definition learnLessState learnL == \{((stateA::State), (stateB::State)).

\( \text{getM stateA} = \text{getM stateB} \wedge (\text{getF stateA}, \text{getF stateB}) \in \text{learnL} \) \\
definition terminationLess F0 decisionVars learnL == \{((stateA::State), (stateB::State)).

\( (\text{stateA, stateB}) \in \text{lexLessState F0 decisionVars} \lor \\
\( \text{stateA, stateB}) \in \text{learnLessState learnL} \) \}

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} trailIsDecreasedByDeciedUnitPropagateAndBackjump: 

\textbf{fixes} stateA::State and stateB::State \\
\textbf{assumes} invariantsHoldInState stateA F0 decisionVars and \\
\textbf{appliedDecide stateA stateB decisionVars} \lor \textbf{appliedUnitPropagate stateA stateB} \lor \textbf{appliedBackjump stateA stateB} \\
\textbf{shows} (\text{getM stateB}, \text{getM stateA}) \in \text{lexLessRestricted} (\text{vars F0} \cup \text{decisionVars}) \\
\textbf{proof} \\
\textbf{from} (\text{appliedDecide stateA stateB decisionVars} \lor \text{appliedUnitPropagate stateA stateB} \lor \text{appliedBackjump stateA stateB}) \\
\textbf{have} invariantsHoldInState stateA F0 decisionVars \\
\textbf{using} transitionsPreserveInvariants \\
\textbf{unfolding} transition-def \\
\textbf{by} auto \\
\textbf{from} (\text{invariantsHoldInState stateA F0 decisionVars}) \\
\textbf{have} *: uniq (\text{elements} (\text{getM stateA})) consistent (\text{elements} (\text{getM stateA})) \subseteq \text{vars F0} \cup \text{decisionVars} \\
\textbf{unfolding} invariantsHoldInState-def \\
\textbf{unfolding} InvariantVarsM-def \\
\textbf{unfolding} InvariantConsistent-def \\
\textbf{unfolding} InvariantUniq-def \\
\textbf{by} auto \\
\textbf{from} (\text{invariantsHoldInState stateB F0 decisionVars}) \\
\textbf{have} **: uniq (\text{elements} (\text{getM stateB})) consistent (\text{elements} (\text{getM stateB})) \\
\textbf{by} auto \\
\textbf{from} (\text{invariantsHoldInState stateB F0 decisionVars}) \\
\textbf{have} ***: uniq (\text{elements} (\text{getM stateB})) consistent (\text{elements} (\text{getM stateB}))
\[ \text{stateB}) \ vars \ (\text{elements \ (getM \ stateB)}) \subseteq \text{vars} \ F_0 \cup \text{decisionVars} \]

**unfolding** \text{invariantsHoldInState-def} 
**unfolding** \text{InvariantVarsM-def} 
**unfolding** \text{InvariantConsistent-def} 
**unfolding** \text{InvariantUniq-def} 

by auto 

\{
assume \text{appliedDecide \ stateA \ stateB \ decisionVars}
\hence (\text{getM \ stateB}, \text{getM \ stateA}) \in \text{lexLess}
\text{unfolding} \ \text{appliedDecide-def}
\text{by} \ (\text{auto \ simp \ add:lexLessAppend})
\text{with} \ * * *
\text{have} ((\text{getM \ stateB}), (\text{getM \ stateA})) \in \text{lexLessRestricted} \ \text{(vars} \ F_0 \cup \text{decisionVars})
\text{unfolding} \ \text{lexLessRestricted-def}
\text{by} \ \text{auto}
\}

moreover 
\{
assume \text{appliedUnitPropagate \ stateA \ stateB}
\hence (\text{getM \ stateB}, \text{getM \ stateA}) \in \text{lexLess}
\text{unfolding} \ \text{appliedUnitPropagate-def}
\text{by} \ (\text{auto \ simp \ add:lexLessAppend})
\text{with} \ * * *
\text{have} (\text{getM \ stateB}, \text{getM \ stateA}) \in \text{lexLessRestricted} \ \text{(vars} \ F_0 \cup \text{decisionVars})
\text{unfolding} \ \text{lexLessRestricted-def}
\text{by} \ \text{auto}
\}

moreover 
\{
assume \text{appliedBackjump \ stateA \ stateB}
\text{then obtain} \ bc::\text{Clause and bl::Literal and level::nat}
\text{where}
\text{isUnitClause bc bl (elements \ \text{(prefixToLevel \ level \ (getM \ stateA)))}}
\text{formulaEntailsClause \ (getF \ stateA) \ bc}
\text{var \ bl \in \text{vars} \ \text{(getF \ stateA)} \cup \text{vars} \ \text{\text{(elements \ (getM \ stateA))}}}
\text{0 \leq \text{level} \ \text{level} < \text{currentLevel} \ \text{(getM \ stateA)}}
\text{getF \ stateB = getF \ stateA}
\text{getM \ stateB = prefixToLevel \ level \ \text{(getM \ stateA)} \ @ \ [(\text{bl, False})]}
\text{unfolding} \ \text{appliedBackjump-def}
\text{by} \ \text{auto}

\text{with} (\text{getM \ stateB = prefixToLevel \ level \ (getM \ stateA)} \ @ \ [(\text{bl, False})])
\text{have} (\text{getM \ stateB}, \text{getM \ stateA}) \in \text{lexLess}
\text{by} \ (\text{simp \ add:lexLessBackjump})
\text{with} \ * * *
\text{have} (\text{getM \ stateB}, \text{getM \ stateA}) \in \text{lexLessRestricted} \ \text{(vars} \ F_0 \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:

```plaintext
fixes stateA::State and stateB::State
assumes invariantsHoldInState stateA F0 decisionVars and transition stateA stateB decisionVars
appliedLearn stateA stateB \(\rightarrow\) (getF stateB, getF stateA) ∈ learnL
shows (stateB, stateA) ∈ terminationLess F0 decisionVars learnL

proof

\[
\{ 
\begin{align*}
\text{assume } & \text{appliedDecide stateA stateB decisionVars }\lor \text{appliedUnitPropagate stateA stateB }\lor \text{appliedBackjump stateA stateB } \\
\text{with } & \langle\text{invariantsHoldInState stateA F0 decisionVars}\rangle \\
\text{have } & (getM stateB, getM stateA) \in \text{lexLessRestricted }\{\text{vars F0 }\cup \text{decisionVars}\} \\
\text{using } & \text{trailIsDecreasedByDecidedUnitPropagateAndBackjump } \\
& \text{by simp} \\
\text{hence } & (stateB, stateA) \in \text{lexLessState F0 decisionVars } \\
\text{unfolding } & \text{lexLessState-def } \\
& \text{by simp} \\
\text{hence } & (stateB, stateA) \in \text{terminationLess F0 decisionVars learnL } \\
\text{unfolding } & \text{terminationLess-def } \\
& \text{by simp} \\
\} \\
\text{moreover } \\
\{ 
\begin{align*}
\text{assume } & \text{appliedLearn stateA stateB } \\
\text{with } & \langle\text{appliedLearn stateA stateB }\rightarrow (getF stateB, getF stateA) \rangle \\
\in & \text{learnL} \\
\text{have } & (getF stateB, getF stateA) \in \text{learnL } \\
& \text{by simp} \\
\text{moreover } \\
\text{from } & \langle\text{appliedLearn stateA stateB}\rangle \\
\text{have } & (getM stateB) = (getM stateA) \\
\text{unfolding } & \text{appliedLearn-def } \\
& \text{by auto} \\
\text{ultimately } \\
\text{have } & (stateB, stateA) \in \text{learnLessState learnL }
\end{align*}
\}
\]
```

241
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 \( \forall Q. \forall state. \; state \in Q \longrightarrow (\exists stateMin \in Q. \forall state'. \; (state', stateMin) \in \text{lexLessState } F0 \; \text{decisionVars} \longrightarrow state' \notin Q) \) 
proof
  { 
    fix Q :: State set and state :: State 
    assume state ∈ Q 
    let \(?Q1 = \{M::LiteralTrail. \exists state. \; state \in Q \land (\text{getM state}) = M\}\) 
    from \(\text{state} \in Q\) 
    have \(\text{getM state} \in \?Q1\) 
      by auto 
    from \(\text{finite decisionVars}\) 
    have \(\text{finite } (\text{vars } F0 \cup \text{decisionVars})\) 
      using \(\text{finiteVarsFormula[of } F0\) 
      by simp 
    hence \(\text{wf } (\text{lexLessRestricted } (\text{vars } F0 \cup \text{decisionVars}))\) 
      using \(\text{wfLexLessRestricted[of } \text{vars } F0 \cup \text{decisionVars}\) 
      by simp 
    with \(\text{getM state} \in ?Q1\) 
    obtain \(M\text{min where } M\text{min} \in ?Q1 \land \forall M'. \; (M', M\text{min}) \in \text{lexLessRestricted } (\text{vars } F0 \cup \text{decisionVars}) \longrightarrow M' \notin ?Q1\) 
    unfolding \(\text{wf-eq-minimal}\) 
    apply \((\text{erule-tac } x=?Q1 \; \text{in } \text{allE})\) 
    apply \((\text{erule-tac } x=\text{getM state} \; \text{in } \text{allE})\) 
    by auto 
    from \(\text{Mmin} \in ?Q1\) obtain \(\text{stateMin}\) 
      where \(\text{stateMin} \in Q \land (\text{getM stateMin}) = M\text{min}\) 
      by auto 
    have \(\forall state'. \; (state', stateMin) \in \text{lexLessState } F0 \; \text{decisionVars} \longrightarrow state' \notin Q\) 
    proof 
      fix state' 
      show \(\text{state'} \in \text{lexLessState } F0 \; \text{decisionVars} \longrightarrow state' \notin Q\) 
      proof 
        assume \(\text{state'} \in \text{lexLessState } F0 \; \text{decisionVars}\) 
        hence \(\text{getM state'}, \text{getM stateMin} \in \text{lexLessRestricted } (\text{vars } F0 \cup \text{decisionVars})\) 
        unfolding \(\text{lexLessState-def}\) 
        by auto 
        from \(\forall M'. \; (M', M\text{min}) \in \text{lexLessRestricted } (\text{vars } F0 \cup \text{decisionVars}) \longrightarrow M' \notin ?Q1\)
\((getM\ state',\ getM\ stateMin)\) ∈ lexLessRestricted \((\text{vars } F0 \cup \text{decisionVars})\) \((getM\ stateMin = Mmin)\)

have \(getM\ state' \notin ?Q1\)
  by simp
with \((getM\ stateMin = Mmin)\)
show \(state' \notin Q\)
  by auto

qed

with \((stateMin \in Q)\)

have \(\exists\ stateMin \in Q.\ \forall\ state'.\ (state',\ stateMin)\) ∈ lexLessState
  \((\text{F0 \ decisionVars} \rightarrow state' \notin Q)\)
  by auto

}\text{ thus } \text{thesis}
  by auto

qed

lemma \text{wfLearnLessState}:
assumes \text{wf learnL}
shows \text{wf} \((\text{learnLessState learnL})\)
unfolding \text{wf-eq-minimal}

proof−
  show \(\forall\ Q\ state.\ state \in Q \rightarrow (\exists\ stateMin \in Q.\ \forall\ state'.\ (state',\ stateMin)\) \in \text{learnLessState learnL} \rightarrow state' \notin Q)\)
    proof−
    \{ fix \(Q :: \text{State set}\) and \(state :: \text{State}\)
      assume \(state \in Q\)
      let \(\?M = (getM\ state)\)
      let \(\?Q1 = \{f :: \text{Formula}.\ \exists \ state.\ state \in Q \land \text{(getM state)} = \?M \land \\text{(getF state)} = f\}\)
      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 (erule-tac \(x=\?Q1\) in \(\text{allE}\))
        apply (erule-tac \(x=\text{getF state}\) in \(\text{allE}\))
        by auto
      from \(\text{FMin} \in \?Q1\) obtain \(\text{stateMin}\)
        where \(\text{stateMin} \in Q\ \text{(getM stateMin)} = \?M \text{ getF stateMin} = \text{FMin}\)
          by auto
        have \(\forall\ state'.\ (state',\ stateMin)\) \in \text{learnLessState learnL} \rightarrow state' \notin Q\)
  }
proof
fix state'
show (state', stateMin) ∈ learnLessState learnL → state' ∉ Q
proof
assume (state', stateMin) ∈ learnLessState learnL
with ⟨getM stateMin = ?M⟩
have getM state' = getM stateMin (getF state', getF stateMin)
in learnL
  unfolding learnLessState-def
  by auto
from (∀ F'. (F', FMin) ∈ learnL → F' ∉ ?Q1)
  ⟨(getF state', getF stateMin) ∈ learnL; (getF stateMin = FMin)⟩
have getF 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) ∈ learnLessState
  learnL → state' ∉ Q)
  by auto
} thus ?thesis
  by auto
qed
qed

Now we can prove the following key lemma which shows that the
termination ordering is well founded.

lemma wfTerminationLess:
fixes F0 :: Formula and decisionVars :: Variable set
assumes finite decisionVars wf learnL
shows wf (terminationLess F0 decisionVars learnL)
unfolding wf-eq-minimal
proof
show (∀ Q state. state ∈ Q → (∃ stateMin ∈ Q. ∀ state'. (state',
  stateMin) ∈ learnLessState F0 decisionVars learnL → state' ∉ Q)
proof
{ fix Q::State set
fix state::State
assume state ∈ Q
have wf (lexLessState F0 decisionVars)
  using wfLexLessState[of decisionVars F0]
  using (finite decisionVars)
  by simp
with \( \text{state} \in Q \) obtain \( \text{state0} \)
where \( \text{state0} \in Q \land \text{state}'. (\text{state}', \text{state0}) \in \text{lexLessState F0} \)
\( \text{decisionVars} \rightarrow \text{state}' \notin Q \)

unfolding \( \text{wf-eq-minimal} \)
by auto
let \( ?Q0 = \{ \text{state. state} \in Q \land (\text{getM state}) = (\text{getM state0}) \} \)
from \( \text{state0} \in Q \)
have \( \text{state0} \in ?Q0 \)
by simp
from \( \text{wf learnL} \)
have \( \text{wf (learnLessState learnL)} \)
using \( \text{wfLearnLessState} \)
by simp
with \( \text{state0} \in ?Q0 \) obtain \( \text{state1} \)
where \( \text{state1} \in ?Q0 \land \text{state}'. (\text{state}', \text{state1}) \in \text{learnLessState learnL} \)
\( \text{learnL} \rightarrow \text{state}' \notin ?Q0 \)

unfolding \( \text{wf-eq-minimal} \)
apply (erule-tac \( x=\text{?Q0} \) 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 \( \forall \text{state}'. (\text{state}', \text{?stateMin}) \in \text{terminationLess F0 decision- Vars learnL} \rightarrow \text{state}' \notin Q \)
proof
fix \( \text{state}' \)
show \( (\text{state}', ?\text{stateMin}) \in \text{terminationLess F0 decisionVars learnL} \rightarrow \text{state}' \notin Q \)
proof
assume \( (\text{state}', ?\text{stateMin}) \in \text{terminationLess F0 decisionVars learnL} \)

hence
\( (\text{state}', ?\text{stateMin}) \in \text{lexLessState F0 decisionVars} \lor \)
\( (\text{state}', ?\text{stateMin}) \in \text{learnLessState learnL} \)

unfolding \( \text{terminationLess-def} \)
by auto
moreover
{ assume \( (\text{state}', ?\text{stateMin}) \in \text{lexLessState F0 decisionVars} \)
  with \( \text{getM state1} = \text{getM state0} \)
  have \( (\text{state}', \text{state0}) \in \text{lexLessState F0 decisionVars} \)
  unfolding \( \text{lexLessState-def} \)
  by simp
  with \( \forall \text{state}'. (\text{state}', \text{state0}) \in \text{lexLessState F0 decisionVars} \)
  \( \rightarrow \text{state}' \notin Q \)
  have \( \text{state}' \notin Q \)
  by simp
}
moreover

{ 
  assume \((state', ?stateMin) \in \text{learnLessState learnL with } \forall state'. (state', state1) \in \text{learnLessState learnL} \rightarrow state' \notin ?Q0) 
  have state' \notin ?Q0 
  by simp 
  from \((state', state1) \in \text{learnLessState learnL}) \langle \text{getM state1} = \text{getM state0} \rangle 
  have \text{getM state'} = \text{getM state0} 
  unfolding \text{learnLessState-def} 
  by auto 
  with \((state' \notin ?Q0) 
  have state' \notin Q 
  by simp 
}
ultimately
show state' \notin Q 
by auto 
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** \(\text{wfTransitionRelation}\):
\[\text{fixes decisionVars :: Variable set and } F0 :: \text{Formula}
\text{assumes finite decisionVars and isInitialState state0 F0 and}
\text{*: } \exists \text{learnL::}(\text{Formula } \times \text{Formula}) \text{ set.}
\text{wf learnL} \land
\text{(\forall stateA stateB. appliedLearn stateA stateB } \rightarrow \text{(getF stateB, getF stateA) } \in \text{learnL})
\text{shows wf \{((stateB, stateA).}
\text{(state0, stateA) } \in \text{transitionRelation decisionVars } \land
\text{(transition stateA stateB decisionVars)\}}\]

**proof**–
\[\text{from } * \text{ obtain learnL::}(\text{Formula } \times \text{Formula}) \text{ set}
\text{where}
\text{wf learnL and} \]
∀ stateA stateB. appliedLearn stateA stateB → (getF stateB, getF stateA) ∈ learnL
  by auto
let ?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 $\neg \ ?thesis$

let $\ ?Q = \{ \text{state}. (\text{state0}, \text{state}) \in \text{transitionRelation decisionVars} \}$

let $\ ?rel = \{ (\text{stateB}, \text{stateA}). (\text{state0}, \text{stateA}) \in \text{transitionRelation decisionVars} \land \text{transition stateA stateB decisionVars} \}$

have $\text{state0} \in \ ?Q$
  unfolding $\text{transitionRelation-def}$
  by simp
  hence $\exists \ \text{state. state} \in \ ?Q$
  by auto

from assms
have $\text{wf} \ ?rel$
  using $\text{wfTransitionRelation[of decisionVars state0 F0]}$
  by auto
  hence $\forall Q. (\exists x. x \in Q) \rightarrow (\exists \text{stateMin} \in Q. \forall \text{state}. (\text{state, stateMin}) \in ?rel \rightarrow \text{state} \notin Q)$
  unfolding $\text{wf-eq-minimal}$
  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 $\text{stateMin}$
    where $\text{stateMin} \in \ ?Q$ and $\forall \text{state}. (\text{state, stateMin}) \in ?rel \rightarrow \text{state} \notin ?Q$
    by auto

from $(\text{stateMin} \in ?Q)$
have $(\text{state0}, \text{stateMin}) \in \text{transitionRelation decisionVars}$
  by simp
  with $(\neg \ ?thesis)$
  have $\neg \ \text{isFinalState stateMin decisionVars}$
    by simp
  then obtain $\text{state'}::\text{State}$
    where $\text{transition stateMin state'}$ decisionVars
    unfolding $\text{isFinalState-def}$
    by auto
  have $(\text{state'}, \text{stateMin}) \in ?rel$
    using $(\text{state0}, \text{stateMin}) \in \text{transitionRelation decisionVars}$
    $(\text{transition stateMin state'}$ decisionVars)
    by simp
  with $\forall \text{state}. (\text{state, stateMin}) \in ?rel \rightarrow \text{state} \notin ?Q$
  have $\text{state'} \notin ?Q$
    by force
moreover
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 form 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::Formula and decisionVars::Variable set
  assumes finite decisionVars and
  ∀: ∃ learnL::(Formula × Formula) set.
    wf learnL ∧
    (∀ stateA stateB. appliedLearn stateA stateB → (getF stateB, getF stateA) ∈ learnL)
  shows ¬ (∃ Q::(State set). ∃ state0 ∈ Q. isInitialState state0 F0 ∧
            (∀ state ∈ Q. (∃ state' ∈ Q. transition state state' decisionVars)))

proof
{
  assume ¬ ?thesis
  then obtain Q::State set and state0::State
    where isInitialState state0 F0 state0 ∈ Q
        ∀ state ∈ Q. (∃ state' ∈ Q. transition state state' decisionVars)
    by auto
  let ?rel = { (stateB, stateA). (state0, stateA) ∈ transitionRelation decisionVars ∧
              transition stateA stateB decisionVars }
  from (finite decisionVars) (isInitialState state0 F0) *
  have wf ?rel
    using 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 simp

let $?Q = \{\text{state} \in Q. (\text{state0}, \text{state}) \in \text{transitionRelation decision-Vars}\}$

from $\langle \text{state0} \in Q \rangle$

have $\text{state0} \in {?Q}$

unfolding $\text{transitionRelation-def}$ by simp

with $\text{wfmin}$

obtain $\text{stateMin}::\text{State}$

where $\text{stateMin} \in {?Q}$ and $\forall y. (y, \text{stateMin}) \in {?\text{rel}} \rightarrow y \notin {?Q}$

apply (erule-tac $x = {?Q}$ in allE)

by auto

from $\langle \text{stateMin} \in {?Q} \rangle$

have $\text{stateMin} \in Q (\text{state0}, \text{stateMin}) \in \text{transitionRelation decisionVars}$

by auto

with $\langle \forall \text{state} \in Q. (\exists \text{state'} \in Q. \text{transition state state'} \text{ decision-Vars}) \rangle$

obtain $\text{state'}::\text{State}$

where $\text{state'} \in Q \text{ transition stateMin state'} \text{ decisionVars}$

by auto

with $\langle (\text{state0}, \text{stateMin}) \in \text{transitionRelation decisionVars} \rangle$

have $\langle \text{state'}, \text{stateMin} \rangle \in {?\text{rel}}$

by simp

with $\langle \forall y. (y, \text{stateMin}) \in {?\text{rel}} \rightarrow y \notin {?Q} \rangle$

have $\text{state'} \notin {?Q}$

by force

from $\langle \text{state'} \in Q. \langle (\text{state0}, \text{stateMin}) \in \text{transitionRelation decision-Vars} \rangle \langle \text{transition stateMin state'} \text{ decisionVars} \rangle \text{ have state'} \in {?Q}$

unfolding $\text{transitionRelation-def}$

using $\text{rtrancl-into-rtrancl}[\text{of state0 stateMin} \{\langle \text{stateA}, \text{stateB} \rangle. \text{transition stateA stateB decisionVars} \} \text{ state'}]$ by simp

with $\langle \text{state'} \notin {?Q} \rangle$

have $\text{False}$

by simp

} thus $?thesis$

by force

qed
6.5 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 ¬ applicableDecide state decisionVars
  
  shows 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[?l elements (getM state)]
  
  using valuationContainsItsLiteralsVariable[opposite ?l elements (getM state)]
  
  by auto
  
  qed
  
**lemma finalConflictingState:**
  
  fixes state :: State
  
  assumes InvariantUniq (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—**
  
  from ⟨InvariantUniq (getM state)⟩
  
  have uniq (elements (getM state))
  
  unfolding InvariantUniq-def
  
  from ⟨InvariantConsistent (getM state)⟩
  
  have consistent (elements (getM state))
  
  unfolding InvariantConsistent-def

252
let \(?c = \text{oppositeLiteralList\( (\text{decisions \( (\text{getM\ state}\))})\)}}
{
  assume \(\neg \?thesis\)
  hence \(?c \neq []\)
  using \(\text{oppositeLiteralListNonempty[\( \text{of\ decisions \( (\text{getM\ state}\))}\]}\)
  by simp
moreover
have \(\text{clauseFalse \(?c \ (\text{elements \( (\text{getM\ state}\))})\)}\)
proof –
{
  fix \(l::\text{Literal}\)
  assume \(l el \?c\)
  hence \(\text{opposite } l \ el \ \text{decisions \( (\text{getM\ state}\))}\)
  using \(\text{literalElListIffOppositeLiteralElOppositeLiteralList \[\text{of \(l \ \?c\)\]}\}
  by simp
  hence \(\text{literalFalse } l \ (\text{elements \( (\text{getM\ state}\))})\)
  using \(\text{markedElementsAreElements[\( \text{of\ opposite \( l \ \text{getM\ state}\)\]}\}
  by simp
}
thus \(?thesis\)
  using \(\text{clauseFalseIffAllLiteralsAreFalse[\( \text{of \(?c \ \text{elements \( (\text{getM\ state}\))}\]}\}
  by simp
qed
moreover
let \(\?l = \text{getLastAssertedLiteral \( (\\text{oppositeLiteralList \(?c\)}) \ \text{elements \( (\text{getM\ state}\))}\)}}\)
have \(\text{isLastAssertedLiteral \(?l \ (\text{oppositeLiteralList \(?c\)}) \ \text{elements \( (\text{getM\ state}\))}\)}\)
  using \(\text{InvariantUniq \( (\text{getM\ state})\)}\)
  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}}\)
  by simp
moreover
have \(\forall \ l. \ l el \ ?c \longrightarrow (\text{opposite } l) \ el \ (\text{decisions \( (\text{getM\ state}\))})\)
proof –
{
  fix \(l::\text{Literal}\)
  assume \(l el \ ?c\)
  hence \(\text{(opposite } l) \ el \ (\text{oppositeLiteralList \(?c\)})\)
  using \(\text{literalElListIffOppositeLiteralElOppositeLiteralList[\( \text{of \(l \ \?c\)\]}\}
  by simp
}
}
thus ?thesis
  by simp
qed
ultimately
have ∃ level. (isBackjumpLevel level (opposite ?l)?c (getM state))
  using \uniq (elements (getM state))
  using allDecisionsThenExistsBackjumpLevel[of getM state ?c opposite ?l]
  by simp
then obtain level::nat
  where isBackjumpLevel level (opposite ?l)?c (getM state)
  by auto
with \consistent (elements (getM state)) \uniq (elements (getM state)) \clauseFalse ?c (elements (getM state))
have isUnitClause ?c (opposite ?l) (elements (prefixToLevel level (getM state)))
  using isBackjumpLevelEnsuresIsUnitInPrefix[of getM state ?c level opposite ?l]
  by simp
moreover
have formulaEntailsClause (getF state) ?c
proof−
  from \clauseFalse ?c (elements (getM state)) \consistent (elements (getM state))
  have \neg clauseTautology ?c
    using tautologyNotFalse[of ?c elements (getM state)]
    by auto
  from \formulaFalse (getF state) (elements (getM state)) \InvariantImpliedLiterals (getF state) (getM state)
    have \neg satisfiable ((getF state) @ val2form (decisions (getM state)))
    using InvariantImpliedLiteralsAndFormulaFalseThenFormulaAndDecisionsAreNotSatisfiable
    by simp
    hence \neg satisfiable ((getF state) @ val2form (oppositeLiteralList ?c))
    by simp
    with \neg clauseTautology ?c
    show ?thesis
    using unsatisfiableFormulaWithSingleLiteralClauses
    by simp
qed
moreover
have var ?l ∈ vars (getF state) ∪ vars (elements (getM state))
proof−
  from isLastAssertedLiteral ?l (oppositeLiteralList ?c) (elements (getM state))
    have ?l el (oppositeLiteralList ?c)
    unfolding isLastAssertedLiteral-def
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
q e d
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
q e d
ultimately
have applicableBackjump state
  unfolding applicableBackjumpCharacterization
  by force
    with ⟨¬ applicableBackjump state⟩
    have False
      by simp
  }
  thus ?thesis
  by auto
q e d

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 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 decisionVars and
    isFinalState state decisionVars
  shows
    (∼ formulaFalse (getF state) (elements (getM state)) ∧ vars (elements (getM state)) ⊇ decisionVars) ∨
    (formulaFalse (getF state) (elements (getM state)) ∧ decisions (getM state) = [])
proof
  from ⟨isInitialState state0 F0: ((state0, state) ∈ transitionRelation decisionVars)⟩
  have invariantsHoldInState state F0 decisionVars
    using invariantsHoldInValidRunsFromInitialState
    by simp
  hence
    *: InvariantUniq (getM state)
    InvariantConsistent (getM state)
    InvariantImpliedLiterals (getF state) (getM state)
    unfolding invariantsHoldInState-def
    by auto
  from ⟨isFinalState state decisionVars⟩
  have **:
    ∼ applicableBackjump state
    ∼ applicableDecide state decisionVars
    unfolding finalStateNonApplicable

  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 ($\text{state0}, \text{state}$) $\in$ transitionRelation $\text{decisionVars}$ and

isFinalState $\text{state} \text{decisionVars}$

**Shows** $\neg \text{formulaFalse} (\text{getF state}) \ (\text{elements (getM state)}) \land \text{vars} \ (\text{elements (getM state)}) \supseteq \text{decisionVars}$

**Proof**

from **assms**

have *; ($\neg \text{formulaFalse} (\text{getF state}) \ (\text{elements (getM state)}) \land \text{vars} \ (\text{elements (getM state)}) \supseteq \text{decisionVars}) \lor$

($\text{formulaFalse} (\text{getF state}) \ (\text{elements (getM state)}) \land \text{decisions} \ (\text{getM state}) = []$)

using finalStateCharacterization [of state0 $F_0$ state decisionVars]

by auto

{ assume formulaFalse (getF state) (elements (getM state))

with *

have formulaFalse (getF state) (elements (getM state)) decisions

(getM state) = []

by auto

with assms

have $\neg$ satisfiable $F_0$

using soundnessForUNSAT

by simp

with (satisfiable $F_0$)

have $\text{False}$

by simp

}

with * show ?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
  initialState 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

258
isInitialState state0 F0 and (state0, state) ∈ transitionRelation decisionVars and isFinalState state decisionVars
shows satisfiable F0 = (¬ formulaFalse (getF state) (elements (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 ==
  ∃ 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)] ∧
getConflictFlag stateB = getConflictFlag stateA ∧
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 getF 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
applicableExplain :: State ⇒ bool
where
applicableExplain 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
applicableLearn :: State ⇒ bool
where
applicableLearn 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 \rightarrow State \rightarrow \text{bool}

\text{where}
\text{appliedBackjump stateA stateB ==}
\exists \ l \ \text{level}.
\text{getConflictFlag stateA} = \text{True} \land
\text{isBackjumpLevel level l (getC stateA) (getM stateA)} \land
\text{getF stateB} = \text{getF stateA} \land
\text{getM stateB} = \text{prefixToLevel level (getM stateA) @ [(l, False)]} \land
\text{getConflictFlag stateB} = \text{False} \land
\text{getC stateB} = []

\text{definition}
applicableBackjump :: State \rightarrow \text{bool}

\text{where}
\text{applicableBackjump state ==} \exists \ \text{state'}. \ \text{appliedBackjump state state'}

Solving starts with the initial formula, the empty trail and in non conflicting state.

\text{definition}
isInitialState :: State \rightarrow \text{Formula} \rightarrow \text{bool}

\text{where}
isInitialState state F0 ==
\text{getF state} = F0 \land
\text{getM state} = [] \land
\text{getConflictFlag state} = \text{False} \land
\text{getC state} = []

Transitions are performed only by using given rules.

\text{definition}
transition :: State \rightarrow State \rightarrow \text{Formula} \rightarrow \text{Variable set} \rightarrow \text{bool}

\text{where}
\text{transition stateA stateB F0 decisionVars ==}
\text{appliedDecide stateA stateB decisionVars} \lor
\text{appliedUnitPropagate stateA stateB F0 decisionVars} \lor
\text{appliedConflict stateA stateB} \lor
\text{appliedExplain stateA stateB} \lor
\text{appliedLearn stateA stateB} \lor
\text{appliedBackjump stateA stateB}

Transition relation is obtained by applying transition rules iteratively. It is defined using a reflexive-transitive closure.

\text{definition}
transitionRelation F0 decisionVars == (\{(stateA, stateB). transition stateA stateB F0 decisionVars\}) ^* \\

Final state is one in which no rules apply.

\text{definition}
isFinalState :: State ⇒ Formula ⇒ Variable set ⇒ bool

where

isFinalState state F0 decisionVars == ¬ (∃ state'. transition state state' F0 decisionVars)

The following several lemmas establish conditions for applicability of different rules.

lemma applicableDecideCharacterization:
fixes stateA::State
shows applicableDecide stateA decisionVars =
(∃ l.
 (var l) ∈ decisionVars ∧
 ¬ l el (elements (getM stateA)) ∧
 ¬ opposite l el (elements (getM stateA)))
(is ?lhs = ?rhs)

proof
assume ?rhs
then obtain l where
*: (var l) ∈ decisionVars ¬ l el (elements (getM stateA)) ¬ opposite
l el (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 (var l) ∈ decisionVars ¬ l el (elements (getM stateA))
 ¬ opposite l el (elements (getM stateA))
unfolding applicableDecide-def
unfolding appliedDecide-def
by auto
thus ?rhs
by auto
qed

lemma applicableUnitPropagateCharacterization:
fixes stateA::State and F0::Formula
shows applicableUnitPropagate stateA F0 decisionVars =
(∃ uc::Clause) (ul::Literal).
formulaEntailsClause (getF stateA) uc ∧
(var ul) ∈ decisionVars ∪ vars F0 ∧
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,
let $getC := []$ \\
from * have appliedBackjump stateA $\exists$ stateB \\
  unfolding appliedBackjump-def \\
  by auto \\
thus $?lhs$ \\
  unfolding applicableBackjump-def \\
  by auto \\
next \\
assume $?lhs$ \\
then obtain stateB $\exists$ level \\
  where getConflictFlag stateA = True \\
  isBackjumpLevel level $\exists$ (getC stateA) (getM stateA) \\
  unfolding applicableBackjump-def \\
  unfolding applicableBackjump-def \\
  by auto \\
thus $?rhs$ \\
  by auto \\
qed \\

lemma applicableExplainCharacterization: fixes stateA::State \\
shows applicableExplain stateA = 
  $(\exists$ $\exists$ reason. \\
  getConflictFlag stateA = True $\land$ \\
  $\exists$ el getC stateA $\land$ \\
  formulaEntailsClause (getF stateA) reason $\land$ \\
  isReason reason (opposite l) (elements (getM stateA)) \\
  ) (is $?lhs = $?rhs) \\
proof \\
assume $?rhs$ \\
then obtain $\exists$ level \\
  where *: \\
  getConflictFlag stateA = True \\
  $\exists$ el (getC stateA) formulaEntailsClause (getF stateA) reason $\land$ \\
  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 $\exists$ stateB \\
  unfolding applicableExplain-def \\
  by auto \\
thus $?lhs$ \\
  unfolding applicableExplain-def \\
  by auto \\
next \\
assume $?lhs$ \\
then obtain stateB $\exists$ reason \\
  where
getConflictFlag stateA = True
l el 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 =
(∃ clause.
  getConflictFlag stateA = False ∧
  formulaEntailsClause (getF stateA) clause ∧
  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( get := 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 applicableConflict-def
  by auto
thus ?rhs
by auto
qed

lemma applicableLearnCharacterization:
fixes stateA::State
shows applicableLearn stateA = 
\{getConflictFlag stateA = True ∧ 
\neg getC stateA el getF stateA\} (is ?lhs = ?rhs)

proof
assume ?rhs

hence \*: getConflictFlag stateA = True \neg 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 \neg (getC stateA) el (getF 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 F0 decisionVars = 
\neg applicableDecide state decisionVars ∧ 
\neg applicableUnitPropagate state F0 decisionVars ∧ 
\neg applicableBackjump state ∧ 
\neg applicableLearn state ∧ 
\neg applicableConflict state ∧ 
\neg 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.

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)

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
InvariantVarsM-def
InvariantVarsF-def
InvariantConsistent-def
InvariantUniq-def
InvariantReasonClauses-def
InvariantEquivalent-def equivalentFormulae-def
InvariantCFalse-def
InvariantCEntailed-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 ⟨invariantsHoldInState stateA F0 decisionVars⟩ have
    InvariantVarsM (getM stateA) F0 decisionVars and
    InvariantVarsF (getF stateA) F0 decisionVars and
    InvariantConsistent (getM stateA) and
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

269
moreover
have InvariantUniq (getM stateB)
  using ⟨getM stateB = getM stateA @ [(l, True)]⟩
  ⟨InvariantUniq (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)]⟩
  ⟨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)]⟩
  ⟨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⟩
  ⟨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 \( uc::\text{Clause} \) and \( ul::\text{Literal} \) where
\[
\begin{aligned}
\text{formulaEntailsClause} (\text{getF stateA}) \ uc \\
(\text{var ul}) \in \text{decisionVars} \cup \text{vars F0} \\
\text{isUnitClause} \ uc \ ul \ (\text{elements} (\text{getM stateA})) \\
\text{getF stateB} = \text{getF stateA} \\
\text{getM stateB} = \text{getM stateA} \uplus [(ul, False)] \\
\text{getConflictFlag stateB} = \text{getConflictFlag stateA} \\
\text{getC stateB} = \text{getC stateA}
\end{aligned}
\]
\[
\text{unfolding appliedUnitPropagate-def} \\
\text{by auto}
\]

\[
\begin{aligned}
\text{from} \ (\text{isUnitClause} \ uc \ ul \ (\text{elements} (\text{getM stateA}))) \\
\text{have} \ ul \ el \ uc \\
\text{unfolding isUnitClause-def} \\
\text{by simp}
\end{aligned}
\]

\[
\begin{aligned}
\text{from} \ (\text{var ul} \in \text{decisionVars} \cup \text{vars F0}) \\
\text{have} \ \text{InvariantVarsM} (\text{getM stateB}) \ F0 \ \text{decisionVars} \\
\text{using} \ (\text{getF stateB} = \text{getF stateA}) \\
\quad (\text{InvariantVarsM} (\text{getM stateA}) \ F0 \ \text{decisionVars}) \\
\quad (\text{getM stateB} = \text{getM stateA} \uplus [(ul, False)]) \\
\quad \text{InvariantVarsMAfterUnitPropagate} \ [\text{of getM stateA} \ F0 \ \text{decisionVars} \ ul \ \text{getM stateB}] \\
\text{by auto}
\end{aligned}
\]

moreover

\[
\begin{aligned}
\text{have} \ \text{InvariantVarsF} (\text{getF stateB}) \ F0 \ \text{decisionVars} \\
\text{using} \ (\text{getF stateB} = \text{getF stateA}) \\
\quad (\text{InvariantVarsF} (\text{getF stateA}) \ F0 \ \text{decisionVars}) \\
\text{by simp}
\end{aligned}
\]

moreover

\[
\begin{aligned}
\text{have} \ \text{InvariantConsistent} (\text{getM stateB}) \\
\text{using} \ (\text{InvariantConsistent} (\text{getM stateA})) \\
\quad (\text{isUnitClause} \ uc \ ul \ (\text{elements} (\text{getM stateA}))) \\
\quad (\text{getM stateB} = \text{getM stateA} \uplus [(ul, False)]) \\
\quad \text{InvariantConsistentAfterUnitPropagate} \ [\text{of getM stateA} \ uc \ ul \ \text{getM stateB}] \\
\text{by simp}
\end{aligned}
\]

moreover

\[
\begin{aligned}
\text{have} \ \text{InvariantUniq} (\text{getM stateB}) \\
\text{using} \ (\text{InvariantUniq} (\text{getM stateA})) \\
\quad (\text{isUnitClause} \ uc \ ul \ (\text{elements} (\text{getM stateA}))) \\
\quad (\text{getM stateB} = \text{getM stateA} \uplus [(ul, False)]) \\
\quad \text{InvariantUniqAfterUnitPropagate} \ [\text{of getM stateA} \ uc \ ul \ \text{getM stateB}] \\
\text{by simp}
\end{aligned}
\]

moreover

\[
\begin{aligned}
\text{have} \ \text{InvariantReasonClauses} (\text{getF stateB}) \ (\text{getM stateB}) \\
\text{using} \ (\text{getF stateB} = \text{getF stateA}) \\
\quad (\text{InvariantReasonClauses} (\text{getF stateA}) \ (\text{getM stateA}))
\end{aligned}
\]

271
isUnitClause uc ul (elements (getM stateA))
(getM stateB = getM stateA @ [(ul, False)])
formulaEntailsClause (getF stateA) uc
InvariantReasonClausesAfterUnitPropagate[of getF stateA getM stateA uc ul getM stateB]
by simp
moreover
have InvariantEquivalent F0 (getF stateB)
using (getF stateB = getF stateA)
by simp
moreover
have InvariantCFalse (getConflictFlag stateB) (getM stateB) (getC stateB)
using (getM stateB = getM stateA @ [(ul, False)])
(getConflictFlag stateB = getConflictFlag stateA)
(getC stateB = getC stateA)
by simp
moreover
have InvariantCEntailed (getConflictFlag stateB) (getF stateB) (getC stateB)
using (getF stateB = getF stateA)
(getConflictFlag stateB = getConflictFlag stateA)
(getC stateB = getC stateA)
by simp
ultimately
have ?thesis
unfolding invariantsHoldInState-def
by auto
}
moreover
{
assume appliedConflict stateA stateB
then obtain clause::Clause where
getConflictFlag stateA = False
formulaEntailsClause (getF stateA) clause
clauseFalse clause (elements (getM stateA))
getF stateB = getF stateA
getM stateB = getM stateA
getConflictFlag stateB = True
gtc stateB = clause
unfolding appliedConflict-def
by auto
}
have \texttt{InvariantVarsM} (\texttt{getM stateB}) \texttt{F0 decisionVars} using \texttt{InvariantVarsM} (\texttt{getM stateA}) \texttt{F0 decisionVars}
\texttt{getM stateB = getM stateA}
by simp
moreover
have \texttt{InvariantVarsF} (\texttt{getF stateB}) \texttt{F0 decisionVars} using \texttt{InvariantVarsF} (\texttt{getF stateA}) \texttt{F0 decisionVars}
\texttt{getF stateB = getF stateA}
by simp
moreover
have \texttt{InvariantConsistent} (\texttt{getM stateB}) using \texttt{InvariantConsistent} (\texttt{getM stateA})
\texttt{getM stateB = getM stateA}
by simp
moreover
have \texttt{InvariantUniq} (\texttt{getM stateB}) using \texttt{InvariantUniq} (\texttt{getM stateA})
\texttt{getM stateB = getM stateA}
by simp
moreover
have \texttt{InvariantReasonClauses} (\texttt{getF stateB}) (\texttt{getM stateB}) using \texttt{InvariantReasonClauses} (\texttt{getF stateA}) (\texttt{getM stateA})
\texttt{getF stateB = getF stateA}
\texttt{getM stateB = getM stateA}
by simp
moreover
have \texttt{InvariantEquivalent F0} (\texttt{getF stateB}) using \texttt{InvariantEquivalent F0} (\texttt{getF stateA})
\texttt{getF stateB = getF stateA}
by simp
moreover
have \texttt{InvariantCEntailed} (\texttt{getConflictFlag stateB}) (\texttt{getM stateB}) (\texttt{getC stateB}) using
\texttt{clauseFalse clause (elements (getM stateA))}
\texttt{getM stateB = getM stateA}
\texttt{getConflictFlag stateB = True}
\texttt{getC stateB = clause}
unfolding \texttt{InvariantCEntailed-def}
by simp
moreover
have \texttt{InvariantCFalse} (\texttt{getConflictFlag stateB}) (\texttt{getM stateB}) (\texttt{getC stateB}) unfolding \texttt{InvariantCFalse-def}
by simp
moreover
have \texttt{InvariantCEntailed} (\texttt{getConflictFlag stateB}) (\texttt{getF stateB}) (\texttt{getC stateB}) unfolding \texttt{InvariantCEntailed-def}
using
\texttt{getConflictFlag stateB = True}
\texttt{formulaEntailsClause (getF stateA) clause}
\texttt{getF stateB = getF stateA}
\langle \text{getC stateB = clause} \rangle \\
\text{by simp} \\
\text{ultimately have ?thesis} \\
\text{unfolding invariantsHoldInState-def} \\
\text{by auto} \\
\rangle \\
\text{moreover} \\
\{ \\
\text{assume appliedExplain stateA stateB} \\
\text{then obtain } l::\text{Literal and reason::Clause where} \\
\text{getConflictFlag stateA = True} \\
\text{l el getC stateA} \\
\text{formulaEntailsClause (getF stateA) reason} \\
\text{isEqual reason (opposite l) (elements (getM stateA))} \\
\text{getF stateB = getF stateA} \\
\text{getM stateB = getM stateA} \\
\text{getConflictFlag stateB = True} \\
\text{getC stateB = resolve (getC stateA) reason l} \\
\text{unfolding appliedExplain-def} \\
\text{by auto} \\
\text{have InvariantVarsM (getM stateB) F0 decisionVars} \\
\text{using (InvariantVarsM (getM stateA) F0 decisionVars) \\
\langle getM stateB = getM stateA \rangle} \\
\text{by simp} \\
\text{moreover} \\
\text{have InvariantVarsF (getF stateB) F0 decisionVars} \\
\text{using (InvariantVarsF (getF stateA) F0 decisionVars) \\
\langle getF stateB = getF stateA \rangle} \\
\text{by simp} \\
\text{moreover} \\
\text{have InvariantConsistent (getM stateB) \\
using (getM stateB = getM stateA) \\
\langle InvariantConsistent (getM stateA) \rangle} \\
\text{by simp} \\
\text{moreover} \\
\text{have InvariantUniq (getM stateB) \\
using (getM stateB = getM stateA) \\
\langle InvariantUniq (getM stateA) \rangle} \\
\text{by simp} \\
\text{moreover} \\
\text{have InvariantReasonClauses (getF stateB) (getM stateB) \\
using (getF stateB = getF stateA) \\
\langle getM stateB = getM stateA \rangle \\
\langle InvariantReasonClauses (getF stateA) (getM stateA) \rangle} \\
\text{274}
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 stateA)
  using
  ⟨InvariantCFalse (getConflictFlag stateA) (getM stateA) (getC stateA)⟩
  ⟨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 stateA)
  using
  ⟨InvariantCEntailed (getConflictFlag stateA) (getF stateA) (getC stateA)⟩
  ⟨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
\[\neg getC\ \text{stateA} \land\ \neg getC\ \text{stateB}\]
\[\text{getF\ stateB} = \text{getF\ stateA} \land [\text{getC\ stateA}]\]
\[\text{getM\ stateB} = \text{getM\ stateA}\]
\[\text{getConflictFlag\ stateB} = \text{True}\]
\[\text{getC\ stateB} = \text{getC\ stateA}\]

- **unfolding** appliedLearn-def
  - by auto

- **from** \(\text{getConflictFlag\ stateA} = \text{True}\) \(\land\) \(\text{InvariantC\Entailed\ (getConflictFlag\ stateA)\ (getF\ stateA)\ (getC\ stateA)\ (getM\ stateA)}\)
  - **have** formulaEntailsClause (getF stateA) (getC stateA)
    - **unfolding** InvariantCEntailed-def
      - by simp
  - **have** InvariantVarsM (getM stateB F0 decisionVars)
    - using (InvariantVarsM (getM stateA F0 decisionVars)
      - \(\land\) \(\text{getM\ stateB} = \text{getM\ stateA}\)
      - by simp
    - moreover
  - **from** \(\text{InvariantC\False\ (getConflictFlag\ stateA)\ (getM\ stateA)\ (getC\ stateA)\ (getM\ stateA) = \text{True}}\)
    - **have** clauseFalse (getC stateA) (elements (getM stateA))
      - **unfolding** InvariantCFalse-def
        - by simp
        - with (InvariantVarsM (getM stateA F0 decisionVars)
          - **have** \(\text{vars\ (getC\ stateA)} \subseteq \text{vars\ F0} \cup\ \text{decisionVars}\)
            - **unfolding** InvariantVarsM-def
              - using valuationContainsItsFalseClausesVariables [of getC stateA elements (getM stateA)]
                - by simp
                - hence InvariantVarsF (getF stateB F0 decisionVars)
                  - using \(\text{getF\ stateB} = \text{getF\ stateA} \land [\text{getC\ stateA}]\)
                    - InvariantVarsFAfterLearn [of getF stateA F0 decisionVars getC stateA getF stateB]
                      - by simp
                - moreover
  - **have** InvariantConsistent (getM stateB)
    - using (InvariantConsistent (getM stateA))
      - \(\land\) \(\text{getM\ stateB} = \text{getM\ stateA}\)
        - by simp
    - moreover
  - **have** InvariantUniq (getM stateB)
    - using (InvariantUniq (getM stateA))
      - \(\land\) \(\text{getM\ stateB} = \text{getM\ stateA}\)
        - by simp
    - moreover
  - **have** InvariantReasonClauses (getF stateB) (getM stateB)
    - using

276
\begin{verbatim}

\begin{proof}
\begin{enumerate}
\item \texttt{InvariantReasonClauses (getF stateA) (getM stateA)}
\item \texttt{formulaEntailsClause (getF stateA) (getC stateA)}
\item \texttt{getF stateB = getF stateA @ [getC stateA]}
\item \texttt{getM stateB = getM stateA}
\item \texttt{InvariantReasonClausesAfterLearn[of getF stateA getM stateA}
\item \texttt{getC stateA getF stateB]}
\item \texttt{by simp}
\item \texttt{moreover}
\item \texttt{have InvariantEquivalent F0 (getF stateB)}
\item \texttt{using}
\item \texttt{InvariantEquivalent F0 (getF stateA)}
\item \texttt{formulaEntailsClause (getF stateA) (getC stateA)}
\item \texttt{getF stateB = getF stateA @ [getC stateA]}
\item \texttt{InvariantEquivalentAfterLearn[of F0 getF stateA getC stateA getF stateB]}
\item \texttt{by simp}
\item \texttt{moreover}
\item \texttt{have InvariantCFalse (getConflictFlag stateB) (getM stateB) (getC stateB)}
\item \texttt{using}
\item \texttt{InvariantCFalse (getConflictFlag stateA) (getM stateA) (getC stateA)}
\item \texttt{getM stateB = getM stateA}
\item \texttt{getConflictFlag stateA = True} 
\item \texttt{getConflictFlag stateB = True} 
\item \texttt{getM stateB = getM stateA}
\item \texttt{getC stateB = getC stateA}
\item \texttt{by simp}
\item \texttt{moreover}
\item \texttt{have InvariantCEntailed (getConflictFlag stateB) (getF stateB) (getC stateB)}
\item \texttt{using}
\item \texttt{InvariantCEntailed (getConflictFlag stateA) (getF stateA) (getC stateA)}
\item \texttt{formulaEntailsClause (getF stateA) (getC stateA)}
\item \texttt{getF stateB = getF stateA @ [getC stateA]}
\item \texttt{getConflictFlag stateA = True} 
\item \texttt{getConflictFlag stateB = True} 
\item \texttt{getC stateB = getC stateA}
\item \texttt{InvariantCEntailedAfterLearn[of getConflictFlag stateA getF stateA getC stateA getF stateB]}
\item \texttt{by simp}
\item \texttt{ultimately}
\item \texttt{have ?thesis}
\item \texttt{unfolding invariantsHoldInState-def}
\item \texttt{by auto}
\end{enumerate}
\end{proof}
\end{verbatim}

277
then obtain \( l::\text{Literal} \) and \( \text{level}::\text{nat} \)

where

\[
\begin{align*}
\text{getConflictFlag stateA} &= \text{True} \\
\text{isBackjumpLevel level l (getC stateA) (getM stateA)} \\
\text{getF stateB} &= \text{getF stateA} \\
\text{getM stateB} &= \text{prefixToLevel level (getM stateA)} @ [(l, \text{False})] \\
\text{getConflictFlag stateB} &= \text{False} \\
\text{getC stateB} &= [] \\
\text{unfolding} \text{ appliedBackjump-def}
\end{align*}
\]

by auto

with (\text{InvariantConsistent (getM stateA)} \cdot \text{InvariantUniq (getM stateA)})

\[
\begin{align*}
\text{(InvariantCFalse (getConflictFlag stateA) (getM stateA) (getC stateA))} \\
have \text{isUnitClause (getC stateA) l (elements (prefixToLevel level (getM stateA)))}
\end{align*}
\]

\text{unfolding} \text{InvariantUniq-def}

\text{unfolding} \text{InvariantConsistent-def}

\text{unfolding} \text{InvariantCFalse-def}

using \text{isBackjumpLevelEnsuresIsUnitInPrefix[of getM stateA getC stateA level l]}

by simp

from (\text{getConflictFlag stateA} = \text{True}) \cdot \text{InvariantCEntailed (getConflictFlag stateA) (getF stateA) (getC stateA)}

have \text{formulaEntailsClause (getF stateA) (getC stateA)}

\text{unfolding} \text{InvariantCEntailed-def}

by simp

from (\text{isBackjumpLevel level l (getC stateA) (getM stateA)})

have \text{isLastAssertedLiteral (opposite l) (oppositeLiteralList (getC stateA)) (elements (getM stateA))}

\text{unfolding} \text{isBackjumpLevel-def}

by simp

hence \( l \in \text{getC stateA} \)

\text{unfolding} \text{isLastAssertedLiteral-def}

using \text{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}) \cdot \text{InvariantCEntailed (getConflictFlag stateA) (getF stateA) (getC stateA)}

have \text{formulaEntailsClause (getF stateA) (getC stateA)}

\text{unfolding} \text{InvariantCEntailed-def}

by simp
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 getC stateA elements (getM stateA)]
  by simp
moreover
from (l el getC stateA)
have var l ∈ vars (getC stateA)
  using clauseContainsItsLiteralsVariable[of l getC stateA]
  by simp
ultimately
have vars (getC stateA) ⊆ vars F0 ∪ decisionVars
  using (InvariantVarsM (getM stateA) F0 decisionVars)
unfolding InvariantVarsM-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)])
InvartiantVarsMAfterBackjump[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)])
InvartiantConsistentAfterBackjump[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[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[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 invariantHoldInState-def
  by auto
}
ultimately
show ?thesis
  using (transition stateA stateB F0 decisionVars)
The consequence is that invariants hold in all valid runs.

lemma invariantsHoldInValidRuns:
  fixes \( F_0 \) :: Formula and decisionVars :: Variable set
  assumes invariantsHoldInState stateA \( F_0 \) decisionVars and 
  \((stateA, stateB) \in \text{transitionRelation} \( F_0 \) decisionVars\)
  shows invariantsHoldInState stateB \( F_0 \) decisionVars
using assms
using transitionsPreserveInvariants
using rtrancl-induct[of stateA stateB
  \{(stateA, stateB), transition stateA stateB \( F_0 \) decisionVars\} \(\lambda x.\)
  invariantsHoldInState x \( F_0 \) decisionVars]
unfolding transitionRelation-def
by auto

lemma invariantsHoldInValidRunsFromInitialState:
  fixes \( F_0 \) :: Formula and decisionVars :: Variable set
  assumes isInitialState state0 \( F_0 \)
  and \((state0, state) \in \text{transitionRelation} \( F_0 \) decisionVars\)
  shows invariantsHoldInState state \( F_0 \) decisionVars
proof–
  from \( \langle \text{isInitialState} \text{ state0} \, \text{ F0} \rangle \)
  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 \( \text{getC} \text{ state} = [] \).
2. SAT states where \( \text{getConflictFlag} \text{ state} = \text{False} \), \( \neg \text{formulaFalse} \text{ state} = F_0 \) (elements \( \text{getM} \text{ state} \)) and \( \text{decisionVars} \subseteq \text{vars} \) (elements \( \text{getM} \text{ state} \)).

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 $F_0 :: \text{Formula and } \text{decisionVars :: Variable set and } \text{state0 :: State and state :: State}$

assumes

isInitialState state0 $F_0$ and

$(state0, state) \in \text{transitionRelation } F_0 \text{ decisionVars}$

getConflictFlag state = True and

gtC state = []

shows $\neg \text{satisfiable } F_0$

**Proof**—

from ⟨isInitialState state0 $F_0$⟩ ⟨(state0, state) ∈ transitionRelation $F_0$ decisionVars⟩

have invariantsHoldInState state $F_0$ decisionVars

using invariantsHoldInValidRunsFromInitialState by simp

hence

InvariantEquivalent $F_0$ (gtF state)

InvariantCEntailed (getConflictFlag state) (gtF state) (gtC state)

unfolding invariantsHoldInState-def by auto

with ⟨getConflictFlag state = True⟩ ⟨gtC state = []⟩

show ?thesis

by (simp add: unsatReportExtensiveExplain)

**Qed**

**Theorem** soundnessForSAT:

fixes $F_0 :: \text{Formula and } \text{decisionVars :: Variable set and } \text{state0 :: State and state :: State}$

assumes

vars $F_0 \subseteq \text{decisionVars and}$

isInitialState state0 $F_0$ and

$(state0, state) \in \text{transitionRelation } F_0 \text{ decisionVars and}$

getConflictFlag state = False

$\neg \text{formulaFalse } (gtF state) (\text{elements } (gtM state))$

vars (elements (gtM state)) $\supseteq \text{decisionVars}$

shows

model (elements (gtM state)) $F_0$

**Proof**—

from ⟨isInitialState state0 $F_0$⟩ ⟨(state0, state) ∈ transitionRelation $F_0$ decisionVars⟩

have invariantsHoldInState state $F_0$ 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 ∪ decision-Vars)\}
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
**lemmas**: Invariants ensure that trails are indeed uniq, consistent and with finite variable sets.

**lemma trailIsDecreasedByDecidedUnitPropagateAndBackjump**:

- **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**

1. **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

2. **from** (invariantsHoldInState stateB 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

3. **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

4. **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

5. **moreover**

6. **assume** appliedUnitPropagate stateA stateB F0 decisionVars
   hence (getM stateB, getM stateA) ∈ lexLess
   unfolding appliedUnitPropagate-def
have \((\text{getM stateB}, \text{getM stateA}) \in \text{lexLessRestricted \ (vars F0 \cup decisionVars)}\)

unfolding \text{lexLessRestricted-def}

by auto

moreover

assume \text{appliedBackjump stateA stateB}

then obtain \(l::\text{Literal \ and \ level::nat}\)

where

\(\text{getConflictFlag stateA} = \text{True}\)
\(\text{isBackjumpLevel level l (getC stateA) (getM stateA)}\)
\(\text{getF stateB} = \text{getF stateA}\)
\(\text{getM stateB} = \text{prefixToLevel level (getM stateA) \@ \((l, \text{False})\)}\)
\(\text{getConflictFlag stateB} = \text{False}\)
\(\text{getC stateB} = []\)

unfolding \text{appliedBackjump-def}

by auto

from \(\text{(isBackjumpLevel level l (getC stateA) (getM stateA))}\)

have \(\text{isLastAssertedLiteral (opposite l) (oppositeLiteralList (getC stateA)) (elements (getM stateA))}\)

unfolding \text{isBackjumpLevel-def}

by simp

hence \((\text{opposite l}) \text{ el elements (getM stateA)}\)

unfolding \text{isLastAssertedLiteral-def}

by simp

hence \(\text{elementLevel (opposite l) (getM stateA) \leq currentLevel (getM stateA)}\)

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

moreover

from \(\text{(isBackjumpLevel level l (getC stateA) (getM stateA))}\)

have \(\theta \leq \text{level \ and \ level < elementLevel (opposite l) (getM stateA)}\)

unfolding \text{isBackjumpLevel-def}

using \(\text{(isLastAssertedLiteral (opposite l) (oppositeLiteralList (getC stateA)) (elements (getM stateA)))}\)

by auto

ultimately

have \(\text{level < currentLevel (getM stateA)}\)

by simp

with \(\theta \leq \text{level} \leadsto \text{getM stateB = prefixToLevel level (getM stateA)} \@ \((l, \text{False})\))\)

have \((\text{getM stateB, getM stateA}) \in \text{lexLess}\)

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

with * **

have \((\text{getM stateB, getM stateA}) \in \text{lexLessRestricted \ (vars F0 \cup decisionVars)}\)
Next we show that \textit{Conflict} decreases the conflict flag in the \textit{boolLess} ordering.

\textbf{lemma \textit{conflictFlagIsDecreasedByConflict}:}
\begin{itemize}
\item \textbf{fixes} stateA::State and stateB::State
\item \textbf{assumes} appliedConflict stateA stateB
\item \textbf{shows} getM stateA = getM stateB and (getConflictFlag stateB, getConflictFlag stateA) ∈ boolLess
\end{itemize}
\textbf{proof—}
\begin{itemize}
\item \textbf{from} (appliedConflict stateA stateB)
\item \textbf{obtain} l::Literal and reason::Clause where
\begin{itemize}
\item getConflictFlag stateA = True
\item el (getC stateA)
\item isReason reason (opposite l) (elements (getM stateA))
\item getF stateB = getF stateA
\item getM stateB = getM stateA
\item getConflictFlag stateB = True
\item getC stateB = resolve (getC stateA) reason l
\end{itemize}
\item \textbf{unfolding} appliedConflict-def
\item \textbf{by} auto
\item \textbf{thus} getM stateA = getM stateB getConflictFlag stateA = getConflictFlag stateB (getC stateB, getC stateA) ∈ boolLess (getM stateA)
\item \textbf{using} multLessResolve[\text{of opposite l getC stateA reason getM stateA}]
\item \textbf{by} auto
\end{itemize}
\textbf{qed}
Finally, we show that Learn decreases the formula in the learn-Less formula ordering.

**Lemma** `formulaIsDecreasedByLearn`:

- **Fixes** `stateA::State` and `stateB::State`
- **Assumes** `appliedLearn stateA stateB`
- **Shows**
  - `getM stateA = getM stateB`
  - `getConflictFlag stateA = getConflictFlag stateB`
  - `getC stateA = getC stateB`
  - `(getF stateB, getF stateA) ∈ learnLess (getC stateA)`

**Proof**—

- **From** `appliedLearn stateA stateB`
- **Have**
  - `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`
- **Unfolding** `appliedLearn-def`
- **By** `auto`
- **Thus**
  - `getM stateA = getM stateB`
  - `getConflictFlag stateA = getConflictFlag stateB`
  - `getC stateA = getC stateB`
  - `(getF stateB, getF stateA) ∈ learnLess (getC stateA)`
- **Unfolding** `learnLess-def`
- **By** `auto`

**QED**

Now we can prove 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**—

- **Assume** `appliedDecide stateA stateB decisionVars` or `appliedUnit-Propagate stateA stateB F0 decisionVars` or `appliedBackjump stateA stateB`
  - **With** `invariantsHoldInState stateA F0 decisionVars`
  - **Have** `(getM stateB, getM stateA) ∈ lexLessRestricted (vars F0 ∪ decisionVars)`
  - **Using** `trailIsDecreasedByDeciedUnitPropagateAndBackjump`
  - **By** `simp`
  - **Hence** `(stateB, stateA) ∈ lexLessState F0 decisionVars`
- **Unfolding** `lexLessState-def`
by simp
hence \((\text{stateB}, \text{stateA}) \in \text{terminationLess } F0 \text{ decisionVars}\)
unfolding terminationLess-def
by simp
}
moreover
{
assume appliedConflict stateA stateB
hence \(\text{getM stateA} = \text{getM stateB} (\text{getConflictFlag stateB, getConflictFlag stateA})\) \(\in \text{boolLess}\)
  using conflictFlagIsDecreasedByConflict
by auto
hence \((\text{stateB}, \text{stateA}) \in \text{boolLessState}\)
unfolding boolLessState-def
by simp
hence \((\text{stateB}, \text{stateA}) \in \text{terminationLess } F0 \text{ decisionVars}\)
unfolding terminationLess-def
by simp
}
moreover
{
assume appliedExplain stateA stateB
hence \(\text{getM stateA} = \text{getM stateB}\)
  \(\text{getConflictFlag stateA} = \text{getConflictFlag stateB}\)
  \(\text{(getC stateB, getC stateA)} \in \text{multLess (getM stateA)}\)
  using conflictClauseIsDecreasedByExplain
by auto
hence \((\text{stateB}, \text{stateA}) \in \text{multLessState}\)
unfolding multLessState-def
unfolding multLess-def
by simp
hence \((\text{stateB}, \text{stateA}) \in \text{terminationLess } F0 \text{ decisionVars}\)
unfolding terminationLess-def
by simp
}
moreover
{
assume appliedLearn stateA stateB
hence \(\text{getM stateA} = \text{getM stateB}\)
  \(\text{getConflictFlag stateA} = \text{getConflictFlag stateB}\)
  \(\text{getC stateA} = \text{getC stateB}\)
  \(\text{(getF stateB, getF stateA)} \in \text{learnLess (getC stateA)}\)
  using formulaIsDecreasedByLearn
by auto
hence \((\text{stateB}, \text{stateA}) \in \text{learnLessState}\)
unfolding learnLessState-def
by simp
hence \((\text{stateB}, \text{stateA}) \in \text{terminationLess } F0 \text{ decisionVars}\)
}
The minimal states with respect to the termination ordering are final i.e., no further transition rules are applicable.

**definition**

\[ \text{isMinimalState} \text{ stateMin } F0 \text{ decisionVars } \equiv (\forall \text{ state }:\text{State} \cdot (\text{state}, \text{stateMin}) \notin \text{terminationLess } F0 \text{ decisionVars}) \]

**lemma** minimalStatesAreFinal:

**fixes** stateA::State

**assumes**

\( \text{invariantsHoldInState state } F0 \text{ decisionVars } \text{ and isMinimalState state } F0 \text{ decisionVars} \)

**shows** isFinalState state F0 decisionVars

**proof**

\{ 
    **assume** \( \neg \text{?thesis} \)
    **then** obtain state':State
    **where** transition state state' F0 decisionVars
    **unfolding** isFinalState-def
    **by** auto
    **with** (invariantsHoldInState state F0 decisionVars)
    **have** (state', state) \(\notin\) 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 \( \text{wf} \) (lexLessState \( F_0 \) decisionVars)
unfolding \( \text{wf-eq-minimal} \)
proof
\[
\forall Q \text{ state. state } \in Q \rightarrow (\exists \text{stateMin} \in Q. \forall \text{state}' . (\text{state}', \text{stateMin}) \in \text{lexLessState} F_0 \text{ decisionVars} \rightarrow \text{state}' \notin Q)
\]
proof
\[
\{ \text{fix } Q :: \text{State set and state :: State}
\text{assume } \text{state} \in Q
\text{let } ?Q1 = \{M::\text{LiteralTrail. } \exists \text{ state. state } \in Q \land (\text{getM state}) = M\}
\text{from } \langle \text{state} \in Q \rangle
\text{have } \text{getM state} \in ?Q1
\text{by auto}
\text{from } \langle \text{finite decisionVars} \rangle
\text{have } \text{finite } \langle \text{vars } F_0 \cup \text{decisionVars} \rangle
\text{using finiteVarsFormula[of } F_0 \rangle
\text{by simp}
\text{hence } \text{wf (lexLessRestricted (vars } F_0 \cup \text{decisionVars))}
\text{using wfLexLessRestricted[of vars } F_0 \cup \text{decisionVars]}
\text{by simp}
\text{with } \langle \text{getM state} \in ?Q1 \rangle
\text{obtain Mmin where Mmin } \in ?Q1 \forall \text{M'}. (\text{M'}, \text{Mmin}) \in \text{lexLessRestricted (vars } F_0 \cup \text{decisionVars} \rightarrow \text{M'} \notin ?Q1
\text{unfolding } \text{wf-eq-minimal}
\text{apply (erule-tac } x=?Q1 \text{ in allE})
\text{apply (erule-tac } x=\text{getM state} \text{ in allE})
\text{by auto}
\text{from } \langle \text{Mmin } \in ?Q1 \rangle \text{ obtain stateMin}
\text{where stateMin } \in Q \text{ (getM stateMin) = Mmin}
\text{by auto}
\text{have } \forall \text{state}'. (\text{state}', \text{stateMin}) \in \text{lexLessState } F_0 \text{ decisionVars}
\rightarrow \text{state}' \notin Q
\text{proof}
\text{fix state'}
\text{show } (\text{state}', \text{stateMin}) \in \text{lexLessState } F_0 \text{ decisionVars} \rightarrow \text{state}' \notin Q
\text{proof}
\text{assume } (\text{state}', \text{stateMin}) \in \text{lexLessState } F_0 \text{ decisionVars}
\text{hence } \text{(getM state'), getM stateMin} \in \text{lexLessRestricted (vars } F_0 \cup \text{decisionVars)}
\text{unfolding } \text{lexLessState-def}
\text{by auto}
\text{from } \forall \text{M'}. (\text{M'}, \text{Mmin}) \in \text{lexLessRestricted (vars } F_0 \cup \text{decisionVars} \rightarrow \text{M'} \notin ?Q1,
\langle \text{(getM state'), getM stateMin} \in \text{lexLessRestricted (vars } F_0 \cup \text{decisionVars)} \rangle \text{(getM stateMin) = Mmin)
\text{have } \text{getM state'} \notin ?Q1
\]
by simp
with (getM stateMin = Mmin)
show state' \notin Q
  by auto
qed
qed
with (stateMin \in Q)
have \exists stateMin \in Q. (\forall state'. (state', stateMin) \in lexLessState
  F0 decisionVars \to state' \notin Q)
  by auto
}
thus ?thesis
  by auto
qed

lemma wfBoolLessState:
  shows wf boolLessState
unfolding wf-eq-minimal
proof-
  show \forall Q state. state \in Q \to (\exists stateMin \in Q. \forall state'. (state',
    stateMin) \in boolLessState \to state' \notin Q)
    proof-
    {
    fix Q :: State set and state :: State
    assume state \in Q
    let ?M = (getM state)
    let ?Q1 = \{b::bool. \exists state. state \in Q \land (getM state) = ?M \land
      (getConflictFlag state) = b\}
    from (state \in Q)
    have getConflictFlag state \in ?Q1
      by auto
    with wfBoolLess
    obtain bMin where bMin \in \?Q1 \forall b'. (b', bMin) \in boolLess \to
      b' \notin \?Q1
      unfolding wf-eq-minimal
      apply (erule-tac x=?Q1 in allE)
      apply (erule-tac x=getConflictFlag state in allE)
      by auto
    from bMin \in ?Q1 obtain stateMin
      where stateMin \in Q (getM stateMin) = ?M getConflictFlag
    stateMin = bMin
      by auto
    have \forall state'. (state', stateMin) \in boolLessState \to state' \notin Q
      proof
      fix state'
      show (state', stateMin) \in boolLessState \to state' \notin Q
        proof
        assume (state', stateMin) \in boolLessState
}
with \( \langle \text{getM stateMin} = ?M \rangle \)
have \( \text{getM state'} = \text{getM stateMin} \quad \text{getConflictFlag stateMin} \in \text{boolLess} \)
unfolding boolLessState-def
by auto
from \( \forall b'. (b', b\text{Min}) \in \text{boolLess} \longrightarrow b' \notin ?Q1 \rangle \)
\( \langle \text{getConflictFlag state'}, \text{getConflictFlag stateMin} \rangle \in \text{boolLess} \)
\( \langle \text{getConflictFlag stateMin} = b\text{Min} \rangle \)
have \( \text{getConflictFlag state'} \notin ?Q1 \)
by simp
with \( \langle \text{getM state'} = \text{getM stateMin} \rangle \langle \text{getM stateMin} = ?M \rangle \)
show \( \text{state'} \notin Q \)
by auto
qed

with \( \langle \text{stateMin} \in Q \rangle \)
have \( \exists \text{stateMin} \in Q. (\forall \text{state}'. (\text{state'}, \text{stateMin}) \in \text{boolLessState} \rightarrow \text{state'} \notin Q) \)
by auto
}
thus ?thesis
by auto
qed

lemma wfMultLessState:
shows \( \text{wf multLessState} \)
unfolding wf-eq-minimal
proof–
show \( \forall Q \quad \text{state. state} \in Q \longrightarrow (\exists \text{stateMin} \in Q. \forall \text{state}'. (\text{state'}, \text{stateMin}) \in \text{multLessState} \rightarrow \text{state'} \notin Q) \)
proof–
{  
fix \( Q : State \quad \text{set and state} : State \)
assume \( \text{state} \in Q \)
let \( ?M = (\text{getM state}) \)
let \( ?Q1 = \{ C::\text{Clause. } \exists \text{ state} \in Q \land (\text{getM state}) = ?M \land (\text{getC state}) = C \} \)
from \( \text{state} \in Q \)
have \( \text{getC state} \in ?Q1 \)
by auto
with \( \text{wfMultLess}[of ?M] \)
obtain \( C\text{min} \quad \text{where C\text{min} } \in ?Q1 \forall \text{ C'}. (\text{C'}, \text{C\text{min}}) \in \text{multLess} \)
\( ?M \longrightarrow C' \notin ?Q1 \)
unfolding wf-eq-minimal
apply (erule-tac \( x=\text{?Q1} \) in allE)
apply (erule-tac \( x=\text{getC state} \) in allE)
by auto
from \( \text{C\text{min} } \in ?Q1 \) obtain \( \text{stateMin} \)
where \( \text{stateMin} \in Q \) \((\text{getM stateMin}) = \text{?M}\) \(\text{getC stateMin} = \text{Cmin}\)

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 \((\text{getM stateMin} = \text{?M})\)

have \(\text{getM state}' = \text{getM stateMin} \) \((\text{getC state}', \text{getC stateMin}) \in \text{multLess ?M} \) \(\text{getM stateMin} = \text{Cmin}\)

by simp

with \((\text{getM state'} = \text{getM stateMin}) \) \((\text{getM stateMin} = \text{?M})\)

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

}\)

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 \(Q :: \text{State set and state :: State}\)

assum \(\text{state} \in Q\)

let \(\text{?M} = (\text{getM state})\)

let \(\text{?C} = (\text{getC state})\)

let \(\text{?conflictFlag} = (\text{getConflictFlag state})\)

let \(\text{?Q1} = (F::\text{Formula}. \exists \text{state}. \text{state} \in Q \wedge (\text{getM state}) = \text{?M} \wedge (\text{getConflictFlag state}) = \text{?conflictFlag})\)

293
∧ (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 getConfFlag 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) (getConfFlag stateMin = ?conflictFlag)
have getM state' = getM stateMin getC state' = getC stateMin
getConfFlag state' = getConfFlag stateMin (getC state', getF stateMin) ∈ learnLess ?C
unfolding learnLessState-def
by auto
from ∀ F'. (F', Fmin) ∈ learnLess ?C → F' ∉ ?Q1
(∀ state'. (state', stateMin) ∈ learnLess ?C) (getF stateMin = Fmin)
have getF state' ∉ ?Q1
by simp
with (getM state' = getM stateMin) (getC state' = getC stateMin) (getConfFlag state' = getConfFlag stateMin)
(getM stateMin = ?M) (getC stateMin = ?C) (getConfFlag stateMin = ?conflictFlag) (getF stateMin = Fmin)
show state' ∉ Q
by auto
qed
qed
have ∃ stateMin ∈ Q
∧ (getM stateMin ∈ Q) ∧ (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** `wfTerminationLess`:
- **fixes** `decisionVars::Variable set` and `F0::Formula`
- **assumes** `finite decisionVars`
- **shows** `wf (terminationLess F0 decisionVars)`

**unfolding** `wf-eq-minimal`

**proof**
- show `∀ Q state. state ∈ Q → (∃ stateMin ∈ Q. ∀ state'. (state', stateMin) ∈ terminationLess F0 decisionVars → state' /∈ Q)

**proof**
- \{  
  fix `Q::State set`
  fix `state::State`
  assume `state ∈ Q`

  from `⟨finite decisionVars⟩`
  have `wf (lexLessState F0 decisionVars)`
  using `wfLexLessState[of decisionVars F0]`
  by simp

  with `state ∈ Q` obtain `state0`
  where `state0 ∈ Q ∀ state'. (state', state0) ∈ lexLessState F0 decisionVars → state' /∈ Q`
  unfolding `wf-eq-minimal`
  by auto

  let `?Q0 = {state. state ∈ Q ∧ (getM state) = (getM state0)}`
  from `⟨state0 ∈ Q⟩`
  have `state0 ∈ ?Q0`
  by simp

  have `wf boolLessState`
  using `wfBoolLessState`

  with `state0 ∈ Q` obtain `state1`
  where `state1 ∈ ?Q0 ∀ state'. (state', state1) ∈ boolLessState → state' /∈ ?Q0`
  unfolding `wf-eq-minimal`
  apply (erule-tac `x=?Q0 in allE`)
  apply (erule-tac `x=state0 in allE`)
  by auto

  let `?Q1 = {state. state ∈ Q ∧ getM state = getM state0 ∧ getConflictFlag state = getConflictFlag state1}`
  from `⟨state1 ∈ ?Q0⟩`
  have `state1 ∈ ?Q1`
  by simp

  have `wf multLessState`
using wfMultLessState

with (state1 ∈ ?Q1) obtain state2
  where state2 ∈ ?Q1 ∀ state'. (state', state2) ∈ multLessState

→ state' ∉ ?Q1
  unfolding wf-eq-minimal
  apply (erule-tac x=state1 in allE)
  apply (erule-tac x=state2 in allE)
  by auto

let ?Q2 = {state. state ∈ Q ∧ getM state = getM state0 ∧
               getConflictFlag state = getConflictFlag state1 ∧
               getC state = getC state2
         } from ⟨state2 ∈ ?Q1⟩

have state2 ∈ ?Q2 by simp

have wf learnLessState
  using wfLearnLessState

  with (state2 ∈ ?Q2) obtain state3
    where state3 ∈ ?Q2 ∀ state'. (state', state3) ∈ learnLessState

→ state' ∉ ?Q2
  unfolding wf-eq-minimal
  apply (erule-tac x=?Q2 in allE)
  apply (erule-tac x=state2 in allE)
  by auto

from ⟨state3 ∈ ?Q2⟩
have state3 ∈ Q by simp

from ⟨state1 ∈ ?Q0⟩
have getM state1 = getM state0 by simp

from ⟨state2 ∈ ?Q1⟩
have getM state2 = getM state0 getConflictFlag state2 = getConflictFlag state1
  by auto

from ⟨state3 ∈ ?Q2⟩
have getM state3 = getM state0 getConflictFlag state3 = getConflictFlag state1
cstate3 = getC state3
  by auto

let ?stateMin = state3
have ∀ state'. (state', ?stateMin) ∈ terminationLess F0 decisionVars → state' ∉ Q
  proof
    fix state'
    show (state', ?stateMin) ∈ terminationLess F0 decisionVars
      → state' ∉ Q
      proof
        assume (state', ?stateMin) ∈ terminationLess F0 decisionVars
        hence

296
(state', stateMin) ∈ lexLessState F0 decisionVars ∨
(state', stateMin) ∈ boolLessState ∨
(state', stateMin) ∈ mulLessState ∨
(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
    → 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) ∈ mulLessState
  from (∀ stateMin ∈ ?Q2: (getM state1 = getM state0: (getM state2 = getM state0)
    (getConflictFlag state2 = getConflictFlag state1)

297
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
}
morerover
{
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
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 wfTransitionRelation:
  fixes decisionVars :: Variable set and F0 :: Formula
  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 stateA::State and stateB::State
        assume (stateB, stateA) ∈ ?rel
        hence (stateB, stateA) ∈ ?rel'
          using ⟨isInitialState state0 F0, stateA decisionVars⟩
          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);
    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 intial state to the final one.

corollary

by simp
qed
qed
fixes decisionVars :: Variable set and F0 :: Formula and state0 :: State
assumes finite decisionVars and isInitialState state0 F0
shows ∃ state. (state0, state) ∈ transitionRelation F0 decisionVars ∧ isFinalState state F0 decisionVars
proof
{
  assume ¬ ?thesis
  let ?Q = {state. (state0, state) ∈ transitionRelation F0 decisionVars}
  let ?rel = {(stateB, stateA). (state0, stateA) ∈ transitionRelation F0 decisionVars ∧ transition stateA stateB F0 decisionVars} ∧ transitionRelF0 decisionVars
  have state0 ∈ ?Q
    unfolding transitionRelation-def
    by simp
  hence ∃ state. state ∈ ?Q
    by auto
  from assms
  have wf ?rel
    using wfTransitionRelation[of decisionVars state0 F0]
    by auto
  hence ∨ Q. (∃ x. x ∈ Q) → (∃ stateMin ∈ Q. ∀ state. (state, stateMin) ∈ ?rel → state /∈ Q)
    unfolding wf-eq-minimal
    by simp
  hence (∃ x. x ∈ ?Q) → (∃ stateMin ∈ ?Q. ∀ state. (state, stateMin) ∈ ?rel → state /∈ ?Q)
    by rule
  with (∃ state. state ∈ ?Q)
  have ∃ stateMin ∈ ?Q. ∀ state. (state, stateMin) ∈ ?rel → state /∈ ?Q
    by simp
  then obtain stateMin
    where stateMin ∈ ?Q and ∀ state. (state, stateMin) ∈ ?rel → state /∈ ?Q
    by auto
  from (stateMin ∈ ?Q)
  have (state0, stateMin) ∈ transitionRelation F0 decisionVars
    by simp
  with (¬ ?thesis)
  have ¬ isFinalState stateMin F0 decisionVars
    by simp
  then obtain state'::State
    where transition stateMin state' F0 decisionVars
    unfolding isFinalState-def
    by auto

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 form 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** $F_0 ::$ Formula and $\text{decisionVars} ::$ Variable set
- **assumes** finite decisionVars
- **shows** $\neg (\exists \ Q :: (\text{State set}). \exists \ state0 \in Q. \text{isInitialState} \ state0 \ F_0 \land$
  
  $\forall \ state \in Q. (\exists \ state' \in Q. \text{transition state state'} \ F_0 \text{ decisionVars})
  )$

**proof** –

- **assume** $\neg ?thesis$
- **then obtain** $Q :: \text{State set}$ and $\text{state0} :: \text{State}$
  
  where isInitialState state0 F0 state0 \in Q
  $\forall \ state \in Q. (\exists \ state' \in Q. \text{transition state state'} \ F_0 \text{ decisionVars})$
  
  by auto
- **let** $?rel = \{(stateB, stateA). (state0, stateA) \in \text{transitionRelation} F0 \text{ decisionVars} \land$

301
transition stateA stateB F0 decisionVars

from (finite decisionVars) ⟨isInitialState state0 F0⟩
have wf ?rel
  using wfTransitionRelation
  by simp
hence wfmin: ∀ x. 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⟩
  ⟨transition stateA stateB F0 decisionVars⟩
  ⟨transition stateA stateB 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
}
Thus, \textit{thesis} by force.

7.5 Completeness

In this section we will first show that each final state is either \textit{SAT} or \textit{UNSAT} state.

\textbf{lemma} finalNonConflictState:

\textbf{fixes} state::State and FO :: Formula
\textbf{assumes}
\textit{getConflictFlag} state = False \textbf{and} \\
\neg applicableDecide state decisionVars \textbf{and} \\
\neg applicableConflict state
\textbf{shows} \neg formulaFalse \ ((\textit{getF} state) \ (\textit{elements} \ (\textit{getM} state))) \textbf{and} \\
vars \ (\textit{elements} \ (\textit{getM} state)) \supseteq \textit{decisionVars}
\textbf{proof}–
\textbf{from} \neg applicableConflict state \ (\textit{getConflictFlag} state = False) \\
\textbf{show} \neg formulaFalse \ ((\textit{getF} state) \ (\textit{elements} \ (\textit{getM} state))) \\
\textbf{unfolding} applicableConflictCharacterization \\
\textbf{by} \ (\textit{auto simp add:formulaFalseIffContainsFalseClause formulaEntailsItsClauses}) \\
\textbf{show} \ vars \ (\textit{elements} \ (\textit{getM} state)) \supseteq \textit{decisionVars}
\textbf{proof}
\textbf{fix} x :: Variable
\textbf{let} \ ?l = Pos x \\
\textbf{assume} x \in \textit{decisionVars}
\textbf{hence} var \ ?l = x \textbf{and} \ var \ ?l \in \textit{decisionVars} \textbf{and} \ var \ (\textit{opposite} \ ?l) \in \textit{decisionVars}
\textbf{by} auto \\
\textbf{with} \neg applicableDecide state decisionVars \\
\textbf{have} literalTrue \ ?l \ (\textit{elements} \ (\textit{getM} state)) \lor \ literalFalse \ ?l \ (\textit{elements} \ (\textit{getM} state)) \\
\textbf{unfolding} applicableDecideCharacterization \\
\textbf{by} force \\
\textbf{with} \ (\textit{var} \ ?l = x) \\
\textbf{show} x \in \textit{vars} \ (\textit{elements} \ (\textit{getM} state)) \\
\textbf{using} valuationContainsItsLiteralsVariable[\textit{of} \ ?l \ (\textit{elements} \ (\textit{getM} state))] \\
\textbf{using} valuationContainsItsLiteralsVariable[\textit{of} \ \textit{opposite} \ ?l \ (\textit{elements} \ (\textit{getM} state))]
\textbf{by} auto \\
\textbf{qed}
\textbf{qed}

\textbf{lemma} finalConflictingState:

\textbf{fixes} state :: State \\
\textbf{assumes}
InvariantUniq \ (\textit{getM} state) \textbf{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 \( \forall \) l. l elgetC state \( \rightarrow \) opposite l el decisions (getM state))
case True
{
  assume getC state \( \neq \) []
  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 \( \neq \) []⟩
  ⟨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 \( \exists \) 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 get\(C\) state = []
  using assms
  using finalConflictingState
  by auto
with True
show ?thesis
  by simp
next
case False
hence \(\neg\)formulaFalse (get\(F\) state) (elements (get\(M\) state)) and vars
  (elements (get\(M\) state)) \(\supseteq\) decisionVars
  using assms
  using finalNonConflictState
  by auto
with False
show ?thesis
  by simp
qed

theorem finalStateCharacterization:
fixes \(F_0\) :: Formula and decisionVars :: Variable set and state\(0\) :: State
  and state :: State
assumes
  isInitialState state\(0\) \(F_0\) and
  (state\(0\), state) \(\in\) transitionRelation \(F_0\) decisionVars and
  isFinalState state \(F_0\) decisionVars
shows
  (getConflictFlag state = False \(\land\)
    \(\neg\)formulaFalse (get\(F\) state) (elements (get\(M\) state)) \(\land\)
    vars (elements (get\(M\) state)) \(\supseteq\) decisionVars) \(\lor\)
  (getConflictFlag state = True \(\land\)
    get\(C\) state = [])
proof
  from ⟨isInitialState state\(0\) \(F_0\); (state\(0\), state) \(\in\) transitionRelation \(F_0\) decisionVars⟩
  have invariantsHoldInState state \(F_0\) decisionVars
    using invariantsHoldInValidRunsFromInitialState
    by simp
hence
  *: InvariantUniq (get\(M\) state)
  InvariantReasonClauses (get\(F\) state) (get\(M\) state)
  InvariantCFalse (getConflictFlag state) (get\(M\) state) (get\(C\) state)
  unfolding invariantsHoldInState-def
    by auto
from ⟨isFinalState state \(F_0\) decisionVars⟩
have **:
¬ applicable Decide state decisionVars
¬ applicable Conflict state
¬ applicable Explain state
¬ applicable Learn state
¬ applicable Backjump 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
satisfiable F0 and

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

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
theorem completenessForUNSAT:
  fixes $F_0 :: $Formula and $decisionVars :: $Variable set and $state_0 :: $State
  assumes
  vars $F_0 \subseteq decisionVars$ and
  $\neg$ satisfactory $F_0$ and
  isInitialState $state_0 F_0$ and
  $(state_0, state) \in$ transitionRelation $F_0 decisionVars$ and
  isFinalState $state F_0 decisionVars$
  shows
  $getConflictFlag state = True \land getC state = []$

proof –
  from $asms$
  have $*: (getConflictFlag state = False \land$
    $\neg$formulaFalse (getF state) (elements (getM state)) \land
    $\neg$vars (elements (getM state)) $\supseteq decisionVars$) \lor
    (getConflictFlag state = True \land
    $getC state = []$)
  using finalStateCharacterization[of $state_0 F_0 state decisionVars$]
  by auto
  { 
    assume $\neg$ getConflictFlag state = True
    with *
    have getConflictFlag state = False \land $\neg$formulaFalse (getF state)
      (elements (getM state)) \land vars (elements (getM state)) $\supseteq decisionVars$
      by simp
    with $asms$
    have satisfactory $F_0$
      using soundnessForSAT[of $F_0 decisionVars state_0 state$]
      unfolding satisfactory-def
      by auto
    with $(\neg$ satisfactory $F_0)$
    have $False$
      by simp
  }
  with $*$ show $thesis$
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-Numerals
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 queue
getQ :: Literal list
— Unit propagation graph
getReason :: Literal ⇒ nat option — index of a clause that is a reason for propagation of a literal
— Two-watch literal scheme
— clause indices instead of clauses are used
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
cGet :: Clause — Conflict analysis clause - always false in M
cGetl :: Literal — Last asserted literal in (opposite cGet)
cGetml :: Literal — Second last asserted literal in (opposite cGet)
cGetn :: nat — Number of literals of (opposite cGet) on the (currentLevel M)

definition
setWatch1 :: nat ⇒ Literal ⇒ State ⇒ State
where
setWatch1 clause literal state =
state(
getWatch1 := (getWatch1 state)(clause := Some literal),
getWatchList := (getWatchList state)(literal := clause ≠
(getWatchList state literal))
)
declare setWatch1-def [code-unfold]

definition
setWatch2 :: nat ⇒ Literal ⇒ State ⇒ State
where
setWatch2 clause literal state =
state(
getWatch2 := (getWatch2 state)(clause := Some literal),
getWatchList := (getWatchList state)(literal := clause ≠
(getWatchList state literal))
)
declare setWatch2-def [code-unfold]

definition
swapWatches :: nat ⇒ State ⇒ State
where
swapWatch clause state ==
  state{ getWatch1 := (getWatch1 state)(clause := (getWatch2 state clause)),
  getWatch2 := (getWatch2 state)(clause := (getWatch1 state clause)) }
}
declare swapWatch-def[code-unfold]

primrec getNonWatchedUnsatisfiedLiteral :: Clause ⇒ Literal ⇒ Literal Trail ⇒ Literal option
where
getNonWatchedUnsatisfiedLiteral [] w1 w2 M = None |
getNonWatchedUnsatisfiedLiteral (literal # clause) w1 w2 M = (if literal ≠ w1 ∧
  literal ≠ w2 ∧
  ¬ (literalFalse literal (elements M)) then
  Some literal
  else
  getNonWatchedUnsatisfiedLiteral 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
        state’)
    else
    state’)

311
else
  \( \text{case (getNonWatchedUnfalsifiedLiteral (\text{nth (getF state')} clause) w1 \ w2 (getM state')) of} \)
  \( \text{Some } l' \Rightarrow \)
  \( \text{notifyWatches-loop literal list' newWl (setWatch2 clause l' state')} \)
  \( | \text{None } \Rightarrow \)
  \( \text{if (literalFalse w1 (elements (getM state'))}) \text{ then} \)
  \( \text{let state'' = (state'\[\text{getConflictFlag} := \text{True}, \text{getConflictClause} := \text{clause }\]) in} \)
  \( \text{notifyWatches-loop literal list' (clause }\#\text{ newWl) state''} \)
  \( \text{else} \)
  \( \text{let state'' = state'\[\text{getQ} := (\text{if w1 el (getQ state')} \text{ then}} \)
  \( \text{(getQ state')} \text{ else} \)
  \( \text{(getQ state')} @ [w1] \text{)} \)
  \( ) \text{ in} \)
  \( \text{let state''' = (setReason w1 clause state'')} \text{ in} \)
  \( \text{notifyWatches-loop literal list' (clause }\#\text{ newWl) state''''} \)
  \)
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‴)
))
))

313
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),
    getCl1 = (Pos 0),
    getCn = 0
  ⋄)

primrec initialize :: Formula ⇒ State ⇒ State
where
initialize [] state = state |
initialize (clause # formula) state = initialize formula (addClause clause state)

definition
findLastAssertedLiteral :: State ⇒ State
where
findLastAssertedLiteral 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
   ⋄)
let state' = state (\ getC ::= 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'''\ )

definition
getBackjumpLevel :: State ⇒ nat
where
getBackjumpLevel 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 (solvee-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 Wl F = (∀ (l::Literal) (c::nat). c ∈ set (Wl l) ⇒ 0 ≤ c ∧ c < length F)

definition InvariantWatchListsUniq :: (Literal ⇒ nat list) ⇒ bool
where
  InvariantWatchListsUniq Wl = (∀ l. uniq (Wl l))

definition InvariantWatchListsCharacterization :: (Literal ⇒ nat list) ⇒ (nat ⇒ Literal option) ⇒ (nat ⇒ Literal option) ⇒ bool
where
  InvariantWatchListsCharacterization Wl w1 w2 = (∀ (c::nat) (l::Literal). c ∈ set (Wl l) = (Some l = (w1 c) ∨ Some l = (w2 c)))

definition InvariantWatchesEl :: Formula ⇒ (nat ⇒ Literal option) ⇒ (nat ⇒
Literal option) ⇒ bool
where
InvariantWatchesEl formula watch1 watch2 ==
  ∀ (clause::nat). 0 ≤ clause ∧ clause < length formula →
  (∃ (w1::Literal) (w2::Literal). watch1 clause = Some w1 ∧
   watch2 clause = Some w2 ∧
   w1 el (nth formula clause) ∧ w2 el (nth formula clause))

definition
InvariantWatchesDiffer :: Formula ⇒ (nat ⇒ Literal option) ⇒ (nat ⇒ Literal option) ⇒ bool
where
InvariantWatchesDiffer formula watch1 watch2 ==
  ∀ (clause::nat). 0 ≤ clause ∧ clause < length formula → watch1 clause ≠ watch2 clause

definition
watchCharacterizationCondition::Literal ⇒ Literal ⇒ LiteralTrail ⇒ Clause ⇒ bool
where
watchCharacterizationCondition w1 w2 M clause =
  (literalFalse w1 (elements M) →
   ( (∃ l. l el clause ∧ literalTrue l (elements M) ∧ elementLevel l M ≤ elementLevel (opposite w1) M) ∨
      (∀ l. l el clause ∧ l ≠ w1 ∧ l ≠ w2 →
       literalFalse l (elements M) ∧ elementLevel (opposite l) M ≤ elementLevel (opposite w1) M)
   )
  )

definition
InvariantWatchCharacterization::Formula ⇒ (nat ⇒ Literal option) ⇒ (nat ⇒ Literal option) ⇒ LiteralTrail ⇒ bool
where
InvariantWatchCharacterization F watch1 watch2 M =
  (∀ c w1 w2. (0 ≤ c ∧ c < length F ∧ Some w1 = watch1 c ∧
    Some w2 = watch2 c) →
    watchCharacterizationCondition w1 w2 M (nth F c) ∧
    watchCharacterizationCondition w2 w1 M (nth F c)
  )

definition
InvariantQCharacterization :: bool ⇒ Literal list ⇒ Formula ⇒ LiteralTrail ⇒ bool
where
InvariantQCharacterization conflictFlag Q F M ==
  ¬ conflictFlag → (∀ l::Literal. l el Q = (∃ c::Clause. c el F ∧ isUnitClause c l (elements M)))

definition
InvariantUniqQ :: Literal list ⇒ bool
where
InvariantUniqQ Q =
  uniq Q

definition
InvariantConflictFlagCharacterization :: bool ⇒ Formula ⇒ Literal-Trail ⇒ bool
where
InvariantConflictFlagCharacterization conflictFlag F M ==
  conflictFlag = formulaFalse F (elements M)

definition
InvariantNoDecisionsWhenConflict :: Formula ⇒ LiteralTrail ⇒ nat ⇒ bool
where
InvariantNoDecisionsWhenConflict F M level =
  (∀ level′. level′ < level → ¬ formulaFalse F (elements (prefixToLevel level′ M)))

definition
InvariantNoDecisionsWhenUnit :: Formula ⇒ LiteralTrail ⇒ nat ⇒ bool
where
InvariantNoDecisionsWhenUnit F M level =
  (∀ level′. level′ < level → ¬ (∃ clause literal. clause el F ∧ isUnitClause clause literal (elements (prefixToLevel level′ M))))

definition
InvariantEquivalentZL :: Formula ⇒ LiteralTrail ⇒ Formula ⇒ bool
where
InvariantEquivalentZL F M F0 =
equivalentFormulae (F @ val2form (elements (prefixToLevel 0 M))) 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 Cll C M ==
    set C ≠ {opposite Cl} →
isLastAssertedLiteral Cll (removeAll Cl (oppositeLiteralList C))
    (elements M)

definition
InvariantClCurrentLevel :: Literal ⇒ LiteralTrail ⇒ bool
where
InvariantClCurrentLevel Cl M ==
   elementLevel Cl M = currentLevel M

definition
InvariantCnCharacterization :: nat ⇒ Clause ⇒ LiteralTrail ⇒ bool
where
InvariantCnCharacterization Cn C M ==
   Cn = length (filter (λ l. elementLevel (opposite l) M = currentLevel M) (remdups C))

definition
InvariantUniqC :: Clause ⇒ bool
where
InvariantUniqC clause = uniq clause

definition
InvariantVarsQ :: Literal list ⇒ Formula ⇒ Variable set ⇒ bool
where
InvariantVarsQ Q F0 Vbl ==
   vars Q ⊆ vars F0 ∪ Vbl

end

theory AssertLiteral
imports SatSolverCode
begin

lemma getNonWatchedUnfalsifiedLiteralSomeCharacterization:
fixes clause :: Clause and w1 :: Literal and w2 :: Literal and M :: LiteralTrail and l :: Literal
assumes
   getNonWatchedUnfalsifiedLiteral clause w1 w2 M = Some l
shows
   l el clause l ≠ w1 l ≠ w2 ¬ literalFalse l (elements M)
using assms
by (induct clause) (auto split: split-if-asm)

lemma getNonWatchedUnfalsifiedLiteralNoneCharacterization:
fixes clause :: Clause and w1 :: Literal and w2 :: Literal and M :: LiteralTrail

assumes
getNonWatchedUnfalsifiedLiteral clause w1 w2 M = None
shows
\forall l. l el clause \land l \neq w1 \land l \neq w2 \rightarrow 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
\forall (c::nat). c \in set Wl \rightarrow 0 \leq c \land c < length (getF state)
shows
let state' = (notifyWatches-loop literal Wl newWl state) in
(getM state') = (getM state) \land
(getF state') = (getF state) \land
(getSATFlag state') = (getSATFlag state) \land
isPrefix (getQ state) (getQ state')
using assms
proof (induct Wl arbitrary: newWl state)
case Nil
thus \_\_\_\_\_case
  unfolding isPrefix-def
  by simp
next
case (Cons clause Wl)
from \forall (c::nat). c \in set (clause \# Wl) \rightarrow 0 \leq c \land c < length
(getF state)
have \( 0 \leq \text{clause} \wedge \text{clause} < \text{length}(\text{getF state}) \)  
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
     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 getM \(?state\)' = getM state \land
     getF \(?state\)' = getF state \land
     getSATFlag \(?state\)' = getSATFlag state \land
     getQ \(?state\)' = getQ state
     unfolding swapWatches-def
     by simp
  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 \(?state\)')))
  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 \textit{InvariantWatchesEl-def} \\
unfolding swapWatches-def \\
by \textit{auto} \\
moreover \\
have \textit{getM} ?state'' = \textit{getM} state \land \\
\textit{getF} ?state'' = \textit{getF} state \land \\
\textit{getSATFlag} ?state'' = \textit{getSATFlag} state \land \\
\textit{getQ} ?state'' = \textit{getQ} state \\
unfolding swapWatches-def \\
by \textit{simp} \\
ultimately \\
show \textit{?thesis} \\
using \textit{Cons(1)}[of ?state'' clause # newWI] \\
using \textit{Cons(3)} \\
using (\textit{getWatch1} ?state' clause = \textit{Some} ?w1) \\
using (\textit{getWatch2} ?state' clause = \textit{Some} ?w2) \\
using (\textit{Some} literal = \textit{getWatch1} state clause) \\
using (\neg \textit{literalTrue} ?w1 (\textit{elements} (\textit{getM} ?state'))) \\
using None \\
using (\textit{literalFalse} ?w1 (\textit{elements} (\textit{getM} ?state'))) \\
by (\textit{simp add: Let-def}) \\
next \\
case \textit{False} \\
let ?state'' = \textit{setReason} ?w1 clause (?state'\langle getQ := (if ?w1 el (getQ ?state') then (getQ ?state') else (getQ ?state') @ [?w1]) \rangle) \\
from \textit{Cons(2)} \\
have \textit{InvariantWatchesEl} (\textit{getF} ?state'') (\textit{getWatch1} ?state'') (\textit{getWatch2} ?state'') \\
unfolding \textit{InvariantWatchesEl-def} \\
unfolding swapWatches-def \\
unfolding \textit{setReason-def} \\
by \textit{auto} \\
moreover \\
have \textit{getM} ?state'' = \textit{getM} state \land \\
\textit{getF} ?state'' = \textit{getF} state \land \\
\textit{getSATFlag} ?state'' = \textit{getSATFlag} state \land \\
\textit{getQ} ?state'' = (if ?w1 el (getQ state) then (getQ state) else (getQ state) @ [?w1]) \\
unfolding swapWatches-def \\
unfolding \textit{setReason-def} \\
by \textit{auto} \\
ultimately \\
show \textit{?thesis} \\
using \textit{Cons(1)}[of ?state'' clause # newWI] \\
using \textit{Cons(3)} \\
using (\textit{getWatch1} ?state' clause = \textit{Some} ?w1) \\
using (\textit{getWatch2} ?state' clause = \textit{Some} ?w2) \\
using (\textit{Some} literal = \textit{getWatch1} state clause) \\
using (\neg \textit{literalTrue} ?w1 (\textit{elements} (\textit{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

327
moreover
have getM ?state" = getM state \and
getF ?state" = getF state \and
getSATFlag ?state" = getSATFlag state \and
getQ ?state" = getQ 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 (~ 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
have getM ?state" = getM state \and
getF ?state" = getF state \and
getSATFlag ?state" = getSATFlag state \and
getQ ?state" = getQ 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 (~ 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 \( ?\text{state}'\) = setReason \(?w1\) clause (\(?\text{state}' (\text{getQ} := (\text{if} ?w1 \text{ el (getQ }\?\text{state}') \text{ then (getQ }\?\text{state}') \text{ else (getQ }\?\text{state}')@[?w1])))\)

from \(\text{Cons}(2)\)

have InvariantWatchesEl (getF \(?\text{state}'\) (getWatch1 \(?\text{state}'\))

\[ \text{getWatch}2 \ ?\text{state}'\)"

unfolding InvariantWatchesEl-def

unfolding setReason-def

by auto

moreover

have getM \(?\text{state}'\) = getM state ∧

getF \(?\text{state}'\) = getF state ∧

getSATFlag \(?\text{state}'\) = getSATFlag state ∧

getQ \(?\text{state}'\) = (\text{if} ?w1 \text{ el (getQ }\?\text{state}') \text{ then (getQ }\?\text{state}') \text{ else (getQ }\?\text{state}')@[?w1])

unfolding setReason-def

by simp

ultimately

show \(?\text{thesis}\)

using \(\text{Cons}(1)[\text{of }\?\text{state}]\)

using \(\text{Cons}(3)\)

using (getWatch1 \(?\text{state}'\) clause = Some \(?w1\))

using (getWatch2 \(?\text{state}'\) clause = Some \(?w2\))

using (\(\neg\) Some literal = getWatch1 state clause)

using (\(\neg\) literalTrue \(?w1\) (elements (getM \(?\text{state}'\)')))

using None

using (\(\neg\) literalFalse \(?w1\) (elements (getM \(?\text{state}'\)')))

unfolding isPrefix-def

by (auto simp add: Let-def split: split-if-asm)

qed

qed

qed

qed

lemma notifyWatchesStartQIrelevent:

fixes literal :: Literal and \(Wl\) :: nat list and \(\text{newWl}\) :: nat list and state :: State

assumes

InvariantWatchesEl (getF stateA) (getWatch1 stateA) (getWatch2 stateA) and

\(\forall (c::nat).\ c \in \text{set } Wl \rightarrow 0 \leq c \land c < \text{length (getF stateA})\) and

getM stateA = getM stateB and

getF stateA = getF stateB and

getWatch1 stateA = getWatch1 stateB and

getWatch2 stateA = getWatch2 stateB and

getConflictFlag stateA = getConflictFlag stateB and

getSATFlag stateA = getSATFlag stateB

shows

329
let state' = (notifyWatches-loop literal Wl newWl stateA) in
let state'' = (notifyWatches-loop literal Wl newWl stateB) in
(getM state') = (getM state'') \land
(getF state') = (getF state'') \land
(getSATFlag state') = (getSATFlag state'') \land
(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 \( \forall (c::nat). \ c \in \text{set} (\text{clause} \# Wl) \rightarrow 0 \leq c \land c < \text{length} (\text{getF stateA}) \)
  have \( 0 \leq \text{clause} \land \text{clause} < \text{length} (\text{getF stateA}) \)
    by auto
  then obtain \( wa::\text{Literal} \) and \( wb::\text{Literal} \)
    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

330
by auto
hence $\text{getWatch}_1 \ ?\text{state}'B$ clause = Some $\?w_1$
  using $(\text{getWatch}_1 \ ?\text{state}'A = \text{getWatch}_1 \ ?\text{state}'B)$
  by simp
let $\?w_2 = wa$
have $\text{getWatch}_2 \ ?\text{state}'A$ clause = Some $\?w_2$
  using $(\text{getWatch}_1 \ ?\text{state}'A$ clause = Some $wa$)
  unfolding swapWatches-def
  by auto
hence $\text{getWatch}_2 \ ?\text{state}'B$ clause = Some $\?w_2$
  using $(\text{getWatch}_2 \ ?\text{state}'A = \text{getWatch}_2 \ ?\text{state}'B)$
  by simp

show $\?thesis$
proof (cases literalTrue $\?w_1$ (elements (getM $\?\text{state}'A$)))
  case True
  hence literalTrue $\?w_1$ (elements (getM $\?\text{state}'B$))
    using $(\text{getM} \ ?\text{state}'A = \text{getM} \ ?\text{state}'B)$
    by simp

from Cons(2)
  have InvariantWatchesEl (getF $\?\text{state}'A$) (getWatch1 $\?\text{state}'A$) (getWatch2 $\?\text{state}'A$)
    unfolding InvariantWatchesEl-def
    unfolding swapWatches-def
    by auto
moreover
have getM $\?\text{state}'A = \text{getM} \ ?\text{state}'A$ ∧
  getF $\?\text{state}'A = \text{getF} \ ?\text{state}'A$ ∧
  getSATFlag $\?\text{state}'A = \text{getSATFlag} \ ?\text{state}'A$ ∧
  getQ $\?\text{state}'A = \text{getQ} \ ?\text{state}'A$
    unfolding swapWatches-def
    by simp
moreover
have getM $\?\text{state}'B = \text{getM} \ ?\text{state}'B$ ∧
  getF $\?\text{state}'B = \text{getF} \ ?\text{state}'B$ ∧
  getSATFlag $\?\text{state}'B = \text{getSATFlag} \ ?\text{state}'B$ ∧
  getQ $\?\text{state}'B = \text{getQ} \ ?\text{state}'B$
    unfolding swapWatches-def
    by simp
ultimately
show $\?thesis$
  using Cons(1)[of $\?\text{state}'A \ ?\text{state}'B$ clause ≠ newWL]
  using $(\text{getM} \ ?\text{state}'A = \text{getM} \ ?\text{state}'B)$
  using $(\text{getF} \ ?\text{state}'A = \text{getF} \ ?\text{state}'B)$
  using $(\text{getWatch}_1 \ ?\text{state}'A = \text{getWatch}_1 \ ?\text{state}'B)$
  using $(\text{getWatch}_2 \ ?\text{state}'A = \text{getWatch}_2 \ ?\text{state}'B)$
using \(\text{getConflictFlag} \ ?\text{state}'A = \text{getConflictFlag} \ ?\text{state}'B\)
using \(\text{getSATFlag} \ ?\text{state}'A = \text{getSATFlag} \ ?\text{state}'B\)
using \(\text{Cons}(3)\)
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 \(\text{Some literal} = \text{getWatch1} \ ?\text{state}'A \text{ clause}\)
using \(\text{getWatch2} \ ?\text{state}'B \text{ clause} = \text{Some} \ ?w1\)
using \(\text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'A))\)
using \(\text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'B))\)
by (simp add: Let-def)

next

case False

hence \(\neg \text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'B))\)

using \(\text{getM} \ ?\text{state}'A = \text{getM} \ ?\text{state}'B\)
by simp

show \(\text{thesis}\)

proof (cases \(\text{getNonWatchedUnfalsifiedLiteral} \ (\text{nth} \ (\text{getF} \ ?\text{state}'A) \text{ clause} \ ?w1 \ ?w2 \ (\text{getM} \ ?\text{state}'A))\))

case (Some \(l')\)

hence \(\text{getNonWatchedUnfalsifiedLiteral} \ (\text{nth} \ (\text{getF} \ ?\text{state}'B) \text{ clause}) \ ?w1 \ ?w2 \ (\text{getM} \ ?\text{state}'B) = \text{Some} \ l'\)

using \(\text{getF} \ ?\text{state}'A = \text{getF} \ ?\text{state}'B\)
using \(\text{getM} \ ?\text{state}'A = \text{getM} \ ?\text{state}'B\)
by simp

have \(l' \in \text{el} (\text{nth} \ (\text{getF} \ ?\text{state}'A) \text{ clause})\)
using \(\text{Some}\)
using \(\text{getNonWatchedUnfalsifiedLiteralSomeCharacterization}\)
by simp

hence \(l' \in \text{el} (\text{nth} \ (\text{getF} \ ?\text{state}'B) \text{ clause})\)
using \(\text{getF} \ ?\text{state}'A = \text{getF} \ ?\text{state}'B\)
by simp

let \(\text{?state}''A = \text{setWatch2} \ ?\text{state}' \ ?\text{state}'A\)
let \(\text{?state}''B = \text{setWatch2} \ ?\text{state}' \ ?\text{state}'B\)

have
\(\text{getM} \ ?\text{state}''A = \text{getM} \ ?\text{state}''B\)
\(\text{getF} \ ?\text{state}''A = \text{getF} \ ?\text{state}''B\)
\(\text{getWatch1} \ ?\text{state}''A = \text{getWatch1} \ ?\text{state}''B\)
\(\text{getWatch2} \ ?\text{state}''A = \text{getWatch2} \ ?\text{state}''B\)
\(\text{getConflictFlag} \ ?\text{state}''A = \text{getConflictFlag} \ ?\text{state}''B\)
\(\text{getSATFlag} \ ?\text{state}''A = \text{getSATFlag} \ ?\text{state}''B\)
using \(\text{Cons}\)
unfolding \(\text{setWatch2-def}\)
unfolding \(\text{swapWatches-def}\)

332
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)
  unfolding getWatch1 ?state"A = getWatch1 ?state"B
  using (getWatch2 ?state"A = getWatch2 ?state"B)
  unfolding 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 l')
  using (getNonWatchedUnfalsifiedLiteral (nth (getF ?state'B)
Let \( \?w1 \) \( \?w2 \) (getM \( ?\text{state}'B \)) = Some \( l' \) by (simp add: Let-def)

next

\begin{itemize}
\item case None
\end{itemize}

hence getNonWatchedUnfalsifiedLiteral (nth (getF \( ?\text{state}'B \)) clause) \( ?w1 \) \( ?w2 \) (getM \( ?\text{state}'B \)) = None

\begin{itemize}
\item using \( \text{getF}\ ?\text{state}'A = \text{getF}\ ?\text{state}'B \) \( \text{getM}\ ?\text{state}'A = \text{getM}\ ?\text{state}'B \)
\end{itemize}

by simp

show \( ?\text{thesis} \)

proof (cases literalFalse \( ?w1 \) (elements (getM \( ?\text{state}'A \)))))

\begin{itemize}
\item case True
\end{itemize}

hence literalFalse \( ?w1 \) (elements (getM \( ?\text{state}'B \)))

\begin{itemize}
\item using \( \text{getM}\ ?\text{state}'A = \text{getM}\ ?\text{state}'B \)
\end{itemize}

by simp

\begin{itemize}
\item let \( ?\text{state}''A = ?\text{state}'A[\text{getConflictFlag} := \text{True}, \text{getConflictClause} := \text{clause}] \)
\item let \( ?\text{state}''B = ?\text{state}'B[\text{getConflictFlag} := \text{True}, \text{getConflictClause} := \text{clause}] \)
\end{itemize}

have \( \text{getM}\ ?\text{state}''A = \text{getM}\ ?\text{state}''B \)

\begin{itemize}
\item getF \( ?\text{state}'A = \text{getF}\ ?\text{state}'B \)
\item getWatch1 \( ?\text{state}'A = \text{getWatch1}\ ?\text{state}'B \)
\item getWatch2 \( ?\text{state}'A = \text{getWatch2}\ ?\text{state}'B \)
\item getConflictFlag \( ?\text{state}'A = \text{getConflictFlag}\ ?\text{state}'B \)
\item getSATFlag \( ?\text{state}'A = \text{getSATFlag}\ ?\text{state}'B \)
\item using \( \text{Cons} \)
\item unfolding \( \text{swapWatches-def} \)
\item by auto
\end{itemize}

from Cons(2)

have \( \text{InvariantWatchesEl}\ (\text{getF}\ ?\text{state}''A)\ (\text{getWatch1}\ ?\text{state}''A)\ (\text{getWatch2}\ ?\text{state}''A) \)

\begin{itemize}
\item unfolding \( \text{InvariantWatchesEl-def} \)
\item unfolding \( \text{swapWatches-def} \)
\item by auto
\end{itemize}

moreover

have \( \text{getM}\ ?\text{state}''A = \text{getM}\ \text{state}A \land \)

\begin{itemize}
\item getF \( ?\text{state}'A = \text{getF}\ \text{state}A \land \)
\item getSATFlag \( ?\text{state}'A = \text{getSATFlag}\ \text{state}A \land \)
\item getQ \( ?\text{state}'A = \text{getQ}\ \text{state}A \land \)
\item unfolding \( \text{swapWatches-def} \)
\item by simp
\end{itemize}

moreover

have \( \text{getM}\ ?\text{state}''B = \text{getM}\ \text{state}B \land \)

\begin{itemize}
\item getF \( ?\text{state}'B = \text{getF}\ \text{state}B \land \)
\item getSATFlag \( ?\text{state}'B = \text{getSATFlag}\ \text{state}B \land \)
\item getQ \( ?\text{state}'B = \text{getQ}\ \text{state}B \land \)
\end{itemize}
unfolding \texttt{swapWatches-def}

by \texttt{simp}

ultimately

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

\textbf{using} \ \texttt{Cons(4)} \ \texttt{Cons(5)}

\textbf{using} \ \texttt{Cons(1)[of \ ?state''A \ ?state''B \ clause \ # \ newWl]}

\textbf{using} \ \texttt{getM \ ?state''A = getM \ ?state''B}

\textbf{using} \ \texttt{getF \ ?state''A = getF \ ?state''B}

\textbf{using} \ \texttt{getWatch1 \ ?state''A = getWatch1 \ ?state''B}

\textbf{using} \ \texttt{getWatch2 \ ?state''A = getWatch2 \ ?state''B}

\textbf{using} \ \texttt{getConflictFlag \ ?state''A = getConflictFlag \ ?state''B}

\textbf{using} \ \texttt{getSATFlag \ ?state''A = getSATFlag \ ?state''B}

\textbf{using} \ \texttt{Cons(3)}

\textbf{using} \ \texttt{⟨\ getWatch1 \ ?state'A \ clause = Some \ ?w1⟩}

\textbf{using} \ \texttt{⟨\ getWatch2 \ ?state'B \ clause = Some \ ?w2⟩}

\textbf{using} \ \texttt{⟨\ Some \ literal = getWatch1 \ stateA \ clause⟩}

\textbf{using} \ \texttt{⟨\ Some \ literal = getWatch1 \ stateB \ clause⟩}

\textbf{using} \ \texttt{⟨\ ¬ \ literalTrue \ ?w1 \ (elements \ (getM \ ?state'A))⟩}

\textbf{using} \ \texttt{⟨\ ¬ \ literalTrue \ ?w1 \ (elements \ (getM \ ?state'B))⟩}

\textbf{using} \ \texttt{⟨\ getNonWatchedUnfalsifiedLiteral \ (nth \ (getF \ ?state'A) \ clause) \ ?w1 \ ?w2 \ (getM \ ?state'A) = None⟩}

\textbf{using} \ \texttt{⟨\ getNonWatchedUnfalsifiedLiteral \ (nth \ (getF \ ?state'B) \ clause) \ ?w1 \ ?w2 \ (getM \ ?state'B) = None⟩}

\textbf{using} \ \texttt{⟨\ literalFalse \ ?w1 \ (elements \ (getM \ ?state'A))⟩}

\textbf{using} \ \texttt{⟨\ literalFalse \ ?w1 \ (elements \ (getM \ ?state'B))⟩}

by \ \texttt{(simp add: Let-def)}

\textbf{next}

\textbf{case} \ \texttt{False}

\textbf{hence} \ \texttt{¬ \ literalFalse \ ?w1 \ (elements \ (getM \ ?state'B))}

\textbf{using} \ \texttt{⟨\ getM \ ?state'A = getM \ ?state'B⟩}

by \ \texttt{simp}

\textbf{let} \ \texttt{?state''A = setReason \ ?w1 \ clause \ (?state'A)[getQ := (if \ ?w1 \ el \ (getQ \ ?state'A) \ then \ (getQ \ ?state'A) \ else \ (getQ \ ?state'A) \ @ \ [?w1])])}

\textbf{let} \ \texttt{?state''B = setReason \ ?w1 \ clause \ (?state'B)[getQ := (if \ ?w1 \ el \ (getQ \ ?state'B) \ then \ (getQ \ ?state'B) \ else \ (getQ \ ?state'B) \ @ \ [?w1])])}

\textbf{have}

\begin{itemize}
  \item \texttt{getM \ ?state''A = getM \ ?state''B}
  \item \texttt{getF \ ?state''A = getF \ ?state''B}
  \item \texttt{getWatch1 \ ?state''A = getWatch1 \ ?state''B}
  \item \texttt{getWatch2 \ ?state''A = getWatch2 \ ?state''B}
  \item \texttt{getConflictFlag \ ?state''A = getConflictFlag \ ?state''B}
  \item \texttt{getSATFlag \ ?state''A = getSATFlag \ ?state''B}
\end{itemize}

\textbf{using} \ \texttt{Cons}

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

335
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
  unfolding setReason-def
  by auto
moreover
have getM ?state"A = getM stateA ∧
  getF ?state"A = getF stateA ∧
  getSATFlag ?state"A = getSATFlag stateA ∧
  getQ ?state"A = (if ?w1 el (getQ stateA) then (getQ stateA)
  else (getQ stateA) @ [?w1])
  unfolding swapWatches-def
  unfolding setReason-def
  by auto
moreover
have getM ?state"B = getM stateB ∧
  getF ?state"B = getF stateB ∧
  getSATFlag ?state"B = getSATFlag stateB ∧
  getQ ?state"B = (if ?w1 el (getQ stateB) then (getQ stateB)
  else (getQ stateB) @ [?w1])
  unfolding swapWatches-def
  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 (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) = None)
using \langle \text{getNonWatchedUnfalsifiedLiteral} \ (\text{nth} \ (\text{getF} \ ?\text{state}'B) \ \text{clause}) \ ?w1 \ ?w2 \ (\text{getM} \ ?\text{state}'B) = \text{None} \rangle

using \langle \neg \text{literalFalse} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'A)) \rangle
using \langle \neg \text{literalFalse} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'B)) \rangle
by \ (\text{simp add:Let-def})

case False
hence Some literal \neq \text{getWatch1} \text{state}B \text{clause}
using Cons
by simp

let ?\text{state}'A = \text{state}A
let ?\text{state}'B = \text{state}B

have
\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
using Cons
by auto

let ?w1 = wa
have \text{getWatch1} \ ?\text{state}'A \text{clause} = \text{Some} \ ?w1
using \ (\text{getWatch1} \text{state}A \text{clause} = \text{Some} \ wa)
by auto
hence \text{getWatch1} \ ?\text{state}'B \text{clause} = \text{Some} \ ?w1
using Cons
by simp
let ?w2 = wb
have \text{getWatch2} \ ?\text{state}'A \text{clause} = \text{Some} \ ?w2
using \ (\text{getWatch2} \text{state}A \text{clause} = \text{Some} \ wb)
by auto
hence \text{getWatch2} \ ?\text{state}'B \text{clause} = \text{Some} \ ?w2
using Cons
by simp

show \ ?thesis
proof \ (\text{cases literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'A)))
case True
hence literalTrue \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'B))
using Cons
by simp

337
show \( ?\text{thesis} \)

using Cons(1) of \(?\text{state}'A \ ?\text{state}'B\) clause \# newWL

using Cons(2) Cons(3) Cons(4) Cons(5) Cons(6) Cons(7) Cons(8) Cons(9)

\begin{align*}
\text{using} & \quad \neg \text{Some literal} = \text{getWatch1} \ ?\text{state}'A\ \text{clause} \\
\text{using} & \quad \neg \text{Some literal} = \text{getWatch1} \ ?\text{state}'B\ \text{clause} \\
\text{using} & \quad \text{getWatch1} \ ?\text{state}'A\ \text{clause} = \text{Some} \ ?w1 \\
\text{using} & \quad \text{getWatch1} \ ?\text{state}'B\ \text{clause} = \text{Some} \ ?w1 \\
\text{using} & \quad \text{getWatch2} \ ?\text{state}'A\ \text{clause} = \text{Some} \ ?w2 \\
\text{using} & \quad \text{getWatch2} \ ?\text{state}'B\ \text{clause} = \text{Some} \ ?w2 \\
\text{using} & \quad \text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'A)) \\
\text{using} & \quad \text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'B)) \\
\text{by} & \quad \text{(simp add: Let-def)}
\end{align*}

next

case False

hence \( \neg \text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'B)) \)

using \( \text{getM} \ ?\text{state}'A = \text{getM} \ ?\text{state}'B \)

by simp

show \(?\text{thesis}\)

proof (cases getNonWatchedUnfalsifiedLiteral \(?\text{nth} \ (\text{getF} \ ?\text{state}'A)\ \text{clause}) \ ?w1 \ ?w2 \ (\text{getM} \ ?\text{state}'A)\)

case \( \text{Some} \ l' \)

hence getNonWatchedUnfalsifiedLiteral \(?\text{nth} \ (\text{getF} \ ?\text{state}'B)\ \text{clause}) \ ?w1 \ ?w2 \ (\text{getM} \ ?\text{state}'B) = \text{Some} \ l' \)

using \( \text{getF} \ ?\text{state}'A = \text{getF} \ ?\text{state}'B \)

using \( \text{getM} \ ?\text{state}'A = \text{getM} \ ?\text{state}'B \)

by simp

have \( l' \ \text{el} \ (\text{nth} \ (\text{getF} \ ?\text{state}'A)\ \text{clause}) \)

using Some

using getNonWatchedUnfalsifiedLiteralSomeCharacterization

by simp

hence \( l' \ \text{el} \ (\text{nth} \ (\text{getF} \ ?\text{state}'B)\ \text{clause}) \)

using \( \text{getF} \ ?\text{state}'A = \text{getF} \ ?\text{state}'B \)

by simp

let \( ?\text{state}'A'' = \text{setWatch2} \ ?\text{clause} \ l' \ ?\text{state}'A \)

let \( ?\text{state}'B'' = \text{setWatch2} \ ?\text{clause} \ l' \ ?\text{state}'B \)

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 setWatch2-def

by auto

338
from Cons(2)

have InvariantWatchesEl (getF ?state"A" (getWatch1 ?state"A") (getWatch2 ?state"A")
  using getF ?state'"A"
  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

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 (getF ?state"A" = getF stateA)

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 state'A clause)

using (∼ Some literal = getWatch1 state'B 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 l')

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 \( \text{literalFalse } \wedge \text{elements (getM state'B)} \)
using \( \{ \text{getM state'A = getM state'B} \) 
by simp

let \( \text{state''A = state'A[getConflictFlag := True, getConflict-Clause := clause]} \)
let \( \text{state''B = state'B[getConflictFlag := True, getConflictClause := clause]} \)
have
\begin{align*}
& \text{getM state''A = getM state''B} \\
& \text{getF state''A = getF state''B} \\
& \text{getWatch1 state''A = getWatch1 state''B} \\
& \text{getWatch2 state''A = getWatch2 state''B} \\
& \text{getConflictFlag state''A = getConflictFlag state''B} \\
& \text{getSATFlag state''A = getSATFlag state''B} \\
\end{align*}
using \( \text{Cons} \)
by auto

from \( \text{Cons(2)} \)
have \( \text{InvariantWatchesEl (getF state''A) (getWatch1 state''A)} \)
\( \text{(getWatch2 state''A)} \)

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

moreover
have \( \text{getM state''A = getM stateA} \wedge \\
\text{getF state''A = getF stateA} \wedge \\
\text{getSATFlag state''A = getSATFlag stateA} \wedge \\
\text{getQ state''A = getQ stateA} \)
by simp

ultimately
show \( \text{thesis} \)
using \( \text{Cons(4) Cons(5)} \)
using \( \text{Cons(1)[of state''A state''B clause # newWl]} \)
using \( \{ \text{getM state'A = getM state'B} \) \\
using \( \{ \text{getF state'A = getF state'B} \) \\
using \( \{ \text{getWatch1 state'A = getWatch1 state'B} \) \\
using \( \{ \text{getWatch2 state'A = getWatch2 state'B} \) \\
using \( \{ \text{getConflictFlag state'A = getConflictFlag state'B} \) \\
using \( \{ \text{getSATFlag state'A = getSATFlag state'B} \) \\
using \( \text{Cons(3)} \)
using \( \{ \text{getWatch1 state'A clause = Some w1} \) \\
using \( \{ \text{getWatch2 state'A clause = Some w2} \) \\
using \( \{ \text{getWatch1 state'B clause = Some w1} \) \\
using \( \{ \text{getWatch2 state'B clause = Some w2} \) \\
using \( \{ \text{Some literal = getWatch1 stateA clause} \) \\
using \( \{ \text{Some literal = getWatch1 stateB clause} \) \\
using \( \{ \text{literalTrue w1 (elements (getM state'A)} \) \\
using \( \{ \text{literalTrue w1 (elements (getM state'B)} \) \\
using \( \text{getNonWatchedUnfalsifiedLiteral (nth (getF state'A)} \)
clause) \(?w1 \?w2\) \((getM \?state'A) = None\)
using \(\langle\text{getNonWatchedUnfalsifiedLiteral} \ (\text{nth} \ (getF \?state'B)\)\)
clause) \(?w1 \?w2\) \((getM \?state'B) = None\)
using \(\langle\text{literalFalse} \?w1 \ (\text{elements} \ (getM \?state'A))\rangle\)
using \(\langle\text{literalFalse} \?w1 \ (\text{elements} \ (getM \?state'B))\rangle\)
by \(\langle\text{simp add: Let-def}\rangle\)

next
case False
hence \(\text{literalFalse} \?w1 \ (\text{elements} \ (getM \?state'A))\)
using \(\langle\text{getM} \?state'A = getM \?state'B\rangle\)
by \(\langle\text{simp}\rangle\)

def
let \(?state''A = \text{setReason} \?w1 \?\text{clause} \ (?state'A)\)
if \(?w1\) \(?\text{el}\) \((\text{getQ} \?state'A)\) then
\((\text{getQ} \?state'A)\) else
\((\text{getQ} \?state'A)@([?w1])\)

let \(?state''B = \text{setReason} \?w1 \?\text{clause} \ (?state'B)\)
if \(?w1\) \(?\text{el}\) \((\text{getQ} \?state'B)\) then
\((\text{getQ} \?state'B)\) else
\((\text{getQ} \?state'B)@([?w1])\)

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 Cons
unfolding \?\text{setReason-def}\)
by auto

from Cons \(2\)
have \(\text{InvariantWatchesEl} \ (\text{getF} \?state''A) \ (\text{getWatch1} \?state''A)\)
\((\text{getWatch2} \?state''A)\)
unfolding \?\text{InvariantWatchesEl-def}\)
unfolding \?\text{setReason-def}\)
by auto
moreover
have \(\text{getM} \?state''A = \text{getM} \?state'A \land\)
\(\text{getF} \?state''A = \text{getF} \?state'A \land\)
\(\text{getSATFlag} \?state''A = \text{getSATFlag} \?state'A \land\)
\(\text{getQ} \?state''A = \text{if} \?w1 \ ?\text{el} \ (\text{getQ} \?state'A) \?\text{then} \ (\text{getQ} \?state'A)\)
else
\((\text{getQ} \?state'A)@([?w1])\)
unfolding \?\text{setReason-def}\)
by auto
ultimately
show \?\text{thesis}\)
using Cons \(4\) Cons \(5\)
using Cons \(1\) \[(\?state''A \?state''B \?\text{clause} \# newWl)\]
using \(\langle\text{getM} \?state''A = \text{getM} \?state''B\rangle\)
using \(\langle\text{getF} \?state''A = \text{getF} \?state''B\rangle\)
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) = 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)

qed
qed
qed

lemma notifyWatchesLoopPreservedWatches:
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
∀ c. c ∉ set Wl ⟷ (getWatch1 state' c) = (getWatch1 state c)
∧ (getWatch2 state' c) = (getWatch2 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 \( w_a::\text{Literal} \) and \( w_b::\text{Literal} \)

where \( \text{getWatch1 state clause} = \text{Some } w_a \) and \( \text{getWatch2 state clause} = \text{Some } w_b \)

using \( \text{Cons} \)

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

by \( \text{auto} \)

show \( \text{?thesis} \)

proof \( \text{(cases Some literal = getWatch1 state clause)} \)

case \( \text{True} \)

let \( w_1 = w_b \)

have \( \text{getWatch1 ?state'} clause = \text{Some } w_1 \)

using \( \text{getWatch2 state clause} = \text{Some } w_b \)

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

by \( \text{auto} \)

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 \( \text{auto} \)

show \( \text{?thesis} \)

proof \( \text{(cases literalTrue } w_1 \text{ (elements } (getM ?state'))} \)

case \( \text{True} \)

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 \( \text{getM ?state'} = \text{getM state } \land \text{getF ?state'} = \text{getF state} \)

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

by \( \text{simp} \)

ultimately

show \( \text{?thesis} \)

using \( \text{Cons(1)}[\text{of ?state'} \text{ clause } \neq \text{newWl}] \)

using \( \text{Cons(3)} \)

using \( \text{getWatch1 ?state'} clause = \text{Some } w_1 \)

using \( \text{getWatch2 ?state'} clause = \text{Some } w_2 \)

using \( \text{Some literal = getWatch1 state clause} \)

using \( \text{literalTrue } w_1 \text{ (elements } (getM ?state')) \)

apply \( \text{(simp add:Let-def)} \)

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

by \( \text{simp} \)

next

case \( \text{False} \)

show \( \text{?thesis} \)
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
  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
  apply (simp add: Let-def)
  unfolding setWatch2-def
  unfolding swapWatches-def
  by simp

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

344
by auto
moreover
have \( \text{getM \ ?state''} = \text{getM \ state} \land \text{getF \ ?state''} = \text{getF \ state} \)
  unfolding \text{swapWatches-def}
  by simp
ultimately
show \(?thesis\)
  using \text{Cons(1)[of \ ?state'' clause \# newW1]}
  using \text{Cons(3)}
  using \(\text{getWatch1 \ ?state'} \declaredclauses = \text{Some \ ?w1}\)
  using \(\text{getWatch2 \ ?state'} \declaredclauses = \text{Some \ ?w2}\)
  using \(\text{Some \ literal = getWatch1 \ state clause}\)
  using \(\neg \text{literalTrue \ ?w1 \ (elements \ (getM \ ?state'))}\)
  using \text{None}
  using \(\text{literalFalse \ ?w1 \ (elements \ (getM \ ?state'))}\)
apply (simp add: Let-def)
unfolding \text{swapWatches-def}
by simp
next
  case False
  let \(?state'' = \text{setReason \ ?w1 \ clause \ (\?state'}\text{(getQ := (if \ ?w1 \ el \ (getQ \ ?state'} \then \ (getQ \ ?state'} \else \ (getQ \ ?state'} @ \[\?w1\])))})\)
  from \text{Cons(2)}
  have \(\text{InvariantWatchesEl \ (getF \ ?state'')} \ (\text{getWatch1 \ ?state'''})\)
    \(\text{(getWatch2 \ ?state'''})\)
    unfolding \text{InvariantWatchesEl-def}
    unfolding \text{swapWatches-def}
    unfolding \text{setReason-def}
    by auto
moreover
have \(\text{getM \ ?state''} = \text{getM \ state} \land \text{getF \ ?state''} = \text{getF \ state} \)
  unfolding \text{swapWatches-def}
  unfolding \text{setReason-def}
  by simp
ultimately
show \(?thesis\)
  using \text{Cons(1)[of \ ?state'' clause \# newW1]}
  using \text{Cons(3)}
  using \(\text{getWatch1 \ ?state'} \declaredclauses = \text{Some \ ?w1}\)
  using \(\text{getWatch2 \ ?state'} \declaredclauses = \text{Some \ ?w2}\)
  using \(\text{Some \ literal = getWatch1 \ state clause}\)
  using \(\neg \text{literalTrue \ ?w1 \ (elements \ (getM \ ?state'))}\)
  using \text{None}
  using \(\neg \text{literalFalse \ ?w1 \ (elements \ (getM \ ?state'))}\)
apply (simp add: Let-def)
unfolding \text{setReason-def}
unfolding swapWatches-def
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)
  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 getM ?state'' = getM state ∧

346
\[ \text{getF } \text{?state}' = \text{getF state} \]

unfolding setWatch2-def
by simp
ultimately
show \(\text{?thesis}\)
  using Cons(1)[of \text{?state}'']
  using Cons(3)
  using (getWatch1 \text{?state}' clause = Some \text{?w1})
  using (getWatch2 \text{?state}' clause = Some \text{?w2})
  using (\neg \text{Some literal = getWatch1 state clause})
  using (\neg \text{literalTrue \text{?w1} (elements (getM \text{?state}''))})
  using Some
  apply (simp add: Let-def)
  unfolding setWatch2-def
  by simp

next
  case None
  show \(\text{?thesis}\)
  proof
    (cases literalFalse \text{?w1} (elements (getM \text{?state}'')))
    case True
    let \text{?state}'' = \text{?state}'[getConflictFlag := True, getConflict-Clause := clause]

    from Cons(2)
    have InvariantWatchesEl (getF \text{?state}'') (getWatch1 \text{?state}'')
    (getWatch2 \text{?state}'')
      unfolding InvariantWatchesEl-def
      by auto
    moreover
    have getM \text{?state}'' = getM state \land
    getF \text{?state}'' = getF state
      by simp
    ultimately
    show \(\text{?thesis}\)
      using Cons(1)[of \text{?state}'']
      using Cons(3)
      using (getWatch1 \text{?state}' clause = Some \text{?w1})
      using (getWatch2 \text{?state}' clause = Some \text{?w2})
      using (\neg \text{Some literal = getWatch1 state clause})
      using (\neg \text{literalTrue \text{?w1} (elements (getM \text{?state}''))})
      using None
      using (literalFalse \text{?w1} (elements (getM \text{?state}'')))
      by (simp add: Let-def)

next
  case False
  let \text{?state}'' = setReason \text{?w1 clause} (\text{?state}'[getQ := (if \text{?w1 el (getQ \text{?state}'')} then (getQ \text{?state}') else (getQ \text{?state}') @ [\text{?w1}])])
  from Cons(2)
  have InvariantWatchesEl (getF \text{?state}'') (getWatch1 \text{?state}'')
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 < length (getF state)

shows
  let state' = (notifyWatches-loop literal Wl newWl state) in
  InvariantWatchesEl (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 ∀ (c::nat). c ∈ set (clause # Wl) ⩾ 0 ∧ c < length (getF state)
    have 0 ⩽ clause and clause < length (getF state)
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 \(?case\\)

**proof** (cases \( \text{Some literal} = \text{getWatch1 state clause} \))

**case** True
let \(?\text{state}' = \text{swapWatches clause state}\\)
let \(?w1 = \text{wb}\\)
have \( \text{getWatch1 ?state' clause} = \text{Some ?w1} \)
using \( \text{getWatch2 state clause} = \text{Some wb} \)
unfolding swapWatches-def
by auto
let \(?w2 = \text{wa}\\)
have \( \text{getWatch2 ?state' clause} = \text{Some ?w2} \)
using \( \text{getWatch1 state clause} = \text{Some wa} \)
unfolding swapWatches-def
by auto
show \(?thesis\\)

**proof** (cases literalTrue \( ?w1 \) (elements \( \text{getM ?state'} \)))

**case** True

from Cons(2)

have \( \text{InvariantWatchesEl (getF ?state') (getWatch1 ?state')} \)
unfolding InvariantWatchesEl-def
unfolding swapWatches-def
by auto
moreover
have \( \text{getF ?state'} = \text{getF state} \)
unfolding swapWatches-def
by simp
ultimately
show \(?thesis\\)

using Cons
using \( \text{Some literal} = \text{getWatch1 state clause} \)
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)
next
**case** False

show \(?thesis\\)

**proof** (cases \( \text{getNonWatchedUnfalsifiedLiteral (nth (getF ?state') clause) ?w1 ?w2 (getM ?state'}) \))

**case** (Some \( l' \))
hence $l' \, el \, (\text{nth} \,(\text{getF} \, ?\text{state}')) \, \text{clause}$

using $\text{getNonWatchedUnfalsifiedLiteralSomeCharacterization}$

by $\text{simp}$

let $?\text{state}'' = \text{setWatch2 clause } l' \, ?\text{state}'$

from $\text{Cons}(2)$

have $\text{InvariantWatchesEl (getF} \, ?\text{state}'') \, (\text{getWatch1} \, ?\text{state}'')$

$(\text{getWatch2} \, ?\text{state}'')$

using $(l' \, el \, (\text{nth} \,(\text{getF} \, ?\text{state}')) \, \text{clause})$:

unfolding $\text{InvariantWatchesEl-def}$

unfolding $\text{swapWatches-def}$

unfolding $\text{setWatch2-def}$

by $\text{auto}$

moreover

have $\text{getF} \, ?\text{state}'' = \text{getF} \, \text{state}$

unfolding $\text{swapWatches-def}$

unfolding $\text{setWatch2-def}$

by $\text{simp}$

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 $\text{Some}$

by $(\text{simp add: Let-def})$

next

case None

show $?\text{thesis}$

proof $(\text{cases literalFalse} \, ?w1 \,(\text{elements} \,(\text{getM} \, ?\text{state}')))$

case True

let $?\text{state}'' = ?\text{state}'[(\text{getConflictFlag} := \text{True}, \text{getConflict-Clause} := \text{clause})]$

from $\text{Cons}(2)$

have $\text{InvariantWatchesEl (getF} \, ?\text{state}'') \, (\text{getWatch1} \, ?\text{state}'')$

$(\text{getWatch2} \, ?\text{state}'')$

unfolding $\text{InvariantWatchesEl-def}$

unfolding $\text{swapWatches-def}$

by $\text{auto}$

moreover

have $\text{getF} \, ?\text{state}'' = \text{getF} \, \text{state}$

unfolding $\text{swapWatches-def}$

by $\text{simp}$

ultimately

show $?\text{thesis}$

using $\text{Cons}$
using \langle getWatch1 ?state' clause = Some ?w1 \rangle
using \langle getWatch2 ?state' clause = Some ?w2 \rangle
using \langle Some literal = getWatch1 state clause \rangle
using \langle \neg literalTrue ?w1 (elements (getM ?state')) \rangle
using None
using \langle literalFalse ?w1 (elements (getM ?state')) \rangle
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 getF ?state'' = getF state
  unfolding swapWatches-def
  unfolding setReason-def
  by simp
ultimately
show ?thesis
using Cons
using \langle getWatch1 ?state' clause = Some ?w1 \rangle
using \langle getWatch2 ?state' clause = Some ?w2 \rangle
using \langle Some literal = getWatch1 state clause \rangle
using \langle \neg literalTrue ?w1 (elements (getM ?state')) \rangle
using None
using \langle \neg literalFalse ?w1 (elements (getM ?state')) \rangle
by (simp add: Let-def)
qed
qed
qed
next
case False
let ?state' = state
let \?w1 = wa
have getWatch1 ?state' clause = Some \?w1
  using \langle getWatch1 state clause = Some wa \rangle
  unfolding swapWatches-def
  by auto
let \?w2 = wb
have getWatch2 ?state' clause = Some \?w2
  using \langle getWatch2 state clause = Some wb \rangle
  unfolding swapWatches-def

351
by auto
show \(?thesis
proof (cases literalTrue \(?w1 \:( \text{elements } (\text{getM } \?\text{state}')) )
  case True
  thus \(?thesis
  using Cons
  using (\neg \text{ Some literal } = \text{getWatch1 state clause})
  using (\text{getWatch1 } ?\text{state' clause } = \text{Some } \?w1);
  using (\text{getWatch2 } ?\text{state' clause } = \text{Some } \?w2);
  using (\text{literalTrue } \?w1 \:( \text{elements } (\text{getM } \?\text{state}')) )
  by (simp add: Let-def)
next
  case False
  show \(?thesis
  proof (cases getNonWatchedUnfalsifiedLiteral \:( \text{nth } (\text{getF } ?\text{state}'))
    \?w1 \?w2 \:( \text{getM } ?\text{state}'))
    case (Some l')
    hence l' el \:( \text{nth } (\text{getF } ?\text{state}'))
    using getNonWatchedUnfalsifiedLiteralSomeCharacterization
    by simp
    let \?state'' = setWatch2 clause l' \?\text{state}'
    from Cons
    have InvariantWatchesEl \:( \text{getF } ?\text{state}')(\?w1)
      \:( \text{getWatch1 } \?\text{state}'' (\text{getWatch2 } ?\text{state}'))
      (\text{getWatch1 } ?\text{state}')(\text{getWatch2 } ?\text{state}'' )
      using (l' el \:( \text{nth } (\text{getF } ?\text{state}'))
        \text{clause})
      unfolding InvariantWatchesEl-def
      unfolding setWatch2-def
      by auto
    moreover
    have \text{getF } ?\text{state}'' = \text{getF state}
      unfolding setWatch2-def
      by simp
    ultimately
    show \(?thesis
      using Cons
      using (\text{getWatch1 } ?\text{state'} clause = \text{Some } \?w1);
      using (\text{getWatch2 } ?\text{state'} clause = \text{Some } \?w2);
      using (\neg \text{ Some literal } = \text{getWatch1 state clause})
      using (\neg \text{ literalTrue } \?w1 \:( \text{elements } (\text{getM } ?\text{state}')) )
      using Some
      by (simp add: Let-def)
next
  case None
  show \(?thesis
  proof (cases literalFalse \?w1 \:( \text{elements } (\text{getM } ?\text{state}')) )
    case True
    let \?state'' = \?\text{state}'\{\text{getConflictFlag } := \text{True}, \text{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'
    (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
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
∀ (c::nat). c ∈ set Wl → 0 ≤ c ∧ c < length (getF state)
shows
let state′ = (notifyWatches-loop literal Wl newWl state) in
InvariantWatchesDiffer (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
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
  unfolding 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 (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 \( getF\ ?state'' = getF\ state \)

unfolding swapWatches-def

by simp

ultimately

show \( ?thesis \)

using Cons

using \( \langle\ getWatch1\ ?state'\ clause = Some\ ?w1\rangle \)

using \( \langle\ getWatch2\ ?state'\ clause = Some\ ?w2\rangle \)

using \( \langle\ Some\ literal = getWatch1\ state\ clause\rangle \)

using \( \langle\ \neg literalTrue\ ?w1\ (elements\ (getM\ ?state'))\rangle \)

using None

using \( \langle\ literalFalse\ ?w1\ (elements\ (getM\ ?state'))\rangle \)

by (simp add: Let-def)

next

case False

let \( ?state'' = setReason\ ?w1\ clause\ ?state'(getQ := (if \?w1

else \( 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

from Cons(3)

have InvariantWatchesDiffer \( (getF\ ?state'')\ (getWatch1\ ?state'')\)

\( (getWatch2\ ?state'')\)

unfolding InvariantWatchesDiffer-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 \( \langle\ getWatch1\ ?state'\ clause = Some\ ?w1\rangle \)

using \( \langle\ getWatch2\ ?state'\ clause = Some\ ?w2\rangle \)

using \( \langle\ Some\ literal = getWatch1\ state\ clause\rangle \)

using \( \langle\ \neg literalTrue\ ?w1\ (elements\ (getM\ ?state'))\rangle \)

using None

using \( \langle\ \neg literalFalse\ ?w1\ (elements\ (getM\ ?state'))\rangle \)

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 \textit{InvariantWatchesDiffer} \((\text{getF }\text{?state''})\) \((\text{getWatch1 }\text{?state''})\) \((\text{getWatch2 }\text{?state''})\)

using \((l' \neq \textit{?w1})\)
using \((\text{getWatch1 }\text{?state'} \text{ clause }= \text{Some }\textit{?w1})\)
using \((\text{getWatch2 }\text{?state'} \text{ clause }= \text{Some }\textit{?w2})\)
unfolding \textit{InvariantWatchesDiffer-def}
unfolding \textit{setWatch2-def}
by \textit{auto}
moreover
have \textit{getF }\text{?state'' }= \text{getF state}
unfolding \textit{setWatch2-def}
by \textit{simp}
ultimately
show \textit{?thesis}
using Cons
using \((\text{getWatch1 }\text{?state'} \text{ clause }= \text{Some }\textit{?w1})\)
using \((\text{getWatch2 }\text{?state'} \text{ clause }= \text{Some }\textit{?w2})\)
using \((\text{\neg Some literal }= \text{getWatch1 state clause})\)
using \((\text{\neg literalTrue }\textit{?w1} \text{ (elements }\text{getM }\text{?state'}))\)
using \textit{Some}
by \((\text{simp add: Let-def})\)

next
case None
show \textit{?thesis}
proof \((\text{cases literalFalse }\textit{?w1} \text{ (elements }\text{getM }\text{?state'}))\)
case True
let \textit{?state'' }= \textit{?state'}\((\textit{getConflictFlag }:= \text{True}, \textit{getConflict-Clause }:= \textit{clause})\)

from Cons(2)
have \textit{InvariantWatchesEl} \((\text{getF }\text{?state''})\) \((\text{getWatch1 }\text{?state''})\) \((\text{getWatch2 }\text{?state''})\)
unfolding \textit{InvariantWatchesEl-def}
by \textit{auto}
moreover
from Cons(3)
have \textit{InvariantWatchesDiffer} \((\text{getF }\text{?state''})\) \((\text{getWatch1 }\text{?state''})\) \((\text{getWatch2 }\text{?state''})\)
unfolding \textit{InvariantWatchesDiffer-def}
by \textit{auto}
moreover
have \textit{getF }\text{?state'' }= \text{getF state}
by \textit{simp}
ultimately
show \textit{?thesis}
using Cons
using \((\text{getWatch1 }\text{?state'} \text{ clause }= \text{Some }\textit{?w1})\)
using \((\text{getWatch2 }\text{?state'} \text{ clause }= \text{Some }\textit{?w2})\)
using \((\text{\neg Some literal }= \text{getWatch1 state clause})\)
using \(\neg \text{literalTrue } ?w1 \ \text{(elements } \text{getM } ?\text{state}')\)
using None
using \(\neg \text{literalFalse } ?w1 \ \text{(elements } \text{getM } ?\text{state}')\)
by (simp add: Let-def)

next

\text{case False}

let \(\text{?state''} = \text{setReason } ?w1 \ \text{clause } (?\text{state'}(?\text{getQ} := (\text{if } ?w1 \ \text{el } (\text{getQ } ?\text{state'}) \ \text{then } (\text{getQ } ?\text{state'}) \ \text{else } (\text{getQ } ?\text{state'}) @ (?w1))))\)

from \(\text{Cons}(2)\)
have \(\text{InvariantWatchesEl } (\text{getF } ?\text{state'')} \ (\text{getWatch1 } ?\text{state''}) (\text{getWatch2 } ?\text{state'')}\)
  unfolding \(\text{InvariantWatchesEl-def}\)
  unfolding \(\text{setReason-def}\)
  by auto
moreover
from \(\text{Cons}(3)\)
  have \(\text{InvariantWatchesDiffer } (\text{getF } ?\text{state'')} \ (\text{getWatch1 } ?\text{state''}) (\text{getWatch2 } ?\text{state'')}\)
  unfolding \(\text{InvariantWatchesDiffer-def}\)
  unfolding \(\text{setReason-def}\)
  by auto
moreover
have \(\text{getF } ?\text{state'''} = \text{getF } ?\text{state}\)
  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 \(\neg \text{Some literal } = \text{getWatch1 } \text{state clause}\)
using \(\neg \text{literalTrue } ?w1 \ \text{(elements } \text{getM } ?\text{state'})\)
using None
using \(\neg \text{literalFalse } ?w1 \ \text{(elements } \text{getM } ?\text{state'})\)
by (simp add: Let-def)
qed

lemma \text{InvariantWatchListsContainOnlyClausesFromFNotifyWatches-Loop}:
\begin{align*}
\text{fixes literal :: Literal} \ \text{and } \ Wl :: \text{nat list} \ \text{and } \text{newWl :: nat list} \ \text{and } \ \text{state :: State}
\end{align*}
assumes
\(\text{InvariantWatchListsContainOnlyClausesFromF } (\text{getWatchList } \text{state})\)
(getF state) and
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
∀ (c::nat). c ∈ set Wl ∨ c ∈ set newWl → 0 ≤ c ∧ c < length (getF state)
shows
let state' = (notifyWatches-loop literal Wl newWl state)
InvariantWatchListsContainOnlyClausesFromF (getWatchList state')
(getF state')
using assms
proof (induct Wl arbitrary: newWl state)
  case Nil
  thus ?case
    unfolding InvariantWatchListsContainOnlyClausesFromF-def
    by simp
next
  case (Cons clause Wl)
  from ∃ c. c ∈ set (clause ≠ Wl) ∨ c ∈ set newWl → 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 InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state') (getF ?state')
    unfolding 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)
    unfolding InvariantWatchListsContainOnlyClausesFromF-def
    unfolding swapWatches-def
    unfolding setWatch2-def
    by auto
moreover
from Cons(3)
  have InvariantWatchesEl (getF ?state) (getWatch1 ?state) (getWatch2 ?state)
    unfolding InvariantWatchesEl-def
    unfolding swapWatches-def
    unfolding setWatch2-def
    by auto

362
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 (∼ 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, getConflictClause := clause⟩

from Cons(2)
have InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state'') (getF ?state'')
  unfolding 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 (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 (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

next 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

364
show \( ?\text{thesis} \)

\[\text{proof} \ (\text{cases literal}\text{True} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}')))\]

\[\text{case True} \]

\[\text{thus } ?\text{thesis} \]

\[\text{using Cons} \]

\[\text{using } (\neg \text{Some literal} = \text{getWatch1 state clause}) \]

\[\text{using } (\text{getWatch1} \ ?\text{state'} clause = \text{Some} \ ?w1) \]

\[\text{using } (\text{getWatch2} \ ?\text{state'} clause = \text{Some} \ ?w2) \]

\[\text{using } (\text{literal}\text{True} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'))) \]

\[\text{by } (\text{simp add: Let-def}) \]

\[\text{next} \]

\[\text{case False} \]

\[\text{show } ?\text{thesis} \]

\[\text{proof} \ (\text{cases getNonWatchedUnfalsifiedLiteral} \ (\text{nth} \ (\text{getF} \ ?\text{state'}) \ 
\text{clause}) \ ?w1 \ ?w2 \ (\text{getM} \ ?\text{state}')) \]

\[\text{case } (\text{Some} \ l') \]

\[\text{hence } l' el \ (\text{nth} \ (\text{getF} ?\text{state'}) \ 
\text{clause}) \]

\[\text{using } \text{getNonWatchedUnfalsifiedLiteralSomeCharacterization} \]

\[\text{by simp} \]

\[\text{let } ?\text{state'}' = \text{setWatch2} \ 
\text{clause} \ l' \ ?\text{state'} \]

\[\text{from Cons(2)} \]

\[\text{have } \text{InvariantWatchListsContainOnlyClausesFromF} \ (\text{getWatchList} \ ?\text{state'}') \ (\text{getF} \ ?\text{state'}) \]

\[\text{using } (\text{clause} < \text{length} \ (\text{getF} \ 
\text{state})) \]

\[\text{unfolding setWatch2-def} \]

\[\text{unfolding InvariantWatchListsContainOnlyClausesFromF-def} \]

\[\text{by auto} \]

\[\text{moreover} \]

\[\text{from Cons(3)} \]

\[\text{have } \text{InvariantWatchesEl} \ (\text{getF} ?\text{state'}) \ (\text{getWatch1} ?\text{state'}') \ (\text{getWatch2} ?\text{state'}') \]

\[\text{using } (l' el \ (\text{nth} \ (\text{getF} \ ?\text{state'}) \ 
\text{clause}) \]

\[\text{unfolding InvariantWatchesEl-def} \]

\[\text{unfolding setWatch2-def} \]

\[\text{by auto} \]

\[\text{moreover} \]

\[\text{have } (\text{getF} \ 
\text{state}) = (\text{getF} \ ?\text{state'}) \]

\[\text{unfolding setWatch2-def} \]

\[\text{by simp} \]

\[\text{ultimately} \]

\[\text{show } ?\text{thesis} \]

\[\text{using Cons} \]

\[\text{using } (\text{getWatch1} ?\text{state'} clause = \text{Some} \ ?w1) \]

\[\text{using } (\text{getWatch2} ?\text{state'} clause = \text{Some} \ ?w2) \]

\[\text{using } (\neg \text{Some literal} = \text{getWatch1 state clause}) \]

\[\text{using } (\neg \text{literal}\text{True} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'))) \]

\[\text{using Some} \]
next
case None
  show ?thesis
proof (cases literalFalse ?w1 (elements (getM ?state')))
  case True
    let ?state" = ?state' (getConflictFlag := True, getConflict-Clause := clause)
  next
case False
  let ?state" = setReason ?w1 clause (?state' (getQ := if ?w1 el (getQ ?state') then (getQ ?state') else (getQ ?state') @ [?w1])))
next
case False
  let ?state" = setReason ?w1 clause (?state' (getQ := if ?w1 el (getQ ?state') then (getQ ?state') else (getQ ?state') @ [?w1])))
next
  case None
    from Cons(3)
    have InvariantWatchesEl (getF ?state") (getWatch1 ?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)
show \( ?\text{thesis} \)
using \( \text{Cons} \)
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 \( \langle \neg \text{Some} \ \text{l literal} = \text{getWatch1} \ \text{state clause} \rangle \)
using \( \langle \neg \text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}')) \rangle \)
using \( \text{None} \)
using \( \langle \neg \text{literalFalse} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}')) \rangle \)
by \( \text{(simp add: Let-def)} \)
qed
qed
qed
qed

lemma \( \text{InvariantWatchListsCharacterizationNotifyWatchesLoop} : \)
fixes \( \text{l literal :: Literal \ and \ Wl :: nat list \ and \ newWl :: nat list \ and \ state :: State} \)
assumes
\( \text{InvariantWatchesEl} \ (\text{getF} \ \text{state}) \ (\text{getWatch1} \ \text{state}) \ (\text{getWatch2} \ \text{state}) \)
and
\( \text{InvariantWatchesDiffer} \ (\text{getF} \ \text{state}) \ (\text{getWatch1} \ \text{state}) \ (\text{getWatch2} \ \text{state}) \)
\( \text{InvariantWatchListsUniq} \ (\text{getWatchList} \ \text{state}) \)
\( \forall \ (c :: \text{nat}). \ c \in \text{set Wl} \implies \theta \leq c \wedge c < \text{length} \ (\text{getF} \ \text{state}) \)
\( \forall \ (c :: \text{nat}). \ (l :: \text{Literal}). \ l \neq \text{l literal} \implies \)
\( (c \in \text{set} \ (\text{getWatchList} \ \text{state} \ l)) = (\text{Some} \ l = \text{getWatch1} \ \text{state} \ c) \)
\( \forall \ (c :: \text{nat}). \ (c \in \text{set newWl} \vee c \in \text{set Wl}) = (\text{Some literal} = (\text{getWatch1} \ \text{state} \ c) \vee \text{Some literal} = (\text{getWatch2} \ \text{state} \ c)) \)
\( \text{set Wl} \cap \text{set newWl} = \{\} \)
uniq Wl
uniq newWl
shows
let \( \text{state}' = (\text{notifyWatches-loop} \ \text{l literal} \ \text{Wl} \ \text{newWl} \ \text{state}) \) in
\( \text{InvariantWatchListsCharacterization} \ (\text{getWatchList} \ \text{state}') \ (\text{getWatch1} \ \text{state}') \ (\text{getWatch2} \ \text{state}') \) 
\( \wedge \text{InvariantWatchListsUniq} \ (\text{getWatchList} \ \text{state}') \)
using assms
proof \( \text{(induct Wl arbitrary: newWl state)} \)
\text{case Nil} 
thus \( ?\text{case} \)
unfolding \( \text{InvariantWatchListsCharacterization-def} \)
unfolding \( \text{InvariantWatchListsUniq-def} \)
by simp
next
\text{case (Cons clause Wl')}
from \( \text{uniq} \ (\text{clause} \ # \ Wl') \)
have \( \text{clause} \notin \text{set Wl'} \)
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')))
unfolding \text{swapWatches-def}

by \text{auto}

moreover

from \text{Cons(3)}

have \text{InvariantWatchesDiffer} (\text{getF \textit{?state'}}) (\text{getWatch1 \textit{?state'}}) (\text{getWatch2 \textit{?state'}})

unfolding \text{InvariantWatchesDiffer-def}

unfolding \text{swapWatches-def}

by \text{auto}

moreover

from \text{Cons(4)}

have \text{InvariantWatchListsUniq} (\text{getWatchList \textit{?state'}})

unfolding \text{InvariantWatchListsUniq-def}

unfolding \text{swapWatches-def}

by \text{auto}

moreover

have (\text{getF \textit{?state'}}) = (\text{getF \textit{state}}) \text{ and } (\text{getWatchList \textit{?state'}}) = (\text{getWatchList \textit{state}})

unfolding \text{swapWatches-def}

by \text{auto}

moreover

have \forall \textit{c, l.} \textit{l \neq literal} \implies

(c \in \text{set}(\text{getWatchList \textit{?state'}}\textit{l})) =

(Some \textit{l} = \text{getWatch1 \textit{?state'}}\textit{c} \lor Some \textit{l} = \text{getWatch2 \textit{?state'}}\textit{c})

\text{c)

using \text{Cons(6)}

using (\text{getWatchList \textit{?state'}}) = (\text{getWatchList \textit{state}})

using \text{swapWatchesEffect}

by \text{auto}

moreover

have \forall \textit{c.} (\textit{c \in set(\text{clause \# newWl}) \lor c \in set Wl'}) =

(Some \text{literal} = \text{getWatch1 \textit{?state'}}\textit{c} \lor Some \text{literal} = \text{getWatch2 \textit{?state'}}\textit{c})

\text{c)

using \text{Cons(7)}

using \text{swapWatchesEffect}

by \text{auto}

ultimately

show \text{?thesis}

using \text{Cons(1)[of \textit{?state'} \text{ clause \# newWl]}}

using \text{Cons(5)}

using (Some \text{literal} = \text{getWatch1 \textit{state} clause})

using (\text{getWatch1 \textit{?state'} clause = Some \textit{w1})}

using (\text{getWatch2 \textit{?state'} clause = Some \textit{w2})}

using (\text{literalTrue \textit{?w1} (elements (getM \textit{?state'}))})

using (\text{uniq Wl'})

using (\text{uniq (clause \# newWl)})

using (set Wl' \cap set (clause \# newWl) = \{\})

by (simp add: Let-def)

next
\textbf{proof (cases \text{getNonWatchedUnfalsifiedLiteral} (\text{nth} (\text{getF ?state'}) clause) \ ?w1 \ ?w2 (\text{getM ?state'}))}

\begin{itemize}
\item \text{case (Some l')}
\item \text{hence \( l' \in \text{nth} (\text{getF ?state'}) \\text{clause}) \land (l' \neq \text{literal} l' \neq l \neq \?w1 \land \neq \?w2)}
\end{itemize}

\begin{itemize}
\item \text{using \text{getNonWatchedUnfalsifiedLiteralSomeCharacterization}}
\item \text{using (getWatch1 ?state' clause = Some \?w1)}
\item \text{using (getWatch2 ?state' clause = Some \?w2)}
\item \text{using (Some literal = getWatch1 state clause)}
\item \text{unfolding \text{swapWatches-def}}
\item \text{by auto}
\end{itemize}

\begin{itemize}
\item \text{let \?state'' = \text{setWatch2 clause l'} ?state'}
\end{itemize}

\begin{itemize}
\item \text{from Cons(2)}
\item \text{have \text{InvariantWatchesEl} (getF ?state'') (getWatch1 ?state'')} (getWatch2 ?state'')
\item \text{using (l' el \text{nth} (\text{getF ?state'}) \text{clause})}
\item \text{unfolding \text{InvariantWatchesEl-def}}
\item \text{unfolding \text{swapWatches-def}}
\item \text{unfolding \text{setWatch2-def}}
\item \text{by auto}
\item \text{moreover}
\item \text{from Cons(3)}
\item \text{have \text{InvariantWatchesDiffer} (getF ?state'') (getWatch1 ?state'')} (getWatch2 ?state'')
\item \text{using (getWatch1 ?state' clause = Some \?w1)}
\item \text{using (l' \neq \?w1)}
\item \text{unfolding \text{InvariantWatchesDiffer-def}}
\item \text{unfolding \text{swapWatches-def}}
\item \text{unfolding \text{setWatch2-def}}
\item \text{by simp}
\item \text{moreover}
\item \text{have clause \notin \text{set (getWatchList state l')}}
\item \text{using (l' \neq \text{literal} l')}
\item \text{using (l' \neq \?w1) \land (l' \neq \?w2)}
\item \text{using (getWatch1 ?state' clause = Some \?w1)}
\item \text{using (getWatch2 ?state' clause = Some \?w2)}
\item \text{using Cons(6)}
\item \text{unfolding \text{swapWatches-def}}
\item \text{by simp}
\item \text{with Cons(4)}
\item \text{have \text{InvariantWatchListsUniq (getWatchList ?state'')}}
\item \text{unfolding \text{InvariantWatchListsUniq-def}}
\item \text{unfolding \text{swapWatches-def}}
\item \text{unfolding \text{setWatch2-def}}
\item \text{using \text{uniqAppendIff}}
\end{itemize}
by force
moreover
have \((\text{getF state}′′) = (\text{getF state})\) and
\((\text{getWatchList state}′′) = (\text{getWatchList state})(l′:= \text{clause} \#\)
\((\text{getWatchList state} l′′))\)
  unfolding swapWatches-def
  unfolding setWatch2-def
by auto
moreover
have \(\forall \ c \ l. \ l \neq \text{literal} \implies \)
\((c \in \text{set (getWatchList state} l)) =\)
\((\text{Some l = getWatch1 state}′′ c \lor \text{Some l = getWatch2 state}′′ c)\)
c)
proof –
{  
  fix \(c::\text{nat} \ \text{and} \ l::\text{Literal}\)
  assume \(l \neq \text{literal}\)
  have \((c \in \text{set (getWatchList state} l)) = (\text{Some l = getWatch1 state}′′ c \lor \text{Some l = getWatch2 state}′′ c)\)
  proof \(\text{(cases c = clause)}\)
    case True
    show \(\text{thesis}\)
    proof
      cases \(l = l′\)
      case True
      thus \(\text{thesis}\)
      using \(\langle c = \text{clause} \rangle\)
      unfolding setWatch2-def
      by simp
    next
    case False
    thus \(\text{thesis}\)
    using Cons(6)
    using \(\langle \text{getWatch1 state}′′ \text{ clause = Some ?w1} \rangle\)
    using \(\langle \text{getWatch2 state}′′ \text{ clause = Some ?w2} \rangle\)
    using \(\langle \text{Some literal = getWatch1 state clause} \rangle\)
    using \(\langle c = \text{clause} \rangle\)
    using swapWatchesEffect
    unfolding swapWatches-def
    unfolding setWatch2-def
    by simp
  qed
next
  case False
  thus \(\text{thesis}\)
  using Cons(6)
using \( l \neq \text{literal} \)

using \( \langle \text{getWatchList ?state}' = (\text{getWatchList state})(l') := \text{clause} \# (\text{getWatchList state} l') \rangle \)

using \( \langle c \neq \text{clause} \rangle \)

unfolding \text{setWatch2-def}

by auto

qed

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

by simp

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) \langle \text{clause} \notin \text{set Wl}' \rangle \langle c \in \text{set newWl} \lor c \in \text{set Wl}' \rangle

have \( c \neq \text{clause} \)

by auto

show \( ?\text{thesis} \)

using \( \langle \text{Some literal} = \text{getWatch1 state} c \lor \text{Some literal} = \text{getWatch2 state} c \rangle \)

using \( c \neq \text{clause} \)

using \text{swapWatchesEffect}

unfolding \text{setWatch2-def}

by simp

qed

next

assume \text{Some literal} = \text{getWatch1 ?state}'' c \lor \text{Some literal}
\[ \text{getWatch2 } \forall \text{c} \in \text{set newWl} \lor \text{c} \in \text{set Wl}' \]

**proof**

\[ \text{have Some literal } \neq \text{getWatch1 } \forall \text{c} \in \text{set newWl} \lor \text{c} \in \text{set Wl}' \]

**using** \((l' \neq \text{literal})\)

**using** \((\text{clause } \neq \text{length (getF state)})\)

**using** \((\text{InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)})\)

**using** \((\text{getWatch1 ?state' clause } = \text{Some ?w1})\)

**using** \((\text{getWatch2 ?state' clause } = \text{Some ?w2})\)

**using** \((\text{Some literal } = \text{getWatch1 state clause})\)

**unfolding** \((\text{InvariantWatchesDiffer-def})\)

**unfolding** \((\text{setWatch2-def})\)

**unfolding** \((\text{swapWatches-def})\)

**by** \((\text{auto})\)

**thus** \((\text{thesis})\)

**using** \((\text{Some literal } = \text{getWatch1 ?state'' c } \lor \text{Some literal } = \text{getWatch2 ?state'' c})\)

**using** \((\text{Cons(7)})\)

**using** \((\text{swapWatchesEffect})\)

**unfolding** \((\text{setWatch2-def})\)

**by** \((\text{auto split: split-if-asm})\)

**qed**

**qed**

**moreover**

\[ \forall \text{c} \in \text{set (clause } \neq \text{newWl}) \lor \text{c} \in \text{set Wl}') = \]

\((\text{Some literal } = \text{getWatch1 ?state' c } \lor \text{Some literal } = \text{getWatch2 ?state' c})\)

**using** \((\text{Cons(7)})\)

**using** \((\text{swapWatchesEffect})\)

**by** \((\text{auto})\)

**ultimately**

**show** \((\text{thesis})\)

**using** \((\text{Cons(1)[of ?state'' newWl]})\)

**using** \((\text{Cons(5)})\)

**using** \((\text{uniq newWl})\)

**using** \((\text{uniq newWl})\)

**using** \((\text{set Wl} \cap \text{set (clause } \neq \text{newWl}) = \{\})\)

**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** \((\text{Some literal } = \text{getWatch1 state clause})\)

**using** \((\neg \text{literalTrue ?w1 (elements (getM ?state'))})\)

**using** \((\text{Some})\)

**by** \((\text{simp add: Let-def fun-upd-def})\)

**next**

**case** \((\text{None})\)

373
show \( \text{?thesis} \)

proof (cases literalFalse \( \text{?w1} \) (elements (getM \( \text{?state} \'))))

case True

let \( \text{?state''} = \text{?state}' (\text{getConflictFlag := True, getConflict-Clause := clause}) \)

from Cons(2)
have InvariantWatchesEl (getF \( \text{?state''} \)) (getWatch1 \( \text{?state''} \))
\(\text{(getWatch2 \( \text{?state''} \))}\)
unfolding InvariantWatchesEl-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 swapWatches-def
by auto
moreover
from Cons(4)
have InvariantWatchListsUniq (getWatchList \( \text{?state''} \))
unfolding InvariantWatchListsUniq-def
unfolding swapWatches-def
by auto
moreover
have (getF \( \text{state} \)) = (getF \( \text{?state''} \)) and (getWatchList \( \text{state} \)) = (getWatchList \( \text{?state''} \))
unfolding swapWatches-def
by auto
moreover
have \( \forall \ c \ l. \ l \neq \text{literal} \rightarrow (c \in \text{set (getWatchList \( \text{?state''} \) l})) = (\text{Some l = getWatch1 \( \text{?state''} \) c} \lor \text{Some l = getWatch2 \( \text{?state''} \) c}) \)
using Cons(6)
using \( (\text{getWatchList \( \text{state} \)) = (\text{getWatchList \( \text{?state''} \))}) \)
using swapWatchesEffect
by auto
moreover
have \( \forall \ c. (c \in \text{set (clause \( \# \) newWl)} \lor c \in \text{set \( Wl \)})) = (\text{Some literal = getWatch1 \( \text{?state''} \) c} \lor \text{Some literal = getWatch2 \( \text{?state''} \) c}) \)
using Cons(7)
using swapWatchesEffect
by auto
ultimately
show \( \text{?thesis} \)
using Cons(1)\([\text{of \( \text{?state''} \) clause \( \# \) newWl}) \)

374
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)

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
    from Cons(3)
    have InvariantWatchesDiffer (getF ?state’’)(getWatch1
    ?state’’)(getWatch2 ?state’’)
      unfolding InvariantWatchesDiffer-def
      unfolding swapWatches-def
      unfolding setReason-def
      by auto
    moreover
    from Cons(4)
    have InvariantWatchListsUniq (getWatchList ?state’’)
      unfolding InvariantWatchListsUniq-def
      unfolding swapWatches-def
      unfolding setReason-def
      by auto
    moreover
    have (getF state) = (getF ?state’’)
     and (getWatchList state) = (getWatchList ?state’’)
     unfolding swapWatches-def
     unfolding setReason-def
     by auto
    moreover
    have ∀ c l. l ≠ literal →
      (c ∈ set (getWatchList ?state’’ l)) =
      (Some l = getWatch1 ?state’’ c ∨ Some l = getWatch2
    ?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

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next

next
by force

show ?thesis
proof (cases literalTrue ?w1 (elements (getM ?state')))
  case True
    from Cons(7) have
      \( \forall c. (c \in \text{set (\text{clause} \# \text{newWl})} \lor c \in \text{set \text{Wl}'}) = \)
      \( (\text{Some literal} = \text{getWatch1 state } c \lor \text{Some literal} = \text{getWatch2 state } c) \)
    by auto
    thus ?thesis
      using Cons(1)[of ?state' clause \# \text{newWl}]
      using Cons(2) Cons(3) Cons(4) Cons(5) Cons(6)
      using \( \neg \text{Some literal} = \text{getWatch1 state clause} \)
      using \( \text{getWatch1 ?state'} clause = \text{Some } ?w1 \)
      using \( \text{getWatch2 ?state'} clause = \text{Some } ?w2 \)
      using \( \text{literalTrue } ?w1 \) (elements (getM ?state'))
      using (uniq (clause \# \text{newWl}))
      using (uniq \text{Wl}')
      using (set \text{Wl}' \cap \text{set (\text{clause} \# \text{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' \# literal l' \# ?w1 l'
            \# ?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 \( l' \neq ?w1 \)

unfolding InvariantWatchesDiffer-def
unfolding setWatch2-def
by simp
moreover
have clause \( \notin \) set \((getWatchList \text{ state } l')\)
using \( l' \neq \text{ literal} \)
using \( l' \neq ?w1 \) \( l' \neq ?w2 \)
using \((getWatch1 \text{ ?state'} \text{ clause } = \text{ Some } ?w1)\)
using \((getWatch2 \text{ ?state'} \text{ clause } = \text{ Some } ?w2)\)
using Cons(6)
by simp

with Cons(4)

have InvariantWatchListsUniq \((getWatchList \text{ ?state'')}\)
unfolding InvariantWatchListsUniq-def
unfolding setWatch2-def

using uniqAppendIff
by force
moreover
have \((getF \text{ ?state'')} = (getF \text{ state}) \text{ and} \)

\((getWatchList \text{ ?state'')} = \text{ (getWatchList \text{ state})(l' := \text{ clause } #} \text{ (getWatchList \text{ state } l')})\)

unfolding setWatch2-def
by auto
moreover

have \( \forall c \text{ l. } l \neq \text{ literal } \rightarrow \)

\((c \in \text{ set } \text{ (getWatchList \text{ ?state'')} l)) = \)

\((\text{ Some } l = \text{ getWatch1 \text{ ?state'')} c \lor \text{ Some } l = \text{ getWatch2 \text{ ?state'')} c)\)

proof
{
  fix c::nat and l::Literal
  assume l \( \neq \) literal
  have \((c \in \text{ set } \text{ (getWatchList \text{ ?state'')} l)) = \text{ (Some } l = \text{ getWatch1 \text{ ?state'')} c \lor \text{ Some } l = \text{ getWatch2 \text{ ?state'')} c)\)
  proof (cases c = \text{ clause})
    case True
    show \(?thesis\)
    proof (cases \( l = l' \))
      case True
      thus \(?thesis\)
      using \( c = \text{ clause} \)
      unfolding setWatch2-def
      by simp
    next
    case False
    show \(?thesis\)
    using Cons(6)
    using \((getWatchList \text{ ?state'')} = \text{ (getWatchList \text{ state})(l' \}

378
\[ := \text{clause} \# (\text{getWatchList state l}') \]

- using \( l \neq l' \)
- using \( l \neq \text{literal} \)
- using \( \text{getWatch1 ?state'} \text{ clause} = \text{Some ?w1} \)
- using \( \text{getWatch2 ?state'} \text{ clause} = \text{Some ?w2} \)
- using \( c = \text{clause} \)

unfolding \text{setWatch2-def}

by simp

qed

to
next

\text{case False}

thus \( \text{thesis} \)

using \( \text{Cons}(6) \)

using \( l \neq \text{literal} \)

using \( \text{getWatchList ?state'} = (\text{getWatchList state})(l') \)

\[ := \text{clause} \# (\text{getWatchList state l}') \]

- using \( c \neq \text{clause} \)

unfolding \text{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 \( \text{Cons}(8) \) \langle \text{clause} \notin \text{set Wl'} \rangle \langle c \in \text{set newWl} \lor c \in \text{set Wl'} \rangle
have \( c \neq \text{clause} \)
by \(\text{auto}\)

show \(?\text{thesis}\)
using \(\langle \text{Some literal} = \text{getWatch1 state} \ c \lor \text{Some literal} = \text{getWatch2 state} \ c \rangle\)
using \(\langle c \neq \text{clause} \rangle\)
unfolding \(\text{setWatch2-def}\)
by \(\text{simp}\)
qed

next
assume \(\text{Some literal} = \text{getWatch1 state'' c} \lor \text{Some literal} = \text{getWatch2 state'' c}\)

show \(c \in \text{set newWl} \lor c \in \text{set Wl'}\)
proof –

have \(\text{Some literal} \neq \text{getWatch1 state'' clause} \land \text{Some literal} \neq \text{getWatch2 state'' clause}\)
using \(\langle \text{l'} \neq \text{literal} \rangle\)
using \(\langle \text{clause} < \text{length (getF state)} \rangle\)
using \(\langle \text{InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)} \rangle\)
using \(\langle \text{getWatch1 state' clause} = \text{Some ?w1} \rangle\)
using \(\langle \text{getWatch2 state' clause} = \text{Some ?w2} \rangle\)
using \(\langle \text{Some literal} = \text{getWatch2 state clause} \rangle\)
unfolding \(\text{InvariantWatchesDiffer-def}\)

unfolding \(\text{setWatch2-def}\)
by \(\text{auto}\)
thus \(?\text{thesis}\)
using \(\langle \text{Some literal} = \text{getWatch1 state'' c} \lor \text{Some literal} = \text{getWatch2 state'' c}\rangle\)
using \(\langle \text{Cons(7)} \rangle\)
unfolding \(\text{setWatch2-def}\)
by \(\langle \text{auto split: split-if-asm} \rangle\)
qed

qed

qed

moreover
have \(\forall c. (c \in \text{set (clause \# newWl)} \lor c \in \text{set Wl'}) = (\text{Some literal} = \text{getWatch1 state'} c \lor \text{Some literal} = \text{getWatch2 state'} c)\)
using \(\langle \text{Cons(7)} \rangle\)
by \(\text{auto}\)
ultimately
show \(?\text{thesis}\)
using \(\langle \text{Cons(1)[of state'' newWl]} \rangle\)
using \(\langle \text{Cons(5)} \rangle\)
using \(\langle \text{uniq Wl'} \rangle\)
using \(\langle \text{uniq newWl} \rangle\)
using (set $W_l' \cap \text{set (clause # newWl)} = \{\})
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 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
  by auto
moreover
from Cons(3)
have InvariantWatchesDiffer (getF ?state') (getWatch1 ?state'') (getWatch2 ?state'')
  unfolding InvariantWatchesDiffer-def
  by auto
moreover
from Cons(4)
have InvariantWatchListsUniq (getWatchList ?state'')
  unfolding InvariantWatchListsUniq-def
  by auto
moreover
have (getF state) = (getF ?state'')
  by auto
moreover
have \(\forall c. l. l \neq \text{literal} \rightarrow\)
  \(\{c \in \text{set (getWatchList ?state'')(l)} =\)
  \((\text{Some } l = \text{getWatch1 ?state''} \lor \text{Some } l = \text{getWatch2 \ ?state''})\ c\)

using Cons(6)
  by simp
moreover
have \(\forall c. (c \in \text{set (clause # newWl)} \lor c \in \text{set Wl'}) =\)
  \((\text{Some literal = getWatch1 ?state''} \lor \text{Some literal = getWatch2 \ ?state''})\ c\)
using Cons(7)
  by auto
ultimately
have let state' = notifyWatches-loop literal Wl' (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 WL'
using uniq (clause # newWL);
using (set WI' \cap 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
have \( \forall \, c \in \mathbb{N}, \, l \neq \text{literal} \rightarrow (c \in \text{set} \left( \text{getWatchList} \, ?\text{state}'' \, l \right)) = \) 
\( (\text{Some} \, l = \text{getWatch1} \, ?\text{state}'' \, c \lor \text{Some} \, l = \text{getWatch2} \, ?\text{state}'' \, c) \)

using Cons(6)
unfolding setReason-def
by auto

moreover
have \( \forall \, c \in \text{set} \left( \text{clause} \# \text{newWl} \right) \lor \, c \in \text{set} \, \text{Wl} \) = 
\( (\text{Some} \, \text{literal} = \text{getWatch1} \, ?\text{state}'' \, c \lor \text{Some} \, \text{literal} = \text{getWatch2} \, ?\text{state}'' \, c) \)

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 \( \left( \text{getWatch1} \, ?\text{state}'' \, \text{clause} = \text{Some} \, w1 \right) \)
using \( \left( \text{getWatch2} \, ?\text{state}'' \, \text{clause} = \text{Some} \, w2 \right) \)
using \( (\neg \, \text{Some} \, \text{literal} = \text{getWatch1} \, ?\text{state}'' \, \text{clause}) \)
using \( (\neg \, \text{literalTrue} \, w1 \, \text{elements} \left( \text{getM} \, ?\text{state}' \right)) \)
using \( \text{None} \)
using \( (\neg \, \text{literalFalse} \, w1 \, \text{elements} \left( \text{getM} \, ?\text{state}' \right)) \)
using \( \text{uniq} \, \text{Wl} \)
using \( \text{uniq} \, \text{(clause} \# \text{newWl}) \)
using \( (\text{set} \, \text{Wl}'' \cap \text{set} \left( \text{clause} \# \text{newWl} \right) = \{\}) \)
by (simp add: Let-def)

qed

qed

qed

qed


lemma NotifyWatchesLoopWatchCharacterizationEffect:
fixes \text{literal} :: \text{Literal} and \text{Wl} :: \text{nat list} and \text{newWl} :: \text{nat list} and \text{state} :: \text{State}

assumes
\( \text{InvariantWatchesEl} \left( \text{getF} \, \text{state} \right) \left( \text{getWatch1} \, \text{state} \right) \left( \text{getWatch2} \, \text{state} \right) \)

and
\( \text{InvariantWatchesDiffer} \left( \text{getF} \, \text{state} \right) \left( \text{getWatch1} \, \text{state} \right) \left( \text{getWatch2} \, \text{state} \right) \)

and
\( \text{InvariantConsistent} \, \left( \text{getM} \, \text{state} \right) \)

\( \text{InvariantUniq} \, \left( \text{getM} \, \text{state} \right) \)

\( \text{InvariantWatchCharacterization} \left( \text{getF} \, \text{state} \right) \left( \text{getWatch1} \, \text{state} \right) \left( \text{getWatch2} \, \text{state} \right) \)

\( \forall \, (c::\mathbb{N}), \, c \in \text{set} \, \text{Wl} \rightarrow 0 \leq c \land c < \text{length} \left( \text{getF} \, \text{state} \right) \)

\( \text{getM} \, \text{state} = \text{M} \, @ \left( \left[ \text{opposite} \left( \text{literal} \right), \text{decision} \right] \right) \)

\( \text{uniq} \, \text{Wl} \)
\forall (c :: \text{nat}). c \in \text{set } Wl \longrightarrow \text{Some literal } = (\text{getWatch1 state } c) \lor \text{Some literal } = (\text{getWatch2 state } c)

shows

let state' = \text{notifyWatches-loop literal } Wl \text{ newWl state in}

\forall (c :: \text{nat}). c \in \text{set } Wl \longrightarrow (\forall w1 w2. (\text{Some } w1 = (\text{getWatch1 state'} c) \land \text{Some } w2 = (\text{getWatch2 state'} c)) \longrightarrow \\
\text{watchCharacterizationCondition } w1 w2 (\text{getM state'}) (\text{nth (getF state') } c) \land \\
\text{watchCharacterizationCondition } w2 w1 (\text{getM state'}) (\text{nth (getF state') } c))

using \text{assms}

proof (\text{induct } Wl \text{ arbitrary: newWl state})

\text{case Nil}

\text{thus ?case}

by simp

next

\text{case (Cons clause } Wl')

\text{from } \forall (c :: \text{nat}). c \in \text{set (\text{clause } \# Wl'}) \longrightarrow 0 \leq c \land c < \text{length (getF state)}:

\text{have } 0 \leq \text{clause } \land \text{clause < length (getF state)}

\text{by auto}

\text{then obtain } wa :: \text{Literal and wb :: Literal}

\text{where getWatch1 state clause } = \text{Some } wa \text{ and getWatch2 state clause } = \text{Some } wb

\text{clause } = \text{Some } wb

using Cons

unfolding \text{InvariantWatchesEl-def}

by auto

\text{have uniq Wl' clause } \notin \text{ set Wl'}

using Cons(9)

by (auto simp add: uniqAppendIff)

show ?case

proof (cases \text{Some literal } = \text{getWatch1 state clause})

\text{case True}

let ?state' = \text{swapWatches clause state}

let ?w1 = wb

\text{have getWatch1 ?state' clause } = \text{Some } ?w1

using \text{(getWatch2 state clause } = \text{Some } wb)

unfolding \text{swapWatches-def}

by auto

let ?w2 = wa

\text{have getWatch2 ?state' clause } = \text{Some } ?w2

using \text{(getWatch1 state clause } = \text{Some } wa)

unfolding \text{swapWatches-def}

by auto

with \text{True have}

?w2 = \text{literal}

unfolding \text{swapWatches-def}
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 (0 ≤ clause ∧ 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 (getWatch2 ?state' clause = Some ?w2)
    using (0 ≤ clause ∧ clause < length (getF state))
  unfolding InvariantWatchesDiffer-def
  unfolding swapWatches-def
by auto

have ¬ literalFalse ?w2 (elements M)
  using (?w2 = literal)
  using Cons(5)
  using Cons(8)
  unfolding InvariantUniq-def
by (simp add: uniqAppendIff)

show ?thesis
proof (cases literalTrue ?w1 (elements (getM ?state'))) case True

  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 getM ?state' = getM state
  getF ?state' = getF state
  unfolding swapWatches-def
  by auto
moreover
have \( \forall (c::\text{nat}). \ c \in \text{set } Wl' \longrightarrow \text{Some literal} = (\text{getWatch1 ?state'} c) \lor \text{Some literal} = (\text{getWatch2 ?state'} c) \)
  using Cons(10)
  unfolding swapWatches-def
  by auto
moreover
have getWatch1 ?fState clause = getWatch1 ?state' clause \land 
  getWatch2 ?fState clause = getWatch2 ?state' clause
  using (\text{clause} \notin \text{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) \land 
  watchCharacterizationCondition ?w2 ?w1 (getM ?fState)
  (getF ?fState ! clause)
proof–
  have (getM ?fState) = (getM state) \land (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⟩
using 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)
using 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 el (nth (getF state) clause))
unfolding watchCharacterizationCondition-def
using 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 \((\text{simp add: Let-def})\)

next
  case False
  show \(\vartheta\)thesis
  proof \((\cases\text{getNonWatchedUnfalsifiedLiteral nth (getF state')}\)
  case \(\text{Some l'}\)
  hence \(l' \in \text{nth (getF state')}\) \(l' \neq ?w1 \neq ?w2 \rightarrow\)
literalFalse \(l' \in \text{elements (getM state')}\)
  using \(\text{getWatch1 state'}\) clause = Some \(?w1\)
  using \(\text{getWatch2 state'}\) clause = Some \(?w2\)
  using \(\text{getNonWatchedUnfalsifiedLiteralSomeCharacterization}\)
  by auto

  let \(\text{state''} = \text{setWatch2 clause l'} \text{ state'}\)
  let \(\text{?FState} = \text{notifyWatches-loop literal WL'} \text{ newWL} \text{ state''}\)

  from Cons(2)
  have \(\text{InvariantWatchesEl (getF state')}\) \(\text{(getWatch2 state')}\)
  (getWatch2 state')
  using \(\text{l' el (nth (getF state')}\) clause\):
  unfolding \(\text{InvariantWatchesEl-def}\)
  unfolding \(\text{swapWatches-def}\)
  unfolding \(\text{setWatch2ndef}\)
  by auto
  moreover
  from Cons(3)
  have \(\text{InvariantWatchesDiffer (getF state')}\) \(\text{(getWatch1 state')}\)
  (getWatch2 state')
  using \(\text{l' \neq ?w1}\)
  using \(\text{getWatch1 state'}\) clause = Some \(?w1\)
  using \(\text{getWatch2 state'}\) clause = Some \(?w2\)
  unfolding \(\text{InvariantWatchesDiffer-def}\)
  unfolding \(\text{swapWatches-def}\)
  unfolding \(\text{setWatch2-def}\)
  by auto
  moreover
  from Cons(4)
  have \(\text{InvariantConsistent (getM state')}\)
  unfolding \(\text{InvariantConsistent-def}\)
  unfolding \(\text{setWatch2-def}\)
  unfolding \(\text{swapWatches-def}\)
  by simp
  moreover
  from Cons(5)
  have \(\text{InvariantUniq (getM state')}\)
  unfolding \(\text{InvariantUniq-def}\)

388
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 el ((getF ?state'') ! c) ∧ literalTrue l (elements M) ∧ elementLevel l M ≤ elementLevel (opposite ww1) M) ∨
      (∀ 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 el (getF state ! clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 −→ literalFalse l (elements M))
      using Cons(8)
      using ⟨l' ≠ ?w1⟩ and ⟨l' ≠ ?w2⟩ and ⟨l el (nth (getF ?state'))⟩
      using ⟨¬ literalFalse l' (elements (getM ?state'))⟩
      using a and b
      using ⟨c = clause⟩
      unfolding swapWatches-def
      unfolding setWatch2-def

389
by auto

moreover

have \((\exists l. \, l \in (\text{getF state ! clause}) \land \text{literalTrue} \, l \, (\text{elements M}) \land \text{elementLevel} \, l \, M \leq \text{elementLevel} \, (\text{opposite} \, w1) \, M) \lor
(\forall l. \, l \in (\text{getF state ! clause}) \land l \neq w1 \land l \neq w2 \to \text{literalFalse} \, l \, (\text{elements M}))\)

using Cons(6)

unfolding InvariantWatchCharacterization-def

unfolding watchCharacterizationCondition-def

using \((0 \leq \text{clause} \land \text{clause} < \text{length} \, (\text{getF state}) \land \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 \((\text{literalFalse} \, w1 \, (\text{elements M}))\)

using \((w1 = w1)\)

unfolding setWatch2-def

unfolding swapWatches-def

by auto

ultimately

show \(?thesis\)

using \((w1 = w1)\)

using \((c = \text{clause})\)

unfolding setWatch2-def

unfolding swapWatches-def

by auto

qed

moreover

\{

fix \, c::\text{nat} \, \text{and} \, w1::\text{Literal} \, \text{and} \, w2::\text{Literal}

assume \, a: \, 0 \leq c \land c < \text{length} \, (\text{getF} \, \text{?state'}') \land \text{Some} \, w1 = (\text{getWatch1} \, \text{?state'}') \land \text{Some} \, w2 = (\text{getWatch2} \, \text{?state'}')

assume \, b: \text{literalFalse} \, w2 \, (\text{elements M})

have \((\exists l. \, l \in ((\text{getF} \, \text{?state''}) \, ! \, c) \land \text{literalTrue} \, l \, (\text{elements M}) \land \text{elementLevel} \, l \, M \leq \text{elementLevel} \, (\text{opposite} \, w2) \, M) \lor
(\forall l. \, l \in ((\text{getF} \, \text{?state''}) \, ! \, c) \land l \neq w1 \land l \neq w2 \to \text{literalFalse} \, l \, (\text{elements M}) \land \text{elementLevel} \, (\text{opposite} \, w2) \, M \leq \text{elementLevel} \, (\text{opposite} \, w2) \, M)\)

proof \,(\cases c = \text{clause})

case False

thus \, ?thesis

using \, a \, \text{and} \, b

using Cons(6)

unfolding InvariantWatchCharacterization-def

unfolding watchCharacterizationCondition-def

unfolding swapWatches-def

390
unfolding setWatch2-def
by auto
next
case True
with a
have \( w_1 = \bar{w_1} \) and \( w_2 = l' \)
  using \( \text{getWatch1 } \bar{\text{state}} \) clause = Some \( \bar{w_1} \)
  using \( \text{getWatch2 } \bar{\text{state}} \) clause = Some \( \bar{w_2} \) [THEN
sym]

unfolding setWatch2-def
unfolding swapWatches-def
by auto
with \( \neg \text{literalFalse } l' \) (elements (getM \( \bar{\text{state}} \))): 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 \( \forall \ (c :: \text{nat}). \ c \in \text{set } \text{Wl} \) \( \rightarrow \) Some literal = \( \text{getWatch1 } \bar{\text{state}}'' \) \( \bar{c} \) \( \lor \) Some literal = \( \text{getWatch2 } \bar{\text{state}}'' \) \( \bar{c} \)
  using Cons(10)
  using \( \text{clause} \notin \text{set } \text{Wl}' \)
  using swapWatchesEffect[of clause state]
  unfolding setWatch2-def
by simp
moreover
have getM \( \bar{\text{state}}'' \) = getM state
  getF \( \bar{\text{state}}'' \) = getF state
  unfolding swapWatches-def
  unfolding setWatch2-def
by auto
moreover
have getWatch1 \( \bar{\text{state}}'' \) clause = Some \( \bar{w_1} \) getWatch2 \( \bar{\text{state}}'' \)
  clause = Some \( l' \)
  using \( \text{getWatch1 } \bar{\text{state}}' \) clause = Some \( \bar{w_1} \)
  unfolding swapWatches-def
  unfolding setWatch2-def
  by auto
  hence getWatch1 \( \bar{\text{state}} \) clause = getWatch1 \( \bar{\text{state}}'' \) clause \( \land \)
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
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
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' el (nth (getF ?state') clause) l' ≠ ?w1 ∧ l' ≠ ?w2⟩ ¬ literalFalse l' (elements (getM ?state'))
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 \( \langle \text{getWatch1 ?state'} \text{ clause} = \text{Some } ?w1 \rangle \) [\( \text{THEN} \]

\[ \text{sym} \]

using \( \langle \text{getWatch2 ?state'} \text{ clause} = \text{Some } ?w2 \rangle \) [\( \text{THEN} \]

\[ \text{sym} \]

unfolding \( \text{InvariantWatchCharacterization-def} \)
unfolding \( \text{watchCharacterizationCondition-def} \)
unfolding \( \text{swapWatches-def} \)
by \( \text{simp} \)
with \( (\forall l. l \in (\text{nth (getF state') clause}) \land l \neq ?w1 \land l \neq ?w2 \implies \text{literalFalse l (elements M)}) \)
have \( \exists l. l \in (\text{getF state'} \text{ clause}) \land \text{literalTrue l (elements M)} \land \text{elementLevel l M} \leq \text{elementLevel (opposite ?w1) M} \)
by \( \text{auto} \)
thus \( \text{?thesis} \)
unfolding \( \text{setWatch2-def} \)
unfolding \( \text{swapWatches-def} \)
by \( \text{simp} \)
\( \text{qed} \)

\( \text{have watchCharacterizationCondition l'} ?w1 (\text{getM ?fState}) \)
\( \text{(getF ?fState') clause}) \)
using \( (\neg \text{literalFalse l' (elements (getM ?state'))}) \)
using \( (\text{getM ?fState} = \text{getM state}) \)
unfolding \( \text{swapWatches-def} \)
unfolding \( \text{watchCharacterizationCondition-def} \)
by \( \text{simp} \)
moreover
have \( \text{watchCharacterizationCondition ?w1 l' (getM ?fState)} \)
\( \text{(getF ?fState') clause}) \)
proof \( (\text{cases literalFalse ?w1 (elements (getM ?fState'))}) \)
case True
hence \( \text{literalFalse ?w1 (elements M)} \)
using \( \text{notifyWatchesLoopPreservedVariables[of ?state'' WI']} \)
literal newWl'
using \( (\text{InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'') (getF ?state'')} = \text{getF state}) \)
using \( \text{Cons(7) Cons(8)} \)
using \( (\neg ?w1 \neq ?w2) \land ?w2 = \text{literal} \)
unfolding \( \text{setWatch2-def} \)
unfolding \( \text{swapWatches-def} \)
by \( \text{(simp add: Let-def)} \)
with \( \text{literalFalse ?w1 (elements M)} \implies \)
\( (\exists l. l \in (\text{nth (getF ?state'')} \text{ clause}) \land \text{literalTrue l (elements M)} \land \text{elementLevel l M} \leq \text{elementLevel (opposite ?w1) M}) \)
obtain l::Literal
where \( l \in (\text{nth (getF ?state'')} \text{ clause}) \land \text{literalTrue l (elements M)} \land \text{elementLevel l M} \leq \text{elementLevel (opposite ?w1) M} \)

393
by auto

hence \text{elementLevel} \ l (\text{getM} \ \text{state}) \leq \text{elementLevel} (\text{opposite} \ ?w1) (\text{getM} \ \text{state})

using Cons(8)

using literalTrue \ l (\text{elements} \ M); \langle \text{literalFalse} \ ?w1 (\text{elements} \ M) \rangle

using elementLevelAppend[of \ l \ M \ [(\text{opposite} \ \text{literal}, \ \text{decision})]]

using elementLevelAppend[of \ opposite \ ?w1 \ M \ [(\text{opposite} \ \text{literal}, \ \text{decision})]]

by auto

thus ?thesis

using \langle \ l \ \epsilon \ (\text{nth} (\text{getF} \ ?\text{state}’’) \ \text{clause}) \rangle; \langle \text{literalTrue} \ l \ \text{(elements} \ M) \rangle

using (\text{getM} \ ?\text{fState} = \text{getM} \ \text{state}); (\text{getF} \ ?\text{fState} = \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 ?thesis

unfolding \text{watchCharacterizationCondition-def}

by simp

qed

ultimately

show ?thesis

by simp

qed

ultimately

show ?thesis

using Cons(1)[of \ ?\text{state}’’ \ 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} \ \text{literal} = \text{getWatch1} \ ?\text{state} \ \text{clause})

using (~ 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 (uniq \ Wl’)

by (simp add: Let-def)

next

case None

show ?thesis

proof (cases literalFalse \ ?w1 (\text{elements} \ (\text{getM} \ ?\text{state}’))]

case True

let \ ?\text{state}’’ = \ ?\text{state}’’\{(\text{getConflictFlag} := \text{True}, \ \text{getConflict-Clause} := \text{clause})

394
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 \( \forall (c::nat). c \in \text{set Wl} \rightarrow \text{Some literal} = (\text{getWatch1} ?state'' c) \lor \text{Some literal} = (\text{getWatch2} ?state'' c) \)
  using Cons(10)
  using (\text{clause} \notin \text{set Wl}'
  using swapWatchesEffect[of clause state]
  by simp
moreover
have getM ?state'' = getM state
  getF ?state'' = getF state
  unfolding swapWatches-def
  by auto
moreover
have \( \text{getWatch1 ?fState clause} = \text{getWatch1 ?state'' clause} \land \text{getWatch2 ?fState clause} = \text{getWatch2 ?state'' clause} \)

using \( \langle \text{clause \notin set Wl} \rangle \)

using \( \langle \text{InvariantWatchesEl (getF ?state'')} (\text{getWatch1 ?state'')} (\text{getWatch2 ?state'')} (\text{getF ?state'' = getF state}) \rangle \)

using Cons(7)

using notifyWatchesLoopPreservedWatches[of ?state'' Wl'

literal clause \# newWl ]

by \( \langle \text{simp add: Let-def} \rangle \)

moreover

have literalFalse ?w1 \langle \text{elements M} \rangle

using \( \langle \text{literalFalse ?w1 \langle \text{elements (getM ?state')} \rangle \rangle \)

\( \langle \text{?w1 \neq ?w2} \rangle \langle \text{?w2 = literal} \rangle \text{Cons(8)} \)

unfolding swapWatches-def

by auto

have \( \neg \text{literalTrue ?w2 \langle \text{elements M} \rangle} \)

using Cons(4)

using Cons(8)

using \( \langle \text{?w2 = literal} \rangle \)

using inconsistentCharacterization[of elements M \& [opposite literal]]

unfolding InvariantConsistent-def

by force

have \( *: \forall l. l el (\text{nth (getF state) clause}) \land l \neq ?w1 \land l \neq ?w2 \rightarrow \)

\( \text{literalFalse l \langle \text{elements M} \rangle \land elementLevel (opposite l) M \leq elementLevel (opposite ?w1) M} \)

proof

have \( \neg (\exists l. l el (\text{nth (getF state) clause}) \land \text{literalTrue l \langle \text{elements M} \rangle}) \)

proof

assume \( \exists l. l el (\text{nth (getF state) clause}) \land \text{literalTrue l \langle \text{elements M} \rangle} \)

show False

proof

from \( \exists l. l el (\text{nth (getF state) clause}) \land \text{literalTrue l \langle \text{elements M} \rangle} \)

obtain l

where \( l el (\text{nth (getF state) clause}) \text{literalTrue l \langle \text{elements M} \rangle} \)

by auto

hence \( l \neq ?w1 l \neq ?w2 \)

using \( \langle \neg \text{literalTrue ?w1 \langle \text{elements (getM ?state')} \rangle} \rangle \)

using \( \langle \neg \text{literalTrue ?w2 \langle \text{elements M} \rangle} \rangle \)

unfolding swapWatches-def

using Cons(8)

by auto
with \( l \in \text{elts} \left( \text{getF} \ ?\text{state}' \right) \)

have \( \text{literalFalse} \ l \left( \begin{array}{c}
\text{elements} \\
\text{getM} \ ?\text{state}'
\end{array} \right) \)

using \( \text{getWatch1} \ ?\text{state}' \left( \begin{array}{c}
\text{clause} \\
= \text{Some} \ ?w1
\end{array} \right) \)

using \( \text{getWatch2} \ ?\text{state}' \left( \begin{array}{c}
\text{clause} \\
= \text{Some} \ ?w2
\end{array} \right) \)

using None

using \( \text{getNonWatchedUnfalsifiedLiteralNoneCharacter-
iz}\[\text{[of nth} \ (\text{getF} \ ?\text{state}') \ \text{clause} \ ?w1 \ ?w2 \ \text{getM} \ ?\text{state}'] \]

unfolding swapWatches-def

by simp

with \( l \neq ?w2 \Rightarrow \text{Cons}(8) \)

have \( \text{literalFalse} \ l \left( \begin{array}{c}
\text{elements} \ M
\end{array} \right) \)

unfolding swapWatches-def

by simp

with \( \text{Cons}(4) \Rightarrow \text{literalTrue} \ l \left( \begin{array}{c}
\text{elements} \ M
\end{array} \right) \)

show \( \text{thesis} \)

unfolding InvariantConsistent-def

using \( \text{Cons}(8) \)

by (auto simp add: inconsistentCharacterization)

qed

with \( \text{InvariantWatchCharacterization} \ (\text{getF} \ ?\text{state}) \ (\text{getWatch1} \ ?\text{state}) \ (\text{getWatch2} \ ?\text{state}) \ (\text{getM} ?\text{state}) \)

show \( \text{thesis} \)

unfolding InvariantWatchCharacterization-def

using \( \text{literalFalse} \ ?w1 \left( \begin{array}{c}
\text{elements} \ M
\end{array} \right) \)

using \( \text{getWatch1} \ ?\text{state}' \left( \begin{array}{c}
\text{clause} \\
= \text{Some} \ ?w1
\end{array} \right) \left( \begin{array}{c}
\text{THEN sym}
\end{array} \right) \)

using \( \text{getWatch2} \ ?\text{state}' \left( \begin{array}{c}
\text{clause} \\
= \text{Some} \ ?w2
\end{array} \right) \left( \begin{array}{c}
\text{THEN sym}
\end{array} \right) \)

using \( 0 \leq \text{clause} \land \text{clause} < \text{length} \ (\text{getF} \ ?\text{state}) \)

unfolding watchCharacterizationCondition-def

unfolding swapWatches-def

by (simp) (blast)

qed

have \( **: \forall \ l. \ l \in \text{elts} \left( \text{getF} \ ?\text{state}' \right) \land l \neq ?w1 \land l \neq ?w2 \Rightarrow \text{literalFalse} \ l \left( \begin{array}{c}
\text{elements} \ (\text{getM} ?\text{state}'')
\end{array} \right) \land \\
\text{elementLevel} \ (\text{opposite} \ l) \ (\text{getM} ?\text{state}'') \leq \text{elementLevel} \ (\text{opposite} \ ?w1) \ (\text{getM} ?\text{state}'') \)

proof –

\{ 

fix \( l::\text{Literal} \)

assume \( l \in \text{elts} \left( \text{getF} \ ?\text{state}' \right) \land l \neq ?w1 \land l \neq ?w2 \)

have \( \text{literalFalse} \ l \left( \begin{array}{c}
\text{elements} \ (\text{getM} ?\text{state}'')
\end{array} \right) \land \\
\text{elementLevel} \ (\text{opposite} \ l) \ (\text{getM} ?\text{state}'') \leq \text{elementLevel} \ (\text{opposite} \ ?w1) \ (\text{getM} ?\text{state}'') \)

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
qed

have (getM ?fState) = (getM state) (getF ?fState) = (getF state)
 using notifyWatchesLoopPreservedVariables[of ?state" WI'
 literal clause # newWl]
 using ! InvariantWatchesEl (getF ?state") (getWatch1 ?state") (getWatch2 ?state") (getF ?state" = getF state)
 using Cons(7)
 unfolding swapWatches-def
 by (auto simp add: Let-def)
hence ∀ l. l el (nth (getF ?fState) clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2

literalseq 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 = getWatch1 state clause)
using (¬ literalTrue ?w1 (elements (getM ?state’')))
using None
using (literalFalse ?w1 (elements (getM ?state’’))
using (uniq Wl’’)
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 Wl’ (clause # newWl)

?state’’

from Cons(2)

have InvariantWatchesEl (getF ?state’’) (getWatch1 ?state’’)

from Cons(3)

have InvariantWatchesDiffer (getF ?state’’) (getWatch1 ?state’’) (getWatch2 ?state’’)

from Cons(4)

have InvariantConsistent (getM ?state’’)

399
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 set Wl' \rightarrow Some literal = (getWatch1 ?state"") c \lor Some literal = (getWatch2 ?state"") c
  using Cons(10)
  using \langle clause / \in 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 getWatch1 ?state' clause = Some ?w1 \rangle
  using \langle getWatch2 ?state' clause = Some ?w2 \rangle
  unfolding swapWatches-def
  by auto
moreover
have getWatch1 ?fState clause = Some ?w1 getWatch2 ?fState clause = Some ?w2
  using \langle getWatch1 ?state"" clause = Some ?w1 \rangle \langle getWatch2 ?state"" clause = Some ?w2 \rangle
  using \langle clause / \in set Wl' \rangle
  using InvariantWatchesEl (getF ?state") (getWatch1 ?state") (getWatch2 ?state") (getF ?state"" = getF state)
  using Cons(7)
  using notifyWatchesLoopPreservedWatches[of ?state"" Wl'

400
let-def

by (auto simp add: Let-def)

moreover

have (getM ?fState) = (getM state) (getF ?fState) = (getF state)

using notifyWatchesLoopPreservedVariables[of ?state"] Wl'

using (InvariantWatchesEl (getF ?state') (getWatch1 ?state') (getWatch2 ?state') (getF ?state" = getF state)

using Cons(7)

unfolding setReason-def

by (auto simp add: Let-def)

ultimately

have ∀ c. c ∈ set Wl' "" (λ w1 w2. Some w1 = getWatch1 ?fState c ∧ Some w2 = getWatch2 ?fState c ""

  watchCharacterizationCondition w1 w2 (getM ?fState) (getF ?fState ! c) ∧

  watchCharacterizationCondition w2 w1 (getM ?fState)

and

?fState = notifyWatches-loop literal (clause # Wl') 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 (∀ literal = getWatch1 state clause)

using None

using (¬ literalTrue ?w1 (elements (getM ?state')))

using None

using (¬ literalFalse ?w1 (elements (getM ?state')))

using (uniq Wl'

by (auto simp add: Let-def)

moreover

have : ∀ l. l el (nth (getF ?state") clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 → 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**: ∀ l. l el (nth (getF ?fState) clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 → 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 ***: \forall l. literalFalse l (elements (getM ?fState)) \rightarrow
 elementLevel (opposite l) (getM ?fState) \leq 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

have (\forall w1 w2. Some w1 = getWatch1 ?fState clause \land Some w2 = getWatch2 ?fState clause \rightarrow
 watchCharacterizationCondition w1 w2 (getM ?fState) (getF ?fState ! clause) \land
 watchCharacterizationCondition w2 w1 (getM ?fState) (getF ?fState ! clause))
proof -
  { fix w1 w2
    assume Some w1 = getWatch1 ?fState clause \land 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) \land
      watchCharacterizationCondition w2 w1 (getM ?fState) (getF ?fState ! clause)
      unfolding watchCharacterizationCondition-def
      using ** ***
      unfolding watchCharacterizationCondition-def
      using (~ literalFalse ?w1 (elements (getM ?state')));
      unfolding swapWatches-def
      by simp
  }
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 \((\neg Some literal = getWatch1 state clause)\)
\(\forall (c::nat). c \in set (clause \neq Wl) \rightarrow Some literal = (getWatch1 state clause c)\)
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 \(?w2\)\)\)
using \((0 \leq clause \land clause < length (getF state))\)
unfolding InvariantWatchesEl-def
by auto

from \((InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state))\)
have \(?w1 \neq \(?w2\)\)
using \((getWatch1 \(?state' clause = Some \(?w1\)\)\)
using \((getWatch2 \(?state' clause = Some \(?w2\)\)\)
using \(0 \leq \text{clause} \land \text{clause} < \text{length} \text{(getF state)}\)

unfolding InvariantWatchesDiffer-def

by auto

have \(\neg \text{literalFalse} \; ?w2 \; (\text{elements} \; M)\)

using \(\langle ?w2 = \text{literal} \rangle\)

using Cons(5)

using Cons(8)

unfolding InvariantUniq-def

by (simp add: uniqAppendIff)

show \(?\text{thesis}\)

proof (cases literalTrue \(?w1 \; (\text{elements} \; \text{(getM} \; \text{?state}'))\))

case True

let \(?fState = \text{notifyWatches-loop} \; \text{literal} \; \text{Wl'} \; \text{(clause} \# \text{ newWl)}\)

?state'

have getWatch1 \(?fState \; \text{clause} = \text{getWatch1} \; \text{?state'} \; \text{clause} \land \)

\text{getWatch2} \; \text{?fState} \; \text{clause} = \text{getWatch2} \; \text{?state'} \; \text{clause}

using \(\langle \text{clause} \notin \text{set} \; \text{Wl'} \rangle\)

using Cons(2)

using Cons(7)

using notifyWatchesLoopPreservedWatches[of \text{?state'} \; \text{Wl'} \; \text{literal clause} \# \text{ newWl}]

by (simp add: Let-def)

moreover

have watchCharacterizationCondition \(?w1 \; \text{?w2} \; (\text{getM} \; \text{?fState})\)

\text{(getF} \; \text{?fState} \; \text{! clause}) \land

\text{watchCharacterizationCondition} \; ?w2 \; ?w1 \; (\text{getM} \; \text{?fState})

\text{(getF} \; \text{?fState} \; \text{! clause})

proof–

have \((\text{getM} \; \text{?fState}) = (\text{getM} \; \text{state}) \land (\text{getF} \; \text{?fState} = \text{getF} \; \text{state}))\)

using notifyWatchesLoopPreservedVariables[of \text{?state'} \; \text{Wl'} \; \text{literal clause} \# \text{ newWl}]

using Cons(2)

using Cons(7)

by (simp add: Let-def)

moreover

have \(\neg \text{literalFalse} \; ?w1 \; (\text{elements} \; M)\)

using 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 elementLevel \((\text{opposite} \; ?w2) \; (\text{getM} \; \text{?state}')) = \text{currentLevel} \; (\text{getM} \; \text{?state}')\)

404
using ⟨?w2 = literal⟩
using Cons(5) Cons(8)
unfolding InvariantUniq-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⟩
unfolding watchCharacterizationCondition-def
using elementLevelLeqCurrentLevel[of ?w1 getM ?state']
using notifyWatchesLoopPreservedVariables[of ?state' Wl' literal clause # newWl]
using ⟨InvariantWatchesEl (getF state) (getWatch1 state)
(getWatch2 state)⟩
using Cons(7)
using 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 \langle l' el (nth (getF ?state') clause) \rangle
  unfolding InvariantWatchesEl-def
  unfolding setWatch2-def
  by auto
moreover
from Cons(3)
  have InvariantWatchesDiffer (getF ?state') (getWatch1 ?state'') (getWatch2 ?state'')
    using \langle l' \not= ?w1 \rangle
    using \langle getWatch1 ?state' clause = Some ?w1 \rangle
    using \langle getWatch2 ?state' clause = 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 ww1::Literal and ww2::Literal
      assume a: 0 \leq c \land c < length (getF ?state'') \land Some ww1 = (getWatch1 ?state'' c) \land Some ww2 = (getWatch2 ?state'' c)
      assume b: literalFalse ww1 (elements M)
      have (\exists l. l el ((getF ?state'') \l c) \land literalTrue l (elements M) \land elementLevel l M \leq elementLevel (opposite ww1) M) \lor
        (\forall l. l el ((getF ?state'') \l c) \land l \not= ww1 \land l \not= ww2 --> literalFalse l (elements M) \land elementLevel (opposite l) M \leq 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 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⟩

unfolding setWatch2-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) a (nth (getF ?state'))

unfolding setWatch2-def
by auto

moreover
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))
  using Cons(6)

unfolding InvariantWatchCharacterization-def

unfolding watchCharacterizationCondition-def

unfolding setWatch2-def
by auto

ultimately
show ?thesis
  using ⟨ww1 = ?w1⟩
  using ⟨c = clause⟩
  using ⟨getWatch2 ?state' clause = Some ?w2⟩
  using ⟨getWatch1 ?state' clause = Some ?w1⟩

by auto
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 W1⟩
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 W1⟩
  using InvariantWatchesEl (getF ?state") (getWatch1 ?state")
  (getWatch2 ?state") : getF ?state" = getF state
  using Cons(7)
  using notifyWatchesLoopPreservedWatches[of ?state" W1'
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" W1'
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
\( \neq \ ?w2 \rightarrow \text{literalFalse} \ l \ (\text{elements} \ M))

using \( l' \ \epsilon \ (\text{nith} \ (\text{getF} \ ?\text{state}')) \ \text{clause}) \ \langle l' \neq \ ?w1 \ \rangle \langle l' \neq \ ?w2 \rangle \ \neg \ \text{literalFalse} \ l' \ (\text{elements} \ (\text{getM} \ ?\text{state}'));

using \text{Cons}(8)

unfolding \ \text{swapWatches-def}

by auto

from \langle \text{literalFalse} \ ?w1 \ (\text{elements} \ M) \rangle \ \text{Cons}(6)

have
\( \exists \ l. \ l \ \epsilon \ (\text{getF} \ ?\text{state} ! \ \text{clause}) \ \wedge \ \text{literalTrue} \ l \ (\text{elements} \ M) \ \wedge \ \text{elementLevel} \ l \ M \leq \ \text{elementLevel} \ (\text{opposite} \ ?w1) \ M \ \vee \ \forall \ l. \ l \ \epsilon \ (\text{getF} \ ?\text{state} ! \ \text{clause}) \ \wedge \ l \neq \ ?w1 \ \wedge \ l \neq \ ?w2 \ \rightarrow \ \text{literalFalse} \ l \ (\text{elements} \ M) \ \wedge \ \text{elementLevel} \ (\text{opposite} \ l) \ M \leq \ \text{elementLevel} \ (\text{opposite} \ ?w1) \ M
\)

using \( 0 \leq \ \text{clause} \ \wedge \ \text{clause} < \ \text{length} \ (\text{getF} \ ?\text{state})

\using \text{getWatch1} \ ?\text{state}' \ \text{clause} = \ \text{Some} \ ?w1 \ [\text{THEN \ sym}]\text{getWatch2} \ ?\text{state}' \ \text{clause} = \ \text{Some} \ ?w2 \ [\text{THEN \ sym}]

\text{unfolding} \ \text{InvariantWatchCharacterization-def}

\text{unfolding} \ \text{watchCharacterizationCondition-def}

by simp

with \( \neg \ \forall \ l. \ l \ \epsilon \ (\text{nith} \ (\text{getF} \ ?\text{state}) \ \text{clause}) \ \wedge \ l \neq \ ?w1 \ \wedge \ l \neq \ ?w2 \ \rightarrow \ \text{literalFalse} \ l \ (\text{elements} \ M));

have \( \exists \ l. \ l \ \epsilon \ (\text{getF} \ ?\text{state} ! \ \text{clause}) \ \wedge \ \text{literalTrue} \ l \ (\text{elements} \ M) \ \wedge \ \text{elementLevel} \ l \ M \leq \ \text{elementLevel} \ (\text{opposite} \ ?w1) \ M
\)

by auto

thus ?thesis

unfolding \ \text{setWatch2-def}

by simp

qed

qed

moreover

have \( \text{watchCharacterizationCondition} \ ?w1 \ ?w1' \ (\text{getM} \ ?\text{fState})\)

\( (\text{getF} \ ?\text{fState} ! \ \text{clause})

using \( \neg \ \text{literalFalse} \ ?w1' \ (\text{elements} \ (\text{getM} \ ?\text{state}'))

using \( \text{getM} \ ?\text{fState} = \ \text{getM} \ \text{state})

unfolding \ \text{watchCharacterizationCondition-def}

by simp

moreover

have \( \text{watchCharacterizationCondition} \ ?w1 \ ?w1' \ (\text{getM} \ ?\text{fState})\)

\( (\text{getF} \ ?\text{fState} ! \ \text{clause})

proof \ (\text{cases} \ \text{literalFalse} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{fState})),)

case \ True

hence \ \text{literalFalse} \ ?w1 \ (\text{elements} \ M)

using \ \text{notifyWatchesLoopPreservedVariables}[\text{of} \ ?\text{state}'' \ \text{fState}'' \ \text{fState}''']

\text{using} \ \text{InvariantWatchesEl} \ (\text{getF} \ ?\text{state}''') \ (\text{getWatch1} \ ?\text{state}''') \ (\text{getWatch2} \ ?\text{state}''') \ (\text{getF} \ ?\text{state}'''' = \ \text{getF} \ ?\text{state})

410
using Cons(7) Cons(8)
using (?w1 ≠ ?w2) (?w2 = literal)
unfolding setWatch2-def
by (simp add: Let-def)
with literalFalse ?w1 (elements M) →
(∃ l. l el (n (getF ?state') clause) ∧ literalTrue l (elements M) ∧
elementLevel l M ≤ elementLevel (opposite ?w1) M)
obtain l::Literal
  where l el (n (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 literalTrue l (elements M) (literalFalse ?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 (n (getF ?state') clause) (literalTrue l (elements M))
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" newW1]
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 1⟩
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'') (getM ?state''
      unfolding InvariantWatchesEl-def
      by auto
    moreover
    from Cons(3)
    have InvariantWatchesDiffer (getF ?state") (getWatch1 ?state'') (getWatch2 ?state'') (getM ?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 WI')
      by simp
moreover
have \textit{getM} ?\textit{state}'' = \textit{getM state} \\
\textit{getF} ?\textit{state}'' = \textit{getF state} \\
by auto \\
moreover
have \textit{getWatch1} ?\textit{fState clause} = \textit{getWatch1} ?\textit{state}'' clause \wedge \\
\textit{getWatch2} ?\textit{fState clause} = \textit{getWatch2} ?\textit{state}'' clause \\
using \langle \textit{clause} \notin \textit{set WI}' \rangle \\
using \langle \textit{InvariantWatchesEl (getF ?state')} (\textit{getWatch1} ?\textit{state}'') (\textit{getWatch2} ?\textit{state}''): (\textit{getF} ?\textit{state}'') = \textit{getF state} \rangle \\
using \langle \textit{Cons}(7) \rangle \\
using \langle \textit{notifyWatchesLoopPreservedWatches[of ?state'' WI} \\
literal clause \# newWI \rangle \\
by \langle \textit{simp add: Let-def} \rangle \\
moreover \\
have \textit{literalFalse} ?\textit{w1} (\textit{elements M}) \\
using \langle \textit{literalFalse} ?\textit{w1} (\textit{elements (getM ?state')}) \rangle \\
\langle ?\textit{w1} \neq ?\textit{w2}, ?\textit{w2} = \textit{literal} Cons(8) \rangle \\
by auto \\
have \neg \textit{literalTrue} ?\textit{w2} (\textit{elements M}) \\
using \langle \textit{Cons}(4) \rangle \\
using \langle \textit{Cons}(8) \rangle \\
using \langle ?\textit{w2} = \textit{literal} \rangle \\
using \langle \textit{inconsistentCharacterization[of elements M @ [opposite literal]} \rangle \\
unfolding \langle \textit{InvariantConsistent-def} \rangle \\
by \langle \textit{force} \rangle \\
have \star: \forall l. \ l el (\textit{nth (getF state) clause}) \wedge l \neq ?\textit{w1} \wedge l \neq ?\textit{w2} \rightarrow \\
\textit{literalFalse} l (\textit{elements M}) \wedge \textit{elementLevel (opposite l) M} \leq \\
\textit{elementLevel (opposite ?w1) M} \\
proof \\
have \neg (\exists l. \ l el (\textit{nth (getF state) clause}) \wedge \textit{literalTrue} l \\
(\textit{elements M})) \\
proof \\
assume \exists l. \ l el (\textit{nth (getF state) clause}) \wedge \textit{literalTrue} l \\
(\textit{elements M}) \\
show False \\
proof - \\
from \langle \exists l. \ l el (\textit{nth (getF state) clause}) \wedge \textit{literalTrue} l \\
(\textit{elements M}) \rangle \\
obtain l \\
where l el (\textit{nth (getF state) clause}) \textit{literalTrue} l \\
(\textit{elements M}) \\
by auto \\
hence l \neq ?w1 l \neq ?w2 \\
using \langle \neg \textit{literalTrue} ?w1 (\textit{elements (getM ?state')}) \rangle;
using \( \neg \text{literalTrue} \ ?w2 \ (\text{elements} \ M) \)
using \( \text{Cons}(8) \)
by auto
with \( \cdot \ el \ (\text{nth} \ (\text{getF state}) \ \text{clause}) \)
have \( \text{literalFalse} \ \cdot \ (\text{elements} \ (\text{getM} \ ?\text{state}')) \)
using \( \text{getWatch1} \ ?\text{state}' \ \text{clause} = \text{Some} \ ?w1 \)
using \( \text{getWatch2} \ ?\text{state}' \ \text{clause} = \text{Some} \ ?w2 \)
using \( \text{None} \)
using \( \text{getNonWatchedUnfalsifiedLiteralNoneCharacterization}[\text{of} \ \text{nth} \ (\text{getF} \ ?\text{state}') \ \text{clause} \ ?w1 \ ?w2 \ \text{getM} \ ?\text{state}'] \)
by \( \text{simp} \)
with \( \cdot \ neq \ ?w2 ; ?w2 = \text{literal} \ \text{Cons}(8) \)
have \( \text{literalFalse} \ \cdot \ (\text{elements} \ M) \)
by \( \text{simp} \)
with \( \text{Cons}(4) \) \( \langle \text{literalTrue} \ \cdot \ (\text{elements} \ M) \rangle \)
show \( \langle \text{thesis} \rangle \)
unfolding \( \text{InvariantConsistent-def} \)
using \( \text{Cons}(8) \)
by \( (\text{auto simp add: inconsistentCharacterization}) \)
qed
qed
with \( \langle \text{InvariantWatchCharacterization} \ (\text{getF state}) \ (\text{getWatch1} \ \text{state}) \ (\text{getWatch2} \ \text{state}) \ M \rangle \)
show \( \langle \text{thesis} \rangle \)
unfolding \( \text{InvariantWatchCharacterization-def} \)
using \( \langle \text{literalFalse} \ ?w1 \ (\text{elements} \ M) \rangle \)
using \( \langle \text{getWatch1} \ ?\text{state}' \ \text{clause} = \text{Some} \ ?w1 \rangle \ [\text{THEN sym}] \)
using \( \langle \text{getWatch2} \ ?\text{state}' \ \text{clause} = \text{Some} \ ?w2 \rangle \ [\text{THEN sym}] \)
using \( \langle 0 \leq \text{clause} \land \text{clause} < \text{length} \ (\text{getF state}) \rangle \)
unfolding \( \text{watchCharacterizationCondition-def} \)
by \( (\text{simp}) \ (\text{blast}) \)
qed

have \( \forall \ \cdot \ \cdot \ (\text{nth} \ (\text{getF} \ ?\text{state}'') \ \text{clause}) \ \land \ \cdot \ neq \ ?w1 \ \land \ \cdot \ neq \ ?w2 \ \rightarrow \)
\( \text{literalFalse} \ \cdot \ (\text{elements} \ (\text{getM} \ ?\text{state}'')) \ \land \)
\( \text{elementLevel} \ (\text{opposite} \ \cdot) \ (\text{getM} \ ?\text{state}'') \leq \text{elementLevel} \ (\text{opposite} ?w1) \ (\text{getM} \ ?\text{state}'') \)
proof -
{
fix \cdot :: \text{Literal} 
assume \( \cdot \ el \ (\text{nth} \ (\text{getF} \ ?\text{state}'') \ \text{clause}) \ \land \ \cdot \ neq \ ?w1 \ \land \ \cdot \ neq \ ?w2 \)

have \( \text{literalFalse} \ \cdot \ (\text{elements} \ (\text{getM} ?\text{state}'')) \ \land \)
\( \text{elementLevel} \ (\text{opposite} \ \cdot) \ (\text{getM} ?\text{state}'') \leq \text{elementLevel} \ (\text{opposite} ?w1) \ (\text{getM} ?\text{state}'') \)

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

416
unfolding setReason-def
by simp
moreover
from Cons(6)
have InvariantWatchCharacterization (getF ?state") (getWatch1 ?state"
M
unfolding setReason-def
unfolding InvariantWatchCharacterization-def
unfolding watchCharacterizationCondition-def
by simp
moreover
have \(\forall (c::nat)\). c \in set Wl' \rightarrow Some literal = (getWatch1 ?state" c) \lor Some literal = (getWatch2 ?state" c)
using Cons(10)
using (clause \notin set Wl')
unfolding setReason-def
by simp
moreover
have getM ?state" = getM state
getF ?state" = getF state
unfolding setReason-def
by auto
moreover
have getWatch1 ?state" clause = Some ?w1 getWatch2 ?state" clause = Some ?w2
using (getWatch1 ?state' clause = Some ?w1)
using (getWatch2 ?state' clause = Some ?w2)
unfolding setReason-def
by auto
moreover
have getWatch1 ?fState clause = Some ?w1 getWatch2 ?fState clause = Some ?w2
using (getWatch1 ?state" clause = Some ?w1) (getWatch2 ?state" clause = Some ?w2)
using (clause \notin 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 (auto simp add: Let-def)
moreover
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)

417
unfolding setReason-def
by (auto simp add: Let-def)
ultimately
have ∀ c. c ∈ set Wl' → (∀ w1 w2. Some w1 = getWatch1 ?fState c ∧ Some w2 = getWatch2 ?fState c → watchCharacterizationCondition w1 w2 (getM ?fState) (getF ?fState ! c) ∧ watchCharacterizationCondition w2 w1 (getM ?fState) (getF ?fState ! c)) and
?fState = notifyWatches-loop literal (clause # Wl') 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 (∼ literalTrue ?w1 (elements (getM ?state')))
using None
using (∼ literalFalse ?w1 (elements (getM ?state')))
using (uniq Wl')
by (auto simp add: Let-def)
moreover
have ∗: ∀ l. l el (nth (getF ?fState) clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 → 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 ?state']
using Cons(8)
unfolding setReason-def
by auto

have +++: ∀ l. l el (nth (getF ?fState) clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 → literalFalse l (elements (getM ?fState))
using *
using (getM ?state'' = getM state)
using (getF ?state'' = getF state)
by auto

have +++: ∀ l. literalFalse l (elements (getM ?fState)) → elementLevel (opposite ?w2) (getM ?fState) ≤ elementLevel (opposite ?w2) (getM ?fState)
proof-
have elementLevel (opposite ?w2) (getM ?fState) = currentLevel (getM ?fState)
using Cons(8)
using \((\text{getM} ?\text{fState}) = (\text{getM} \text{ state})\) 
using \((\neg \text{literalFalse} ?w2 \ (\text{elements} M))\) 
using \((?w2 = \text{literal})\) 
using \(\text{elementOnCurrentLevel\{of opposite} \ ?w2 \ M \text{ decision}\)\] by simp 
thus \(?\text{thesis}\) by (simp add: \(\text{elementLevelLeqCurrentLevel}\))

qed

have \((\forall \ w1 \ w2. \ Some \ w1 = \text{getWatch1} \ ?\text{fState} \ \text{clause} \land \ Some \ w2 = \text{getWatch2} \ ?\text{fState} \ \text{clause} \rightarrow \) \(\text{watchCharacterizationCondition} \ w1 \ w2 \ (\text{getM} ?\text{fState}) \ (\text{getF} ?\text{fState} ! \ \text{clause}) \land \) 
\(\text{watchCharacterizationCondition} \ w2 \ w1 \ (\text{getM} ?\text{fState}) \ (\text{getF} ?\text{fState} ! \ \text{clause})\))
proof−
{ 
fix \(\ w1 \ w2\)
assume Some \(\ w1 = \text{getWatch1} \ ?\text{fState} \ \text{clause} \land \ Some \ w2 = \text{getWatch2} \ ?\text{fState} \ \text{clause} \)

hence \(\ w1 = ?w1 \ w2 = ?w2\)
using \(\text{getWatch1} ?\text{fState} \ \text{clause} = \text{Some} \ ?w1\)
using \(\text{getWatch2} ?\text{fState} \ \text{clause} = \text{Some} \ ?w2\)
by auto

hence \(\text{watchCharacterizationCondition} \ w1 \ w2 \ (\text{getM} ?\text{fState}) \ (\text{getF} ?\text{fState} ! \ \text{clause}) \land \)
\(\text{watchCharacterizationCondition} \ w2 \ w1 \ (\text{getM} ?\text{fState})\)
\((\text{getF} ?\text{fState} ! \ \text{clause})\)
unfolding \(\text{watchCharacterizationCondition-def}\)
using ** ***
unfolding \(\text{watchCharacterizationCondition-def}\)
using \(\text{(getM} ?\text{fState}) = (\text{getM} \text{ state})\) \(\text{(getF} \ ?\text{fState}) = \)
\((\text{getF} \text{ state})\)
using \(\neg \text{literalFalse} ?w1 \ (\text{elements} (\text{getM} \ ?\text{state}'))\)
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 :: 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) and
InvariantConsistent (getM state)
∀ (c::nat). c ∈ set Wl → Some literal = (getWatch1 state c) ∨
Some literal = (getWatch2 state c)
literalFalse literal (elements (getM state))
uniq Wl
shows
let state' = notifyWatches-loop literal Wl newWl state in
getConflictFlag state' =
(getConflictFlag state ∨
(∃ clause. clause ∈ set Wl ∧ clauseFalse (nth (getF state)
clause) (elements (getM 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
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 \( w_2 = w_a \)

have \( \text{getWatch2} \ ?\text{state}' \ clause = \text{Some} \ ?w_2 \)
using \( \text{getWatch1} \ ?\text{state} \ clause = \text{Some} \ w_a \)
unfolding \( \text{swapWatches-def} \)
by \( \text{auto} \)

from \( \langle \text{Some literal} = \text{getWatch1} \ ?\text{state} \ clause \rangle \)
\( \langle \text{getWatch2} \ ?\text{state}' \ clause = \text{Some} \ ?w_2 \rangle \)
\( \langle \text{literalFalse} \ ?\text{literal} (\text{elements} (\text{getM} \ ?\text{state})) \rangle \)
have \( \text{literalFalse} \ ?w_2 \ (\text{elements} (\text{getM} \ ?\text{state})) \)
unfolding \( \text{swapWatches-def} \)
by \( \text{simp} \)

from \( \langle \text{InvariantWatchesEl} (\text{getF} \ ?\text{state}) (\text{getWatch1} \ ?\text{state}) (\text{getWatch2} \ ?\text{state}) \rangle \)
have \( \ ?w_1 \ ?\text{el} \ (\text{nth} (\text{getF} \ ?\text{state}) \ ?\text{clause}) \)
using \( \langle \text{getWatch1} \ ?\text{state}' \ clause = \text{Some} \ ?w_1 \rangle \)
using \( \langle \text{getWatch2} \ ?\text{state}' \ clause = \text{Some} \ ?w_2 \rangle \)
using \( \langle \text{clause} < \text{length} (\text{getF} \ ?\text{state}) \rangle \)
unfolding \( \text{InvariantWatchesEl-def} \)
unfolding \( \text{swapWatches-def} \)
by \( \text{auto} \)

moreover
have \( \text{getF} \ ?\text{state}' = \text{getF} \ ?\text{state} \land \)
\( \text{getM} \ ?\text{state}' = \text{getM} \ ?\text{state} \land \)
\( \text{getConflictFlag} \ ?\text{state}' = \text{getConflictFlag} \ ?\text{state} \)
unfolding \( \text{swapWatches-def} \)
by \( \text{simp} \)

moreover
have \( \forall \ c. \ c \in \text{set} \ Wl' \rightarrow \text{Some} \ ?\text{literal} = \text{getWatch1} \ ?\text{state}' \ ?\text{c} \land \)
\( \text{Some} \ ?\text{literal} = \text{getWatch2} \ ?\text{state}' \ ?\text{c} \)
using \( \text{Cons(5)} \)
unfolding \( \text{swapWatches-def} \)
by \( \text{auto} \)

moreover
have \( \neg \ \text{clauseFalse} (\text{nth} (\text{getF} \ ?\text{state}) \ ?\text{clause}) (\text{elements} (\text{getM} \ ?\text{state})) \)

421
using \(<w_1\) el \(<\text{nth} (getF \ ?state')\) \\
using \(<\text{literalTrue} \ ?w_1\) \(<\text{elements} (getM \ ?state')\) \\
using \(<\text{InvariantConsistent} \ (getM \ state)\) \\
unfolding \(<\text{InvariantConsistent-def}\) \\
unfolding \(<\text{swapWatches-def}\) \\
by \(<\text{auto simp add: clauseFalseIffAllLiteralsAreFalse inconsistentCharacterization}\) \\
ultimately show \(<?thesis>\) \\
using \(<\text{Cons(1)[of \ ?state' \ clause \ # \ newWl]}\) \\
using \(<\text{Cons(3) \ Cons(4) \ Cons(6)}\) \\
using \(<\text{getWatch1 \ ?state' \ clause} = \text{Some \ ?w_1}\) \\
using \(<\text{getWatch2 \ ?state' \ clause} = \text{Some \ ?w_2}\) \\
using \(<\text{Some \ literal} = \text{getWatch1 \ state \ clause}\) \\
using \(<\text{literalTrue \ ?w_1\) \(<\text{elements} (getM \ ?state')\)\) \\
using \(<\text{uniq \ Wl}\) \\
by \(<\text{auto simp add:Let-def}\) \\
next case False \\
show \(<?thesis>\) \\
proof \(<\text{cases getNonWatchedUnfalsifiedLiteral \ <\text{nth} (getF \ ?state')\) clause} \ ?w_1 \ ?w_2 \ (getM \ ?state')\) \\
  case \(<\text{Some \ l}'\) \\
  hence \(<l'\) el \(<\text{nth} (getF \ ?state')\) clause \(<\neg \text{literalFalse} \ l'} \ (<\text{elements} (getM \ ?state'))\) \\
  using \(<\text{getNonWatchedUnfalsifiedLiteralSomeCharacterization}\) \\
  by \text{auto} \\
let \(<\text{?state''} = \text{setWatch2 \ clause \ l'} \ ?state'\) \\
from \(<\text{Cons(2)}\) \\
  have \(<\text{InvariantWatchesEl} \ \ (getF \ ?state'')\) \ (<\text{getWatch1 \ ?state'')}\) \ (<\text{getWatch2 \ ?state'')}\) \\
  using \(<\text{l}' \ el \ (<\text{nth} (getF \ ?state'')\) \ (<\text{clause})\)\) \\
  unfolding \(<\text{InvariantWatchesEl-def}\) \\
  unfolding \(<\text{swapWatches-def}\) \\
  unfolding \(<\text{setWatch2-def}\) \\
  by \text{auto} \\
moreover \\
  from \(<\text{Cons(4)}\) \\
  have \(<\text{InvariantConsistent} \ (getM \ ?state'')\) \\
  unfolding \(<\text{setWatch2-def}\) \\
  unfolding \(<\text{swapWatches-def}\) \\
  by \text{simp} \\
moreover \\
  have \(<\text{getM \ ?state''} = \text{getM} \ state \land \text{getF \ ?state''} = \text{getF} \ state \land \text{getConflictFlag \ ?state'')} = \text{getConflictFlag} \ state\) \\
  unfolding \(<\text{swapWatches-def}\)
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 ∉ 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 ≠ ?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

423
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 ∧
  getSATFlag ?state" = getSATFlag state
  unfolding swapWatches-def
  by simp
moreover have ∀ c. c ∈ set Wl → Some literal = getWatch1 ?state"
c ∨ Some literal = getWatch2 ?state"
c
  using Cons(5)
  using (clause ∉ set Wl)
  unfolding swapWatches-def
  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))):
  unfolding swapWatches-def
  by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
ultimately show ?thesis
  using Cons(1)[of ?state" clause ≠ newWI]
  using Cons(3) Cons(4) Cons(6)
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  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 \texttt{InvariantWatchesEl-def}
unfolding \texttt{swapWatches-def}
unfolding \texttt{setReason-def}
by \texttt{auto}
moreover
from \texttt{Cons(4)}
have \texttt{InvariantConsistent (getM ?state''})
unfolding \texttt{swapWatches-def}
unfolding \texttt{setReason-def}
by \texttt{simp}
moreover
have \texttt{getM ?state''} = \texttt{getM state} \land
\texttt{getF ?state''} = \texttt{getF state} \land
\texttt{getSATFlag ?state''} = \texttt{getSATFlag state}
unfolding \texttt{swapWatches-def}
unfolding \texttt{setReason-def}
by \texttt{simp}
moreover
have \(\forall c. c \in \texttt{set Wl} \rightarrow \texttt{Some literal} = \texttt{getWatch1 ?state''} c \land \texttt{getWatch2 ?state''} c\)
using \texttt{Cons(5)}
using \texttt{clause \notin \texttt{set Wl'}}
unfolding \texttt{swapWatches-def}
unfolding \texttt{setReason-def}
by \texttt{auto}
moreover
have \(\neg \texttt{clauseFalse (nth (getF state) clause) (elements (getM state))} \)
using \(\texttt {?w1 el (nth (getF state) clause)}\)
using \(\neg \texttt{literalFalse ?w1 (elements (getM ?state'))} \)
using \texttt{InvariantConsistent (getM state)}
unfolding \texttt{InvariantConsistent-def}
unfolding \texttt{swapWatches-def}
by \(\texttt{auto simp add: clauseFalseIffAllLiteralsAreFalse inconsistentCharacterization}\)
ultimately
show \texttt{?thesis}
using \texttt{Cons(1)[of ?state'' clause \# newWl]}
using \texttt{Cons(3) Cons(4) Cons(6)}
using \texttt{(getWatch1 ?state' clause = Some ?w1)}
using \texttt{(getWatch2 ?state' clause = Some ?w2)}
using \texttt{(Some literal = getWatch1 state clause)}
using \texttt{\neg literalTrue ?w1 (elements (getM ?state'))} \)
using \texttt{None}
using \texttt{\neg literalFalse ?w1 (elements (getM ?state'))} \)
using \texttt{uniq WI'}
apply \texttt{(simp add: Let-def)}
unfolding \texttt{setReason-def}
unfolding \texttt{swapWatches-def}
by auto
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

  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')))
unfolding swapWatches-def
by (auto simp add: clauseFalseIffAllLiteralsAreFalse inconsistentCharacterization)

thus thesis
using True
using Cons(1)[of thesis state clause ≠ new Wl]
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 thesis'' = 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 ⟨clause /∈ set Wl′⟩
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
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 ≠ ?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
by auto
moreover
from Cons(4)
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 \( \forall c. \ c \in \text{set } Wl' \rightarrow \text{Some literal } = \text{getWatch1 } ?\text{state}'' \)
c \lor \text{Some literal} = \text{getWatch2 } ?\text{state}'' \ c
  using Cons(5)
  using (\text{clause } \notin \text{ set } Wl')
  unfolding \text{setWatch2-def}
  by auto
moreover
have \( \text{clauseFalse} (\text{nth } (\text{getF } ?\text{state}) \ \text{clause}) \ (\text{elements } (\text{getM } ?\text{state})) \)
  using \( \forall l. l \in (\text{nth } (\text{getF } ?\text{state}') \ \text{clause}) \ \land l \neq ?w1 \ \land l \neq ?w2 \)
  \( \rightarrow \ \text{literalFalse } l \ (\text{elements } (\text{getM } ?\text{state}')) \)
  using (\text{literalFalse } ?w1 (\text{elements } (\text{getM } ?\text{state}')))
  using (\text{literalFalse } ?w2 (\text{elements } (\text{getM } ?\text{state}')))
  by (auto simp add: \text{clauseFalseIfAllLiteralsAreFalse})
ultimately
show ?thesis
  using Cons(1)[of ?state'" clause # newWl]
  using Cons(3) Cons(4) Cons(6)
  using (\text{getWatch1 } ?\text{state}' clause = \text{Some } ?w1)
  using (\text{getWatch2 } ?\text{state}' clause = \text{Some } ?w2)
  using (\text{\neg Some literal } = \text{getWatch1 } ?\text{state} clause)
  using (\text{\neg literalTrue } ?w1 (\text{elements } (\text{getM } ?\text{state}')))
  using None
  using (\text{literalFalse } ?w1 (\text{elements } (\text{getM } ?\text{state}')))
  using (\text{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 \text{InvariantWatchesEl } (\text{getF } ?\text{state}'") (\text{getWatch1 } ?\text{state}'")
                               (\text{getWatch2 } ?\text{state}'")
          unfolding \text{InvariantWatchesEl-def}
          unfolding setReason-def
          by auto
moreover
  from Cons(4)
  have \text{InvariantConsistent } (\text{getM } ?\text{state}'")
          unfolding setReason-def
          by simp
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}
          unfolding setReason-def

429
by simp
moreover
have \( \forall c. \ c \in \text{set } Wl' \longrightarrow \text{Some literal} = \text{getWatch1 } \text{?state}'' \ c \)
  using Cons(\(5\))
  using \(\{\text{clause} \notin \text{set } Wl'\}\)
  unfolding setReason-def
  by auto
moreover
have \(\neg \text{clauseFalse } (\text{nth } (\text{getF } \text{state}) \text{ clause}) (\text{elements } (\text{getM } \text{state}))\)
  using \(\{\text{?w1 el } (\text{nth } (\text{getF } \text{state}) \text{ clause})\}\)
  using \(\neg \text{literalFalse } \text{?w1 } (\text{elements } (\text{getM } \text{state}''))\)
  using \(\neg \text{InvariantConsistent } (\text{getM } \text{state})\)
  unfolding InvariantConsistent-def
  by (auto simp add: clauseFalseIffAllLiteralsAreFalse inconsistentCharacterization)
ultimately
show \(\text{?thesis}\)
  using Cons(\(1\)][of \(\text{?state}'' \text{ clause} # \text{newWl}\])
  using Cons(\(3\)) Cons(\(4\)) Cons(\(6\)
  using (\text{getWatch1 } \text{?state}' \text{ clause} = \text{Some } \text{?w1})
  using (\text{getWatch2 } \text{?state}' \text{ clause} = \text{Some } \text{?w2})
  using (\neg \text{Some literal} = \text{getWatch1 } \text{state } \text{clause})
  using (\neg \text{literalTrue } \text{?w1 } (\text{elements } (\text{getM } \text{state}''))\)
  using None
  using (\neg \text{literalFalse } \text{?w1 } (\text{elements } (\text{getM } \text{state}''))\)
  using (uniq Wl')
  apply (simp add: Let-def)
  unfolding setReason-def
  by auto
  qed
  qed
  qed
  qed
  qed

lemma NotifyWatchesLoopQEffect:
fixes literal :: Literal and Wl :: nat list and newWl :: nat list and state :: State
assumes
\(\text{(getM } \text{state}) = M @ [(\text{opposite literal}, \text{decision})]\) and
\(\text{InvariantWatchesEl } (\text{getF } \text{state}) (\text{getWatch1 } \text{state}) (\text{getWatch2 } \text{state})\) and
\(\text{InvariantWatchesDiffer } (\text{getF } \text{state}) (\text{getWatch1 } \text{state}) (\text{getWatch2 } \text{state})\) and
\(\forall (c::\text{nat}).\ c \in \text{set } Wl \longrightarrow 0 \leq c \land c < \text{length } (\text{getF } \text{state})\) and
\(\text{InvariantConsistent } (\text{getM } \text{state})\) 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 Wl 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 wa
  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′ = swapWatches clause state
let ?w1 = wb
have \texttt{getWatch1} \texttt{?state'} \texttt{clause} = \texttt{Some} \texttt{?w1}
  using \texttt{(getWatch2} \texttt{state} \texttt{clause} = \texttt{Some} \texttt{wb})
unfolding \texttt{swapWatches-def}
by \texttt{auto}

let \texttt{?w2} = \texttt{wa}
have \texttt{getWatch2} \texttt{?state'} \texttt{clause} = \texttt{Some} \texttt{?w2}
  using \texttt{(getWatch1} \texttt{state} \texttt{clause} = \texttt{Some} \texttt{wa})
unfolding \texttt{swapWatches-def}
by \texttt{auto}

have \texttt{?w2} = \texttt{literal}
  using \texttt{(Some} \texttt{literal} = \texttt{getWatch1} \texttt{state} \texttt{clause})
  using \texttt{(getWatch2} \texttt{?state'} \texttt{clause} = \texttt{Some} \texttt{?w2})
unfolding \texttt{swapWatches-def}
by \texttt{simp}

hence \texttt{literalFalse} ?w2 (\texttt{elements} \texttt{(getM} \texttt{state}))
  using \texttt{(getM} \texttt{state}) = \texttt{M} \texttt{@[} \texttt{(opposite literal, decision)}\texttt{]}
by \texttt{simp}

from \texttt{InvariantWatchesEl} \texttt{(getF} \texttt{state}) \texttt{(getWatch1} \texttt{state}) \texttt{(getWatch2} \texttt{state})
  have \texttt{?w1 el} \texttt{(nth} \texttt{(getF} \texttt{state}) \texttt{clause}) \texttt{?w2 el} \texttt{(nth} \texttt{(getF} \texttt{state}) \texttt{clause})
    using \texttt{(getWatch1} \texttt{?state'} \texttt{clause} = \texttt{Some} \texttt{?w1})
    using \texttt{(getWatch2} \texttt{?state'} \texttt{clause} = \texttt{Some} \texttt{?w2})
    using \texttt{(clause < length} \texttt{(getF} \texttt{state})
    unfolding \texttt{InvariantWatchesEl-def}
    unfolding \texttt{InvariantWatchesEl-def}
    unfolding \texttt{InvariantWatchesEl-def}
    by \texttt{auto}

from \texttt{InvariantWatchesDiffer} \texttt{(getF} \texttt{state}) \texttt{(getWatch1} \texttt{state}) \texttt{(getWatch2} \texttt{state})
  have \texttt{?w1} \neq \texttt{?w2}
    using \texttt{(getWatch1} \texttt{?state'} \texttt{clause} = \texttt{Some} \texttt{?w1})
    using \texttt{(getWatch2} \texttt{?state'} \texttt{clause} = \texttt{Some} \texttt{?w2})
    using \texttt{(clause < length} \texttt{(getF} \texttt{state})
    unfolding \texttt{InvariantWatchesDiffer-def}
    unfolding \texttt{InvariantWatchesDiffer-def}
    unfolding \texttt{InvariantWatchesDiffer-def}
    by \texttt{auto}

show \texttt{?thesis}
proof (cases \texttt{literalTrue} ?w1 \texttt{(elements} \texttt{(getM} \texttt{?state')}\texttt{))}
  case True
  from \texttt{Cons}(3)
    have \texttt{InvariantWatchesEl} \texttt{(getF} \texttt{?state')} \texttt{(getWatch1} \texttt{?state')} \texttt{(getWatch2} \texttt{?state')}
      unfolding \texttt{InvariantWatchesEl-def}
unfolding swapWatches-def
by auto
moreover
from Cons(4)
have InvariantWatchesDiffer (getF ?state') (getWatch1 ?state')
  (getWatch2 ?state')
  unfolding InvariantWatchesDiffer-def
  unfolding swapWatches-def
  by auto
moreover
have getF ?state' = getF state 
  getM ?state' = getM state 
  getQ ?state' = getQ state 
  getConflictFlag ?state' = getConflictFlag state
unfolding swapWatches-def
by simp
moreover
have \forall c. c \in set Wl' \rightarrow Some literal = getWatch1 ?state' c \lor
  Some literal = getWatch2 ?state' c
using Cons(7)
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 \neg (\exists l. isUnitClause (nth (getF state) clause) l (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: isUnitClause-def inconsistentCharacterization)
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 (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) ¬ 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' el (nth (getF ?state') clause))
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)
using (getWatch1 ?state' clause = Some ?w1)
using (getWatch2 ?state' clause = Some ?w2)
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
∀ Some literal = getWatch2 ?state'' c  
using Cons(7)  
using ⟨clause ∈ set Wl⟩ 
unfolding swapWatches-def 
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 el ((getF ?state'') ! c) ∧ literalTrue l (elements M) ∧ elementLevel l M ≤ elementLevel (opposite ww1) M) ∨  
   (∀ 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 
   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 el (getF state ! clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 → literalFalse l (elements M))  
   using Cons(2)  
   using l' ≠ ?w1 and l' ≠ ?w2; l' el (nth (getF ?state'))  
   clause):  
   using ⟨¬ literalFalse l' (elements (getM ?state'))⟩  
   using a and b
using \( c = \text{clause} \)
unfolding swapWatches-def
unfolding setWatch2-def
by auto
moreover
have \( (\exists l \in \text{el (getF state) ! clause}) \land \text{literalTrue l (elements M)} \) \land
\( \text{elementLevel l M} \leq \text{elementLevel (opposite ?w1) M} \) \lor
(\( \forall l \in \text{el (getF state) ! clause}) \land l \neq ?w1 \land l \neq ?w2 \rightarrow \text{literalFalse l (elements M)} \)
using Cons(9)
unfolding InvariantWatchCharacterization-def
unfolding watchCharacterizationCondition-def
using \( \langle \text{clause < length (getF state)} \rangle \)
using \( \langle \text{getWatch1 ?state} \text{ clause = Some ?w1}\rangle [\text{THEN sym}] \)
using \( \langle \text{getWatch2 ?state} \text{ clause = Some ?w2}\rangle [\text{THEN sym}] \)

using \( \langle \text{literalFalse ww1 (elements M)} \rangle \)
using \( \langle \text{ww1 = ?w1} \rangle \)

unfolding setWatch2-def
unfolding swapWatches-def
by auto
ultimately
show ?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::nat and ww1::Literal and ww2::Literal 
assume a: \( 0 \leq c \land c < \text{length (getF ?state")} \land \text{Some ww1 = (getWatch1 ?state" c) \land Some ww2 = (getWatch2 ?state" c)} \)

assume b: \text{literalFalse ww2 (elements M)} \}

have \( (\exists l \in \text{el (getF ?state") ! c}) \land \text{literalTrue l (elements M)} \) \land \text{elementLevel l M} \leq \text{elementLevel (opposite ww2) M} \land
(\( \forall l \in \text{el (getF ?state") ! c}) \land l \neq \text{ww1} \land l \neq \text{ww2} \rightarrow \text{literalFalse l (elements M)} \land \text{elementLevel (opposite l) M} \leq \text{elementLevel (opposite ww2) M} \)
proof (cases \( c = \text{clause} \))
case False
thus ?thesis
using a and b
using Cons(9)
unfolding InvariantWatchCharacterization-def
unfolding watchCharacterizationCondition-def
unfolding swapWatches-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
unfolding swapWatches-def

by auto

with (∼ literalFalse l' (elements (getM ?state')));

Cons(2)

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 ∼ (∃ l. isUnitClause (nth (getF state) clause) l (elements (getM state))))

proof –
{
  assume ∼ ?thesis
  then obtain l
    where isUnitClause (nth (getF state) clause) l (elements (getM state))

    by auto
    with l' el (nth (getF ?state') clause); (∼ literalFalse l'
    (elements (getM ?state')))
    have l = l'

    unfolding isUnitClause-def
    unfolding swapWatches-def
    by auto
    with (l' ≠ ?w1) have
    literalFalse ?w1 (elements (getM ?state'))
using \langle\text{isUnitClause} (\text{nth} (\text{getF} \text{ state}) \text{ clause}) \  l \ (\text{elements} (\text{getM} \text{ state}))\rangle

\begin{align*}
\text{using} \; \langle\text{?w1 el} (\text{nth} (\text{getF} \text{ state}) \text{ clause})\rangle
\text{unfolding} \; \text{isUnitClause-def} \\
\text{unfolding} \; \text{swapWatches-def} \\
\text{by} \; \text{simp} \\
\text{with} \; (\text{?w1} \neq \text{?w2}) \; (\text{?w2} = \text{literal}) \\
\text{Cons}(2) \\
\text{have} \; \text{literalFalse} \; \text{?w1} \; (\text{elements} \; M) \\
\text{unfolding} \; \text{swapWatches-def} \\
\text{by} \; \text{simp} \\
\end{align*}

\text{from} \; \langle\text{isUnitClause} (\text{nth} (\text{getF} \text{ state}) \text{ clause}) \  l \ (\text{elements} (\text{getM} \text{ state}))\rangle

\text{Cons}(6) \\
\text{have} \; \lnot \; (\exists \; l. \; (l \text{ el} \; (\text{nth} (\text{getF} \text{ state}) \text{ clause}) \land \text{literalTrue} \; l \\
\; (\text{elements} \; (\text{getM} \text{ state})))) \\
\text{using} \; \text{containsTrueNotUnit[of - (\text{nth} (\text{getF} \text{ state}) \text{ clause})}] \\
\text{elements} \; (\text{getM} \text{ state})] \\
\text{unfolding} \; \text{InvariantConsistent-def} \\
\text{by} \; \text{auto} \\

\text{from} \; \langle\text{InvariantWatchCharacterization} \; (\text{getF} \text{ state}) \; (\text{getWatch1 state}) \; (\text{getWatch2 state}) \; M\rangle

\langle\text{clause} < \text{length} \; (\text{getF} \text{ state})\rangle \\
\langle\text{literalFalse} \; \text{?w1} \; (\text{elements} \; M)\rangle \\
\langle\text{getWatch1 \; state'} \; \text{clause} = \text{Some} \; \text{?w1} \; \text{[THEN sym]}\rangle \\
\langle\text{getWatch2 \; state'} \; \text{clause} = \text{Some} \; \text{?w2} \; \text{[THEN sym]}\rangle \\
\text{have} \; (\exists \; l. \; l \text{ el} \; (\text{getF} \text{ state} ! \text{ clause}) \land \text{literalTrue} \; l \\
\; (\text{elements} \; (\text{getM} \text{ state}))) \\
\; \land \; \text{elementLevel} \; l \; M \leq \text{elementLevel} \; (\text{opposite} \; \text{?w1}) \; M \lor \\
\; (\forall \; l. \; l \text{ el} \; (\text{getF} \text{ state} ! \text{ clause}) \land l \neq \text{?w1} \land l \neq \text{?w2} \; \rightarrow \\
\; \text{literalFalse} \; l \; (\text{elements} \; M)) \\
\text{unfolding} \; \text{InvariantWatchCharacterization-def} \\
\text{unfolding} \; \text{watchCharacterizationCondition-def} \\
\text{unfolding} \; \text{swapWatches-def} \\
\text{by} \; \text{auto} \\

\text{with} \; (\lnot \; (\exists \; l. \; (l \text{ el} \; (\text{nth} (\text{getF} \text{ state}) \text{ clause}) \land \text{literalTrue} \; l \\
\; (\text{elements} \; (\text{getM} \text{ state})))))) \\
\text{Cons}(2) \\
\text{have} \; (\forall \; l. \; l \text{ el} \; (\text{getF} \text{ state} ! \text{ clause}) \land l \neq \text{?w1} \land l \neq \text{?w2} \; \rightarrow \\
\; \text{literalFalse} \; l \; (\text{elements} \; M)) \\
\text{by} \; \text{auto} \\

\text{with} \; (l' \; \text{ el} \; (\text{getF} \; \text{state'} \; ! \text{ clause}) \lor l' \neq \text{?w1} \land l' \neq \text{?w2} \; \rightarrow \\
\; \text{literalFalse} \; l' \; (\text{elements} \; (\text{getM} \; \text{state'}))) \\
\text{Cons}(2) \\
\text{have} \; \text{False} \\
\text{unfolding} \; \text{swapWatches-def} \\
\text{by} \; \text{simp} \\
\}

438
thus \( ? \text{thesis} \)

by auto

qed

ultimately

show \( ? \text{thesis} \)

using \( \text{Cons}(1) \)[of \( \text{state}'' \) newWl]

using \( \text{Cons}(2) \) \( \text{Cons}(5) \) \( \text{Cons}(6) \)

using \( \langle \text{getWatch1 ?state'} \text{ clause} = \text{Some} \ ?w1 \rangle \)

using \( \langle \text{getWatch2 ?state'} \text{ clause} = \text{Some} \ ?w2 \rangle \)

using \( \langle \neg \text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}')) \rangle \)

using \( \langle \text{uniq Wl} \rangle \)

using \( \langle \text{getWatch1} \ ?\text{state} \text{ clause} \rangle \)

using \( \langle \text{getWatch2} \ ?\text{state} \text{ clause} \rangle \)

using \( \langle \neg \text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}')) \rangle \)

using \( \langle \text{uniq Wl} \rangle \)

by \( (\text{simp add: Let-def}) \)

next

case None

hence \( \forall l. \ l \text{ el} \ (\text{nth} \ (\text{getF} \ ?\text{state}') \text{ clause}) \land l \neq \ ?w1 \land l \neq \ ?w2 \rightarrow \neg \text{literalFalse} \ l \ (\text{elements} \ (\text{getM} \ ?\text{state}')) \)

using \( \text{getNonWatchedUnfalsifiedLiteralNoneCharacterization} \)

by simp

show \( ? \text{thesis} \)

proof (cases literalFalse \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}')))

case True

let \( ?\text{state}'' = ?\text{state}'[\text{getConflictFlag} := \text{True}, \text{getConflict-Clause} := \text{clause}] \)

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 \( \forall c \in \text{set } Wl' \rightarrow \text{Some literal} = \text{getWatch1 } ?\text{state}'' c \)
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 (\( \forall l. l \in \text{el} (\text{nth } (\text{getF } ?\text{state}) \text{ clause}) \land l \neq ?w1 \land l \neq ?w2 \rightarrow \text{literalFalse } l (\text{elements } (\text{getM } ?\text{state})) \))
using (\( \exists l. \text{isUnitClause } (\text{nth } (\text{getF } ?\text{state}) \text{ clause}) l (\text{elements } (\text{getM } ?\text{state})) \))
unfolding isUnitClause-def
by (simp add: clauseFalseIffAllLiteralsAreFalse)
hence \( \neg (\exists l. \text{isUnitClause } (\text{nth } (\text{getF } ?\text{state}) \text{ clause}) l (\text{elements } (\text{getM } ?\text{state})) ) \)
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 (literalTrue ?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") (getWatch2 ?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
using Cons(7)
using (clause \notin 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)
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 (¬ 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 (let 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 ⟨let state' = notifyWatches-loop literal (clause # Wl) newWl state in
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
  ?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⟩
443
using Cons(5) unfolding swapWatches-def
unfolding setReason-def
by (simp add: Let-def)
hence set (getQ ?state") ⊆ set (getQ (notifyWatches-loop
literal Wl' (clause # newWl) ?state"))
using prefixIsSubset[of getQ ?state" getQ (notifyWatches-loop
literal Wl' (clause # newWl) ?state")]
by auto hence l ∈ set (getQ (notifyWatches-loop literal Wl' (clause #
newWl) ?state"))
using {l = w1}
unfolding swapWatches-def
unfolding setReason-def
by auto
}
thus ?thesis using (notifyWatches-loop literal Wl' (clause #
newWl) ?state"") = notifyWatches-loop literal (clause # Wl') newWl state
by (simp add: Let-def)
next case False hence c ∈ set Wl'
using {c ∈ set (clause # Wl')}
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 #
Wl') newWl state in
?Cond1 state' ?state" ∧ ?Cond2 Wl' state' ?state"
∧ c ∈ set Wl'}
have let state' = notifyWatches-loop literal (clause #
Wl') newWl state in l ∈ set (getQ state')
by (simp add: Let-def)
}
thus ?thesis by (simp add: Let-def)
qed
}
hence ?Cond2 (clause # Wl') (notifyWatches-loop literal
(clause # Wl') newWl state) state
by (simp add: Let-def)

444
ultimately

show \$thesis
  by (simp add: Let-def)
qed
qed
qed

next
case False
let \$state' = state
let \$w1 = wa
have getWatch1 \$state' clause = Some \$w1
  using \(\text{getWatch1 state clause} = \text{Some wa}\)
  unfolding swapWatches-def
  by auto
let \$w2 = wb
have getWatch2 \$state' clause = Some \$w2
  using \(\text{getWatch2 state clause} = \text{Some wb}\)
  unfolding swapWatches-def
  by auto

from \(\neg \text{Some literal} = \text{getWatch1 state clause}\)
  \(\forall (c :: \text{nat}). c \in \text{set (clause \# Wl')} \rightarrow \text{Some literal} = (\text{getWatch1 state c})\)
  \(\vee \text{Some literal} = (\text{getWatch2 state c})\)
have Some literal = getWatch2 state clause
  by auto
hence \$w2 = literal
  using \(\text{getWatch2 \$state' clause} = \text{Some \$w2}\)
  by simp
hence literalFalse \$w2 (elements (getM state))
  using Cons (2)
  by simp

from InvariantWatchesEl \(\text{(getF state)}\) (getWatch1 state) (getWatch2 state)
  have \$w1 el (nth (getF state) clause) \$w2 el (nth (getF state) clause)
  using \(\text{getWatch1 \$state' clause} = \text{Some \$w1}\)
  using \(\text{getWatch2 \$state' clause} = \text{Some \$w2}\)
  using \(\text{clause < length (getF state)}\)
  unfolding InvariantWatchesEl-def
  unfolding swapWatches-def
  by auto

from InvariantWatchesDiffer \(\text{(getF state)}\) (getWatch1 state) (getWatch2 state)
  have \$w1 \neq \$w2
  using \(\text{getWatch1 \$state' clause} = \text{Some \$w1}\)
using \langle getWatch2 ?state' clause = Some ?w2 \rangle
using \langle clause < length (getF state) \rangle
unfolding InvariantWatchesDiffer-def
unfolding swapWatches-def
by auto

show ?thesis
proof (cases literalTrue ?w1 (elements (getM ?state')))
case True
  have \neg (\exists l. isUnitClause (nth (getF state) clause) l (elements (getM state)))
    using \langle ?w1 el (nth (getF state) clause) \rangle
    using \langle literalTrue ?w1 (elements (getM ?state')) \rangle
    unfolding InvariantConsistent-def
    by (auto simp add: isUnitClause-def inconsistentCharacterization)
thus ?thesis
using True
using Cons(1)[of ?state' clause \neq newWl]
using Cons(2) Cons(3) Cons(4) Cons(5) Cons(6) Cons(7) Cons(8) Cons(9)
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'))
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) \neg literalFalse l' (elements (getM ?state')) l' \neq ?w1 l' \neq ?w2
    using getNonWatchedUnfalsifiedLiteralSomeCharacterization
    by auto

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
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
unfolding \textit{watchCharacterizationCondition-def}
unfolding \textit{setWatch2-def}
by \texttt{auto}

next
  case True
  with \(a\)
  have \(\text{ww1} = \textit{?w1} \text{ and } \text{ww2} = \textit{l'}\)
    using \(\text{getWatch1 ?state' clause} = \textit{Some ?w1}\)
    using \(\text{getWatch2 ?state' clause} = \textit{Some ?w2}\)[THEN
sym]
unfolding \textit{setWatch2-def}
by \texttt{auto}

have \(\neg (\forall \text{l. } \text{l el (getF state} ! \text{ clause}) \land \text{l} \neq \textit{?w1} \land \text{l} \neq \textit{?w2} \longrightarrow \text{literalFalse l (elements M)})\)
  using \(\text{l'} \neq \textit{?w1} \text{ and } \text{l'} \neq \textit{?w2}\) \(\text{l'} \text{ el (nth (getF ?state')) clause})\):
    using \(\neg \text{literalFalse l'} (\text{elements (getM ?state'))}\)
    using \texttt{Cons(2)}
    using \texttt{a and b}
    using \(\langle c = \text{clause}\rangle\)
unfolding \textit{setWatch2-def}
by \texttt{auto}
moreover
have \(\exists \text{l. } \text{l el (getF state} ! \text{ clause}) \land \text{literalTrue l (elements M)} \land \text{elementLevel l} \leq \text{elementLevel (opposite ?w1) M} \lor \langle \forall \text{l. } \text{l el (getF state} ! \text{ clause}) \land \text{l} \neq \textit{?w1} \land \text{l} \neq \textit{?w2} \longrightarrow \text{literalFalse l (elements M)}\rangle\)
  using \texttt{Cons(9)}
unfolding \textit{InvariantWatchCharacterization-def}
unfolding \textit{watchCharacterizationCondition-def}
using \(\langle \text{clause < length (getF state)}\rangle\)
  using \(\text{getWatch1 ?state' clause} = \textit{Some ?w1}\)[THEN
sym]
  using \(\text{getWatch2 ?state' clause} = \textit{Some ?w2}\)[THEN
sym]
unfolding \(\text{literalFalse \text{ww1 (elements M)}}\)
using \(\text{ww1} = \textit{?w1}\)
unfolding \textit{setWatch2-def}
by \texttt{auto}
ultimately
show \textit{thesis}
  using \(\text{ww1} = \textit{?w1}\)
  using \(\langle c = \text{clause}\rangle\)
unfolding \textit{setWatch2-def}
by \texttt{auto}
qed
}
moreover
\{ 
  \textbf{fix}\ c::\textsc{nat} \ \textbf{and} \ \textit{ww1::Literal} \ \textbf{and} \ \textit{ww2::Literal} \\
  \textbf{assume}\ a: \ 0 \leq c \wedge c < \text{length} \ (\text{getF} ?\text{state}'' ) \ \wedge \ \exists\ \textit{ww1} = (\text{getWatch1} ?\text{state}'' \ c) \ \wedge \ \exists\ \textit{ww2} = (\text{getWatch2} ?\text{state}'' \ c) \\
  \textbf{assume}\ b: \ \text{literalFalse} \ \textit{ww2} \ (\\text{elements} \ M) \\
  \textbf{have}\ (\exists l. \ l \ (\text{set} \ (\text{getF} ?\text{state}'' ) \ ! c) \ \wedge \ \text{literalTrue} \ l \ (\\text{elements} \ M) \ \wedge \ \text{elementLevel} \ l \ M \ \leq \ \text{elementLevel} \ (\text{opposite} \ \textit{ww2} ) \ M) \ \vee \\
  (\forall \ l. \ l \ (\text{set} \ (\text{getF} ?\text{state}'' ) ! c) \ \wedge \ l \neq \ \textit{ww1} \ \wedge \ l \neq \ \textit{ww2} \ \longrightarrow \\
  \ \text{literalFalse} \ l \ (\\text{elements} \ M) \ \wedge \ \text{elementLevel} \ (\text{opposite} \ l) \ M \ \leq \ \text{elementLevel} \ (\text{opposite} \ \textit{ww2} ) \ M \\
  \textbf{proof} \ (\text{cases} \ c = \text{clause}) \\
  \text{case} \ \text{False} \\
  \textbf{thus} \ \textit{?thesis} \\
  \textbf{using}\ a \ \textbf{and} \ b \ \\
  \textbf{using} \ \textit{Cons}(9) \\
  \textbf{unfolding} \ \textit{InvariantWatchCharacterization-def} \\
  \textbf{unfolding} \ \textit{watchCharacterizationCondition-def} \\
  \textbf{unfolding} \ \textit{setWatch2-def} \\
  \textbf{by} \ \textit{auto} \\
  \textbf{next} \\
  \textbf{case} \ \text{True} \\
  \textbf{with} \ a \\
  \textbf{have} \ \textit{ww1} = ?w1 \ \textbf{and} \ \textit{ww2} = l' \\
  \textbf{using} \ (\text{getWatch1} ?\text{state}' \ \text{clause} = \text{Some} \ ?w1) \\
  \textbf{using} \ (\text{getWatch2} ?\text{state}' \ \text{clause} = \text{Some} \ ?w2)\ [\text{THEN} \ \textit{sym}] \\
  \textbf{unfolding} \ \textit{setWatch2-def} \\
  \textbf{by} \ \textit{auto} \\
  \textbf{with} \ (\neg \ \text{literalFalse} \ l' \ (\\text{elements} \ (\text{getM} ?\text{state}'))) \ b \\
  \textit{Cons}(2) \\
  \textbf{have} \ \text{False} \\
  \textbf{unfolding} \ \textit{setWatch2-def} \\
  \textbf{by} \ \textit{simp} \\
  \textbf{thus} \ \textit{?thesis} \\
  \textbf{by} \ \textit{simp} \\
  \textbf{qed} \\
\} \\
\textbf{ultimately} \\
\textbf{show} \ \textit{?thesis} \\
\textbf{unfolding} \ \textit{InvariantWatchCharacterization-def} \\
\textbf{unfolding} \ \textit{watchCharacterizationCondition-def} \\
\textbf{by} \ \textit{blast} \\
\textbf{qed} \\
\textbf{moreover} \\
\textbf{have}\ (\neg \ (\exists \ l. \ \text{isUnitClause} \ \text{(nth} \ (\text{getF} \ \text{state}) \ \text{clause}) \ l \ (\\text{elements} \ (\text{getM} \ \text{state}))) \\
\} \\
\textbf{proof}--
\}
ultimately 
\textbf{show} \ \textit{?thesis} \\
\textbf{unfolding} \ \textit{InvariantWatchCharacterization-def} \\
\textbf{unfolding} \ \textit{watchCharacterizationCondition-def} \\
\textbf{by} \ \textit{blast} \\
\textbf{qed} \\
\textbf{moreover} \\
\textbf{have}\ (\neg \ (\exists \ l. \ \text{isUnitClause} \ \text{(nth} \ (\text{getF} \ \text{state}) \ \text{clause}) \ l \ (\\text{elements} \ (\text{getM} \ \text{state}))) \\
\} \\
\textbf{proof}--

449
assume \( \neg \text{thesis} \)
then obtain \( l \)
where \( \text{isUnitClause} \ (\text{nth} \ (\text{getF} \ \text{state}) \ \text{clause}) \ l \ (\text{elements} \ (\text{getM} \ \text{state})) \)
  by auto
with \( l' \ el \ (\text{nth} \ (\text{getF} \ ?\text{state}') \ \text{clause}); \ (\neg \ \text{literalFalse} \ l') \ (\text{elements} \ (\text{getM} \ ?\text{state}')) \)
  have \( l = l' \)
  unfolding \( \text{isUnitClause-def} \)
  by auto
with \( (l' \neq \ ?w1) \ have \)
  \text{literalFalse} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'))
  using \( \text{isUnitClause} \ (\text{nth} \ (\text{getF} \ \text{state}) \ \text{clause}) \ l \ (\text{elements} \ (\text{getM} \ \text{state})) \)
  using \( (?w1 \neq \ ?w2) \ (\neg \ ?w2 = \text{literal}) \)
  Cons(2)
  have \( \text{literalFalse} \ ?w1 \ (\text{elements} \ M) \)
  by simp
from \( \text{isUnitClause} \ (\text{nth} \ (\text{getF} \ \text{state}) \ \text{clause}) \ l \ (\text{elements} \ (\text{getM} \ \text{state})) \)
  Cons(6)
  have \( \neg \ (\exists \ l. \ (l \ el \ (\text{nth} \ (\text{getF} \ \text{state}) \ \text{clause}) \ \wedge \ \text{literalTrue} \ l \ (\text{elements} \ (\text{getM} \ \text{state})))) \)
  using \( \text{containsTrueNotUnit[of - (\text{nth} \ (\text{getF} \ \text{state}) \ \text{clause}) \ \text{elements} \ (\text{getM} \ \text{state})]} \)
  unfolding \( \text{InvariantConsistent-def} \)
  by auto
from \( \text{InvariantWatchCharacterization} \ (\text{getF} \ \text{state}) \ (\text{getWatch1} \ \text{state}) \ (\text{getWatch2} \ \text{state}) \ M \)
  \( \langle \text{clause} < \text{length} \ (\text{getF} \ \text{state}) \rangle \)
  \( \langle \text{literalFalse} \ ?w1 \ (\text{elements} \ M) \rangle \)
  \( \langle \text{getWatch1} \ ?\text{state}' \ \text{clause} = \text{Some} \ ?w1 \ \text{THEN} \ \text{sym} \rangle \)
  \( \langle \text{getWatch2} \ ?\text{state}' \ \text{clause} = \text{Some} \ ?w2 \ \text{THEN} \ \text{sym} \rangle \)
  have \( (\exists \ l. \ l \ el \ (\text{getF} \ \text{state} ! \ \text{clause}) \ \wedge \ \text{literalTrue} \ l \ (\text{elements} \ M) \ \wedge \ \text{elementLevel} \ l \ M \ \leq \ \text{elementLevel} \ (\text{opposite} \ ?w1) \ M) \ \vee \)
  \( (\forall \ l. \ l \ el \ (\text{getF} \ \text{state} ! \ \text{clause}) \ \wedge \ l \neq \ ?w1 \ \wedge \ l \neq \ ?w2 \ \rightarrow \ \text{literalFalse} \ l \ (\text{elements} \ M) ) \)
  unfolding \( \text{InvariantWatchCharacterization-def} \)
  unfolding \( \text{watchCharacterizationCondition-def} \)
  unfolding \( \text{swapWatches-def} \)
  by auto
with \( \neg \ (\exists \ l. \ (l \ el \ (\text{nth} \ (\text{getF} \ \text{state}) \ \text{clause}) \ \wedge \ \text{literalTrue} \ l \ (\text{elements} \ (\text{getM} \ \text{state})))) \)

450
Cons(2)
  have (∀ l. l ∈ (get F state ! clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 → literalFalse l (elements M))
  by auto
  with (l' ∈ (get F ?state' ! clause) ∨ l' ≠ ?w1 ∨ l' ≠ ?w2) (¬ literalFalse l' (elements (get M ?state')))
Cons(2)
  have False
  unfolding swapWatches-def
  by simp
  }
thus ?thesis
  by auto
qed
ultimately
show ?thesis
using Cons(1)[of ?state'' new Wl]
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 (get M ?state')))
using (uniq Wl')
using Some
by (simp add: Let-def)
next
case None
hence ∀ l. l ∈ (nth (get F ?state') clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 → literalFalse l (elements (get M ?state'))
  using getNonWatchedUnfalsifiedLiteralNoneCharacterization
  by simp
show ?thesis
proof (cases literalFalse ?w1 (elements (get M ?state')))
  case True
  let ?state'' = ?state'[(getConflictFlag := True, getConflict-Clause := clause)]

  from Cons(3)
  have InvariantWatchesEl (get F ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
    unfolding InvariantWatchesEl-def
    by auto
  moreover
  from Cons(4)
  have InvariantWatchesDiffer (get F ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
    unfolding 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)
  unfolding setWatch1-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 l1 l el (nth (getF ?state') clause) ∧ l1 ≠ ?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 ∨ Some literal = getWatch2 ?state'' c
  using Cons(7)
  unfolding setReason-def
  by auto
moreover
have InvariantWatchCharacterization (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'') M
  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''
  using Cons(1)[of ?state'' clause # newWl]
  using Cons(2) Cons(5) Cons(6) Cons(7)
  using (uniq Wl')
  by (simp add: Let-def)
moreover

have \texttt{notifyWatches-loop literal \(Wl\)' (clause \# new\(Wl\)) \(?state''\) = notifyWatches-loop literal (clause \# \(Wl\)') new\(Wl\) state

  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')))

ultimately

have let \(state' = \) notifyWatches-loop literal (clause \# \(Wl\)') new\(Wl\) state in

  \(?Cond1\) state' \(?state'' \land \) ?Cond2 \(Wl\) state \(?state''\)

by simp

have isUnitClause (nth (getF state) clause) \(?w1\) (elements (getM state))

  using (\(\forall\) \(l\). \(l\) el (nth (getF ?state') clause) \land \(?w1\ \land \(l\) \neq \(?w2\)))
  using (?w1 el (nth (getF state) clause))
  using (?w2 el (nth (getF state) clause))
  using (\(\neg\) literalFalse \(?w2\) (elements (getM state)))
  using (\(\neg\) 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 \# \(Wl\)') new\(Wl\) state in
  \(l\) \(\in\) set (getQ state') \(\setminus\) set (getQ state)
  have \(\exists\) clause. clause el (getF state) \land literal el clause \land
  isUnitClause clause \(l\) (elements (getM state))
  proof (cases \(l\) \(\neq\) \(?w1\))
    case True
    hence let \(state' = \) notifyWatches-loop literal (clause \# \(Wl\)') new\(Wl\) state in
    \(l\) \(\in\) set (getQ state') \(\setminus\) set (getQ \(?state'\))
  using (let \(state' = \) notifyWatches-loop literal (clause \# \(Wl\)') new\(Wl\) state in
  \(l\) \(\in\) set (getQ state') \(\setminus\) set (getQ state)
  unfolding setReason-def
  unfolding swapWatches-def
  by (simp add: Let-def)
with \(\text{let state}' = \text{notifyWatches-loop literal (clause \# Wl')}\) newWl state in
\[\text{show} \ ?\text{thesis}\]
unfolding setReason-def
unfolding swapWatches-def
by (simp add: Let-def del: notifyWatches-loop.simps)

next
case False
thus \(\text{thesis}\) using
\[\langle \text{isUnitClause (nth (getF state) clause) el (getF state)} \rangle \]
\[\langle \text{elements (getM state)} \rangle \]
\[\langle \text{isPrefix (getQ state'' (notifyWatches-loop literal Wl' (clause \# newWl) ?state''))} \rangle \]
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 set (getQ ?state") ⊆ set (getQ (notifyWatches-loop literal W1' (clause # newWl) ?state"))

using prefixIsSubset[of getQ ?state" getQ (notifyWatches-loop literal W1' (clause # newWl) ?state")]

    by auto
    hence l ∈ set (getQ (notifyWatches-loop literal W1' (clause # newWl) ?state"))
    using (l = ?w1)
    unfolding swapWatches-def
    unfolding setReason-def
    by auto

} thus ?thesis

using (notifyWatches-loop literal W1' (clause # newWl)

?state" = notifyWatches-loop literal (clause # W1') newWl state)

by (simp add: Let-def)

next

    case False
    hence c ∈ set W1'
    using (c ∈ set (clause # W1'))
    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 # W1') newWl state in
      ?Cond1 state' ?state" ∧ ?Cond2 W1' state' ?state")
      c ∈ set W1')
      have let state' = notifyWatches-loop literal (clause # W1') newWl state in l ∈ set (getQ state')
    by (simp add: Let-def)

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

qed

} hence ?Cond2 (clause # W1') (notifyWatches-loop literal (clause # W1') newWl state) state

by (simp add: Let-def)

ultimately

show ?thesis

by (simp add: Let-def)
lemma InvariantUniqQAfterNotifyWatchesLoop:
fixes literal :: Literal and Wl :: nat list and newWl :: nat list and state :: State
assumes
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state) and
\forall (c::nat). c \in set Wl \rightarrow 0 \leq c \land c < length (getF state) and
InvariantUniqQ (getQ state)
shows
let state' = notifyWatches-loop literal Wl newWl state in
InvariantUniqQ (getQ 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 set (clause # Wl') \rightarrow 0 \leq c \land c < length (getF state):
  have 0 \leq clause \land 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

qed
qed
qed
qed
qed
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
  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 (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 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" 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 ⟨¬ 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 # 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 ⟨¬ 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

459
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

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

thesis

proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state')

clause) ?w1 ?w2 (getM ?state'))

case (Some l')

hence l' el (nth (getF ?state'))

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'))

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(2) 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

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'')
  unfolding InvariantWatchesEl-def
  by auto
moreover
have getM ?state'' = getM state ∧
  getF ?state'' = getF state ∧
  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 (∼ 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 getM ?state'' = getM state
    getF ?state'' = getF state
    getQ ?state'' = (if ?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 ?w1 el (getQ state) then (getQ state) else (getQ state) @ [?w1])
    unfolding InvariantUniqQ-def

462
by \((\text{simp add: uniqAppendIff})\)
ultimately

show \(?\text{thesis}\)
using \(\text{Cons}(1)[\text{of } ?\text{state}']\)
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 \(\text{\neg Some literal } = \text{getWatch1 state clause}\)
using \(\text{\neg literalTrue } ?w1 \text{ (elements (getM } ?\text{state}')})\)
using None
using \(\text{\neg literalFalse } ?w1 \text{ (elements (getM } ?\text{state}')})\)
unfolding isPrefix-def
unfolding InvariantUniqQ-def
by \((\text{simp add: Let-def split: split-if-asm})\)
qed
qed
qed
qed

lemma InvariantConflictClauseCharacterizationAfterNotifyWatches:
assumes \( (\text{getM state}) = M @ [\text{(opposite literal, decision}] \) and
\(\text{InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)}\)
and
\(\forall \ (c::\text{nat}). \ c \in \text{set } Wl \mapsto 0 \leq c \land c < \text{length (getF state)} \) and
\(\forall \ (c::\text{nat}). \ c \in \text{set } Wl \mapsto \text{Some literal } = \text{(getWatch1 state } c) \) \quad \forall
\(\text{Some literal } = \text{(getWatch2 state } c)\) and
\(\text{InvariantConflictClauseCharacterization (getConflictFlag state) (getConflictClause state) (getF state) (getM state)}\)
uniq \(Wl\)
shows
let \(\text{state}' = \text{(notifyWatches-loop literal } Wl \text{ newWl state) in}
\text{InvariantConflictClauseCharacterization (getConflictFlag state') (getConflictClause state') (getF state') (getM state')}
using \(\text{assms}\)
proof \((\text{induct } Wl \text{ arbitrary: newWl state})\)
case Nil
thus \(?\text{case}\)
by \((\text{simp})\)
next
case \((\text{Cons clause } Wl')\)

from \((\text{uniq (clause } \# \text{ Wl')})\)
have \(\text{clause } \notin \text{ set } Wl' \text{ uniq } Wl'\)
by \((\text{auto simp add:uniqAppendIff})\)

from \((\forall \ (c::\text{nat}). \ c \in \text{set } (\text{clause } \# \text{ Wl'}) \mapsto 0 \leq c \land c < \text{length (getF state)})\)
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
by auto
with True have
?w2 = literal
unfolding swapWatches-def
by simp
hence literalFalse ?w2 (elements (getM state))
using Cons(2)
by simp
show ?thesis
proof (cases literalTrue ?w1 (elements (getM ?state’)))
case True
from Cons(3)
have InvariantWatchesEl (getF ?state’) (getWatch1 ?state’)
(getWatch2 ?state’)
unfolding InvariantWatchesEl-def
unfolding swapWatches-def
by auto
moreover
have $\forall c. c \in \text{set Wl’} \rightarrow \text{Some literal} = \text{getWatch1 ?state’ c} \lor$
Some literal = getWatch2 ?state’ c
using Cons(5)
unfolding swapWatches-def
by auto
moreover
have getM ?state’ = getM state \land

464
\begin{verbatim}
getF ?state' = getF state ∧
geconflictFlag ?state' = getConflictFlag state ∧
geconflictClause ?state' = getConflictClause state

unfolding swapWatches-def
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 (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 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(5)
  using clause ∉ set Wl'
  using swapWatchesEffect[of clause state]
  unfolding setWatch2-def
  by simp
moreover
have getM ?state'' = getM state ∧
geconflict ?state'' = getF state ∧
geconflictFlag ?state'' = getConflictFlag state ∧
geconflictClause ?state'' = getConflictClause state

465
\end{verbatim}
unfolding swapWatches-def
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 ⟨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(3)
  have InvariantWatchesEl (getF ?state’’) (getWatch1 ?state’’)
  have InvariantWatchesEl (getF ?state’’) (getWatch2 ?state’’)
    using Cons(5)
    using ⟨clause / set WI’⟩
    using swapWatchesEffect[of clause state]
    by simp
  moreover
  have ∀ l. l el (nth (getF ?state’’) clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 → literalFalse l (elements (getM ?state’’))
    using None
    using ⟨getWatch1 ?state’ clause = Some ?w1⟩
    using ⟨getWatch2 ?state’ clause = Some ?w2⟩
    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: clauseFalseIffAllLiteralsAreFalse)

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'')

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 \( \forall (c :: \text{nat}). c \in \text{set } Wl' \rightarrow \text{Some literal} = (\text{getWatch1 state" c}) \lor \text{Some literal} = (\text{getWatch2 state" c}) \)

using Cons(5)
using (clause \# set \( Wl' \))
using swapWatchesEffect[of clause state]

unfolding setReason-def
by simp

ultimately

show \(?\text{thesis} \)

using Cons(1) of \( ?\text{state} \) clause \# newWl
using Cons(2) Cons(4) Cons(6) Cons(7)
using (\text{getWatch1 state'} clause = Some \(?w1\))
using (\text{getWatch2 state'} clause = Some \(?w2\))
using (\text{Some literal} = \text{getWatch1 state clause})
using (\# literalTrue \(?w1\) (elements (\text{getM state'})))
using None
using (\# literalFalse \(?w1\) (elements (\text{getM state'})))
using (uniq \( Wl' \))
by (simp add: Let-def)

qed
qed
qed

next

case \( \text{False} \)

let \( ?w1 = wa \)

have \( \text{getWatch1 state'} clause = \text{Some } ?w1 \)
using (\text{getWatch1 state clause} = \text{Some } wa)
by auto

let \( ?w2 = wb \)

have \( \text{getWatch2 state'} clause = \text{Some } ?w2 \)
using (\text{getWatch2 state clause} = \text{Some } wb)
by auto

from (\# Some literal = \text{getWatch1 state clause})

(\forall (c :: \text{nat}). c \in (\text{clause } \# Wl') \rightarrow \text{Some literal} = (\text{getWatch1 state c})) \lor (\text{Some literal} = (\text{getWatch2 state c}))

have \( \text{Some literal} = \text{getWatch2 state clause} \)
by auto

hence \( ?w2 = \text{literal} \)
using (\text{getWatch2 state'} clause = \text{Some } ?w2)
by simp

hence \( \text{literalFalse } ?w2 \) (elements (\text{getM state}))
using Cons(2)
by simp

show \(?\text{thesis} \)

proof (cases literalTrue \(?w1\) (elements (\text{getM state'})))

468
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 \ Wl' \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, getConflict-Clause := clause)
from Cons(3)
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 \land
  \ getConflictFlag \ ?state'' \land
  \ getConflictClause \ ?state'' = clause
  by simp
moreover
have \ \forall \ (c::nat). \ c \in \ set \ Wl' \longrightarrow \ Some \ literal = (getWatch1 \ ?state'' \ c) \lor \ Some \ literal = (getWatch2 \ ?state'' \ c)
  using Cons(5)
  using \ (clause \notin \ set \ Wl')
  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 \ (getWatch1 \ ?state' clause = Some \ ?w1)
  using \ (getWatch2 \ ?state' clause = Some \ ?w2)
  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 \ (literalFalse \ ?w1 \ (elements \ (getM \ ?state')))
  using \ (literalFalse \ ?w2 \ (elements \ (getM \ state)));
  by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
moreover
have \ (nth \ (getF \ state) \ clause) \ el \ (getF \ state)
using \(0 \leq \text{clause} \land \text{clause} < \text{length}\ (\text{getF state})\)
using \(\text{nth-mem}[\text{of clause getF state}]\)
by \(\text{simp}\)
ultimately

show \(?\text{thesis}\)
using \(\text{Cons}(1)[\text{of } \text{?state}']\)
using \(\text{Cons}(2) \text{ Cons}(4) \text{ Cons}(6) \text{ Cons}(7)\)
using \(\text{getWatch1 }\text{?state}' \text{ clause } = \text{Some } ?w1\)
using \(\text{getWatch2 }\text{?state}' \text{ clause } = \text{Some } ?w2\)
using \(\neg \text{Some literal } = \text{getWatch1 state clause}\)
using \(\neg \text{literalTrue } ?w1 \text{ (elements } (\text{getM }\text{?state}'))\)
using \(\text{None}\)
using \(\neg \text{literalFalse } ?w1 \text{ (elements } (\text{getM }\text{?state}'))\)
using \(\text{uniq Wl}'\)
using \(\text{getWatch1 }\text{?state}' \text{ clause } = \text{Some } ?w1\)
using \(\neg \text{Some literal } = \text{getWatch1 state clause}\)
using \(\neg \text{literalTrue } ?w1 \text{ (elements } (\text{getM }\text{?state}'))\)
using \(\text{uniq Wl}'\)
using \(\text{getWatch2 }\text{?state}' \text{ clause } = \text{Some } ?w2\)
using \(\neg \text{Some literal } = \text{getWatch1 state clause}\)
using \(\neg \text{literalTrue } ?w1 \text{ (elements } (\text{getM }\text{?state}'))\)

471
using None
using (~ literalFalse ?w1 (elements (getM ?state')))
using (uniq Wl')
by (simp add: Let-def)
qed
qed
qed
qed

lemma InvariantGetReasonIsReasonQSubset:
assumes Q ⊆ Q' and
InvariantGetReasonIsReason GetReason F M Q'
shows
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
shows
let state' = notifyIsReason (newWl state Q)
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))
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 ?case
proof (cases $\text{Some literal} = \text{getWatch1 state clause}$)
case True
let $?state' = \text{swapWatches clause state}$
let $?w1 = wb
have $\text{getWatch1 } ?state' \text{ clause} = \text{Some } ?w1$
using $(\text{getWatch2 state clause} = \text{Some} wb)$
unfolding swapWatches-def
by auto
let $?w2 = wa
have $\text{getWatch2 } ?state' \text{ clause} = \text{Some } ?w2$
using $(\text{getWatch1 state clause} = \text{Some} wa)$
unfolding swapWatches-def
by auto
with True have $?w2 = \text{literal}$
unfolding swapWatches-def
by simp
hence $\text{literalFalse } ?w2 \ (\text{elements } (\text{getM state}))$
using Cons(6)
by simp

from $\text{InvariantWatchesEl } (\text{getF state}) \ (\text{getWatch1 state}) \ (\text{getWatch2 state})$
have $?w1 \in (\text{nth } (\text{getF state}) \text{ clause}) \ ?w2 \in (\text{nth } (\text{getF state}) \text{ clause})$
using $(\text{getWatch1 } ?state' \text{ clause} = \text{Some } ?w1)$
using $(\text{getWatch2 } ?state' \text{ clause} = \text{Some } ?w2)$
using $(0 \leq \text{clause} \land \text{clause} < \text{length } (\text{getF state}))$
unfolding InvariantWatchesEl-def
unfolding swapWatches-def
by auto

show ?thesis
proof (cases $\text{literalTrue } ?w1 \ (\text{elements } (\text{getM } ?state'))$)
case True

from Cons(2)
have $\text{InvariantWatchesEl } (\text{getF } ?state') \ (\text{getWatch1 } ?state')$ $(\text{getWatch2 } ?state')$
unfolding InvariantWatchesEl-def
unfolding swapWatches-def
by auto
moreover
have \( \forall c. c \in \text{set } Wl' \rightarrow \text{Some literal } = \text{getWatch1 } ?\text{state}' \ c \ \lor \ \text{Some literal } = \text{getWatch2 } ?\text{state}' \ c \)
using Cons(4)
unfolding swapWatches-def
by auto
moreover
have getM ?\text{state}' = getM state \land
getF ?\text{state}' = getF state \land
getQ ?\text{state}' = getQ state \land
getReason ?\text{state}' = getReason state
unfolding swapWatches-def
by simp
ultimately
show ?thesis
using Cons(1)[of ?\text{state}' Q clause # newWl]
using Cons(3) Cons(6) Cons(7)
using (getWatch1 ?\text{state}' clause = Some ?w1)
using (getWatch2 ?\text{state}' clause = Some ?w2)
using (Some literal = getWatch1 state clause)
using (literalTrue ?w1 (elements (getM ?\text{state}')))
using (uniq Wl')
by (simp add:Let-def)
next
case False
show ?thesis
proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?\text{state}')) clause) ?w1 ?w2 (getM ?\text{state}')
case (Some l')
hence l' el (nth (getF ?\text{state}')) clause
using getNonWatchedUnfalsifiedLiteralSomeCharacterization
by simp
let ?\text{state}'' = setWatch2 clause l' ?\text{state}'
from Cons(2)
have InvariantWatchesEl (getF ?\text{state}') (getWatch1 ?\text{state}'')
(getWatch2 ?\text{state}'')
using l' el (nth (getF ?\text{state}')) clause:
unfolding InvariantWatchesEl-def
unfolding swapWatches-def
unfolding setWatch2-def
by auto
moreover
have \( \forall (c::\text{nat}). c \in \text{set } Wl' \rightarrow \text{Some literal } = \text{getWatch1 } ?\text{state}'' \ c \ \lor \ \text{Some literal } = \text{getWatch2 } ?\text{state}'' \ c \)

474
using \(\text{Cons}(4)\)
using \(\text{clause} \notin \text{set } Wl\)
using \(\text{swapWatchesEffect}[\text{of clause state}]\)
unfolding \(\text{setWatch2-def}\)
by simp
moreover
have \(\text{getM ?state''} = \text{getM state} \land\)
  \(\text{getF ?state''} = \text{getF state} \land\)
  \(\text{getQ ?state''} = \text{getQ state} \land\)
  \(\text{getReason ?state''} = \text{getReason state}\)
unfolding \(\text{swapWatches-def}\)
unfolding \(\text{setWatch2-def}\)
by simp
ultimately
show \(?\text{thesis}\)
using \(\text{Cons}(1)[\text{of ?state''}\ Q \text{newWl}]\)
using \(\text{Cons}(3)\ \text{Cons}(6)\ \text{Cons}(7)\)
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 \(\neg \text{literalTrue ?w1} (\text{elements } (\text{getM ?state'}))\)
using Some
using \(\text{uniq Wl}\)
by (simp add: Let-def)

next
case None
  hence \(\forall l. l \in (\text{nths } (\text{getF ?state'})) \land l \neq \text{?w1} \land l \neq \text{?w2} \rightarrow \text{literalFalse} l (\text{elements } (\text{getM ?state'}))\)
  using \(\text{getNonWatchedUnfalsifiedLiteralNoneCharacterization}\)
  by simp
  show \(?\text{thesis}\)
proof (cases \(\text{literalFalse } ?w1 (\text{elements } (\text{getM ?state'}))\))
case True
  let \(?\text{state''} = )\text{state'}(\text{getConflictFlag := True, getConflict-Clause := clause})\)

  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 auto
moreover
  have \(\forall c. c \in \text{set } Wl' \rightarrow \text{Some literal} = \text{getWatch1 ?state''} c \lor \text{Some literal} = \text{getWatch2 ?state''} c\)
  using \(\text{Cons}(4)\)
  unfolding \(\text{swapWatches-def}\)
  by auto
moreover
have \( \text{getM} \ ?\text{state}''' = \text{getM} \ \text{state} \land \\
\text{getF} \ ?\text{state}''' = \text{getF} \ \text{state} \land \\
\text{getQ} \ ?\text{state}''' = \text{getQ} \ \text{state} \land \\
\text{getReason} \ ?\text{state}''' = \text{getReason} \ \text{state} \)
unfolding \( \text{swapWatches-def} \)
by simp
ultimately
show \( \text{?thesis} \)
using \( \text{Cons}(1) \) \( \text{of} \ ?\text{state}''' \text{Qclause} \ # \ \text{newWI} \)
using \( \text{Cons}(3) \) \( \text{Cons}(6) \) \( \text{Cons}(7) \)
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 \( \neg \text{literalTrue} \ ?w1 \ (\text{elements} (\text{getM} \ ?\text{state}') \)\)
using None
using \( \text{(literalFalse} \ ?w1 \ (\text{elements} (\text{getM} \ ?\text{state}') \)\)
using \( \text{uniq WI}' \)
by (simp add: \( \text{Let-def} \))

next
case False
let \( ?\text{state}'' = \text{setReason} \ ?w1 \ \text{clause} \ (\ ?\text{state}'(\text{getQ := (if} \ ?w1 \ \text{el} (\text{getQ} \ ?\text{state}') \ \text{then} (\text{getQ} \ ?\text{state}') \ \text{else} (\text{getQ} \ ?\text{state}') @ [\ ?w1]))) \)
let \( ?\text{state0} = \text{notifyWatches-loop} \ \text{literal WI}' \ (\text{clause} \ # \ \text{newWI}) \ ?\text{state}'' \)

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 \( \text{getM} \ ?\text{state}''' = \text{getM} \ \text{state} \land \\
\text{getF} \ ?\text{state}''' = \text{getF} \ \text{state} \land \\
\text{getQ} \ ?\text{state}''' = (\text{if} \ ?w1 \ \text{el} (\text{getQ} \ ?\text{state}) \ \text{then} (\text{getQ} \ \text{state}) \ \text{else} (\text{getQ} \ \text{state}) @ [\ ?w1]) \)
\( \text{getReason} \ ?\text{state}''' = (\text{getReason} \ \text{state})(?w1 := \text{Some clause}) \)
unfolding \( \text{swapWatches-def} \)
unfolding \( \text{setReason-def} \)
by auto
moreover
hence \( \forall (\ c::\text{nat} \). \ c \in \text{set WI}' \longrightarrow \text{Some literal} = (\text{getWatch1} \ ?\text{state}''' \ c) \ \lor \text{Some literal} = (\text{getWatch2} \ ?\text{state}''' \ c) \)
using \( \text{Cons}(4) \)
using \( \text{clause} \notin \text{set WI}' \)
using \( \text{swapWatchesEffect}[\text{of} \ \text{clause} \ \text{state}] \)
unfolding \( \text{setReason-def} \)
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'))):
    using (?w1 el (nth (getF state) clause));
    using (?w2 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 \swapWatches-def
  by auto
ultimately
  have \InvariantGetReasonIsReason (getReason \?state0) (getF \?state0)
    (Q \cup (set (getQ \?state0) \set (getQ state)))
  using \InvariantGetReasonIsReason Q Subset\{Q \cup (set (getQ \?state0) \set (getQ state))\}
  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 \{\neg literalTrue \?w1 (elements (getM \?state'))\}
  using None
  using \{\neg literalFalse \?w1 (elements (getM \?state'))\}
  using \{uniq Wl'\}
    by (simp add: Let-def)
ultimately
  show \?thesis
    by simp
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
  have \?w2 = literal
    using \{0 \leq clause \land clause < length (getF state)\}
  using \{getWatch1 \?state' clause = Some \?w1\}
  using \{getWatch2 \?state' clause = Some \?w2\}
  using Cons(4)
  using False
    by simp
hence literalFalse \?w2 (elements (getM state))
using \texttt{Cons(6)}
by simp

from \texttt{InvariantWatchesEl} \((\text{getF state}) \ (\text{getWatch1 state}) \ (\text{getWatch2 state})\)
have \(?w1\) el \((\text{nth} \ (\text{getF state}) \ \text{clause})\) \(?w2\) 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 \((0 \leq \text{clause} \land \text{clause} < \text{length} \ (\text{getF state})\)

unfolding InvariantWatchesEl-def
unfolding swapWatches-def
by auto

show \(?\text{thesis}\)
proof (cases \text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}')))\)
case True
thus \(?\text{thesis}\)
using \texttt{Cons(1)[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 \((\text{getWatch1} \ ?\text{state} \ \text{clause} = \text{Some} \ ?w1)\)
using \((\text{getWatch2} \ ?\text{state} \ \text{clause} = \text{Some} \ ?w2)\)
using \((\text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}')))\)
using \(\texttt{uniq Wl'}\)
by \texttt{(simp add:Let-def)}

next

case False
show \(?\text{thesis}\)
proof (cases \text{getNonWatchedUnfalsifiedLiteral} \ (\text{nth} \ (\text{getF ?state}')) \ \text{clause} \ ?w1 \ ?w2 \ (\text{getM ?state'}))
case \(\text{Some} \ l'\)
hence \(?w1\) el \((\text{nth} \ (\text{getF ?state}')) \ \text{clause}\)
using \(\text{getNonWatchedUnfalsifiedLiteralSomeCharacterization}\)
by simp

let \(?\text{state''} = \text{setWatch2} \ \text{clause} \ l' \ ?\text{state}'\)

from \texttt{Cons(2)}
have \(\text{InvariantWatchesEl} \ (\text{getF ?state''}) \ (\text{getWatch1 ?state''})\)
(\texttt{getWatch2 ?state''})
using \(\text{nth} \ (\text{getF ?state''}) \ \text{clause}\)
unfolding InvariantWatchesEl-def
unfolding setWatch2-def
by auto
moreover
have \(\forall \ c. \ c \in \text{set Wl'} \rightarrow \text{Some literal} = \text{getWatch1 ?state''} \ c \)
\(\lor \ \text{Some literal} = \text{getWatch2 \ ?state''} \ c\)
using \texttt{Cons(4)}
using ⟨clause \notin 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 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 ⟨uniq WL⟩

using Some 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, getConflictClause := 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 \notin set WL'⟩

unfolding setWatch2-def by simp

moreover have getM ?'state'' = getM state ∧
  getF ?'state'' = getF state ∧

480
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 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])))
let ?state0 = notifyWatches-loop literal Wl' (clause # newWl)
?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 Wl' → Some literal = (getWatch1 ?state" c) ∨ Some literal = (getWatch2 ?state" c)
  using Cons(4)
  using (clause ∉ set Wl')
  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'))):
using \( \langle ?w1 \text{ el (nth (getF state) clause) \rangle} \)

using \( \langle ?w2 \text{ el (nth (getF state) clause) \rangle} \)

using \( \langle \neg \text{ literalTrue ?w1 (elements (getM ?state'))} \rangle \)

using \( \langle \neg \text{ literalFalse ?w1 (elements (getM ?state'))} \rangle \)

using \( \langle \text{ literalFalse ?w2 (elements (getM state))} \rangle \)

unfolding isUnitClause-def by auto

hence InvariantGetReasonIsReason \((\text{getReason ?state'}) (\text{getF ?state'}) (Q \cup \{?w1\})\)

using Cons(7)

using \((\text{getM ?state''} = \text{getM state})\)

using \((\text{getF ?state''} = \text{getF state})\)

using \((\text{getQ ?state''} = \text{if ?w1 el (getQ state) then (getQ state) else (getQ state) \&@[?w1])})\)

using \((\text{getReason ?state''} = (\text{getReason state})(?w1 := \text{Some clause})\)

using \((0 \leq \text{ clause} \land \text{ clause} < \text{ length (getF state)})\)

using \((\neg \text{ literalTrue ?w1 (elements (getM ?state'))})\)

using \((\text{isUnitClause (nth (getF state) clause) ?w1 (elements (getM state)))})\)

unfolding InvariantGetReasonIsReason-def by auto

moreover have \((\lambda a. \text{ if a = ?w1 then Some clause else getReason state a}) = \text{getReason ?state''})\)

unfolding setReason-def by (auto simp add: fun-upd-def)

ultimately have InvariantGetReasonIsReason \((\text{getReason ?state0}) (\text{getF ?state0}) (\text{getM ?state0}) (Q \cup \{?w1\})\)

using Cons(7)

using \((\text{getM ?state''} = \text{getM state})\)

using \((\text{getF ?state''} = \text{getF state})\)

using \((\text{getQ ?state''} = \text{if ?w1 el (getQ state) then (getQ state) else (getQ state) \&@[?w1])}\)

using \((\text{getReason ?state''} = (\text{getReason state})(?w1 := \text{Some clause})\)

using \((0 \leq \text{ clause} \land \text{ clause} < \text{ length (getF state)})\)

by auto

ultimately have InvariantGetReasonIsReason \((\text{getReason ?state0}) (\text{getF ?state0}) (\text{getM ?state0}) (Q \cup \{?w1\})\)

using \((\text{getM ?state''} = \text{getM state})\)

using \((\text{getF ?state''} = \text{getF state})\)

using \((\text{getQ ?state''} = \text{if ?w1 el (getQ state) then (getQ state) else (getQ state) \&@[?w1])}\)

by auto

ultimately have InvariantGetReasonIsReason \((\text{getReason ?state0}) (\text{getF ?state0}) (\text{getM ?state0}) (Q \cup \{?w1\})\)

using \((\text{getM ?state''} = \text{getM state})\)

using \((\text{getF ?state''} = \text{getF state})\)

using \((\text{getQ ?state''} = \text{if ?w1 el (getQ state) then (getQ state) else (getQ state) \&@[?w1])}\)

by auto

ultimately have InvariantGetReasonIsReason \((\text{getReason ?state0}) (\text{getF ?state0}) (\text{getM ?state0}) (Q \cup \{?w1\})\)

using \((\text{getM ?state''} = \text{getM state})\)

using \((\text{getF ?state''} = \text{getF state})\)

using \((\text{getQ ?state''} = \text{if ?w1 el (getQ state) then (getQ state) else (getQ state) \&@[?w1])}\)

getReason ?state0 getF ?state0 getM ?state0
by simp
moreover
have notifyWatches-loop literal (clause ≠ \text{Wl}) new\text{Wl} state
= ?state0
using (getWatch1 ?state' clause = Some ?w1)
using (getWatch2 ?state' clause = Some ?w2)
using (\neg \text{Some literal} = \text{getWatch1 state clause})
using None
using (\neg \text{literalTrue} ?w1 (\text{elements} (\text{getM} ?state')))
using (uniq \text{Wl})
by (simp add: Let-def)
ultimately
show ?thesis
by simp
qed
qed
qed
qed

lemma assertLiteralEffect:
fixes state::\text{State} and l::\text{Literal} and d::\text{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 gets [l, d])] getWatchList (state gets [l, d])]
(opposite l]
by (auto simp add: Let-def)

lemma WatchInvariantsAfterAssertLiteral:
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
InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)
shows
let state' = (assertLiteral literal decision state) in
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')
using assms
unfolding assertLiteral-def
unfolding notifyWatches-def
using InvariantWatchesElNotifyWatchesLoop[of state][getM := getM state @ [(literal, decision)]]] getWatchList state (opposite literal) opposite literal []
using InvariantWatchesDifferNotifyWatchesLoop[of state][getM := getM state @ [(literal, decision)]]] getWatchList state (opposite literal) opposite literal []
using InvariantWatchListsContainOnlyClausesFromFNotifyWatchesLoop[of state][getM := getM state @ [(literal, decision)]]] getWatchList state (opposite literal) opposite literal []
using InvariantWatchListsCharacterizationNotifyWatchesLoop[of state][getM := getM state @ [(literal, decision)]]] getWatchList (state[getM := getM state @ [(literal, decision)]]]) (opposite literal) opposite literal []
unfolding InvariantWatchListsContainOnlyClausesFromF-def
unfolding InvariantWatchListsCharacterization-def
unfolding InvariantWatchListsUniq-def
by (auto simp add: Let-def)

lemma InvariantWatchCharacterizationAfterAssertLiteral:
assumes
InvariantConsistent ((getM state @ [(literal, decision)])) and
InvariantUniq ((getM state @ [(literal, decision)])) and
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)
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 getWatch-
           List ?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)
      watchCharacterizationCondition w2 w1 (getM state) (getF
        state ! c)
      using ⟨InvariantWatchCharacterization (getF state) (getWatch1
            state) (getWatch2 state) (getM state)⟩
      using ⟨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 \texttt{assertLiteralEffect[of state literal decision]}

by simp

with \texttt{(watchCharacterizationCondition \texttt{w1 w2 (getM state)}}

\texttt{(getF state \texttt{c})} -

\texttt{have} \((\exists \ l. \ l \in ((\texttt{getF state}) \texttt{c}) \land \texttt{literalTrue l (elements (getM state))} \land \texttt{elementLevel l (getM state)} \leq \texttt{elementLevel (opposite w1) (getM state)}) \lor \)

\texttt{(\forall \ l. \ l \in ((\texttt{getF state}) \texttt{c}) \land l \neq w1 \land l \neq w2 \quad \texttt{literalFalse l (elements (getM state))} \land \texttt{elementLevel (opposite l) (getM state)} \leq \texttt{elementLevel (opposite w1) (getM state)}) (is \texttt{?a state} \lor \texttt{?b state})}

\texttt{unfolding watchCharacterizationCondition-def}

using \texttt{asms}

using \texttt{assertLiteralEffect[of state literal decision]}

using \texttt{(w1 \neq opposite literal)}

by simp

\texttt{have \texttt{?a ?state'} \lor \texttt{?b ?state'}}

\texttt{proof (cases \texttt{?b state})}

\texttt{case True}

\texttt{show \texttt{?thesis}}

\texttt{proof-}

\{

fix \texttt{l}

assume \texttt{l \in (nth (\texttt{getF ?state'}) \texttt{c}) \land l \neq w1 \land l \neq w2}

\texttt{have \texttt{literalFalse l (elements (getM ?state'))} \land \texttt{elementLevel (opposite l) (getM ?state')} \leq \texttt{elementLevel (opposite w1) (getM ?state')}}

\texttt{proof-}

from \texttt{True} \texttt{(\texttt{l \in (nth (\texttt{getF ?state'}) \texttt{c}) \land l \neq w1 \land l \neq w2)}, \texttt{has \texttt{literalFalse l (elements (getM state))} \land \texttt{elementLevel (opposite l) (getM state)} \leq \texttt{elementLevel (opposite w1) (getM state)}}

\texttt{using \texttt{asms}}

\texttt{using \texttt{assertLiteralEffect[of state literal decision]}}

by auto

\texttt{thus \texttt{?thesis}}

\texttt{using \texttt{(\texttt{literalFalse w1 (elements (getM state))})} \land \texttt{elementLevelAppend[of opposite w1 getM state [[(literal, decision)]]}}

\texttt{using \texttt{elementLevelAppend[of opposite l getM state [[(literal, decision)]]}}

\texttt{using \texttt{asms}}

\texttt{using \texttt{assertLiteralEffect[of state literal decision]}}

by auto

\texttt{qed}

\texttt{thus \texttt{?thesis}}

\texttt{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

moreover

have watchCharacterizationCondition w2 w1 (getM ?state') ((getF ?state') ! c)

proof−

{
assume literalFalse w2 (elements (getM ?state'))
with (w2 \neq opposite literal)
have literalFalse w3 (elements (getM state))
using assms
using assertLiteralEffect[of state literal decision]
by simp
with \langle watchCharacterizationCondition w2 w1 (getM state) (getF state \{c\}) \rangle
have (\exists l. l el ((getF state) \{c\}) \wedge literalTrue l (elements (getM state)))
\wedge elementLevel l (getM state) \leq elementLevel (opposite w2 (getM state))
\lor
(\forall l. l el ((getF state) \{c\}) \wedge l \neq w2 \wedge l \neq w1 \rightarrow
literalFalse l (elements (getM state)) \wedge
elementLevel (opposite l) (getM state) \leq elementLevel (opposite w2) (getM state))
(is ?a state \lor ?b state)
unfolding watchCharacterizationCondition-def
using assms
using assertLiteralEffect[of state literal decision]
using (w2 \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')) \wedge
    elementLevel (opposite l) (getM ?state') \leq elementLevel (opposite w2) (getM ?state')
  proof-
    from True \langle l el (nth (getF ?state') c) : l \neq w1 : l \neq w2 \rangle
    have literalFalse l (elements (getM state))
    elementLevel (opposite l) (getM state) \leq elementLevel (opposite w2) (getM state)
    using assms
    using assertLiteralEffect[of state literal decision]
    by auto
    thus ?thesis
    using \langle literalFalse w2 (elements (getM state))\rangle
    using elementLevelAppend[of opposite w2 getM state [[(literal, decision)]]]
  using elementLevelAppend[of opposite l getM state [[(literal, decision)]]]
  using assms
  using assertLiteralEffect[of state literal decision]
 489
by auto
qed
}
thus ?thesis
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 w2) (getM state)
by auto

from (w2 ≠ opposite literal)
(literalFalse w2 (elements (getM ?state')))
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 (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 w2) ((getM state) @ [(literal, decision)])
using elementLevel l (getM state) ≤ elementLevel (opposite w2) (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
ultimately
show \( \text{thesis} \)
by simp
qed
\}
thus \( \text{thesis} \)
unfolding InvariantWatchCharacterization-def
by (simp add: Let-def)
qed

lemma assertLiteralConflictFlagEffect:
assumes
\begin{align*}
\text{InvariantConsistent} \ (\text{getM state} @ [[\text{literal}, \text{decision}]]) \\
\text{InvariantUniq} \ (\text{getM state} @ [[\text{literal}, \text{decision}]]) \\
\text{InvariantWatchListsContainOnlyClausesFromF} \ (\text{getWatchList state} (\text{getF state})) \\
\text{InvariantWatchListsUniq} \ (\text{getWatchList state}) \\
\text{InvariantWatchListsCharacterization} \ (\text{getWatchList state} (\text{getWatch1 state}) (\text{getWatch2 state})) \\
\text{InvariantWatchesEl} \ (\text{getF state}) (\text{getWatch1 state}) (\text{getWatch2 state}) \\
\text{InvariantWatchCharacterization} \ (\text{getF state}) (\text{getWatch1 state}) (\text{getWatch2 state}) (\text{getM state})
\end{align*}
shows
\begin{align*}
\text{let} \ state' = \text{assertLiteral} \ \text{literal} \ \text{decision} \ \text{state} \text{ in} \\
\text{getConflictFlag state} = \text{getConflictFlag state} \lor \\
(\exists \text{ clause}. \ \text{clause} \in \text{set} \ (\text{getWatchList state} \ \text{opposite literal} (\text{getF state})) \land \\
\text{clauseFalse} \ \text{clause} \ ((\text{elements} \ \text{getM state}) @ [\text{literal}])))
\end{align*}

\begin{proof}
\begin{align*}
\text{let} \ ?state = \text{state}([\text{getM} := \text{getM state} @ [[\text{literal}, \text{decision}]])] \\
\text{let} \ ?state' = \text{assertLiteral} \ \text{literal} \ \text{decision} \ \text{state} \text{ in} \\
\text{have} \ \text{getConflictFlag} \ ?state' = \text{getConflictFlag} \ ?state \lor \\
(\exists \text{ clause}. \ \text{clause} \in \text{set} \ (\text{getWatchList} \ ?state \ (\text{opposite literal}) \land \\
\text{clauseFalse} \ (\text{nth} \ (\text{getF} \ ?state) \ \text{clause}) \ (\text{elements} \ (\text{getM} \ ?state)))))
\end{align*}

using NotifyWatchesLoopConflictFlagEffect[of \ ?state \ getWatchList ?state \ (\text{opposite literal}) \ opposite literal []]
using \text{InvariantConsistent} \ (\text{getM state} @ [[\text{literal}, \text{decision}]])
using \text{InvariantWatchListsContainOnlyClausesFromF} \ (\text{getWatchList state} (\text{getF state}))
using \text{InvariantWatchListsUniq} \ (\text{getWatchList state})
using \text{InvariantWatchListsCharacterization} \ (\text{getWatchList state} (\text{getWatch1 state}) (\text{getWatch2 state}))
using \text{InvariantWatchesEl} \ (\text{getF state}) (\text{getWatch1 state}) (\text{getWatch2 state})
\end{proof}
unfolding InvariantWatchListsUniq-def
unfolding InvariantWatchListsCharacterization-def
unfolding InvariantWatchListsContainOnlyClausesFromF-def
unfolding assertLiteral-def
unfolding notifyWatches-def
by (simp add: Let-def)

moreover
have \( (\exists \text{ clause} . \text{ clause} \in \text{ set (getWatchList ?state (opposite literal))} \land \text{ clauseFalse (n} t\text{h (getF ?state) clause) (elements (getM ?state}}) =
\quad (\exists \text{ clause} . \text{ clause el (getF state) \land opposite literal el clause \land clauseFalse clause ((elements (getM state)) @ [literal])})
\)
(is \( ?lhs = ?rhs \))

proof
  assume ?lhs
  then obtain clause
    where \( \text{ clause \in \text{ set (getWatchList ?state (opposite literal))} \land clauseFalse (n} t\text{h (getF ?state) clause) (elements (getM ?state}})
  by auto

  have \( \text{getWatch1 ?state clause = Some (opposite literal)} \lor \text{getWatch2 ?state clause = Some (opposite literal)} \land \text{ clause < length (getF ?state)} \land \exists w1 w2. \text{getWatch1 ?state clause = Some w1 \land getWatch2 ?state clause = Some w2} \land w1 \text{ el (n} t\text{h (getF ?state) clause) \land w2 \text{ el (n} t\text{h (getF ?state) clause))\land using assms}\)
  unfolding InvariantWatchListsContainOnlyClausesFromF-def
  unfolding InvariantWatchesEl-def
  unfolding InvariantWatchListsCharacterization-def
  by auto
  hence \( (n} t\text{h (getF ?state) clause) el (getF ?state) \land \text{ opposite literal el (n} t\text{h (getF ?state) clause)}\land using nth-mem[of clause getF ?state]

  by auto
  thus \( ?rhs \)

  using \( (\text{clauseFalse (n} t\text{h (getF ?state) clause) (elements (getM ?state}}))\)

  by auto

next
  assume ?rhs
  then obtain clause
    where \( \text{ clause el (getF ?state) \land opposite literal el clause \land clauseFalse clause ((elements (getM state)) @ [literal])}\)

  by auto
then obtain \( ci \)
  where \( \text{clause} = \text{(nth (getF ?state) ci) ci < length (getF ?state)} \)
  by (auto simp add: in-set-conv-nth)
moreover
from (\( ci < \text{length (getF ?state)} \))
obtain \( w1 \ w2 \)
  where \( \text{getWatch1 state ci = Some w1 getWatch2 state ci = Some w2} \)
  \( w1 \ el \text{(nth (getF state) ci)} \)
  \( w2 \ el \text{(nth (getF state) ci)} \)
  using assms
  unfolding InvariantWatchesEl-def
  by auto
have \( \text{getWatch1 state ci = Some (opposite literal) \lor getWatch2 state ci = Some (opposite literal)} \)
proof−
  { assume \( \neg \text{thesis} \)
    with \( \text{clauseFalse clause ((elements (getM state)) @ [literal])} \)
    \( \text{clause = (nth (getF ?state) ci)} \)
    \( \text{getWatch1 state ci = Some w1 getWatch2 state ci = Some w2} \)
    \( w1 \ el \text{(nth (getF state) ci)} \)
    \( w2 \ el \text{(nth (getF state) ci)} \)
    have \( \text{literalFalse w1 (elements (getM state)) literalFalse w2 (elements (getM state))} \)
    by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
  }

from \( \text{InvariantConsistent ((getM state) @ [(literal, decision)])} \)
\( \text{clauseFalse clause ((elements (getM state)) @ [literal])} \)
  have \( \neg (\exists \ l. \ l \ el \text{clause \land literalTrue l (elements (getM state))}) \)
  unfolding InvariantConsistent-def
  by (auto simp add: inconsistentCharacterization clauseFalseIfAllLiteralsAreFalse)

from \( \text{InvariantUniq ((getM state) @ [(literal, decision)])} \)
  have \( \neg \text{literalTrue literal (elements (getM state))} \)
  unfolding InvariantUniq-def
  by (auto simp add: uniqAppendIff)

from \( \text{InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)} \)
  \( \text{literalFalse w1 (elements (getM state)) literalFalse w2 (elements (getM state))} \)
  \( \neg (\exists \ l. \ l \ el \text{clause \land literalTrue l (elements (getM state))}) \)
  \( \text{getWatch1 state ci = Some w1 THEN sym} \)
  \( \text{getWatch2 state ci = Some w2 THEN sym} \)
  \( ci < \text{length (getF ?state)} \)

493
\langle \text{clause} = (\text{nth} (\text{getF} \ ?\text{state}) \text{ci}) \rangle

\text{have } \forall \ l, l' \in \text{clause} \land l \neq w1 \land l \neq w2 \rightarrow \text{literalFalse} \ l

\text{(elements} (\text{getM} \ \text{state}))

\text{unfolding } \text{InvariantWatchCharacterization-def}

\text{unfolding } \text{watchCharacterizationCondition-def}

\text{by auto}

\text{hence } \text{literalTrue} \ \text{literal} (\text{elements} (\text{getM} \ \text{state}))

\text{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}))

\text{using } \langle \text{opposite literal} \ \text{el} \ \text{clause} \rangle

\text{using } (\text{getWatch1} \ \text{state} \ \text{ci} = \text{Some} \ w1)

\text{using } (\text{getWatch2} \ \text{state} \ \text{ci} = \text{Some} \ w2)

\text{by auto}

\text{with } \langle \neg \text{literalTrue} \ \text{literal} (\text{elements} (\text{getM} \ \text{state})) \rangle

\text{have False}

\text{by simp}

\}

\text{thus } ?\text{thesis}

\text{by auto}

\text{qed}

\text{ultimately}

\text{show } ?\text{lhs}

\text{using } \text{assms}

\text{using } \langle \text{clauseFalse} \ \text{clause} \ (\text{elements} (\text{getM} \ \text{state})) @ [\text{literal}] \rangle

\text{unfolding } \text{InvariantWatchListsCharacterization-def}

\text{by force}

\text{qed}

\text{ultimately}

\text{show } ?\text{thesis}

\text{by auto}

\text{qed}

\text{lemma } \text{InvariantConflictFlagCharacterizationAfterAssertLiteral:}

\text{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})

\text{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{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}}
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 @ [(literal, decision)]]
let ?state' = assertLiteral literal decision state

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)]
using (InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state))
using (InvariantConsistent ((getM state) @ [(literal, 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 \rightarrow 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 \rightarrow getConflictFlag ?state');

using assms

unfolding InvariantConflictFlagCharacterization-def
by (auto simp add: Let-def formulaFalseAppendValuation)

next
case False
hence \( \neg \text{formulaFalse} \left( \text{getF} \ \text{state} \right) \left( \text{elements} \left( \text{getM} \ \text{state} \right) \right) \)

using \( \text{InvariantConflictFlagCharacterization} \left( \text{getConflictFlag} \ \text{state} \right) \left( \text{getF} \ \text{state} \right) \left( \text{getM} \ \text{state} \right) \)

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

by simp

have \( \star \star : \forall \ \text{clause}. \ \text{clause} \notin \text{set} \left( \text{getWatchList} \ ?\text{state} \left( \text{opposite literal} \right) \right) \land \)

\begin{align*}
0 \leq \text{clause} & \land \text{clause} < \text{length} \left( \text{getF} \ ?\text{state} \right) \rightarrow \neg \text{clauseFalse} \left( \text{nth} \left( \text{getF} \ ?\text{state} \right) \ \text{clause} \right) \left( \text{elements} \ \text{getM} \ ?\text{state} \right) \\
\end{align*}

proof

\{ fix \ \text{clause} \\
assume \ \text{clause} \notin \text{set} \left( \text{getWatchList} \ ?\text{state} \left( \text{opposite literal} \right) \right) \\
and \ 
\begin{align*}
0 \leq \text{clause} & \land \text{clause} < \text{length} \left( \text{getF} \ ?\text{state} \right) \\
\end{align*}

from \( \begin{align*}
0 \leq \text{clause} & \land \text{clause} < \text{length} \left( \text{getF} \ ?\text{state} \right) \\
\end{align*} \)

obtain \( w1 :: \text{Literal} \ \text{and} \ w2 :: \text{Literal} \)

where \( \begin{align*}
\text{getWatch1} \ \text{state} \ \text{clause} = \text{Some} \ w1 & \ \text{and} \\
\text{getWatch2} \ ?\text{state} \ \text{clause} = \text{Some} \ w2 & \ \text{and} \\
w1 \ \text{el} \ \left( \text{nth} \left( \text{getF} \ ?\text{state} \right) \ \text{clause} \right) & \ \text{and} \\
w2 \ \text{el} \ \left( \text{nth} \left( \text{getF} \ ?\text{state} \right) \ \text{clause} \right) \\
\end{align*} \)

using \( \text{InvariantWatchesEl} \left( \text{getF} \ \text{state} \right) \left( \text{getWatch1} \ \text{state} \right) \left( \text{getWatch2} \ \text{state} \right) \)

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

by auto

have \( \neg \text{clauseFalse} \left( \text{nth} \left( \text{getF} \ ?\text{state} \right) \ \text{clause} \right) \left( \text{elements} \left( \text{getM} \ ?\text{state} \right) \right) \)

proof

from \( \begin{align*}
\text{clause} \notin \text{set} \left( \text{getWatchList} \ ?\text{state} \left( \text{opposite literal} \right) \right) \\
\end{align*} \)

have \( w1 \neq \text{opposite literal} \ \text{and} \ w2 \neq \text{opposite literal} \)

using \( \text{InvariantWatchListsCharacterization} \left( \text{getWatchList} \ \text{state} \right) \left( \text{getWatch1} \ \text{state} \right) \left( \text{getWatch2} \ \text{state} \right) \)

using \( \begin{align*}
\text{getWatch1} \ ?\text{state} \ \text{clause} = \text{Some} \ w1 & \ \text{and} \\
\text{getWatch2} \ ?\text{state} \ \text{clause} = \text{Some} \ w2 \\
\end{align*} \)

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

by auto

from \( \neg \text{formulaFalse} \left( \text{getF} \ \text{state} \right) \left( \text{elements} \left( \text{getM} \ \text{state} \right) \right) \)

have \( \neg \text{clauseFalse} \left( \text{nth} \left( \text{getF} \ ?\text{state} \right) \ \text{clause} \right) \left( \text{elements} \left( \text{getM} \ \text{state} \right) \right) \)

using \( \begin{align*}
0 \leq \text{clause} & \land \text{clause} < \text{length} \left( \text{getF} \ ?\text{state} \right) \\
\end{align*} \)

by \( \text{(simp add: formulaFalseIffContainsFalseClause)} \)
show \(?\text{thesis}\)
proof (cases \text{literalFalse } w1 \ (\text{elements } (\text{getM state})) \lor \text{literalFalse } w2 \ (\text{elements } (\text{getM state})))
case True

with (\text{InvariantWatchCharacterization } (\text{getF state}) \ (\text{getWatch1 state}) \ (\text{getWatch2 state}) \ (\text{getM state}))
have \(\exists \ l. \ l \in (\text{nth } (\text{getF state}) \ \text{clause}) \land \text{literalTrue } l \ (\text{elements } (\text{getM state}))) \lor\\\ (\forall \ l. \ l \in (\text{nth } (\text{getF state}) \ \text{clause}) \land (l \neq w1 \land l \neq w2) \rightarrow \text{literalFalse } l \ (\text{elements } (\text{getM state})))

using \text{getWatch1 } ?\text{state} \ \text{clause} = \text{Some } w1 \ [\text{THEN } \text{sym}]
using \text{getWatch2 } ?\text{state} \ \text{clause} = \text{Some } w2 \ [\text{THEN } \text{sym}]
using \(0 \leq \text{clause} \land \text{clause} < \text{length } (\text{getF } ?\text{state})\)
unfolding \text{InvariantWatchCharacterization-def}
unfolding \text{watchCharacterizationCondition-def}
by auto
thus \(?\text{thesis}\)
proof (cases \forall \ l. \ l \in (\text{nth } (\text{getF state}) \ \text{clause}) \land (l \neq w1 \land l \neq w2) \rightarrow \text{literalFalse } l \ (\text{elements } (\text{getM state})))
case True
  have \(\neg \text{literalFalse } w1 \ (\text{elements } (\text{getM state})) \lor \neg \text{literalFalse } w2 \ (\text{elements } (\text{getM state})))
  proof
   from \(\neg \text{clauseFalse } (\text{nth } (\text{getF } ?\text{state}) \ \text{clause}) \ (\text{elements } (\text{getM state})))
   obtain l::Literal
   where l \in (\text{nth } (\text{getF } ?\text{state}) \ \text{clause}) \land \neg \text{literalFalse } l \ (\text{elements } (\text{getM state}))
   by (auto simp add: \text{clauseFalseIffAllLiteralsAreFalse})
   with True
   show \(?\text{thesis}\)
   by auto
qed
  hence \(\neg \text{literalFalse } w1 \ (\text{elements } (\text{getM } ?\text{state})) \lor \neg \text{literalFalse } w2 \ (\text{elements } (\text{getM } ?\text{state}))
  using \(w1 \neq \text{opposite literal } \land w2 \neq \text{opposite literal})
  by auto
  thus \(?\text{thesis}\)
  using \(w1 \in (\text{nth } (\text{getF } ?\text{state}) \ \text{clause}) \land w2 \in (\text{nth } (\text{getF } ?\text{state}) \ \text{clause}))
  by (auto simp add: \text{clauseFalseIffAllLiteralsAreFalse})
next
case False
then obtain l::Literal

497
where \( l \in \text{elts} (\text{getM state}) \) and \( \text{literalTrue} \ l \)
using $\$
by auto
thus ?thesis
using \( \langle \text{InvariantConsistent} ((\text{getM state}) \mapsto [(\text{literal, decision})]) \rangle \)
unfolding \( \text{InvariantConsistent-def} \)
by (auto simp add: \( \text{clauseFalseIffAllLiteralsAreFalse} \)
\( \text{inconsistentCharacterization} \))
qed
next
case False
thus ?thesis
using \( \langle w \in \text{elts} (\text{getF ?state}) \rangle \) \( \langle w \neq \text{opposite literal} \rangle \)
by (auto simp add: \( \text{clauseFalseIffAllLiteralsAreFalse} \))
qed
qed

show ?thesis
proof (cases \( \text{getConflictFlag ?state} \))
case True
from \( \langle \neg \text{getConflictFlag state} \rangle \) \( \langle \text{getConflictFlag ?state} \rangle \)
obtain clause :: nat
where
clause \( \in \) set (\( \text{getWatchList ?state} \) (opposite literal)) and
clauseFalse (\( \text{nth} (\text{getF ?state}) \) clause) (\( \text{elts} (\text{getM ?state}) \))
using $\$
by auto
from \( \langle \text{InvariantWatchListsContainOnlyClausesFromF} ((\text{getWatchList state}) \ (\text{getF state}) \ (\text{getF ?state})) \) \( \langle \text{clause} \in \text{set} (\text{getWatchList ?state} \ (\text{opposite literal})) \rangle \) \( \langle \text{clauseFalse} (\text{nth} (\text{getF ?state}) \) \text{clause}) \ (\text{elts} (\text{getM ?state})) \rangle \)
using $\$
by simp
\( \langle \text{clauseFalse} (\text{nth} (\text{getF ?state}) \) \text{clause}) \ (\text{elts} (\text{getM ?state})) \rangle \)
have \( \langle \text{formulaFalse} (\text{getF ?state}) \ (\text{elts} (\text{getM ?state})) \rangle \) \( \langle \text{Get} \mapsto \langle \text{Get} \mapsto \text{containsFalseClause} \rangle \rangle \)
by (auto simp add: \( \text{Let-def} \) \( \text{formulaFalseIffContainsFalseClause} \))
thus ?thesis
using \( \langle \neg \text{getConflictFlag state} \rangle \) \( \langle \text{getConflictFlag ?state} \rangle \)
unfolding \( \text{InvariantConflictFlagCharacterization-def} \)
using \( \text{assms} \)
using assertLiteralEffect[of state literal decision]
by (simp add: Let-def)

next

case False

hence \( \forall \) clause::nat. clause \( \in \) set (getWatchList ?state (opposite literal)) \( \implies \)

\( \neg \) clauseFalse (nth (getF ?state) clause) (elements (getM ?state))

using *
by auto

with **

have \( \forall \) clause. \( 0 \leq \) clause \( \land \) clause < length (getF ?state) \( \implies \)

\( \neg \) clauseFalse (nth (getF ?state) clause) (elements (getM ?state))

by (auto simp add: set-conv-nth formulaFalseIffContainsFalse-Clause)

thus \( ?\)thesis

using \( \neg \) getConflictFlag state \( \iff \) getConflictFlag ?state'

using assms

unfolding InvariantConflictFlagCharacterization-def

by (auto simp add: Let-def assertLiteralEffect)

qed

qed

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

using InvariantConflictClauseCharacterizationAfterNotifyWatches[of ?state0 getM state opposite literal decision
getWatchList ?state0 (opposite literal) ]]

unfolding assertLiteral-def
unfolding notifyWatches-def
unfolding InvariantWatchListsUniq-def
unfolding InvariantWatchListsContainOnlyClausesFromF-def
unfolding InvariantWatchListsCharacterization-def
unfolding InvariantConflictClauseCharacterization-def
by (simp add: Let-def clauseFalseAppendValuation)

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)
  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 uc ∃ uc. uc el (getF state) ∧
              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
fix \( ul \)
assume \( ul \in \text{?ulSet} \)
then obtain \( uc \)
where \( uc \in (\text{getF state}) \) opposite literal \( el \) \( uc \) isUnitClause \( uc \) ul ((\text{elements} (\text{getM state})) @ [\text{literal}])
  by auto
then obtain \( uci \)
where \( uc = (\text{nth} (\text{getF state}) \ uci)\) \( uci < \text{length} (\text{getF state}) \)
  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 \( \neg \ ?\text{thesis} \)
  
  from \( \text{InvariantWatchesEl (getF state) (getWatch1 state)} \)
  \( \text{(getWatch2 state)} \)
  obtain \( wl1 \) \( wl2 \)
  where \( ?w1 = \text{Some} \) \( wl1 \) \( ?w2 = \text{Some} \) \( wl2 \) \( el \) \( \text{(getF state ! uci)} \)
    unfolding \( \text{InvariantWatchesEl-def} \)
    using \( \text{uci} < \text{length} (\text{getF state}) \)
    by force

  with \( \text{InvariantWatchCharacterization (getF state) (getWatch1 state)} \)
  \( \text{(getWatch2 state)} \) \( \text{(getM state)} \)
  have watchCharacterizationCondition \( wl1 \) \( wl2 \) \( \text{(getM state)} \)
  (\text{getF state ! uci})
    watchCharacterizationCondition \( wl2 \) \( wl1 \) \( \text{(getM state)} \) \( \text{(getF state ! uci)} \)
    using \( \text{uci} < \text{length} (\text{getF state}) \)
    unfolding \( \text{InvariantWatchCharacterization-def} \)
    by auto
  
  from \( \text{isUnitClause uc ul ((\text{elements} (\text{getM state})) @ [\text{literal}])} \)
  have \( \neg (\exists l. l \in uc \land (\text{literalTrue} l ((\text{elements} (\text{getM state})) @ [\text{literal}]))) \)
    using containsTrueNotUnit
    using \( \text{InvariantConsistent ((getM state) @ [(\text{literal, decision})]}} \)
    unfolding \( \text{InvariantConsistent-def} \)
    by auto
  
  from \( \text{InvariantUniq ((getM state) @ [(\text{literal, decision})]} \)
  have \( \neg \text{literal el (\text{elements} (\text{getM state}))} \)
unfolding $\text{InvariantUniq-def}$
by (simp add: uniqAppendIff)

from $\sim \neg \text{thesis}$
(\(?w1 = \text{Some } wl1\) \(\neg\))
(\(?w2 = \text{Some } wl2\)
have \(wl1 \neq \text{opposite literal } wl2 \neq \text{opposite literal }
by auto

from $\text{InvariantWatchesDiffer \ (getF state) \ (getWatch1 state) \ (getWatch2 state)\rangle}$
(\(?wl1 \neq \text{wl2} \rangle$)
using (\(?w1 = \text{Some } wl1\) \(\neg\))
(\(?w2 = \text{Some } wl2\)$
unfolding $\text{InvariantWatchesDiffer-def}$
using (\(?uc < \text{length } \text{getF state}\)\rangle
by auto

have $\text{literaFalse } \text{wl1 \ (elements } \text{getM state}) \lor \text{literaFalse } \text{wl2 \ (elements } \text{getM state})$
proof (cases \(ul = \text{wl1}\) )
  case True
  with (\(?wl1 \neq \text{wl2} \rangle$)
  have \(ul \neq \text{wl2}\)
  by simp
  with (\text{isUnitClause } \text{uc } \text{ul} ((\text{elements } \text{getM state}) \circ \text{[litera]}))
  \(\text{wl2 \ neq \text{opposite literal } \text{wl2 el } \text{getF state } \text{uci})\)
  \(\text{uc = (getF state } \text{uci)}\)$
  show \(\text{thesis}\)
  unfolding $\text{isUnitClause-def}$
  by auto

next
  case False
  with (\text{isUnitClause } \text{uc } \text{ul} ((\text{elements } \text{getM state}) \circ \text{[litera]}))
  \(\text{wl1 \ neq \text{opposite literal } \text{wl1 el } \text{getF state } \text{uci})\)
  \(\text{uc = (getF state } \text{uci)}\)$
  show \(\text{thesis}\)
  unfolding $\text{isUnitClause-def}$
  by auto

qed

with (\text{watchCharacterizationCondition } \text{wl1 } \text{wl2 } \text{getM state})
(\text{getF state } \text{uci})$)
(\text{watchCharacterizationCondition } \text{wl2 } \text{wl1 } \text{getM state})
(\text{getF state } \text{uci})$)
(\(\exists \ l. \ l \text{ el uc } \land \text{literalTrue } \text{l (elements } \text{getM state}) \circ \text{[litera]}))$)
\(\text{uc = (getF state } \text{uci)}\)$
(\(?w1 = \text{Some } \text{wl1}\) \(\neg\))
(\(?w2 = \text{Some } \text{wl2}\)$
have \(\forall \ l. \ l \text{ el uc } \land \ l \neq \text{wl1 } \land \ l \neq \text{wl2 } \rightarrow \text{literaFalse } \text{l (elements } \text{getM state})\)$
unfolding \texttt{watchCharacterizationCondition-def} 
by \texttt{auto} 
with \langle \texttt{\texttt{wl}1 \neq \texttt{opposite literal}; \texttt{wl}2 \neq \texttt{opposite literal; \texttt{opposite literal \texttt{el uc}}}}\rangle 
have \texttt{literalTrue literal (elements (\texttt{getM state})}) 
by \texttt{auto} 
with \langle \texttt{\neg literal el (elements (\texttt{getM state})})\rangle 
have \texttt{False} 
by \texttt{simp} 
\} thus \texttt{?thesis} 
by \texttt{auto} 
\qed 

with \texttt{\langle \texttt{InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)}\rangle} 
have \texttt{aci \in set (getWatchList state (opposite literal))} 
unfolding \texttt{InvariantWatchListsCharacterization-def} 
by \texttt{auto} 
thus \texttt{ul \in set (getQ ?state''')} 
using \texttt{(uc el (getF state))} 
using \texttt{\langle \texttt{isUnitClause uc ul \ ((elements (\texttt{getM state}) @ [\texttt{l literal}]).)\rangle}} 
using \texttt{\langle \texttt{uc = (getF state ! uci).}\rangle} 
unfolding \texttt{assertLiteral-def} 
unfolding \texttt{notifyWatches-def} 
using \texttt{assms} 
using \texttt{\langle \texttt{NotifyWatchesLoopQEffect[of \texttt{\texttt{state'} getM state opposite}}\rangle} 
literal decision \texttt{getWatchList \texttt{\texttt{state'}} (opposite literal \texttt{]} \texttt{]} \texttt{\]} 
unfolding \texttt{InvariantWatchListsCharacterization-def} 
unfolding \texttt{InvariantWatchListsUniq-def} 
unfolding \texttt{InvariantWatchListsContainOnlyClausesFromF-def} 
by \texttt{(auto simp add: \texttt{Let-def})} 
\qed 

moreover 
have \texttt{set (getQ state) \subseteq set (getQ ?state''')} 
using \texttt{assms} 
using \texttt{\langle \texttt{assertLiteralEffect[of state literal decision]}\rangle} 
using \texttt{prefixIsSubset[of getQ state getQ ?state'']} 
by \texttt{simp} 
ultimately 
show \texttt{?thesis} 
by \texttt{(auto simp add: \texttt{Let-def})} 
\qed 

\textbf{lemma} \texttt{InvariantQCharacterizationAfterAssertLiteral}: 
\textbf{assumes} 
\texttt{InvariantConsistent ((\texttt{getM state}) @ [(\texttt{l literal, decision})])} 
\texttt{InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and} 

503
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 \( \text{state}' = (\text{assertLiteral \ literal \ decision \ state}) \) in 
InvariantQCharacterization \( (\text{getConflictFlag \ state'}) \ (\text{removeAll \ literal \ (getQ \ state')}) \ (\text{getF \ state'}) \ (\text{getM \ state'}) \) 
proof
let \( ?\text{state} = \text{state}[@ \text{getM := getM \ state} \ @ ([\text{literal}, \text{decision}])] \) 
let \( ?\text{state}' = \text{assertLiteral \ literal \ decision \ state} \) 

have \( \forall \ l. \ l \in \text{set \ (getQ \ ?\text{state}')} \rightarrow \text{set \ (getQ \ ?\text{state})} \rightarrow 
(\exists \ \text{clause}. \ \text{clause} \in \text{set \ (getWatchList \ ?\text{state} (\text{opposite \ literal})}) 
\rightarrow 
\text{(isUnitClause \ clause \ l \ (elements \ (getM \ ?\text{state})))}) \)
using NotifyWatchesLoopQEffect[\[ of \ ?\text{state} \text{getM \ state \ opposite \ literal \ decision \ getWatchList \ ?\text{state} (\text{opposite \ literal}) \]]
using \text{assms}
unfolding InvariantWatchListsUniq-def 
unfolding InvariantWatchListsCharacterization-def 
unfolding InvariantWatchListsContainOnlyClausesFromF-def 
unfolding InvariantWatchCharacterization-def 
unfolding assertLiteral-def 
unfolding notifyWatches-def 
by \( \text{(auto \ simp \ add: \ Let-def)} \) 

have \( \forall \ \text{clause}. \ \text{clause} \in \text{set \ (getWatchList \ ?\text{state} (\text{opposite \ literal})}) \rightarrow 
(\forall \ l. \ \text{isUnitClause \ nth \ (getF \ ?\text{state}) \ l \ (elements \ (getM \ ?\text{state}))) \rightarrow 
\text{l \in \set \ (getQ \ ?\text{state}')}) \)
using NotifyWatchesLoopQEffect[\[ of \ ?\text{state} \text{getM \ state \ opposite \ literal \ decision \ getWatchList \ ?\text{state} (\text{opposite \ literal}) \]]
using \text{assms}
unfolding InvariantWatchListsUniq-def 
unfolding InvariantWatchListsCharacterization-def 
unfolding InvariantWatchListsContainOnlyClausesFromF-def 
unfolding InvariantWatchCharacterization-def 
unfolding assertLiteral-def 
unfolding notifyWatches-def
by (simp add: Let-def)

have getConflictFlag state —→ getConflictFlag ?state'
proof
  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 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 ¬ getConflictFlag ?state'
  with ⟨getConflictFlag state —→ getConflictFlag ?state'⟩
  have ¬ getConflictFlag state
    by simp

  have ∀ l. l el (removeAll literal (getQ ?state')) =
    (∃ c. c el (getF ?state') ∧ isUnitClause c l (elements (getM ?state')))
  proof
    fix l :: Literal
    show l el (removeAll literal (getQ ?state')) =
      (∃ c. c el (getF ?state') ∧ isUnitClause c l (elements (getM ?state')))
  proof
    assume l el (removeAll literal (getQ ?state'))
    hence l el (getQ ?state') l ≠ literal
      by auto
    show ∃ c. c el (getF ?state') ∧ isUnitClause c l (elements (getM ?state'))
  proof (cases l el (getQ state))
    case True
      from (¬ getConflictFlag state)
        :InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)

505
\( l \in (\text{getQ state}) \)

obtain \( c :: \text{Clause} \)

where \( c \in (\text{getF state}) \) isUnitClause \( c l \) (elements (getM state))

unfolding InvariantQCharacterization-def
by auto

show \(?thesis\)
proof (cases \( l \neq \) opposite literal)
  case True
  hence opposite \( l \neq \) literal
  by auto

from (isUnitClause \( c l \) (elements (getM state)))
  (opposite \( l \neq \) literal) \( l \neq \) literal
have isUnitClause \( c l \) ((elements (getM state)) @ [\text{literal}])
  using isUnitClauseAppendValuation[of \( c l \) elements (getM state) \text{literal}]
  by simp
thus \(?thesis\)
  using assms
  using assertLiteralEffect[of \text{state} literal decision]
  by auto
next
  case False
  hence opposite \( l = \) literal
  by simp

from (isUnitClause \( c l \) (elements (getM state)))
have clauseFalse \( c \) (elements (getM \text{state}'))
  using assms
  using assertLiteralEffect[of \text{state} literal decision]
  using unitBecomesFalse[of \( c l \) elements (getM state)]
  by simp
with (\( c \in (\text{getF state}) \))
have formulaFalse (\text{getF state}) (elements (getM \text{state}'))
  by (auto simp add: formulaFalseIfContainsFalseClause)

from assms
have InvariantConflictFlagCharacterization (getConflictFlag \text{state}') (getF \text{state}') (getM \text{state}')
  using InvariantConflictFlagCharacterizationAfterAssertLiteral
  by (simp add: Let-def)
with (formulaFalse (\text{getF state}) (elements (getM \text{state}')))
have getConflictFlag \text{state}'
  using assms
  using assertLiteralEffect[of \text{state} literal decision]
unfolding InvariantConflictFlagCharacterization-def
by auto
with (¬ getConflictFlag ?state')
show ?thesis
by simp
qed
next
case False
then obtain c::Clause
where c el (getF ?state') ∧ isUnitClause c l (elements (getM ?state'))
using *
using ⟨l el (getQ ?state')⟩
using assms
using assertLiteralEffect[of state literal decision]
by auto
thus ?thesis
using formulaEntailsItsClauses[of c getF ?state']
by auto
qed
next
assume ∃c. c el (getF ?state') ∧ isUnitClause c l (elements (getM ?state'))
then obtain c::Clause
where c el (getF ?state') isUnitClause c l (elements (getM ?state'))
by auto
then obtain ci::nat
where 0 ≤ ci ci < length (getF ?state') c = (nth (getF ?state') ci)
using set-conv-nth[of getF ?state']
by auto
then obtain w1::Literal and w2::Literal
where getWatch1 state ci = Some w1 and getWatch2 state ci = Some w2 and getWatch2 state
using ⟨InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)⟩
using ⟨c = (nth (getF ?state') ci)⟩
unfolding InvariantWatchesEl-def
using assms
using assertLiteralEffect[of state literal decision]
by auto
hence w1 ≠ w2
using ⟨ci < length (getF ?state')⟩
using ⟨InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 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)
  (\(\Rightarrow\) getConflictFlag state)
  (\(\langle\) c el (getF ?state')\(\rangle\))
  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 \(\neq\) 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 \(\in\) set (getWatchList ?state (opposite literal)))
case True
with **
  (isUnitClause c l (elements (getM ?state')))
  (\(c = \) (nth (getF ?state') ci))
  have l \(\in\) 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 \(\text{isUnitClause-def}\)
using \(\text{assms}\)
using \(\text{assertLiteralEffect[of state literal decision]}\)
by simp
ultimately
show \(?\text{thesis}\)
by simp

next
case 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 \(\text{getWatch1 state ci = Some w1} \ \text{and} \ \text{getWatch2 state ci = Some w2}\)
    unfolding \(\text{InvariantWatchListsCharacterization-def}\)
    by auto
  have \(\text{literalFalse} \ w1 \ (\text{elements} \ (\text{getM state})) \ \text{or} \ \text{literalFalse} \ w2 \ (\text{elements} \ (\text{getM state}))\)
proof−
  { 
    assume \(\neg \ ?\text{thesis}\)
    hence \(\neg \ \text{literalFalse} \ w1 \ (\text{elements} \ (\text{getM state})) \ \text{or} \ \text{literalFalse} \ w2 \ (\text{elements} \ (\text{getM state}))\)
    using \(\text{w1 \neq \text{opposite literal} \ and} \ \text{w2 \neq \text{opposite literal}}\)
    using \(\text{assms}\)
    using \(\text{assertLiteralEffect[of state literal decision]}\)
    by auto
    with \(\text{w1 \neq w2,}} \ \text{w1 el c} \ \text{or} \ \text{w2 el c}\)
    have \(\neg \text{isUnitClause} c l \ (\text{elements} \ (\text{getM state}'))\)
    unfolding \(\text{isUnitClause-def}\)
    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 \$: \(\exists \ l. \ l\ el c \ \cap \ \text{literalTrue} \ l \ (\text{elements} \ (\text{getM state}))\)
\(\lor\)
\(\forall \ l. \ l\ el c \ \land\ l \neq w1 \ \land\ l \neq w2 \ \rightarrow \text{literalFalse} \ l \ (\text{elements} \ (\text{getM state}))))\)
using (ci < length (getF ?state'))
using (c = (nth (getF ?state') ci))
using (getWatch1 state ci = Some w1)[THEN sym] and
⟨getWatch2 state ci = Some w2⟩[THEN sym]
using assms
using assertLiteralEffect[of state literal decision]
unfolding InvariantWatchCharacterization-def
unfolding watchCharacterizationCondition-def
by auto
thus ?thesis

proof (cases ∀ l. l el c ∧ l ≠ w1 ∧ l ≠ w2 → literalFalse
l (elements (getM state)))
    case True
      with (isUnitClause c l (elements (getM ?state'))) 
      have literalFalse w1 (elements (getM state)) →
        ¬ literalFalse w2 (elements (getM state)) ∧ ¬
literalTrue w2 (elements (getM state)) ∧ l = w2
        literalFalse w2 (elements (getM state)) →
        ¬ literalFalse w1 (elements (getM state)) ∧ ¬
literalTrue w1 (elements (getM state)) ∧ l = w1
    unfolding isUnitClause-def
    using assms
    using assertLiteralEffect[of state literal decision]
    by auto

      with (literalFalse w1 (elements (getM state)) ∨ literalFalse
w2 (elements (getM state)))
      have (literalFalse w1 (elements (getM state)) ∧ ¬ literalFalse
w2 (elements (getM state)) ∧ ¬ literalTrue w2 (elements (getM state))
          ∧ l = w2) ∨
        (literalFalse w2 (elements (getM state)) ∧ ¬ literalFalse
w1 (elements (getM state)) ∧ ¬ literalTrue w1 (elements (getM state))
          ∧ l = w1)
      by blast
    hence isUnitClause c l (elements (getM state))
      using (w1 el c ⊔ w2 el c ⊔ True
      unfolding isUnitClause-def
      by auto
thus ?thesis
using (¬ isUnitClause c l (elements (getM state)));
by simp
next
    case False
    then obtain l':Literal where
        l' el c literalTrue l' (elements (getM state))
      using $
      by auto
    hence literalTrue l' (elements (getM ?state'))
using assertLiteralEffect[of state literal decision]
by auto

from ⟨InvariantConsistent ((getM state) @ [(literal, decision)])⟩
(l' el c) ⟨literalTrue l' (elements (getM ?state'))⟩
show ?thesis
using containsTrueNotUnit[of l' c elements (getM ?state')] using isUnitClause c l (elements (getM ?state'))
using assms
using assertLiteralEffect[of state literal decision]
unfolding InvariantConsistent-def
by auto
qed
qed
qed
qed
qed
}

thus ?thesis
unfolding InvariantQCharacterization-def
by simp
qed

lemma AssertLiteralStartQIrelevant:
fixes literal :: Literal and Wl :: nat list and newWl :: nat list and state :: State
assumes InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)
shows
let state' = (assertLiteral literal decision (state[ getQ := Q' | ])) in
let state'' = (assertLiteral literal decision (state[ getQ := Q'' | ])) in
(getM state') = (getM state'') ∧
(getF state') = (getF state'') ∧
(getSATFlag state') = (getSATFlag state'') ∧
(getConflictFlag state') = (getConflictFlag state'')

using assms
unfolding assertLiteral-def
unfolding notifyWatches-def
unfolding InvariantWatchListsContainOnlyClausesFromF-def
using notifyWatchesStartQIrelevant[of state[ getQ := Q', getM := getM state @ [(literal, decision)] ]
getWatchList (state[ getM := getM state @ [(literal, decision)]])) (opposite literal)
state[ getQ := Q'', getM := getM state @ [(literal, decision)] ]

511
opposite literal \[ \square \]
by \((\text{simp add: Let-def})\)

**lemma** assertedLiteralIsNotUnit:
assumes
\[\begin{align*}
& \text{InvariantConsistent } ((\text{getM } \text{state}) \& \ [(\text{literal, decision}))] \\
& \text{InvariantWatchListsContainOnlyClausesFromF } \text{(getWatchList state)} \\
& \text{(getF state)} \quad \text{and} \\
& \text{InvariantWatchListsUniq } \text{(getWatchList state)} \quad \text{and} \\
& \text{InvariantWatchListsCharacterization } \text{(getWatchList state)} \quad \text{(getWatch1 state)} \quad \text{(getWatch2 state)} \\
& \text{InvariantWatchesEl } \text{(getF state)} \quad \text{(getWatch1 state)} \quad \text{(getWatch2 state)} \quad \text{(getM state)}
\end{align*}\]
and
\[\begin{align*}
& \text{InvariantWatchesDiffer } \text{(getF state)} \quad \text{(getWatch1 state)} \quad \text{(getWatch2 state)} \quad \text{(getM state)} \\
& \text{InvariantWatchCharacterization } \text{(getF state)} \quad \text{(getWatch1 state)} \quad \text{(getWatch2 state)} \quad \text{(getM state)}
\end{align*}\]
shows
let state' = assertLiteral literal decision state in
\[\neg \text{ literal } \in (\text{set } \text{(getQ } \text{state'}) - \text{set } \text{(getQ state)})\]

**proof**
\[\begin{align*}
& \{ \\
& \text{ let } \text{?state }= \text{state}(\text{getM } := \text{getM } \text{state} \& \ [(\text{literal, decision})]) \\
& \text{ let } \text{?state'} = \text{assertLiteral literal decision state} \\
& \text{ assume } \neg \text{ ?thesis} \\
& \text{ have } \forall l. \ l \in \text{set } \text{(getQ ?state')} - \text{set } \text{(getQ ?state)} \longrightarrow \ \\
& \quad (\exists \text{ clause. clause el } \text{(getF } \text{?state}) \land \text{isUnitClause clause l} \ \\
& \quad \text{(elements } \text{(getM } \text{?state)))) \\
& \quad \text{ using NotifyWatchesLoopQEffect[of } \text{?state getM state opposite literal decision } \text{getWatchList ?state (opposite literal) } \square \] \\
& \quad \text{ using assms} \\
& \quad \text{ unfolding InvariantWatchListsUniq-def} \\
& \quad \text{ unfolding InvariantWatchListsCharacterization-def} \\
& \quad \text{ unfolding InvariantWatchListsContainsOnlyClausesFromF-def} \\
& \quad \text{ unfolding InvariantWatchCharacterization-def} \\
& \quad \text{ unfolding assertLiteral-def} \\
& \quad \text{ unfolding notifyWatches-def} \\
& \quad \text{ by } \text{(auto simp add: Let-def)} \\
& \text{ with } \neg \text{ ?thesis} \\
& \text{ obtain clause} \\
& \quad \text{ where isUnitClause clause literal (elements } \text{(getM } \text{?state)}) \\
& \quad \text{ by } \text{(auto simp add: Let-def)} \\
& \text{ hence False} \\
& \text{ unfolding isUnitClause-def} \\
& \text{ by simp} \\
& \}
\]
thus ?thesis
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)
  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)
  ¬ 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 InvariantUniqQAAfterAssertLiteral:
assumes
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state)
\begin{verbatim}
(getF state) and
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantUniqQ (getQ state)
shows
  let state' = assertLiteral literal decision state in
    InvariantUniqQ (getQ state')
using assms
using InvariantUniqQAAfterNotifyWatchesLoop of state getM := getM state @ [(lLiteral, decision)]
getWatchList (state getM := getM state @ [(lLiteral, decision)]) (opposite literal)
opposite literal []
unfolding assertLiteral-def
unfolding notifyWatches-def
unfolding InvariantWatchListsContainOnlyClausesFromF-def
by (auto simp add: Let-def)

lemma InvariantsNoDecisionsWhenConflictNorUnitAfterAssertLiteral:
assumes
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
  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))
  decision \rightarrow \neg (getConflictFlag state) \land (getQ state) = []
shows
  let state' = assertLiteral literal decision state in
    InvariantNoDecisionsWhenConflict (getF state') (getM state')
    (currentLevel (getM state')) \land
    InvariantNoDecisionsWhenUnit (getF state') (getM state') (currentLevel (getM state'))
proof-
  { 
    let ?state' = assertLiteral literal decision state
    fix level
    assume level < currentLevel (getM ?state')
    have \neg formulaFalse (getF ?state') (elements (prefixToLevel level (getM ?state'))) \land
        \neg (\exists clause literal. clause el (getF ?state') \land
              isUnitClause clause literal (elements (prefixToLevel level (getM ?state'))))
  }
\end{verbatim}
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) ∧ isInUnitClause 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]
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
?

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)
state) (getM state)
InvariantVarsQ (getQ state) F0 Vbl
InvariantVarsF (getF state) F0 Vbl

shows
let state′ = assertLiteral literal decision state in
InvariantVarsQ (getQ state′) F0 Vbl

proof –
let ?Q′ = {ul. ∃uc. uc el (getF state) ∧
           (opposite literal) el uc ∧ isUnitClause uc ul (elements
           (getM state)) @ [literal])}
let ?state′ = assertLiteral literal decision state
have vars ?Q′ ⊆ vars (getF state)
proof
fix vbl::Variable
assume vbl ∈ vars ?Q′
then obtain ul::Literal
    where ul ∈ ?Q′ var ul = vbl
    by auto
then obtain uc::Clause
    where uc el (getF state) isUnitClause uc ul (elements (getM
    state)) @ [literal])
    by auto
hence vars uc ⊆ vars (getF state) var ul ∈ vars uc
    using formulaContainsItsClausesVariables[of getF state]
    using clauseContainsItsLiteralsVariable[of ul uc]
    unfolding isUnitClause-def
    by auto
thus vbl ∈ vars (getF state)
    using (var ul = vbl)
    by auto
qed
thus ?thesis
using assms
using assertLiteralQEffect[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 \( \text{?uLiteral} = \text{hd} (\text{getQ state}) \)
let \( \text{?state'} = \text{applyUnitPropagate state} \)
\begin{verbatim}
obtain uClause
  where isUnitClause uClause \( \text{?uLiteral} \) (elements (getM state)) and
  (getM ?state') = (getM state) @ \([\text{?uLiteral}, \text{False}]\]
  using \( \text{assms} \)
  using applyUnitPropagateEffect[of state]
by (auto simp add: Let-def)
thus \( ?\text{thesis} \) using \( \text{assms} \)
using InvariantConsistentAfterUnitPropagate[of getM state uClause ?uLiteral getM ?state']
by (auto simp add: Let-def)
qed
\end{verbatim}

**lemma InvariantUniqAfterApplyUnitPropagate:**

assumes

\( \text{InvariantUniq} (\text{getM state}) \)
\( \text{InvariantWatchesEl} (\text{getF state}) (\text{getWatch1 state}) (\text{getWatch2 state}) \)

and

\( \text{InvariantWatchListsContainOnlyClausesFromF} (\text{getWatchList state}) (\text{getF state}) \)
\( \text{and} \)
\( \text{InvariantQCharacterization} (\text{getConflictFlag state}) (\text{getQ state}) (\text{getF state}) (\text{getM state}) \)
\( \text{getQ state} \neq [] \)
\( \neg (\text{getConflictFlag state}) \)

shows

let \( \text{state'} = \text{applyUnitPropagate state} \) in

\( \text{InvariantUniq} (\text{getM state'}) \)

**proof**

let \( \text{?uLiteral} = \text{hd} (\text{getQ state}) \)
let \( \text{?state'} = \text{applyUnitPropagate state} \)
\begin{verbatim}
obtain uClause
  where isUnitClause uClause \( \text{?uLiteral} \) (elements (getM state)) and
  (getM ?state') = (getM state) @ \([\text{?uLiteral}, \text{False}]\]
  using \( \text{assms} \)
  using applyUnitPropagateEffect[of state]
by (auto simp add: Let-def)
thus \( ?\text{thesis} \) using \( \text{assms} \)
using InvariantUniqAfterUnitPropagate[of getM state uClause ?uLiteral getM ?state']
by (auto simp add: Let-def)
qed
\end{verbatim}

**lemma InvariantWatchCharacterizationAfterApplyUnitPropagate:**

assumes

\( \text{InvariantConsistent} (\text{getM state}) \)
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)
InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)
InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
\( \neg (\text{getConflictFlag state}) \neq [] \)
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 InvariantWatchCharacterizationAfterAssertLiteral[of state ?uLiteral False]
    using assertLiteralEffect
    unfolding applyUnitPropagate-def
    by (simp add: Let-def)
qed

lemma InvariantConflictFlagCharacterizationAfterApplyUnitPropagate:
assumes
  InvariantConsistent (getM state)
  InvariantUniq (getM state)
$\text{InvariantWatchListsContainOnlyClausesFromF}$ (getWatchList state) \((getF\text{ state})\) \text{ and } $\text{InvariantWatchListsUniq}$ (getWatchList state) \text{ and } $\text{InvariantWatchListsCharacterization}$ (getWatchList state) (getWatch1 state) (getWatch2 state)

\text{and}

$\text{InvariantWatchesDiffer}$ (getF state) (getWatch1 state) (getWatch2 state) \text{ and } $\text{InvariantWatchCharacterization}$ (getWatchList state) (getWatch1 state) (getWatch2 state) (getM state)

$\neg$ getConflictFlag state

\text{shows}

let state' = (applyUnitPropagate state) in

$\text{InvariantConflictFlagCharacterization}$ (getConflictFlag state') (getF state') (getM state')

\text{proof}–

let ?uLiteral = hd (getQ state)
let ?state' = assertLiteral ?uLiteral False state
let ?state'' = applyUnitPropagate state

have $\text{InvariantConsistent}$ (getM ?state')
using assms
using $\text{InvariantConsistentAfterApplyUnitPropagate}$[of state]
unfolding applyUnitPropagate-def
by (auto simp add: Let-def)

moreover

have $\text{InvariantUniq}$ (getM ?state')
using assms
using $\text{InvariantUniqAfterApplyUnitPropagate}$[of state]
unfolding applyUnitPropagate-def
by (auto simp add: Let-def)

ultimately

show $\text{thesis}$
using assms
using $\text{InvariantConflictFlagCharacterizationAfterAssertLiteral}$[of state ?uLiteral False]
using assertLiteralEffect
unfolding applyUnitPropagate-def
by (simp add: Let-def)

qed

\text{lemma} $\text{InvariantConflictClauseCharacterizationAfterApplyUnitPropagate}$:
assumes
  \( \text{InvariantWatchesEl} \ (\text{getF state}) \ (\text{getWatch1 state}) \ (\text{getWatch2 state}) \)
  \( \text{and} \)
  \( \text{InvariantWatchListsContainOnlyClausesFromF} \ (\text{getWatchList state}) \ (\text{getF state}) \)
  \( \text{InvariantWatchListsCharacterization} \ (\text{getWatchList state}) \ (\text{getWatch1 state}) \ (\text{getWatch2 state}) \) \( \text{and} \)
  \( \text{InvariantWatchListsUniq} \ (\text{getWatchList state}) \)
  \( \neg \ (\text{getConflictFlag state}) \)

shows
  \( \text{let state'} = \text{applyUnitPropagate state} \) in
  \( \text{InvariantConflictClauseCharacterization} \ (\text{getConflictFlag state'}) \ (\text{getConflictClause state'}) \ (\text{getF state'}) \ (\text{getM state'}) \)
  \( \text{using assms} \)
  \( \text{using InvariantConflictClauseCharacterizationAfterAssertLiteral[of state hd (getQ state) False]} \)
  \( \text{unfolding applyUnitPropagate-def} \)
  \( \text{unfolding InvariantWatchesEl-def} \)
  \( \text{unfolding InvariantWatchListsContainOnlyClausesFromF-def} \)
  \( \text{unfolding InvariantWatchListsCharacterization-def} \)
  \( \text{unfolding InvariantWatchListsUniq-def} \)
  \( \text{unfolding InvariantConflictClauseCharacterization-def} \)
  \( \text{by (simp add: Let-def)} \)

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

assumes
  \( \text{InvariantConsistent} \ (\text{getM state}) \)
  \( \text{InvariantWatchListsContainOnlyClausesFromF} \ (\text{getWatchList state}) \ (\text{getF state}) \) \( \text{and} \)
  \( \text{InvariantWatchListsUniq} \ (\text{getWatchList state}) \) \( \text{and} \)
  \( \text{InvariantWatchListsCharacterization} \ (\text{getWatchList state}) \ (\text{getWatch1 state}) \ (\text{getWatch2 state}) \)
  \( \text{and} \)
  \( \text{InvariantWatchesEl} \ (\text{getF state}) \ (\text{getWatch1 state}) \ (\text{getWatch2 state}) \)
  \( \text{and} \)
  \( \text{InvariantWatchesDiffer} \ (\text{getF state}) \ (\text{getWatch1 state}) \ (\text{getWatch2 state}) \)
  \( \text{and} \)
  \( \text{InvariantWatchCharacterization} \ (\text{getF state}) \ (\text{getWatch1 state}) \ (\text{getWatch2 state}) \ (\text{getM state}) \)
  \( \text{InvariantConflictFlagCharacterization} \ (\text{getConflictFlag state}) \ (\text{getF state}) \ (\text{getM state}) \)
  \( \text{InvariantQCharacterization} \ (\text{getConflictFlag state}) \ (\text{getQ state}) \ (\text{getF state}) \ (\text{getM state}) \)
  \( \text{InvariantUniqQ} \ (\text{getQ state}) \)
  \( (\text{getQ state}) \neq [] \)
  \( \neg (\text{getConflictFlag state}) \)

shows
  \( \text{let state'' = applyUnitPropagate state in} \)
  \( \text{InvariantQCharacterization} \ (\text{getConflictFlag state''}) \ (\text{getQ state''}) \ (\text{getF state''}) \ (\text{getM state''}) \)
proof –
let \( ?uLiteral = \text{hd} \ (\text{getQ} \ \text{state}) \)
let \( ?\text{state}' = \text{assertLiteral} \ ?uLiteral \ \text{False} \ \text{state} \)
let \( ?\text{state}'' = \text{applyUnitPropagate} \ \text{state} \)
have \( \text{InvariantConsistent} \ (\text{getM} \ ?\text{state}') \)
  using \( \text{assms} \)
  using \( \text{InvariantConsistentAfterApplyUnitPropagate}[\text{of} \ \text{state}] \)
  unfolding \( \text{applyUnitPropagate-def} \)
  by (auto simp add: Let-def)

hence \( \text{InvariantQCharacterization} \ (\text{getConflictFlag} \ ?\text{state}') \ (\text{removeAll} \ ?uLiteral \ (\text{getQ} \ ?\text{state}')) \ (\text{getF} \ ?\text{state}') \ (\text{getM} \ ?\text{state}') \)
  using \( \text{assms} \)
  using \( \text{InvariantQCharacterizationAfterAssertLiteral}[\text{of} \ \text{state} \ ?uLiteral \ \text{False}] \)
  by (simp add: Let-def)

moreover
have \( \text{InvariantUniqQ} \ (\text{getQ} \ ?\text{state}') \)
  using \( \text{assms} \)
  using \( \text{InvariantUniqQAfterAssertLiteral}[\text{of} \ \text{state} \ ?uLiteral \ \text{False}] \)
  by (simp add: Let-def)

have \( ?uLiteral = (\text{hd} \ (\text{getQ} \ ?\text{state}')) \)
proof –
obtain \( s \)
  where \( (\text{getQ} \ \text{state}) @ s = \text{getQ} \ ?\text{state}' \)
  using \( \text{assms} \)
  using \( \text{assertLiteralEffect}[\text{of} \ \text{state} \ ?uLiteral \ \text{False}] \)
  unfolding \( \text{isPrefix-def} \)
  by auto

hence \( \text{getQ} \ ?\text{state}' = (\text{getQ} \ \text{state}) @ s \)
  by (rule sym)
thus \( \text{thesis} \)
  using \( \text{getQ} \ \text{state} \neq [] \)
  using \( \text{hd-append}[\text{of} \ \text{getQ} \ \text{state} \ s] \)
  by auto
qed

hence \( \text{set} \ (\text{getQ} \ ?\text{state}''') = \text{set} \ (\text{removeAll} \ ?uLiteral \ (\text{getQ} \ ?\text{state}')) \)
  using \( \text{assms} \)
  using \( \text{InvariantUniqQ} \ (\text{getQ} \ ?\text{state}')) \)
  unfolding \( \text{InvariantUniqQ-def} \)
  using \( \text{uniqHeadTailSet}[\text{of} \ \text{getQ} \ ?\text{state}] \)
  unfolding \( \text{applyUnitPropagate-def} \)
  by (simp add: Let-def)

ultimately
show \( \text{thesis} \)
  unfolding \( \text{InvariantQCharacterization-def} \)
  unfolding \( \text{applyUnitPropagate-def} \)
by (simp add: Let-def)
qed

lemma InvariantUniqQAfterApplyUnitPropagate:
assumes
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
InvariantWatchListsContainOnlyClausesFromF (getWatchList state)
InvariantUniqQ (getQ state)
getQ state \neq []
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 \neq []'
  have getQ ?state' \neq []
    by simp
ultimately
show ?thesis
  using \langle getQ state \neq [] \rangle
  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'))

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⟩
                unfoldingInvariantGetReasonIsReason-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

526
where \(\text{getReason state} (hd (getQ state)) = \text{Some reason } 0 \leq \text{reason} \land \text{reason} < \text{length (getF state)}\)

\(\text{isUnitClause (nth (getF state) reason) (hd (getQ state))}
\) (\(\text{elements (getM state)}\)) \lor
\(\text{clauseFalse (nth (getF state) reason) (elements (getM state))}\)

\(\text{using (InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state))):}\)

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

\(\text{by auto}\)

\(\text{hence isUnitClause (nth (getF state) reason) (hd (getQ state)) (elements (getM state))}\)

\(\text{using (\sim getConflictFlag state):}\)

\(\text{using (InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state))}\)

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

\(\text{using nth-mem [of reason getF state]}\)

\(\text{using formulaFalseIffContainsFalseClause [of getF state elements (getM state)]}\)

\(\text{by simp}\)

\(\text{thus ?thesis}\)

\(\text{using (getReason state (hd (getQ state)) = Some reason \(0 \leq \text{reason} \land \text{reason} < \text{length (getF state)})}:}\)

\(\text{using isUnitClauseIsReason [of nth (getF state) reason hd (getQ state) elements (getM state) [hd (getQ state)]]}\)

\(\text{by simp}\)

\(\text{qed}\)

moreover

\{\}

\text{fix literal::Literal}\n
\text{assume currentLevel (getM state0) > 0}\n
\text{hence currentLevel (getM state) > 0}\n
\(\text{unfolding currentLevel-def}\)

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

\text{assume literal el removeAll (hd (getQ state0)) (getQ state0)}\n
\text{hence literal \neq hd (getQ state) literal el getQ state}\n
\(\text{by auto}\)

then obtain reason

\text{where getReason state literal = Some reason } 0 \leq \text{reason} \land \text{reason} < \text{length (getF state) and}\n
\(\text{\sim: isUnitClause (nth (getF state) reason) literal (elements (getM state))}\) \lor
\(\text{clauseFalse (nth (getF state) reason) (elements (getM state))}\)

\(\text{using (currentLevel (getM state) > 0),}\)

\(\text{using (InvariantGetReasonIsReason (getReason state) (getF state))}\)
state) (getM state) (set (getQ state))

unfolding InvariantGetReasonIsReason-def
by auto

hence \( \exists \) reason. getReason ?state0 literal = Some reason \( \land 0 \leq \)
reason \( \land \) reason < length (getF ?state0) \( \land \)
(isUnitClause (nth (getF ?state0) reason) literal (elements
(getM ?state0))) \( \lor \)
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 \( \neq \) hd (getQ state))
  using (\( 0 \leq \) reason \( \land \) 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 \( \neq \) hd (getQ state))
  using (\( 0 \leq \) reason \( \land \) 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 (\( 0 \leq \) reason \( \land \) reason < length (getF state))
  using clauseFalseAppendValuation[of nth (getF state) reason
elements (getM state) [hd (getQ state)]]
  by auto
qed

}  
ultimately
show ?thesis
  unfolding InvariantGetReasonIsReason-def
  by auto
qed

hence InvariantGetReasonIsReason (getReason ?state') (getF ?state')
  (getM ?state') (set (removeAll (hd (getQ state)) (getQ state)) \cup (set
  (getQ ?state') \setminus (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
    using assertLiteralEffect[of state hd (getQ state) False]
    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 \neq []
    by simp

have set (removeAll (hd (getQ state)) (getQ state)) \cup (set (getQ
  ?state') \setminus (set (getQ state))) =
  set (removeAll (hd (getQ state)) (getQ ?state'))
    using getQ ?state' = getQ state @ s
    using getQ state \neq []
    by auto

have uniq (getQ ?state')
  using assms
  using InvariantUniqQAAfterAssertLiteral[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 `InvariantEquivalentZLAfterApplyUnitPropagate`:
assumes
`InvariantEquivalentZL (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)`
and
¬ (getConflictFlag state)
getQ state ≠ []

shows
let `state' = applyUnitPropagate state`
in `InvariantEquivalentZL (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 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)
  using (getM state = prefixToLevel 0 (getM state))
  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)
thus \( \text{thesis} \)
using \( \langle \text{InvariantEquivalenceZL} \ (\text{getF state}) \ (\text{getM state}) \ \Phi \rangle \)
unfolding \( \text{InvariantEquivalenceZL-def} \)
by \( \text{(simp add: Let-def)} \)
qed

lemma \( \text{InvariantVarsQ}\text{TL} \):
assumes
\( \text{InvariantVarsQ} \ Q \ F0 \ Vbl \)
\( Q \neq [] \)
shows
\( \text{InvariantVarsQ} \ (\text{tl} \ Q) \ F0 \ Vbl \)
proof−
have \( \text{InvariantVarsQ} \ ((\text{hd} \ Q) \neq (\text{tl} \ Q)) \ F0 \ Vbl \)
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 \( \text{thesis} \)
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
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
using assertLiteralEffect[of state hd (getQ state) False]
unfolding applyUnitPropagate-def
by (simp add: Let-def)
moreover
have (getQ ?state') ≠ []
  using assms
  using assertLiteralEffect[of state hd (getQ state) False]
  using (getQ state) ≠ []
  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 lexLessState (Vbl::Variable set) == { (state1, state2). (getM state1, getM state2) ∈ lexLessRestricted Vbl }

lemma exhaustiveUnitPropagateTermination:
  fixes
    state::State and Vbl::Variable set
  assumes
    InvariantUniq (getM state)
    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)
    InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state)
    (getM state)
    InvariantUniqQ (getQ state)
    InvariantVarsM (getM state) F0 Vbl
    InvariantVarsQ (getQ state) F0 Vbl
    InvariantVarsF (getF state) F0 Vbl
    finite Vbl
  shows
    exhaustiveUnitPropagate-dom state
  using asms
  proof (induct rule: wf-induct[of lexLessState (vars F0 ∪ Vbl)])
    case 1
    show ?case
      unfolding wf-eq-minimal
      proof (cases)
        show ∀ Q (state::State). state ∈ Q → (∃ stateMin∈Q. ∀ state'. (state', stateMin) ∈ lexLessState (vars F0 ∪ Vbl) → state' ∉ Q)
          proof (fix Q :: State set and state :: State
                    assume state ∈ Q
                    let ?Q1 = { M::LiteralTrail. ∃ state. state ∈ Q ∧ (getM state)
from \( \{ \text{state} \in Q \} \)
have \( \text{getM state} \in {?Q1} \)
by auto
have \( \text{wf (lexLessRestricted (vars F0 \cup Vbl))} \)
using \( \{ \text{finite Vbl} \} \)
using finiteVarsFormula[of F0]
using \( \text{wfLexLessRestricted[of vars F0 \cup Vbl]} \)
by simp
with \( \{ \text{getM state} \in {?Q1} \} \)
obtain \( \text{Mmin where } \text{Mmin} \in {?Q1} \forall M'. (M', \text{Mmin}) \in \text{lexLessRestricted (vars F0 \cup Vbl}) \longrightarrow M' \notin {?Q1} \)
unfolding \( \text{wf-eq-minimal} \)
apply (erule-tac \( x = {?Q1} \) in allE)
apply (erule-tac \( x = \text{getM state} \) in allE)
by auto
from \( \{ \text{Mmin} \in {?Q1} \} \) obtain \( \text{stateMin} \)
where \( \text{stateMin} \in Q \ (\text{getM stateMin}) = \text{Mmin} \)
by auto
have \( \forall \text{state}'. (\text{state}', \text{stateMin}) \in \text{lexLessState (vars F0 \cup Vbl)} \longrightarrow \text{state'} \notin Q \)
proof
fix \( \text{state}' \)
show \( (\text{state}', \text{stateMin}) \in \text{lexLessState (vars F0 \cup Vbl}) \longrightarrow \text{state'} \notin Q \)
proof
assume \( (\text{state}', \text{stateMin}) \in \text{lexLessState (vars F0 \cup Vbl}) \)
hence \( (\text{getM state}', \text{getM stateMin}) \in \text{lexLessRestricted (vars F0 \cup Vbl}) \)
unfolding \( \text{lexLessState-def} \)
by auto
from \( (\forall M'. (M', \text{Mmin}) \in \text{lexLessRestricted (vars F0 \cup Vbl}) \longrightarrow M' \notin {?Q1} \) \)
\((\text{getM state}', \text{getM stateMin}) \in \text{lexLessRestricted (vars F0 \cup Vbl)}) \ (\text{getM stateMin} = \text{Mmin}) \)
have \( \text{getM state'} \notin {?Q1} \)
by simp
with \( \text{getM stateMin} = \text{Mmin} \)
show \( \text{state'} \notin Q \)
by auto
qed
qed
with \( \{ \text{stateMin} \in Q \} \)
have \( \exists \text{stateMin} \in Q. (\forall \text{state}'. (\text{state}', \text{stateMin}) \in \text{lexLessState (vars F0 \cup Vbl}) \longrightarrow \text{state'} \notin Q) \)
by auto
}
thus \( {?\text{thesis}} \)
by auto

535
qed

qed

next

case (2 state')

note ih = this

show ?case

proof (cases getQ state' = [] \or getConflictFlag state')

<table>
<thead>
<tr>
<th>case</th>
<th>False</th>
</tr>
</thead>
</table>

let ?state'' = applyUnitPropagate state'

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'') 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]

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)

536
moreover have InvariantUniq\(Q\) (get\(?\text{state}'\))
  using \(\text{ih}\)
  using InvariantUniq\(QA\)fterApplyUnitPropagate[of \text{state}']
  using False
  by (simp add: Let-def)
moreover have Invariant\(C\)onsistent (get\(?\text{state}'\))
  using \(\text{ih}\)
  using Invariant\(C\)onsistent\(A\)fterApplyUnitPropagate[of \text{state}']
  using False
  by (simp add: Let-def)
moreover have Invariant\(U\)uniq (get\(?\text{state}'\))
  using \(\text{ih}\)
  using Invariant\(U\)uniq\(A\)fterApplyUnitPropagate[of \text{state}']
  using False
  by (simp add: Let-def)
moreover have Invariant\(V\)ars\(M\) (get\(?\text{state}'\)) \(F0\) Vbl Invariant\(V\)ars\(Q\) (get\(?\text{state}'\)) \(F0\) Vbl
  using \(\text{ih}\)
  using (~ (get\(Q\) \(\text{state}'\) = [] \∨ getConflictFlag \(\text{state}'\))
  using Invariants\(V\)ars\(A\)fterApplyUnitPropagate[of \text{state}' \(F0\) Vbl]
  by (auto simp add: Let-def)
moreover have Invariant\(V\)ars\(F\) (get\(F\) \(?\text{state}'\)) \(F0\) Vbl
  unfolding applyUnitPropagate-def
  using assertLiteralEffect[of \text{state}' \(hd\) (get\(Q\) \(\text{state}'\)) False]
  using \(\text{ih}\)
  by (simp add: Let-def)
moreover have \(?\text{state}', \text{state}'\) ∈ lexLess\(S\)tate (vars \(F0\) ∪ Vbl)
proof –
  have get\(M\) \(?\text{state}'\) = get\(M\) \(\text{state}'\) @ [(hd (get\(Q\) \(\text{state}'\)), False)]
    unfolding applyUnitPropagate-def
    using \(\text{ih}\)
    using assertLiteralEffect[of \text{state}' \(hd\) (get\(Q\) \(\text{state}'\)) False]
    by (simp add: Let-def)
thus \(?\text{thesis}\)
  unfolding lexLess\(S\)tate-def
  unfolding lexLess\(R\)estricted-def
  using lexLess\(A\)pend[of [(hd (get\(Q\) \(\text{state}'\)), False)] get\(M\) \(\text{state}'\)]
  using Invariant\(C\)onsistent (get\(?\text{state}'\))
  unfolding Invariant\(C\)onsistent-def
  using Invariant\(C\)onsistent (get\(?\text{state}'\))
  unfolding Invariant\(C\)onsistent-def
  using Invariant\(U\)uniq (get\(?\text{state}'\))
  unfolding Invariant\(U\)uniq-def

537
using $\langle$InvariantUniq $(\text{getM state}')\rangle$
unfolding InvariantUniq-def
using $\langle$InvariantVarsM $(\text{getM ?state}'') F0 Vbl\rangle$
using $\langle$InvariantVarsM $(\text{getM state}') F0 Vbl\rangle$
unfolding InvariantVarsM-def
by simp
ged
ultimately
have $\text{exhaustiveUnitPropagate-dom ?state''}$
using $\text{ih}$
by auto
thus $?thesis$
using $\text{exhaustiveUnitPropagate-dom.intros[of state']}$
using False
by simp
next
case True
show $?thesis$
apply (rule exhaustiveUnitPropagate-dom.intros)
using True
by simp
ged

lemma exhaustiveUnitPropagatePreservedVariables:
assumes
$\text{exhaustiveUnitPropagate-dom state}$
$\text{InvariantWatchListsContainOnlyClausesFromF (getWatchList state)}$
$(\text{getF state}) \text{ and}$
$\text{InvariantWatchListsUniq (getWatchList state)} \text{ and}$
$\text{InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state)}$
$(\text{getWatch1 state}) \text{ (getWatch2 state)}$
$\text{InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)}$
and
$\text{InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)}$
shows
let $\text{state} = \text{exhaustiveUnitPropagate state in}$
$(\text{getSATFlag state'}) = (\text{getSATFlag state})$
using assms
proof (induct state rule: exhaustiveUnitPropagate-dom.induct)
case (step state')
note $\text{ih} = \text{this}$
show $?case$
proof (cases $(\text{getConflictFlag state'}) \lor (\text{getQ state'}) = []$)
case True
with $\text{exhaustiveUnitPropagate.simps[of state']}$
have $\text{exhaustiveUnitPropagate state'} = \text{state'}$
by simp

538
thus \( \text{thesis} \)
   by (simp only: Let-def)
next
  case False
  let \( \text{state}'' = \text{applyUnitPropagate state}' \)

  have \( \text{exhaustiveUnitPropagate state}' = \text{exhaustiveUnitPropagate state}'' \)
    using exhaustiveUnitPropagate.simps[of state]
    using False
    by simp
  moreover
  have \( \text{InvariantWatchListsContainOnlyClausesFromF} (\text{getWatchList state}'' \text{state}''') \)
    (getF state''') and
    \( \text{InvariantWatchListsUniq} (\text{getWatchList state}''') \) and
    \( \text{InvariantWatchListsCharacterization} (\text{getWatchList state}''') \) (getWatch1 state''')
    (getWatch2 state''')
    (getWatch1 state''') (getWatch2 state''')
    and
    \( \text{InvariantWatchesEl} (\text{getF state}''') \) (getWatch1 state''') (getWatch2 state''')
    \( \text{InvariantWatchesDiffer} (\text{getF state}''') \) (getWatch1 state''') (getWatch2 state''')
    using ih
    using \( \text{WatchInvariantsAfterAssertLiteral} [\text{of state}'' \text{hd} (\text{getQ state}')] \text{False} \)
    unfolding applyUnitPropagate-def
    by (auto simp add: Let-def)
  moreover
  have \( \text{getSATFlag state}'' = \text{getSATFlag state}' \)
    unfolding applyUnitPropagate-def
    using assertLiteralEffect[of state'' hd (getQ state')] False
    using ih
    by (simp add: Let-def)
  ultimately
  show \( \text{thesis} \)
  using ih
  using False
  by (simp add: Let-def)
qed
qed

lemma exhaustiveUnitPropagatePreservesCurrentLevel:
assumes
  exhaustiveUnitPropagate-dom state
  \( \text{InvariantWatchListsContainOnlyClausesFromF} (\text{getWatchList state}) \) (getF state) and
  \( \text{InvariantWatchListsUniq} (\text{getWatchList state}) \) and
  \( \text{InvariantWatchListsCharacterization} (\text{getWatchList state}) \) (getWatch1 state) (getWatch2 state)
  \( \text{InvariantWatchesEl} (\text{getF state}) \) (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
    using 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]
      using ih
      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') ∧
    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')
\begin{align*}
\text{(getF state') (getM state') } & \land \\
\text{InvariantQCharacterization (getConflictFlag state') (getQ state')} & \\
\text{(getF state') (getM state') } & \land \\
\text{InvariantUniqQ (getQ state') } & \land \\
\text{InvariantVarsQ (getQ state') F0 Vbl } & \land \\
\text{InvariantVarsM (getM state') F0 Vbl } & \land \\
\text{InvariantVarsF (getF state') F0 Vbl } & \land \\
\end{align*}

\text{using} \ \text{assms}

\text{proof (induct state rule: exhaustiveUnitPropagate-dom.induct)}

\text{case (step state')}

\text{note} \ \text{ih} = \text{this}

\text{show} \ ?\text{case}

\text{proof (cases (getConflictFlag state') } \lor \ (\text{getQ state'}) = [])

\text{case} True

\text{with exhaustionUnitPropagate}.\text{.simps[of state']}

\text{have exhaustionUnitPropagate state' = state'}

\text{by} \ \text{simp}

\text{thus} \ ?\text{thesis}

\text{using} \ \text{ih}

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

\text{next}

\text{case} False

\text{let} \ ?\text{state''} = \text{applyUnitPropagate state'}

\text{have exhaustionUnitPropagate state' = exhaustionUnitPropagate}

?\text{state''}

\text{using} \ \text{exhaustionUnitPropagate}.\text{.simps[of state']}

\text{using} \ \text{False}

\text{by} \ \text{simp}

\text{moreover}

\text{have InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state'') (getF ?state'')} \ \text{and}

\text{InvariantWatchListsUniq (getWatchList ?state'') and}

\text{InvariantWatchListsCharacterization (getWatchList ?state'')} (getWatch1 ?state'') (getWatch2 ?state'') \ \text{and}

\text{Invariant WatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')}

\text{using} \ \text{ih}

\text{using} \ \text{WatchInvariantsAfterAssertLiteral[of state' hd (getQ state')} False\text{]}

\text{unfolding} \ \text{applyUnitPropagate-def}

\text{by} \ (\text{auto simp add: Let-def})

\text{moreover}

\text{have InvariantWatchCharacterization (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')} (getM ?state'') \ \text{using} \ \text{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'') \(\forall\) Vbl InvariantVarsQ (getQ ?state'') \(\forall\) Vbl
  using ih
  using \(\neg (\text{getConflictFlag state'} \lor \text{getQ state'} = [])\)
  using InvariantsVarsAfterApplyUnitPropagate[of state'] \(\forall\) Vbl
  by (auto simp add: Let-def)
moreover
have InvariantVarsF (getF ?state'') \(\forall\) Vbl
  unfolding applyUnitPropagate-def
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)
sows
  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')
ote ih = this
show 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") using ih(2) ih(3) ih(4) ih(5) ih(6) ih(7)
using WatchInvariantsAfterAssertLiteral[of state' hd (getQ state')]
end

unfolding applyUnitPropagate-def
by (auto simp add: Let-def)
moreover
have InvariantConflictClauseCharacterization (getConflictFlag ?state") (getConflictClause ?state") (getF ?state") (getM ?state")
using ih(2) ih(3) ih(4) ih(5) ih(6)
using (∼ (getConflictFlag state' ∨ getQ state' = [])).
using InvariantConflictClauseCharacterizationAfterApplyUnitPropagate[of state']
by (auto simp add: Let-def)
ultimately
show thesis
using ih(1) ih(2)
using False
by (simp only: Let-def) (blast)
qed
qed

lemma InvariantsNoDecisionsWhenConflictNorUnitAfterExhaustivePropagate:
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)
  InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state)
shows
let state′ = exhaustiveUnitPropagate state in
InvariantNoDecisionsWhenConflict (getF state′) (getM state′) (currentLevel (getM state′)) ∧
InvariantNoDecisionsWhenUnit (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′) ∨ (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″) and
    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 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 InvariantNoDecisionsWhenUnit (getF ?state'') (getM ?state'')
(currentLevel (getM state))
InvariantNoDecisionsWhenConflict (getF state) (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'')
    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 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 InvariantQCharacterization-def
  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


lemma `InvariantEquivalentZLAfterExhaustiveUnitPropagate`:
assumes
    exhaustiveUnitPropagate-dom state
    `InvariantConsistent` (getM state)
    `InvariantUniq` (getM state)
    `InvariantEquivalentZL` (getF state) (getM state) Phi
    `InvariantWatchListsContainOnlyClausesFromF` (getWatchList state)

550
(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)
InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state)
InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
InvariantUniqQ (getQ state)

shows
let state' = exhaustiveUnitPropagate state in
InvariantEquivalentZL (getF state') (getM state') Phi

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 (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'')
  qed
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]
   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 \( \text{InvariantEquivalentZL} (\text{getF state'}) (\text{getM state'}) \Phi \)
  using ih
  using \( \text{InvariantEquivalentZLAfterApplyUnitPropagate[state']} \Phi \)
  using False
  by (simp add: Let-def)
moreover
have currentLevel (getM state') = currentLevel (getM state'')
  unfolding applyUnitPropagate-def
  using assertLiteralEffect[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

lemma conflictFlagOrQEmptyAfterExhaustiveUnitPropagate:
assumes
exhaustiveUnitPropagate-dom state
shows
let state' = exhaustiveUnitPropagate state in
  (getConflictFlag state') \lor (getQ 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[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[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) (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') 
  ∧
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') 
  ∧
InvariantConflictFlagCharacterization (getConflictFlag state') (getF state') (getM state') 
  ∧
InvariantConflictClauseCharacterization (getConflictFlag state') (getF state') (getM state') 
  ∧
InvariantQCharacterization (getConflictFlag state') (getQ state') (getF state') (getM state') 
  ∧
InvariantGetReasonIsReason (getReason state') (getF state') (getM state') (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
  unfolding ⟨¬ clauseTrue ?clause' (elements (getM state))⟩
  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]
  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 ?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 InvariantConsistent-def
  unfolding InvariantUniq-def
  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
  unfolding InvariantUniq-def
  using assertLiteralEffect[of state hd ?clause' False]
  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 \( \texttt{?state''} = \texttt{?state'} \) \( \texttt{getM := (getM ?state') @ [\texttt{(hd ?clause}, } \) '\texttt{False}] \) \)

have InvariantQCharacterization (getConflictFlag \texttt{?state'}) (getQ \texttt{?state'}) (getF \texttt{?state'}) (getM \texttt{?state'})

proof (cases getConflictFlag \texttt{state'}
  case \texttt{True}
    hence getConflictFlag \texttt{?state'}
      using \texttt{assms}
      using assertLiteralConflictFlagEffect[\texttt{of state hd ?clause'} \texttt{False}]
    by (auto simp add: Let-def)
  thus \texttt{?thesis}
    using \texttt{assms}
    unfolding InvariantQCharacterization-def
    by simp
next
  case \texttt{False}
    with \( (\texttt{getConflictFlag state} \vee \texttt{(getQ state) = []} \) \)
    have \texttt{getQ state} = []
      by simp
    thus \texttt{?thesis}
      using InvariantQCharacterizationAfterAssertLiteralNotInQ[\texttt{of state hd ?clause'} \texttt{False}]
      using \texttt{assms}
      using \( \texttt{(uniq (elements (getM \texttt{?state'}) \) \)} \)
      using \( \texttt{(consistent (elements (getM \texttt{?state'}) \) \)} \)
      unfolding InvariantConsistent-def
      unfolding InvariantUniq-def
      using assertLiteralEffect[\texttt{of state hd ?clause'} \texttt{False}]
      by (auto simp add: Let-def)
  qed
moreover
have InvariantUniqQ (getQ \texttt{?state'})
  using \texttt{assms}
  using InvariantUniqQAfterAssertLiteral[\texttt{of state hd ?clause'} \texttt{False}]
  by (simp add: Let-def)
moreover
have currentLevel \( \texttt{(getM \texttt{?state'})} = 0 \)
  using \texttt{assms}
  using \( \sim \texttt{clauseTrue \texttt{?clause'} (elements (getM state))} \)
  using \( \sim \texttt{?clause'} = [] \)
  using assertLiteralEffect[\texttt{of state hd ?clause'} \texttt{False}]

558
unfolding \textit{addClause-def}

unfolding \textit{currentPage-def}

by (simp add: Let-def markedElementsAppend)

moreover

hence \textit{InvariantGetReasonIsReason} (\textit{getReason} ?state') (\textit{getF} ?state') (\textit{getM} ?state') (\textit{set} (\textit{getQ} ?state'))

unfolding \textit{InvariantGetReasonIsReason-def}

using \textit{elementLevelLeqCurrentLevel} [af - \textit{getM} ?state']

by auto

moreover

have var \{hd ?clause'\} \in \textit{vars} F0

using \langle ?clause' \neq [] \rangle

using \textit{hd-in-set}[of ?clause']

using \langle \textit{vars} ?clause' \subseteq \textit{vars} F0 \rangle

using \textit{clauseContainsItsLiteralsVariable}[of hd ?clause' ?clause']

by auto

hence \textit{InvariantVarsQ} (\textit{getQ} ?state') F0 Vbl

\textit{InvariantVarsM} (\textit{getM} ?state') F0 Vbl

\textit{InvariantVarsF} (\textit{getF} ?state') F0 Vbl

using \textit{InvariantWatchListsContainOnlyClausesFromF} (\textit{getWatchList} state) (\textit{getF} state)

using \textit{InvariantWatchesEl} (\textit{getF} state) (\textit{getWatch1} state)

(\textit{getWatch2} state):

using \textit{InvariantWatchListsUniq} (\textit{getWatchList} state)

using \textit{InvariantWatchListsCharacterization} (\textit{getWatchList} state) (\textit{getWatch1} state) (\textit{getWatch2} state):

using \textit{InvariantWatchesDiffer} (\textit{getF} state) (\textit{getWatch1} state)

(\textit{getWatch2} state):

using \textit{InvariantWatchCharacterization} (\textit{getF} state) (\textit{getWatch1} state) (\textit{getWatch2} state) (\textit{getM} state)

using \textit{InvariantVarsF} (\textit{getF} state) F0 Vbl;

using \textit{InvariantVarsM} (\textit{getM} state) F0 Vbl;

using \textit{InvariantVarsQ} (\textit{getQ} state) F0 Vbl;

using \langle \textit{consistent} (\textit{elements} (\textit{getM} ?state')) \rangle

using \langle \textit{uniq} (\textit{elements} (\textit{getM} ?state')) \rangle

using \textit{assertLiteralEffect}[of state hd ?clause' False]

using \textit{varsAppendValuation}[of elements (\textit{getM} state) [hd ?clause']]

using \textit{InvariantVarsQAAfterAssertLiteral}[of state hd ?clause' False F0 Vbl]

unfolding \textit{InvariantVarsM-def}

unfolding \textit{InvariantConsistent-def}

unfolding \textit{InvariantUniq-def}

by (auto simp add: Let-def)

moreover

have \textit{exhaustiveUnitPropagate-dom} ?state'

using \textit{exhaustiveUnitPropagateTermination}[of ?state' F0 Vbl]

using \textit{InvariantUniqQ} (\textit{getQ} ?state')

using \textit{InvariantWatchListsContainOnlyClausesFromF} (\textit{getWatchList} state) (\textit{getF} state)

(\textit{getWatch1} state) (\textit{getWatch2} state) (\textit{getM} state) (\textit{getQ} state) (\textit{getWatch1} state) (\textit{getWatch2} state) (\textit{getM} state) (\textit{getQ} state)

using \textit{assertLiteralEffect}[of state hd ?clause' False]

using \textit{varsAppendValuation}[of elements (\textit{getM} state) [hd ?clause']]
\( ?\text{state}' \) (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')⟩

using ⟨InvariantQCharacterization (getConflictFlag ?state') (getF ?state') (getM ?state')⟩

using ⟨InvariantWatchCharacterization (getF ?state') (getWatch1 ?state') (getWatch2 ?state')⟩

using ⟨InvariantConflictFlagCharacterization (getConflictFlag ?state') (getF ?state')⟩

using ⟨consistent (elements (getM ?state'))⟩

using ⟨uniq (elements (getM ?state'))⟩

using ⟨finite Vbl⟩

using ⟨InvariantVarsQ (getQ ?state') F0 Vbl⟩

using ⟨InvariantVarsM (getM ?state') F0 Vbl⟩

using ⟨InvariantVarsF (getF ?state') F0 Vbl⟩

unfolding InvariantConsistent-def

unfolding InvariantUniq-def

by simp

ultimately

show ?thesis

using ⟨exhaustiveUnitPropagate-dom ?state⟩

using ⟨InvariantsAfterExhaustiveUnitPropagate[of ?state]⟩

using ⟨InvariantConflictClauseCharacterizationAfterExhaustivePropagate[of ?state]⟩

using ⟨conflictFlagOrQEmptyAfterExhaustiveUnitPropagate[of ?state]⟩

using ⟨exhaustiveUnitPropagatePreservesCurrentLevel[of ?state]⟩

using ⟨InvariantGetReasonIsReasonAfterExhaustiveUnitPropagate[of ?state]⟩

using assms

using ⟨assertLiteralEffect[of state hd ?clause' False]⟩

unfolding InvariantConsistent-def

unfolding InvariantUniq-def

by (auto simp only:Let-def)

next

case False

thus ?thesis

proof ⟨cases clauseTautology ?clause'⟩

case True

thus ?thesis

using assms

using ⟨\neg ?clause' = []⟩

using ⟨\neg clauseTrue ?clause' (elements (getM state))⟩

using ⟨length ?clause' \neq 1⟩

560
next
  case False
  from ( \( ?\text{clause} \neq [] \) \( \text{length} \ ?\text{clause} \neq 1 \) )
  have \( \text{length} \ ?\text{clause} > 1 \)
    by (induct (\( ?\text{clause} \)) auto)

  hence \( \text{nth} \ ?\text{clause} \neq \text{nth} \ ?\text{clause} \neq 1 \)
  using \text{distinct-remdups}[\text{of} \ ?\text{clause}] \text{of} \ ?\text{clause} \neq 0 1\)
  using (\( ?\text{clause} \neq [] \))
  by auto

  let \(?\text{state}' = \text{let clauseIndex} = \text{length} \ (\text{getF} \ ?\text{state}) \ in \text{let state}' = \text{state} \ (| \text{getF} := (\text{getF} \ ?\text{state}) @ \ ?\text{clause} | ) |)
    in let state'' = \text{setWatch1} \text{clauseIndex} \text{(nth} \ ?\text{clause} \neq 1) \text{ in \text{state}''in}

  have \text{InvariantWatchesEl} \ (\text{getF} \ ?\text{state}') \ (\text{getWatch1} \ ?\text{state}') \ (\text{getWatch2} \ ?\text{state}')
    using \text{InvariantWatchesEl} \ (\text{getF} \ ?\text{state}) \ (\text{getWatch1} \ ?\text{state}) \ (\text{getWatch2} \ ?\text{state})
    using \text{InvariantWatchesEl-def}\text{setWatch2-def}
    by (auto simp add: \text{Let-def nth-append})

  moreover
  have \text{InvariantWatchesDiffer} \ (\text{getF} \ ?\text{state}') \ (\text{getWatch1} \ ?\text{state}') \ (\text{getWatch2} \ ?\text{state}')
    using \text{InvariantWatchesDiffer} \ (\text{getF} \ ?\text{state}) \ (\text{getWatch1} \ ?\text{state}) \ (\text{getWatch2} \ ?\text{state})
    using \text{InvariantWatchesDiffer-def}\text{setWatch1-def}
    unfolding \text{setWatch2-def}
    by (auto simp add: \text{Let-def nth-append})

  moreover
  have \text{InvariantWatchListsContainOnlyClausesFromF} \ (\text{getWatchList} \ ?\text{state}') \ (\text{getF} \ ?\text{state}')
    using \text{InvariantWatchListsContainOnlyClausesFromF-def}
(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)
unfolding InvariantWatchCharacterization-def
using (\langle \text{nth} \ (\text{getF} \ ?\text{state'}) \ c \rangle = (\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 \langle \text{0} \leq c \land c < \text{length} \ (\text{getF} \ ?\text{state'}) \rangle
have \langle \text{length} \ (\text{getF} \ \text{state}) \rangle
unfolding InvariantWatchesEl-def
unfolding setWatch2-def
by (auto simp add: Let-def)
from \langle \text{InvariantWatchesEl} \ (\text{getF} \ ?\text{state'}) \ (\text{getWatch1} \ ?\text{state'}) \ (\text{getWatch2} \ ?\text{state'}) \rangle
obtain w1 w2
where \langle w1 \text{ el ?clause'} \ w2 \text{ el ?clause'} \rangle
\langle \text{getWatch1} \ ?\text{state'} \ (\text{length} \ (\text{getF} \ \text{state})) = \text{Some} \ w1 \rangle
\langle \text{getWatch2} \ ?\text{state'} \ (\text{length} \ (\text{getF} \ \text{state})) = \text{Some} \ w2 \rangle
unfolding InvariantWatchesEl-def
unfolding setWatch2-def
unfolding setWatch1-def
by (auto simp add: Let-def)
hence \langle w1 = \text{www1} \rangle \ \langle w2 = \text{www2} \rangle
using \langle \text{Some} \ \text{www1} = (\text{getWatch1} \ ?\text{state'}) \ \text{c} \rangle \ \langle \text{Some} \ \text{www2} = (\text{getWatch2} \ ?\text{state'}) \ \text{c} \rangle
using \langle \text{c} = \text{length} \ (\text{getF} \ \text{state}) \rangle
by auto
have \langle \text{\neg literalFalse} \ w1 \ (\text{elements} \ (\text{getM} \ ?\text{state'})) \ \rangle
\langle \text{\neg literalFalse} \ w2 \ (\text{elements} \ (\text{getM} \ ?\text{state'})) \ \rangle
using \langle w1 \text{ el ?clause'} \ w2 \text{ el ?clause'} \rangle
using *
unfolding setWatch2-def
unfolding setWatch1-def
by (auto simp add: Let-def)
thus \langle \text{\neg ?thesis} \ \langle w1 = \text{www1} \rangle \ \langle w2 = \text{www2} \rangle \ \rangle
unfolding watchCharacterizationCondition-def
unfolding setWatch2-def
unfolding setWatch1-def
by (auto simp add: Let-def)
qed

} thus \langle \text{\neg ?thesis} \ \rangle
unfolding InvariantWatchCharacterization-def
by auto
qed
moreover
have \text{\forall \ l. 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′)

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 \$\text{thesis}\$
    using \(\langle a1 \text{ el } ?\text{clause}'\rangle\)
    by auto
  qed
qed
then obtain \(l'::\text{Literal}\)
  where \(l \neq l' \text{ el } ?\text{clause}'\)
  by auto
with *
  have \(\neg \text{literalFalse } l' \text{ (elements } (\text{getM state}))\)
    by simp
  hence False
    using \(\text{isUnitClause } ?\text{clause}' l \text{ (elements } (\text{getM state}));\)
    using \(\langle l \neq l' \text{ el } ?\text{clause}'\rangle\)
    unfolding \(\text{isUnitClause-def}\)
    by auto
  }
  thus \$\text{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 \(\text{InvariantConflictClauseCharacterization } \text{(getConflictFlag state)} \text{(getConflictClause state)} \text{(getF state } @ [?\text{clause}']) \text{(getM state)}\)
proof (cases \text{getConflictFlag state})
  case False
  thus \$\text{thesis}\$
    unfolding \(\text{InvariantConflictClauseCharacterization-def}\)
    by simp
next
  case True
  hence \text{getConflictClause state } < \text{length } (\text{getF state})
    using \(\text{InvariantConflictClauseCharacterization } \text{(getConflictFlag state)} \text{(getConflictClause state)} \text{(getF state)} \text{(getM state)}\)
    unfolding \(\text{InvariantConflictClauseCharacterization-def}\)
    by (auto simp add: \(\text{Let-def}\))
  hence \(\text{nth } ((\text{getF state} } @ [?\text{clause}']) \text{(getConflictClause state)} = \text{nth } (\text{getF state}) \text{(getConflictClause state)}\)
    by (simp add: \(\text{nth-append}\))
  thus \$\text{thesis}\$
    using \(\text{InvariantConflictClauseCharacterization } \text{(getConflictFlag state)} \text{(getConflictClause state)} \text{(getF state)} \text{(getM state)}\)

565
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
InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state) and
InvariantWatchCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state) (getM state)
and
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′) Phi′′) ∨
(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 \(\text{cases clauseTrue } ?\text{clause'} (\text{elements (getM state)})\)
case True
show \(?thesis
proof-
from True
have clauseTrue clause (\text{elements (getM state)})
using clauseTrueRemoveDuplicateLiterals
[of removeFalseLiterals clause (\text{elements (getM state)}) elements (getM state)]
using clauseTrueRemoveFalseLiterals
[of elements (getM state) clause]
using \(\text{InvariantConsistent (getM state)}\)
unfolding \(\text{InvariantConsistent-def}\)
by simp
show \(?thesis
proof \(\text{cases getSATFlag state = UNDEF}\)
case True
thus \(?thesis
using *
using \(\text{clauseTrue clause (elements (getM state))}\)
using \(\text{getM state = prefixToLevel 0 (getM state)}\)
using satisfiedClauseCanBeRemoved
[of getF state (elements (prefixToLevel 0 (getM state))) Phi clause]
using \(\text{clauseTrue ?\text{clause'} (elements (getM state)})\)
unfolding addClause-def
unfolding \(\text{InvariantEquivalentZL-def}\)
by auto
next
case False
thus \(?thesis
using *
using \(\text{clauseTrue ?\text{clause'} (elements (getM state)})\)
using \(\text{satisfiableAppend[of Phi [clause]]}\)
unfolding addClause-def
by force
qed
qed
next
case False
show \(?thesis
proof \(\text{cases ?\text{clause'} = []}\)
case True
show \(?thesis
proof \(\text{cases getSATFlag state = UNDEF}\)
case True

568
thus \( \text{thesis} \)

using *

using falseAndDuplicateLiteralsCanBeRemoved

\[
\text{of getF state (elements (prefixToLevel 0 (getM state))) [] Phi clause}
\]

using \( \text{getM state = prefixToLevel 0 (getM state)} \)

using formulaWithEmptyClauseIsUnsatisfiable\[ \text{of (getF state @ val2form (elements (getM state)) @ [][]) \]

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 *

unfolding satisfiableAppend\[ \text{of Phi [clause]} \]

by force

qed

next
case False

thus \( \text{thesis} \)

proof (cases length ?\text{clause}' = 1)

case True

from \( \langle \text{length ?\text{clause}' = 1} \rangle \)

have \( \langle \text{hd ?\text{clause}' = ?\text{clause}' \rangle \)

using lengthOneCharacterisation\[ \text{of ?\text{clause}' \rangle} \]

by simp

with \( \langle \text{length ?\text{clause}' = 1} \rangle \)

have val2form (elements (getM state)) @ [?\text{clause}'] = val2form \((\text{elements (getM state)}) @ ?\text{clause}' \]

using val2formAppend\[ \text{of elements (getM state) ?\text{clause}' \]

using val2formOfSingleLiteralValuation\[ \text{of ?\text{clause}' \]

by auto

let \( \text{state}' = \text{assertLiteral (hd ?\text{clause}') False state} \)

have addClause clause state = exhaustiveUnitPropagate ?state'

using (\( \sim \text{clauseTrue ?\text{clause}' (elements (getM state))} \)

using \( \langle \sim ?\text{clause}' = [] \rangle \)

using \( \langle \text{length ?\text{clause}' = 1} \rangle \)

unfolding addClause-def

by (simp add: Let-def)

moreover

from \( \langle \text{?\text{clause}' \neq []} \rangle \)
have \( \text{hd} \ ?\text{clause}' \in \text{set} \ ?\text{clause}' \)
using \text{hd-in-set}{[\text{of} \ ?\text{clause}']}
by simp
with **
have \( \neg \text{literalFalse} \ (\text{hd} \ ?\text{clause}') \ (\text{elements} \ (\text{getM} \ \text{state})) \)
by simp
hence consistent \((\text{elements} \ ((\text{getM} \ \text{state}) @ [(\text{hd} \ ?\text{clause}', False)])\))
using assms
unfolding \text{InvariantConsistent-def} 
using \text{consistentAppendElement}{[\text{of} \ \text{elements} \ ((\text{getM} \ \text{state}) \ \text{hd} \ ?\text{clause}')]}
by simp
hence consistent \((\text{elements} \ ((\text{getM} \ \text{state}'))\))
using assms
using \text{assertLiteralEffect}{[\text{of} \ \text{state} \ \text{hd} \ ?\text{clause}' \ False]}
by simp
moreover
from \( \neg \text{clauseTrue} \ ?\text{clause}' \ (\text{elements} \ (\text{getM} \ \text{state}')) \)
have \text{uniq} \((\text{elements} \ ((\text{getM} \ \text{state}')))\)
using assms
using \text{assertLiteralEffect}{[\text{of} \ \text{state} \ \text{hd} \ ?\text{clause}' \ False]}
using \( \text{hd} \ ?\text{clause}' \in \text{set} \ ?\text{clause}' \)
unfolding \text{InvariantUniq-def} 
by (simp add: \text{uniqAppendIff clauseTrueIffContainsTrueLiteral})
moreover
have \text{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}') \ and 
\text{InvariantWatchesEl} \ ((\text{getF} \ \text{state}') \ (\text{getWatch1} \ \text{state}') \ (\text{getWatch2} \ \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{getWatch1} \ \text{state}') \ (\text{getWatch2} \ \text{state}') 
using assms 
using \text{WatchInvariantsAfterAssertLiteral}{[\text{of} \ \text{state} \ \text{hd} \ ?\text{clause}']} 
by (auto simp add: \text{Let-def})
moreover
have \text{InvariantWatchCharacterization} \ ((\text{getF} \ \text{state}') \ (\text{getWatch1} \ \text{state}') \ (\text{getWatch2} \ \text{state}') \ (\text{getM} \ \text{state}') 
using assms 
using \text{InvariantWatchCharacterizationAfterAssertLiteral}{[\text{of} \ \text{state} \ \text{hd} \ ?\text{clause}' \ False]}
using \( \text{uniq} \ ((\text{elements} \ ((\text{getM} \ \text{state}')))) 
using \( \text{consistent} \ ((\text{elements} \ ((\text{getM} \ \text{state}')))) 
unfolding \text{InvariantConsistent-def} 
unfolding \text{InvariantUniq-def}
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]
proof (cases getConflictFlag state)
case True
hence getConflictFlag ?state'
using assms
using `assertLiteralConflictFlagEffect`[of state hd ?clause' False]
using (uniq (elements (getM ?state')))
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]
used 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']

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 var ?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))
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 \( ?\text{state} \)′
using exhaustiveUnitPropagateTermination[of \( ?\text{state} \)′ F0 Vbl]
using \( \langle \text{InvariantUniqQ} \ (\text{getQ} \ ?\text{state} \)′) \rangle 
using \( \langle \text{InvariantWatchListsContainOnlyClausesFromF} \ (\text{getWatchList} \ ?\text{state} \)′) \ (\text{getF} \ ?\text{state} \)′) \ (\text{getWatch1} \ ?\text{state} \)′) \ (\text{getWatch2} \ ?\text{state} \)′) \rangle 
using \( \langle \text{InvariantWatchListsUniq} \ (\text{getWatchList} \ ?\text{state} \)′) \rangle 
using \( \langle \text{InvariantWatchListsCharacterization} \ (\text{getWatchList} \ ?\text{state} \)′) \ (\text{getWatch1} \ ?\text{state} \)′) \ (\text{getWatch2} \ ?\text{state} \)′) \rangle 
using \( \langle \text{InvariantWatchesEl} \ (\text{getF} \ ?\text{state} \)′) \ (\text{getWatch1} \ ?\text{state} \)′) \ (\text{getWatch2} \ ?\text{state} \)′) \rangle 
using \( \langle \text{InvariantWatchesDiffer} \ (\text{getF} \ ?\text{state} \)′) \ (\text{getWatch1} \ ?\text{state} \)′) \ (\text{getWatch2} \ ?\text{state} \)′) \rangle 
using \( \langle \text{InvariantConflictFlagCharacterization} \ (\text{getConflictFlag} \ ?\text{state} \)′) \ (\text{getF} \ ?\text{state} \)′) \ (\text{getM} \ ?\text{state} \)′) \rangle 
using \( \langle \text{InvariantVarConsistent-def} \rangle 
by simp
moreover
have \( \neg \text{clauseTautology} \) \( ?\text{clause} \)′
proof
\{
  assume \( \neg \ ?\text{thesis} \)
  then obtain \( l′ \)
  where \( l′ \) el \( ?\text{clause} \) opposite \( l′ \) el \( ?\text{clause} \)
  by (auto simp add: clauseTautologyCharacterization)
  have False
  proof (cases \( l′ \) el \( ?\text{clause} \)′)
    case True
    have opposite \( l′ \) el \( ?\text{clause} \)′
    proof
      \{
        assume \( \neg \ ?\text{thesis} \)
        hence literalFalse \( l′ \) (elements \( (\text{getM} \ ?\text{state}) \))
        using \( l′ \) el \( ?\text{clause} \)
        using \( \langle \text{opposite} \ l′ \ el \ ?\text{clause} \rangle \)
        using \( \neg \text{clauseTrue} \ ?\text{clause}′ \ (\text{elements} \ (\text{getM} \ ?\text{state})) \)
        using clauseTrueIffContainsTrueLiteral[of \( ?\text{clause}′ \) elements \( (\text{getM} \ ?\text{state}) \)]
      unfolding removeFalseLiterals-def
    \}
  \}
\}

573
by auto

hence False
using \( l' \) el ?clause''
unfolding removeFalseLiterals-def
by auto

} thus ?thesis
by auto

qed

have \( \forall \ x. \ x \ el \ ?\text{clause}' \rightarrow x = l' \)
using \( l' \) el ?clause''
using \( \langle \text{length} \ ?\text{clause}' = 1 \rangle \)
using lengthOneImpliesOnlyElement[of \( ?\text{clause}' \)]
by simp
thus ?thesis
using \( \langle \text{opposite} l' \ el \ ?\text{clause}' \rangle \)
by auto

next
case False
hence literalFalse \( l' \) (elements (getM state))
using \( l' \) el clause''
unfolding removeFalseLiterals-def
by simp
hence \( \neg \) literalFalse (opposite \( l' \)) (elements (getM state))
using \( \langle \text{InvariantConsistent} (getM state) \rangle \)
unfolding InvariantConsistent-def
by (auto simp add: inconsistentCharacterization)
hence opposite \( l' \) el ?clause''
using \( \langle \text{opposite} l' \ el \ ?\text{clause}' \rangle \)
unfolding removeFalseLiterals-def
by auto
thus ?thesis
using \( \langle \text{literalFalse} \ l' \ (\text{elements (getM state)}) \rangle \)
using \( \langle \neg \, \text{clauseTrue} \ ?\text{clause}' \ (\text{elements (getM state)}) \rangle \)
by (simp add: clauseTrueIffContainsTrueLiteral)
qed

} thus ?thesis
by auto

qed

moreover

note clc = calculation

show ?thesis

proof (cases getSATFlag state = UNDEF)
case True
hence InvariantEquivalentZL (getF state) (getM state) Phi
using assms
by simp

hence 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
hence \texttt{getSATFlag (exhaustiveUnitPropagate ?state')} = \texttt{FALSE}

using clc
using exhaustiveUnitPropagatePreservedVariables[of ?state']
by (auto simp only: Let-def)
moreover
have \texttt{\neg satisfiable (Phi @ [clause])}
using satisfiableAppend[of Phi [clause]]
using \texttt{(\neg satisfiable Phi)}
by auto
ultimately
show \texttt{?thesis}
using clc
using \texttt{(\neg clauseTautology clause)}
by (simp only: Let-def) simp

qed
next
case \texttt{False}
thus \texttt{?thesis}
proof (cases clauseTautology ?clause')
  case \texttt{True}
  moreover
  hence \texttt{clauseTautology clause}
  unfolding removeFalseLiterals-def
  by (auto simp add: clauseTautologyCharacterization)
  ultimately
  show \texttt{?thesis}
  using *
  using \texttt{(\neg ?clause' = [])}
  using \texttt{(\neg clauseTrue ?clause' (elements (getM state)))}
  using \texttt{(length ?clause' \neq 1)}
  using satisfiableAppend[of Phi [clause]]
  unfolding addClause-def
  by (auto simp add: Let-def)
next
case \texttt{False}
have \texttt{\neg clauseTautology clause}
proof-
{
  assume \texttt{\neg ?thesis}
  then obtain \texttt{l'}
  where \texttt{l' el clause opposite l' el clause}
  by (auto simp add: clauseTautologyCharacterization)
  have \texttt{False}
  proof (cases \texttt{l' el ?clause'})
    case \texttt{True}
    hence \texttt{\neg opposite l' el ?clause'}
    using \texttt{(\neg clauseTautology ?clause')}

576
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)))]

  using (getM state = prefixToLevel 0 (getM state));
  using (¬ ?clause' = []);
  using (¬ clauseTrue ?clause' (elements (getM state)));
  using (length ?clause' ≠ 1);
  using (¬ clauseTautology ?clause');
  using (¬ clauseTautology ?clause);
  using (getSATFlag state = UNDEF);
unfolding addClause-def
unfolding InvariantEquivalentZL-def
unfolding setWatch1-def
unfolding setWatch2-def
using clauseOrderIrrelevant[of getF state [?clause'] val2form
(elements (getM state))]
using equivalentFormulaeTransitivity[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
qed
qed

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)
  and
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
  and
  InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)
and

578
InvariantWatchCharacterization \( (\text{getF state}) \) \( (\text{getWatch1 state}) \) \( (\text{getWatch2 state}) \) \( (\text{getM state}) \)

InvariantConflictFlagCharacterization \( (\text{getConflictFlag state}) \) \( (\text{getF state}) \) \( (\text{getM state}) \)

InvariantConflictClauseCharacterization \( (\text{getConflictFlag state}) \) \( (\text{getConflictClause state}) \) \( (\text{getF state}) \) \( (\text{getM state}) \)

InvariantQCharacterization \( (\text{getConflictFlag state}) \) \( (\text{getQ state}) \) \( (\text{getF state}) \) \( (\text{getM state}) \)

InvariantGetReasonIsReason \( (\text{getReason state}) \) \( (\text{getF state}) \) \( (\text{getM state}) \)

InvariantUniqQ \( (\text{getQ state}) \)

\( \text{currentLevel} (\text{getM state}) = 0 \)

finite \( \text{Vbl} \)

InvariantConsistent \( (\text{getM state}') \) \( \land \)

InvariantUniq \( (\text{getM state}') \) \( \land \)

InvariantWatchListsContainOnlyClausesFromF \( (\text{getWatchList state}') \) \( (\text{getF state}') \) \( (\text{getM state}') \)

InvariantWatchListsUniq \( (\text{getWatchList state}') \) \( (\text{getWatch1 state}') \) \( (\text{getWatch2 state}') \)

InvariantWatchesEl \( (\text{getF state}') \) \( (\text{getWatch1 state}') \) \( (\text{getWatch2 state}') \)

InvariantWatchesDiffer \( (\text{getF state}') \) \( (\text{getWatch1 state}') \) \( (\text{getWatch2 state}') \)

InvariantWatchCharacterization \( (\text{getF state}') \) \( (\text{getWatch1 state}') \) \( (\text{getWatch2 state}') \) \( (\text{getM state}') \)

InvariantConflictFlagCharacterization \( (\text{getConflictFlag state}') \) \( (\text{getF state}') \) \( (\text{getM state}') \)

InvariantConflictClauseCharacterization \( (\text{getConflictFlag state}') \) \( (\text{getConflictClause state}') \) \( (\text{getF state}') \) \( (\text{getM state}') \)

InvariantQCharacterization \( (\text{getConflictFlag state}') \) \( (\text{getQ state}') \) \( (\text{getF state}') \) \( (\text{getM state}') \)

InvariantGetReasonIsReason \( (\text{getReason state}') \) \( (\text{getF state}') \) \( (\text{getM state}') \)

InvariantUniqQ \( (\text{getQ state}') \)

\( \text{currentLevel} (\text{getM state}') = 0 \) (is ?Inv state')

using assms

**proof** (induct Phi arbitrary: state)
case Nil
thus \( ?\text{case} \)
  by simp
next
  case \( (\text{Cons clause } \Phi') \)
  let \( ?\text{state}' = \text{addClause clause } \text{state} \)
  have \( ?\text{Inv } ?\text{state}' \)
    using Cons
    using InvariantsAfterAddClause[of state \( F_0 \) Vbl clause]
    using formulaContainsItsClausesVariables[of clause \( F_0 \)]
    by (simp add: Let-def)
  thus \( ?\text{case} \)
    using Cons(1)[of \(?\text{state}'\)] :finite Vbl \( \text{Cons}(18) \text{Cons}(19) \text{Cons}(20) \text{Cons}(21) \text{Cons}(22) \)
    by (simp add: Let-def)
qed

lemma InvariantEquivalentZLAfterInitializationStep:
  fixes \( \Phi :: \text{Formula} \)
  assumes
    \( \text{getSATFlag state} = \text{UNDEF} \land \text{InvariantEquivalentZL } (\text{getF state}) \) (getM state) (filter \( (\lambda \text{ c. } \neg \text{clauseTautology } \text{c}) \Phi) \) \( \lor \)
    \( (\text{getSATFlag state} = \text{FALSE} \land \neg \text{satisfiable} (\text{filter } (\lambda \text{ c. } \neg \text{clauseTautology } \text{c}) \Phi)) \)
    \text{InvariantConsistent } (\text{getM state})
    \text{InvariantUniq } (\text{getM state})
    \text{InvariantWatchListsContainOnlyClausesFromF } (\text{getWatchList state})
    (\text{getF state}) \text{ and }
    \text{InvariantWatchListsUniq } (\text{getWatchList state}) \text{ and }
    \text{InvariantWatchListsCharacterization } (\text{getWatchList state}) (\text{getWatch1 state}) (\text{getWatch2 state})
    \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 }
    \text{InvariantWatchCharacterization } (\text{getF state}) (\text{getWatch1 state}) (\text{getWatch2 state}) (\text{getM state})
    \text{InvariantConflictFlagCharacterization } (\text{getConflictFlag state}) (\text{getF state}) (\text{getM state})
    \text{InvariantConflictClauseCharacterization } (\text{getConflictFlag state}) (\text{getConflictClause state}) (\text{getF state}) (\text{getM state})
    \text{InvariantQCharacterization } (\text{getConflictFlag state}) (\text{getQ state}) (\text{getF state}) (\text{getM state})
    \text{InvariantNoDecisionsWhenConflict } (\text{getF state}) (\text{getM state}) (\text{currentLevel } (\text{getM state}))
    \text{InvariantNoDecisionsWhenUnit } (\text{getF state}) (\text{getM state}) (\text{currentLevel } (\text{getM state}))
    \text{InvariantGetReasonIsReason } (\text{getReason state}) (\text{getF state}) (\text{getM state}) (\text{set } (\text{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') (?filt ?Phi''') ∨
  getSATFlag ?state' = FALSE ∧ ¬ satisfiable (?filt ?Phi''')
  using formulaContainsItsClausesVariables[of clause F0]
  using Invariant EquivalentZL AfterAddClause[of state ?filt Phi F0 Vbl clause]
  by (simp add: Let-def)

hence getSATFlag ?state' = UNDEF ∧ InvariantEquivalentZL (getF ?state') (getM ?state') (filter (Phi @ [clause])) ∨
getSATFlag ?state' = FALSE ∧ ¬ satisfiable (?filt (Phi @ [clause]))
  by auto
moreover
from Cons
have Invariant Consistent (getM ?state') ∧
  InvariantUniq (getM ?state') ∧
  InvariantWatchLists Contain Only Clauses From F (getWatchList ?state') (getF ?state') ∧
  InvariantWatchLists Uniq (getWatchList ?state') ∧
  InvariantWatchLists Characterization (getWatchList ?state') (getWatch1 ?state') (getWatch2 ?state') ∧
  Invariant Watches El (getF ?state') (getWatch1 ?state') (getWatch2
INVARIANT WATCHES DIFFER
(getF ?state') (getWatch1 ?state') (getWatch2 ?state') ∧
INVARIANT WATCH CHARACTERIZATION (getF ?state') (getWatch1 ?state')
(getWatch2 ?state') (getM ?state') ∧
INVARIANT CONFLICT FLAG CHARACTERIZATION (getConflictFlag ?state') (getF ?state')
(getM ?state') ∧
INVARIANT CONFLICT CLAUSE CHARACTERIZATION (getConflictClause ?state')
(getConflictFlag ?state') (getF ?state') (getM ?state') ∧
INVARIANT Q CHARACTERIZATION (getConflictFlag ?state') (getQ ?state')
(getF ?state') (getM ?state') ∧
INVARIANT GET REASON IS REASON (getReason ?state') (getF ?state')
(getM ?state') ∧
INVARIANT NO UNIT (getConflictFlag ?state') (getQ ?state') (getF ?state')
(getM ?state') ∧
INVARIANT VARS M (getM ?state') (getConflictFlag ?state') (getQ ?state') (getF ?state')
(getM ?state') (currentLevel ?state') = 0
using formulaContainsItsClausesVariables [of clause F0]
using InvariantsAfterAddClause
by (simp add: Let-def)
moreover hence INVARIANT NO DECISIONS WHEN CONFLICT (getF ?state') (getM ?state')
(currentLevel ?state')
INVARIANT NO DECISIONS WHEN UNIT (getF ?state') (getM ?state') (currentLevel ?state')
unfolding INVARIANT NO DECISIONS WHEN CONFLICT-def
unfolding INVARIANT NO DECISIONS WHEN UNIT-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

LEMMATA INVARIANTS AFTER INITIALIZATION:
shows
let state' = (initialize F0 initialState) in
INVARIANT CONSISTENT (getM state') ∧
INVARIANT VARS M (getM state') (getConflictFlag state') (getQ state') (getF state')
(currentLevel state') = 0
INVARIANT WATCH LIST CHARACTERIZATION (getWatchList state') (getWatchList state') (getWatch1 state')
(getWatch2 state') (getM state') ∧
INVARIANT WATCHES EL (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') ∧
   .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') ∧
   .InvariantVarsM (getM state') F0 {} ∧
   .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 state' = (initialize $F_0$ initialState) in
let $F_0'$ = (filter ($\lambda c. \neg \text{clauseTautology } c$) $F_0$) in
(getSATFlag state') = UNDEF \land \text{InvariantEquivalentZL (getF state')} (getM state') $F_0$)

\( \text{getSATFlag state'} = \text{FALSE} \land \neg \text{satisfiable } F_0' \)

using \text{InvariantEquivalentZLAfterInitializationStep[of initialState]}
\text{F0 F0}]

unfolding \text{initialState-def}
unfolding \text{InvariantEquivalentZL-def}
unfolding \text{InvariantConsistent-def}
unfolding \text{InvariantUniq-def}
unfolding \text{InvariantWatchesEl-def}
unfolding \text{InvariantWatchesDiffer-def}
unfolding \text{InvariantWatchListsContainOnlyClausesFromF-def}
unfolding \text{InvariantWatchListsUniq-def}
unfolding \text{InvariantWatchListsCharacterization-def}
unfolding \text{InvariantConflictClauseCharacterization-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 \( M \))
shows \( \text{clauseFalse} \ c \ (\text{elements} \ (\text{prefixToLevel} \ (\text{elementLevel} \ l \ M) \ M)) \)

proof
{
  fix \( l'::\text{Literal} \)
  assume \( l' \ el \ c \)
  hence \( \text{literalFalse} \ l' \ (\text{elements} \ M) \)
    using \( \langle \text{clauseFalse} \ c \ (\text{elements} \ M) \rangle \)
    by (simp add: clauseFalseIffAllLiteralsAreFalse)
  hence \( \text{literalTrue} \ (\text{opposite} \ l') \ (\text{elements} \ M) \)
    by simp

  have \( \text{opposite} \ l' \ el \ \text{oppositeLiteralList} \ c \)
    using \( \langle \text{l'} \ el \ c \rangle \)
    using literalElListIffOppositeLiteralElOppositeLiteralList[of l' c]
    by simp

  have \( \text{elementLevel} \ (\text{opposite} \ l') \ M \leq \text{elementLevel} \ l \ M \)
    using lastAssertedLiteralHasHighestElementLevel[of l oppositeLiteralList c M]
    using \( \langle \text{isLastAssertedLiteral} \ l \ (\text{oppositeLiteralList} \ c) \ (\text{elements} \ M) \rangle \)
    using \( \langle \text{uniq} \ (\text{elements} \ M) \rangle \)
    using \( \langle \text{opposite} \ l' \ el \ \text{oppositeLiteralList} \ c \rangle \)
    by auto
  hence \( \text{opposite} \ l' \ el \ (\text{elements} \ (\text{prefixToLevel} \ (\text{elementLevel} \ l \ M) \ M)) \)
    using elementLevelLtLevelImpliesMemberPrefixToLevel[of opposite l' M elementLevel l M]
    using \( \langle \text{literalTrue} \ (\text{opposite} \ l') \ (\text{elements} \ M) \rangle \)
    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 \( \neq [] \land \)
  (let \( Cl = \text{getLastAssertedLiteral} \ (\text{oppositeLiteralList} \ \text{clause}) \ (\text{elements} \ M) \) in
    InvariantClCurrentLevel Cl M)
proof -
  have clause \neq []
proof -
  {
    assume \neg \thesis
    hence clauseFalse clause (elements (prefixToLevel ((currentLevel M) - 1) M))
        by simp
    hence False
      using \langle InvariantNoDecisionsWhenConflict F M (currentLevel M) \rangle
      using \langle currentLevel M > 0 \rangle
      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 \leq currentLevel M
      using elementLevelLeqCurrentLevel[of ?Cl M]
      by simp
moreover
have elementLevel ?Cl M \geq 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 \neq []);
      by simp
    hence clauseFalse clause (elements (prefixToLevel (elementLevel ?Cl M) M))
      using clauseFalseInPrefixToLastAssertedLiteral[of ?Cl clause M]
      using (clauseFalse clause (elements M))
      using (uniq (elements M))
      by simp
    hence False
      using (clause el F)
      using \langle InvariantNoDecisionsWhenConflict F M (currentLevel M) \rangle
  }
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') ∧
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') ∧
InvariantUniqC (getC state')
proof—
let ?M0 = elements (prefixToList 0 (getM state))
let ?oppM0 = oppositeLiteralList ?M0
let ?clause' = nth (getF state) (getConflictClause state)
let ?clause'' = list-diff ?clause' ?oppM0
let ?clause = remdups ?clause''
let ?l = getLeastAssertedLiteral (oppositeLiteralList ?clause') (elements (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 InvariantNoDecisionsWhenConflictEnsuresCurrentLevelCl[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

588
using \(\langle\text{elementLevel } ?l \ (\text{getM state}) = \text{currentLevel} \ (\text{getM state})\rangle\)

using \(\langle\text{currentLevel} \ (\text{getM state}) > 0: \) by simp

\}

thus \(\text{thesis}\)

by auto

qed

hence \(\neg \text{opposite } ?l \ \text{el } ?\text{oppM0}\)

using literalElListIffOppositeLiteralElOppositeLiteralList[of \(\text{?l el elements (prefixToLevel 0 (getM state))}\)]

by simp

have \(\text{opposite } ?l \ \text{el } ?\text{clause''}\)

using \(\langle\text{opposite } ?l \ \text{el } ?\text{clause'}\rangle\)

using \(\langle\neg \text{opposite } ?l \ \text{el } ?\text{oppM0}\rangle\)

using listDiffIff[of \(\text{opposite } ?l \ \text{el } ?\text{clause'} \ \text{oppM0}\)]

by simp

hence \(\text{?l el (oppositeLiteralList ?clause''))}\)

using literalElListIffOppositeLiteralElOppositeLiteralList[of \(\text{opposite } ?l \ ?\text{clause'')}\)]

by simp

have \(\text{set (oppositeLiteralList ?clause'') } \subseteq \text{set (oppositeLiteralList ?clause')}\)

proof

fix \(x\)

assume \(x \in \text{set (oppositeLiteralList ?clause'')}\)

thus \(x \in \text{set (oppositeLiteralList ?clause')}\)

using literalElListIffOppositeLiteralElOppositeLiteralList[of \(\text{opposite } x \ ?\text{clause'')}\)]

using literalElListIffOppositeLiteralElOppositeLiteralList[of \(\text{opposite } x \ ?\text{clause'}\)]

using listDiffIff[of \(\text{opposite } x \ ?\text{clause'} \ \text{oppositeLiteralList (elements (prefixToLevel 0 (getM state)))}\)]

by auto

qed

have \(\text{isLastAssertedLiteral } ?l \ (\text{oppositeLiteralList ?clause'')} \ (\text{elements (getM state)})\)

using \(\langle\text{?l el (oppositeLiteralList ?clause'')}\rangle\)

using \(\langle\text{set (oppositeLiteralList ?clause'') } \subseteq \text{set (oppositeLiteralList ?clause')}\rangle\)

using \(\langle\text{isLastAssertedLiteral } ?l \ (\text{oppositeLiteralList ?clause'')} \ (\text{elements (getM state)})\rangle\)

using \(\langle\text{isLastAssertedLiteralSubset of } ?l \ \text{oppositeLiteralList ?clause'} \ \text{elements (getM state) oppositeLiteralList ?clause'')}\)

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 \(?F0 = (getF state) @ val2form ?M0\)

have formulaEntailsClause ?F0 ?clause'
using formulaEntailsClause (getF state) ?clause'
by (simp add: formulaEntailsClauseAppend)

hence formulaEntailsClause ?F0 ?clause''
using 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 InvariantEquivalentZL (getF state) (getM state) F0
unfolding InvariantEquivalentZL-def
unfolding formulaEntailsClause-def
unfolding equivalentFormulae-def
by auto

show \(?thesis\)
using (isLastAssertedLiteral \(?l'\) (oppositeLiteralList ?clause) (elements (getM state))
using (\(?l = \?l'\))
using (elementLevel \(?l\) (getM state) = currentLevel (getM state))
using (clauseFalse \(?clause\) (elements (getM state)))
using formulaEntailsClause F0 \(?clause\)
unfolding applyConflict-def
unfolding setConflictAnalysisClause-def
unfolding InvariantClCharacterization-def
unfolding InvariantClCurrentLevel-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
InvariantClCharacterization (getCl state) (getC state) (getM state)
InvariantClCurrentLevel (getCl state) (getM state)
InvariantCnCharacterization (getCn state) (getC state) (getM state)
  shows
(getCn state = 1) = isUIP (opposite (getCl state)) (getC state) (getM state)
proof
  let ?clls = filter (λ 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 ⟨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 literalElListIfOppositeLiteralElOppositeLiteralList[of oppo-
                   site ?Cl getC state]
    by simp

  hence opposite ?Cl el ?clls
    using ⟨InvariantClCurrentLevel (getCl state) (getM state)⟩
    unfolding InvariantClCurrentLevel-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 \texttt{assms}
using \langle \texttt{length ?clls > 0} \rangle
\textbf{unfolding} \texttt{InvariantCnCharacterization-def}
by \texttt{(simp (no-asmp))}
then obtain literal1::\texttt{Literal} and literal2::\texttt{Literal}
where literal1 el ?clls literal2 el ?clls literal1 \neq literal2
using \langle \texttt{uniq ?clls} \rangle
using \langle ?clls \neq [] \rangle
using \texttt{lengthGtOneTwoDistinctElements[of ?clls]}
by \texttt{auto}
then obtain literal::\texttt{Literal}
where literal el ?clls literal \neq opposite ?Cl
using \langle opposite ?Cl el ?clls \rangle
by \texttt{auto}
hence \neg \texttt{isUIP} (opposite ?Cl) (getC state) (getM state)
using \langle opposite ?Cl el ?clls \rangle
\textbf{unfolding} \texttt{isUIP-def}
by \texttt{auto}
\}\moreover
\}\moreover
\{\texttt{assume getCn state = 1}
hence \texttt{length ?clls = 1}
\textbf{using} \langle \texttt{InvariantCnCharacterization (getCn state) (getC state)} \rangle
\texttt{(getM state) :}
\textbf{unfolding} \texttt{InvariantCnCharacterization-def}
by \texttt{auto}
\{\texttt{fix literal::Literal}
\texttt{assume literal el (getC state) literal \neq opposite ?Cl}
\texttt{have elementLevel (opposite literal) (getM state) < currentLevel}
\texttt{(getM state) :}
\textbf{proof}\texttt{–}
\texttt{have elementLevel (opposite literal) (getM state) \leq currentLevel}
\texttt{(getM state) :}
\texttt{using elementLevelLeqCurrentLevel[of opposite literal getM state]}
by \texttt{simp}
\}\moreover
\texttt{have elementLevel (opposite literal) (getM state) \neq currentLevel}
\texttt{(getM state) :}
\texttt{proof}\texttt{–}
\{\texttt{assume \neg ?thesis}
\texttt{with \langle literal el (getC state) \rangle}
\texttt{have literal el ?clls}
by \texttt{simp}
\texttt{hence False}
\texttt{using \langle length ?clls = 1 \rangle}
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))
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) (getC state) (getM state)
using CnEqual1IffUIP[of state]
using assms
by simp

have ¬ ?Cl el (decisions (getM state))
proof−
{
assume ¬ ?thesis
hence isUIP (opposite ?Cl) (getC state) (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

595
with \( \neg \text{isUIP} \ (\text{opposite } ?C) \ (\text{getC state}) \ (\text{getM state}) \)

have False
  by simp
} thus \(?thesis\)
  by auto
qed

have elementLevel ?C (\text{getM state}) = \text{currentLevel} (\text{getM state})
  using \(\text{InvariantClCurrentLevel} \ (\text{getCl state}) \ (\text{getM state})\):
  unfolding \(\text{InvariantClCurrentLevel-def}\)
  by simp
hence elementLevel ?C (\text{getM state}) > 0
  using \(\text{currentLevel} (\text{getM state}) > 0\)
  by simp

obtain reason
  where isReason \(\text{nth} \ (\text{getF state}) \ \text{reason} \ ?C \ \{\ \text{elements} \ (\text{getM state})\}\)
  getReason state ?C = \text{Some} \ \text{reason} \ 0 \leq \text{reason} \land \text{reason} < \text{length} (\text{getF state})
  using \(\text{InvariantGetReasonIsReason} \ (\text{getReason state}) \ (\text{getF state}) \ (\text{getM state}) \ (\text{set} \ (\text{getQ state}))\):
  unfolding \(\text{InvariantGetReasonIsReason-def}\)
  using \(\text{literalTrue} \ ?C \ \{\ \text{elements} \ (\text{getM state})\}\)
  using \(\neg \ ?C \ \text{el} \ \{\ \text{decisions} \ (\text{getM state})\}\)
  using \(\text{elementLevel} ?C (\text{getM state}) > 0\)
  by auto

let \(?res = \text{resolve} \ (\text{getC state}) \ (\text{getF state}) \ ! \ \text{reason} \ (\text{opposite } ?C)\)

obtain ol::Literal
  where ol el (\text{getC state})
  ol \neq \text{opposite } ?C
  elementLevel \ (\text{opposite } \text{el}) \ (\text{getM state}) \geq \text{elementLevel} ?C \ (\text{getM state})
  using \(\text{isLastAssertedLiteral} \ ?C \ \{\ \text{oppositeLiteralList} \ (\text{getC state})\}\)
  \{\ \text{elements} \ (\text{getM state})\}\):
  unfolding \(\text{isUIP-def}\)
  by auto
hence ol el ?res
  unfolding \(\text{resolve-def}\)
  by simp
hence \(?res \neq []\)
  by auto
have opposite ol el \{\ \text{oppositeLiteralList} \ ?res\}
  using \{ol el ?res\}
  using \(\text{literalElListIFFOppositeLiteralElOppositeLiteralList[of} \ ?res\]\)
  by simp
have opposite ol el (oppositeLiteralList (getC state))
  using (ol el (getC state));
  using literalElListIffOppositeLiteralElOppositeLiteralList[ol el 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 [ol el getC state] (getM 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 [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 formulaEntailsClause (getF state) ?rc
  using ⟨0 ≤ reason ∧ reason < length (getF state)⟩
  using nth-mem[of reason getF state]
  by (simp add: formulaEntailsItsClauses)
hence formulaEntailsClause ?F0 ?rc
  by (simp add: formulaEntailsClauseAppend)

hence formulaEntailsClause F0 ?rc
  using ⟨InvariantEquivalentZL (getF state) (getM state) F0⟩
  unfolding InvariantEquivalentZL-def
  unfolding formulaEntailsClause-def
  unfolding equivalentFormulae-def
  by simp

hence formulaEntailsClause F0 ?res
  using ⟨getConflictFlag state⟩
  using ⟨InvariantCEntailed (getConflictFlag state) F0 (getC state)⟩
  using InvariantCEntailedAfterExplain[of getConflictFlag state F0 getConflictFlag state reason ?res getCl state]
  unfolding InvariantCEntailed-def
  by auto
hence formulaEntailsClause ?F0 ?res
  using ⟨InvariantEquivalentZL (getF state) (getM state) F0⟩
  unfolding InvariantEquivalentZL-def
  unfolding formulaEntailsClause-def
  unfolding equivalentFormulae-def
  by simp

hence formulaEntailsClause ?F0 ?C
  using formulaEntailsClauseRemoveEntailedLiteralOpposites[of ?F0 ?res ?M0]
    using val2formIsEntailed[of getF state ?M0 []]
  unfolding formulaEntailsClause-def
  by (auto simp add: clauseTrueIffContainsTrueLiteral)

hence formulaEntailsClause F0 ?C
  using ⟨InvariantEquivalentZL (getF state) (getM state) F0⟩
  unfolding InvariantEquivalentZL-def
  unfolding formulaEntailsClause-def
  unfolding equivalentFormulae-def
  by simp

let ?ll = getLastAssertedLiteral (oppositeLiteralList ?res) (elements (getM state))
have isLastAssertedLiteral ?ll (oppositeLiteralList ?res) (elements (getM state))
  using ⟨?res ≠ []⟩
  using ⟨clauseFalse ?res (elements (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 \texttt{literalElListIffOppositeLiteralElOppositeLiteralList}\left[\text{of \texttt{?ll elements (prefixToLevel 0 (getM state))}}\right]

by simp

have \texttt{opposite ?ll el \texttt{?C'}}
using \langle \texttt{opposite ?ll el \texttt{?res}} \rangle
using \langle \neg \texttt{opposite \texttt{?ll el \texttt{?oppM0}} \rangle 
using \texttt{listDiffIff}\left[\text{of opposite \texttt{?ll \texttt{?res \texttt{?oppM0}}} \right]
by simp

hence \texttt{?ll el (oppositeLiteralList \texttt{?C'})}
using \texttt{literalElListIffOppositeLiteralElOppositeLiteralList}\left[\text{of opposite \texttt{?ll \texttt{?C'}} \right]
by simp

have \texttt{set (oppositeLiteralList \texttt{?C'}) \subseteq set (oppositeLiteralList \texttt{?res})}
proof
fix \texttt{x}
assume \texttt{x \in set (oppositeLiteralList \texttt{?C'})}
thus \texttt{x \in set (oppositeLiteralList \texttt{?res})}
using \texttt{literalElListIffOppositeLiteralElOppositeLiteralList}\left[\text{of opposite \texttt{x \texttt{?C'}} \right]
using \texttt{literalElListIffOppositeLiteralElOppositeLiteralList}\left[\text{of opposite \texttt{x \texttt{?res}}} \right]
using \texttt{listDiffIff}\left[\text{of opposite \texttt{x \texttt{?res \texttt{oppM0}}} \right]
by auto
qed

have \texttt{isLastAssertedLiteral \texttt{?ll (oppositeLiteralList \texttt{?C'}) (elements (getM state))}}
using \langle \texttt{?ll el (oppositeLiteralList \texttt{?C'})} \rangle
using \langle \texttt{set (oppositeLiteralList \texttt{?C'}) \subseteq set (oppositeLiteralList \texttt{?res})} \rangle
using \texttt{isLastAssertedLiteral \texttt{?ll (oppositeLiteralList \texttt{?res}) (elements (getM state))}}
using \texttt{isLastAssertedLiteralSubset}\left[\text{of \texttt{?ll oppositeLiteralList \texttt{?res \texttt{elements (getM state) oppositeLiteralList \texttt{?C'}}}} \right]
by auto
moreover
have \texttt{set (oppositeLiteralList \texttt{?C'}) = set (oppositeLiteralList \texttt{?C'})}
unfolding \texttt{oppositeLiteralList-def}
by simp
ultimately
have \texttt{isLastAssertedLiteral \texttt{?ll (oppositeLiteralList \texttt{?C'}) (elements (getM state))}}
unfolding \texttt{isLastAssertedLiteral-def}
by auto

hence \texttt{?ll el (oppositeLiteralList \texttt{?C'})}
unfolding \texttt{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[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 ?ll (oppositeLiteralList ?C) (elements (getM state));
  using (?ll = ?l')
  using (elementLevel ?ll (getM state) = currentLevel (getM state))
  using (getReason state ?Cl = Some reason)
using ⟨clauseFalse ?C (elements (getM state))⟩
using ⟨formulaEntailsClause F0 ?C⟩
unfolding applyExplain-def
unfolding InvariantCFalse-def
unfolding InvariantCEntailed-def
unfolding InvariantClCharacterization-def
unfolding InvariantClCurrentLevel-def
unfolding InvariantCnCharacterization-def
unfolding InvariantUniqC-def
unfolding setConflictAnalysisClause-def
by (simp add: findLastAssertedLiteral-def countCurrentLevelLiterals-def Let-def uniqDistinct distinct-remdups-id)
qed

definition
multLessState = { (state1, state2). (getM state1 = getM state2) ∧ (getC state1, getC state2) ∈ multLess (getM state1) }

lemma ApplyExplainUIPTermination:
assumes
InvariantUniq (getM state)
InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state))
InvariantCFalse (getConflictFlag state) (getM state) (getC state)
InvariantClCurrentLevel (getCl state) (getM state)
InvariantClCharacterization (getCl state) (getC state) (getM state)
InvariantCnCharacterization (getCn state) (getC state) (getM state)
InvariantCEntailed (getConflictFlag state) F0 (getC state)
InvariantEquivalentZL (getF state) (getM state) F0
getConflictFlag state
currentLevel (getM state) > 0
shows
applyExplainUIP-dom state
using assms
proof (induct rule: wf-induct[of multLessState])
case 1
thus ?case
unfolding wf-eq-minimal
proof –
show ∀ Q (state::State). state ∈ Q → (∃ stateMin ∈ Q. ∀ state’. (state’, stateMin) ∈ multLessState → state’ /∈ Q)
proof –
{ fix Q :: State set and state :: State
assume state ∈ Q

602
let \(?M = (getM \text{ state})\)
let \(?Q1 = \{C::\text{Clause}. \exists \text{ state}. \text{ state} \in Q \land (getM \text{ state}) = ?M \land (getC \text{ state}) = C\}\)
from \(<\text{state} \in Q>\)
have \(getC \text{ state} \in ?Q1\)
by auto
with \(wfMultLess[of \ ?M]\)
obtain \(Cmin\) where \(Cmin \in ?Q1\)
\((\forall C'. Cmin \in ?Q1 \land (getC C' \land getC Cmin)) \in \text{ multLess} ?M \rightarrow C' \notin ?Q1\)
unfolding \(\text{ wf-eq-minimal}\)
apply \((\text{ erule-tac } x=?Q1 \text{ in allE})\)
by auto
from \(<Cmin \in ?Q1>\) obtain \(stateMin\)
where \(stateMin \in Q\) \((getM stateMin) = \ ?M\ \text{ getC stateMin} = \ Cmin\)
by auto
have \((\forall state'. (state', stateMin) \in \text{ multLessState} \rightarrow state' \notin Q)\)
proof
fix \(state'\)
show \((state', stateMin) \in \text{ multLessState} \rightarrow state' \notin Q)\)
proof
assume \((state', stateMin) \in \text{ multLessState}\)
with \((getM stateMin = ?M)\)
have \((getM state') = getM stateMin \ \text{ (getC state') \in \text{ multLess} ?M \rightarrow \text{ getC stateMin})\)
by auto
unfolding \(\text{ multLessState-def}\)
by auto
from \((\forall C'. (C', Cmin) \in \text{ multLess} ?M \rightarrow C' \notin ?Q1)\)
\((\text{ getC state', getC stateMin}) \in \text{ multLess} ?M \ \text{ (getC stateMin)}\)
= \(Cmin\)
have \((\text{ getC state'} \notin ?Q1)\)
by simp
with \((getM state' = getM stateMin) \ \text{ (getM stateMin = ?M)}\)
show \(state' \notin Q)\)
by auto
qed
qed
with \((\text{ stateMin } \in Q)\)
have \(\exists \text{ stateMin } \in Q. \ (\forall state'. (state', stateMin) \in \text{ multLessState} \rightarrow state' \notin Q)\)
by auto
} thus \(\text{ thesis}\)
by auto
qed
qed
next

603
case (2 state')

note ih = this

proof (cases getN state' = 1)
  case True
  show 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') ∈ multLessState

proof
  have getM ?state'' = getM state'
    unfolding applyExplain-def
    unfolding setConflictAnalysisClause-def
    by (auto split: option.split simp add: findLastAssertedLiteral-def countCurrentLevelLiterals-def Let-def)

  let Cl = getCl state'
  let oppM0 = oppositeLiteralList (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'))
        using 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'))
getReason 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
  (getM state' = getM state) \wedge
  (getF state' = getF state) \wedge
  (getQ state' = getQ state) \wedge
  (getWatch1 state' = getWatch1 state) \wedge
  (getWatch2 state' = getWatch2 state) \wedge
  (getWatchList state' = getWatchList state) \wedge
  (getConflictFlag state' = getConflictFlag state) \wedge
  (getConflictClause state' = getConflictClause state) \wedge
  (getSATFlag state' = getSATFlag state) \wedge
  (getReason state' = getReason 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

607
using \( \langle \text{applyExplainUIP state}' = \text{state}' \rangle \)
by (simp add: Let-def)
qed

lemma isUIPApplyExplainUIP:
assumes applyExplainUIP-dom state
InvarIntUni (getM state)
InvarIntFalse (getConflictFlag state) (getM state) (getC state)
InvarIntEntailed (getConflictFlag state) F0 (getC state)
InvarIntClCharacterization (getCl state) (getC state) (getM state)
InvarIntClCharacterization (getCn state) (getC state) (getM state)
InvarIntClCurrentLevel (getCl state) (getM state)
InvarIntGetReasonIsReason (getReason state) (getF state) (getM state)
(set (getQ state))
InvarIntClCurrentLevel (getM state) currentLevel > 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 \( \text{ih} = \text{this} \)
show ?case
proof (cases getCn state' = 1)
case True
with applyExplainUIP.simps[of state']
have applyExplainUIP state' = state'
  by simp
thus ?thesis
  using \( \text{ih} \)
  using CnEqual1IffUIP[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 InvarIntUni (getM ?state'')
  InvarIntGetReasonIsReason (getReason ?state'') (getF ?state'')
  (getM ?state'') (set (getQ ?state''))
  InvarIntClCurrentLevel (getF ?state'') (getM ?state'') F0
  getConflictFlag ?state''
currentLevel (getM ?state'') > 0

608
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") (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

lemma InvariantsClAfterExplainUIP:
assumes
applyExplainUIP-dom state
InvariantUniq (getM state)
InvariantCFalse (getConflictFlag state) (getM state) (getC state)
InvariantCEntailed (getConflictFlag state) F0 (getC state)
InvariantClCharacterization (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
  currentLevel (getM state) > 0
shows
let state' = applyExplainUIP state in
  InvariantCFalse (getConflictFlag state') (getM state') (getC state')
  InvariantCEntailed (getConflictFlag state') F0 (getC state') 
  InvariantClCharacterization (getCl state') (getC state') (getM state') 
  InvariantCnCharacterization (getCn state') (getC state') (getM state') 

609
InvariantCnCharacterization \((\mathrm{getCn\ state'})\ (\mathrm{getC\ state'})\ (\mathrm{getM\ state'})\) 
\(\land\)
InvariantClCurrentLevel \((\mathrm{getCl\ state'})\ (\mathrm{getM\ state'})\) 
\(\land\)
InvariantUniqC \((\mathrm{getC\ state'})\)
using asms

proof\((\mathrm{induct\ state\ rule: \ applyExplainUIP-dom.induct})\)
    case \((\mathrm{step\ state'})\)
    note \(\mathrm{ih} = \mathrm{this}\)
    show \(?\mathrm{case}\)
    proof\((\mathrm{cases\ getC\ state'})\)
        case \(\mathrm{True}\)
        with \(\mathrm{applyExplainUIP}\)\[\text{of state'}\]
        have \(\mathrm{applyExplainUIP}\ \mathrm{state'} = \mathrm{state'}\)
            by simp
        thus \(?\mathrm{thesis}\)
            using asms
            using \(\mathrm{ih}\)
            by \((\mathrm{auto\ simp\ only: \ Let-def})\)
        next
        case \(\mathrm{False}\)
        let \(?\mathrm{state'}''\) = \(\mathrm{applyExplain}\)\[\mathrm{getCl\ state'}\)\(\mathrm{state'}\)
        let \(?\mathrm{state'}\) = \(\mathrm{applyExplainUIP}\ ?\mathrm{state'}''\)
            from \(\mathrm{applyExplainUIP}\)\[\text{of state'}\] \(\mathrm{False}\)
        have \(\mathrm{applyExplainUIP}\ \mathrm{state'} = ?\mathrm{state'}\)
            by \((\mathrm{simp\ add: \ Let-def})\)
        moreover
        have \(\mathrm{InvariantUniq}\ (\mathrm{getM\ ?\mathrm{state'}''})\)
            \(\mathrm{InvariantGetReasonIsReason}\ (\mathrm{getReason\ ?\mathrm{state'}''})\ (\mathrm{getF\ ?\mathrm{state'}''})\ (\mathrm{getM\ ?\mathrm{state'}''})\)
            \(\mathrm{set}\ (\mathrm{getQ\ ?\mathrm{state'}''})\)
            \(\mathrm{InvariantEquivalentZL}\ (\mathrm{getF\ ?\mathrm{state'}''})\ (\mathrm{getM\ ?\mathrm{state'}''})\ F0\)
            \(\mathrm{getConflictFlag\ ?\mathrm{state'}''}\)
            \(\mathrm{currentLevel\ (getM\ ?\mathrm{state'}'') > 0}\)
            using \(\mathrm{ih}\)
            unfolding \(\mathrm{applyExplain-def}\)
            unfolding \(\mathrm{setConflictAnalysisClause-def}\)
            by \((\mathrm{auto\ split: \ option.split\ simp\ add: \ findLastAssertedLiteral-def\ countCurrentLevelLiterals-def\ Let-def})\)
        moreover
        have \(\mathrm{InvariantCFalse}\ (\mathrm{getConflictFlag\ ?\mathrm{state'}''})\ (\mathrm{getM\ ?\mathrm{state'}''})\)
            \(\mathrm{getC\ ?\mathrm{state'}''})\)
            \(\mathrm{InvariantCEnlailed}\ (\mathrm{getConflictFlag\ ?\mathrm{state'}''})\ F0\ (\mathrm{getC\ ?\mathrm{state'}''})\)
            \(\mathrm{InvariantClCharacterization}\ (\mathrm{getCl\ ?\mathrm{state'}''})\ (\mathrm{getC\ ?\mathrm{state'}''})\ (\mathrm{getM\ ?\mathrm{state'}''})\)
            \(\mathrm{InvariantCnCharacterization}\ (\mathrm{getCn\ ?\mathrm{state'}''})\ (\mathrm{getC\ ?\mathrm{state'}''})\ (\mathrm{getM\ ?\mathrm{state'}''})\)
            \(\mathrm{InvariantClCurrentLevel}\ (\mathrm{getCl\ ?\mathrm{state'}''})\ (\mathrm{getM\ ?\mathrm{state'}''})\)
            \(\mathrm{InvariantUniqC}\ (\mathrm{getC\ ?\mathrm{state'}''})\)
            using \(\mathrm{False}\)
            using \(\mathrm{ih}\)

610
using InvariantsClAfterApplyExplain[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...
shows...
proof
let ?oppC = oppositeLiteralList (getC state)
let ?Cl = getC 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 (InvariantClCurrentLevel (getCl state) (getM state));
using (currentLevel (getM state) > 0);
unfolding InvariantClCurrentLevel-def
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

show ?thesis
proof (cases getC state = [opposite ?Cl])
case True
thus ?thesis
using backjumpLevelZero[of opposite ?Cl oppositeLiteralList ?oppC getM state]
using (isLastAssertedLiteral ?Cl ?oppC (elements (getM state)));
using True
using (elementLevel ?Cl (getM state) > 0);
unfolding getBackjumpLevel-def
unfolding isMinimalBackjumpLevel-def
by (simp add: Let-def)
next
let ?Cll = getCll state
case False
with (InvariantCllCharacterization (getCl state) (getCll state) (getC state) (getM state))
(InvariantUniqC (getC state));
have isLastAssertedLiteral ?Cll (removeAll ?Cl ?oppC) (elements (getM state))
unfolding InvariantCllCharacterization-def
unfolding InvariantUniqC-def
using uniqOneElementCharacterization[of getC state opposite ?Cl]
by simp
hence ?Cll el ?oppC ?Cll ≠ ?Cl
unfolding isLastAssertedLiteral-def
by auto
hence opposite ?Cll el (getC state)
by auto

show ?thesis
using backjumpLevelLastLast[of opposite ?Cl getC state getM state opposite ?Cll]
using (isUIP (opposite (getCl state)) (getC state) (getM state))
using \langle \text{clauseFalse} \ (\text{get}C \ \text{state}) \ (\text{elements} \ (\text{get}M \ \text{state})) \rangle
using \langle \text{isLastAssertedLiteral} \ ?Cll \ (\text{removeAll} \ ?Cl \ ?oppC) \ (\text{elements} \ (\text{get}M \ \text{state})) \rangle
using \langle \text{InvariantUniq} \ (\text{get}M \ \text{state}) \rangle
using \langle \text{InvariantUniqC} \ (\text{get}C \ \text{state}) \rangle
using \langle \text{uniqOneElementCharacterization} \rangle
\text{of get}C \ \text{state} \ \text{opposite} \ ?Cll \ \text{elements} \ (\text{get}M \ \text{state})
\text{using False}
\text{using} \langle \text{opposite} \ ?Cll \ \text{el} \ (\text{get}C \ \text{state}) \rangle
\text{unfolding isMinimalBackjumpLevel-def}
\text{unfolding getBackjumpLevel-def}
\text{by} \ (\text{auto simp add: Let-def})
proof
\text{by} \ (\text{auto simp add: let-def})
\end{proof}

\text{lemma} \ \text{applyLearnPreservedVariables}:
\begin{align*}
\text{let state}' &= \text{applyLearn} \ \text{state} \ \text{in} \\
\text{get}M \ \text{state}' &= \text{get}M \ \text{state} \land \\
\text{get}Q \ \text{state}' &= \text{get}Q \ \text{state} \land \\
\text{get}C \ \text{state}' &= \text{get}C \ \text{state} \land \\
\text{get}Cl \ \text{state}' &= \text{get}Cl \ \text{state} \land \\
\text{getConflictFlag} \ \text{state}' &= \text{getConflictFlag} \ \text{state} \land \\
\text{getConflictClause} \ \text{state}' &= \text{getConflictClause} \ \text{state} \land \\
\text{get}F \ \text{state}' &= \begin{cases} \\
\text{if get}C \ \text{state} = \begin{bmatrix} \text{opposite} \ (\text{get}Cl \ \text{state}) \end{bmatrix} \ \text{then} \\
\text{get}F \ \text{state} \\
\text{else} \\
(\text{get}F \ \text{state} \ \ominus \ [\text{get}C \ \text{state}]) \\
\end{cases}
\end{align*}
\text{proof} \ (\text{cases get}C \ \text{state} = \begin{bmatrix} \text{opposite} \ (\text{get}Cl \ \text{state}) \end{bmatrix})
\text{case True}
\text{thus} \ \text{?thesis}
\text{unfolding applyLearn-def}
\text{unfolding setWatch1-def}
\text{unfolding setWatch2-def}
\text{by} \ (\text{simp add: Let-def})
\text{next}
\text{case False}
\text{thus} \ \text{?thesis}
\text{unfolding applyLearn-def}
\text{unfolding setWatch1-def}
\text{unfolding setWatch2-def}
\text{by} \ (\text{simp add: Let-def})
lemma 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
InvariantCFalse (getConflictFlag state) (getM state) (getC state)
InvariantUniqC (getC state)
shows
let state' = (applyLearn state) in
InvariantWatchesEl (getF state') (getWatch1 state') (getWatch2 state') ∧
InvariantWatchesDiffer (getF state') (getWatch1 state') (getWatch2 state') ∧
InvariantWatchCharacterization (getF state') (getWatch1 state') (getWatch2 state') (getM state') ∧
InvariantWatchListsContainOnlyClausesFromF (getWatchList state') (getF state') ∧
InvariantWatchListsUniq (getWatchList state') ∧
InvariantWatchListsCharacterization (getWatchList state') (getWatch1 state') (getWatch2 state')
proof (cases getC state ≠ [opposite (getCl state)])
case False
thus ?thesis
  using assms
  unfolding applyLearn-def
  unfolding InvariantClCharacterization-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
  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 ?ll 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 ⟨set (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)⟩ unfolding applyLearn-def
    using ⟨set (getC state) ≠ [opposite ?l]⟩ unfolding setWatch1-def
    unfolding setWatch2-def
    by (auto simp add: Let-def)
moreover
  have getF ?state' ! clause = (getC state)
    using ⟨clause = length (getF state)⟩ unfolding applyLearn-def
    using ⟨set (getC state) ≠ [opposite ?l]⟩ 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')
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)
        (getWatch2 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 ?ll}|⟩
        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})
    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)
  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)
  unfolding 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
}
(?l (getM state) ∧
  \text{elementLevel} \text{(opposite l)} (getM state) \leq \text{elementLevel}

(? ll (getM state)

proof -

\{ 

fix l

\text{assume} \ l \text{ el} (getC state) \ l \neq \text{opposite} \ ?l \ l \neq \text{opposite} \ ?ll

\text{hence} \ \text{opposite} \ l \text{ el} ?oppC

\text{using} \ \text{literalElListIffOppositeLiteralElOppositeLiteralList}[\text{of} \ l \text{ getC state}]

\text{by simp}

\text{moreover}

\text{from} \ (l \neq \text{opposite} \ ?l)

\text{have} \ \text{opposite} \ l \neq \ ?l

\text{using} \ \text{oppositeSymmetry}[\text{of} \ l \ ?l]

\text{by blast}

\text{ultimately}

\text{have} \ \text{opposite} \ l \text{ el} (\text{removeAll} \ ?l \ ?oppC)

\text{by simp}

\text{from} \ \langle \ \text{clauseFalse} \ (\text{getC state}) \ (\text{elements} \ (\text{getM state})) \rangle

\text{have} \ \text{literalFalse} \ l \ (\text{elements} \ (\text{getM state}))

\text{using} \ \langle \ \text{isLastAssertedLiteral} \ ?l \ ?oppC \ (\text{elements} \ (\text{getM state})) \rangle

\text{using} \ \text{lastAssertedLiteralHasHighestElementLevel}[\text{of} \ ?l \ ?oppC \ \text{getM state}]

\text{using} \ \langle \ \text{isLastAssertedLiteral} \ ?ll \ (\text{removeAll} \ ?l \ ?oppC) \ (\text{elements} \ (\text{getM state})) \rangle

\text{using} \ \langle \ \text{opposite} \ l \text{ el} ?oppC \ \text{getM state} \rangle

\text{unfolding} \ \text{InvariantUniq-def}

\text{using} \ \text{lastAssertedLiteralHasHighestElementLevel}[\text{of} \ ?ll \ \text{removeAll} \ ?l \ ?oppC \ \text{getM state}]

\text{using} \ \text{lastAssertedLiteralHasHighestElementLevel}[\text{of} \ ?l \ ?oppC \ \text{getM state}]

\text{using} \ \text{lastAssertedLiteralHasHighestElementLevel}[\text{of} \ ?ll \ \text{removeAll} \ ?l \ ?oppC]

\text{by simp}

\text{thus} \ ?thesis

\text{by simp}

qed

\text{moreover}

\text{have} \ \text{getF} \ ?state' \ ! \text{clause} = (\text{getC state})

\text{using} \ \langle \text{clause} = \text{length} \ (\text{getF state}) \rangle

\text{using} \ \langle \text{set} \ (\text{getC state}) \neq \{ \text{opposite} \ ?l} \rangle

621
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) ≠ {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 ∨ l = opposite ?ll)
    case True
    hence getWatchList ?state' l = (length (getF state)) # getWatchList state l
      using (set (getC state) ≠ {opposite ?l});
      unfolding applyLearn-def
      unfolding setWatch1-def
      unfolding setWatch2-def
      using (?ll ≠ ?l);
      by (auto simp add: Let-def nth-append)
  moreover
  have length (getF state) ∉ 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) ≠ {opposite ?l});
      unfolding applyLearn-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)⟩
      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)
    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)⟩
using \( \{ \text{set} \ (\text{getC} \ \text{state}) \neq \{ \text{opposite} \ \text{?l} \} \} \)

unfolding applyLearn-def

unfolding setWatch1-def

unfolding setWatch2-def

by (auto simp add: Let-def)

moreover

have InvariantCllCharacterization \( \{ \text{getCl} \ ?\text{state}' \} \) \( \{ \text{getCll} \ ?\text{state}' \} \) \( \{ \text{getC} \ ?\text{state}' \} \) \( \{ \text{getM} \ ?\text{state}' \} \)

unfolding InvariantCllCharacterization-def

using \( \{ \text{isLastAssertedLiteral} \ ?\text{ll} \ (\text{removeAll} \ ?\text{l} \ ?\text{oppC}) \} \) \( \{ \text{elements} \ (\text{getM} \ \text{state}) \} \)

unfolding applyLearn-def

unfolding setWatch1-def

unfolding setWatch2-def

by (auto simp add: Let-def)

ultimately

show \( ?\text{thesis} \)

by simp

qed

lemma InvariantCllCharacterizationAfterApplyLearn:

assumes

InvariantUniq \( \{ \text{getM} \ \text{state} \} \)

InvariantCllCharacterization \( \{ \text{getCl} \ \text{state} \} \) \( \{ \text{getC} \ \text{state} \} \) \( \{ \text{getM} \ \text{state} \} \)

InvariantCllFalse \( \{ \text{getConflictFlag} \ \text{state} \} \) \( \{ \text{getM} \ \text{state} \} \) \( \{ \text{getC} \ \text{state} \} \)

InvariantUniqC \( \{ \text{getC} \ \text{state} \} \)

getConflictFlag state

shows

let \( \text{state}' = \text{applyLearn} \ \text{state} \) in

InvariantCllCharacterization \( \{ \text{getCl} \ \text{state}' \} \) \( \{ \text{getCll} \ \text{state}' \} \) \( \{ \text{getC} \ \text{state}' \} \) \( \{ \text{getM} \ \text{state}' \} \)

proof (cases get\( \text{C} \ \text{state} \neq \{ \text{opposite} (\text{getCl} \ \text{state}) \} \))

case False

thus \( ?\text{thesis} \)

using assms

unfolding applyLearn-def

unfolding InvariantCllCharacterization-def

by (simp add: Let-def)

next

case True

let \( ?\text{oppC} = \text{oppositeLiteralList} (\text{getC} \ \text{state}) \)

let \( ?\text{l} = \text{getCl} \ \text{state} \)

let \( ?\text{ll} = \text{getLastAssertedLiteral} (\text{removeAll} \ ?\text{l} \ ?\text{oppC}) \) \( \{ \text{elements} \ (\text{getM} \ \text{state}) \} \)

have clauseFalse \( \{ \text{getC} \ \text{state} \} \) \( \{ \text{elements} \ (\text{getM} \ \text{state}) \} \)
using \langle \text{getConflictFlag state} \rangle
\text{using} \langle \text{InvariantCFalse (getConflictFlag state) (getM state) (getC state)} \rangle

unfolding \text{InvariantCFalse-def}
by simp

from True
have set \((\text{getC state}) \neq \{\text{opposite ?}l\}\)

\text{using} \langle \text{InvariantUniqC (getC state)} \rangle
\text{using uniqOneElementCharacterization[of getC state opposite ?l]}

unfolding \text{InvariantUniqC-def}
by (simp add: Let-def)

have isLastAssertedLiteral ?l oppC (elements (getM state))

\text{using} \langle \text{InvariantClCharacterization (getCl state) (getC state) (getM state)} \rangle

unfolding \text{InvariantClCharacterization-def}
by simp

have opposite ?l el (getC state)

\text{using} \langle \text{isLastAssertedLiteral ?}l \ oppC (elements (getM state)) \rangle

unfolding \text{isLastAssertedLiteral-def}

\text{using} \langle \text{literalElListIffOppositeLiteralElOppositeLiteralList[of ?l oppC]} \rangle

by simp

have removeAll ?l oppC \neq []

proof –
{ 
assume \neg \text{thesis}

hence set oppC \subseteq \{?l\}

\text{using set-removeAll[of ?l oppC]}

by auto

have set \((\text{getC state}) \subseteq \{\text{opposite ?}l\}\)

proof

fix \(x\)

assume \(x \in \text{set (getC state)}\)

hence opposite \(x \in \text{set oppC}\)

\text{using} \langle \text{literalElListIffOppositeLiteralElOppositeLiteralList[of x getC state]} \rangle

by simp

hence opposite \(x \in \{?l\}\)

\text{using} \langle \text{set oppC \subseteq \{?l\}} \rangle

by auto

thus \(x \in \{\text{opposite ?}l\}\)

\text{using oppositeSymmetry[of x ?l]}

by force

qed

hence False

626
using \( \text{set \ (getC state) \neq \{ \text{opposite \ ?l} \}} \)
using \( \text{opposite \ ?l \ el \ getC state} \)
by (auto simp add: Let-def)

\begin{verbatim}
    \} thus \text{?thesis}
    by auto
\end{verbatim}

\textbf{q}ed

\begin{verbatim}
have clauseFalse \( \text{oppositeLiteralList \ (removeAll \ ?l \ ?oppC)} \) \(\text{(elements \ (getM state))}\)
    using \( \text{clauseFalse \ (getC state) \ (elements \ (getM state))}\)
    using \( \text{oppositeLiteralListRemove \ [\text{of \ ?l \ ?oppC]} \)\)
    by (simp add: clauseFalse_iffAllLiteralsAreFalse)
moreover
have \( \text{oppositeLiteralList \ (removeAll \ ?l \ ?oppC) \neq \[]} \)
    using \( \text{removeAll \ ?l \ ?oppC \neq \[]} \)
    using \( \text{oppositeLiteralListNonempty} \) by simp
ultimately
have \( \text{isLastAssertedLiteral \ ?ll \ (removeAll \ ?l \ ?oppC) \ (elements \ (getM state))}\)
    using \( \text{getLastAssertedLiteralCharacterization [of \ (removeAll \ ?l \ ?oppC) \ elements \ (getM state)]}\)
    unfolding \( \text{InvariantUniq} \ (\text{getM state})\)
    by auto
thus \text{?thesis}
    using \( \text{set \ (getC state) \neq \{ \text{opposite \ ?l} \}} \)
    unfolding \text{applyLearn-def}
    unfolding \text{setWatch1-def}
    unfolding \text{setWatch2-def}
    unfolding \text{InvariantCllCharacterization-def}
    by (auto simp add: Let-def)
\end{verbatim}

\textbf{q}ed

\textbf{lemma InvariantConflictClauseCharacterizationAfterApplyLearn:}
\textbf{assumes}
\( \text{getConflictFlag state} \)
\(\text{InvariantConflictClauseCharacterization \ (getConflictFlag state) \ (getConflictClause state) \ (getF state) \ (getM state)}\)
\textbf{shows}
\( \text{let state' = applyLearn state in} \)
\(\text{InvariantConflictClauseCharacterization \ (getConflictFlag state')} \)
\(\text{(getConflictClause state')} \) \(\text{(getF state')} \) \(\text{(getM state')}\)
\textbf{proof=}
\begin{verbatim}
    have \( \text{getConflictClause state < length \ (getF state)} \)
    using \text{assms}
    unfolding \text{InvariantConflictClauseCharacterization-def}
    by (auto simp add: Let-def)
\end{verbatim}

627
hence \( \text{nth} \left( \left( \text{getF state} \right) \oplus \left[ \text{getC state} \right] \right) \left( \text{getConflictClause state} \right) = \text{nth} \left( \text{getF state} \right) \left( \text{getConflictClause state} \right) \)

by (simp add: nth-append)

thus \(?\text{thesis}\)

using \(\text{InvariantConflictClauseCharacterization} \left( \text{getConflictFlag state} \right) \left( \text{getConflictClause state} \right) \left( \text{getF state} \right) \left( \text{getM state} \right) \)

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

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

unfolding \(\text{setWatch1-def}\)

unfolding \(\text{setWatch2-def}\)

by (auto simp add: \(\text{Let-def} \left( \text{clauseFalseAppendValuation} \right)\))

qed

lemma \(\text{InvariantGetReasonIsReasonAfterApplyLearn}\):

assumes

\(\text{InvariantGetReasonIsReason} \left( \text{getReason state} \right) \left( \text{getF state} \right) \left( \text{getM state} \right) \left( \text{set} \left( \text{getQ state} \right) \right)\)

shows

\(\text{let state'} = \text{applyLearn state in} \)

\(\text{InvariantGetReasonIsReason} \left( \text{getReason state'} \right) \left( \text{getF state'} \right) \left( \text{getM state'} \right) \left( \text{set} \left( \text{getQ state'} \right) \right)\)

proof (cases \(\text{getC state} = \text{[opposite} \left( \text{getCl state} \right)\))

case True

thus \(?\text{thesis}\)

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

using \(\text{assms}\)

by (simp add: \(\text{Let-def}\))

next

case False

have \(\text{InvariantGetReasonIsReason} \left( \text{getReason state} \right) \left( \left( \text{getF state} \right) \oplus \left[ \text{getC state} \right] \right) \left( \text{getM state} \right) \left( \text{set} \left( \text{getQ state} \right) \right)\)

using \(\text{assms}\)

using \(\text{nth-append[of getF state [getC state]}\)

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

by auto

thus \(?\text{thesis}\)

using False

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

unfolding \(\text{setWatch1-def}\)

unfolding \(\text{setWatch2-def}\)

by (simp add: \(\text{Let-def}\))

qed

lemma \(\text{InvariantQCharacterizationAfterApplyLearn}\):

assumes

\(\text{getConflictFlag state}\)

\(\text{InvariantQCharacterization} \left( \text{getConflictFlag state} \right) \left( \text{getQ state} \right) \left( \text{getF state} \right) \left( \text{getM state} \right)\)
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 getC ?state' = getC 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 `InvariantCllCharacterizationAfterApplyLearn[of state]`
by (simp add: Let-def)

**hence** `isMinimalBackjumpLevel (getBackjumpLevel[of state'] (opposite ?l) (getC[of state']) (getM[of state'])`
using `assms`
using `⟨getM[of state'] = getM state ⟩ ⟨getC[of state'] = getC state ⟩ ⟨getCl[of 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[of state'] < elementLevel[of ?l (getM[of state'])`
unfolding `isMinimalBackjumpLevel-def`
unfolding `isBackjumpLevel-def`
by simp

**hence** `getBackjumpLevel[of state'] < currentLevel(of ?l getM[of state'])`

using `elementLevelLeqCurrentLevel[of ?l getM[of state']]`
by simp

**have** `InvariantNoDecisionsWhenConflict [getC state] (getM[of state'])
(getBackjumpLevel[of state']) ∧
InvariantNoDecisionsWhenUnit [getC state] (getM[of state'])
(getBackjumpLevel[of state'])`
proof -

{ fix clause::Clause
  assume clause el [getC state]
  **hence** clause = getC state
  by simp

  **have** `(∀ level'. level' < (getBackjumpLevel[of state']) →
  ¬ clauseFalse clause (elements (prefixToLevel level' (getM[of state'])))) ∧
  (∀ level'. level' < (getBackjumpLevel[of state']) →
  ¬ (∃ l, isUnitClause clause l (elements (prefixToLevel level' (getM[of state'])))))` (is `?false ∧ ?unit`)
proof (cases getC state = [opposite ?l])
case True
  thus ?thesis
  using `⟨getM[of state'] = getM state ⟩ ⟨getC[of state'] = getC state ⟩`
  unfolding `getBackjumpLevel-def`
  by (simp add: Let-def)
next
case False
  **hence** `getF[of state'] = getF state @ [getC state]`
  unfolding `applyLearn-def`
  unfolding `setWatch2-def`
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)`
    unfolding ⟨getM ?state' = getM state; getC ?state' = getC state⟩
    using `ClauseFalse (getC state) (elements (getM state))`
    unfolding `isMinimalBackjumpLevel (getBackjumpLevel ?state')`
    using `isMinimalBackjumpLevelEnsuresIsNotUnitBeforePrefix[of getM ?state' getC ?state' getBackjumpLevel ?state' opposite ?]`
    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)`
    unfolding `isMinimalBackjumpLevel (getBackjumpLevel ?state')`
    using `ClauseFalse (getC state) (elements (getM state))`
    unfolding `isBackjumpLevelEnsuresIsUnitInPrefix[of getM ?state' getC ?state' getBackjumpLevel ?state' opposite ?]`
    unfolding `isMinimalBackjumpLevel-def`
    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: clauseFalseIffAll_literals_AreFalse)
  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))
      qed
using \langle \text{level'} < \text{getBackjumpLevel } ?\text{state}' \rangle

using isPrefixPrefixToLevelLowerLevel[of level' getBackjumpLevel ?state' getM state]
  by simp
then obtain s
where prefixToLevel level' (getM state) @ s = prefixToLevel (getBackjumpLevel ?state') (getM state)
  unfolding isPrefix-def
  by auto
hence prefixToLevel (getBackjumpLevel ?state') (getM state) = prefixToLevel level' (getM state) @ s
  by (rule sym)
thus \langle ?\text{thesis}\rangle
  using \langle \text{getM } ?\text{state}' = \text{getM state} \rangle
  using \langle \text{clause } = \text{getC state} \rangle
  using \langle \neg \text{clauseFalse } (\text{getC state}) \ (\text{elements } \text{prefixToLevel (getBackjumpLevel ?state') (getM state))} \rangle
  unfolding isPrefix-def
  by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
qed
qed
qed
ultimately
show \langle ?\text{thesis}\rangle
  by simp
qed
qed

} thus \langle ?\text{thesis}\rangle
  unfolding InvariantNoDecisionsWhenConflict-def
  unfolding InvariantNoDecisionsWhenUnit-def
  by (auto simp add: formulaFalseIffContainsFalseClause)
qed
ultimately
show \langle ?\text{thesis}\rangle
  by (simp add: Let-def)
qed

lemma InvariantEquivalentZLAfterApplyLearn:
  assumes
  InvariantEquivalentZL (getF state) (getM state) F0 and
  InvariantCEntailed (getConflictFlag state) F0 (getC state) and
  getConflictFlag state
  shows
    let state' = applyLearn state in
    InvariantEquivalentZL (getF state') (getM state') F0
proof -
  let \langle ?M0 = \text{val2form } (\text{elements } \text{prefixToLevel } 0 \ (\text{getM state}))\rangle
  have equivalentFormulae F0 (getF state @ ?M0)
  using \langle InvariantEquivalentZL (getF state) (getM state) F0 \rangle
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])]
  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 getF 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) (getCll state) (getC 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 ∧
isUnitClause bClause bLiteral (elements prefix) ∧
(getM state') = prefix @ [(bLiteral, False)]) ∧
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
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)
  InvariantWatchesEl (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( λ getConflictFlag := False, getQ := [], getM := prefix ) in
  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( λ 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 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 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))
elementLevel l (getM state) ≤ elementLevel (opposite w1) (getM state)
  by auto

  have literalTrue l (elements (getM ?state'))
    using elementLevel (opposite w1) (getM state) ≤ ?level
    using elementLevelLtLevelImpliesMemberPrefixToLevel[of l getM state ?level]
    using elementLevel l (getM state) ≤ elementLevel (opposite w1) (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 w1) (getM ?state') =
          elementLevel (opposite w1) (getM state))
    using elementLevel l (getM state) ≤ elementLevel (opposite w1) (getM state)
      using (l el ?clause);
      by auto

next
  case False
  {
    fix l
    assume l el ?clause l \neq w1 l \neq w2
    hence literalFalse l (elements (getM state))
      elementLevel (opposite l) (getM state) ≤ elementLevel (opposite w1) (getM state)
    using (literalFalse w1 (elements (getM state)));
    using False
      using (watchCharacterizationCondition w1 w2 (getM state) ?clause);
    unfolding watchCharacterizationCondition-def
    by auto

    have literalFalse l (elements (getM ?state')) \wedge
        elementLevel (opposite l) (getM ?state') ≤ elementLevel (opposite w1) (getM ?state')
      proof
        have literalFalse l (elements (getM ?state'))
          using (elementLevel (opposite w1) (getM state) ≤ ?level)
          using elementLevelLtLevelImpliesMemberPrefixToLevel[of opposite l getM state ?level]
using \( \text{elementLevel} \) (opposite \( l \)) \((\text{getM state})\)
\[ \text{elementLevel} \] (opposite \( w1 \)) \((\text{getM state})\)
by simp
moreover
from \( \text{literalFalse} \) \((\text{elements} \text{ (getM state)})\)
have \( \text{elementLevel} \) (opposite \( l \)) \((\text{getM state})\)
using \( \text{elementLevelPrefixElement} \)
by simp
ultimately
show \( \text{thesis} \)
using \( \text{elementLevel} \) (opposite \( w1 \)) \((\text{getM state})\)
\[ \text{elementLevel} \] (opposite \( w1 \)) \((\text{getM state})\)
using \( \text{elementLevelPrefixElement} \)
by simp

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

moreover
\{ 
assume \( \text{literalFalse} \) \((\text{elements} \text{ (getM state)})\)

hence \( \text{literalFalse} \) \((\text{elements} \text{ (getM state)})\)
using \( \text{isPrefixPrefixToLevel}[\text{of ?level getM state}] \)

using \( \text{isPrefixElements}[\text{of prefixToLevel ?level (getM state)}] \)

getM state
using \( \text{prefixIsSubset}[\text{of elements (prefixToLevel ?level (getM state)) elements (getM state)}] \)
by auto

from \( \text{literalFalse} \) \((\text{elements} \text{ (getM state)})\)

have \( \text{elementLevel} \) (opposite \( w2 \)) \((\text{getM state})\) \(\leq\) \( ?\text{level} \)
using \( \text{prefixToLevelElementsElementLevel}[\text{of opposite w2 ?level getM state}] \)
by simp

from \( \text{literalFalse} \) \((\text{elements} \text{ (getM state)})\)

have \( \text{elementLevel} \) (opposite \( w2 \)) \((\text{getM state})\)

using \( \text{elementLevelPrefixElement} \)
by simp

have \( ?a \) \( ?\text{state'} w2 \text{ w1} \lor ?b \) \( ?\text{state'} w2 \text{ w1} \)

proof (cases \( ?a \) \( \text{state'} w2 \text{ w1} \))
case True
then obtain l
where l ∈ ?clause literalTrue l (elements (getM state))
elementLevel l (getM state) ≤ elementLevel (opposite w2)
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 ∈ ?clause);
by auto
next
case False
{
fix l
assume l ∈ ?clause l ≠ w1 l ≠ w2
hence literalFalse l (elements (getM state))
elementLevel 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 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 \text{elementLevelLtLevelImpliesMemberPrefixToLevel}[\text{of } \text{opposite } l \text{ getM } \text{state } ?\text{level}]
using \langle \text{elementLevel} (\text{opposite } l) (\text{getM } \text{state}) \leq \text{elementLevel} (\text{opposite } w2) (\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 ?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 } w2) (\text{getM } \text{state}) \leq \text{elementLevel} (\text{opposite } l) (\text{getM } \text{state}) \rangle
using \langle \text{d el } ?\text{clause} \rangle
by auto
qed
}
thus ?thesis
by auto
qed
}
ultimately
show ?thesis
unfolding watchCharacterizationCondition-def
by auto
qed
}
thus ?thesis
unfolding InvariantWatchCharacterization-def
by auto
qed

lemma InvariantConsistentAfterApplyBackjump:
assumes
InvariantConsistent (\text{getM } \text{state})
InvariantUniq (\text{getM } \text{state})
InvariantWatchesEl (\text{getF } \text{state}) (\text{getWatch1 } \text{state}) (\text{getWatch2 } \text{state})
and
InvariantWatchListsContainOnlyClausesFromF (\text{getWatchList } \text{state}) (\text{getF } \text{state}) and
getConflictFlag state
InvariantCFalse (\text{getConflictFlag } \text{state}) (\text{getM } \text{state}) (\text{getC } \text{state})
and
  InvariantUniqC (getC state)
  InvariantCEntailed (getConflictFlag state) F0 (getC state) and
  InvariantCIClarification (getCI state) (getC state) (getM state)
and
  InvariantCIIClarification (getCI state) (getCl state) (getC state)
  (getM state) and
  InvariantClCurrentLevel (getCl 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

643
\textbf{InvariantUniqC} (getC state)
\textbf{InvariantCEntailed} (getConflictFlag state) \( F \theta \) (getC state) \textbf{and} 
\textbf{InvariantClCharacterization} (getCl state) (getC state) (getM state)
\textbf{and} 
\textbf{InvariantClCharacterization} (getCl state) (getCl state) (getC state)
(getM state) \textbf{and} 
\textbf{InvariantClCurrentLevel} (getCl state) (getM state)
\textbf{and} 
\textbf{InvariantClCharacterization} (getCl state) (getCl state) (getC state) (getM state) 
\textbf{and} 
\textbf{currentLevel} (getM state) > 0 
isUIP (opposite (getCl state)) (getC state) (getM state) 
\textbf{shows} 
\textbf{let} state' = applyBackjump state in 
\textbf{InvariantUniq} (getM state')
\textbf{proof} –
\textbf{let} ?l = getCl state 
\textbf{let} ?bClause = getC state 
\textbf{let} ?bLiteral = opposite ?l 
\textbf{let} ?level = getBackjumpLevel state 
\textbf{let} ?prefix = prefixToLevel ?level (getM state) 
\textbf{let} ?state'' = applyBackjump state
\textbf{have} clauseFalse (getC state) (elements (getM state))
\textbf{using} (getConflictFlag state)
\textbf{using} (\textbf{InvariantCFalse} (getConflictFlag state) (getM state) (getC state))
\textbf{unfolding} \textbf{InvariantCFalse-def}
\textbf{by} simp 
\textbf{have} isUnitClause ?bClause ?bLiteral (elements ?prefix) \textbf{and}
\textbf{getM} ?state'' = ?prefix @ (??bLiteral, False) \textbf{using} assms
\textbf{using} applyBackjumpEffect[of state] 
\textbf{by} (auto simp add: Let-def)
\textbf{thus} ?thesis 
\textbf{using} (\textbf{InvariantUniq} (getM state))
\textbf{using} isPrefixPrefixToLevel 
\textbf{by} (auto simp add: Let-def)
\textbf{qed}

\textbf{lemma} WatchInvariantsAfterApplyBackjump:
\textbf{assumes} 
\textbf{InvariantConsistent} (getM state)
\textbf{InvariantUniq} (getM state)
\textbf{InvariantWatchesEl} (getF state) (getWatch1 state) (getWatch2 state)
\textbf{and} 
\textbf{InvariantWatchesDiffer} (getF state) (getWatch1 state) (getWatch2 state) \textbf{and} 

644
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)

getConflictFlag state
InvariantUniqC (getC state)
InvariantCFalse (getConflictFlag state) (getM state) (getC state)
and
InvariantCEntailed (getConflictFlag state) F0 (getC state) and
InvariantCICharacterization (getC state) (getM state) and
InvariantClCurrentLevel (getC state) (getM state)

isUIP (opposite (getC state)) (getC state) (getM state)
currentLevel (getM state) > 0

shows
let state' = (applyBackjump state) in
  InvariantWatchesEl (getF state) (getWatchList state)' (getWatch1 state) (getWatch2 state)
  and
  InvariantWatchesDiffer (getF state) (getWatchList 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)
(is let state' = (applyBackjump state) in ?inv state')

proof
let ?l = getC state
let ?level = getBackjumpLevel state
let ?prefix = prefixToLevel ?level (getM state)
let ?state' = state[] getConflictFlag := False, getQ := [], getM := ?prefix ]
let ?state'' = setReason (opposite (getC state)) (length (getF state)
- 1) ?state'
let ?state0 = assertLiteral (opposite (getC state)) False ?state''

have getF ?state' = getF state getWatchList ?state' = getWatchList state
getWatch1 ?state' = getWatch1 state getWatch2 ?state' = 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 InvariantUniqQAfterApplyBackjump:
assumes
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state)
  (getF state) and
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
shows
  let state' = applyBackjump state in
  InvariantUniqQ (getQ state')

646
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 InvariantUniqQAfterAssertLiteral[of ?state' opposite ?l False]
using InvariantUniqQAfterAssertLiteral[of ?state'' opposite ?l 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)

647
InvariantCEntailed (getConflictFlag state) F0 (getC state) and
InvariantClCharacterization (getCl state) (getC state) (getM state)
and
InvariantClCharacterization (getCl state) (getCl 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
InvariantQCharacterization (getConflictFlag state"') (getQ state")
(getF 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 []
let ?state"" = setReason (opposite (getCl state)) (length (getF state) − 1) ?state'

let ?state'1 = assertLiteral (opposite ?l) False ?state'
let ?state"1 = assertLiteral (opposite ?l) False ?state"

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
hence InvariantQCharacterization (getConflictFlag ?state') (getQ
?state') (getF ?state') (getM ?state')
InvariantConflictFlagCharacterization (getConflictFlag ?state')
(getF ?state') (getM ?state')
unfolding InvariantQCharacterization-def
unfolding InvariantConflictFlagCharacterization-def
using ⟨InvariantNoDecisionsWhenConflict (getF state) (getM state)
(currentLevel (getM state))⟩
using ⟨InvariantNoDecisionsWhenUnit (getF state) (getM state)
(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 InvariantConsistentAfterApplyBackjumpPrefix[of state] using assms
by (simp add: Let-def)
moreover
have ¬ (opposite ?l el (getQ ?state′1) ¬ (opposite ?l el (getQ ?state"1'))
  using assertLiteralIsUnit[of ?state' opposite ?l False]
  using assertLiteralIsUnit[of ?state" opposite ?l False]
  using InvariantQCharacterization (getConflictFlag ?state') (getQ ?state') (getF ?state') (getM ?state')
  using InvariantConsistent (?prefix @ [(opposite ?l, False)])
  using InvariantWatchCharacterization (getF ?state) (getWatch1 ?state) (getWatch2 ?state)
  using InvariantWatchCharacterizationInBackjumpPrefix[of state]
  unfolding applyBackjump-def
  unfolding setReason-def
  by (auto simp add: Let-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
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 (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 $\neq \text{opposite} (getCl state)$
InvariantNoDecisionsWhenUnit (butlast (getF state)) (getM state)
(currentLevel (getM state))
InvariantNoDecisionsWhenConflict (butlast (getF state)) (getM state)
(currentLevel (getM state))
getF state $\neq []$
last (getF state) = getC 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)
currentLevel (getM state) > 0
isUIP (opposite (getCl state)) (getC state) (getM state)
shows
let state"" = (applyBackjump state) in
InvariantQCharacterization (getConflictFlag state") (getQ state")
(getF 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 :=

650
let $\text{state}'' = \text{setReason} \ (\text{opposite} \ (\text{getCl} \ \text{state})) \ (\text{length} \ (\text{getF} \ \text{state}) - 1) \ \text{?state}'$

have $\text{?level} < \text{elementLevel} \ ?l \ (\text{getM} \ \text{state})$
  using assms
  using $\text{isMinimalBackjumpLevelGetBackjumpLevel[of state]}$
  unfolding $\text{isMinimalBackjumpLevel-def}$
  unfolding $\text{isBackjumpLevel-def}$
  by (simp add: Let-def)

hence $\text{?level} < \text{currentLevel} \ (\text{getM} \ \text{state})$
  using $\text{elementLevelLeqCurrentLevel[of ?l getM state]}$
  by simp

have $\text{isUnitClause} \ (\text{last} \ (\text{getF} \ \text{state})) \ (\text{opposite} \ ?l) \ (\text{elements} \ ?\text{prefix})$
  using $\text{last} \ (\text{getF} \ \text{state}) = \text{getC} \ \text{state}$
  using $\text{isMinimalBackjumpLevelGetBackjumpLevel[of state]}$
  using $\text{InvariantUniq (getM state)}$
  using $\text{InvariantConsistent (getM state)}$
  using $\text{getConflictFlag state}$
  using $\text{InvariantUniqC (getC state)}$
  using $\text{InvariantCFalse (getConflictFlag state) (getM state) (getC state)}$
    using $\text{isBackjumpLevelEnsuresIsUnitInPrefix[of getM state getC state getBackjumpLevel state opposite ?l]}$
  using $\text{InvariantClCharacterization (getCl state) (getC state) (getM state)}$
    using $\text{InvariantClCharacterization (getCl state) (getClI state) (getC state) (getM state)}$
  using $\text{InvariantClCurrentLevel (getCl state) (getM state)}$
  using $\text{currentLevel} \ (\text{getM state}) > 0$
  using $\text{isUIP (opposite (getCl state)) (getC state) (getM state)}$
  unfolding $\text{isMinimalBackjumpLevel-def}$
  unfolding $\text{InvariantUniq-def}$
  unfolding $\text{InvariantConsistent-def}$
  unfolding $\text{InvariantCFalse-def}$
  by (simp add: Let-def)

hence $\neg \text{clauseFalse} \ (\text{last} \ (\text{getF} \ \text{state})) \ (\text{elements} \ ?\text{prefix})$
  unfolding $\text{isUnitClause-def}$
  by (auto simp add: clauseFalseIffAllLiteralsAreFalse)

have $\text{InvariantConsistent ( ?prefix @ [(opposite ?l, False)])}$
  using assms
  using $\text{InvariantConsistentAfterApplyBackjump[of state F0]}$
  using $\text{assertLiteralEffect}$
  unfolding $\text{applyBackjump-def}$
  unfolding $\text{setReason-def}$
  by (auto simp add: Let-def split: split-if-asm)
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)

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 \( \forall \, l \, c \, e \in (\text{butlast} (\text{getF} \, \text{state})) \rightarrow \neg \, \text{isUnitClause} \, c \, l \) (elements (getM ?state'))
    using (InvariantNoDecisionsWhenUnit (butlast (getF state))
    (getM state) (currentLevel (getM state)))
    using (?level < currentLevel (getM state))
    unfolding InvariantNoDecisionsWhenUnit-def
    by simp

  have \( \forall \, l \, (\exists \, c \, e \in (\text{getF} \, \text{state})) \wedge \text{isUnitClause} \, c \, l \) (elements (getM ?state')) = (l = opposite ?l)
proof
  fix l
  show (\exists \, c \, e \in (\text{getF} \, \text{state})) \wedge \text{isUnitClause} \, c \, l \) (elements (getM ?state')) = (l = opposite ?l) (\text{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 \( \forall \, l \, c \, e \in (\text{butlast} (\text{getF} \, \text{state})) \rightarrow \neg \, \text{isUnitClause} \, c \, l \) (elements (getM ?state')):
  using (isUnitClause c l (elements ?prefix))
  by auto
next
case False
from (getF state \# [])
have butlast (getF state) @ [last (getF state)] = getF state
using \texttt{append-butlast-last-id[off getF state]}
by \texttt{simp}
hence \texttt{getF state = butlast (getF state) @ [last (getF state)]}
by \texttt{(rule sym)}
with \texttt{(c el getF state)}
have \texttt{c el butlast (getF state) \lor c el [last (getF state)]}
using \texttt{set-append[off butlast (getF state) [last (getF state)]]}
by \texttt{auto}
hence \texttt{c = last (getF state)}
using \texttt{\neg c el (butlast (getF state))}
by \texttt{simp}
thus \texttt{\?thesis}
using \texttt{(isUnitClause (last (getF state)) (opposite \?l) (elements \?prefix))}
using \texttt{(isUnitClause c l (elements \?prefix))}
unfolding \texttt{isUnitClause-def}
by \texttt{auto}
qed
next
from \texttt{(getF state \neq \[])}
have \texttt{last (getF state) el (getF state)}
by \texttt{auto}
assume \texttt{\?rhs}
thus \texttt{\?rhs}
using \texttt{(isUnitClause (last (getF state)) (opposite \?l) (elements \?prefix))}
using \texttt{(last (getF state) el (getF state))}
by \texttt{auto}
qed
qed
thus \texttt{\?thesis}
unfolding \texttt{InvariantQCharacterization-def}
by \texttt{simp}
qed
hence \texttt{InvariantQCharacterization (getConflictFlag ?state'1) (getQ ?state'1) (getF ?state'1) (getM ?state'1)}
by \texttt{simp}
hence \texttt{InvariantQCharacterization (getConflictFlag ?state''1) (getQ ?state''1) (getF ?state''1) (getM ?state''1)}
unfolding \texttt{setReason-def}
by \texttt{simp}

have \texttt{InvariantWatchCharacterization (getF ?state'1) (getWatch1 ?state'1) (getWatch2 ?state'1) (getM ?state'1)}
using \texttt{InvariantWatchCharacterizationInBackjumpPrefix[off state]}
using \texttt{assms}
by \texttt{(simp add: Let-def)}
hence \texttt{InvariantWatchCharacterization (getF ?state''1) (getWatch1 ?state''1) (getWatch2 ?state''1) (getM ?state''1)}
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'') (getF ?state'') (getM ?state'')
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

654
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 ⟨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 ⟨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)
  unfolding setReason-def 
  by (simp add: Let-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))
  using arg-cong[of state (getConflictFlag := False, getQ := [], getM := ?prefix)]
  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 AssertLiteralStartQirrelevant[of ?state01 opposite ?l False []]
  unfolding InvariantWatchesEl (getF ?state01) (getWatch1 ?state01) (getWatch2 ?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 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
  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
  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)
next
case True
let ?state02 = setReason (opposite (getCl state)) (length (getF state) − 1)
  state[(getConflictFlag := False, getM := ?prefix)]
have InvariantWatchesEl (getF ?state02) (getWatch1 ?state02) (getWatch2 ?state02)
  using InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
state)
    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),
    getQ := [] \}

    let ?stateTmp'' = assertLiteral (opposite (getCl state)) False
    (setReason (opposite (getCl state))) (length (getF state) - 1)
    state \{ getConflictFlag := False,
    getM := prefixToLevel (getBackjumpLevel state) (getM state),
    getQ := [opposite (getCl state)] \}

    have getM ?stateTmp' = getM ?stateTmp''
    getF ?stateTmp' = getF ?stateTmp''
    getSATFlag ?stateTmp' = getSATFlag ?stateTmp''
    getConflictFlag ?stateTmp' = getConflictFlag ?stateTmp''
    using AssertLiteralStartQIrrelevant[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,
    getQ := [],
    getM := ?prefix,
    getReason := getReason state(opposite ?l \rightarrow

length (getF state) - 1)
state

\[
\begin{align*}
\text{getReason} & := \text{getReason state}(\text{opposite } \text{?}\text{l} \mapsto \text{length })(\text{getF state}) - 1), \\
\text{getConflictFlag} & := \text{False}, \\
\text{getM} & := \text{prefixToLevel } (\text{getBackjumpLevel state })(\text{getM state}), \\
\text{getQ} & := []
\end{align*}
\]

\[\lambda x. \text{assertLiteral } (\text{opposite } \text{?}\text{l} ) \text{ False } x\]

unfolding applyBackjump-def

by (auto simp add: Let-def)

moreover

have \(\text{?state}\text{Tmp}'' = \text{?state}''2\)

unfolding setReason-def

using arg-cong[of state \(\{\text{getReason} := \text{getReason state}(\text{opposite } \text{?}\text{l} \mapsto \text{length })(\text{getF state}) - 1), \\
\text{getConflictFlag} := \text{False}, \\
\text{getM} := \text{prefixToLevel } (\text{getBackjumpLevel state })(\text{getM state})\}\) state

getReason := getReason state(\text{opposite } \text{?}\text{l} \mapsto \text{length })(\text{getF state}) - 1), \\
\text{getConflictFlag} := \text{False}, \\
\text{getM} := \text{prefixToLevel } (\text{getBackjumpLevel state })(\text{getM state})\) state

getReason := getReason state(\text{opposite } \text{?}\text{l} \mapsto \text{length })(\text{getF state}) - 1), \\
\text{getConflictFlag} := \text{False}, \\
\text{getM} := \text{prefixToLevel } (\text{getBackjumpLevel state })(\text{getM state})\)

by simp

ultimately

have \(\text{getConflictFlag } \text{?stateB} = \text{getConflictFlag } \text{?state}''2\)

getF \text{?stateB} = getF \text{?state}''2

getM \text{?stateB} = getM \text{?state}''2

by auto

have \(\text{set } (\text{getQ } \text{?stateB}) = \text{set } (\text{removeAll } (\text{opposite } \text{?}\text{l}) (\text{getQ } \text{?state}''2))\)

proof–

have \(\text{set } (\text{getQ } \text{?stateB}) = \text{set}(\text{getQ } \text{?state}''2) - \{\text{opposite } \text{?}\text{l}\}\)

proof–

let \(\text{ulSet} = \{\text{ul. } (\exists \text{ uc } . \text{ uc el } (\text{getF } \text{?state}''1) \wedge \\
\text{?l el uc } \wedge \\
isUnitClause uc ul ((\text{elements } (\text{getM } \text{?state}''1)) @ [\text{opposite } \text{?}\text{l}])\}\)

have \(\text{set } (\text{getQ } \text{?state}''2) = \{\text{opposite } \text{?}\text{l}\} \cup \text{ulSet}\)

using assertLiteralQEffect[of \text{?state}''1 \text{ opposite } \text{?}\text{l} \text{ False}] use assms

using \(\langle \text{InvariantConsistent } (\text{?prefix } \wedge [\text{opposite } \text{?}\text{l}, \text{ False}])\rangle\)

using \(\langle \text{InvariantUniq } (\text{?prefix } \wedge [\text{opposite } \text{?}\text{l}, \text{ False}])\rangle\)

using \(\langle \text{InvariantWatchCharacterization } (\text{getF } \text{?state}''1)\rangle\)
\begin{verbatim}
(setWatch1 ?\text{state}''1) (setWatch2 ?\text{state}''1) (getM ?\text{state}''1)

unfolding setReason-def
by (simp add: Let-def)
moreover
have set (getQ ?\text{state}B) = ?ulSet
using assertLiteralQEffect[of ?\text{state}'' opposite ?l False]
using assms
using (InvariantConsistent (?prefix @ [(opposite ?l, False)]))
using (InvariantUniq (?prefix @ [(opposite ?l, False)]))
using (InvariantWatchCharacterization (getF ?\text{state}'' (getWatch1 ?\text{state}'') (getWatch2 ?\text{state}'') (getM ?\text{state}'')))
using (getBackjumpLevel state > 0)
unfolding applyBackjump-def
unfolding setReason-def
by (simp add: Let-def)
moreover
have \neg (opposite ?l) \in ?ulSet
using assertedLiteralIsNotUnit[of ?\text{state}'' opposite ?l False]
using assms
using (InvariantConsistent (?prefix @ [(opposite ?l, False)]))
using (InvariantUniq (?prefix @ [(opposite ?l, False)]))
using (InvariantWatchCharacterization (getF ?\text{state}'' (getWatch1 ?\text{state}'') (getWatch2 ?\text{state}'') (getM ?\text{state}'')))
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 (setConflictFlag ?\text{state}''2)
(removeAll (opposite ?l) (getQ ?\text{state}''2)) (getF ?\text{state}''2) (getM ?\text{state}''2)
using (set (getQ ?\text{state}B) = set (removeAll (opposite ?l) (getQ ?\text{state}''2)))
using (getConflictFlag ?\text{state}B = getConflictFlag ?\text{state}''2)
using (getF ?\text{state}B = getF ?\text{state}''2)
using (getM ?\text{state}B = getM ?\text{state}''2)
unfolding InvariantQCharacterization-def
by (simp add: Let-def)
qed
qed
\end{verbatim}
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)
  InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state) and
  InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state) and
  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) (getC state) (getM state) and
  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' = state[] getConflictFlag := False, getQ := [], getM := ?prefix []
  let ?state'' = setReason (opposite ?l) (length (getF state) - 1) ?state'
  let ?stateB = applyBackjump state

  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 elementLeqCurrentLevel[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
  unfolding InvariantConsistentAfterApplyBackjump[of state F0]
  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'']
  unfolding 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)

662
state) and
InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state) and

InvariantUniqC (getC state)
getC state ≠ [opposite (getCl state)]
InvariantNoDecisionsWhenConflict (butlast (getF state)) (getM state)
(currentLevel (getM state))
getF state ≠ [] last (getF state) = getC 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)

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' = state{| getConflictFlag := False, getQ := [], getM := ?prefix |
let $?state'' = setReason (opposite ?l) (length (getF state) − 1) $?state'
let $?stateB = applyBackjump state

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

hence InvariantConflictFlagCharacterization (getConflictFlag state') (butlast (getF state')) (getM state')
using InvariantNoDecisionsWhenConflict (butlast (getF state))
(getM state) (currentLevel (getM state))
  unfolding InvariantNoDecisionsWhenConflict-def
  unfolding InvariantConflictFlagCharacterization-def
  by simp

moreover
  have isBackjumpLevel (getBackjumpLevel state) (opposite (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 (getC state)
    using (InvariantCFalse (getC 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 (getC state) (getM state) (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-asm)
ultimately

show thesis
  using InvariantConflictFlagCharacterizationAfterAssertLiteral[of ?state]
  using InvariantConflictFlagCharacterizationAfterAssertLiteral[of ?state]
  using InvariantWatchCharacterizationInBackjumpPrefix[of state]
  using assms
  unfolding 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 = 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'
    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
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)
  (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 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
  
  \{ 
    \begin{align*}
      & \text{fix } l::\text{Literal} \\
      & \text{assume } \ast \colon l \in (\text{elements } ?prefix) \land \neg l \in (\text{decisions } ?prefix) \land \\
      & \quad \text{elementLevel } l \in (\text{elements } \text{getM state}) \land \neg l \in (\text{decisions } \text{getM state}) \land elementLevel l \in \text{getM state} > 0 \\
      & \quad \text{using } (\text{InvariantUniq } \text{getM state}) \\
      & \quad \text{unfolding } \text{InvariantUniq-def} \\
      & \quad \text{using } \text{isPrefixPrefixToLevel[of } ?level \text{ getM state]} \\
      & \quad \text{using } \text{isPrefixElements[of } ?prefix \text{ getM state]} \\
      & \quad \text{using } \text{prefixIsSubset[of elements } ?prefix \text{ elements getM state]} \\
      & \quad \text{using } \text{markedElementsTrailMemPrefixAreMarkedElementsPrefix[of getM state } ?prefix l] \\
      & \quad \text{using } \text{elementLevelPrefixElement[of } l \text{ getBackjumpLevel state getM state]} \\
      & \quad \text{by auto} \\
    \end{align*} \\
    \text{with } \text{assms} \\
    \begin{align*}
      & \text{obtain } \text{reason} \\
      & \quad \begin{cases}
        \text{where } \text{reason } < \text{length } (\text{getF state}) \text{ isReason (nth } \text{getF state} \\
        \text{ reason) } l \in (\text{elements } \text{getM state}) \\
        \text{getReason state } l = \text{Some reason} \\
      \end{cases} \\
      & \quad \text{unfolding } \text{InvariantGetReasonIsReason-def} \\
      & \quad \text{by auto} \\
      & \quad \text{hence } \exists \text{ reason. getReason state } l = \text{Some reason} \land \\
      & \quad \quad \text{reason } < \text{length } (\text{getF state}) \land \\
      & \quad \quad \quad \text{isReason (nth } \text{getF state} \text{ reason) } l \in (\text{elements } ?prefix) \\
      & \quad \quad \quad \text{using } \text{isReasonHoldsInPrefix[of } l \text{ elements } ?prefix \text{ elements getM state} \\
      & \quad \quad \quad \text{using } \text{isPrefixPrefixToLevel[of } ?level \text{ getM state]} \\
      & \quad \quad \quad \text{using } \text{isPrefixElements[of } ?prefix \text{ getM state]} \\
      & \quad \quad \quad \text{using } \ast \\
      & \quad \quad \quad \text{by auto} \\
    \end{align*} \\
  \}

  \text{thus } ?\text{thesis} \\
  \text{unfolding } \text{InvariantGetReasonIsReason-def} \\
  \text{by auto} \\
\text{qed}

  \begin{align*}
    \text{let } \text{?stateM } = \text{?state'' [ getM := getM ?state'' @ [(opposite } ?l, False)]} \\
  \end{align*}

  \text{have } \ast\ast \colon \text{getM } ?\text{stateM } = ?\text{prefix } @ [(\text{opposite } ?l, \text{False})] \\
  \text{getF } ?\text{stateM } = \text{getF state} \\
  \text{getQ } ?\text{stateM } = [] \\
  \text{getWatchList } ?\text{stateM } = \text{getWatchList state}

  \text{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

    obtain reason
where reason < length (getF state)
isReason (nth (getF state) reason) l (elements ?prefix)
getReason state l = Some reason
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 ?b
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 [] \land \\
last (\text{getF state}) = \text{getC state} \\
using \text{last-conv-nth}[\text{of getF state}] \\
by \text{simp} \\

hence \text{isUnitClause} (\text{nth} (\text{getF state}) \ ?\text{reason}) \ l \ (\text{elements} \ ?\text{prefix}) \\
using \text{assms} \\
using \text{applyBackjumpEffect}[\text{of state F0}] \\
using \text{(simp add: Let-def)} \\

hence \text{isReason} (\text{nth} (\text{getF state}) \ ?\text{reason}) \ l \ (\text{elements} \ (\text{getM} \ ?\text{stateM})) \\
using ** \\
using \text{isUnitClauseIsReason}[\text{of nth} (\text{getF state}) \ ?\text{reason} \ l \ \text{elements} \ ?\text{prefix} \ [\text{opposite} ?l]] \\
using \text{(simp add: Let-def)} \\

moreover \\
have \ ?\text{reason} < \text{length} (\text{getF state}) \\
using \ (\text{?level} \neq 0) \\
using \ (\text{getBackjumpLevel state} > 0 \rightarrow \text{getF state} \neq [] \land \\
last (\text{getF state}) = \text{getC state}) \\
by \text{simp} \\
ultimately \\
show \ ?\text{thesis} \\
using \ (\text{?level} \neq 0) \\
using \ (\text{opposite} ?l) \\
using ** \\
by \text{auto} \\
qed \\

\text{thus} \ ?\text{thesis} \\
\text{unfolding} \ \text{InvariantGetReasonIsReason-def} \\
\text{unfolding} \ \text{setReason-def} \\
by \text{auto} \\
\text{qed} \\

\text{thus} \ ?\text{thesis} \\
\text{using} \ \text{InvariantGetReasonIsReasonAfterNotifyWatches}[\text{of} \ ?\text{stateM}] \\
\text{getWatchList} \ ?\text{stateM} \ ?l \ ?l \ ?\text{prefix} \ False \ {[]} \\
\text{by} \ \text{auto} \\
\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{by} \ \text{auto} \\
\text{unfolding} \ \text{InvariantWatchListsCharacterization-def} \\

670
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)
  getC 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)

  getConflictFlag 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 ¬ formulaFalse (getF ?state') (elements (prefixToLevel level (getM ?state')))) (is ?false)
using ⟨InvariantNoDecisionsWhenConflict (getF state) (getM state) (currentLevel (getM state))⟩
unfolding InvariantNoDecisionsWhenConflict-def
using ⟨level < ?level⟩
using ⟨?level < currentLevel (getM state)⟩
using prefixToLevelPrefixToLevelHigherLevel[of level ?level getM state, THEN sgm]
using ⟨getF ?state' = getF state⟩
using ⟨prefixToLevel level (getM ?state') = prefixToLevel level ?prefix⟩
using prefixToLevelPrefixToLevelHigherLevel[of level ?level getM state, THEN sgm]
by (auto simp add: formulaFalseIffContainsFalseClause)

moreover
have ¬ (∃ clause literal.
  clause el (getF ?state') ∧
  isUnitClause clause literal (elements (prefixToLevel level (getM ?state')))) (is ?unit)
using ⟨InvariantNoDecisionsWhenUnit (getF state) (getM state) (currentLevel (getM state))⟩
unfolding InvariantNoDecisionsWhenUnit-def
using ⟨level < ?level⟩
using ⟨?level < currentLevel (getM state)⟩
using ⟨getF ?state' = getF state⟩
using ⟨prefixToLevel level (getM ?state') = prefixToLevel level ?prefix⟩
using prefixToLevelPrefixToLevelHigherLevel[of level ?level getM state, THEN sgm]
by simp
ultimately
have ?false ∧ ?unit
  by simp

} thus ?thesis
unfolding InvariantNoDecisionsWhenConflict-def
unfolding InvariantNoDecisionsWhenUnit-def
by (auto simp add: Let-def)
qed

lemma InvariantsNoDecisionsWhenConflictNorUnitAfterApplyBackjump-2:
assumes
  InvariantConsistent (getM 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
InvariantClCurrentLevel (getCl state) (getC state)
(getM state) and

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 \(?\text{level} < \text{elementLevel} \ ?l\) (getM state)
  using assms
  using isMinimalBackjumpLevelGetBackjumpLevel[of state]
  unfolding isMinimalBackjumpLevel-def
  unfolding isBackjumpLevel-def
  by (simp add: Let-def)

hence \(?\text{level} < \text{currentLevel} \ (\text{getM state})\)
  using elementLevelLeqCurrentLevel[of \(?l\) getM state]
  by simp

have \(\text{currentLevel} \ (\text{getM ?state'}) = \text{currentLevel} \ ?\text{prefix}\)
  using \(\langle \text{getM ?state'} = ?\text{prefix} \ @ \ [(?b\text{Literal}, \text{False})] \rangle\)
  using markedElementsAppend[of \(?\text{prefix} \ [(?b\text{Literal}, \text{False})]\)]
  unfolding currentLevel-def
  by simp

hence \(\text{currentLevel} \ (\text{getM ?state'}) \leq \ ?\text{level}\)
  using currentLevelPrefixToLevel[of \(?l\) getM state]
  by simp

show \(?\text{thesis}\)
proof

  \{ 

  fix level
  assume level < currentLevel \ (\text{getM ?state'})
  hence level < currentLevel \ ?\text{prefix}
    using \(\langle \text{currentLevel} \ (\text{getM ?state'}) = \text{currentLevel} \ ?\text{prefix}\ \rangle\)
    by simp
  hence prefixToLevel level \ (\text{getM} \ (\text{applyBackjump state})) = prefixToLevel level \ ?\text{prefix}
    using \(\langle \text{getM ?state'} = ?\text{prefix} \ @ \ [(?b\text{Literal}, \text{False})] \rangle\)
    using prefixToLevelAppend[of level \ ?\text{prefix} \ [(?b\text{Literal}, \text{False})]]
    by simp
  have level < \ ?\text{level}
    using \(\langle \text{level} < \text{currentLevel} \ ?\text{prefix}\ \rangle\)
    using \(\langle \text{currentLevel} \ (\text{getM ?state'}) \leq \ ?\text{level}\ \rangle\)
    using \(\langle \text{currentLevel} \ (\text{getM ?state'}) = \text{currentLevel} \ ?\text{prefix}\ \rangle\)
    by simp
  hence prefixToLevel level \ (\text{getM ?state'}) = prefixToLevel level \ ?\text{prefix}
    using \(\langle \text{getM ?state'} = ?\text{prefix} \ @ \ [(?b\text{Literal}, \text{False})] \rangle\)
    using prefixToLevelAppend[of level \ ?\text{prefix} \ [(?b\text{Literal}, \text{False})]]
    by simp

  have \(\neg \text{formulaFalse} \ (\text{butlast} \ (\text{getF ?state'})) \ (\text{elements} \ (\text{prefixToLevel level} \ (\text{getM ?state'})))\)
    using \(\langle \text{getF ?state'} = \text{getF state} \rangle\)

\}

675
```
using ⟨InvariantNoDecisionsWhenConflict (butlast (getF state)) (getM state) (currentLevel (getM state))⟩
using ⟨level < level⟩
using ⟨?level < currentLevel (getM state)⟩
using ⟨prefixToLevel level (getM ?state') = prefixToLevel level⟩

using prefixToLevelPrefixToLevelHigherLevel[of level ?level getM state, THEN sym]
unfolding InvariantNoDecisionsWhenConflict-def
by (auto simp add: formulaFalseIffContainsFalseClause)
moreover
have ¬ clauseFalse (last (getF ?state')) (elements (prefixToLevel level (getM ?state')))
  using ⟨getF ?state' = getF state⟩
  using ⟨InvariantNoDecisionsWhenConflict [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 InvariantNoDecisionsWhenConflict-def
by (auto simp add: formulaFalseIffContainsFalseClause)
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 ¬ formulaFalse (getF ?state') (elements (prefixToLevel level (getM ?state')) (is ?false)
  using ⟨getF ?state' = getF state⟩
  using set-append[of butlast (getF state) [last (getF state)]]
  by (auto simp add: formulaFalseIffContainsFalseClause)

have ¬ (∃ clause literal.
  clause el (butlast (getF ?state')) ∧
  isUnitClause clause literal (elements (prefixToLevel level (getM ?state'))))
  using ⟨InvariantNoDecisionsWhenUnit (butlast (getF state)) (getM state) (currentLevel (getM state))⟩
  unfolding InvariantNoDecisionsWhenUnit-def
  using ⟨level < level⟩
  using ⟨?level < currentLevel (getM state)⟩
  using ⟨getF ?state' = getF state⟩
  using ⟨prefixToLevel level (getM ?state') = prefixToLevel level⟩
```

676
using \textit{prefixToLevel} \textit{prefixToLevelHigherLevel}[of level ?level
\textit{getM} state, \textit{THEN} \textit{sym}]
by simp
moreover
have \(\neg (\exists \ l. \text{isUnitClause} \ (\text{last} \ (\text{getF} \ ?\text{state}')) \ l \ (\text{elements} \ (\text{prefixToLevel} \ \textit{level} \ (\text{getM} \ ?\text{state}'))))\)
using \(\text{getF} \ ?\text{state}' = \text{getF} \text{state}\)
using \(\text{InvariantNoDecisionsWhenUnit} \ \textit{getC} \text{state} \ (\text{getM} \text{state}) \ (\text{getBackjumpLevel} \text{state})\)
using \(\text{last} \ (\text{getF} \text{state}) = \text{getC} \text{state}\)
using \(\text{level} < ?\text{level}\)
using \(\text{prefixToLevel} \ \textit{level} \ (\text{getM} \ ?\text{state}') = \text{prefixToLevel} \ \textit{level} \ ?\text{prefix}\)
unfolding \textit{InvariantNoDecisionsWhenUnit-def}
by simp
moreover
from \(\text{getF} \text{state} \neq []\)
have \(\text{butlast} \ (\text{getF} \text{state}) \ @ [\text{last} \ (\text{getF} \text{state})] = \text{getF} \text{state}\)
using \textit{append-butlast-last-id}[of \textit{getF} \text{state}]
by simp
hence \(\text{getF} \text{state} = \text{butlast} \ (\text{getF} \text{state}) \ @ [\text{last} \ (\text{getF} \text{state})]\)
by (rule \textit{sym})
ultimately
have \(\neg (\exists \ \textit{clause} \ \textit{literals}. \ \text{clause} \ \textit{el} \ (\text{getF} \ ?\text{state}') \ \land \ \text{isUnitClause} \ \textit{clause} \ \textit{literals} \ (\text{elements} \ (\text{prefixToLevel} \ \textit{level} \ (\text{getM} \ ?\text{state}'))))\) \((is \ ?unit)\)
using \(\text{getF} \ ?\text{state}' = \text{getF} \text{state}\)
using \textit{set-append}[of \textit{butlast} \ (\text{getF} \text{state}) [\text{last} \ (\text{getF} \text{state})]]
by auto
have \?false \land ?unit
using \(\langle \text{?false} \rangle \langle \text{?unit} \rangle\)
by simp
\}
thus \?thesis
unfolding \textit{InvariantNoDecisionsWhenConflict-def}
unfolding \textit{InvariantNoDecisionsWhenUnit-def}
by (auto simp add: \textit{Let-def})
qed
qed

lemma \textit{InvariantEquivalentZLAfterApplyBackjump}:
assumes
\textit{InvariantConsistent} \ (\textit{getM} \text{state})

677
InvariantUniq \((\text{getM state})\)
\(\text{InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)}\)
and
\(\text{InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)}\) and
\(\text{getConflictFlag state}\)
\(\text{InvariantUniqC (getC state)}\)
\(\text{InvariantCFalse (getConflictFlag state) (getM state) (getC state)}\)
and
\(\text{InvariantCEntailed (getConflictFlag state) F0 (getC state) and}\)
\(\text{InvariantClCharacterization (getCl state) (getC state) (getM state)}\) and
\(\text{InvariantClCurrentLevel (getCl state) (getM state)}\)
\(\text{InvariantEquivalentZL (getF state) (getM state) F0}\)
\(\text{isUIP (opposite (getCl state)) (getC state) (getM state) currentLevel (getM state) > 0}\)
shows
\(\text{let state' = applyBackjump state in}\)
\(\text{InvariantEquivalentZL (getF state') (getM state') F0}\)

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 \(?bClause\)
  isUnitClause \(?bClause\) \(?bLiteral\) (elements \(?prefix\))
  getM ?state' = \(?prefix\) @ \([\{(\?bLiteral, False)\}]\)
  getF ?state' = getF state
using assms
using applyBackjumpEffect[of state F0]
by (auto simp add: Let-def)

note * = this
show ?thesis

proof (cases ?level = 0)
case False
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
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

next
case True
hence prefixToLevel 0 (getM ?state') = ?prefix @ [(?bLiteral, False)]
  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 equivalentFormulas-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: formulaTrue Append)
  hence literalTrue ?bLiteral valuation
  using ⟨model valuation F0⟩
  unfolding formulaEntailsLiteral (F0 @ val2form (elements ?prefix)) ?bLiteral
  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)]⟩

680
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)
  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
  InvariantCliCharacterization (getCl state) (getC state) (getM state)
and
  InvariantCliCharacterization (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

681
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 = getC 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
    unfolding literalElListIffOppositeLiteralElOppositeLiteralList[of ?bLiteral getC state]
    by simp

682
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]
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
¬ vars(elements (getM state)) ⊇ Vbl and
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchListsContainOnlyClausesFromF (getWatchList state)
(getF state)
shows
let literal = selectLiteral state Vbl in
let state' = applyDecide state Vbl in
  var literal /∈ vars(elements (getM state)) ∧
  var literal ∈ Vbl ∧
  getM state' = getM state ® [(literal, True)] ∧
  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
¬ vars(elements (getM state)) ⊇ Vbl and
InvariantConsistent (getM state) and
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchListsContainOnlyClausesFromF (getWatchList state)
(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
case False
  hence prefixToLevel 0 (getM state @ [(?l, True)]) =
    getM state @ (prefixToLevel-aux [(?l, True)] 0 (currentLevel (getM state)));
  using prefixToLevelAppend[of 0 getM state [(?l, True)]]
  by simp
  hence prefixToLevel 0 (getM state @ [(?l, True)]) = getM state
  by simp
  thus ?thesis
  using (getM ?state' = getM state @ [(?l, True)])
  by simp
qed

thus ?thesis
  unfolding InvariantEquivalentZL-def
  using ⟨getF ?state' = getF state⟩
  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)])
    proof−
      { fix l::Literal
        assume *: l el (elements (getM ?stateM)) ¬ l el (decisions (getM ?stateM)) elementLevel l (getM ?stateM) > 0

687
have ∃ reason. getReason ?stateM l = Some reason ∧
0 ≤ reason ∧ reason < length (getF ?stateM) ∧
isReason (getF ?stateM ! reason) l (elements (getM ?stateM))

proof
  (cases l el (elements (getM state)))
  case True
  moreover
  hence ¬ l el (decisions (getM state))
  using *
  by (simp add: markedElementsAppend)
  moreover
  have elementLevel l (getM state) > 0
  proof
    { assume ¬ ?thesis
      with *
      have l = ?l
        using True
        using elementLevelAppend [of l getM state [(l, True)]]
        by simp
      hence var ?l ∈ vars (elements (getM state))
        using True
        using valuationContainsItsLiteralsVariable [of l elements (getM state)]
        by simp
      hence False
        using ⟨ ¬ vars (elements (getM state)) ⊇ Vbl ⟩
        unfolding selectLiteral-def
        by auto
    } thus ?thesis
    by auto
  qed
  ultimately
  obtain reason
  where getReason state l = Some reason ∧
  0 ≤ reason ∧ reason < length (getF state) ∧
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 ?thesis
  using isReasonAppend [of nth (getF ?stateM) reason l elements (getM state) [(l, True)]]
  by auto
  next
  case False
  hence l = ?l
  using *
  by auto

688
hence \( l \) el \((\text{decisions \( (\text{getM} \ ?\text{stateM}) \))}\)
using markedElementIsMarkedTrue\([\text{of} \ l \ \text{getM} \ ?\text{stateM}]\)
by auto
with *
have False
by auto
thus \(?\text{thesis}\)
by simp
qed

\}
thus \(?\text{thesis}\)
using \((\text{getQ} \ \text{state} = [])\)
unfolding InvariantGetReasonIsReason-def
by auto
qed
thus \(?\text{thesis}\)
using \(\text{assms}\)
using InvariantGetReasonIsReasonAfterNotifyWatches\([\text{of} \ ?\text{stateM} \text{getWatchList} \ ?\text{stateM} \ (\text{opposite} \ ?l)\text{opposite} \ ?l \text{getM} \ \text{state} \ True \ \{} \ \\[]\)\text{applyDecide-def}\nunfolding assertLiteral-def
unfolding notifyWatches-def
unfolding InvariantWatchListsCharacterization-def
unfolding InvariantWatchListsContainOnlyClausesFromF-def
unfolding InvariantWatchListsUniq-def
using \((\text{getQ} \ \text{state} = [])\)
by (simp add: \text{Let-def})
qed

lemma InvariantsVarsAfterApplyDecide:
assumes
\(\neg \ \text{vars} \ (\text{elements} \ (\text{getM} \ \text{state})) \supseteq Vbl\)
InvariantConsistent \((\text{getM} \ \text{state})\)
InvariantUniq \((\text{getM} \ \text{state})\)
InvariantWatchListsContainOnlyClausesFromF \((\text{getWatchList} \ \text{state}) \ (\text{getF} \ \text{state})\)
InvariantWatchListsUniq \((\text{getWatchList} \ \text{state})\)
InvariantWatchListsCharacterization \((\text{getWatchList} \ \text{state}) \ (\text{getWatch1} \ \text{state}) \ (\text{getWatch2} \ \text{state})\)
InvariantWatchesEl \((\text{getF} \ \text{state}) \ (\text{getWatch1} \ \text{state}) \ (\text{getWatch2} \ \text{state})\)
InvariantWatchesDiffer \((\text{getF} \ \text{state}) \ (\text{getWatch1} \ \text{state}) \ (\text{getWatch2} \ \text{state})\)
InvariantWatchCharacterization \((\text{getF} \ \text{state}) \ (\text{getWatch1} \ \text{state}) \ (\text{getWatch2} \ \text{state}) \ (\text{getM} \ \text{state})\)
InvariantVarsM \((\text{getM} \ \text{state}) \ \text{F0} \ Vbl\)
InvariantVarsF \((\text{getF} \ \text{state}) \ \text{F0} \ Vbl\)
\text{getQ} \ \text{state} = []
689
shows
let state′ = applyDecide state Vbl in
InvariantVarsM (getM state′) F0 Vbl ∧
InvariantVarsF (getF state′) F0 Vbl ∧
InvariantVarsQ (getQ state′) F0 Vbl

proof
let ?state′ = applyDecide state Vbl
let ?l = selectLiteral state Vbl

have InvariantVarsM (getM ?state′) F0 Vbl
  InvariantVarsF (getF ?state′) F0 Vbl
using assms
using applyDecideEffect[of Vbl state]
using varsAppendValuation[of elements (getM state) [?l]]
unfolding InvariantVarsM-def
by (auto simp add: Let-def)

moreover
have InvariantVarsQ (getQ ?state′) F0 Vbl
using InvariantVarsQAfterAssertLiteral[of state ?l True F0 Vbl]
using assms
using InvariantConsistentAfterApplyDecide[of Vbl state]
using InvariantUniqAfterApplyDecide[of Vbl state]
using assertLiteralEffect[of state ?l True]
unfolding applyDecide-def
unfolding InvariantVarsQ-def
by (simp add: Let-def)

ultimately
show ?thesis
  by (simp add: Let-def)

qed
end

theory SolveLoop
imports UnitPropagate ConflictAnalysis Decide
begin

lemma soundnessForUNSAT:
assumes
  equivalentFormulae (F @ val2form M) F0
  formulaFalse F M
shows
  ¬ satisfiable F0
proof 
  have formulaEntailsValuation (F val2form M) M
    using val2formIsEntailed[of F M []]
    by simp
  moreover
  have formulaFalse (F val2form M) M
    using formulaFalse F M
    by (simp add: formulaFalseAppend)
  ultimately
  have ¬ satisfiable (F val2form M)
    using formulaFalseInEntailedValuationIsUnsatisfiable[of F val2form M]
    by simp
thus thesis
  using equivalentFormulae (F val2form M) F0
  by (simp add: satisfiableEquivalent)
qed

lemma soundnessForSat:
  fixes F0 :: Formula and F :: Formula and M :: LiteralTrail
  assumes vars F0 ⊆ Vbl and InvariantVarsF F F0 Vbl and InvariantConsistent M and InvariantEquivalentZL F M F0 and
    ¬ formulaFalse (elements M) and vars (elements M) ⊇ Vbl
  shows model (elements M) F0
proof 
  from (InvariantConsistent M)
  have consistent (elements M)
    unfolding InvariantConsistent-def
  .
  moreover
  from (InvariantVarsF F F0 Vbl)
  have vars F ⊆ vars F0 ∪ Vbl
    unfolding InvariantVarsF-def
  .
  with vars F0 ⊆ Vbl
  have vars F ⊆ Vbl
    by auto
  with vars (elements M) ⊇ Vbl
  have vars F ⊆ vars (elements M)
    by simp
  hence formulaTrue (elements M) ∨ formulaFalse (elements M)
    by (simp add:totalValuationForFormulaDefinesItsValue)
  with ¬ formulaFalse (elements M)
  have formulaTrue (elements M)
    by simp
  ultimately
  have model (elements M) F
    by simp
  moreover
obtain $s$ where $\text{elements} (\text{prefixToLevel} \ 0 \ M) @ s = \text{elements} M$
using $\text{isPrefixPrefixToLevel}[\text{of} \ 0 \ M]$
using $\text{isPrefixElements}[\text{of} \ \text{prefixToLevel} \ 0 \ M \ M]$
unfolding $\text{isPrefix-def}$ by auto

hence $\text{elements} M = \text{elements} (\text{prefixToLevel} \ 0 \ M) @ s$
by (rule $\text{sym}$)

hence $\text{formulaTrue} (\text{val2form} (\text{elements} (\text{prefixToLevel} \ 0 \ M))) (\text{elements} M)$
using $\text{val2formFormulaTrue}[\text{of} \ \text{elements} (\text{prefixToLevel} \ 0 \ M) \ \text{elements} M]$
by auto

hence $\text{model} (\text{elements} M) (\text{val2form} (\text{elements} (\text{prefixToLevel} \ 0 \ M)))$
using $\langle \text{consistent} (\text{elements} M) \rangle$
by simp

ultimately
show $\text{?thesis}$
using $\langle \text{InvariantEquivalentZL} \ F \ M \ F0 \rangle$
unfolding $\text{InvariantEquivalentZL-def}$
unfolding $\text{equivalentFormulae-def}$
using $\text{formulaTrueAppend}[\text{of} \ F \ \text{val2form} (\text{elements} (\text{prefixToLevel} \ 0 \ M)) \ \text{elements} M]$
by auto

qed

definition
satFlagLessState = \{(state1::\text{State}, state2::\text{State}). (\text{getSATFlag} state1) \neq \text{UNDEF} \land (\text{getSATFlag} state2) = \text{UNDEF}\}$

lemma wellFoundedSatFlagLessState:
shows $\text{wf} \ \text{satFlagLessState}$
unfolding $\text{wf-eq-minimal}$

proof –
show $\forall Q \ \text{state}. \ \text{state} \in Q \longrightarrow (\exists \text{stateMin} \in Q. \ \forall \text{state}'. (\text{state}', \text{stateMin}) \in \text{satFlagLessState} \longrightarrow \text{state}' \notin Q)$
proof –
{ fix state::\text{State} and Q::\text{State set}
assume state $\in Q$
have $\exists \text{stateMin} \in Q. \ \forall \text{state}'. (\text{state}', \text{stateMin}) \in \text{satFlagLessState} \longrightarrow \text{state}' \notin Q$
proof (cases $\exists \text{stateDef} \in Q. (\text{getSATFlag} \ \text{stateDef}) \neq \text{UNDEF}$)
case True
then obtain stateDef where stateDef $\in Q (\text{getSATFlag} \ \text{stateDef}) \neq \text{UNDEF}$
by auto

692
have \( \forall \text{state}. \, (\text{state}', \text{stateDef}) \in \text{satFlagLessState} \rightarrow \text{state}' \notin Q \)
proof
  fix \text{state}'
  show \((\text{state}', \text{stateDef}) \in \text{satFlagLessState} \rightarrow \text{state}' \notin Q \)
  proof
    assume \((\text{state}', \text{stateDef}) \in \text{satFlagLessState} \)
    hence \(\text{getSATFlag stateDef} = \text{UNDEF} \)
    unfolding \text{satFlagLessState-def} with \(\text{getSATFlag stateDef} \neq \text{UNDEF} \) have False
      by simp
    thus \(\text{state}' \notin Q \)
      by simp
    qed
  qed
  with \((\text{stateDef} \in Q) \)
  show ?thesis
    by auto
next
  case False
  have \(\forall \text{state}. \, (\text{state}', \text{state}) \in \text{satFlagLessState} \rightarrow \text{state}' \notin Q \)
  proof
    fix \text{state}'
    show \((\text{state}', \text{state}) \in \text{satFlagLessState} \rightarrow \text{state}' \notin Q \)
    proof
      assume \((\text{state}', \text{state}) \in \text{satFlagLessState} \)
      hence \(\text{getSATFlag state'} \neq \text{UNDEF} \)
        unfolding \text{satFlagLessState-def} with False
      show \(\text{state}' \notin Q \)
        by auto
    qed
  qed
  with \((\text{state} \in Q) \)
  show ?thesis
    by auto
qed

definition \(\text{lexLessState1 Vbl} = \{(\text{state1} ::: \text{State}, \text{state2} ::: \text{State})\), \(\text{getSATFlag state1} = \text{UNDEF} \land \text{getSATFlag state2} = \text{UNDEF} \land \)
\((\text{getM state1}, \text{getM state2}) \in \text{lexLessRestricted Vbl} \)
lemma wellFoundedLexLessState1:
  assumes finite Vbl
  shows wf (lexLessState1 Vbl)
  unfolding wf-eq-minimal
  proof
    show \( \forall Q \text{ state. state} \in Q \longrightarrow (\exists \text{ stateMin} \in Q. \forall \text{ state}', (\text{ state}', \text{ stateMin}) \in \text{ lexLessState1 Vbl} \longrightarrow \text{ state}' \notin Q) \)
    proof
      { 
        fix \( Q :: \text{ State set and state :: State} \)
        assume state \( \in Q \)
        let \( ?Q1 = \{ M::\text{LiteralTrail}. \exists \text{ state. state} \in Q \land \text{ getSATFlag state} = \text{ UNDEF} \land (\text{ getM state}) = M \} \)
        have \( \exists \text{ stateMin} \in Q. (\forall \text{ state}'. (\text{ state}', \text{ stateMin}) \in \text{ lexLessState1 Vbl} \longrightarrow \text{ state}' \notin Q) \)
          proof (cases \( ?Q1 \neq \{ \} \))
            case True
            then obtain M::\text{LiteralTrail}
              where M \( \in ?Q1 \)
              by auto
            then obtain MMin::\text{LiteralTrail}
              where MMin \( \in ?Q1 \land \forall \text{ M}'. (\text{ M}', \text{ MMin}) \in \text{ lexLessRestricted Vbl} \longrightarrow \text{ M'} \notin ?Q1 \)
              using wfLexLessRestricted[of Vbl] (finite Vbl)
              unfolding wf-eq-minimal
              apply simp
              apply (erule-tac \( ?Q1 \) in allE)
              by auto
            from \( MMin \in ?Q1 \) obtain stateMin
              where stateMin \( \in Q \land \text{ getM stateMin} = MMin \land \text{ getSATFlag stateMin} = \text{ UNDEF} \)
              by auto
              have \( \forall \text{ state}'. (\text{ state}', \text{ stateMin}) \in \text{ lexLessState1 Vbl} \longrightarrow \text{ state}' \notin Q \)
                proof
                  fix state'
                  show (\text{ state}', \text{ stateMin}) \( \in \text{ lexLessState1 Vbl} \longrightarrow \text{ state}' \notin Q \)
                    proof
                      assume (\text{ state}', \text{ stateMin}) \( \in \text{ lexLessState1 Vbl} \)
                      hence \text{ getSATFlag state'} = \text{ UNDEF} \land (\text{ getM state}', \text{ getM stateMin}) \( \in \text{ lexLessRestricted Vbl} \)
                      unfolding \text{ lexLessState1-def}
                      by auto
                      hence \text{ getM state'} \( \notin ?Q1 \)
                        using \( \forall \text{ M}'. (\text{ M}', \text{ MMin}) \in \text{ lexLessRestricted Vbl} \longrightarrow \text{ M'} \neq M \)
                  by auto
                by auto
              by (auto simp add: \( ?Q1 \neq \{ \} \))
            from \( MMin \in ?Q1 \) obtain stateMin
              where stateMin \( \in Q \land \text{ getM stateMin} = MMin \land \text{ getSATFlag stateMin} = \text{ UNDEF} \)
              by auto
              have \( \forall \text{ state}'. (\text{ state}', \text{ stateMin}) \in \text{ lexLessState1 Vbl} \longrightarrow \text{ state}' \notin Q \)
                proof
                  fix state'
                  show (\text{ state}', \text{ stateMin}) \( \in \text{ lexLessState1 Vbl} \longrightarrow \text{ state}' \notin Q \)
                    proof
                      assume (\text{ state}', \text{ stateMin}) \( \in \text{ lexLessState1 Vbl} \)
                      hence \text{ getSATFlag state'} = \text{ UNDEF} \land (\text{ getM state}', \text{ getM stateMin}) \( \in \text{ lexLessRestricted Vbl} \)
                      unfolding \text{ lexLessState1-def}
                      by auto
                      hence \text{ getM state'} \( \notin ?Q1 \)
                        using \( \forall \text{ M}'. (\text{ M}', \text{ MMin}) \in \text{ lexLessRestricted Vbl} \longrightarrow \text{ M'} \neq M \)
                    by auto
                  by (auto simp add: \( ?Q1 \neq \{ \} \))
                by auto
          } 
        from \( \exists \text{ stateMin} \in Q. (\forall \text{ state}'. (\text{ state}', \text{ stateMin}) \in \text{ lexLessState1 Vbl} \longrightarrow \text{ state}' \notin Q) \)
          proof
            cases \( ?Q1 \neq \{ \} \)
            case True
            then obtain M::\text{LiteralTrail}
              where M \( \in ?Q1 \)
              by auto
            then obtain MMin::\text{LiteralTrail}
              where MMin \( \in ?Q1 \land \forall \text{ M}'. (\text{ M}', \text{ MMin}) \in \text{ lexLessRestricted Vbl} \longrightarrow \text{ M'} \notin ?Q1 \)
              using wfLexLessRestricted[of Vbl] (finite Vbl)
              unfolding wf-eq-minimal
              apply simp
              apply (erule-tac \( ?Q1 \) in allE)
              by auto
            from \( MMin \in ?Q1 \) obtain stateMin
              where stateMin \( \in Q \land \text{ getM stateMin} = MMin \land \text{ getSATFlag stateMin} = \text{ UNDEF} \)
              by auto
              have \( \forall \text{ state}'. (\text{ state}', \text{ stateMin}) \in \text{ lexLessState1 Vbl} \longrightarrow \text{ state}' \notin Q \)
                proof
                  fix state'
                  show (\text{ state}', \text{ stateMin}) \( \in \text{ lexLessState1 Vbl} \longrightarrow \text{ state}' \notin Q \)
                    proof
                      assume (\text{ state}', \text{ stateMin}) \( \in \text{ lexLessState1 Vbl} \)
                      hence \text{ getSATFlag state'} = \text{ UNDEF} \land (\text{ getM state}', \text{ getM stateMin}) \( \in \text{ lexLessRestricted Vbl} \)
                      unfolding \text{ lexLessState1-def}
                      by auto
                      hence \text{ getM state'} \( \notin ?Q1 \)
                        using \( \forall \text{ M}'. (\text{ M}', \text{ MMin}) \in \text{ lexLessRestricted Vbl} \longrightarrow \text{ M'} \neq M \)
                    by auto
                  by (auto simp add: \( ?Q1 \neq \{ \} \))
                by auto
          } 
        from \( \exists \text{ stateMin} \in Q. (\forall \text{ state}'. (\text{ state}', \text{ stateMin}) \in \text{ lexLessState1 Vbl} \longrightarrow \text{ state}' \notin Q) \)
          proof
            cases \( ?Q1 \neq \{ \} \)
            case True
            then obtain M::\text{LiteralTrail}
              where M \( \in ?Q1 \)
              by auto
            then obtain MMin::\text{LiteralTrail}
              where MMin \( \in ?Q1 \land \forall \text{ M}'. (\text{ M}', \text{ MMin}) \in \text{ lexLessRestricted Vbl} \longrightarrow \text{ M'} \notin ?Q1 \)
              using wfLexLessRestricted[of Vbl] (finite Vbl)
              unfolding wf-eq-minimal
              apply simp
              apply (erule-tac \( ?Q1 \) in allE)
              by auto
            from \( MMin \in ?Q1 \) obtain stateMin
              where stateMin \( \in Q \land \text{ getM stateMin} = MMin \land \text{ getSATFlag stateMin} = \text{ UNDEF} \)
              by auto
              have \( \forall \text{ state}'. (\text{ state}', \text{ stateMin}) \in \text{ lexLessState1 Vbl} \longrightarrow \text{ state}' \notin Q \)
                proof
                  fix state'
                  show (\text{ state}', \text{ stateMin}) \( \in \text{ lexLessState1 Vbl} \longrightarrow \text{ state}' \notin Q \)
                    proof
                      assume (\text{ state}', \text{ stateMin}) \( \in \text{ lexLessState1 Vbl} \)
                      hence \text{ getSATFlag state'} = \text{ UNDEF} \land (\text{ getM state}', \text{ getM stateMin}) \( \in \text{ lexLessRestricted Vbl} \)
                      unfolding \text{ lexLessState1-def}
                      by auto
                      hence \text{ getM state'} \( \notin ?Q1 \)
                        using \( \forall \text{ M}'. (\text{ M}', \text{ MMin}) \in \text{ lexLessRestricted Vbl} \longrightarrow \text{ M'} \neq M \)
                    by auto
                  by (auto simp add: \( ?Q1 \neq \{ \} \))
                by auto
          } 
      } 
    } 
  } 
}
\[ ?Q1 \]

using \((\text{getM } \text{stateMin}) = \text{MMin})
by auto
thus \(\text{state}' \notin Q\)
using \((\text{getSATFlag } \text{state}') = \text{UNDEF})
by auto
qed

thus \(?\text{thesis}\)
using \((\text{stateMin} \in Q)\)
by auto

next
case \(\text{False}\)

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{getSATFlag } \text{state} = \text{UNDEF}
unfolding \text{lexLessState1-def}
by simp
hence \((\text{getM } \text{state}) \in ?Q1\)
using \((\text{state} \in Q)\)
by auto
hence \(\text{False}\)
using \(\text{False}\)
by auto
thus \(\text{state}' \notin Q\)
by simp
qed
qed

thus \(?\text{thesis}\)
using \((\text{state} \in Q)\)
by auto

qed

qed

definition
\text{terminationLessState1 Vbl} = \{(\text{state1}::\text{State}, \text{state2}::\text{State})\).\n(\text{state1}, \text{state2}) \in \text{satFlagLessState} \lor \n(\text{state1}, \text{state2}) \in \text{lexLessState1 Vbl}\}

lemma \text{wellFoundedTerminationLessState1:}
assumes \text{finite Vbl}
shows wf (terminationLessState1 Vbl)

unfolding wf-eq-minimal

proof –

  show ∀ Q state. state ∈ Q → (∃ stateMin ∈ Q. ∀ state’. (state’, stateMin) ∈ terminationLessState1 Vbl → state’ /∈ Q)

  proof –

  { fix Q::State set
    fix state::State
    assume state ∈ Q

    have ∃ stateMin∈Q. ∀ state’. (state’, stateMin) ∈ termination-LessState1 Vbl → state’ /∈ Q

    proof –

    obtain state0
      where state0 ∈ Q ∀ state’. (state’, state0) ∈ satFlagLessState

    using wellFoundedSatFlagLessState
    unfolding wf-eq-minimal
    using ⟨state ∈ Q⟩
    by auto

    show ?thesis

    proof (cases getSATFlag state0 = UNDEF)

    case False

    hence ∀ state’. (state’, state0) ∈ terminationLessState1 Vbl → state’ /∈ Q

    using ∀ state’. (state’, state0) ∈ satFlagLessState → state’ /∈ Q

    unfolding terminationLessState1-def
    unfolding lexLessState1-def

    by simp

    thus ?thesis

    using ⟨state0 ∈ Q⟩

    by auto

    next

    case True

    then obtain state1

    where state1 ∈ Q ∀ state’. (state’, state1) ∈ lexLessState1 Vbl → state’ /∈ Q

    using (finite Vbl)
    using ⟨state ∈ Q⟩

    using wellFoundedLexLessState1[of Vbl]

    unfolding wf-eq-minimal
    by auto

    have ∀ state’. (state’, state1) ∈ terminationLessState1 Vbl → state’ /∈ Q

    using ∀ state’. (state’, state1) ∈ lexLessState1 Vbl → state’ /∈ Q

    unfolding terminationLessState1-def
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 \text{transTerminationLessState1}:
\text{trans} (\text{terminationLessState1 Vbl})
proof –
{ fix \(x::\text{State} \text{ and } y::\text{State} \text{ and } z::\text{State} \)
  assume \((x, y) \in \text{terminationLessState1 Vbl} \text{ and } (y, z) \in \text{terminationLessState1 Vbl} \)
  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 \(\text{getSATFlag} z = \text{UNDEF} \)
    using \((y, z) \in \text{terminationLessState1 Vbl} \)
    unfolding terminationLessState1-def
    unfolding satFlagLessState-def
    unfolding lexLessState1-def
    by auto
    thus \(?thesis
    using (\text{getSATFlag} x \neq \text{UNDEF})
    unfolding terminationLessState1-def
    unfolding satFlagLessState-def
    by simp
  
  next
    case False
    with \((x, y) \in \text{terminationLessState1 Vbl} \)
    have \(\text{getSATFlag} x = \text{UNDEF} \text{ getsATFlag} y = \text{UNDEF} \text{ (getM} x, \text{getM} y) \in \text{lexLessRestricted Vbl} \)
    unfolding terminationLessState1-def
    unfolding lexLessState1-def
    by auto
    hence \(\text{getSATFlag} z = \text{UNDEF} \text{ (getM} y, \text{getM} z) \in \text{lexLessRe-}
strict Vbl
  using \((y, z) \in \text{terminationLessState1 Vbl}\);
  unfolding \text{terminationLessState1-def}
  unfolding \text{satFlagLessState-def}
  unfolding \text{lexLessState1-def}
  by auto
  thus \(?\text{thesis}\)
  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 \(?\text{thesis}\)
  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 assms
  unfolding \text{transTerminationLessState1[of Vbl]}
  by blast

lemma \text{TerminationLessAfterExhaustiveUnitPropagate}:
  assumes
  \text{exhaustiveUnitPropagate-dom state}
  InvariantUniq \((\text{getM state})\)
  InvariantConsistent \((\text{getM state})\)
  InvariantWatchListsContainOnlyClausesFromF \((\text{getWatchList state})\)
  \((\text{getF state})\) 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 \((\text{getF state})\) \((\text{getWatch1 state})\) \((\text{getWatch2 state})\)
  InvariantWatchCharacterization \((\text{getF state})\) \((\text{getWatch1 state})\) \((\text{getWatch2 state})\)
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'')
        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 False
    by (simp add: Let-def)
moreover
have InvariantConflictFlagCharacterization (getConflictFlag ?state") (getF ?state") (getM ?state")
  using ih
  using InvariantConflictFlagCharacterizationAfterApplyUnitPropagate[of state']
    unfolding 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')⟩
  using ⟨InvariantWatchesEl (getF state') (getWatch1 state') (getWatch2 state')⟩
  using ⟨getSATFlag state'' = UNDEF⟩
  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
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
\begin{verbatim}
InvariantVarsQ \((\text{getQ} \ \text{state})\) \(F0 \ \text{Vbl}\)
InvariantVarsF \((\text{getF} \ \text{state})\) \(F0 \ \text{Vbl}\)

shows

let \(\text{state}' = \text{solve-loop-body} \ \text{state} \ \text{Vbl}\) in

\((\text{InvariantConsistent} \ (\text{getM} \ \text{state}')) \wedge
\text{InvariantUniq} \ (\text{getM} \ \text{state}') \wedge
\text{InvariantWatchesEl} \ (\text{getF} \ \text{state}') \ (\text{getWatch1} \ \text{state}') \ (\text{getWatch2} \ \text{state}') \wedge
\text{InvariantWatchesDiffer} \ (\text{getF} \ \text{state}') \ (\text{getWatch1} \ \text{state}') \ (\text{getWatch2} \ \text{state}') \wedge
\text{InvariantWatchCharacterization} \ (\text{getConflictFlag} \ \text{state}') \ (\text{getQ} \ \text{state}') \ (\text{getF} \ \text{state}') \ (\text{getM} \ \text{state}') \wedge
\text{InvariantConflictFlagCharacterization} \ (\text{getConflictFlag} \ \text{state}') \ (\text{getQ} \ \text{state}') \ (\text{getF} \ \text{state}') \ (\text{getM} \ \text{state}') \wedge
\text{InvariantConflictClauseCharacterization} \ (\text{getConflictClause} \ \text{state}') \ (\text{getF} \ \text{state}') \ (\text{getM} \ \text{state}') \wedge
\text{InvariantUniqQ} \ (\text{getQ} \ \text{state}') \ F0 \ \text{Vbl} \wedge
\text{InvariantVarsM} \ (\text{getM} \ \text{state}') \ F0 \ \text{Vbl} \wedge
\text{InvariantVarsQ} \ (\text{getQ} \ \text{state}') \ F0 \ \text{Vbl} \wedge
\text{InvariantVarsF} \ (\text{getF} \ \text{state}') \ F0 \ \text{Vbl} \wedge
\) \((\text{state}', \ \text{state}) \in \text{terminationLessState1} \ (\text{vars} \ F0 \cup \text{Vbl})\) \wedge
\((\text{getSATFlag} \ \text{state}' = \text{FALSE} \rightarrow \neg \text{satisfiable} \ F0') \wedge
\) \((\text{getSATFlag} \ \text{state}' = \text{TRUE} \rightarrow \text{satisfiable} \ F0')\) \wedge
\((\text{is} \ \text{let} \ \text{state}' = \text{solve-loop-body} \ \text{state} \ \text{Vbl} \ \text{in} \ \?\text{inv'} \ \text{state}' \wedge \ ?\text{inv''} \) \text{state}' \wedge - )

proof –

let \(?\text{state-up} = \text{exhaustiveUnitPropagate} \ \text{state}\)

have \text{exhaustiveUnitPropagate-dom} \ \text{state}
using \text{exhaustiveUnitPropagateTermination[of state F0 \ Vbl]}
using \text{assms}
by \text{simp}

have \(?\text{inv'} \ ?\text{state-up}
using \text{assms}

703
\end{verbatim}
using (exhaustiveUnitPropagate-dom state)
using InvariantsAfterExhaustiveUnitPropagate[of state]
using InvariantConflictClauseCharacterizationAfterExhaustivePropagate[of state]
   by (simp add: Let-def)
have ?inv'' ?state-up
   using assms
   using (exhaustiveUnitPropagate-dom state)
   using InvariantsNoDecisionsWhenConflictNorUnitAfterExhaustivePropagate[of state]
   by (simp add: Let-def)
have InvariantEquivalentZL (getF ?state-up) (getM ?state-up) F0'
   using assms
   using (exhaustiveUnitPropagate-dom state)
   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 (exhaustiveUnitPropagate-dom state)
   using InvariantGetReasonIsReasonAfterExhaustiveUnitPropagate[of state]
   by (simp add: Let-def)
have getSATFlag ?state-up = getSATFlag state
   using exhaustiveUnitPropagatePreservedVariables[of state]
   using assms
   using (exhaustiveUnitPropagate-dom state)
   by (simp add: Let-def)
have getConflictFlag ?state-up ∨ getQ ?state-up = []
   using conflictFlagOrQEmptyAfterExhaustiveUnitPropagate[of state]
   using (exhaustiveUnitPropagate-dom state)
   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 (exhaustiveUnitPropagate-dom state)
   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 TerminationLessAfterExhaustiveUnitPropagate[of state]
   using (exhaustiveUnitPropagate-dom state)
   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 ¬ satisfies F0'
      using InvariantEquivalentZL (getF ?state-up) (getM ?state-up) F0'
      unfolding InvariantEquivalentZL-def
      using soundnessForUNSAT[of getF ?state-up elements (getM ?state-up) F0']
      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)
(state-c)
InvariantCEntailed (getConflictFlag ?state-c) F0' (getC
?state-c)
InvariantCICharacterization (getCl ?state-c) (getC ?state-c)
(state-c)
InvariantCnCharacterization (getCn ?state-c) (getC ?state-c)
(state-c)
getM (getM ?state-c)
InvariantClCurrentLevel (getCl ?state-c) (getM ?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))
  using (getM ?state-c = getM ?state-up)
  using (getSATFlag ?state-c = getSATFlag state)
  using (InvariantUniq (getM state))
  using (InvariantConsistent (getM state))
  using (getM ?state-c = getM ?state-up) F0 Vbl
  using (getSATFlag ?state-up = getSATFlag state)
  using (getSATFlag state = UNDEF)
  unfolding InvariantConsistent-def
  unfolding InvariantUniq-def
  unfolding InvariantVarsM-def
  unfolding terminationLessState1-def
  unfolding satFlagLessState-def
  unfolding lexLessState1-def
  unfolding lexLessRestricted-def
  by auto

let ?state-euip = applyExplainUIP ?state-c
let ?l′ = getCl ?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) F0)
  using (currentLevel (getM ?state-c) > 0)
  using (?inv′ ?state-c)
    using InvariantCFalse (getConflictFlag ?state-c) (getM ?state-c) (getC ?state-c)
using $\langle$ InvariantCEntailed $(getConflictFlag ?state-c) F0'$ $(getC ?state-c) \rangle$

using $\langle$ InvariantCCharacterization $(getCl ?state-c) $(getC ?state-c) $(getM ?state-c) \rangle$
using $\langle$ InvariantCnCharacterization $(getCn ?state-c) $(getC ?state-c) $(getM ?state-c) \rangle$
using $\langle$ InvariantClCurrentLevel $(getCl ?state-c) $(getM ?state-c) \rangle$
using $\langle$ InvariantUniqC $(getC ?state-c) \rangle$

using $\langle$ InvariantCFalse $(getConflictFlag ?state-euip) $(getM ?state-euip) $(getC ?state-euip) \rangle$
using $\langle$ InvariantCEntailed $(getConflictFlag ?state-euip) F0'$ $(getC ?state-euip) \rangle$
using $\langle$ InvariantCCharacterization $(getCl ?state-euip) $(getC ?state-euip) $(getM ?state-euip) \rangle$
using $\langle$ InvariantCnCharacterization $(getCn ?state-euip) $(getC ?state-euip) $(getM ?state-euip) \rangle$
using $\langle$ InvariantClCurrentLevel $(getCl ?state-euip) $(getM ?state-euip) \rangle$
using $\langle$ InvariantUniqC $(getC ?state-euip) \rangle$

have $\langle$ simp $\rangle$

using $\langle$ InvariantCEntailed $(getConflictFlag ?state-euip) F0'$ $(getC ?state-euip) \rangle$
using $\langle$ InvariantCCharacterization $(getCl ?state-euip) $(getC ?state-euip) $(getM ?state-euip) \rangle$
using $\langle$ InvariantCnCharacterization $(getCn ?state-euip) $(getC ?state-euip) $(getM ?state-euip) \rangle$
using $\langle$ InvariantClCurrentLevel $(getCl ?state-euip) $(getM ?state-euip) \rangle$
using $\langle$ InvariantUniqC $(getC ?state-euip) \rangle$

have $\langle$ simp $\rangle$

have $\langle$ InvariantCFalse $(getConflictFlag ?state-c) $(getM ?state-c) $(getC ?state-c) \rangle$
InvariantCEntailed $(getConflictFlag ?state-c) F0'$ $(getC ?state-c) \rangle$

using $\langle$ InvariantCCharacterization $(getCl ?state-c) $(getC ?state-c) $(getM ?state-c) \rangle$
using $\langle$ InvariantCnCharacterization $(getCn ?state-c) $(getC ?state-c) $(getM ?state-c) \rangle$
using $\langle$ InvariantClCurrentLevel $(getCl ?state-c) $(getM ?state-c) \rangle$
using $\langle$ InvariantEquivalentZL $(getF ?state-c) $(getM ?state-c) \rangle$

by $\langle$ auto simp only: Let-def $\rangle$

using $\langle$ InvariantCCharacterization $(getCl ?state-c) $(getC ?state-c) $(getM ?state-c) \rangle$
using $\langle$ InvariantCnCharacterization $(getCn ?state-c) $(getC ?state-c) $(getM ?state-c) \rangle$
using $\langle$ InvariantClCurrentLevel $(getCl ?state-c) $(getM ?state-c) \rangle$

F0' $\rangle$

using $\langle$ InvariantUniqC $(getC ?state-c) \rangle$
using $\langle$ getConflictFlag ?state-c $\rangle$
using $\langle$ currentLevel $(getM ?state-c) > 0 $\rangle$
using $\langle$ InvariantGetReasonIsReason $(getReason ?state-c) $(getF ?state-c) $(getM ?state-c) $(set $(getQ ?state-c)) \rangle$
using $\langle$ applyExplainUIP-dom ?state-c $\rangle$
using $\langle$ InvariantsClAfterExplainUIP[of ?state-c F0'] $\rangle$
by $\langle$ auto simp only: Let-def $\rangle$

708
have \( \text{InvariantEquivalentZL} \ (\text{getF} \ ?\text{state-euip}) \ (\text{getM} \ ?\text{state-euip}) \) \\
\( F_0' \)  \\
using \( \langle \text{InvariantEquivalentZL} \ (\text{getF} \ ?\text{state-c}) \ (\text{getM} \ ?\text{state-c}) \rangle \) \\
\( F_0' \)  \\
using \( \langle \text{applyExplainUIP-dom} \ ?\text{state-c} \rangle \) \\
using \( \text{ApplyExplainUIPPreservedVariables}[\text{of} \ ?\text{state-c}] \) \\
by \( \langle \text{simp only: Let-def} \rangle \) \\

have \( \text{InvariantGetReasonIsReason} \ (\text{getReason} \ ?\text{state-euip}) \ (\text{getF} \ ?\text{state-euip}) \ (\text{getM} \ ?\text{state-euip}) \ (\text{set} \ (\text{getQ} \ ?\text{state-euip})) \) \\
using \( \langle \text{InvariantGetReasonIsReason} \ (\text{getReason} \ ?\text{state-c}) \ (\text{getF} \ ?\text{state-c}) \ (\text{set} \ (\text{getQ} \ ?\text{state-c})) \rangle \) \\
using \( \langle \text{applyExplainUIP-dom} \ ?\text{state-c} \rangle \) \\
using \( \text{ApplyExplainUIPPreservedVariables}[\text{of} \ ?\text{state-c}] \) \\
by \( \langle \text{simp only: Let-def} \rangle \) \\

have \( \text{getConflictFlag} \ ?\text{state-euip} \) \\
using \( \langle \text{getConflictFlag} \ ?\text{state-c} \rangle \) \\
using \( \langle \text{applyExplainUIP-dom} \ ?\text{state-c} \rangle \) \\
using \( \text{ApplyExplainUIPPreservedVariables}[\text{of} \ ?\text{state-c}] \) \\
by \( \langle \text{simp add: Let-def} \rangle \) \\

hence \( \text{getSATFlag} \ ?\text{state-euip} = \text{getSATFlag state} \) \\
using \( \langle \text{getSATFlag} \ ?\text{state-c} = \text{getSATFlag state} \rangle \) \\
using \( \langle \text{applyExplainUIP-dom} \ ?\text{state-c} \rangle \) \\
using \( \text{ApplyExplainUIPPreservedVariables}[\text{of} \ ?\text{state-c}] \) \\
by \( \langle \text{simp add: Let-def} \rangle \) \\

have \( \text{isUIP} \ (\text{opposite} \ (\text{getCl} \ ?\text{state-euip})) \ (\text{getC} \ ?\text{state-euip}) \ (\text{getM} \ ?\text{state-euip}) \) \\
(\text{getM} \ ?\text{state-euip}) \\
using \( \langle \text{applyExplainUIP-dom} \ ?\text{state-c} \rangle \) \\
using \( \langle \text{simp add: Let-def} \rangle \) \\

have \( \text{isUIP} \ (\text{opposite} \ (\text{getCl} \ ?\text{state-euip})) \ (\text{getC} \ ?\text{state-euip}) \) \\
(\text{getM} \ ?\text{state-euip}) \\
using \( \langle \text{applyExplainUIP-dom} \ ?\text{state-c} \rangle \) \\
using \( \langle \text{simp add: Let-def} \rangle \) \\

709
have \( \text{currentLevel} \ (\text{getM} \ ?\text{state-euip}) > 0 \)
using \( \text{applyExplainUIP-dom} \ ?\text{state-c} \)
using \( \text{ApplyExplainUIPPreservedVariables[of \ ?\text{state-c}] \) using \( \text{currentLevel} \ (\text{getM} \ ?\text{state-c}) > 0 \)
by \( \text{(simp add: Let-def)} \)

have \( \text{InvariantVarsM} \ (\text{getM} \ ?\text{state-euip}) \ F_0 \ Vbl \)
\( \text{InvariantVarsQ} \ (\text{getQ} \ ?\text{state-euip}) \ F_0 \ Vbl \)
\( \text{InvariantVarsF} \ (\text{getF} \ ?\text{state-euip}) \ F_0 \ Vbl \)
using \( \text{InvariantVarsM} \ (\text{getM} \ ?\text{state-c}) \ F_0 \ Vbl \)
using \( \text{InvariantVarsQ} \ (\text{getQ} \ ?\text{state-c}) \ F_0 \ Vbl \)
using \( \text{InvariantVarsF} \ (\text{getF} \ ?\text{state-c}) \ F_0 \ Vbl \)
using \( \text{applyExplainUIP-dom} \ ?\text{state-c} \)
using \( \text{ApplyExplainUIPPreservedVariables[of \ ?\text{state-c}] \) by \( \text{(auto simp add: Let-def)} \)

have \( \text{getM} \ ?\text{state-euip} = \text{getM} \ ?\text{state} \lor (\text{state-euip}, \text{state}) \in \text{terminationLessState1} \ (\text{vars} F_0 \cup Vbl) \)
using \( \text{getM} \ ?\text{state-c} = \text{getM} \ ?\text{state} \lor (\text{state-euip}, \text{state}) \in \text{terminationLessState1} \ (\text{vars} F_0 \cup Vbl) \)
using \( \text{applyExplainUIP-dom} \ ?\text{state-c} \)
using \( \text{ApplyExplainUIPPreservedVariables[of \ ?\text{state-c}] \) unfolding \( \text{terminationLessState1-def} \)
unfolding \( \text{satFlagLessState-def} \)
unfolding \( \text{lexLessState1-def} \)
unfolding \( \text{lexLessRestricted-def} \)
by \( \text{(simp add: Let-def)} \)

let \( ?\text{state-l} = \text{applyLearn} \ ?\text{state-euip} \)
let \( ?l'' = \text{getCl} \ ?\text{state-l} \)

have \( ?\text{state-l} = \text{getM} \ ?\text{state-l} \land \text{getQ} \ ?\text{state-l} \land \text{getC} \ ?\text{state-l} \land \text{getCl} \ ?\text{state-l} \land \text{getConflictFlag} \ ?\text{state-l} = \text{getConflictFlag} \ ?\text{state-euip} \land \text{getConflictClause} \ ?\text{state-l} = \text{getConflictClause} \ ?\text{state-euip} \land \text{getF} \ ?\text{state-l} = \text{(if} \ \text{getC} \ ?\text{state-euip} = \text{[opposite} \ ?l'] \ \text{then} \ \text{getF} \ ?\text{state-euip} \ \text{else} \ \text{(getF} \ ?\text{state-euip} \ ?\text{getC} \ ?\text{state-euip}) \text{)} \)
using \( \text{applyLearnPreservedVariables[of \ ?\text{state-euip}] \) by \( \text{(simp add: Let-def)} \)

have \( ?\text{inv}'' \ ?\text{state-l} \)
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 ⟨InvariantCFalse (getConflictFlag ?state-euip) (getM ?state-euip)⟩
    (getC ?state-euip)
    (getM ?state-euip)
    using ⟨isUIP (opposite (getCl ?state-euip)) (getC ?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))
  have InvariantNoDecisionsWhenUnit (getF ?state-euip) (getM ?state-l) (currentLevel (getM ?state-l))
  have InvariantNoDecisionsWhenConflict [getC ?state-euip] (getM ?state-euip)
\(?l\) (getBackjumpLevel \(?l\))

\textit{InvariantNoDecisionsWhenUnit} \(\text{get}C \ ?\text{equip}\) \(\text{get}M \ ?l\) (getBackjumpLevel \(?l\))

\textbf{using} \textit{InvariantNoDecisionsWhenConflictNorUnitAfterApplyLearn}\[\text{of} \ ?\text{equip}\]

\textbf{using} \(\langle \text{inv} \ ?\text{equip}\rangle\)

\textbf{using} \(\langle \text{inv'} \ ?\text{equip}\rangle\)

\textbf{using} \(\langle \text{getConflictFlag} \ ?\text{equip}\rangle\)

\textbf{using} \(\langle \text{InvariantUniqC} \ (\text{get}C \ ?\text{equip})\rangle\)

\textbf{using} \(\langle \text{InvariantCFalse} \ (\text{getConflictFlag} \ ?\text{equip}) \ (\text{get}M \ ?\text{equip}) \ (\text{get}C \ ?\text{equip})\rangle\)

\textbf{using} \(\langle \text{InvariantClCharacterization} \ (\text{get}C \ ?\text{equip}) \ (\text{get}C \ ?\text{equip}) \ (\text{get}M \ ?\text{equip}) \ (\text{get}M \ ?\text{equip})\rangle\)

\textbf{using} \(\langle \text{currentLevel} \ (\text{get}M \ ?\text{equip}) > 0\rangle\)

\textbf{by} (auto simp only: Let-def)

\textbf{have} \(\text{isUIP} \ (\text{opposite} \ (\text{get}C \ ?\text{equip})) \ (\text{get}C \ ?\text{equip}) \ (\text{get}M \ ?\text{equip})\)

\textbf{using} \(\langle \text{isUIP} \ (\text{opposite} \ (\text{get}C \ ?\text{equip})) \ (\text{get}C \ ?\text{equip}) \ (\text{get}M \ ?\text{equip})\rangle\)

\textbf{by} simp

\textbf{have} \(\text{InvariantClCurrentLevel} \ (\text{get}C \ ?\text{equip}) \ (\text{get}C \ ?\text{equip}) \ (\text{get}M \ ?\text{equip})\)

\textbf{using} \(\langle \text{InvariantClCurrentLevel} \ (\text{get}C \ ?\text{equip}) \ (\text{get}M \ ?\text{equip})\rangle\)

\textbf{by} simp

\textbf{have} \(\text{InvariantCEntailed} \ (\text{getConflictFlag} \ ?\text{equip}) F0' \ (\text{get}C \ ?\text{equip})\)

\textbf{using} \(\langle \text{InvariantCEntailed} \ (\text{getConflictFlag} \ ?\text{equip}) F0' \ (\text{get}C \ ?\text{equip})\rangle\)

\textbf{by} simp

\textbf{have} \(\text{InvariantCFalse} \ (\text{getConflictFlag} \ ?\text{equip}) \ (\text{get}M \ ?\text{equip}) \ (\text{get}C \ ?\text{equip})\)

\textbf{using} \(\langle \text{InvariantCFalse} \ (\text{getConflictFlag} \ ?\text{equip}) \ (\text{get}M \ ?\text{equip}) \ (\text{get}C \ ?\text{equip})\rangle\)

\textbf{by} simp

712
have \texttt{InvariantUniqC}\ (getC \ ?state-l)
using \texttt{InvariantUniqC}\ (getC \ ?state-euip)
using $\$
by simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp

have \texttt{InvariantCLCharacterization}\ (getCl \ ?state-l) (getC \ ?state-l)
\texttt{(getM \ ?state-l)}
using \texttt{InvariantCLCharacterization}\ (getCl \ ?state-euip) (getC \ ?state-euip)
\texttt{using $\$}\nby simp
using $\langle \text{InvariantCFalse} \rangle$
\[ \text{getConflictFlag \ ?state-euip} \ (\text{getM \ ?state-euip}) \ (\text{getC \ ?state-euip}) \]
by \text{auto}

\text{have getConflictFlag \ ?state-l}
using \(\langle \text{InvariantVarsFAfterApplyLearn[ of \ ?state-euip F0 Vbl} \rangle\)
by \text{auto}

\text{have getConflictFlag \ ?state-l}
using \(\langle \text{getConflictFlag \ ?state-euip} \rangle\)
using $\langle \text{getConflictFlag \ ?state-euip} \rangle$
by \text{simp}

\text{have getSATFlag \ ?state-l = getSATFlag state}
using \(\langle \text{getSATFlag \ ?state-euip = getSATFlag state} \rangle\)
unfolding \text{applyLearn-def}
unfolding \text{setWatch2-def}
unfolding \text{setWatch1-def}
by \(\langle \text{simp add: Let-def} \rangle\)

\text{have currentLevel \ (getM \ ?state-l) > 0}
using \(\langle \text{currentLevel \ (getM \ ?state-euip) > 0} \rangle\)
using $\langle \text{currentLevel \ (getM \ ?state-euip) > 0} \rangle$
by \text{simp}

\text{have getM \ ?state-l = getM state} \lor \ (\ ?state-l, \ state) \in \text{terminationLessState1 (vars F0 \cup Vbl)}
\text{proof (cases getM \ ?state-euip = getM state)}
\text{case True}
\text{thus ?thesis}
using $\langle \text{currentLevel \ (getM \ ?state-euip) > 0} \rangle$
by \text{simp}
\text{next}
\text{case False}
with \(\langle \text{getM \ ?state-euip = getM state} \lor \ (\ ?state-euip, \ state) \in \text{terminationLessState1 (vars F0 \cup Vbl)} \rangle\)
\text{have \ (\ ?state-euip, \ state) \in \text{terminationLessState1 (vars F0 \cup Vbl)}}
by \text{simp}
hence \(\ ?state-l, \ state) \in \text{terminationLessState1 (vars F0 \cup Vbl)}$
using $\langle \text{currentLevel \ (getM \ ?state-euip) > 0} \rangle$
using \(\langle \text{getSATFlag \ ?state-l = getSATFlag state} \rangle\)
using \(\langle \text{getSATFlag \ ?state-euip = getSATFlag state} \rangle\)
unfolding \text{terminationLessState1-def}
unfolding \text{satFlagLessState-def}
unfolding \text{lexLessState1-def}
unfolding \text{lexLessRestricted-def}
by \(\langle \text{simp add: Let-def} \rangle\)
thus thesis
  by simp
qed

let ?state-bj = applyBackjump ?state-l

have ?inv' ?state-bj ∧
  InvariantVarsM (getM ?state-bj) F0 Vbl ∧
  InvariantVarsQ (getQ ?state-bj) F0 Vbl ∧
  InvariantVarsF (getF ?state-bj) F0 Vbl
proof (cases getC ?state-l = [opposite ?l'])
case True
  thus thesis
  using WatchInvariantsAfterApplyBackjump[of ?state-l F0']
  using InvariantUniqAfterApplyBackjump[of ?state-l F0']
  using InvariantConsistentAfterApplyBackjump[of ?state-l F0']
  using invariantQCharacterizationAfterApplyBackjump-1[of ?state-l F0']
  using InvariantConflictFlagCharacterizationAfterApplyBackjump-1[of ?state-l F0']
  using InvariantUniqQAfterApplyBackjump[of ?state-l]
  using InvariantConflictClauseCharacterizationAfterApplyBackjump[of ?state-l]
  using InvariantVarsAfterApplyBackjump[of ?state-l F0' F0 Vbl]
  using (?inv' ?state-b)
  using (getConflictFlag ?state-l)
    using (InvariantClCurrentLevel (getC ?state-l) (getM ?state-l))
    using (InvariantClC (getC ?state-l))
    using (InvariantCFalse (getConflictFlag ?state-l) (getM ?state-l) (getC ?state-l))
    using (InvariantCEntailed (getConflictFlag ?state-l) F0' (getC ?state-l))
    using (InvariantClCharacterization (getC ?state-l) (getM ?state-l))
    using (InvariantClCharacterization (getC ?state-l) (getCl ?state-l) (getM ?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 (InvariantEquivalentZL (getF ?state-l) (getM ?state-l) F0"

715
next

\begin{align*}
\text{using } & (\text{ InvariantVarsM (getM ?state-l) F0 Vbl}) \\
\text{using } & (\text{ InvariantVarsQ (getQ ?state-l) F0 Vbl}) \\
\text{using } & (\text{ InvariantVarsF (getF ?state-l) F0 Vbl}) \\
\text{using } & (\text{ vars F0' } \subseteq \text{ vars F0}) \\
\text{using } & $ \\
& \text{ by (simp add: Let-def) }
\end{align*}

\begin{align*}
\text{next} \\
\text{case False} \\
\text{thus } ?thesis \\
\text{using } & (\text{ WatchInvariantsAfterApplyBackjump[of ?state-l F0']} \\
\text{using } & (\text{ InvariantUniqAfterApplyBackjump[of ?state-l F0']} \\
& \text{ using } (\text{ InvariantConsistentAfterApplyBackjump[of ?state-l F0']}) \\
\text{using } & (\text{ invariantQCharacterizationAfterApplyBackjump-2[of ?state-l F0']} \\
\text{using } & (\text{ InvariantConflictFlagCharacterizationAfterApplyBackjump-2[of ?state-l F0']} \\
\text{using } & (\text{ InvariantUniqQAfterApplyBackjump[of ?state-l F0']} \\
\text{using } & (\text{ InvariantConflictClauseCharacterizationAfterApplyBackjump[of ?state-l]}) \\
& \text{ using } (\text{ InvariantsVarsAfterApplyBackjump[of ?state-l F0']} \\
\text{using } & (\text{ getConflictFlag ?state-l}) \\
& \text{ using } (\text{ InvariantCICurrentLevel (getCl ?state-l) (getM ?state-l)}) \\
& \text{ using } (\text{ InvariantUniqC (getC ?state-l)}) \\
& \text{ using } (\text{ InvariantCFalse (getConflictFlag ?state-l) (getM ?state-l)}) \\
& \text{ using } (\text{ InvariantCEntailed (getConflictFlag ?state-l) F0' (getC ?state-l)}) \\
& \text{ using } (\text{ InvariantCICharacterization (getCl ?state-l) (getC ?state-l)}) \\
& \text{ using } (\text{ InvariantCIICharacterization (getC ?state-l) (getCll ?state-l) (getM ?state-l)}) \\
& \text{ using } (\text{ isUIP (opposite (getC ?state-l)) (getC ?state-l) (getM ?state-l)}) \\
& \text{ using } (\text{ currentLevel (getM ?state-l) > 0}) \\
& \text{ using } (\text{ InvariantNoDecisionsWhenConflict (getF ?state-euip) (getM ?state-l) (currentLevel (getM ?state-l))}) \\
& \text{ using } (\text{ InvariantNoDecisionsWhenUnit (getF ?state-euip) (getM ?state-l) (currentLevel (getM ?state-l))}) \\
& \text{ using } (\text{ InvariantNoDecisionsWhenConflict [getC ?state-euip] (getM ?state-l) (getBackjumpLevel ?state-l)}) \\
& \text{ using } (\text{ InvariantNoDecisionsWhenUnit [getC ?state-euip] (getM ?state-l) (getBackjumpLevel ?state-l)}) \\
& \text{ using } $ \\
& \text{ using } (\text{ InvariantEquivalentZL (getF ?state-l) (getM ?state-l)})
\end{align*}
using \langle\text{InvariantVarsM} (\text{getM} ?\text{state-l}) F0 \text{ Vbl}\rangle
using \langle\text{InvariantVarsQ} (\text{getQ} ?\text{state-l}) F0 \text{ Vbl}\rangle
using \langle\text{InvariantVarsF} (\text{getF} ?\text{state-l}) F0 \text{ Vbl}\rangle
using \langle\text{vars} F0' \subseteq \text{vars} F0\rangle
by (\text{simp add: Let-def})

qed

have \text{inv''} ?\text{state-bj}
proof (cases \text{getC} ?\text{state-l} = \text{[opposite} ?l'\text{]})
  case True
  thus \text{thesis}
    using \text{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' (\text{getC} ?\text{state-l}))
    using (\text{InvariantClCharacterization} (\text{getCl} ?\text{state-l}) (\text{getM} ?\text{state-l}))
    using (\text{InvariantCllCharacterization} (\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 $\
      \text{by (simp add: Let-def)\
    next}
  case False
  thus \text{thesis}
    using \text{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 $\langle \text{InvariantClCharacterization} \ (\text{getCl} \ ?\text{state-l}) \ (\text{getC} \ ?\text{state-l}) \ (\text{getM} \ ?\text{state-l}) \rangle$

using $\langle \text{InvariantClCharacterization} \ (\text{getCl} \ ?\text{state-l}) \ (\text{getCll} \ ?\text{state-l}) \ (\text{getM} \ ?\text{state-l}) \rangle$

using $\langle \text{isUIP} \ (\text{opposite} \ (\text{getCl} \ ?\text{state-l})) \ (\text{getC} \ ?\text{state-l}) \ (\text{getM} \ ?\text{state-l}) \rangle$

using $\langle \text{currentLevel} \ (\text{getM} \ ?\text{state-l}) \rangle$

using $\langle \text{InvariantNoDecisionsWhenConflict} \ (\text{getF} \ ?\text{state-euip}) \ (\text{getM} \ ?\text{state-l}) \ \text{currentLevel} \ (\text{getM} \ ?\text{state-l}) \rangle$

using $\langle \text{InvariantNoDecisionsWhenUnit} \ (\text{getF} \ ?\text{state-euip}) \ (\text{getM} \ ?\text{state-l}) \ \text{currentLevel} \ (\text{getM} \ ?\text{state-l}) \rangle$

using $\langle \text{InvariantNoDecisionsWhenConflict} \ (\text{getC} \ ?\text{state-euip}) \ (\text{getBackjumpLevel} \ ?\text{state-l}) \rangle$

using $\langle \text{InvariantNoDecisionsWhenUnit} \ (\text{getC} \ ?\text{state-euip}) \ (\text{getM} \ ?\text{state-l}) \ \text{getBackjumpLevel} \ ?\text{state-l}) \rangle$

using $\langle \text{InvariantUniqC} \ (\text{getC} \ ?\text{state-l}) \rangle$

using $\langle \text{InvariantCFalse} \ (\text{getConflictFlag} \ ?\text{state-l}) \ (\text{getM} \ ?\text{state-l}) \ (\text{getC} \ ?\text{state-l}) \rangle$

have getBackjumpLevel ?state-l > 0 $\rightarrow$ (getF ?state-l) ≠ [] ∧ (last (getF ?state-l) = (getC ?state-l))

proof (cases getC ?state-l = [opposite ?l])

<table>
<thead>
<tr>
<th>Case</th>
<th>Proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>thus $\Rightarrow$ thesis unfolding getBackjumpLevel-def by simp</td>
</tr>
<tr>
<td>False</td>
<td>thus $\Rightarrow$ thesis using $\langle \text{simp add: Let-def} \rangle$</td>
</tr>
</tbody>
</table>

hence $\langle \text{InvariantGetReasonIsReason} \ (\text{getReason} \ ?\text{state-bj}) \ (\text{getF} \ ?\text{state-bj}) \ (\text{getM} \ ?\text{state-bj}) \ (\text{set} \ (\text{getQ} \ ?\text{state-bj})) \rangle$

using $\langle \text{InvariantGetReasonIsReason} \ (\text{getReason} \ ?\text{state-l}) \ (\text{getF} \ ?\text{state-l}) \ (\text{getM} \ ?\text{state-l}) \ (\text{set} \ (\text{getQ} \ ?\text{state-l})) \rangle$

using $\langle \text{isUIP} \ (\text{opposite} \ (\text{getCl} \ ?\text{state-l})) \ (\text{getM} \ ?\text{state-l}) \ \text{getBackjumpLevel} \ ?\text{state-l}) \rangle$

using $\langle \text{InvariantCFalse} \ (\text{getConflictFlag} \ ?\text{state-l}) \ (\text{getM} \ ?\text{state-l}) \ (\text{getC} \ ?\text{state-l}) \rangle$

using $\langle \text{InvariantUniqC} \ (\text{getC} \ ?\text{state-l}) \rangle$

using $\langle \text{InvariantClCharacterization} \ (\text{getCl} \ ?\text{state-l}) \ (\text{getC} \ ?\text{state-l}) \ (\text{getM} \ ?\text{state-l}) \rangle$

718
?state-l (getM ?state-l)
  using InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l),
  using ⟨currentLevel (getM ?state-l) > 0⟩
  using InvariantGetReasonIsReasonAfterApplyBackjump[of ?state-l F0’]
  by (simp only: Let-def)

  have InvariantEquivalenceZL (getF ?state-l) (getM ?state-l)
  F0’
  using InvariantEquivalenceZL (getF ?state-l) (getM ?state-l)

  using ⟨?inv’ ?state-l⟩
  using (getConflictFlag ?state-l)
  using (isUIP (opposite (getCl ?state-l)) (getC ?state-l) (getM ?state-l))

  using InvariantCFalse (getConflictFlag ?state-l) (getM ?state-l)
  (getC ?state-l)


  using ⟨currentLevel (getM ?state-l) > 0⟩
  by (simp only: Let-def)

  have getSATFlag ?state-bj = getSATFlag state
  using ⟨getSATFlag ?state-l = getSATFlag state⟩
  using ⟨?inv’ ?state-l⟩
  using applyBackjumpPreservedVariables[of ?state-l]
  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 ⟨?inv’ ?state-l⟩
  using InvariantCEntailed (getConflictFlag ?state-l) F0’ (getC ?state-l)
using ⟨InvariantUniqC (getC ?state-l))⟩
using ⟨InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l))⟩
using ⟨isUfP (opposite (getCl ?state-l)) (getC ?state-l) (getM ?state-l))⟩
using ⟨currentLevel (getM ?state-l) > 0⟩

by (simp add: Let-def)

hence getBackjumpLevel ?state-l < elementLevel (getCl ?state-l) (getM ?state-l))

unfolding isMinimalBackjumpLevel-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 ⟨isUfP (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 ∧
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 transTerminationLessState1[if ?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 ∪ Vbl)

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'

721
using soundnessForSat \{ F_0' \} Vbl getF ?state-up getM ?state-up\)
using \{ invariantEquivalentZL (getF ?state-up) (getM ?state-up) \}

F_0'

using \{ ?inv' ?state-up \}
using \{ invariantVarsF (getF ?state-up) F_0 Vbl \}
using (\neg getConflictFlag ?state-up)
using \{ vars F_0 \subseteq Vbl \}
using \{ vars F_0' \subseteq vars F_0 \}
using True
unfolding invariantConflictFlagCharacterization-def
unfolding satisfiable-def
unfolding invariantVarsF-def
by blast
moreover
let ?state' = ?state-up \{ getSATFlag := TRUE \}
have (\langle ?state', state \rangle) \in terminationLessState1 (vars F_0 \cup Vbl)
using (getSATFlag state = UNDEF)
unfolding terminationLessState1-def
unfolding satFlagLessState-def
by simp
ultimately
show ?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) \}

F_0'

using \{ invariantVarsM (getM ?state-up) F_0 Vbl \}
using \{ invariantVarsQ (getQ ?state-up) F_0 Vbl \}
using (\neg getConflictFlag ?state-up)
unfolding solve-loop-body-def
by (simp add: Let-def)

next


next case False
let ?literal = selectLiteral ?state-up Vbl
let ?state-d = applyDecide ?state-up Vbl

have \{ invariantConsistent (getM ?state-d) \}
using \{ invariantConsistentAfterApplyDecide \}
using \{ False \}
using \{ ?inv' ?state-up \}
by (simp add: Let-def)
moreover
have \{ invariantUniq (getM ?state-d) \}
using \{ invariantUniqAfterApplyDecide \}
using \{ False \}
using \(?\text{inv}'\) \(?\text{state-up}\)
by (simp add: \text{Let-def})
moreover
have \text{InvariantQCharacterization AfterApplyDecide} \[\text{of Vbl} \?
\text{state-up}\]
using False
using \(?\text{inv}'\) \(?\text{state-up}\)
using \(\neg\) \text{getConflictFlag} \?(\text{state-up})
using \text{exhaustiveUnitPropagate-dom state}
using \text{conflictFlagOrQEmptyAfterExhaustiveUnitPropagate}[\text{of state}]
by (simp add: \text{Let-def})
moreover
have \text{InvariantConflictFlagCharacterization} \[\text{of ?state-up \?
literal True}\]
using \(?\text{inv}'\) \?(\text{state-up})
using \text{assertLiteralEffect}
unfolding \text{applyDecide-def}
by (simp only: \text{Let-def})
moreover
have \text{InvariantConflictClauseCharacterization} \[\text{of ?state-up \?
literal True}\]
using \(?\text{inv}'\) \?(\text{state-up})
using \text{assertLiteralEffect}
unfolding \text{applyDecide-def}
by (simp only: \text{Let-def})
moreover
have \text{InvariantNoDecisionsWhenConflict} \[\text{of state}\]
underline{\text{InvariantNoDecisionsWhenUnit}} \[\text{of state}\]
\underline{\text{InvariantNoDecisionsWhenConflictNorUnit}} \[\text{of state}\]
\underline{\text{InvariantsNoDecisionsWhenConflictNorUnit}} \[\text{of state}\]
unfolding \text{applyDecide-def}
by (auto simp add: \text{Let-def})

723
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 (InvariantVarsM (getM ?state-up) F0 Vbl)
using (InvariantVarsM (getM ?state-d) F0 Vbl)
using (getSATFlag state = UNDEF)
using (?inv' ?state-up)
using (InvariantConsistent (getM ?state-d))
using (InvariantUniq (getM ?state-d))
using lexLessAppend[of [(selectLiteral ?state-up Vbl, True)] getM

?state-up]
unfolding applyDecide-def
unfolding terminationLessState1-def
unfolding lexLessState1-def
unfolding lexLessRestricted-def
unfolding InvariantVarsM-def
unfolding InvariantUniq-def
unfolding InvariantConsistent-def
by (simp add: Let-def)

hence (?state-d, state) ∈ terminationLessState1 (vars F0 ∪ Vbl)
using (?state-up = state ∨ (?state-up, state) ∈ termination-

LessState1 (vars F0 ∪ Vbl))
using transTerminationLessState1I[of ?state-d ?state-up vars
F0 ∪ Vbl state]
by auto
ultimately
show ?thesis
using (?inv' ?state-up)
using (getSATFlag state = UNDEF)
using (~ getConflictFlag ?state-up)
using False
using WatchInvariantsAfterAssertLiteral[of ?state-up ?literal
True]
using InvariantWatchCharacterizationAfterAssertLiteral[of
?state-up ?literal True]
using InvariantUniqQAAfterAssertLiteral[of ?state-up ?literal
True]
using assertLiteralEffect[of ?state-up ?literal True]
unfolding solve-loop-body-def
unfolding applyDecide-def
unfolding selectLiteral-def
by (simp add: Let-def)
qed
qed
qed

lemma 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
  getSATFlag 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[af terminationLessState1 (vars F0 ∪ Vbl)])
case 1
thus ?case
  using finite Vbl;
  using finiteVarsFormula[af F0]
  using wellFoundedTerminationLessState1[af vars F0 ∪ Vbl]
by simp
next
case (2 state')
note ih = this
show ?case
proof (cases getSATFlag state' = UNDEF)
  case False
    show ?thesis
      apply (rule solve-loop-dom.intros)
      using False
      by simp
next
case True
  let ?state'" = solve-loop-body state' Vbl
  have
    InvariantConsistent (getM ?state"")
    InvariantUniq (getM ?state"")
    InvariantWatchesEl (getF ?state"") (getWatch1 ?state"") (getWatch2 ?state"")
    InvariantWatchesDiffer (getF ?state"") (getWatch1 ?state"") (getWatch2 ?state"")
    InvariantWatchCharacterization (getF ?state"") (getWatch1 ?state"") (getWatch2 ?state"")
    InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state"") (getF ?state"")
    InvariantWatchListsUniq (getWatchList ?state"")
    InvariantWatchListsCharacterization (getWatchList ?state"") (getWatchList ?state"") (getWatchList ?state"")
    getSATFlag ?state"") = FALSE \rightarrow \neg \text{satisfiable } F0'
    getSATFlag ?state"") = TRUE \rightarrow \text{satisfiable } F0'
(* ?state", state') \in \text{terminationLessState1 (vars } F0 \cup Vbl)
using Invariants.AfterSolveLoopBody[of state' F0' Vbl F0]
using 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)
using True
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 \rightarrow InvariantEquivalentZL (getF state) (getM state) F0'
and
   InvariantConflictClauseCharacterization (getConflictFlag state) (getConflictClause state) (getF state) (getM state)
   getSATFlag state = FALSE \rightarrow \neg satisfiable F0'
   getSATFlag state = TRUE \rightarrow satisfiable F0'
finite Vbl
vars F0' \subseteq vars F0
vars F0 \subseteq Vbl
InvariantVarsM (getM state) F0 Vbl

728
\[\text{InvariantVarsF} \ (\text{getF\ state}) \ F0 \ Vbl\]
\[\text{InvariantVarsQ} \ (\text{getQ\ state}) \ F0 \ Vbl\]

shows
\[\text{let state}' = \text{solve-loop\ state\ Vbl\ in}\]
\[\quad (\text{getSATFlag\ state}' = \text{FALSE} \land \neg\ \text{satisfiable\ F0'}) \lor (\text{getSATFlag\ state}' = \text{TRUE} \land \text{satisfiable\ F0'})\]

using assms

proof (induct state\ Vbl\ rule: solve-loop-dom.induct)
case (step state'\ Vbl)

note \(ih = this\)

show ?case
proof (cases getSATFlag state' = UNDEF)
case False
with solve-loop.simps[of state']
have solve-loop state'\ Vbl = state'
by simp
thus ?thesis
using False
using ih(19) ih(20)
using ExtendedBool.nchotomy
by (auto simp add: Let-def)

next
case True
let ?state'' = solve-loop-body state'\ Vbl
have solve-loop state'\ Vbl = solve-loop ?state''\ Vbl
  using solve-loop.simps[of state']
  using True
  by (simp add: Let-def)
moreover
have InvariantEquivalentZL (getF state')\ (getM state')\ F0'
  using True
  using ih(17)
  by simp
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
(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
InvariantConflictClauseCharacterization (getConflictFlag ?state)
(getConflictClause ?state) (getF ?state) (getM ?state)
InvariantGetReasonIsReason (getReason ?state) (getF ?state)
(getM ?state) (set (getQ ?state))
InvariantEquivalentZL (getF ?state) (getM ?state) F0'
InvariantVarsM (getM ?state) F0 Vbl
InvariantVarsQ (getQ ?state) F0 Vbl
InvariantVarsF (getF ?state) F0 Vbl
getSATFlag ?state = FALSE → ¬ satisfiable F0'
getSATFlag ?state = TRUE → satisfiable F0'
using ih(1) ih(3) ih(4) ih(5) ih(6) ih(7) ih(8) ih(9) ih(10)
ih(11) ih(12) ih(13) ih(14)
ihar(15) ih(16) ih(18) ih(21) ih(22) ih(23) ih(24) ih(25)
ihar(26)
using InvariantsAfterSolveLoopBody[of state F0 Vbl F0]
using True
by (auto simp only: Let-def)
ultimately
show ?thesis
using True
using ih(2)
using ih(21)
using ih(22)
using ih(23)
by (simp add: Let-def)
qed
qed

end
theory FunctionalImplementation
imports Initialization SolveLoop
begin

8.2 Total correctness theorem

theorem correctness:
shows
(solve F0 = TRUE ∧ satisfiable F0) ∨ (solve F0 = FALSE ∧ ¬ satisfiable F0)
proof−
let \(?i\)state = initialize \(F_0\) initialState
let \(?F_0'\) = filter (\(\lambda c. \neg\) clauseTautology \(c\)) \(F_0\)

have
  InvariantConsistent (getM ?i\)state)
  InvariantUniq (getM ?i\)state)
  InvariantWatchesEl (getF ?i\)state) (getWatch1 ?i\)state) (getWatch2 ?i\)state) and
  InvariantWatchesDiffer (getF ?i\)state) (getWatch1 ?i\)state) (getWatch2 ?i\)state) and
  InvariantWatchCharacterization (getF ?i\)state) (getWatch1 ?i\)state) (getWatch2 ?i\)state) (getM ?i\)state)
  InvariantWatchListsContainOnlyClausesFromF (getWatchList ?i\)state) (getF ?i\)state) (getM ?i\)state)
  InvariantWatchListsUniq (getWatchList ?i\)state)
  InvariantWatchListsCharacterization (getWatchList ?i\)state) (getWatch1 ?i\)state) (getWatch2 ?i\)state) (getM ?i\)state)
  InvariantConflictFlagCharacterization (getConflictFlag ?i\)state) (getF ?i\)state) (getM ?i\)state)
  InvariantConflictClauseCharacterization (getConflictClause ?i\)state) (getF ?i\)state) (getM ?i\)state)
  InvariantNoDecisionsWhenConflict (getF ?i\)state) (getM ?i\)state) (currentLevel (getM ?i\)state)) and
  InvariantNoDecisionsWhenUnit (getF ?i\)state) (getM ?i\)state) (currentLevel (getM ?i\)state)) and
  InvariantGetReasonIsReason (getReason ?i\)state) (getF ?i\)state) (getM ?i\)state) (set (getQ ?i\)state)) and
  InvariantConflictClauseCharacterization (getConflictFlag ?i\)state) (getF ?i\)state) (getM ?i\)state)
  getSATFlag ?i\)state = UNDEF \(\rightarrow\) InvariantEquivalentZL (getF ?i\)state) (getM ?i\)state)
  getSATFlag ?i\)state = FALSE \(\rightarrow\) \(\neg\) satisfiable \(F_0'\)
  getSATFlag ?i\)state = TRUE \(\rightarrow\) satisfiable \(F_0\)
using
  assms
  InvariantsAfterInitialization[of \(F_0\)]
  InvariantEquivalentZLAfterInitialization[of \(F_0\)]
unfolding
  InvariantVarsM-def
  InvariantVarsF-def
  InvariantVarsQ-def
by
  (auto simp add: Let-def)

moreover
hence
  solve-loop-dom ?i\)state (vars \(F_0\))
using
  SolveLoopTermination[of ?i\)state \(\rightarrow\) satisfiable \(F_0'\) vars \(F_0\) \(F_0\)]
using
  finiteVarsFormula[of \(F_0\)]
using
  varsSubsetFormula[of \(F_0'\) \(F_0\)]
by
  auto

731
ultimately

show ?thesis
  using finiteVarsFormula[of F0]
  using SATFlagAfterSolveLoop[of ?istate vars F0 ?F0’ F0]
  using satisfiableFilterTautologies[of F0]
  unfolding solve-def
  using varsSubsetFormula[of ?F0’ F0]
  by (auto simp add: Let-def)

qed

end

References

