SAT Solver verification

By Filip Marić

December 14, 2021

Abstract

This document contains formall 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 (similiar to the description given in [3]), and
3. a solver with a specific conflict analysis algorithm (similiar 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

theory *MoreList*
imports *Main 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 similiar form.

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

lemma *listEqualsButlastAppendLast*:
assumes $list \neq []$
shows $list = (butlast\ list)\ @\ [last\ list]$
⟨*proof*⟩

lemma *lastListInList* [*simp*]:
assumes $list \neq []$
shows $last\ list \in set\ list$
⟨*proof*⟩

lemma *butlastIsSubset*:
shows *set* (*butlast list*) ⊆ *set list*
⟨*proof*⟩

lemma *setListIsSetButlastAndLast*:
shows *set list* ⊆ *set* (*butlast list*) ∪ {*last list*}
⟨*proof*⟩

lemma *butlastAppend*:
shows *butlast* (*list1* @ *list2*) = (*if list2* = [] *then butlast list1 else* (*list1* @ *butlast list2*))
⟨*proof*⟩

1.2 *removeAll* - element removal

lemma *removeAll-multiset*:
assumes *distinct a x* ∈ *set a*
shows *mset a* = {#*x*#} + *mset* (*removeAll x a*)
⟨*proof*⟩

lemma *removeAll-map*:
assumes ∀ *x y*. *x* ≠ *y* ⟶ *f x* ≠ *f y*
shows *removeAll* (*f x*) (*map f a*) = *map f* (*removeAll x a*)
⟨*proof*⟩

1.3 *uniq* - no duplicate elements.

uniq list holds iff there are no repeated elements in a list. Obsolete: same as *distinct* in *List.thy*.

primrec *uniq* :: *'a list* => *bool*
where
uniq [] = *True* |
uniq (*h*#*t*) = (*h* ∉ *set t* ∧ *uniq t*)

lemma *uniqDistinct*:
uniq l = *distinct l*
⟨*proof*⟩

lemma *uniqAppend*:
assumes *uniq* (*l1* @ *l2*)
shows *uniq l1 uniq l2*
⟨*proof*⟩

lemma *uniqAppendIff*:
uniq (*l1* @ *l2*) = (*uniq l1* ∧ *uniq l2* ∧ *set l1* ∩ *set l2* = {}) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *uniqAppendElement*:

assumes *uniq l*
shows *e* ∉ *set l* = *uniq* (*l* @ [*e*])
⟨*proof*⟩

lemma *uniqImpliesNotLastMemButlast*:
assumes *uniq l*
shows *last l* ∉ *set* (*butlast l*)
⟨*proof*⟩

lemma *uniqButlastNotUniqListImpliesLastMemButlast*:
assumes *uniq* (*butlast l*) ¬ *uniq l*
shows *last l* ∈ *set* (*butlast l*)
⟨*proof*⟩

lemma *uniqRemdups*:
shows *uniq* (*remdups x*)
⟨*proof*⟩

lemma *uniqHeadTailSet*:
assumes *uniq l*
shows *set* (*tl l*) = (*set l*) − {*hd l*}
⟨*proof*⟩

lemma *uniqLengthEqCardSet*:
assumes *uniq l*
shows *length l* = *card* (*set l*)
⟨*proof*⟩

lemma *lengthGtOneTwoDistinctElements*:
assumes
uniq l length l > *1 l* ≠ []
shows
∃ *a1 a2*. *a1* ∈ *set l* ∧ *a2* ∈ *set l* ∧ *a1* ≠ *a2*
⟨*proof*⟩

1.4 *firstPos* - first position of an element

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.

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

lemma *firstPosEqualZero*:
shows (*firstPos a* (*m # M'*) = *0*) = (*a* = *m*)
⟨*proof*⟩

lemma *firstPosLeLength*:
assumes $a \in set\ l$
shows $firstPos\ a\ l < length\ l$
⟨*proof*⟩

lemma *firstPosAppend*:
assumes $a \in set\ l$
shows $firstPos\ a\ l = firstPos\ a\ (l\ @\ l')$
⟨*proof*⟩

lemma *firstPosAppendNonMemberFirstMemberSecond*:
assumes $a \notin set\ l1$ **and** $a \in set\ l2$
shows $firstPos\ a\ (l1\ @\ l2) = length\ l1 + firstPos\ a\ l2$
⟨*proof*⟩

lemma *firstPosDomainForElements*:
shows $(0 \leq firstPos\ a\ l \wedge firstPos\ a\ l < length\ l) = (a \in set\ l)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *firstPosEqual*:
assumes $a \in set\ l$ **and** $b \in set\ l$
shows $(firstPos\ a\ l = firstPos\ b\ l) = (a = b)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *firstPosLast*:
assumes $l \neq []$ $uniq\ l$
shows $(firstPos\ x\ l = length\ l - 1) = (x = last\ l)$
⟨*proof*⟩

1.5 *precedes* - ordering relation induced by *firstPos*

definition *precedes* :: *'a => 'a => 'a list => bool*
where
$precedes\ a\ b\ l == (a \in set\ l \wedge b \in set\ l \wedge\ firstPos\ a\ l <= firstPos\ b\ l)$

lemma *noElementsPrecedesFirstElement*:
assumes $a \neq b$
shows $\neg\ precedes\ a\ b\ (b\ \#\ list)$
⟨*proof*⟩

lemma *lastPrecedesNoElement*:
assumes $uniq\ l$
shows $\neg(\exists\ a.\ a \neq last\ l \wedge precedes\ (last\ l)\ a\ l)$
⟨*proof*⟩

lemma *precedesAppend*:
assumes $precedes\ a\ b\ l$

shows *precedes a b* (*l* @ *l'*)
⟨*proof*⟩

lemma *precedesMemberHeadMemberTail*:
assumes *a* ∈ *set l1* **and** *b* ∉ *set l1* **and** *b* ∈ *set l2*
shows *precedes a b* (*l1* @ *l2*)
⟨*proof*⟩

lemma *precedesReflexivity*:
assumes *a* ∈ *set l*
shows *precedes a a l*
⟨*proof*⟩

lemma *precedesTransitivity*:
assumes
precedes a b l **and** *precedes b c l*
shows
precedes a c l
⟨*proof*⟩

lemma *precedesAntisymmetry*:
assumes
a ∈ *set l* **and** *b* ∈ *set l* **and**
precedes a b l **and** *precedes b a l*
shows
a = *b*
⟨*proof*⟩

lemma *precedesTotalOrder*:
assumes *a* ∈ *set l* **and** *b* ∈ *set l*
shows *a*=*b* ∨ *precedes a b l* ∨ *precedes b a l*
⟨*proof*⟩

lemma *precedesMap*:
assumes *precedes a b list* **and** ∀ *x y*. *x* ≠ *y* ⟶ *f x* ≠ *f y*
shows *precedes* (*f a*) (*f b*) (*map f list*)
⟨*proof*⟩

lemma *precedesFilter*:
assumes *precedes a b list* **and** *f a* **and** *f b*
shows *precedes a b* (*filter f list*)
⟨*proof*⟩

definition
precedesOrder list == {(*a*, *b*). *precedes a b list* ∧ *a* ≠ *b*}

lemma *transPrecedesOrder*:
trans (*precedesOrder list*)

⟨*proof*⟩

lemma *wellFoundedPrecedesOrder*:
shows *wf* (*precedesOrder list*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *uniqListImpliesUniqPrefix*:
assumes *isPrefix p l* **and** *uniq l*
shows *uniq p*
⟨*proof*⟩

lemma *firstPosPrefixElement*:
assumes *isPrefix p l* **and** *a* ∈ *set p*
shows *firstPos a p* = *firstPos a l*
⟨*proof*⟩

lemma *laterInPrefixRetainsPrecedes*:
assumes
isPrefix p l **and** *precedes a b l* **and** *b* ∈ *set p*
shows
precedes a b p
⟨*proof*⟩

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

primrec *list-diff* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
list-diff x [] = *x* |
list-diff x (*y*#*ys*) = *list-diff* (*removeAll y x*) *ys*

lemma [*simp*]:
shows *list-diff* [] *y* = []
⟨*proof*⟩

lemma [*simp*]:
shows *list-diff* (*x* # *xs*) *y* = (*if x* ∈ *set y then list-diff xs y else x* # *list-diff xs y*)
⟨*proof*⟩

lemma *listDiffIff*:
shows (*x* ∈ *set a* ∧ *x* ∉ *set b*) = (*x* ∈ *set* (*list-diff a b*))
⟨*proof*⟩

lemma *listDiffDoubleRemoveAll*:
assumes *x* ∈ *set a*
shows *list-diff b a* = *list-diff b* (*x* # *a*)
⟨*proof*⟩

lemma *removeAllListDiff*[*simp*]:
shows *removeAll x* (*list-diff a b*) = *list-diff* (*removeAll x a*) *b*
⟨*proof*⟩

lemma *listDiffRemoveAllNonMember*:
assumes *x* ∉ *set a*
shows *list-diff a b* = *list-diff a* (*removeAll x b*)
⟨*proof*⟩

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*)
⟨*proof*⟩

1.8 *remdups* - removing duplicates

lemma *remdupsRemoveAllCommute*[*simp*]:
shows *remdups* (*removeAll a list*) = *removeAll a* (*remdups list*)
⟨*proof*⟩

lemma *remdupsAppend*:
shows *remdups* (*a* @ *b*) = *remdups* (*list-diff a b*) @ *remdups b*
⟨*proof*⟩

lemma *remdupsAppendSet*:
shows *set* (*remdups* (*a* @ *b*)) = *set* (*remdups a* @ *remdups* (*list-diff b a*))
⟨*proof*⟩

lemma *remdupsAppendMultiSet*:
shows *mset* (*remdups* (*a* @ *b*)) = *mset* (*remdups a* @ *remdups* (*list-diff b a*))
⟨*proof*⟩

lemma *remdupsListDiff*:

remdups (*list-diff a b*) = *list-diff* (*remdups a*) (*remdups b*)
⟨*proof*⟩

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

lemma *multisetEmptyLeI*:
multiset-le {#} *a r*
⟨*proof*⟩

lemma *multisetUnionLessMono2*:
shows
trans r ⟹ (*b1*, *b2*) ∈ *mult r* ⟹ (*a* + *b1*, *a* + *b2*) ∈ *mult r*
⟨*proof*⟩

lemma *multisetUnionLessMono1*:
shows
trans r ⟹ (*a1*, *a2*) ∈ *mult r* ⟹ (*a1* + *b*, *a2* + *b*) ∈ *mult r*
⟨*proof*⟩

lemma *multisetUnionLeMono2*:
assumes
trans r
multiset-le b1 b2 r
shows
multiset-le (*a* + *b1*) (*a* + *b2*) *r*
⟨*proof*⟩

lemma *multisetUnionLeMono1*:
assumes
trans r
multiset-le a1 a2 r
shows
multiset-le (*a1* + *b*) (*a2* + *b*) *r*
⟨*proof*⟩

lemma *multisetLeTrans*:
assumes
trans r
multiset-le x y r
multiset-le y z r

shows
multiset-le x z r
⟨*proof*⟩

lemma *multisetUnionLeMono*:
assumes
trans r
multiset-le a1 a2 r
multiset-le b1 b2 r
shows
multiset-le (*a1* + *b1*) (*a2* + *b2*) *r*
⟨*proof*⟩

lemma *multisetLeListDiff*:
assumes
trans r
shows
multiset-le (*mset* (*list-diff a b*)) (*mset a*) *r*
⟨*proof*⟩

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*⟩

lemma *SimpleLevi*:
shows (*p* @ *s* = *a* # *list*) =
(*p* = [] ∧ *s* = *a* # *list* ∨
(∃ *t*. *p* = *a* # *t* ∧ *t* @ *s* = *list*))
⟨*proof*⟩

1.10 Single element lists

lemma *lengthOneCharacterisation*:
shows (*length l* = *1*) = (*l* = [*hd l*])
⟨*proof*⟩

lemma *lengthOneImpliesOnlyElement*:
assumes *length l* = *1* **and** *a* : *set l*
shows ∀ *a'*. *a'* : *set l* ⟶ *a'* = *a*
⟨*proof*⟩

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* :: $'a \Rightarrow 'b \Rightarrow$ *bool* (**infixl** *el 55*)

overloading *literalElClause* ≡ *member* :: *Literal* ⇒ *Clause* ⇒ *bool*
begin
definition [*simp*]: ((*literal*::*Literal*) *el* (*clause*::*Clause*)) == *literal* ∈ *set clause*
end

overloading *clauseElFormula* ≡ *member* :: *Clause* ⇒ *Formula* ⇒ *bool*
begin
definition [*simp*]: ((*clause*::*Clause*) *el* (*formula*::*Formula*)) == *clause* ∈ *set formula*
end

overloading *el-literal* ≡ (*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*)
⟨*proof*⟩

2.1.3 Variables

The variable of a given literal

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

Set of variables of a given clause, formula or valuation

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

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

consts *vars* :: *'a* ⇒ *Variable set*

overloading *vars-clause* ≡ *vars* :: *Clause* ⇒ *Variable set*
begin
definition [*simp*]: *vars* (*clause*::*Clause*) == *varsClause clause*
end

overloading *vars-formula* ≡ *vars* :: *Formula* ⇒ *Variable set*
begin
definition [*simp*]: *vars* (*formula*::*Formula*) == *varsFormula formula*
end

overloading *vars-set* ≡ *vars* :: *Literal set* ⇒ *Variable set*
begin

definition [*simp*]: *vars* (*s*::*Literal set*) == {*vbl*. ∃ *l*. *l* ∈ *s* ∧ *var l* = *vbl*}
end

lemma *clauseContainsItsLiteralsVariable*:
fixes *literal* :: *Literal* **and** *clause* :: *Clause*
assumes *literal el clause*
shows *var literal* ∈ *vars clause*
⟨*proof*⟩

lemma *formulaContainsItsLiteralsVariable*:
fixes *literal* :: *Literal* **and** *formula*::*Formula*
assumes *literal el formula*
shows *var literal* ∈ *vars formula*
⟨*proof*⟩

lemma *formulaContainsItsClausesVariables*:
fixes *clause* :: *Clause* **and** *formula* :: *Formula*
assumes *clause el formula*
shows *vars clause* ⊆ *vars formula*
⟨*proof*⟩

lemma *varsAppendFormulae*:
fixes *formula1* :: *Formula* **and** *formula2* :: *Formula*
shows *vars* (*formula1* @ *formula2*) = *vars formula1* ∪ *vars formula2*
⟨*proof*⟩

lemma *varsAppendClauses*:
fixes *clause1* :: *Clause* **and** *clause2* :: *Clause*
shows *vars* (*clause1* @ *clause2*) = *vars clause1* ∪ *vars clause2*
⟨*proof*⟩

lemma *varsRemoveLiteral*:
fixes *literal* :: *Literal* **and** *clause* :: *Clause*
shows *vars* (*removeAll literal clause*) ⊆ *vars clause*
⟨*proof*⟩

lemma *varsRemoveLiteralSuperset*:
fixes *literal* :: *Literal* **and** *clause* :: *Clause*
shows *vars clause* − {*var literal*} ⊆ *vars* (*removeAll literal clause*)
⟨*proof*⟩

lemma *varsRemoveAllClause*:
fixes *clause* :: *Clause* **and** *formula* :: *Formula*
shows *vars* (*removeAll clause formula*) ⊆ *vars formula*
⟨*proof*⟩

lemma *varsRemoveAllClauseSuperset*:
fixes *clause* :: *Clause* **and** *formula* :: *Formula*

shows *vars formula − vars clause ⊆ vars* (*removeAll clause formula*)
⟨*proof*⟩

lemma *varInClauseVars*:
fixes *variable* :: *Variable* **and** *clause* :: *Clause*
shows *variable* ∈ *vars clause* = (∃ *literal*. *literal el clause* ∧ *var literal* = *variable*)
⟨*proof*⟩

lemma *varInFormulaVars*:
fixes *variable* :: *Variable* **and** *formula* :: *Formula*
shows *variable* ∈ *vars formula* = (∃ *literal*. *literal el formula* ∧ *var literal* = *variable*) (**is** *?lhs formula* = *?rhs formula*)
⟨*proof*⟩

lemma *varsSubsetFormula*:
fixes *F* :: *Formula* **and** *F′* :: *Formula*
assumes ∀ *c*::*Clause*. *c el F* ⟶ *c el F′*
shows *vars F* ⊆ *vars F′*
⟨*proof*⟩

lemma *varsClauseVarsSet*:
fixes
clause :: *Clause*
shows
vars clause = *vars* (*set clause*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *oppositeSymmetry* [*simp*]:
fixes *literal1*::*Literal* **and** *literal2*::*Literal*
shows (*opposite literal1* = *literal2*) = (*opposite literal2* = *literal1*)
⟨*proof*⟩

lemma *oppositeUniqueness* [*simp*]:
fixes *literal1*::*Literal* **and** *literal2*::*Literal*
shows (*opposite literal1* = *opposite literal2*) = (*literal1* = *literal2*)

⟨*proof*⟩

lemma *oppositeIsDifferentFromLiteral* [*simp*]:
fixes *literal*::*Literal*
shows *opposite literal* ≠ *literal*
⟨*proof*⟩

lemma *oppositeLiteralsHaveSameVariable* [*simp*]:
fixes *literal*::*Literal*
shows *var* (*opposite literal*) = *var literal*
⟨*proof*⟩

lemma *literalsWithSameVariableAreEqualOrOpposite*:
fixes *literal1*::*Literal* **and** *literal2*::*Literal*
shows (*var literal1* = *var literal2*) = (*literal1* = *literal2* ∨ *opposite literal1* = *literal2*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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
oppositeLiteralList :: *Literal list* ⇒ *Literal list*
where
oppositeLiteralList clause == *map opposite clause*

lemma *literalElListIffOppositeLiteralElOppositeLiteralList*:
fixes *literal* :: *Literal* **and** *literalList* :: *Literal list*
shows *literal el literalList* = (*opposite literal*) *el* (*oppositeLiteralList literalList*)
⟨*proof*⟩

lemma *oppositeLiteralListIdempotency* [*simp*]:
fixes *literalList* :: *Literal list*
shows *oppositeLiteralList* (*oppositeLiteralList literalList*) = *literalList*
⟨*proof*⟩

lemma *oppositeLiteralListRemove*:
fixes *literal* :: *Literal* **and** *literalList* :: *Literal list*
shows *oppositeLiteralList* (*removeAll literal literalList*) = *removeAll* (*opposite literal*) (*oppositeLiteralList literalList*)
⟨*proof*⟩

lemma *oppositeLiteralListNonempty*:
fixes *literalList* :: *Literal list*
shows (*literalList* ≠ []) = ((*oppositeLiteralList literalList*) ≠ [])
⟨*proof*⟩

lemma *varsOppositeLiteralList*:
shows *vars* (*oppositeLiteralList clause*) = *vars clause*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *varsSubsetValuation*:
fixes *valuation1* :: *Valuation* **and** *valuation2* :: *Valuation*
assumes *set valuation1* ⊆ *set valuation2*
shows *vars valuation1* ⊆ *vars valuation2*
⟨*proof*⟩

lemma *varsAppendValuation*:
fixes *valuation1* :: *Valuation* **and** *valuation2* :: *Valuation*
shows *vars* (*valuation1* @ *valuation2*) = *vars valuation1* ∪ *vars valuation2*
⟨*proof*⟩
lemma *varsPrefixValuation*:
fixes *valuation1* :: *Valuation* **and** *valuation2* :: *Valuation*
assumes *isPrefix valuation1 valuation2*
shows *vars valuation1* ⊆ *vars valuation2*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *clauseFalseIffAllLiteralsAreFalse*:
fixes *clause* :: *Clause* **and** *valuation* :: *Valuation*
shows *clauseFalse clause valuation* = (∀ *literal*. *literal el clause* ⟶ *literalFalse literal valuation*)
⟨*proof*⟩

lemma *clauseFalseRemove*:
assumes *clauseFalse clause valuation*
shows *clauseFalse* (*removeAll literal clause*) *valuation*
⟨*proof*⟩

lemma *clauseFalseAppendValuation*:
fixes *clause* :: *Clause* **and** *valuation* :: *Valuation* **and** *valuation'* :: *Valuation*
assumes *clauseFalse clause valuation*
shows *clauseFalse clause* (*valuation* @ *valuation'*)
⟨*proof*⟩

lemma *clauseTrueAppendValuation*:
fixes *clause* :: *Clause* **and** *valuation* :: *Valuation* **and** *valuation'* :: *Valuation*
assumes *clauseTrue clause valuation*
shows *clauseTrue clause* (*valuation* @ *valuation'*)
⟨*proof*⟩

lemma *emptyClauseIsFalse*:
fixes *valuation* :: *Valuation*
shows *clauseFalse* [] *valuation*
⟨*proof*⟩

lemma *emptyValuationFalsifiesOnlyEmptyClause*:
fixes *clause* :: *Clause*
assumes *clause* ≠ []
shows ¬ *clauseFalse clause* []
⟨*proof*⟩

lemma *valuationContainsItsFalseClausesVariables*:
fixes *clause*::*Clause* **and** *valuation*::*Valuation*
assumes *clauseFalse clause valuation*
shows *vars clause* ⊆ *vars valuation*
⟨*proof*⟩

2.2.4 True/False formulae

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

primrec

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

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

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

lemma *formulaTrueIffAllClausesAreTrue*:
fixes *formula* :: *Formula* **and** *valuation* :: *Valuation*
shows *formulaTrue formula valuation* = (∀ *clause*. *clause el formula* ⟶ *clauseTrue clause valuation*)
⟨*proof*⟩

lemma *formulaFalseIffContainsFalseClause*:
fixes *formula* :: *Formula* **and** *valuation* :: *Valuation*
shows *formulaFalse formula valuation* = (∃ *clause*. *clause el formula* ∧ *clauseFalse clause valuation*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *formulaTrueCommutativity*:
fixes *f1* :: *Formula* **and** *f2* :: *Formula* **and** *valuation* :: *Valuation*
shows *formulaTrue* (*f1* @ *f2*) *valuation* = *formulaTrue* (*f2* @ *f1*) *valuation*
⟨*proof*⟩

lemma *formulaTrueSubset*:
fixes *formula* :: *Formula* **and** *formula′* :: *Formula* **and** *valuation* :: *Valuation*
assumes
formulaTrue: *formulaTrue formula valuation* **and**
subset: ∀ (*clause*::*Clause*). *clause el formula′* ⟶ *clause el formula*
shows *formulaTrue formula′ valuation*
⟨*proof*⟩

lemma *formulaTrueAppend*:
fixes *formula1* :: *Formula* **and** *formula2* :: *Formula* **and** *valuation* :: *Valuation*
shows *formulaTrue* (*formula1* @ *formula2*) *valuation* = (*formulaTrue formula1 valuation* ∧ *formulaTrue formula2 valuation*)
⟨*proof*⟩

lemma *formulaTrueRemoveAll*:
fixes *formula* :: *Formula* **and** *clause* :: *Clause* **and** *valuation* :: *Valuation*
assumes *formulaTrue formula valuation*
shows *formulaTrue* (*removeAll clause formula*) *valuation*
⟨*proof*⟩

lemma *formulaFalseAppend*:
fixes *formula* :: *Formula* **and** *formula′* :: *Formula* **and** *valuation* :: *Valuation*
assumes *formulaFalse formula valuation*
shows *formulaFalse* (*formula* @ *formula′*) *valuation*
⟨*proof*⟩

lemma *formulaTrueAppendValuation*:
fixes *formula* :: *Formula* **and** *valuation* :: *Valuation* **and** *valuation′* :: *Valuation*
assumes *formulaTrue formula valuation*
shows *formulaTrue formula* (*valuation* @ *valuation′*)
⟨*proof*⟩

lemma *formulaFalseAppendValuation*:
fixes *formula* :: *Formula* **and** *valuation* :: *Valuation* **and** *valuation′* :: *Valuation*
assumes *formulaFalse formula valuation*
shows *formulaFalse formula* (*valuation* @ *valuation′*)
⟨*proof*⟩

lemma *trueFormulaWithSingleLiteralClause*:
fixes *formula* :: *Formula* **and** *literal* :: *Literal* **and** *valuation* :: *Valuation*
assumes *formulaTrue* (*removeAll* [*literal*] *formula*) (*valuation* @ [*literal*])
shows *formulaTrue formula* (*valuation* @ [*literal*])
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *val2FormAreSingleLiteralClauses*:
fixes *clause* :: *Clause* **and** *valuation* :: *Valuation*
shows *clause el val2form valuation* ⟶ (∃ *literal*. *clause* = [*literal*] ∧ *literal el valuation*)
⟨*proof*⟩

lemma *val2formOfSingleLiteralValuation*:
assumes *length v* = *1*
shows *val2form v* = [[*hd v*]]
⟨*proof*⟩

lemma *val2FormRemoveAll*:
fixes *literal* :: *Literal* **and** *valuation* :: *Valuation*
shows *removeAll* [*literal*] (*val2form valuation*) = *val2form* (*removeAll literal valuation*)
⟨*proof*⟩

lemma *val2formAppend*:
fixes *valuation1* :: *Valuation* **and** *valuation2* :: *Valuation*
shows *val2form* (*valuation1* @ *valuation2*) = (*val2form valuation1* @ *val2form valuation2*)
⟨*proof*⟩

lemma *val2formFormulaTrue*:
fixes *valuation1* :: *Valuation* **and** *valuation2* :: *Valuation*
shows *formulaTrue* (*val2form valuation1*) *valuation2* = (∀ (*literal* :: *Literal*). *literal el valuation1* ⟶ *literal el valuation2*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *clauseTrueAndClauseFalseImpliesInconsistent*:
fixes *clause* :: *Clause* **and** *valuation* :: *Valuation*
assumes *clauseTrue clause valuation* **and** *clauseFalse clause valuation*
shows *inconsistent valuation*
⟨*proof*⟩

lemma *formulaTrueAndFormulaFalseImpliesInconsistent*:
fixes *formula* :: *Formula* **and** *valuation* :: *Valuation*
assumes *formulaTrue formula valuation* **and** *formulaFalse formula valuation*
shows *inconsistent valuation*
⟨*proof*⟩

lemma *inconsistentAppend*:
fixes *valuation1* :: *Valuation* **and** *valuation2* :: *Valuation*
assumes *inconsistent* (*valuation1* @ *valuation2*)
shows *inconsistent valuation1* ∨ *inconsistent valuation2* ∨ (∃ *literal*. *literalTrue literal valuation1* ∧ *literalFalse literal valuation2*)
⟨*proof*⟩

lemma *consistentAppendElement*:
assumes *consistent v* **and** ¬ *literalFalse l v*
shows *consistent* (*v* @ [*l*])
⟨*proof*⟩

lemma *inconsistentRemoveAll*:
fixes *literal* :: *Literal* **and** *valuation* :: *Valuation*
assumes *inconsistent* (*removeAll literal valuation*)
shows *inconsistent valuation*
⟨*proof*⟩

lemma *inconsistentPrefix*:
assumes *isPrefix valuation1 valuation2* **and** *inconsistent valuation1*
shows *inconsistent valuation2*
⟨*proof*⟩

lemma *consistentPrefix*:
assumes *isPrefix valuation1 valuation2* **and** *consistent valuation2*
shows *consistent valuation1*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

lemma *totalValuationForClauseDefinesItsValue*:
fixes *clause* :: *Clause* **and** *valuation* :: *Valuation*
assumes *totalClause*: *total valuation* (*vars clause*)
shows *clauseTrue clause valuation* ∨ *clauseFalse clause valuation*
⟨*proof*⟩

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*⟩

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*⟩

lemma *totalValuationForFormulaDefinesItsValue*:
assumes *totalFormula*: *total valuation* (*vars formula*)
shows *formulaTrue formula valuation* ∨ *formulaFalse formula valuation*
⟨*proof*⟩

lemma *totalRemoveAllSingleLiteralClause*:
fixes *literal* :: *Literal* **and** *valuation* :: *Valuation* **and** *formula* :: *Formula*
assumes *varLiteral*: *var literal* ∈ *vars valuation* **and** *totalRemoveAll*: *total valuation* (*vars* (*removeAll* [*literal*] *formula*))
shows *total valuation* (*vars formula*)
⟨*proof*⟩

2.2.8 Models and satisfiability

Model of a formula is a consistent valuation under which formula/clause is true

consts *model* :: *Valuation* ⇒ *'a* ⇒ *bool*

overloading *modelFormula* ≡ *model* :: *Valuation* ⇒ *Formula* ⇒ *bool*
begin
definition [*simp*]: *model valuation* (*formula*::*Formula*) ==
consistent valuation ∧ (*formulaTrue formula valuation*)
end

overloading *modelClause* ≡ *model* :: *Valuation* ⇒ *Clause* ⇒ *bool*
begin
definition [*simp*]: *model valuation* (*clause*::*Clause*) ==
consistent valuation ∧ (*clauseTrue clause valuation*)
end

Checks if a formula has a model

definition *satisfiable* :: *Formula* ⇒ *bool*
where
satisfiable formula == ∃ *valuation*. *model valuation formula*

lemma *formulaWithEmptyClauseIsUnsatisfiable*:
fixes *formula* :: *Formula*
assumes ([]::*Clause*) *el formula*
shows ¬ *satisfiable formula*
⟨*proof*⟩

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*⟩

lemma *satisfiableAppend*:
fixes *formula1* :: *Formula* **and** *formula2* :: *Formula*
assumes *satisfiable* (*formula1* @ *formula2*)
shows *satisfiable formula1 satisfiable formula2*
⟨*proof*⟩

lemma *modelExpand*:
fixes *formula* :: *Formula* **and** *literal* :: *Literal* **and** *valuation* :: *Valuation*
assumes *model valuation formula* **and** *var literal* $\notin$ *vars valuation*
shows *model* (*valuation* @ [*literal*]) *formula*
⟨*proof*⟩

2.2.9 Tautological clauses

lemma *tautologyNotFalse*:
fixes *clause* :: *Clause* **and** *valuation* :: *Valuation*
assumes *clauseTautology clause consistent valuation*
shows $\neg$ *clauseFalse clause valuation*
⟨*proof*⟩

lemma *tautologyInTotalValuation*:
assumes
clauseTautology clause
vars clause $\subseteq$ *vars valuation*
shows
clauseTrue clause valuation
⟨*proof*⟩

lemma *modelAppendTautology*:
assumes
model valuation F clauseTautology c
vars valuation $\supseteq$ *vars F* $\cup$ *vars c*
shows
model valuation (*F* @ [*c*])
⟨*proof*⟩

lemma *satisfiableAppendTautology*:
assumes
satisfiable F clauseTautology c
shows
satisfiable (*F* @ [*c*])
⟨*proof*⟩

lemma *modelAppendTautologicalFormula*:
fixes
F :: *Formula* **and** *F′* :: *Formula*
assumes
model valuation F ∀ *c*. *c el F′* ⟶ *clauseTautology c*
vars valuation ⊇ *vars F* ∪ *vars F′*
shows
model valuation (*F* @ *F′*)
⟨*proof*⟩

lemma *satisfiableAppendTautologicalFormula*:
assumes
satisfiable F ∀ *c*. *c el F′* ⟶ *clauseTautology c*
shows
satisfiable (*F* @ *F′*)
⟨*proof*⟩

lemma *satisfiableFilterTautologies*:
shows *satisfiable F* = *satisfiable* (*filter* (% *c*. ¬ *clauseTautology c*) *F*)
⟨*proof*⟩

lemma *modelFilterTautologies*:
assumes
model valuation (*filter* (% *c*. ¬ *clauseTautology c*) *F*)
vars F ⊆ *vars valuation*
shows *model valuation F*
⟨*proof*⟩

2.2.10 Entailment

Formula entails literal if it is true in all its models

definition *formulaEntailsLiteral* :: *Formula* ⇒ *Literal* ⇒ *bool*
where
formulaEntailsLiteral formula literal ==
∀ (*valuation*::*Valuation*). *model valuation formula* ⟶ *literalTrue literal valuation*

Clause implies literal if it is true in all its models

definition *clauseEntailsLiteral* :: *Clause* ⇒ *Literal* ⇒ *bool*
where
clauseEntailsLiteral clause literal ==
∀ (*valuation*::*Valuation*). *model valuation clause* ⟶ *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*⟩

lemma *clauseEntailsLiteralThenFormulaEntailsLiteral*:
fixes *clause* :: *Clause* **and** *formula* :: *Formula* **and** *literal* :: *Literal*
assumes *clause el formula* **and** *clauseEntailsLiteral clause literal*
shows *formulaEntailsLiteral formula literal*
⟨*proof*⟩

lemma *formulaEntailsLiteralAppend*:
fixes *formula* :: *Formula* **and** *formula′* :: *Formula* **and** *literal* :: *Literal*
assumes *formulaEntailsLiteral formula literal*
shows *formulaEntailsLiteral* (*formula* @ *formula′*) *literal*
⟨*proof*⟩

lemma *formulaEntailsLiteralSubset*:
fixes *formula* :: *Formula* **and** *formula′* :: *Formula* **and** *literal* :: *Literal*
assumes *formulaEntailsLiteral formula literal* **and** ∀ (*c*::*Clause*) . *c el formula* ⟶ *c el formula′*
shows *formulaEntailsLiteral formula′ literal*
⟨*proof*⟩

lemma *formulaEntailsLiteralRemoveAll*:

fixes *formula* :: *Formula* **and** *clause* :: *Clause* **and** *literal* :: *Literal*
assumes *formulaEntailsLiteral* (*removeAll clause formula*) *literal*
shows *formulaEntailsLiteral formula literal*
⟨*proof*⟩

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*⟩

lemma *formulaEntailsItsClauses*:
fixes *clause* :: *Clause* **and** *formula* :: *Formula*
assumes *clause el formula*
shows *formulaEntailsClause formula clause*
⟨*proof*⟩

lemma *formulaEntailsClauseAppend*:
fixes *clause* :: *Clause* **and** *formula* :: *Formula* **and** *formula′* :: *Formula*
assumes *formulaEntailsClause formula clause*
shows *formulaEntailsClause* (*formula* @ *formula′*) *clause*
⟨*proof*⟩

lemma *formulaUnsatIffImpliesEmptyClause*:
fixes *formula* :: *Formula*
shows *formulaEntailsClause formula* [] = (¬ *satisfiable formula*)
⟨*proof*⟩

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*⟩

lemma *formulaEntailsFormulaIffEntailsAllItsClauses*:
fixes *formula* :: *Formula* **and** *formula′* :: *Formula*
shows *formulaEntailsFormula formula formula′* = (∀ *clause*::*Clause*. *clause el formula′* ⟶ *formulaEntailsClause formula clause*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *formulaEntailsFormulaThatEntailsClause*:

fixes *formula1* :: *Formula* **and** *formula2* :: *Formula* **and** *clause* :: *Clause*
assumes *formulaEntailsFormula formula1 formula2* **and** *formulaEntailsClause formula2 clause*
shows *formulaEntailsClause formula1 clause*
⟨*proof*⟩

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*⟩

lemma *formulaFalseInEntailedValuationIsUnsatisfiable*:
fixes *formula* :: *Formula* **and** *valuation* :: *Valuation*
assumes *formulaFalse formula valuation* **and**
formulaEntailsValuation formula valuation
shows ¬ *satisfiable formula*
⟨*proof*⟩

lemma *formulaFalseInEntailedOrPureValuationIsUnsatisfiable*:
fixes *formula* :: *Formula* **and** *valuation* :: *Valuation*
assumes *formulaFalse formula valuation* **and**
∀ *literal'*. *literal' el valuation* ⟶ *formulaEntailsLiteral formula literal'* ∨ ¬ *opposite literal' el formula*
shows ¬ *satisfiable formula*
⟨*proof*⟩

lemma *unsatisfiableFormulaWithSingleLiteralClause*:
fixes *formula* :: *Formula* **and** *literal* :: *Literal*
assumes ¬ *satisfiable formula* **and** [*literal*] *el formula*
shows *formulaEntailsLiteral* (*removeAll* [*literal*] *formula*) (*opposite literal*)
⟨*proof*⟩

lemma *unsatisfiableFormulaWithSingleLiteralClauses*:
fixes *F*::*Formula* **and** *c*::*Clause*
assumes ¬ *satisfiable* (*F* @ *val2form* (*oppositeLiteralList c*)) ¬ *clauseTautology c*
shows *formulaEntailsClause F c*
⟨*proof*⟩

lemma *satisfiableEntailedFormula*:
fixes *formula0* :: *Formula* **and** *formula* :: *Formula*

assumes *formulaEntailsFormula formula0 formula*
shows *satisfiable formula0* $\longrightarrow$ *satisfiable formula*
⟨*proof*⟩

lemma *val2formIsEntailed*:
shows *formulaEntailsValuation* (*F′* @ *val2form valuation* @ *F″*) *valuation*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *equivalentFormulaeReflexivity*:
fixes *formula* :: *Formula*
shows *equivalentFormulae formula formula*
⟨*proof*⟩

lemma *equivalentFormulaeSymmetry*:
fixes *formula1* :: *Formula* **and** *formula2* :: *Formula*
shows *equivalentFormulae formula1 formula2* = *equivalentFormulae formula2 formula1*
⟨*proof*⟩

lemma *equivalentFormulaeTransitivity*:
fixes *formula1* :: *Formula* **and** *formula2* :: *Formula* **and** *formula3* :: *Formula*
assumes *equivalentFormulae formula1 formula2* **and** *equivalentFormulae formula2 formula3*
shows *equivalentFormulae formula1 formula3*
⟨*proof*⟩

lemma *equivalentFormulaeAppend*:
fixes *formula1* :: *Formula* **and** *formula1′* :: *Formula* **and** *formula2* :: *Formula*
assumes *equivalentFormulae formula1 formula1′*
shows *equivalentFormulae* (*formula1* @ *formula2*) (*formula1′* @ *formula2*)

⟨*proof*⟩

lemma *satisfiableEquivalent*:
fixes *formula1* :: *Formula* **and** *formula2* :: *Formula*
assumes *equivalentFormulae formula1 formula2*
shows *satisfiable formula1* = *satisfiable formula2*
⟨*proof*⟩

lemma *satisfiableEquivalentAppend*:
fixes *formula1* :: *Formula* **and** *formula1'* :: *Formula* **and** *formula2* :: *Formula*
assumes *equivalentFormulae formula1 formula1'* **and** *satisfiable* (*formula1* @ *formula2*)
shows *satisfiable* (*formula1'* @ *formula2*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *clauseOrderIrrelevant*:
shows *equivalentFormulae* (*F1* @ *F* @ *F'* @ *F2*) (*F1* @ *F'* @ *F* @ *F2*)
⟨*proof*⟩

lemma *extendEquivalentFormulaWithEntailedClause*:
fixes *formula1* :: *Formula* **and** *formula2* :: *Formula* **and** *clause* :: *Clause*
assumes *equivalentFormulae formula1 formula2* **and** *formulaEntailsClause formula2 clause*
shows *equivalentFormulae formula1* (*formula2* @ [*clause*])
⟨*proof*⟩

lemma *entailsLiteralRelpacePartWithEquivalent*:
assumes *equivalentFormulae F F'* **and** *formulaEntailsLiteral* (*F1* @ *F* @ *F2*) *l*
shows *formulaEntailsLiteral* (*F1* @ *F'* @ *F2*) *l*
⟨*proof*⟩

2.2.12 Remove false and duplicate literals of a clause

definition
removeFalseLiterals :: *Clause* ⇒ *Valuation* ⇒ *Clause*
where

removeFalseLiterals clause valuation = *filter* (λ *l*. $\neg$ *literalFalse l valuation*) *clause*

lemma *clauseTrueRemoveFalseLiterals*:
assumes *consistent v*
shows *clauseTrue c v* = *clauseTrue* (*removeFalseLiterals c v*) *v*
⟨*proof*⟩

lemma *clauseTrueRemoveDuplicateLiterals*:
shows *clauseTrue c v* = *clauseTrue* (*remdups c*) *v*
⟨*proof*⟩

lemma *removeDuplicateLiteralsEquivalentClause*:
shows *equivalentFormulae* [*remdups clause*] [*clause*]
⟨*proof*⟩

lemma *falseLiteralsCanBeRemoved*:

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

lemma *falseAndDuplicateLiteralsCanBeRemoved*:

assumes *equivalentFormulae* (*F1* @ *val2form v* @ *F2*) *F'*
shows *equivalentFormulae* (*F1* @ *val2form v* @ [*remdups* (*removeFalseLiterals c v*)] @ *F2*) (*F'* @ [*c*])
(**is** *equivalentFormulae ?lhs ?rhs*)
⟨*proof*⟩

lemma *satisfiedClauseCanBeRemoved*:
assumes
equivalentFormulae (*F* @ *val2form v*) *F'*
clauseTrue c v
shows *equivalentFormulae* (*F* @ *val2form v*) (*F'* @ [*c*])
⟨*proof*⟩

lemma *formulaEntailsClauseRemoveEntailedLiteralOpposites*:
assumes
formulaEntailsClause F clause
formulaEntailsValuation F valuation
shows
formulaEntailsClause F (*list-diff clause* (*oppositeLiteralList valuation*))
⟨*proof*⟩

2.2.13 Resolution

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

lemma *resolventIsEntailed*:
fixes *clause1* :: *Clause* **and** *clause2* :: *Clause* **and** *literal* :: *Literal*
shows *formulaEntailsClause* [*clause1*, *clause2*] (*resolve clause1 clause2 literal*)
⟨*proof*⟩

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*⟩

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*⟩

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** *formulaEntailsClause formula uClause*
shows *formulaEntailsLiteral* (*formula* @ *val2form valuation*) *uLiteral*
⟨*proof*⟩

lemma *isUnitClauseRemoveAllUnitLiteralIsFalse*:
fixes *uClause* :: *Clause* **and** *uLiteral* :: *Literal* **and** *valuation* :: *Valuation*
assumes *isUnitClause uClause uLiteral valuation*
shows *clauseFalse* (*removeAll uLiteral uClause*) *valuation*
⟨*proof*⟩

lemma *isUnitClauseAppendValuation*:
assumes *isUnitClause uClause uLiteral valuation l* ≠ *uLiteral l* ≠ *opposite uLiteral*
shows *isUnitClause uClause uLiteral* (*valuation* @ [*l*])
⟨*proof*⟩

lemma *containsTrueNotUnit*:
assumes
l el c **and** *literalTrue l v* **and** *consistent v*
shows
¬ (∃ *ul*. *isUnitClause c ul v*)
⟨*proof*⟩

lemma *unitBecomesFalse*:
assumes
isUnitClause uClause uLiteral valuation
shows
clauseFalse uClause (*valuation* @ [*opposite uLiteral*])
⟨*proof*⟩

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* ⇒ *Literal* ⇒ *Valuation* ⇒ *bool*
where
(*isReason clause literal valuation*) ==
(*literal el clause*) ∧
(*clauseFalse* (*removeAll literal clause*) *valuation*) ∧
(∀ *literal′*. *literal′ el* (*removeAll literal clause*)
⟶ *precedes* (*opposite literal′*) *literal valuation* ∧ *opposite literal′* ≠ *literal*)

lemma *isReasonAppend*:
fixes *clause* :: *Clause* **and** *literal* :: *Literal* **and** *valuation* :: *Valuation* **and** *valuation′* :: *Valuation*
assumes *isReason clause literal valuation*
shows *isReason clause literal* (*valuation* @ *valuation′*)
⟨*proof*⟩

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*⟩

lemma *isReasonHoldsInPrefix*:
fixes *prefix* :: *Valuation* **and** *valuation* :: *Valuation* **and** *clause* :: *Clause* **and** *literal* :: *Literal*
assumes
literal el prefix **and**
isPrefix prefix valuation **and**
isReason clause literal valuation
shows
isReason clause literal prefix
⟨*proof*⟩

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*⟩

lemma *lastAssertedLiteralIsUniq*:
fixes *literal* :: *Literal* **and** *literal′* :: *Literal* **and** *literalList* :: *Literal list* **and** *valuation* :: *Valuation*
assumes
lastL: *isLastAssertedLiteral literal literalList valuation* **and**
lastL′: *isLastAssertedLiteral literal′ literalList valuation*
shows *literal* = *literal′*
⟨*proof*⟩

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*⟩

lemma *isLastAssertedLiteralSubset*:
assumes
isLastAssertedLiteral l c M
set c′ ⊆ *set c*
l el c′
shows
isLastAssertedLiteral l c′ M
⟨*proof*⟩

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*⟩

end

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*
⟨*proof*⟩

lemma *eitherMarkedOrNotMarkedElement*:
shows *a* = (*element a*, *True*) ∨ *a* = (*element a*, *False*)
⟨*proof*⟩

lemma *eitherMarkedOrNotMarked*:
assumes *e* ∈ *set* (*elements M*)
shows (*e*, *True*) ∈ *set M* ∨ (*e*, *False*) ∈ *set M*
⟨*proof*⟩

lemma *elementMemElements* [*simp*]:
assumes *x* ∈ *set M*
shows *element x* ∈ *set* (*elements M*)
⟨*proof*⟩

lemma *elementsAppend* [*simp*]:
shows *elements* (*a* @ *b*) = *elements a* @ *elements b*
⟨*proof*⟩

lemma *elementsEmptyIffTrailEmpty* [*simp*]:
shows (*elements list* = []) = (*list* = [])
⟨*proof*⟩

lemma *elementsButlastTrailIsButlastElementsTrail* [*simp*]:
shows *elements* (*butlast M*) = *butlast* (*elements M*)
⟨*proof*⟩

lemma *elementLastTrailIsLastElementsTrail* [*simp*]:
assumes *M* ≠ []
shows *element* (*last M*) = *last* (*elements M*)
⟨*proof*⟩

lemma *isPrefixElements*:
assumes *isPrefix a b*
shows *isPrefix* (*elements a*) (*elements b*)
⟨*proof*⟩

lemma *prefixElementsAreTrailElements*:
assumes
isPrefix p M
shows
set (*elements p*) ⊆ *set* (*elements M*)
⟨*proof*⟩

lemma *uniqElementsTrailImpliesUniqElementsPrefix*:
assumes
isPrefix p M **and** *uniq* (*elements M*)
shows
uniq (*elements p*)
⟨*proof*⟩

lemma [*simp*]:
assumes (*e*, *d*) ∈ *set M*
shows *e* ∈ *set* (*elements M*)
⟨*proof*⟩

lemma *uniqImpliesExclusiveTrueOrFalse*:
assumes
(*e*, *d*) ∈ *set M* **and** *uniq* (*elements M*)
shows
¬ (*e*, ¬ *d*) ∈ *set M*
⟨*proof*⟩

3.2 Marked trail elements

primrec
markedElements :: *'a Trail* ⇒ *'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*))
⟨*proof*⟩

lemma *markedElementIsMarkedTrue*:
shows (*m* ∈ *set* (*markedElements M*)) = ((*m*, *True*) ∈ *set M*)
⟨*proof*⟩

lemma *markedElementsAppend*:
shows *markedElements* (*M1* @ *M2*) = *markedElements M1* @ *markedElements M2*
⟨*proof*⟩

lemma *markedElementsAreElements*:
assumes $m \in set\ (markedElements\ M)$
shows $m \in set\ (elements\ M)$
⟨*proof*⟩

lemma *markedAndMemberImpliesIsMarkedElement*:
assumes *marked m m* ∈ *set M*
shows (*element m*) ∈ *set* (*markedElements M*)
⟨*proof*⟩

lemma *markedElementsPrefixAreMarkedElementsTrail*:
assumes *isPrefix p M m* ∈ *set* (*markedElements p*)
shows *m* ∈ *set* (*markedElements M*)
⟨*proof*⟩

lemma *markedElementsTrailMemPrefixAreMarkedElementsPrefix*:
assumes
uniq (*elements M*) **and**
isPrefix p M **and**
m ∈ *set* (*elements p*) **and**
m ∈ *set* (*markedElements M*)
shows
m ∈ *set* (*markedElements p*)
⟨*proof*⟩

3.3 Prefix before/upto a trail element

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

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

lemma *prefixBeforeElement e t* = *takeWhile* (λ *e'*. *element e'* ≠ *e*) *t*
⟨*proof*⟩

lemma *prefixBeforeElement e t* = *take* (*firstPos e* (*elements t*)) *t*

⟨*proof*⟩

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

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*
⟨*proof*⟩

lemma *isPrefixPrefixToElement*:
shows *isPrefix* (*prefixToElement e t*) *t*
⟨*proof*⟩

lemma *isPrefixPrefixBeforeElement*:
shows *isPrefix* (*prefixBeforeElement e t*) *t*
⟨*proof*⟩

lemma *prefixToElementContainsTrailElement*:
assumes *e* ∈ *set* (*elements M*)
shows *e* ∈ *set* (*elements* (*prefixToElement e M*))
⟨*proof*⟩

lemma *prefixBeforeElementDoesNotContainTrailElement*:
assumes *e* ∈ *set* (*elements M*)
shows *e* ∉ *set* (*elements* (*prefixBeforeElement e M*))
⟨*proof*⟩

lemma *prefixToElementAppend*:
shows *prefixToElement e* (*M1* @ *M2*) =
(*if e* ∈ *set* (*elements M1*) *then*
prefixToElement e M1
else
M1 @ *prefixToElement e M2*
)
⟨*proof*⟩

lemma *prefixToElementToPrefixElement*:
assumes

isPrefix p M **and** *e* ∈ *set* (*elements p*)
shows
prefixToElement e M = *prefixToElement e p*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *markedElementsToAreMarkedElements*:
assumes *m* ∈ *set* (*markedElementsTo e M*)
shows *m* ∈ *set* (*markedElements M*)
⟨*proof*⟩

lemma *markedElementsToNonMemberAreAllMarkedElements*:
assumes *e* ∉ *set* (*elements M*)
shows *markedElementsTo e M* = *markedElements M*
⟨*proof*⟩

lemma *markedElementsToAppend*:
shows *markedElementsTo e* (*M1* @ *M2*) =
(*if e* ∈ *set* (*elements M1*) *then*
markedElementsTo e M1
else
markedElements M1 @ *markedElementsTo e M2*
)
⟨*proof*⟩

lemma *markedElementsEmptyImpliesMarkedElementsToEmpty*:
assumes *markedElements M* = []
shows *markedElementsTo e M* = []
⟨*proof*⟩

lemma *markedElementIsMemberOfItsMarkedElementsTo*:
assumes
uniq (*elements M*) **and** *marked e* **and** *e* ∈ *set M*
shows
element e ∈ *set* (*markedElementsTo* (*element e*) *M*)
⟨*proof*⟩

lemma *markedElementsToPrefixElement*:
assumes *isPrefix p M* **and** *e* ∈ *set* (*elements p*)
shows *markedElementsTo e M* = *markedElementsTo e p*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *removeLastMarkedFromMarkedElementsToLastMarkedAreAll-MarkedElementsInPrefixLastMarked*:
assumes
markedElements M ≠ []
shows
removeAll (*lastMarked M*) (*markedElementsTo* (*lastMarked M*) *M*)
= *markedElements* (*prefixBeforeElement* (*lastMarked M*) *M*)
⟨*proof*⟩

lemma *markedElementsToLastMarkedAreAllMarkedElements*:
assumes
uniq (*elements M*) **and** *markedElements M* ≠ []
shows
markedElementsTo (*lastMarked M*) *M* = *markedElements M*
⟨*proof*⟩

lemma *lastTrailElementMarkedImpliesMarkedElementsToLastElementAre-AllMarkedElements*:
assumes
marked (*last M*) **and** *last* (*elements M*) ∉ *set* (*butlast* (*elements M*))
shows
markedElementsTo (*last* (*elements M*)) *M* = *markedElements M*
⟨*proof*⟩

lemma *lastMarkedIsMemberOfItsMarkedElementsTo*:
assumes
uniq (*elements M*) **and** *markedElements M* ≠ []
shows
lastMarked M ∈ *set* (*markedElementsTo* (*lastMarked M*) *M*)
⟨*proof*⟩

lemma *lastTrailElementNotMarkedImpliesMarkedElementsToLAreMarkedEle-*

mentsToLInButlastTrail:
assumes ¬ *marked* (*last M*)
shows *markedElementsTo e M* = *markedElementsTo e* (*butlast M*)
⟨*proof*⟩

3.6 Level of a trail element

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

definition
elementLevel :: *'a* ⇒ *'a Trail* ⇒ *nat*
where
elementLevel e t = *length* (*markedElementsTo e t*)

lemma *elementLevelMarkedGeq1*:
assumes
uniq (*elements M*) **and** *e* ∈ *set* (*markedElements M*)
shows
elementLevel e M >= *1*
⟨*proof*⟩

lemma *elementLevelAppend*:
assumes *a* ∈ *set* (*elements M*)
shows *elementLevel a M* = *elementLevel a* (*M* @ *M'*)
⟨*proof*⟩

lemma *elementLevelPrecedesLeq*:
assumes
precedes a b (*elements M*)
shows
elementLevel a M ≤ *elementLevel b M*
⟨*proof*⟩

lemma *elementLevelPrecedesMarkedElementLt*:
assumes
uniq (*elements M*) **and**
e ≠ *d* **and**
d ∈ *set* (*markedElements M*) **and**
precedes e d (*elements M*)
shows
elementLevel e M < *elementLevel d M*
⟨*proof*⟩

lemma *differentMarkedElementsHaveDifferentLevels*:
assumes
uniq (*elements M*) **and**
a ∈ *set* (*markedElements M*) **and**

$b \in set\ (markedElements\ M)$ **and**
$a \neq b$
shows $elementLevel\ a\ M \neq elementLevel\ b\ M$
⟨*proof*⟩

3.7 Current trail level

Current level is the number of marked elements in the trail

definition
$currentLevel :: 'a\ Trail \Rightarrow nat$
where
$currentLevel\ t = length\ (markedElements\ t)$

lemma *currentLevelNonMarked*:
shows $currentLevel\ M = currentLevel\ (M\ @\ [(l, False)])$
⟨*proof*⟩

lemma *currentLevelPrefix*:
assumes $isPrefix\ a\ b$
shows $currentLevel\ a <= currentLevel\ b$
⟨*proof*⟩

lemma *elementLevelLeqCurrentLevel*:
shows $elementLevel\ a\ M \leq currentLevel\ M$
⟨*proof*⟩

lemma *elementOnCurrentLevel*:
assumes $a \notin set\ (elements\ M)$
shows $elementLevel\ a\ (M\ @\ [(a, d)]) = currentLevel\ (M\ @\ [(a, d)])$
⟨*proof*⟩

3.8 Prefix to a given trail level

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

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

definition
$prefixToLevel :: nat \Rightarrow 'a\ Trail \Rightarrow 'a\ Trail$
where

prefixToLevel-def: (*prefixToLevel l t*) == (*prefixToLevel-aux t l 0*)

lemma *isPrefixPrefixToLevel-aux*:
shows ∃ *s*. *prefixToLevel-aux t l i* @ *s* = *t*
⟨*proof*⟩

lemma *isPrefixPrefixToLevel*:
shows (*isPrefix* (*prefixToLevel l t*) *t*)
⟨*proof*⟩

lemma *currentLevelPrefixToLevel-aux*:
assumes *l* ≥ *i*
shows *currentLevel* (*prefixToLevel-aux M l i*) <= *l* − *i*
⟨*proof*⟩

lemma *currentLevelPrefixToLevel*:
shows *currentLevel* (*prefixToLevel level M*) ≤ *level*
⟨*proof*⟩

lemma *currentLevelPrefixToLevelEq-aux*:
assumes *l* ≥ *i currentLevel M* >= *l* − *i*
shows *currentLevel* (*prefixToLevel-aux M l i*) = *l* − *i*
⟨*proof*⟩

lemma *currentLevelPrefixToLevelEq*:
assumes
level ≤ *currentLevel M*
shows
currentLevel (*prefixToLevel level M*) = *level*
⟨*proof*⟩

lemma *prefixToLevel-auxIncreaseAuxilaryCounter*:
assumes *k* ≥ *i*
shows *prefixToLevel-aux M l i* = *prefixToLevel-aux M* (*l* + (*k* − *i*))
k
⟨*proof*⟩

lemma *isPrefixPrefixToLevel-auxLowerLevel*:
assumes *i* ≤ *j*
shows *isPrefix* (*prefixToLevel-aux M i k*) (*prefixToLevel-aux M j k*)
⟨*proof*⟩

lemma *isPrefixPrefixToLevelLowerLevel*:
assumes *level* < *level*′
shows *isPrefix* (*prefixToLevel level M*) (*prefixToLevel level*′ *M*)
⟨*proof*⟩

lemma *prefixToLevel-auxPrefixToLevel-auxHigherLevel*:

assumes $i \leq j$
shows *prefixToLevel-aux a i k* = *prefixToLevel-aux* (*prefixToLevel-aux a j k*) *i k*
⟨*proof*⟩

lemma *prefixToLevelPrefixToLevelHigherLevel*:
assumes *level* ≤ *level′*
shows *prefixToLevel level M* = *prefixToLevel level* (*prefixToLevel level′ M*)
⟨*proof*⟩

lemma *prefixToLevelAppend-aux1*:
assumes
l ≥ *i* **and** *l* − *i* < *currentLevel a*
shows
prefixToLevel-aux (*a* @ *b*) *l i* = *prefixToLevel-aux a l i*
⟨*proof*⟩

lemma *prefixToLevelAppend-aux2*:
assumes
i ≤ *l* **and** *currentLevel a* + *i* ≤ *l*
shows *prefixToLevel-aux* (*a* @ *b*) *l i* = *a* @ *prefixToLevel-aux b l* (*i* + (*currentLevel a*))
⟨*proof*⟩

lemma *prefixToLevelAppend*:
shows *prefixToLevel level* (*a* @ *b*) =
(*if level* < *currentLevel a then*
prefixToLevel level a
else
a @ *prefixToLevel-aux b level* (*currentLevel a*)
)
⟨*proof*⟩

lemma *isProperPrefixPrefixToLevel*:
assumes *level* < *currentLevel t*
shows ∃ *s*. (*prefixToLevel level t*) @ *s* = *t* ∧ *s* ≠ [] ∧ (*marked* (*hd s*))
⟨*proof*⟩

lemma *prefixToLevelElementsElementLevel*:
assumes
e ∈ *set* (*elements* (*prefixToLevel level M*))
shows
elementLevel e M ≤ *level*
⟨*proof*⟩

lemma *elementLevelLtLevelImpliesMemberPrefixToLevel-aux*:

assumes
$e \in set(elements\ M)$ **and**
$elementLevel\ e\ M + i \leq level$ **and**
$i \leq level$
shows
$e \in set\ (elements\ (prefixToLevel\text{-}aux\ M\ level\ i))$
⟨*proof*⟩

lemma *elementLevelLtLevelImpliesMemberPrefixToLevel*:
assumes
$e \in set\ (elements\ M)$ **and**
$elementLevel\ e\ M \leq level$
shows
$e \in set\ (elements\ (prefixToLevel\ level\ M))$
⟨*proof*⟩

lemma *literalNotInEarlierLevelsThanItsLevel*:
assumes
$level < elementLevel\ e\ M$
shows
$e \notin set\ (elements\ (prefixToLevel\ level\ M))$
⟨*proof*⟩

lemma *elementLevelPrefixElement*:
assumes $e \in set\ (elements\ (prefixToLevel\ level\ M))$
shows $elementLevel\ e\ (prefixToLevel\ level\ M) = elementLevel\ e\ M$
⟨*proof*⟩

lemma *currentLevelZeroTrailEqualsItsPrefixToLevelZero*:
assumes $currentLevel\ M = 0$
shows $M = prefixToLevel\ 0\ M$
⟨*proof*⟩

3.9 Number of literals of every trail level

primrec
$levelsCounter\text{-}aux :: 'a\ Trail \Rightarrow nat\ list \Rightarrow nat\ list$
where
$levelsCounter\text{-}aux\ []\ l = l$
| $levelsCounter\text{-}aux\ (h\ \#\ t)\ l =$
(*if* $(marked\ h)$ *then*
$levelsCounter\text{-}aux\ t\ (l\ @\ [1])$
else
$levelsCounter\text{-}aux\ t\ (butlast\ l\ @\ [Suc\ (last\ l)])$
)

definition
$levelsCounter :: 'a\ Trail \Rightarrow nat\ list$
where

levelsCounter t = levelsCounter-aux t [0]

lemma *levelsCounter-aux-startIrellevant*:
∀ *y*. *y* ≠ [] ⟶ *levelsCounter-aux a* (*x* @ *y*) = (*x* @ *levelsCounter-aux a y*)
⟨*proof*⟩

lemma *levelsCounter-auxSuffixContinues*: ∀ *l*. *levelsCounter-aux* (*a* @ *b*) *l* = *levelsCounter-aux b* (*levelsCounter-aux a l*)
⟨*proof*⟩

lemma *levelsCounter-auxNotEmpty*: ∀ *l*. *l* ≠ [] ⟶ *levelsCounter-aux a l* ≠ []
⟨*proof*⟩

lemma *levelsCounter-auxIncreasesFirst*:
∀ *m n l1 l2*. *levelsCounter-aux a* (*m* # *l1*) = *n* # *l2* ⟶ *m* <= *n*
⟨*proof*⟩

lemma *levelsCounterPrefix*:
assumes (*isPrefix p a*)
shows *?rest*. *rest* ≠ [] ∧ *levelsCounter a* = *butlast* (*levelsCounter p*) @ *rest* ∧ *last* (*levelsCounter p*) ≤ *hd rest*
⟨*proof*⟩

lemma *levelsCounterPrefixToLevel*:
assumes *p* = *prefixToLevel level a level* ≥ *0 level* < *currentLevel a*
shows *?rest* . *rest* ≠ [] ∧ (*levelsCounter a*) = (*levelsCounter p*) @ *rest*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *isPrefixPrefixBeforeLastMarked*:
shows *isPrefix* (*prefixBeforeLastMarked M*) *M*

⟨*proof*⟩

lemma *lastMarkedNotInPrefixBeforeLastMarked*:
assumes *uniq* (*elements M*) **and** *markedElements M* ≠ []
shows ¬ (*lastMarked M*) ∈ *set* (*elements* (*prefixBeforeLastMarked M*))
⟨*proof*⟩

lemma *uniqImpliesPrefixBeforeLastMarkedIsPrefixBeforeLastMarked*:
assumes *markedElements M* ≠ [] **and** (*lastMarked M*) ∉ *set* (*elements M*)
shows *prefixBeforeLastMarked M* = *prefixBeforeElement* (*lastMarked M*) *M*
⟨*proof*⟩

lemma *markedElementsAreElementsBeforeLastDecisionAndLastDecision*:
assumes *markedElements M* ≠ []
shows (*markedElements M*) = (*markedElements* (*prefixBeforeLastMarked M*)) @ [*lastMarked M*]
⟨*proof*⟩

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 (*F0*::*Formula*) (*F*::*Formula*) == *equivalentFormulae F0 F*

definition
InvariantVarsM (*M*::*LiteralTrail*) (*F0*::*Formula*) (*Vbl*::*Variable set*) == *vars* (*elements M*) ⊆ *vars F0* ∪ *Vbl*

definition
InvariantVarsF (*F*::*Formula*) (*F0*::*Formula*) (*Vbl*::*Variable set*) == *vars F* ⊆ *vars F0* ∪ *Vbl*

The following invariants are used in conflict analysis.

definition
InvariantCFalse (*conflictFlag*::*bool*) (*M*::*LiteralTrail*) (*C*::*Clause*) == *conflictFlag* ⟶ *clauseFalse C* (*elements M*)

definition

InvariantCEntailed (*conflictFlag*::*bool*) (*F*::*Formula*) (*C*::*Clause*) == *conflictFlag* ⟶ *formulaEntailsClause F C*

definition
InvariantReasonClauses (*F*::*Formula*) (*M*::*LiteralTrail*) ==
∀ *literal*. *literal el* (*elements M*) ∧ ¬ *literal el decisions M* ⟶
(∃ *clause*. *formulaEntailsClause F clause* ∧ *isReason clause literal* (*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* :: *LiteralTrail* **and** *F* :: *Formula*
assumes ∀ *l*. *l el elements M* ⟶ *formulaEntailsLiteral* (*F* @ *val2form* (*decisionsTo l M*)) *l*
shows ∀ *l*. *l el elements M* ⟶ *formulaEntailsLiteral* (*F* @ *val2form* (*decisions M*)) *l*
⟨*proof*⟩

lemma *InvariantImpliedLiteralsAndElementsEntailLiteralThenDecisionsEntailLiteral*:
fixes *M* :: *LiteralTrail* **and** *F* :: *Formula* **and** *literal* :: *Literal*
assumes *InvariantImpliedLiterals F M* **and** *formulaEntailsLiteral* (*F* @ (*val2form* (*elements M*))) *literal*
shows *formulaEntailsLiteral* (*F* @ *val2form* (*decisions M*)) *literal*
⟨*proof*⟩

lemma *InvariantImpliedLiteralsAndFormulaFalseThenFormulaAndDecisionsAreNotSatisfiable*:
fixes *M* :: *LiteralTrail* **and** *F* :: *Formula*
assumes *InvariantImpliedLiterals F M* **and** *formulaFalse F* (*elements M*)
shows ¬ *satisfiable* (*F* @ *val2form* (*decisions M*))
⟨*proof*⟩

lemma *InvariantImpliedLiteralsHoldsForPrefix*:
fixes *M* :: *LiteralTrail* **and** *prefix* :: *LiteralTrail* **and** *F* :: *Formula*
assumes *InvariantImpliedLiterals F M* **and** *isPrefix prefix M*
shows *InvariantImpliedLiterals F prefix*
⟨*proof*⟩

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*⟩

lemma *InvariantReasonClausesHoldsForPrefixElements*:
fixes *F*::*Formula* **and** *M*::*LiteralTrail* **and** *p*::*LiteralTrail*
assumes *InvariantReasonClauses F p* **and**
isPrefix p M **and**
literal el (*elements p*) **and** ¬ *literal el decisions M*
shows ∃ *clause*. *formulaEntailsClause F clause* ∧ *isReason clause literal* (*elements M*)
⟨*proof*⟩

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* :: *LiteralTrail* **and** *literal* :: *Literal* **and** *M'* :: *LiteralTrail*
assumes *InvariantUniq M* **and**
var literal ∉ *vars* (*elements M*) **and**
M' = *M* @ [(*literal*, *True*)]
shows *InvariantUniq M'*
⟨*proof*⟩

lemma *InvariantImpliedLiteralsAfterDecide*:
fixes *F* :: *Formula* **and** *M* :: *LiteralTrail* **and** *literal* :: *Literal* **and** *M'* :: *LiteralTrail*
assumes *InvariantImpliedLiterals F M* **and**
var literal ∉ *vars* (*elements M*) **and**
M' = *M* @ [(*literal*, *True*)]
shows *InvariantImpliedLiterals F M'*
⟨*proof*⟩

lemma *InvariantVarsMAfterDecide*:
fixes *F* :: *Formula* **and** *F0* :: *Formula* **and** *M* :: *LiteralTrail* **and** *literal* :: *Literal* **and** *M'* :: *LiteralTrail*
assumes *InvariantVarsM M F0 Vbl* **and**
var literal ∈ *Vbl* **and**
M' = *M* @ [(*literal*, *True*)]
shows *InvariantVarsM M' F0 Vbl*
⟨*proof*⟩

lemma *InvariantConsistentAfterDecide*:
fixes *M* :: *LiteralTrail* **and** *literal* :: *Literal* **and** *M'* :: *LiteralTrail*

assumes *InvariantConsistent M* **and**
var literal $\notin$ *vars* (*elements M*) **and**
$M' = M$ @ [(*literal*, *True*)]
shows *InvariantConsistent M'*
⟨*proof*⟩

lemma *InvariantReasonClausesAfterDecide*:
fixes *F* :: *Formula* **and** *M* :: *LiteralTrail* **and** *M'* :: *LiteralTrail*
assumes *InvariantReasonClauses F M* **and** *InvariantUniq M* **and**
$M' = M$ @ [(*literal*, *True*)]
shows *InvariantReasonClauses F M'*
⟨*proof*⟩

lemma *InvariantCFalseAfterDecide*:
fixes *conflictFlag*::*bool* **and** *M*::*LiteralTrail* **and** *C*::*Clause*
assumes *InvariantCFalse conflictFlag M C* **and** $M' = M$ @ [(*literal*, *True*)]
shows *InvariantCFalse conflictFlag M' C*
⟨*proof*⟩

UnitPropagate transition rule.

lemma *InvariantImpliedLiteralsHoldsForUnitLiteral*:
fixes *M* :: *LiteralTrail* **and** *F* :: *Formula* **and** *uClause* :: *Clause* **and** *uLiteral* :: *Literal*
assumes *InvariantImpliedLiterals F M* **and**
formulaEntailsClause F uClause **and** *isUnitClause uClause uLiteral* (*elements M*) **and**
$M' = M$ @ [(*uLiteral*, *False*)]
shows *formulaEntailsLiteral* (*F* @ *val2form* (*decisionsTo uLiteral M'*)) *uLiteral*
⟨*proof*⟩

lemma *InvariantImpliedLiteralsAfterUnitPropagate*:
fixes *M* :: *LiteralTrail* **and** *F* :: *Formula* **and** *uClause* :: *Clause* **and** *uLiteral* :: *Literal*
assumes *InvariantImpliedLiterals F M* **and**
formulaEntailsClause F uClause **and** *isUnitClause uClause uLiteral* (*elements M*) **and**
$M' = M$ @ [(*uLiteral*, *False*)]
shows *InvariantImpliedLiterals F M'*
⟨*proof*⟩

lemma *InvariantVarsMAfterUnitPropagate*:
fixes *F* :: *Formula* **and** *F0* :: *Formula* **and** *M* :: *LiteralTrail* **and** *uClause* :: *Clause* **and** *uLiteral* :: *Literal* **and** *M'* :: *LiteralTrail*
assumes *InvariantVarsM M F0 Vbl* **and**
var uLiteral $\in$ *vars F0* $\cup$ *Vbl* **and**
$M' = M$ @ [(*uLiteral*, *False*)]
shows *InvariantVarsM M' F0 Vbl*

⟨*proof*⟩

lemma *InvariantConsistentAfterUnitPropagate*:
fixes M :: *LiteralTrail* **and** F :: *Formula* **and** M' :: *LiteralTrail* **and** *uClause* :: *Clause* **and** *uLiteral* :: *Literal*
assumes *InvariantConsistent* M **and**
isUnitClause uClause uLiteral (*elements* M) **and**
$M' = M$ @ [(*uLiteral*, *False*)]
shows *InvariantConsistent* M'
⟨*proof*⟩

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*⟩

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*⟩

lemma *InvariantCFalseAfterUnitPropagate*:
fixes M :: *LiteralTrail* **and** F :: *Formula* **and** M' :: *LiteralTrail* **and** *uClause* :: *Clause* **and** *uLiteral* :: *Literal*
assumes *InvariantCFalse conflictFlag* M C **and**
$M' = M$ @ [(*uLiteral*, *False*)]
shows *InvariantCFalse conflictFlag* M' C
⟨*proof*⟩

Backtrack transition rule.

lemma *InvariantImpliedLiteralsAfterBacktrack*:
fixes F::*Formula* **and** M::*LiteralTrail*
assumes *InvariantImpliedLiterals* F M **and** *InvariantUniq* M **and** *InvariantConsistent* M **and**
decisions $M \neq []$ **and** *formulaFalse* F (*elements* M)
$M' =$ (*prefixBeforeLastDecision* M) @ [(*opposite* (*lastDecision* M), *False*)]
shows *InvariantImpliedLiterals* F M'
⟨*proof*⟩

lemma *InvariantConsistentAfterBacktrack*:
fixes *F*::*Formula* **and** *M*::*LiteralTrail*
assumes *InvariantUniq M* **and** *InvariantConsistent M* **and**
decisions M ≠ [] **and**
M' = (*prefixBeforeLastDecision M*) @ [(*opposite* (*lastDecision M*), *False*)]
shows *InvariantConsistent M'*
⟨*proof*⟩

lemma *InvariantUniqAfterBacktrack*:
fixes *F*::*Formula* **and** *M*::*LiteralTrail*
assumes *InvariantUniq M* **and** *InvariantConsistent M* **and**
decisions M ≠ [] **and**
M' = (*prefixBeforeLastDecision M*) @ [(*opposite* (*lastDecision M*), *False*)]
shows *InvariantUniq M'*
⟨*proof*⟩

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*⟩

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* (*elements p*) **and**
M' = *p* @ [(*bLiteral*, *False*)]
shows *InvariantImpliedLiterals F M'*
⟨*proof*⟩

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*⟩

lemma *InvariantConsistentAfterBackjump*:
fixes *F*::*Formula* **and** *M*::*LiteralTrail* **and** *p*::*LiteralTrail* **and** *bClause*::*Clause*

and *bLiteral*::*Literal*
assumes *InvariantConsistent M* **and**
isPrefix p M **and** *isUnitClause bClause bLiteral* (*elements p*) **and**
$M' = p$ @ [(*bLiteral*, *False*)]
shows *InvariantConsistent M'*
⟨*proof*⟩

lemma *InvariantUniqAfterBackjump*:
fixes *F*::*Formula* **and** *M*::*LiteralTrail* **and** *p*::*LiteralTrail* **and** *bClause*::*Clause* **and** *bLiteral*::*Literal*
assumes *InvariantUniq M* **and**
isPrefix p M **and** *isUnitClause bClause bLiteral* (*elements p*) **and**
$M' = p$ @ [(*bLiteral*, *False*)]
shows *InvariantUniq M'*
⟨*proof*⟩

lemma *InvariantReasonClausesAfterBackjump*:
fixes *F*::*Formula* **and** *M*::*LiteralTrail* **and** *p*::*LiteralTrail* **and** *bClause*::*Clause* **and** *bLiteral*::*Literal*
assumes *InvariantReasonClauses F M* **and** *InvariantUniq M* **and**
isPrefix p M **and** *isUnitClause bClause bLiteral* (*elements p*) **and** *formulaEntailsClause F bClause* **and**
$M' = p$ @ [(*bLiteral*, *False*)]
shows *InvariantReasonClauses F M'*
⟨*proof*⟩

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*]
shows *InvariantImpliedLiterals F' M*
⟨*proof*⟩

lemma *InvariantReasonClausesAfterLearn*:
fixes *F* :: *Formula* **and** *F'* :: *Formula* **and** *M* :: *LiteralTrail* **and** *C* :: *Clause*
assumes *InvariantReasonClauses F M* **and**
formulaEntailsClause F C **and**
$F' = F$ @ [*C*]
shows *InvariantReasonClauses F' M*
⟨*proof*⟩

lemma *InvariantVarsFAfterLearn*:
fixes *F0* :: *Formula* **and** *F* :: *Formula* **and** *F'* :: *Formula* **and** *C* :: *Clause*
assumes *InvariantVarsF F F0 Vbl* **and**

vars C $\subseteq$ (*vars F0*) $\cup$ *Vbl* **and**
F' = *F* @ [*C*]
shows *InvariantVarsF F' F0 Vbl*
⟨*proof*⟩

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*⟩

lemma *InvariantCEntailedAfterLearn*:
fixes *F0* :: *Formula* **and** *F* :: *Formula* **and** *F'* :: *Formula* **and** *C* :: *Clause*
assumes *InvariantCEntailed conflictFlag F C* **and**
F' = *F* @ [*C*]
shows *InvariantCEntailed conflictFlag F' C*
⟨*proof*⟩

Explain transition rule.

lemma *InvariantCFalseAfterExplain*:
fixes *conflictFlag*::*bool* **and** *M*::*LiteralTrail* **and** *C*::*Clause* **and** *literal* :: *Literal*
assumes *InvariantCFalse conflictFlag M C* **and**
opposite literal el C **and** *isReason reason literal* (*elements M*) **and**
C' = *resolve C reason* (*opposite literal*)
shows *InvariantCFalse conflictFlag M C'*
⟨*proof*⟩

lemma *InvariantCEntailedAfterExplain*:
fixes *conflictFlag*::*bool* **and** *M*::*LiteralTrail* **and** *C*::*Clause* **and** *literal* :: *Literal* **and** *reason* :: *Clause*
assumes *InvariantCEntailed conflictFlag F C* **and**
formulaEntailsClause F reason **and** *C'* = (*resolve C reason* (*opposite l*))
shows *InvariantCEntailed conflictFlag F C'*
⟨*proof*⟩

Conflict transition rule.

lemma *invariantCFalseAfterConflict*:
fixes *conflictFlag*::*bool* **and** *conflictFlag'*::*bool* **and** *M*::*LiteralTrail* **and** *F* :: *Formula* **and** *clause* :: *Clause* **and** *C'* :: *Clause*
assumes *conflictFlag* = *False* **and**
formulaFalse F (*elements M*) **and** *clause el F clauseFalse clause* (*elements M*) **and**

C′ = *clause* **and** *conflictFlag′* = *True*
shows *InvariantCFalse conflictFlag′ M C′*
⟨*proof*⟩

lemma *invariantCEntailedAfterConflict*:
fixes *conflictFlag*::*bool* **and** *conflictFlag′*::*bool* **and** *M*::*LiteralTrail* **and** *F* :: *Formula* **and** *clause* :: *Clause* **and** *C′* :: *Clause*
assumes *conflictFlag* = *False* **and**
formulaFalse F (*elements M*) **and** *clause el F* **and** *clauseFalse clause* (*elements M*) **and**
C′ = *clause* **and** *conflictFlag′* = *True*
shows *InvariantCEntailed conflictFlag′ F C′*
⟨*proof*⟩

UNSAT report

lemma *unsatReport*:
fixes *F* :: *Formula* **and** *M* :: *LiteralTrail* **and** *F0* :: *Formula*
assumes *InvariantImpliedLiterals F M* **and** *InvariantEquivalent F0 F* **and**
decisions M = [] **and** *formulaFalse F* (*elements M*)
shows ¬ *satisfiable F0*
⟨*proof*⟩

lemma *unsatReportExtensiveExplain*:
fixes *F* :: *Formula* **and** *M* :: *LiteralTrail* **and** *F0* :: *Formula* **and** *C* :: *Clause* **and** *conflictFlag* :: *bool*
assumes *InvariantEquivalent F0 F* **and** *InvariantCEntailed conflictFlag F C* **and**
conflictFlag **and** *C* = []
shows ¬ *satisfiable F0*
⟨*proof*⟩

SAT Report

lemma *satReport*:
fixes *F0* :: *Formula* **and** *F* :: *Formula* **and** *M*::*LiteralTrail*
assumes *vars F0* ⊆ *Vbl* **and** *InvariantVarsF F F0 Vbl* **and** *InvariantConsistent M* **and** *InvariantEquivalent F0 F* **and**
¬ *formulaFalse F* (*elements M*) **and** *vars* (*elements M*) ⊇ *Vbl*
shows *model* (*elements M*) *F0*
⟨*proof*⟩

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 stricly lower then the level of its last asserted literal

definition
isUIP l c M ==
isLastAssertedLiteral (*opposite l*) (*oppositeLiteralList c*)(*elements M*)
∧
(∀ *l′*. *l′ el c* ∧ *l′* ≠ *l* ⟶ *elementLevel* (*opposite l′*) *M* < *elementLevel* (*opposite l*) *M*)

Backjump level is a nonegative integer such that it is stricly lower than the level of the last asserted literal of a clause, and greater or equal then levels of all its other literals.

definition
isBackjumpLevel level l c M ==
isLastAssertedLiteral (*opposite l*) (*oppositeLiteralList c*)(*elements M*)
∧
0 ≤ *level* ∧ *level* < *elementLevel* (*opposite l*) *M* ∧
(∀ *l′*. *l′ el c* ∧ *l′* ≠ *l* ⟶ *elementLevel* (*opposite l′*) *M* ≤ *level*)

lemma *lastAssertedLiteralHasHighestElementLevel*:
fixes *literal* :: *Literal* **and** *clause* :: *Clause* **and** *M* :: *LiteralTrail*
assumes *isLastAssertedLiteral literal clause* (*elements M*) **and** *uniq* (*elements M*)
shows ∀ *l′*. *l′ el clause* ∧ *l′ el elements M* ⟶ *elementLevel l′ M* <= *elementLevel literal M*
⟨*proof*⟩

When backjump clause contains only a single literal, then the backjump level is 0.

lemma *backjumpLevelZero*:
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*⟩

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*:
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*⟩

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*)
⟨*proof*⟩

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

lemma *isBackjumpLevelEnsuresIsUnitInPrefix*:
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*) **and** *isBackjumpLevel level l c M*
shows *isUnitClause c l* (*elements* (*prefixToLevel level M*))
⟨*proof*⟩

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
isMinimalBackjumpLevel level l c M ==
isBackjumpLevel level l c M ∧
(*if set c* ≠ {*l*} *then*
(∃ *ll*. *ll el c* ∧ *elementLevel* (*opposite ll*) *M* = *level*)
else
level = *0*
)

lemma *isMinimalBackjumpLevelCharacterization*:
assumes
isUIP l c M
clauseFalse c (*elements M*)
uniq (*elements M*)
shows
isMinimalBackjumpLevel level l c M =
(*isBackjumpLevel level l c M* ∧
(∀ *level′*. *level′* < *level* ⟶ ¬ *isBackjumpLevel level′ l c M*)) (**is** *?lhs* = *?rhs*)

⟨*proof*⟩

lemma *isMinimalBackjumpLevelEnsuresIsNotUnitBeforePrefix*:
fixes M :: *LiteralTrail* **and** *conflictFlag* :: *bool* **and** c :: *Clause* **and** l :: *Literal*
assumes *consistent* (*elements* M) **and** *uniq* (*elements* M) **and**
clauseFalse c (*elements* M) *isMinimalBackjumpLevel level* l c M **and**
$level' < level$
shows ¬ (∃ l'. *isUnitClause* c l' (*elements* (*prefixToLevel level'* M)))
⟨*proof*⟩

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

lemma *allDecisionsThenUIP*:
fixes M :: *LiteralTrail* **and** c:: *Clause*
assumes (*uniq* (*elements* M)) **and**
∀ l'. l' *el* c ⟶ (*opposite* l') *el* (*decisions* M)
isLastAssertedLiteral (*opposite* l) (*oppositeLiteralList* c) (*elements* M)
shows *isUIP* l c M
⟨*proof*⟩

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

lemma *lastDecisionThenUIP*:
fixes M :: *LiteralTrail* **and** c:: *Clause*
assumes (*uniq* (*elements* M)) **and**
(*opposite* l) *el* (*decisions* M)
clauseFalse c (*elements* M)
isLastAssertedLiteral (*opposite* l) (*oppositeLiteralList* c) (*elements* M)
shows *isUIP* l c M
⟨*proof*⟩

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

lemma *allDecisionsThenExistsBackjumpLevel*:
fixes M :: *LiteralTrail* **and** c:: *Clause*
assumes (*uniq* (*elements* M)) **and**
∀ l'. l' *el* c ⟶ (*opposite* l') *el* (*decisions* M)
isLastAssertedLiteral (*opposite* l) (*oppositeLiteralList* c) (*elements* M)
shows ∃ *level*. (*isBackjumpLevel level* l c M)
⟨*proof*⟩

Explain is applicable to each non-decision literal in a clause.

lemma *explainApplicableToEachNonDecision*:
fixes F :: *Formula* **and** M :: *LiteralTrail* **and** *conflictFlag* :: *bool* **and** C :: *Clause* **and** *literal* :: *Literal*

assumes *InvariantReasonClauses F M* **and** *InvariantCFalse conflictFlag M C* **and**
conflictFlag = *True* **and** *opposite literal el C* **and** ¬ *literal el* (*decisions M*)
shows ∃ *clause*. *formulaEntailsClause F clause* ∧ *isReason clause literal* (*elements M*)
⟨*proof*⟩

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*⟩

4.4.1 Trail ordering

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

definition
decisionLess = {(*l1*::('*a*∗*bool*), *l2*::('*a*∗*bool*)). *isDecision l1* ∧ ¬ *isDecision l2*}
definition
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*⟩

lemma *lexLessBackjump*:
assumes *p* = *prefixToLevel level a* **and** *level* >= *0* **and** *level* < *currentLevel a*
shows (*p* @ [(*x*, *False*)], *a*) ∈ *lexLess*
⟨*proof*⟩

lemma *lexLessBacktrack*:
assumes p = *prefixBeforeLastDecision a decisions a* $\neq$ []
shows (p @ [(x, *False*)], a) $\in$ *lexLess*
⟨*proof*⟩

The following several lemmas prover that *lexLess* is acyclic. This property will play an important role in building a well-founded ordering based on *lexLess*.

lemma *transDecisionLess*:
shows *trans decisionLess*
⟨*proof*⟩

lemma *translexLess*:
shows *trans lexLess*
⟨*proof*⟩

lemma *irreflexiveDecisionLess*:
shows (x, x) $\notin$ *decisionLess*
⟨*proof*⟩

lemma *irreflexiveLexLess*:
shows (x, x) $\notin$ *lexLess*
⟨*proof*⟩

lemma *acyclicLexLess*:
shows *acyclic lexLess*
⟨*proof*⟩

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

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

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

lemma *finiteVarsClause*:
fixes c :: *Clause*
shows *finite* (*vars* c)
⟨*proof*⟩

lemma *finiteVarsFormula*:
fixes F :: *Formula*

shows *finite* (*vars F*)
⟨*proof*⟩

lemma *finiteListDecompose*:
shows *finite* {(*a*, *b*). *l* = *a* @ *b*}
⟨*proof*⟩

lemma *finiteListDecomposeSet*:
fixes *L* :: *'a list set*
assumes *finite L*
shows *finite* {(*a*, *b*). ∃ *l*. *l* ∈ *L* ∧ *l* = *a* @ *b*}
⟨*proof*⟩

lemma *finiteUniqAndConsistentTrailsWithGivenVariableSet*:
fixes *V* :: *Variable set*
assumes *finite V*
shows *finite* {(*M*::*LiteralTrail*). *vars* (*elements M*) = *V* ∧ *uniq* (*elements M*) ∧ *consistent* (*elements M*)}
(**is** *finite* (*?trails V*))
⟨*proof*⟩

lemma *finiteUniqAndConsistentTrailsWithGivenVariableSuperset*:
fixes *V* :: *Variable set*
assumes *finite V*
shows *finite* {(*M*::*LiteralTrail*). *vars* (*elements M*) ⊆ *V* ∧ *uniq* (*elements M*) ∧ *consistent* (*elements M*)} (**is** *finite* (*?trails V*))
⟨*proof*⟩

Since the restricted ordering is acyclic and its domain is finite, it has to be well-founded.

lemma *wfLexLessRestricted*:
assumes *finite Vbl*
shows *wf* (*lexLessRestricted Vbl*)
⟨*proof*⟩

lexLessRestricted is also transitive.

lemma *transLexLessRestricted*:
shows *trans* (*lexLessRestricted Vbl*)
⟨*proof*⟩

4.4.2 Conflict clause ordering

The ordering of conflict clauses is the multiset ordering induced by the ordering of elements in the trail. Since, resolution operator is defined so that it removes all occurrences of clashing literal, it is also neccessary to remove duplicate literals before comparison.

definition

multLess M = inv-image (mult (precedesOrder (elements M))) (λ x. mset (remdups (oppositeLiteralList x)))

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

lemma *multLessResolve*:
assumes
opposite l el C **and**
isReason reason l (elements M)
shows
(resolve C reason (opposite l), C) ∈ multLess M
⟨*proof*⟩

lemma *multLessListDiff*:
assumes
(a, b) ∈ multLess M
shows
(list-diff a x, b) ∈ multLess M
⟨*proof*⟩

lemma *multLessRemdups*:
assumes
(a, b) ∈ multLess M
shows
(remdups a, remdups b) ∈ multLess M ∧
(remdups a, b) ∈ multLess M ∧
(a, remdups b) ∈ multLess M
⟨*proof*⟩

Now we show that *multLess* is well-founded.

lemma *wfMultLess*:
shows *wf (multLess M)*
⟨*proof*⟩

4.4.3 ConflictFlag ordering

A trivial ordering on Booleans. It will be used for the *Conflict* transition rule.

definition
boolLess = {(True, False)}

We show that it is well-founded

lemma *transBoolLess*:
shows *trans boolLess*
⟨*proof*⟩

lemma *wfBoolLess*:

shows *wf boolLess*
⟨*proof*⟩

4.4.4 Formulae ordering

A partial ordering of formulae, based on a membersip of a single fixed clause. This ordering will be used for the *Learn* transtion rule.

definition *learnLess* (*C*::*Clause*) == {((*F1*::*Formula*), (*F2*::*Formula*)). *C el F1* ∧ ¬ *C el F2*}

We show that it is well founded

lemma *wfLearnLess*:
fixes *C*::*Clause*
shows *wf* (*learnLess C*)
⟨*proof*⟩

4.4.5 Properties of well-founded relations.

lemma *wellFoundedEmbed*:
fixes *rel* :: ('*a* × '*a*) *set* **and** *rel'* :: ('*a* × '*a*) *set*
assumes ∀ *x y*. (*x*, *y*) ∈ *rel* ⟶ (*x*, *y*) ∈ *rel'* **and** *wf rel'*
shows *wf rel*
⟨*proof*⟩

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 :: *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
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*)) ∧

  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* ⇒ *Formula* ⇒ *bool*
where
appliedUnitPropagate stateA stateB F0 ==
 ∃ (*uc*::*Clause*) (*ul*::*Literal*).
  uc el F0 ∧
  isUnitClause uc ul (*elements* (*getM stateA*)) ∧

  getM stateB = *getM stateA* @ [(*ul*, *False*)]

definition
applicableUnitPropagate :: *State* ⇒ *Formula* ⇒ *bool*
where
applicableUnitPropagate state F0 == ∃ *state′*. *appliedUnitPropagate state state′ F0*

definition
appliedBacktrack :: *State* ⇒ *State* ⇒ *Formula* ⇒ *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 preformed 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 state′ F0 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*⟩

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*⟩

lemma *applicableBacktrackCharacterization*:
fixes *stateA*::*State*
shows *applicableBacktrack stateA F0* =
(*formulaFalse F0* (*elements* (*getM stateA*)) ∧
decisions (*getM stateA*) ≠ []) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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* **and** *state* :: *State*
assumes
isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation F0 decisionVars*

formulaFalse F0 (*elements* (*getM state*))
decisions (*getM state*) = []
shows ¬ *satisfiable F0*

⟨*proof*⟩

theorem *soundnessForSAT*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
vars F0 ⊆ *decisionVars* **and**

isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation F0 decisionVars*

¬ *formulaFalse F0* (*elements* (*getM state*))
vars (*elements* (*getM state*)) ⊇ *decisionVars*

shows
model (*elements* (*getM state*)) *F0*

⟨*proof*⟩

5.4 Termination

We now define a termination ordering on the set of states based on the *lexLessRestricted* trail ordering. This ordering will be central in termination proof.

definition *terminationLess* (*F0*::*Formula*) *decisionVars* == {((*stateA*::*State*), (*stateB*::*State*)).
(*getM stateA*, *getM stateB*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)}

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

lemma *trailIsDecreasedByDeciedUnitPropagateAndBacktrack*:
fixes *stateA*::*State* **and** *stateB*::*State*
assumes *invariantsHoldInState stateA F0 decisionVars* **and**
appliedDecide stateA stateB decisionVars ∨ *appliedUnitPropagate stateA stateB F0* ∨ *appliedBacktrack stateA stateB F0*
shows (*getM stateB*, *getM stateA*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)
⟨*proof*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

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

theorem *wfTransitionRelation*:
fixes *decisionVars* :: *Variable set* **and** *F0* :: *Formula* **and** *state0* :: *State*
assumes *finite decisionVars* **and** *isInitialState state0 F0*
shows *wf* {(*stateB*, *stateA*).
(*state0*, *stateA*) ∈ *transitionRelation F0 decisionVars* ∧ (*transition stateA stateB F0 decisionVars*)}

⟨*proof*⟩

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
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*⟩

Now we prove the final strong termination result which states that there cannot be infinite chains of transitions. If there is an infinite transition chain that starts from an initial state, its elements would for a set that would contain initial state and for every element of that set there would be another element of that set that is directly reachable from it. We show that no such set exists.

corollary *noInfiniteTransitionChains*:
fixes *F0*::*Formula* **and** *decisionVars*:: *Variable set*
assumes *finite decisionVars*
shows ¬ (∃ *Q*::(*State set*). ∃ *state0* ∈ *Q*. *isInitialState state0 F0* ∧

(∀ *state* ∈ *Q*. (∃ *state′* ∈ *Q*. *transition state state′ F0 decisionVars*))
)
⟨*proof*⟩

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
¬ *applicableDecide state decisionVars*
shows *vars* (*elements* (*getM state*)) ⊇ *decisionVars*
⟨*proof*⟩

lemma *finalConflictingState*:
fixes *state* :: *State*
assumes
¬ *applicableBacktrack state F0* **and**
formulaFalse F0 (*elements* (*getM state*))
shows
decisions (*getM state*) = []
⟨*proof*⟩

lemma *finalStateCharacterizationLemma*:
fixes *state* :: *State*
assumes
¬ *applicableDecide state decisionVars* **and**
¬ *applicableBacktrack state F0*
shows
(¬ *formulaFalse F0* (*elements* (*getM state*)) ∧ *vars* (*elements* (*getM state*)) ⊇ *decisionVars*) ∨

(*formulaFalse F0* (*elements* (*getM state*)) ∧ *decisions* (*getM state*) = [])
⟨*proof*⟩

theorem *finalStateCharacterization*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation F0 decisionVars* **and**
isFinalState state F0 decisionVars
shows
(¬ *formulaFalse F0* (*elements* (*getM state*)) ∧ *vars* (*elements* (*getM state*)) ⊇ *decisionVars*) ∨
(*formulaFalse F0* (*elements* (*getM state*)) ∧ *decisions* (*getM state*) = [])

⟨*proof*⟩

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 ¬ *formulaFalse F0* (*elements* (*getM state*)) ∧ *vars* (*elements* (*getM state*)) ⊇ *decisionVars*

⟨*proof*⟩

theorem *completenessForUNSAT*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
vars F0 ⊆ *decisionVars* **and**
¬ *satisfiable F0* **and**

isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation F0 decisionVars* **and**
isFinalState state F0 decisionVars

shows
formulaFalse F0 (*elements* (*getM state*)) ∧ *decisions* (*getM state*) = []

⟨*proof*⟩

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*)))

⟨*proof*⟩

end

6 Transition system of Nieuwenhuis, Oliveras and Tinelli.

theory *NieuwenhuisOliverasTinelli*
imports *SatSolverVerification*
begin

This theory formalizes the transition rule system given by Nieuwenhuis et al. in [3]

6.1 Specification

record *State* =
getF :: *Formula*
getM :: *LiteralTrail*

definition
appliedDecide:: *State* ⇒ *State* ⇒ *Variable set* ⇒ *bool*
where
appliedDecide stateA stateB decisionVars ==
∃ *l*.
(*var l*) ∈ *decisionVars* ∧
¬ *l el* (*elements* (*getM stateA*)) ∧

$\neg$ *opposite l el* (*elements* (*getM stateA*)) $\wedge$

getF stateB = *getF stateA* $\wedge$
getM stateB = *getM stateA* @ [(*l*, *True*)]

definition
applicableDecide :: *State* $\Rightarrow$ *Variable set* $\Rightarrow$ *bool*
where
applicableDecide state decisionVars == $\exists$ *state'*. *appliedDecide state state' decisionVars*

definition
appliedUnitPropagate :: *State* $\Rightarrow$ *State* $\Rightarrow$ *bool*
where
appliedUnitPropagate stateA stateB ==
$\exists$ (*uc*::*Clause*) (*ul*::*Literal*).
uc el (*getF stateA*) $\wedge$
isUnitClause uc ul (*elements* (*getM stateA*)) $\wedge$

getF stateB = *getF stateA* $\wedge$
getM stateB = *getM stateA* @ [(*ul*, *False*)]

definition
applicableUnitPropagate :: *State* $\Rightarrow$ *bool*
where
applicableUnitPropagate state == $\exists$ *state'*. *appliedUnitPropagate state state'*

definition
appliedBackjump :: *State* $\Rightarrow$ *State* $\Rightarrow$ *bool*
where
appliedBackjump stateA stateB ==
$\exists$ *bc bl level*.
isUnitClause bc bl (*elements* (*prefixToLevel level* (*getM stateA*)))
$\wedge$
formulaEntailsClause (*getF stateA*) *bc* $\wedge$
var bl $\in$ *vars* (*getF stateA*) $\cup$ *vars* (*elements* (*getM stateA*)) $\wedge$
0 $\leq$ *level* $\wedge$ *level* $<$ (*currentLevel* (*getM stateA*)) $\wedge$

getF stateB = *getF stateA* $\wedge$
getM stateB = *prefixToLevel level* (*getM stateA*) @ [(*bl*, *False*)]

definition
applicableBackjump :: *State* $\Rightarrow$ *bool*
where
applicableBackjump state == $\exists$ *state'*. *appliedBackjump state state'*

definition

appliedLearn :: *State* ⇒ *State* ⇒ *bool*
where
appliedLearn stateA stateB ==
∃ *c*.
(*formulaEntailsClause* (*getF stateA*) *c*) ∧
(*vars c*) ⊆ *vars* (*getF stateA*) ∪ *vars* (*elements* (*getM stateA*))
∧
getF stateB = *getF stateA* @ [*c*] ∧
getM stateB = *getM stateA*

definition
applicableLearn :: *State* ⇒ *bool*
where
applicableLearn state == (∃ *state′*. *appliedLearn state state′*)

Solving starts with the initial formula and the empty trail.

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

Transitions are preformed only by using given rules.

definition
transition stateA stateB decisionVars ==
appliedDecide stateA stateB decisionVars ∨
appliedUnitPropagate stateA stateB ∨
appliedLearn stateA stateB ∨
appliedBackjump stateA stateB

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

definition
transitionRelation decisionVars == ({(*stateA*, *stateB*). *transition stateA stateB decisionVars*})^*

Final state is one in which no rules apply

definition
isFinalState :: *State* ⇒ *Variable set* ⇒ *bool*
where
isFinalState state decisionVars == ¬ (∃ *state′*. *transition state state′ 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*⟩

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*⟩

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*⟩

lemma *applicableLearnCharacterization*:
fixes *stateA*::*State*
shows *applicableLearn stateA* =
(∃ *c*. *formulaEntailsClause* (*getF stateA*) *c* ∧
vars c ⊆ *vars* (*getF stateA*) ∪ *vars* (*elements* (*getM stateA*)))
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

Final states are the ones where no rule is applicable.

lemma *finalStateNonApplicable*:
fixes *state*::*State*
shows *isFinalState state decisionVars* =
(¬ *applicableDecide state decisionVars* ∧
¬ *applicableUnitPropagate state* ∧
¬ *applicableBackjump state* ∧
¬ *applicableLearn state*)
⟨*proof*⟩

6.2 Invariants

Invariants that are relevant for the rest of correctness proof.

definition
invariantsHoldInState :: *State* ⇒ *Formula* ⇒ *Variable set* ⇒ *bool*
where
invariantsHoldInState state F0 decisionVars ==
  InvariantImpliedLiterals (*getF state*) (*getM state*) ∧
  InvariantVarsM (*getM state*) *F0 decisionVars* ∧
  InvariantVarsF (*getF state*) *F0 decisionVars* ∧
  InvariantConsistent (*getM state*) ∧
  InvariantUniq (*getM state*) ∧
  InvariantEquivalent F0 (*getF state*)

Invariants hold in initial states.

lemma *invariantsHoldInInitialState*:
  fixes *state* :: *State* **and** *F0* :: *Formula*
  assumes *isInitialState state F0*
  shows *invariantsHoldInState state F0 decisionVars*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *invariantsHoldInValidRunsFromInitialState*:
  fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set*
  assumes *isInitialState state0 F0*
  and (*state0*, *state*) ∈ *transitionRelation decisionVars*
  shows *invariantsHoldInState state F0 decisionVars*
⟨*proof*⟩

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 $\neg$ *formulaFalse F0* (*elements* (*getM state*)) and *decisionVars* $\subseteq$ *vars* (*elements* (*getM state*))

The soundness theorems claim that if *UNSAT* state is reached the formula is unsatisfiable and if *SAT* state is reached, the formula is satisfiable.

Completeness theorems claim that every final state is either *UNSAT* or *SAT*. A consequence of this and soundness theorems, is that if formula is unsatisfiable the solver will finish in an *UNSAT* state, and if the formula is satisfiable the solver will finish in a *SAT* state.

6.3 Soundness

theorem *soundnessForUNSAT*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
isInitialState state0 F0 **and**
(*state0*, *state*) $\in$ *transitionRelation decisionVars*

formulaFalse (*getF state*) (*elements* (*getM state*))
decisions (*getM state*) = []

shows $\neg$ *satisfiable F0*

⟨*proof*⟩

theorem *soundnessForSAT*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
vars F0 $\subseteq$ *decisionVars* **and**

isInitialState state0 F0 **and**
(*state0*, *state*) $\in$ *transitionRelation decisionVars*

$\neg$ *formulaFalse* (*getF state*) (*elements* (*getM state*))
vars (*elements* (*getM state*)) $\supseteq$ *decisionVars*
shows
model (*elements* (*getM state*)) *F0*

⟨*proof*⟩

6.4 Termination

This system is terminating, but only under assumption that there is no infinite derivation consisting only of applications of rule *Learn*. We will formalize this condition by requiring that there there exists an ordering *learnL* on the formulae that is well-founded such that the state is decreased with each application of the *Learn* rule. If such ordering exists, the termination ordering is built as a lexicographic combination of *lexLessRestricted* trail ordering and the *learnL* ordering.

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

(*getM stateA*, *getM stateB*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)}
definition *learnLessState learnL* == {((*stateA*::*State*), (*stateB*::*State*)).

getM stateA = *getM stateB* ∧ (*getF stateA*, *getF stateB*) ∈ *learnL*}
definition *terminationLess F0 decisionVars learnL* ==
{((*stateA*::*State*), (*stateB*::*State*)).
(*stateA*,*stateB*) ∈ *lexLessState F0 decisionVars* ∨
(*stateA*,*stateB*) ∈ *learnLessState learnL*}

We want to show that every valid transition decreases a state with respect to the constructed termination ordering. Therefore, we show that *Decide*, *UnitPropagate* and *Backjump* rule decrease the trail with respect to the restricted trail ordering *lexLessRestricted*. Invariants ensure that trails are indeed uniq, consistent and with finite variable sets. By assumption, *Learn* rule will decrease the formula component of the state with respect to the *learnL* ordering.

lemma *trailIsDecreasedByDeciedUnitPropagateAndBackjump*:
fixes *stateA*::*State* **and** *stateB*::*State*
assumes *invariantsHoldInState stateA F0 decisionVars* **and**
appliedDecide stateA stateB decisionVars ∨ *appliedUnitPropagate stateA stateB* ∨ *appliedBackjump stateA stateB*
shows (*getM stateB*, *getM stateA*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)
⟨*proof*⟩

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*:
fixes *stateA*::*State* **and** *stateB*::*State*
assumes *invariantsHoldInState stateA F0 decisionVars* **and** *transition stateA stateB decisionVars*

appliedLearn stateA stateB $\longrightarrow$ (*getF stateB*, *getF stateA*) $\in$ *learnL*
shows (*stateB*, *stateA*) $\in$ *terminationLess F0 decisionVars learnL*
⟨*proof*⟩

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 == ($\forall$ *state*::*State*. (*state*, *stateMin*) $\notin$ *terminationLess F0 decisionVars learnL*)

lemma *minimalStatesAreFinal*:
fixes *stateA*::*State*
assumes *∗*: $\forall$ (*stateA*::*State*) (*stateB*::*State*). *appliedLearn stateA stateB* $\longrightarrow$ (*getF stateB*, *getF stateA*) $\in$ *learnL* **and**
invariantsHoldInState state F0 decisionVars **and** *isMinimalState state F0 decisionVars learnL*
shows *isFinalState state decisionVars*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *wfLearnLessState*:
assumes *wf learnL*
shows *wf* (*learnLessState learnL*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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 neccessary.

theorem *wfTransitionRelation*:
fixes *decisionVars* :: *Variable set* **and** *F0* :: *Formula*
assumes *finite decisionVars* **and** *isInitialState state0 F0* **and**
∗: $\exists$ *learnL*::(*Formula* $\times$ *Formula*) *set*.
wf learnL $\wedge$

(∀ *stateA stateB*. *appliedLearn stateA stateB* ⟶ (*getF stateB*, *getF stateA*) ∈ *learnL*)
shows *wf* {(*stateB*, *stateA*).
(*state0*, *stateA*) ∈ *transitionRelation decisionVars* ∧ (*transition stateA stateB decisionVars*)}

⟨*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 intial 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*⟩

Now we prove the final strong termination result which states that there cannot be infinite chains of transitions. If there is an infinite transition chain that starts from an initial state, its elements would for a set that would contain initial state and for every element of that set there would be another element of that set that is directly reachable from it. We show that no such set exists.

corollary *noInfiniteTransitionChains*:
fixes *F0*::*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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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 decisionVars* **and**
isFinalState state decisionVars
shows ¬ *formulaFalse* (*getF state*) (*elements* (*getM state*)) ∧ *vars* (*elements* (*getM state*)) ⊇ *decisionVars*

⟨*proof*⟩

theorem *completenessForUNSAT*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
vars F0 ⊆ *decisionVars* **and**

¬ *satisfiable F0* **and**

isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation decisionVars* **and**
isFinalState state decisionVars
shows
formulaFalse (*getF state*) (*elements* (*getM state*)) ∧ *decisions* (*getM state*) = []

⟨*proof*⟩

theorem *partialCorrectness*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
vars F0 ⊆ *decisionVars* **and**

isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation decisionVars* **and**
isFinalState state decisionVars
shows
satisfiable F0 = (¬ *formulaFalse* (*getF state*) (*elements* (*getM state*)))

⟨*proof*⟩

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 ==

$\exists\ l\ reason.$
$getConflictFlag\ stateA = True \wedge$
$l\ el\ getC\ stateA \wedge$
$formulaEntailsClause\ (getF\ stateA)\ reason \wedge$
$isReason\ reason\ (opposite\ l)\ (elements\ (getM\ stateA)) \wedge$

$getF\ stateB = getF\ stateA \wedge$
$getM\ stateB = getM\ stateA \wedge$
$getConflictFlag\ stateB = True \wedge$
$getC\ stateB = resolve\ (getC\ stateA)\ reason\ l$

definition
$applicableExplain :: State \Rightarrow bool$
where
$applicableExplain\ state == \exists\ state'.\ appliedExplain\ state\ state'$

definition
$appliedLearn :: State \Rightarrow State \Rightarrow bool$
where
$appliedLearn\ stateA\ stateB ==$
$getConflictFlag\ stateA = True \wedge$
$\neg\ getC\ stateA\ el\ getF\ stateA \wedge$

$getF\ stateB = getF\ stateA\ @\ [getC\ stateA] \wedge$
$getM\ stateB = getM\ stateA \wedge$
$getConflictFlag\ stateB = True \wedge$
$getC\ stateB = getC\ stateA$

definition
$applicableLearn :: State \Rightarrow bool$
where
$applicableLearn\ state == \exists\ state'.\ appliedLearn\ state\ state'$

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.

definition
$appliedBackjump :: State \Rightarrow State \Rightarrow bool$
where
$appliedBackjump\ stateA\ stateB ==$
$\exists\ l\ level.$
$getConflictFlag\ stateA = True \wedge$
$isBackjumpLevel\ level\ l\ (getC\ stateA)\ (getM\ stateA) \wedge$

$getF\ stateB = getF\ stateA \wedge$
$getM\ stateB = prefixToLevel\ level\ (getM\ stateA)\ @\ [(l, False)] \wedge$
$getConflictFlag\ stateB = False \wedge$
$getC\ stateB = []$

definition
applicableBackjump :: *State* ⇒ *bool*
where
applicableBackjump state == ∃ *state′*. *appliedBackjump state state′*

Solving starts with the initial formula, the empty trail and in non conflicting state.

definition
isInitialState :: *State* ⇒ *Formula* ⇒ *bool*
where
isInitialState state F0 ==
    getF state = *F0* ∧
    getM state = [] ∧
    getConflictFlag state = *False* ∧
    getC state = []

Transitions are preformed only by using given rules.

definition
transition :: *State* ⇒ *State* ⇒ *Formula* ⇒ *Variable set* ⇒ *bool*
where
transition stateA stateB F0 decisionVars==
    appliedDecide *stateA stateB decisionVars* ∨
    appliedUnitPropagate stateA stateB F0 decisionVars ∨
    appliedConflict *stateA stateB* ∨
    appliedExplain *stateA stateB* ∨
    appliedLearn *stateA stateB* ∨
    appliedBackjump *stateA stateB*

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

definition
transitionRelation F0 decisionVars == ({(*stateA*, *stateB*). *transition stateA stateB F0 decisionVars*})^*

Final state is one in which no rules apply

definition
isFinalState :: *State* ⇒ *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*.

$(var\ l) \in decisionVars \wedge$
$\neg\ l\ el\ (elements\ (getM\ stateA)) \wedge$
$\neg\ opposite\ l\ el\ (elements\ (getM\ stateA)))$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*⟩

lemma *applicableBackjumpCharacterization*:
fixes *stateA*::*State*
shows *applicableBackjump stateA* =
(∃ *l level*.
getConflictFlag stateA = *True* ∧
isBackjumpLevel level l (*getC stateA*) (*getM stateA*)
) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *applicableExplainCharacterization*:
fixes *stateA*::*State*
shows *applicableExplain stateA* =
(∃ *l reason*.
getConflictFlag stateA = *True* ∧
l el getC stateA ∧
formulaEntailsClause (*getF stateA*) *reason* ∧
isReason reason (*opposite l*) (*elements* (*getM stateA*))
)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*⟩

lemma *applicableLearnCharacterization*:
fixes *stateA*::*State*

shows *applicableLearn stateA* =
(*getConflictFlag stateA* = *True* ∧
¬ *getC stateA el getF stateA*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

Final states are the ones where no rule is applicable.

lemma *finalStateNonApplicable*:
fixes *state*::*State*
shows *isFinalState state F0 decisionVars* =
(¬ *applicableDecide state decisionVars* ∧
¬ *applicableUnitPropagate state F0 decisionVars* ∧
¬ *applicableBackjump state* ∧
¬ *applicableLearn state* ∧
¬ *applicableConflict state* ∧
¬ *applicableExplain state*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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

1. *UNSAT* states where *getConflictFlag state* = *True* and *getC state* = [].
2. *SAT* states where *getConflictFlag state* = *False*, ¬ *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.

7.3 Soundness

theorem *soundnessForUNSAT*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation F0 decisionVars*

getConflictFlag state = *True* **and**
getC state = []
shows ¬ *satisfiable F0*
⟨*proof*⟩

theorem *soundnessForSAT*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
vars F0 $\subseteq$ *decisionVars* **and**

isInitialState state0 F0 **and**
(*state0*, *state*) $\in$ *transitionRelation F0 decisionVars* **and**

getConflictFlag state = *False*
$\neg$ *formulaFalse* (*getF state*) (*elements* (*getM state*))
vars (*elements* (*getM state*)) $\supseteq$ *decisionVars*
shows
model (*elements* (*getM state*)) *F0*
⟨*proof*⟩

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*) $\in$ *lexLessRestricted* (*vars F0* $\cup$ *decisionVars*)}
definition *boolLessState* == {((*stateA*::*State*), (*stateB*::*State*)).
getM stateA = *getM stateB* $\wedge$
(*getConflictFlag stateA*, *getConflictFlag stateB*) $\in$ *boolLess*}
definition *multLessState* == {((*stateA*::*State*), (*stateB*::*State*)).
getM stateA = *getM stateB* $\wedge$
getConflictFlag stateA = *getConflictFlag stateB* $\wedge$
(*getC stateA*, *getC stateB*) $\in$ *multLess* (*getM stateA*)}
definition *learnLessState* == {((*stateA*::*State*), (*stateB*::*State*)).
getM stateA = *getM stateB* $\wedge$
getConflictFlag stateA = *getConflictFlag stateB* $\wedge$
getC stateA = *getC stateB* $\wedge$
(*getF stateA*, *getF stateB*) $\in$ *learnLess* (*getC stateA*)}

definition *terminationLess F0 decisionVars* == {((*stateA*::*State*), (*stateB*::*State*)).
(*stateA*,*stateB*) $\in$ *lexLessState F0 decisionVars* $\vee$
(*stateA*,*stateB*) $\in$ *boolLessState* $\vee$
(*stateA*,*stateB*) $\in$ *multLessState* $\vee$
(*stateA*,*stateB*) $\in$ *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 *lexLessRestricted*. Invariants ensure that trails are indeed uniq, consistent and with finite variable sets.

lemma *trailIsDecreasedByDeciedUnitPropagateAndBackjump*:
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*⟩

Next we show that $Conflict$ decreases the conflict flag in the *boolLess* ordering.

lemma *conflictFlagIsDecreasedByConflict*:
fixes *stateA*::*State* **and** *stateB*::*State*
assumes *appliedConflict stateA stateB*
shows *getM stateA* = *getM stateB* **and** (*getConflictFlag stateB*, *getConflictFlag stateA*) ∈ *boolLess*
⟨*proof*⟩

Next we show that $Explain$ decreases the conflict clause with respect to the *multLess* clause ordering.

lemma *conflictClauseIsDecreasedByExplain*:
fixes *stateA*::*State* **and** *stateB*::*State*
assumes *appliedExplain stateA stateB*
shows
getM stateA = *getM stateB* **and**
getConflictFlag stateA = *getConflictFlag stateB* **and**
(*getC stateB*, *getC stateA*) ∈ *multLess* (*getM stateA*)
⟨*proof*⟩

Finally, we show that $Learn$ decreases the formula in the *learnLess* formula ordering.

lemma *formulaIsDecreasedByLearn*:
fixes *stateA*::*State* **and** *stateB*::*State*
assumes *appliedLearn stateA stateB*
shows
getM stateA = *getM stateB* **and**
getConflictFlag stateA = *getConflictFlag stateB* **and**
getC stateA = *getC stateB* **and**
(*getF stateB*, *getF stateA*) ∈ *learnLess* (*getC stateA*)
⟨*proof*⟩

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*⟩

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*⟩

We now prove that termination ordering is well founded. We start with several auxiliary lemmas, one for each component of the termination ordering.

lemma *wfLexLessState*:
fixes *decisionVars* :: *Variable set* **and** *F0* :: *Formula*
assumes *finite decisionVars*
shows *wf* (*lexLessState F0 decisionVars*)
⟨*proof*⟩

lemma *wfBoolLessState*:
shows *wf boolLessState*
⟨*proof*⟩

lemma *wfMultLessState*:
shows *wf multLessState*
⟨*proof*⟩

lemma *wfLearnLessState*:
shows *wf learnLessState*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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
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*⟩

Now we prove the final strong termination result which states that there cannot be infinite chains of transitions. If there is an infinite transition chain that starts from an initial state, its elements would for a set that would contain initial state and for every element of that set there would be another element of that set that is directly reachable from it. We show that no such set exists.

corollary *noInfiniteTransitionChains*:
fixes *F0*::*Formula* **and** *decisionVars*::*Variable set*
assumes *finite decisionVars*
shows ¬ (∃ *Q*::(*State set*). ∃ *state0* ∈ *Q*. *isInitialState state0 F0* ∧

(∀ *state* ∈ *Q*. (∃ *state′* ∈ *Q*. *transition state state′ F0 decisionVars*))
)
⟨*proof*⟩

7.5 Completeness

In this section we will first show that each final state is either *SAT* or *UNSAT* state.

lemma *finalNonConflictState*:
fixes *state*::*State* **and** *FO* :: *Formula*
assumes
getConflictFlag state = *False* **and**

$\neg$ *applicableDecide state decisionVars* **and**
$\neg$ *applicableConflict state*
shows $\neg$ *formulaFalse* (*getF state*) (*elements* (*getM state*)) **and**
vars (*elements* (*getM state*)) $\supseteq$ *decisionVars*
⟨*proof*⟩

lemma *finalConflictingState*:
fixes *state* :: *State*
assumes
InvariantUniq (*getM state*) **and**
InvariantReasonClauses (*getF state*) (*getM state*) **and**
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) **and**
$\neg$ *applicableExplain state* **and**
$\neg$ *applicableBackjump state* **and**
getConflictFlag state
shows
getC state = []
⟨*proof*⟩

lemma *finalStateCharacterizationLemma*:
fixes *state* :: *State*
assumes
InvariantUniq (*getM state*) **and**
InvariantReasonClauses (*getF state*) (*getM state*) **and**
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) **and**
$\neg$ *applicableDecide state decisionVars* **and**
$\neg$ *applicableConflict state*
$\neg$ *applicableExplain state* **and**
$\neg$ *applicableBackjump state*
shows
(*getConflictFlag state* = *False* $\wedge$
$\neg$*formulaFalse* (*getF state*) (*elements* (*getM state*)) $\wedge$
vars (*elements* (*getM state*)) $\supseteq$ *decisionVars*) $\vee$
(*getConflictFlag state* = *True* $\wedge$
getC state = [])
⟨*proof*⟩

theorem *finalStateCharacterization*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
isInitialState state0 F0 **and**
(*state0*, *state*) $\in$ *transitionRelation F0 decisionVars* **and**
isFinalState state F0 decisionVars
shows
(*getConflictFlag state* = *False* $\wedge$
$\neg$*formulaFalse* (*getF state*) (*elements* (*getM state*)) $\wedge$
vars (*elements* (*getM state*)) $\supseteq$ *decisionVars*) $\vee$

$(getConflictFlag\ state = True \wedge$
$getC\ state = [])$

⟨*proof*⟩

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**
$(state0, state) \in transitionRelation\ F0\ decisionVars$ **and**
isFinalState state F0 decisionVars

shows $getConflictFlag\ state = False \wedge \neg formulaFalse\ (getF\ state)\ (elements\ (getM\ state)) \wedge$
$vars\ (elements\ (getM\ state)) \supseteq decisionVars$

⟨*proof*⟩

theorem *completenessForUNSAT*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
$vars\ F0 \subseteq decisionVars$ **and**

$\neg$ *satisfiable F0* **and**

isInitialState state0 F0 **and**
$(state0, state) \in transitionRelation\ F0\ decisionVars$ **and**
isFinalState state F0 decisionVars

shows
$getConflictFlag\ state = True \wedge getC\ state = []$

⟨*proof*⟩

theorem *partialCorrectness*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
$vars\ F0 \subseteq decisionVars$ **and**

isInitialState state0 F0 **and**

(*state0*, *state*) ∈ *transitionRelation F0 decisionVars* **and**
isFinalState state F0 decisionVars

shows
satisfiable F0 = (¬ *getConflictFlag state*)

⟨*proof*⟩

end

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

theory *SatSolverCode*
imports *SatSolverVerification HOL−Library.Code-Target-Numeral*
begin

8.1 Specification

lemma [*code-unfold*]:
fixes *literal* :: *Literal* **and** *clause* :: *Clause*
shows *literal el clause* = *List.member clause literal*
⟨*proof*⟩

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
getC :: *Clause* — Conflict analysis clause - always false in M
getCl :: *Literal* — Last asserted literal in (opposite getC)
getCll :: *Literal* — Second last asserted literal in (opposite getC)
getCn :: *nat* — Number of literals of (opposite getC) on the (currentLevel M)

definition
setWatch1 :: *nat* ⇒ *Literal* ⇒ *State* ⇒ *State*
where
setWatch1 clause literal state =
state(| *getWatch1* := (*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
swapWatches clause state ==
state(| *getWatch1* := (*getWatch1 state*)(*clause* := (*getWatch2 state clause*)),
getWatch2 := (*getWatch2 state*)(*clause* := (*getWatch1 state clause*)) |)

declare *swapWatches-def*[*code-unfold*]

primrec *getNonWatchedUnfalsifiedLiteral* :: *Clause* ⇒ *Literal* ⇒ *Literal* ⇒ *LiteralTrail* ⇒ *Literal option*
where
getNonWatchedUnfalsifiedLiteral [] *w1 w2 M* = *None* |
getNonWatchedUnfalsifiedLiteral (*literal* # *clause*) *w1 w2 M* =

```
(if literal ≠ w1 ∧
    literal ≠ w2 ∧
    ¬ (literalFalse literal (elements M)) then
        Some literal
 else
        getNonWatchedUnfalsifiedLiteral clause w1 w2 M
)
```

definition
setReason :: *Literal* ⇒ *nat* ⇒ *State* ⇒ *State*
where
setReason literal clause state =
    state(| *getReason* := (*getReason state*)(*literal* := *Some clause*) |)

declare *setReason-def*[*code-unfold*]

primrec *notifyWatches-loop*::*Literal* ⇒ *nat list* ⇒ *nat list* ⇒ *State* ⇒ *State*
where
notifyWatches-loop literal [] *newWl state* = *state*(| *getWatchList* := (*getWatchList state*)(*literal* := *newWl*) |) |
notifyWatches-loop literal (*clause* # *list'*) *newWl state* =
    (*let state'* = (*if Some literal* = (*getWatch1 state clause*) *then*
                    (*swapWatches clause state*)
                else
                    state) *in*
    case (*getWatch1 state' clause*) *of*
        None ⇒ *state*
    | *Some w1* ⇒ (
    case (*getWatch2 state' clause*) *of*
        None ⇒ *state*
    | *Some w2* ⇒
    (*if* (*literalTrue w1* (*elements* (*getM state'*))) *then*
        notifyWatches-loop literal list' (*clause* # *newWl*) *state'*
     else
      (*case* (*getNonWatchedUnfalsifiedLiteral* (*nth* (*getF state'*) *clause*) *w1 w2* (*getM state'*)) *of*
          Some l' ⇒
              notifyWatches-loop literal list' newWl (*setWatch2 clause l' state'*)
        | *None* ⇒
              (*if* (*literalFalse w1* (*elements* (*getM state'*))) *then*
                      let state'' = (*state'*(| *getConflictFlag* := *True*, *getConflictClause* := *clause* |)) *in*
                notifyWatches-loop literal list' (*clause* # *newWl*) *state''*
              else
                    let state'' = *state'*(| *getQ* := (*if w1 el* (*getQ state'*) *then*

```
                                        (getQ state')
                                    else
                                        (getQ state') @ [w1]
                                    )
                                ⦈ in
                    let state''' = (setReason w1 clause state'') in
                    notifyWatches-loop literal list' (clause # newWl) state'''
                )
        )
    )
    )
    )
```

definition
notifyWatches :: *Literal* ⇒ *State* ⇒ *State*
where
notifyWatches literal state ==
    notifyWatches-loop literal (*getWatchList state literal*) [] *state*

declare *notifyWatches-def*[*code-unfold*]

definition
assertLiteral :: *Literal* ⇒ *bool* ⇒ *State* ⇒ *State*
where
assertLiteral literal decision state ==
    let state' = (*state*⦇ *getM* := (*getM state*) @ [(*literal*, *decision*)] ⦈)
in
    notifyWatches (*opposite literal*) *state'*

definition
applyUnitPropagate :: *State* ⇒ *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'''
  )))
 ))


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),
    getCll = (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‴(| getCll := ll |)
        )
```

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 (solve-loop-body state Vbl) Vbl)
  ⟹ solve-loop-dom state Vbl

definition solve::Formula ⇒ ExtendedBool
where
solve F0 =
    (getSATFlag
        (solve-loop
            (initialize F0 initialState) (vars F0)
        )
```

)

definition
InvariantWatchListsContainOnlyClausesFromF :: (*Literal* ⇒ *nat list*) ⇒ *Formula* ⇒ *bool*
where
InvariantWatchListsContainOnlyClausesFromF 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 ==

$\forall$ (*clause*::*nat*). $0 \leq$ *clause* $\wedge$ *clause* $<$ *length formula* $\longrightarrow$ *watch1 clause* $\neq$ *watch2 clause*

definition
watchCharacterizationCondition::*Literal* $\Rightarrow$ *Literal* $\Rightarrow$ *LiteralTrail* $\Rightarrow$ *Clause* $\Rightarrow$ *bool*
where
watchCharacterizationCondition w1 w2 M clause =
(*literalFalse w1* (*elements M*) $\longrightarrow$
(($\exists$ *l*. *l el clause* $\wedge$ *literalTrue l* (*elements M*) $\wedge$ *elementLevel l M* $\leq$ *elementLevel* (*opposite w1*) *M*) $\vee$
($\forall$ *l*. *l el clause* $\wedge$ *l* $\neq$ *w1* $\wedge$ *l* $\neq$ *w2* $\longrightarrow$
literalFalse l (*elements M*) $\wedge$ *elementLevel* (*opposite l*) *M* $\leq$ *elementLevel* (*opposite w1*) *M*)
)
)

definition
InvariantWatchCharacterization::*Formula* $\Rightarrow$ (*nat* $\Rightarrow$ *Literal option*) $\Rightarrow$ (*nat* $\Rightarrow$ *Literal option*) $\Rightarrow$ *LiteralTrail* $\Rightarrow$ *bool*
where
InvariantWatchCharacterization F watch1 watch2 M =
($\forall$ *c w1 w2*. (*0* $\leq$ *c* $\wedge$ *c* $<$ *length F* $\wedge$ *Some w1* = *watch1 c* $\wedge$ *Some w2* = *watch2 c*) $\longrightarrow$
watchCharacterizationCondition w1 w2 M (*nth F c*) $\wedge$
watchCharacterizationCondition w2 w1 M (*nth F c*)
)

definition
InvariantQCharacterization :: *bool* $\Rightarrow$ *Literal list* $\Rightarrow$ *Formula* $\Rightarrow$ *LiteralTrail* $\Rightarrow$ *bool*
where
InvariantQCharacterization conflictFlag Q F M ==
$\neg$ *conflictFlag* $\longrightarrow$ ($\forall$ (*l*::*Literal*). *l el Q* = ($\exists$ (*c*::*Clause*). *c el F* $\wedge$ *isUnitClause c l* (*elements M*)))

definition
InvariantUniqQ :: *Literal list* $\Rightarrow$ *bool*
where
InvariantUniqQ Q =
uniq Q

definition
InvariantConflictFlagCharacterization :: *bool* $\Rightarrow$ *Formula* $\Rightarrow$ *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* ⟶

$$(\exists\ (reason{::}nat).\ (GetReason\ literal) = Some\ reason \wedge 0 \leq reason \wedge reason < length\ F \wedge (isUnitClause\ (nth\ F\ reason)\ literal\ (elements\ M) \vee clauseFalse\ (nth\ F\ reason)\ (elements\ M)))$$

definition
$InvariantConflictClauseCharacterization :: bool \Rightarrow nat \Rightarrow Formula \Rightarrow LiteralTrail \Rightarrow bool$
where
$InvariantConflictClauseCharacterization\ conflictFlag\ conflictClause\ F\ M == conflictFlag \longrightarrow (conflictClause < length\ F \wedge clauseFalse\ (nth\ F\ conflictClause)\ (elements\ M))$

definition
$InvariantClCharacterization :: Literal \Rightarrow Clause \Rightarrow LiteralTrail \Rightarrow bool$

where
$InvariantClCharacterization\ Cl\ C\ M == isLastAssertedLiteral\ Cl\ (oppositeLiteralList\ C)\ (elements\ M)$

definition
$InvariantCllCharacterization :: Literal \Rightarrow Literal \Rightarrow Clause \Rightarrow LiteralTrail \Rightarrow bool$
where
$InvariantCllCharacterization\ Cl\ Cll\ C\ M == set\ C \neq \{opposite\ Cl\} \longrightarrow isLastAssertedLiteral\ Cll\ (removeAll\ Cl\ (oppositeLiteralList\ C))\ (elements\ M)$

definition
$InvariantClCurrentLevel :: Literal \Rightarrow LiteralTrail \Rightarrow bool$
where
$InvariantClCurrentLevel\ Cl\ M == elementLevel\ Cl\ M = currentLevel\ M$

definition
$InvariantCnCharacterization :: nat \Rightarrow Clause \Rightarrow LiteralTrail \Rightarrow bool$
where
$InvariantCnCharacterization\ Cn\ C\ M == Cn = length\ (filter\ (\lambda\ l.\ elementLevel\ (opposite\ l)\ M = currentLevel\ M)\ (remdups\ C))$

definition
$InvariantUniqC :: Clause \Rightarrow bool$

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*)
⟨*proof*⟩

lemma *getNonWatchedUnfalsifiedLiteralNoneCharacterization*:
fixes *clause* :: *Clause* **and** *w1* :: *Literal* **and** *w2* :: *Literal* **and** *M* :: *LiteralTrail*
assumes
  getNonWatchedUnfalsifiedLiteral clause w1 w2 M = *None*
shows
  ∀ *l*. *l el clause* ∧ *l* ≠ *w1* ∧ *l* ≠ *w2* ⟶ *literalFalse l* (*elements M*)
⟨*proof*⟩

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′)
⟨*proof*⟩

lemma *notifyWatchesLoopPreservedVariables*:
fixes *literal* :: *Literal* **and** *Wl* :: *nat list* **and** *newWl* :: *nat list* **and** *state* :: *State*
assumes
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
∀ (*c*::*nat*). *c* ∈ *set Wl* ⟶ *0* ≤ *c* ∧ *c* < *length (getF state)*
shows
let state′ = (notifyWatches-loop literal Wl newWl state) in
(getM state′) = (getM state) ∧
(getF state′) = (getF state) ∧
(getSATFlag state′) = (getSATFlag state) ∧
isPrefix (getQ state) (getQ state′)

⟨*proof*⟩

lemma *notifyWatchesStartQIreleveant*:
fixes *literal* :: *Literal* **and** *Wl* :: *nat list* **and** *newWl* :: *nat list* **and** *state* :: *State*
assumes
InvariantWatchesEl (getF stateA) (getWatch1 stateA) (getWatch2 stateA) **and**
∀ (*c*::*nat*). *c* ∈ *set Wl* ⟶ *0* ≤ *c* ∧ *c* < *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
let state′ = (notifyWatches-loop literal Wl newWl stateA) in
let state″ = (notifyWatches-loop literal Wl newWl stateB) in
(getM state′) = (getM state″) ∧
(getF state′) = (getF state″) ∧
(getSATFlag state′) = (getSATFlag state″) ∧
(getConflictFlag state′) = (getConflictFlag state″)

⟨*proof*⟩

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
$\forall$ (*c*::*nat*). $c \in$ *set Wl* $\longrightarrow$ $0 \leq c \wedge c <$ *length* (*getF state*)
shows
let state$'$ = (*notifyWatches-loop literal Wl newWl state*) *in*
$\forall$ *c*. $c \notin$ *set Wl* $\longrightarrow$ (*getWatch1 state*$'$ *c*) = (*getWatch1 state c*) $\wedge$ (*getWatch2 state*$'$ *c*) = (*getWatch2 state c*)

⟨*proof*⟩

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
$\forall$ (*c*::*nat*). $c \in$ *set Wl* $\longrightarrow$ $0 \leq c \wedge c <$ *length* (*getF state*)
shows
let state$'$ = (*notifyWatches-loop literal Wl newWl state*) *in*
InvariantWatchesEl (*getF state*$'$) (*getWatch1 state*$'$) (*getWatch2 state*$'$)
⟨*proof*⟩

lemma *InvariantWatchesDifferNotifyWatchesLoop*:
fixes *literal* :: *Literal* **and** *Wl* :: *nat list* **and** *newWl* :: *nat list* **and** *state* :: *State*
assumes
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
$\forall$ (*c*::*nat*). $c \in$ *set Wl* $\longrightarrow$ $0 \leq c \wedge c <$ *length* (*getF state*)
shows
let state$'$ = (*notifyWatches-loop literal Wl newWl state*) *in*
InvariantWatchesDiffer (*getF state*$'$) (*getWatch1 state*$'$) (*getWatch2 state*$'$)
⟨*proof*⟩

lemma *InvariantWatchListsContainOnlyClausesFromFNotifyWatches-Loop*:
fixes *literal* :: *Literal* **and** *Wl* :: *nat list* **and** *newWl* :: *nat list* **and** *state* :: *State*
assumes
InvariantWatchListsContainOnlyClausesFromF (*getWatchList 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*) *in*
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state′*) (*getF state′*)
⟨*proof*⟩

lemma *InvariantWatchListsCharacterizationNotifyWatchesLoop*:
fixes *literal* :: *Literal* **and** *Wl* :: *nat list* **and** *newWl* :: *nat list* **and** *state* :: *State*
assumes
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchListsUniq (*getWatchList state*)
∀ (*c*::*nat*). *c* ∈ *set Wl* ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF state*)
∀ (*c*::*nat*) (*l*::*Literal*). *l* ≠ *literal* ⟶
(*c* ∈ *set* (*getWatchList state l*)) = (*Some l* = *getWatch1 state c* ∨ *Some l* = *getWatch2 state c*)
∀ (*c*::*nat*). (*c* ∈ *set newWl* ∨ *c* ∈ *set Wl*) = (*Some literal* = (*getWatch1 state c*) ∨ *Some literal* = (*getWatch2 state c*))
set Wl ∩ *set newWl* = {}
uniq Wl
uniq newWl
shows
let state′ = (*notifyWatches-loop literal Wl newWl state*) *in*
InvariantWatchListsCharacterization (*getWatchList state′*) (*getWatch1 state′*) (*getWatch2 state′*) ∧
InvariantWatchListsUniq (*getWatchList state′*)
⟨*proof*⟩

lemma *NotifyWatchesLoopWatchCharacterizationEffect*:
fixes *literal* :: *Literal* **and** *Wl* :: *nat list* **and** *newWl* :: *nat list* **and** *state* :: *State*
assumes
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantConsistent (*getM state*) **and**
InvariantUniq (*getM state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) *M*
∀ (*c*::*nat*). *c* ∈ *set Wl* ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF state*) **and**
getM state = *M* @ [(*opposite literal*, *decision*)]

uniq Wl
∀ (*c*::*nat*). *c* ∈ *set Wl* ⟶ *Some literal* = (*getWatch1 state c*) ∨ *Some literal* = (*getWatch2 state c*)

shows
let state′ = *notifyWatches-loop literal Wl newWl state in*
∀ (*c*::*nat*). *c* ∈ *set Wl* ⟶ (∀ *w1 w2*.(*Some w1* = (*getWatch1 state′ c*) ∧ *Some w2* = (*getWatch2 state′ c*)) ⟶
(*watchCharacterizationCondition w1 w2* (*getM state′*) (*nth* (*getF state′*) *c*) ∧
watchCharacterizationCondition w2 w1 (*getM state′*) (*nth* (*getF state′*) *c*))
)
⟨*proof*⟩

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*))))
⟨*proof*⟩

lemma *NotifyWatchesLoopQEffect*:
fixes *literal* :: *Literal* **and** *Wl* :: *nat list* **and** *newWl* :: *nat list* **and** *state* :: *State*
assumes
(*getM state*) = *M* @ [(*opposite literal*, *decision*)] **and**
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*) **and**
InvariantConsistent (*getM 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*))
⟨*proof*⟩

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
∀ (*c*::*nat*). *c* ∈ *set Wl* ⟶ 0 ≤ *c* ∧ *c* < *length* (*getF state*) **and**
InvariantUniqQ (*getQ state*)
shows
let state′ = *notifyWatches-loop literal Wl newWl state in*
InvariantUniqQ (*getQ state′*)

⟨*proof*⟩

lemma *InvariantConflictClauseCharacterizationAfterNotifyWatches*:
assumes
(*getM state*) = *M* @ [(*opposite literal*, *decision*)] **and**
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**
InvariantConflictClauseCharacterization (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*)
uniq Wl
shows
let state′ = (*notifyWatches-loop literal Wl newWl state*) *in*
InvariantConflictClauseCharacterization (*getConflictFlag state′*) (*getConflictClause state′*) (*getF state′*) (*getM state′*)
⟨*proof*⟩

lemma *InvariantGetReasonIsReasonQSubset*:
assumes $Q \subseteq Q'$ **and**
InvariantGetReasonIsReason GetReason F M Q'
shows
InvariantGetReasonIsReason GetReason F M Q
⟨*proof*⟩

lemma *InvariantGetReasonIsReasonAfterNotifyWatches*:
assumes
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
$\forall$ (*c*::*nat*). $c \in$ *set Wl* $\longrightarrow$ $0 \leq c \wedge c <$ *length* (*getF state*) **and**
$\forall$ (*c*::*nat*). $c \in$ *set Wl* $\longrightarrow$ *Some literal* = (*getWatch1 state c*) $\vee$ *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' = *notifyWatches-loop literal Wl newWl state in*
let Q' = *Q* ∪ (*set* (*getQ state'*) − *set* (*getQ state*)) *in*
InvariantGetReasonIsReason (*getReason state'*) (*getF state'*) (*getM state'*) *Q'*
⟨*proof*⟩

lemma *assertLiteralEffect*:
fixes *state*::*State* **and** *l*::*Literal* **and** *d*::*bool*
assumes
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
shows
(*getM* (*assertLiteral l d state*)) = (*getM state*) @ [(*l*, *d*)] **and**
(*getF* (*assertLiteral l d state*)) = (*getF state*) **and**
(*getSATFlag* (*assertLiteral l d state*)) = (*getSATFlag state*) **and**
isPrefix (*getQ state*) (*getQ* (*assertLiteral l d state*))
⟨*proof*⟩

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′*)

⟨*proof*⟩

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*⟩

lemma *assertLiteralConflictFlagEffect*:
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*)
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
shows
let state′ = assertLiteral literal decision state in
getConflictFlag state′ = (getConflictFlag state ∨
(∃ clause. clause el (getF state) ∧
opposite literal el clause ∧
clauseFalse clause ((elements (getM state)) @ [literal])))
⟨*proof*⟩

lemma *InvariantConflictFlagCharacterizationAfterAssertLiteral*:
assumes
InvariantConsistent ((*getM state*) @ [(*literal*, *decision*)])
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **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
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
shows
let state′ = (assertLiteral literal decision state) in
InvariantConflictFlagCharacterization (*getConflictFlag state′*) (*getF state′*) (*getM state′*)
⟨*proof*⟩

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*⟩

lemma *assertLiteralQEffect*:
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) (getM state)
shows
let state′ = assertLiteral literal decision state in
set (getQ state′) = set (getQ state) ∪
{ ul. (∃ uc. uc el (getF state) ∧
opposite literal el uc ∧
isUnitClause uc ul ((elements (getM state)) @ [literal])) }
*(***is** *let state′ = assertLiteral literal decision state in*
set (getQ state′) = set (getQ state) ∪ ?ulSet)
⟨*proof*⟩

lemma *InvariantQCharacterizationAfterAssertLiteral*:
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*)
shows
let state′ = (*assertLiteral literal decision state*) *in*
InvariantQCharacterization (*getConflictFlag state′*) (*removeAll literal* (*getQ state′*)) (*getF state′*) (*getM state′*)
⟨*proof*⟩

lemma *AssertLiteralStartQIreleveant*:
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″*)

⟨*proof*⟩

lemma *assertedLiteralIsNotUnit*:
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*)
shows
let state′ = *assertLiteral literal decision state in*
¬ *literal* ∈ (*set* (*getQ state′*) − *set*(*getQ state*))
⟨*proof*⟩

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*⟩

lemma *InvariantUniqQAfterAssertLiteral*:
assumes
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*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′*)
⟨*proof*⟩

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 ⟶ ¬ (*getConflictFlag state*) ∧ (*getQ state*) = []
shows
let state′ = *assertLiteral literal decision state in*

InvariantNoDecisionsWhenConflict (*getF state'*) (*getM state'*) (*currentLevel* (*getM state'*)) ∧
InvariantNoDecisionsWhenUnit (*getF state'*) (*getM state'*) (*currentLevel* (*getM state'*))
⟨*proof*⟩

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*) (*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*⟩

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*⟩

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*⟩

lemma *InvariantUniqAfterApplyUnitPropagate*:
assumes
InvariantUniq (getM state)
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) **and**
InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
getQ state ≠ []
¬ (getConflictFlag state)
shows
let state′ = applyUnitPropagate state in
InvariantUniq (getM state′)

⟨*proof*⟩

lemma *InvariantWatchCharacterizationAfterApplyUnitPropagate*:
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*)
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
(*getQ state*) ≠ []
¬ (*getConflictFlag state*)
shows
let state′ = applyUnitPropagate state in
InvariantWatchCharacterization (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) (*getM state′*)
⟨*proof*⟩

lemma *InvariantConflictFlagCharacterizationAfterApplyUnitPropagate*:
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*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
¬ *getConflictFlag state*
getQ state ≠ []
shows
let state′ = (*applyUnitPropagate state*) *in*
InvariantConflictFlagCharacterization (*getConflictFlag state′*) (*getF state′*) (*getM state′*)
⟨*proof*⟩

lemma *InvariantConflictClauseCharacterizationAfterApplyUnitPropagate*:
assumes
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)

and
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*)
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchListsUniq (*getWatchList state*)
¬ *getConflictFlag state*
shows
let state′ = applyUnitPropagate state in
InvariantConflictClauseCharacterization (*getConflictFlag state′*) (*getConflictClause state′*) (*getF state′*) (*getM state′*)
⟨*proof*⟩

lemma *InvariantQCharacterizationAfterApplyUnitPropagate*:
assumes
InvariantConsistent (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
InvariantUniqQ (*getQ state*)
(*getQ state*) ≠ []
¬ (*getConflictFlag state*)
shows
let state″ = applyUnitPropagate state in
InvariantQCharacterization (*getConflictFlag state″*) (*getQ state″*) (*getF state″*) (*getM state″*)
⟨*proof*⟩

lemma *InvariantUniqQAfterApplyUnitPropagate*:
assumes
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*)
InvariantUniqQ (*getQ state*)
getQ state ≠ []
shows

let state'' = applyUnitPropagate state in
InvariantUniqQ (getQ state'')
⟨*proof*⟩

lemma *InvariantNoDecisionsWhenConflictNorUnitAfterUnitPropagate*:
assumes
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)
InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state)
InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
InvariantNoDecisionsWhenConflict (getF state) (getM state) (currentLevel (getM state))
InvariantNoDecisionsWhenUnit (getF state) (getM state) (currentLevel (getM state))
shows
let state' = applyUnitPropagate state in
InvariantNoDecisionsWhenConflict (getF state') (getM state') (currentLevel (getM state')) ∧
InvariantNoDecisionsWhenUnit (getF state') (getM state') (currentLevel (getM state'))
⟨*proof*⟩

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*⟩

lemma *InvariantEquivalentZLAfterApplyUnitPropagate*:

assumes
InvariantEquivalentZL (*getF state*) (*getM state*) *Phi*
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 state′ = *applyUnitPropagate state in*
InvariantEquivalentZL (*getF state′*) (*getM state′*) *Phi*

⟨*proof*⟩

lemma *InvariantVarsQTl*:
assumes
InvariantVarsQ Q F0 Vbl
Q ≠ []
shows
InvariantVarsQ (*tl Q*) *F0 Vbl*
⟨*proof*⟩

lemma *InvariantsVarsAfterApplyUnitPropagate*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*)
(*getF state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*) **and**
InvariantQCharacterization False (*getQ state*) (*getF state*) (*getM state*) **and**
getQ state ≠ []
¬ *getConflictFlag state*
InvariantVarsM (*getM state*) *F0 Vbl* **and**
InvariantVarsQ (*getQ state*) *F0 Vbl* **and**
InvariantVarsF (*getF state*) *F0 Vbl*
shows

let state′ = applyUnitPropagate state in
InvariantVarsM (getM state′) F0 Vbl ∧
InvariantVarsQ (getQ state′) F0 Vbl
⟨*proof*⟩

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*)
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
shows
exhaustiveUnitPropagate-dom state
⟨*proof*⟩

lemma *exhaustiveUnitPropagatePreservedVariables*:
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*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
shows
let state′ = exhaustiveUnitPropagate state in
(*getSATFlag state′*) = (*getSATFlag state*)
⟨*proof*⟩

lemma *exhaustiveUnitPropagatePreservesCurrentLevel*:
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*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
shows
let state′ = exhaustiveUnitPropagate state in
currentLevel (*getM state′*) = *currentLevel* (*getM state*)
⟨*proof*⟩

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′*) (*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*

⟨*proof*⟩

lemma *InvariantConflictClauseCharacterizationAfterExhaustivePropagate*:
assumes
exhaustiveUnitPropagate-dom state
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantConflictClauseCharacterization (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*)
shows
let state′ = exhaustiveUnitPropagate state in
InvariantConflictClauseCharacterization (*getConflictFlag state′*) (*getConflictClause state′*) (*getF state′*) (*getM state′*)
⟨*proof*⟩

lemma *InvariantsNoDecisionsWhenConflictNorUnitAfterExhaustive-Propagate*:
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*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
InvariantUniqQ (*getQ state*)
InvariantNoDecisionsWhenConflict (*getF state*) (*getM state*) (*currentLevel* (*getM state*))
InvariantNoDecisionsWhenUnit (*getF state*) (*getM state*) (*currentLevel* (*getM state*))
shows
let state′ = exhaustiveUnitPropagate state in
InvariantNoDecisionsWhenConflict (*getF state′*) (*getM state′*) (*currentLevel* (*getM state′*)) ∧
InvariantNoDecisionsWhenUnit (*getF state′*) (*getM state′*) (*currentLevel* (*getM state′*))
⟨*proof*⟩

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′*))
⟨*proof*⟩

lemma *InvariantEquivalentZLAfterExhaustiveUnitPropagate*:
assumes
exhaustiveUnitPropagate-dom state
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantEquivalentZL (*getF state*) (*getM state*) *Phi*
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
InvariantUniqQ (*getQ state*)
shows
let state′ = *exhaustiveUnitPropagate state in*
InvariantEquivalentZL (*getF state′*) (*getM state′*) *Phi*

⟨*proof*⟩

lemma *conflictFlagOrQEmptyAfterExhaustiveUnitPropagate*:
assumes
exhaustiveUnitPropagate-dom state
shows

let state′ = exhaustiveUnitPropagate state in
(getConflictFlag state′) ∨ (getQ state′ = [])
⟨*proof*⟩

end
theory *Initialization*
imports *UnitPropagate*
begin

lemma *InvariantsAfterAddClause*:
fixes *state*::*State* **and** *clause* :: *Clause* **and** *Vbl* :: *Variable set*
assumes
InvariantConsistent (getM state)
InvariantUniq (getM state)
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) **and**
InvariantWatchListsUniq (getWatchList state) **and**
InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state) **and**
InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)
InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state)
InvariantConflictClauseCharacterization (getConflictFlag state) (getConflictClause state) (getF state) (getM state)
InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
InvariantUniqQ (getQ state)
InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state))
currentLevel (getM state) = 0
(getConflictFlag state) ∨ (getQ state) = []
InvariantVarsM (getM state) F0 Vbl
InvariantVarsQ (getQ state) F0 Vbl
InvariantVarsF (getF state) F0 Vbl
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'*) (*getM state'*) ∧
InvariantConflictFlagCharacterization (*getConflictFlag state'*) (*getF state'*) (*getM state'*) ∧
InvariantConflictClauseCharacterization (*getConflictFlag state'*) (*getConflictClause state'*) (*getF state'*) (*getM state'*) ∧
InvariantQCharacterization (*getConflictFlag state'*) (*getQ state'*) (*getF state'*) (*getM state'*) ∧
InvariantGetReasonIsReason (*getReason state'*) (*getF state'*) (*getM state'*) (*set* (*getQ state'*)) ∧
InvariantUniqQ (*getQ state'*) ∧
InvariantVarsQ (*getQ state'*) *F0 Vbl* ∧
InvariantVarsM (*getM state'*) *F0 Vbl* ∧
InvariantVarsF (*getF state'*) *F0 Vbl* ∧
currentLevel (*getM state'*) = *0* ∧
((*getConflictFlag state'*) ∨ (*getQ state'*) = [])

⟨*proof*⟩

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*)
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*)
(*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*⟩

lemma *InvariantsAfterInitializationStep*:
fixes
state :: *State* **and** *Phi* :: *Formula* **and** *Vbl*:: *Variable set*
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantConflictClauseCharacterization (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)

InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*))
InvariantUniqQ (*getQ state*)
(*getConflictFlag state*) ∨ (*getQ state*) = []
currentLevel (*getM state*) = *0*
finite Vbl
InvariantVarsM (*getM state*) *F0 Vbl*
InvariantVarsQ (*getQ state*) *F0 Vbl*
InvariantVarsF (*getF state*) *F0 Vbl*
state′ = *initialize Phi state*
set Phi ⊆ *set F0*
shows
InvariantConsistent (*getM state′*) ∧
InvariantUniq (*getM state′*) ∧
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state′*) (*getF state′*) ∧
InvariantWatchListsUniq (*getWatchList state′*) ∧
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′*) ∧
InvariantConflictClauseCharacterization (*getConflictFlag state′*) (*getConflictClause state′*) (*getF state′*) (*getM state′*) ∧
InvariantQCharacterization (*getConflictFlag state′*) (*getQ state′*) (*getF state′*) (*getM state′*) ∧
InvariantUniqQ (*getQ state′*) ∧
InvariantGetReasonIsReason (*getReason state′*) (*getF state′*) (*getM state′*) (*set* (*getQ state′*)) ∧
InvariantVarsM (*getM state′*) *F0 Vbl* ∧
InvariantVarsQ (*getQ state′*) *F0 Vbl* ∧
InvariantVarsF (*getF state′*) *F0 Vbl* ∧
((*getConflictFlag state′*) ∨ (*getQ state′*) = []) ∧
currentLevel (*getM state′*) = *0* (**is** *?Inv state′*)
⟨*proof*⟩

lemma *InvariantEquivalentZLAfterInitializationStep*:
fixes *Phi* :: *Formula*
assumes
(*getSATFlag state* = *UNDEF* ∧ *InvariantEquivalentZL* (*getF state*) (*getM state*) (*filter* (λ *c*. ¬ *clauseTautology c*) *Phi*)) ∨
(*getSATFlag state* = *FALSE* ∧ ¬ *satisfiable* (*filter* (λ *c*. ¬ *clauseTautology c*) *Phi*))
InvariantConsistent (*getM state*)

InvariantUniq (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*)
(*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1
state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2
state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2
state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF
state*) (*getM state*)
InvariantConflictClauseCharacterization (*getConflictFlag state*) (*getConflictClause
state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF
state*) (*getM state*)
InvariantNoDecisionsWhenConflict (*getF state*) (*getM state*) (*currentLevel
*(*getM state*))
InvariantNoDecisionsWhenUnit (*getF state*) (*getM state*) (*currentLevel
*(*getM state*))
InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM
state*) (*set* (*getQ state*))
InvariantUniqQ (*getQ state*)
finite Vbl
InvariantVarsM (*getM state*) *F0 Vbl*
InvariantVarsQ (*getQ state*) *F0 Vbl*
InvariantVarsF (*getF state*) *F0 Vbl*
(*getConflictFlag state*) ∨ (*getQ state*) = []
currentLevel (*getM state*) = *0*
F0 = *Phi* @ *Phi′*
shows
let state′ = *initialize Phi′ state in*
(*getSATFlag state′* = *UNDEF* ∧ *InvariantEquivalentZL* (*getF
state′*) (*getM state′*) (*filter* (λ *c*. ¬ *clauseTautology c*) *F0*)) ∨
(*getSATFlag state′* = *FALSE* ∧ ¬*satisfiable* (*filter* (λ *c*. ¬ *clause-
Tautology c*) *F0*))
⟨*proof*⟩

lemma *InvariantsAfterInitialization*:
shows
let state′ = (*initialize F0 initialState*) *in*
InvariantConsistent (*getM state′*) ∧
InvariantUniq (*getM state′*) ∧
InvariantWatchListsContainOnlyClausesFromF (*getWatchList
state′*) (*getF state′*) ∧
InvariantWatchListsUniq (*getWatchList state′*) ∧
InvariantWatchListsCharacterization (*getWatchList state′*) (*getWatch1*

state′) (*getWatch2 state′*) ∧
InvariantWatchesEl (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) ∧
InvariantWatchesDiffer (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) ∧
InvariantWatchCharacterization (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) (*getM state′*) ∧
InvariantConflictFlagCharacterization (*getConflictFlag state′*) (*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*
⟨*proof*⟩

lemma *InvariantEquivalentZLAfterInitialization*:
fixes *F0* :: *Formula*
shows
let state′ = (*initialize F0 initialState*) *in*
let F0′ = (*filter* (λ *c*. ¬ *clauseTautology c*) *F0*) *in*
(*getSATFlag state′* = *UNDEF* ∧ *InvariantEquivalentZL* (*getF state′*) (*getM state′*) *F0′*) ∨
(*getSATFlag state′* = *FALSE* ∧ ¬ *satisfiable F0′*)
⟨*proof*⟩

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 *clauseFalse c* (*elements* (*prefixToLevel* (*elementLevel l M*) *M*))
⟨*proof*⟩

lemma *InvariantNoDecisionsWhenConflictEnsuresCurrentLevelCl*:
assumes
InvariantNoDecisionsWhenConflict F M (*currentLevel M*)
clause el F
clauseFalse clause (*elements M*)
uniq (*elements M*)
currentLevel M > *0*
shows
clause ≠ [] ∧
(*let Cl* = *getLastAssertedLiteral* (*oppositeLiteralList clause*) (*elements M*) *in*
InvariantClCurrentLevel Cl M)
⟨*proof*⟩

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*⟩

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*⟩

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*))
getCn 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′*) ∧
InvariantUniqC (*getC state′*)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *ApplyExplainUIPPreservedVariables*:
assumes
applyExplainUIP-dom state
shows
let state′ = applyExplainUIP state in
(*getM state′ = getM state*) ∧
(*getF state′ = getF state*) ∧
(*getQ state′ = getQ state*) ∧
(*getWatch1 state′ = getWatch1 state*) ∧
(*getWatch2 state′ = getWatch2 state*) ∧
(*getWatchList state′ = getWatchList state*) ∧
(*getConflictFlag state′ = getConflictFlag state*) ∧
(*getConflictClause state′ = getConflictClause state*) ∧
(*getSATFlag state′ = getSATFlag state*) ∧
(*getReason state′ = getReason state*)
(**is** *let state′ = applyExplainUIP state in ?p state state′*)
⟨*proof*⟩

lemma *isUIPApplyExplainUIP*:
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*)
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*
isUIP (*opposite* (*getCl state′*)) (*getC state′*) (*getM state′*)
⟨*proof*⟩

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′*) ∧
InvariantClCurrentLevel (*getCl state′*) (*getM state′*) ∧
InvariantUniqC (*getC state′*)
⟨*proof*⟩

lemma *oneElementSetCharacterization*:
shows
(*set l* = {*a*}) = ((*remdups l*) = [*a*])
⟨*proof*⟩

lemma *uniqOneElementCharacterization*:
assumes
uniq l
shows
(*l* = [*a*]) = (*set l* = {*a*})
⟨*proof*⟩

lemma *isMinimalBackjumpLevelGetBackjumpLevel*:
assumes
InvariantUniq (*getM state*)

InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*)
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*)
InvariantCllCharacterization (*getCl state*) (*getCll 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
isMinimalBackjumpLevel (*getBackjumpLevel state*) (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
⟨*proof*⟩

lemma *applyLearnPreservedVariables*:
let state′ = *applyLearn state in*
getM state′ = *getM state* ∧
getQ state′ = *getQ state* ∧
getC state′ = *getC state* ∧
getCl state′ = *getCl state* ∧
getConflictFlag state′ = *getConflictFlag state* ∧
getConflictClause state′ = *getConflictClause state* ∧
getF state′ = (*if getC state* = [*opposite* (*getCl state*)] *then*
getF state
else
(*getF state* @ [*getC state*])
)
⟨*proof*⟩

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*⟩

lemma *InvariantCllCharacterizationAfterApplyLearn*:
assumes
InvariantUniq (*getM state*)
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*)
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*)
InvariantUniqC (*getC state*)
getConflictFlag state
shows
let state′ = *applyLearn state in*
InvariantCllCharacterization (*getCl state′*) (*getCll state′*) (*getC state′*) (*getM state′*)
⟨*proof*⟩

lemma *InvariantConflictClauseCharacterizationAfterApplyLearn*:
assumes
getConflictFlag state
InvariantConflictClauseCharacterization (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*)
shows
let state′ = *applyLearn state in*
InvariantConflictClauseCharacterization (*getConflictFlag state′*) (*getConflictClause state′*) (*getF state′*) (*getM state′*)
⟨*proof*⟩

lemma *InvariantGetReasonIsReasonAfterApplyLearn*:
assumes
InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM*

state) (*set* (*getQ state*))
shows
let state′ = applyLearn state in
InvariantGetReasonIsReason (*getReason state′*) (*getF state′*) (*getM state′*) (*set* (*getQ state′*))

⟨*proof*⟩

lemma *InvariantQCharacterizationAfterApplyLearn*:
assumes
getConflictFlag state
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
shows
let state′ = applyLearn state in
InvariantQCharacterization (*getConflictFlag state′*) (*getQ state′*) (*getF state′*) (*getM state′*)
⟨*proof*⟩

lemma *InvariantUniqQAfterApplyLearn*:
assumes
InvariantUniqQ (*getQ state*)
shows
let state′ = applyLearn state in
InvariantUniqQ (*getQ state′*)
⟨*proof*⟩

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′*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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
  InvariantCllCharacterization (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⟩

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
⟨proof⟩


lemma InvariantWatchCharacterizationInBackjumpPrefix:
assumes
 InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2
state) (getM state)

shows
  let l = getCl state in
   let level = getBackjumpLevel state in
```

let prefix = prefixToLevel level (getM state) in
let state′ = state(| getConflictFlag := False, getQ := [], getM := prefix |) in
InvariantWatchCharacterization (getF state′) (getWatch1 state′) (getWatch2 state′) (getM state′)
⟨*proof*⟩

lemma *InvariantConsistentAfterApplyBackjump*:
assumes
InvariantConsistent (getM state)
InvariantUniq (getM state)
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) **and**

getConflictFlag state
InvariantCFalse (getConflictFlag state) (getM state) (getC state) **and**
InvariantUniqC (getC state)
InvariantCEntailed (getConflictFlag state) F0 (getC state) **and**
InvariantClCharacterization (getCl state) (getC state) (getM state)
and
InvariantCllCharacterization (getCl state) (getCll 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*⟩

lemma *InvariantUniqAfterApplyBackjump*:
assumes
InvariantConsistent (getM state)
InvariantUniq (getM state)
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) **and**

getConflictFlag state
InvariantCFalse (getConflictFlag state) (getM state) (getC state) **and**
InvariantUniqC (getC state)
InvariantCEntailed (getConflictFlag state) F0 (getC state) **and**
InvariantClCharacterization (getCl state) (getC state) (getM state)
and

InvariantCllCharacterization (*getCl state*) (*getCll 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*
InvariantUniq (*getM state′*)
⟨*proof*⟩

lemma *WatchInvariantsAfterApplyBackjump*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*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**
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*) **and**
InvariantCllCharacterization (*getCl state*) (*getCll state*) (*getC state*) (*getM state*) **and**
InvariantClCurrentLevel (*getCl state*) (*getM state*)

isUIP (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
currentLevel (*getM state*) > *0*
shows
let state′ = (*applyBackjump state*) *in*
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'*)
(**is** *let state' = (applyBackjump state) in ?inv state'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

lemma *invariantQCharacterizationAfterApplyBackjump-1*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*) **and**
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*) **and**
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*) **and**

InvariantUniqC (*getC state*)
getC state = [*opposite* (*getCl state*)]
InvariantNoDecisionsWhenUnit (*getF state*) (*getM state*) (*currentLevel* (*getM state*))
InvariantNoDecisionsWhenConflict (*getF state*) (*getM state*) (*currentLevel* (*getM state*))

getConflictFlag state
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*)
InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*) **and**
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*)
and

InvariantCllCharacterization (*getCl state*) (*getCll 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*⟩

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 (*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 (*butlast* (*getF state*)) (*getM state*) (*currentLevel* (*getM 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
InvariantCllCharacterization (*getCl state*) (*getCll 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*⟩

lemma *InvariantConflictFlagCharacterizationAfterApplyBackjump-1*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*)
(*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **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
InvariantCllCharacterization (*getCl state*) (*getCll 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*⟩

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*) **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**
InvariantCllCharacterization (*getCl state*) (*getCll 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*⟩

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*⟩

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)
InvariantCllCharacterization (getCl state) (getCll state) (getC state) (getM state)
InvariantClCurrentLevel (getCl state) (getM state)
isUIP (opposite (getCl state)) (getC state) (getM state)
0 < currentLevel (getM state)
InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state))
getBackjumpLevel state > 0 ⟶ getF state ≠ [] ∧ last (getF state) = getC state
shows
let state′ = applyBackjump state in
InvariantGetReasonIsReason (getReason state′) (getF state′) (getM state′) (set (getQ state′))

⟨*proof*⟩

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**
InvariantCllCharacterization (*getCl state*) (*getCll 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*⟩

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
InvariantCllCharacterization (*getCl state*) (*getCll state*) (*getC state*)
(*getM state*) **and**
InvariantClCurrentLevel (*getCl state*) (*getM state*)

isUIP (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
currentLevel (*getM state*) > *0*
shows
let state′ = *applyBackjump state in*
InvariantNoDecisionsWhenConflict (*getF state′*) (*getM state′*)
(*currentLevel* (*getM state′*)) ∧
InvariantNoDecisionsWhenUnit (*getF state′*) (*getM state′*)
(*currentLevel* (*getM state′*))
⟨*proof*⟩

lemma *InvariantEquivalentZLAfterApplyBackjump*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*)
(*getF state*) **and**

getConflictFlag state
InvariantUniqC (*getC state*)
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) **and**
InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*) **and**
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*)
and
InvariantCllCharacterization (*getCl state*) (*getCll 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*
shows
let state′ = *applyBackjump state in*
InvariantEquivalentZL (*getF state′*) (*getM state′*) *F0*

⟨*proof*⟩

lemma *InvariantsVarsAfterApplyBackjump*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)

InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*)
(*getF state*) **and**

InvariantWatchListsUniq (*getWatchList state*)
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*) **and**

getConflictFlag state
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) **and**
InvariantUniqC (*getC state*) **and**
InvariantCEntailed (*getConflictFlag state*) *F0'* (*getC state*) **and**
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*)
and
InvariantCllCharacterization (*getCl state*) (*getCll state*) (*getC state*)
(*getM state*) **and**
InvariantClCurrentLevel (*getCl state*) (*getM state*)
InvariantEquivalentZL (*getF state*) (*getM state*) *F0'*

isUIP (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
currentLevel (*getM state*) > *0*

vars F0' ⊆ *vars F0*

InvariantVarsM (*getM state*) *F0 Vbl*
InvariantVarsF (*getF state*) *F0 Vbl*
InvariantVarsQ (*getQ state*) *F0 Vbl*
shows
let state' = *applyBackjump state in*
InvariantVarsM (*getM state'*) *F0 Vbl* ∧
InvariantVarsF (*getF state'*) *F0 Vbl* ∧
InvariantVarsQ (*getQ state'*) *F0 Vbl*

⟨*proof*⟩

end

theory *Decide*
imports *AssertLiteral*
begin

lemma *applyDecideEffect*:
assumes
$\neg$ *vars*(*elements* (*getM state*)) $\supseteq$ *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 $\notin$ *vars* (*elements* (*getM state*)) $\wedge$
var literal $\in$ *Vbl* $\wedge$
getM state′ = *getM state* @ [(*literal*, *True*)] $\wedge$
getF state′ = *getF state*
⟨*proof*⟩

lemma *InvariantConsistentAfterApplyDecide*:
assumes
$\neg$ *vars*(*elements* (*getM state*)) $\supseteq$ *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′*)
⟨*proof*⟩

lemma *InvariantUniqAfterApplyDecide*:
assumes
$\neg$ *vars*(*elements* (*getM state*)) $\supseteq$ *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′*)
⟨*proof*⟩

lemma *InvariantQCharacterizationAfterApplyDecide*:
assumes
$\neg$ *vars*(*elements* (*getM state*)) $\supseteq$ *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*⟩

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*⟩

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*⟩

lemma *InvariantsVarsAfterApplyDecide*:
assumes
¬ *vars* (*elements* (*getM state*)) ⊇ *Vbl*
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*)
InvariantWatchListsUniq (*getWatchList state*)
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)

InvariantVarsM (*getM state*) *F0 Vbl*
InvariantVarsF (*getF state*) *F0 Vbl*
getQ state = []
shows
let state′ = applyDecide state Vbl in
InvariantVarsM (*getM state′*) *F0 Vbl* ∧
InvariantVarsF (*getF state′*) *F0 Vbl* ∧
InvariantVarsQ (*getQ state′*) *F0 Vbl*
⟨*proof*⟩

end

theory *SolveLoop*
imports *UnitPropagate ConflictAnalysis Decide*
begin

lemma *soundnessForUNSAT*:
assumes
equivalentFormulae (*F* @ *val2form M*) *F0*
formulaFalse F M
shows
¬ *satisfiable F0*

⟨*proof*⟩

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 F* (*elements M*) **and** *vars* (*elements M*) ⊇ *Vbl*
shows *model* (*elements M*) *F0*
⟨*proof*⟩

definition
satFlagLessState = {(*state1*::*State*, *state2*::*State*). (*getSATFlag state1*) ≠ *UNDEF* ∧ (*getSATFlag state2*) = *UNDEF*}

lemma *wellFoundedSatFlagLessState*:
shows *wf satFlagLessState*
⟨*proof*⟩

definition
lexLessState1 Vbl = {(*state1*::*State*, *state2*::*State*).
getSATFlag state1 = *UNDEF* ∧ *getSATFlag state2* = *UNDEF* ∧
(*getM state1*, *getM state2*) ∈ *lexLessRestricted Vbl*
}

lemma *wellFoundedLexLessState1*:
assumes
finite Vbl
shows
wf (*lexLessState1 Vbl*)
⟨*proof*⟩

definition
terminationLessState1 Vbl = {(*state1*::*State*, *state2*::*State*).
(*state1*, *state2*) ∈ *satFlagLessState* ∨
(*state1*, *state2*) ∈ *lexLessState1 Vbl*}

lemma *wellFoundedTerminationLessState1*:
assumes *finite Vbl*
shows *wf* (*terminationLessState1 Vbl*)
⟨*proof*⟩

lemma *transTerminationLessState1*:
trans (*terminationLessState1 Vbl*)
⟨*proof*⟩

lemma *transTerminationLessState1I*:
assumes
(*x*, *y*) ∈ *terminationLessState1 Vbl*

$(y, z) \in terminationLessState1\ Vbl$
shows
$(x, z) \in terminationLessState1\ Vbl$
⟨*proof*⟩

lemma *TerminationLessAfterExhaustiveUnitPropagate*:
assumes
exhaustiveUnitPropagate-dom state
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*)
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*)
⟨*proof*⟩

lemma *InvariantsAfterSolveLoopBody*:
assumes
getSATFlag state = *UNDEF*
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2*

state) (*getM state*) **and**
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*)
(*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1
state*) (*getWatch2 state*) **and**
InvariantUniqQ (*getQ state*) **and**
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF
state*) (*getM state*) **and**
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF
state*) (*getM state*) **and**
InvariantNoDecisionsWhenConflict (*getF state*) (*getM state*) (*currentLevel
*(*getM state*)) **and**
InvariantNoDecisionsWhenUnit (*getF state*) (*getM state*) (*currentLevel
*(*getM state*)) **and**
InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM
state*) (*set* (*getQ state*)) **and**
InvariantEquivalentZL (*getF state*) (*getM state*) *F0′* **and**
InvariantConflictClauseCharacterization (*getConflictFlag state*) (*getConflictClause
state*) (*getF state*) (*getM state*) **and**
finite Vbl
vars F0′ ⊆ *vars F0*
vars F0 ⊆ *Vbl*
InvariantVarsM (*getM state*) *F0 Vbl*
InvariantVarsQ (*getQ state*) *F0 Vbl*
InvariantVarsF (*getF state*) *F0 Vbl*
shows
let state′ = *solve-loop-body state Vbl in*
(*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′*) (*getM state′*) ∧
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state′*)
(*getF state′*) ∧
InvariantWatchListsUniq (*getWatchList state′*) ∧
InvariantWatchListsCharacterization (*getWatchList state′*) (*getWatch1
state′*) (*getWatch2 state′*) ∧
InvariantQCharacterization (*getConflictFlag state′*) (*getQ state′*)
(*getF state′*) (*getM state′*) ∧
InvariantConflictFlagCharacterization (*getConflictFlag state′*) (*getF
state′*) (*getM state′*) ∧
InvariantConflictClauseCharacterization (*getConflictFlag state′*)
(*getConflictClause state′*) (*getF state′*) (*getM state′*) ∧
InvariantUniqQ (*getQ state′*)) ∧
(*InvariantNoDecisionsWhenConflict* (*getF state′*) (*getM state′*)

(*currentLevel* (*getM state'*)) ∧
InvariantNoDecisionsWhenUnit (*getF state'*) (*getM state'*) (*currentLevel* (*getM state'*))) ∧
InvariantEquivalentZL (*getF state'*) (*getM state'*) *F0'* ∧
InvariantGetReasonIsReason (*getReason state'*) (*getF state'*) (*getM state'*) (*set* (*getQ state'*)) ∧
InvariantVarsM (*getM state'*) *F0 Vbl* ∧
InvariantVarsQ (*getQ state'*) *F0 Vbl* ∧
InvariantVarsF (*getF state'*) *F0 Vbl* ∧
(*state'*, *state*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*) ∧
((*getSATFlag state'* = *FALSE* ⟶ ¬ *satisfiable F0'*) ∧
(*getSATFlag state'* = *TRUE* ⟶ *satisfiable F0'*))
(**is** *let state'* = *solve-loop-body state Vbl in ?inv' state'* ∧ *?inv''* *state'* ∧ -)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *SATFlagAfterSolveLoop*:
assumes
solve-loop-dom state Vbl
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*) **and**
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantUniqQ (*getQ state*) **and**
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*) **and**
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*) **and**
InvariantNoDecisionsWhenConflict (*getF state*) (*getM state*) (*currentLevel* (*getM state*)) **and**
InvariantNoDecisionsWhenUnit (*getF state*) (*getM state*) (*currentLevel* (*getM state*)) **and**
InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*)) **and**
getSATFlag state = *UNDEF* ⟶ *InvariantEquivalentZL* (*getF state*) (*getM state*) *F0′* **and**
InvariantConflictClauseCharacterization (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*)
getSATFlag state = *FALSE* ⟶ ¬ *satisfiable F0′*
getSATFlag state = *TRUE* ⟶ *satisfiable F0′*
finite Vbl
vars F0′ ⊆ *vars F0*
vars F0 ⊆ *Vbl*

InvariantVarsM (*getM state*) *F0 Vbl*
InvariantVarsF (*getF state*) *F0 Vbl*
InvariantVarsQ (*getQ state*) *F0 Vbl*
shows
let state′ = solve-loop state Vbl in
(*getSATFlag state′ = FALSE* ∧ ¬ *satisfiable F0′*) ∨ (*getSATFlag state′ = TRUE* ∧ *satisfiable F0′*)
⟨*proof*⟩

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*⟩

end

References

[1] S. Krstic and A. Goel. Architecting solvers for sat modulo theories: Nelson-oppen with dpll. In *FroCos*, pages 1–27, 2007.

[2] F. Maric. Formalization and implementation of modern sat solvers. *submitted to Journal of Automated Reasoning*, 2008.

[3] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, Nov. 2006.