Propositional Resolution and Prime Implicates Generation

Nicolas Peltier
Laboratory of Informatics of Grenoble/CNRS
University Grenoble Alps
April 20, 2020

Abstract

We provide formal proofs in Isabelle-HOL (using mostly structured Isar proofs) of the soundness and completeness of the Resolution rule in propositional logic. The completeness proofs take into account the usual redundancy elimination rules (namely tautology elimination and subsumption), and several refinements of the Resolution rule are considered: ordered resolution (with selection functions), positive and negative resolution, semantic resolution and unit resolution (the latter refinement is complete only for clause sets that are Horn-renamable). We also define a concrete procedure for computing saturated sets and establish its soundness and completeness. The clause sets are not assumed to be finite, so that the results can be applied to formulas obtained by grounding sets of first-order clauses (however, a total ordering among atoms is assumed to be given).

Next, we show that the unrestricted Resolution rule is deductive-complete, in the sense that it is able to generate all (prime) implicates of any set of propositional clauses (i.e., all entailment-minimal, non-valid, clausal consequences of the considered set). The generation of prime implicates is an important problem, with many applications in artificial intelligence and verification (for abductive reasoning, knowledge compilation, diagnosis, debugging etc.). We also show that implicates can be computed in an incremental way, by fixing an ordering among all the atoms and resolving upon these atoms one by one in the considered order (with no backtracking). This feature is critical for the efficient computation of prime implicates. Building on these results, we provide a procedure for computing such implicates and establish its soundness and completeness.

Contents

1 Syntax of Propositional Clausal Logic 2
2 Semantics 4
1 Syntax of Propositional Clausal Logic

We define the usual syntactic notions of clausal propositional logic. The set of atoms may be arbitrary (even uncountable), but a well-founded total order is assumed to be given.

code

theory Propositional-Resolution

imports Main

begin

locale propositional-atoms =
  fixes atom-ordering :: ('at × 'at) set
  assumes atom-ordering-wf : (wf atom-ordering)
  and atom-ordering-total: (\( \forall x y. (x \neq y \rightarrow ((x, y) \in atom-ordering \lor (y, x) \in atom-ordering)) \))

\[
\text{atom-ordering-trans: } \forall x y z. (x, y) \in \text{atom-ordering} \rightarrow (y, z) \in \text{atom-ordering} \\
\text{atom-ordering-irrefl: } \forall x y. (x, y) \in \text{atom-ordering} \rightarrow (y, x) \notin \text{atom-ordering}
\]

begin

Literals are defined as usual and clauses and formulas are considered as sets. Clause sets are not assumed to be finite (so that the results can be applied to sets of clauses obtained by grounding first-order clauses).

\text{datatype} 'a Literal = Pos 'a | Neg 'a

\text{definition} atoms = \{ x :: at. True \}

\text{fun} atom :: 'a Literal \Rightarrow 'a
\text{where}
(\text{atom} \ (\text{Pos} \ A)) = A |
(\text{atom} \ (\text{Neg} \ A)) = A

\text{fun} complement :: 'a Literal \Rightarrow 'a Literal
\text{where}
(\text{complement} \ (\text{Pos} \ A)) = (\text{Neg} \ A) |
(\text{complement} \ (\text{Neg} \ A)) = (\text{Pos} \ A)

\text{lemma} atom-property : A = \ (\text{atom} \ L) \implies (L = (\text{Pos} \ A) \lor L = (\text{Neg} \ A))
\text{by} (metis atom.elims)

\text{fun} positive :: 'a Literal \Rightarrow bool
\text{where}
(\text{positive} \ (\text{Pos} \ A)) = True |
(\text{positive} \ (\text{Neg} \ A)) = False

\text{fun} negative :: 'a Literal \Rightarrow bool
\text{where}
(\text{negative} \ (\text{Pos} \ A)) = False |
(\text{negative} \ (\text{Neg} \ A)) = True

\text{type-synonym} 'a Clause = 'a Literal set

\text{type-synonym} 'a Formula = 'a Clause set

Note that the clauses are not assumed to be finite (some of the properties below hold for infinite clauses).

The following functions return the set of atoms occurring in a clause or formula.

\text{fun} atoms-clause :: 'a Clause \Rightarrow 'a set
\text{where} atoms-clause C = \{ A. \ \exists L. \ L \in C \land A = \text{atom}(L) \}

\text{fun} atoms-formula :: 'a Formula \Rightarrow 'a set
\text{where} atoms-formula S = \{ A. \ \exists C. \ C \in S \land A = \text{atoms-clause}(C) \}

lemma atoms-formula-subset: \( S_1 \subseteq S_2 \implies \text{atoms-formula } S_1 \subseteq \text{atoms-formula } S_2 \)
by auto

lemma atoms-formula-union: \( \text{atoms-formula } (S_1 \cup S_2) = \text{atoms-formula } S_1 \cup \text{atoms-formula } S_2 \)
by auto

The following predicate is useful to state that every clause in a set fulfills some property.

definition all-fulfill :: \( 'a \text{ Clause} \Rightarrow \text{bool} \Rightarrow 'a \text{ Formula} \Rightarrow \text{bool} \)
where \( \text{all-fulfill } P S = (\forall C. C \in S \implies (P C)) \)

The order on atoms induces a (non total) order among literals:

fun literal-ordering :: \( 'a \text{ Literal} \Rightarrow 'a \text{ Literal} \Rightarrow \text{bool} \)
where \( \text{literal-ordering } L_1 L_2 = ((\text{atom } L_1, \text{atom } L_2) \in \text{atom-ordering}) \)

lemma literal-ordering-trans : 
assumes \( \text{literal-ordering } A B \)
assumes \( \text{literal-ordering } B C \)
shows \( \text{literal-ordering } A C \)
using \( \text{assms}(1) \text{ assms}(2) \text{ atom-ordering-trans } \text{literal-ordering}. \text{simps} \) by blast

definition strictly-maximal-literal :: \( 'a \text{ Clause} \Rightarrow 'a \text{ Literal} \Rightarrow \text{bool} \)
where \( \text{strictly-maximal-literal } S A \equiv (A \in S) \land (\forall B. (B \in S \land A \neq B) \implies (\text{literal-ordering } B A)) \)

2 Semantics

We define the notions of interpretation, satisfiability and entailment and establish some basic properties.

type-synonym \( 'a \text{ Interpretation} = 'a \text{ set} \)

fun validate-literal :: \( 'a \text{ Interpretation} \Rightarrow 'a \text{ Literal} \Rightarrow \text{bool} \) \( \text{infix } \mid \)
where \( \text{validate-literal } I (\text{Pos } A) = (A \in I) | \)
\( \text{validate-literal } I (\text{Neg } A) = (A \notin I) \)

fun validate-clause :: \( 'a \text{ Interpretation} \Rightarrow 'a \text{ Clause} \Rightarrow \text{bool} \) \( \text{infix } \mid \)
where \( \text{validate-clause } I C = (\exists L. (L \in C) \land (\text{validate-literal } I L)) \)

fun validate-formula :: \( 'a \text{ Interpretation} \Rightarrow 'a \text{ Formula} \Rightarrow \text{bool} \) \( \text{infix } \mid \)
where
(validate-formula I S) = (∀ C. (C ∈ S → (validate-clause I C)))

**definition** satisfiable :: 'at Formula ⇒ bool
**where**
(satisfiable S) ≡ (∃ I. (validate-formula I S))

We define the usual notions of entailment between clauses and formulas.

**definition** entails :: 'at Formula ⇒ 'at Clause ⇒ bool
**where**
(entails S C) ≡ (∀ I. (validate-formula I S) → (validate-clause I C))

**lemma** entails-member:
- assumes C ∈ S
- shows entails S C
**using** assms unfolding entails-def by simp

**definition** entails-formula :: 'at Formula ⇒ 'at Formula ⇒ bool
**where**
(entails-formula S1 S2) = (∀ C ∈ S2. (entails S1 C))

**definition** equivalent :: 'at Formula ⇒ 'at Formula ⇒ bool
**where**
(equivalent S1 S2) = (entails-formula S1 S2 ∧ entails-formula S2 S1)

**lemma** equivalent-symmetric:
equivalent S1 S2 =⇒ equivalent S2 S1
by (simp add: equivalent-def)

**lemma** entailment-implies-validity:
- assumes entails-formula S1 S2
- assumes validate-formula I S1
- shows validate-formula I S2
**using** assms entails-def entails-formula-def by auto

**lemma** validity-implies-entailment:
- assumes ∀ I. validate-formula I S1 → validate-formula I S2
- shows entails-formula S1 S2
by (meson assms entails-def entails-formula-def validate-formula.elims(2))

**lemma** entails-transitive:
- assumes entails-formula S1 S2
- assumes entails-formula S2 S3
- shows entails-formula S1 S3
by (meson assms entailment-implies-validity validity-implies-entailment)

**lemma** equivalent-transitive:
- assumes equivalent S1 S2
- assumes equivalent S2 S3
- shows equivalent S1 S3
**using** assms entails-transitive equivalent-def by auto

**lemma** entailment-subset:
assumes $S_2 \subseteq S_1$
shows entails-formula $S_1\ S_2$
proof
have $\forall L\ \La.\ L \notin \La \lor\ entails L\ L$
  by (meson entails-member)
thus thesis
  by (meson assms entails-formula-def rev-subsetD)
qed

lemma entailed-formula-entails-implicates:
assumes entails-formula $S_1\ S_2$
assumes entails $S_2\ C$
shows entails $S_1\ C$
using assms entailment-implies-validity entails-def by blast

3 Inference Rules

We first define an abstract notion of a binary inference rule.

type-synonym 'a BinaryRule = 'a Clause ⇒ 'a Clause ⇒ 'a Clause ⇒ bool
definition less-restrictive :: 'at BinaryRule ⇒ 'at BinaryRule ⇒ bool
where
  (less-restrictive R1 R2) = ($\forall P1\ P2\ C.\ (R2\ P1\ P2\ C) \rightarrow ((R1\ P1\ P2\ C) \lor (R1\ P2\ P1\ C)))$

The following functions allow to generate all the clauses that are deducible
from a given clause set (in one step).

fun all-deducible-clauses :: 'at BinaryRule ⇒ 'at Formula ⇒ 'at Formula
where all-deducible-clauses R S = { C. $\exists P1\ P2.\ P1 \in S \land P2 \in S \land (R\ P1\ P2\ C) }$

fun add-all-deducible-clauses :: 'at BinaryRule ⇒ 'at Formula ⇒ 'at Formula
where add-all-deducible-clauses R S = (S \cup all-deducible-clauses R S)
definition derived-clauses-are-finite :: 'at BinaryRule ⇒ bool
where derived-clauses-are-finite R
  = ($\forall P1\ P2\ C.\ (finite\ P1 \rightarrow finite\ P2 \rightarrow (R\ P1\ P2\ C) \rightarrow finite\ C))$

lemma less-restrictive-and-finite :
assumes less-restrictive R1 R2
assumes derived-clauses-are-finite R1
shows derived-clauses-are-finite R2
by (metis assms derived-clauses-are-finite-def less-restrictive-def)

We then define the unrestricted resolution rule and usual resolution refi-

ments.
3.1 Unrestricted Resolution

definition resolvent :: 'at BinaryRule
where
(resolvent P1 P2 C) ≡
(∃A. ((Pos A) ∈ P1 ∧ (Neg A) ∈ P2 ∧ (C = ((P1 − {Pos A})) ∪ (P2 − {Neg A}))))

For technical convience, we now introduce a slightly extended definition in which resolution upon a literal not occurring in the premises is allowed (the obtained resolvent is then redundant with the premises). If the atom is fixed then this version of the resolution rule can be turned into a total function.

fun resolvent-upon :: 'at Clause ⇒ 'at Clause ⇒ 'at ⇒ 'at Clause
where
(resolvent-upon P1 P2 A) =
((P1 − {Pos A}) ∪ (P2 − {Neg A}))

lemma resolvent-upon-is-resolvent :
assumes Pos A ∈ P1
assumes Neg A ∈ P2
shows resolvent P1 P2 (resolvent-upon P1 P2 A)
using assms unfolding resolvent-def by auto

lemma resolvent-is-resolvent-upon :
assumes resolvent P1 P2 C
shows ∃A. C = resolvent-upon P1 P2 A
using assms unfolding resolvent-def by auto

lemma resolvent-is-finite :
shows derived-clauses-are-finite resolvent
proof (rule ccontr)
  assume ¬derived-clauses-are-finite resolvent
  then have ∃P1 P2 C. ¬(resolvent P1 P2 C → finite P1 → finite P2 → finite C)
  unfolding derived-clauses-are-finite-def by blast
  then obtain P1 P2 C where resolvent P1 P2 C finite P1 finite P2 and ¬finite C by blast
  from (resolvent P1 P2 C) (finite P1) (finite P2) and (¬finite C) show False
  unfolding resolvent-def using finite-Diff and finite-Union by auto
qed

In the next subsections we introduce various resolution refinements and show that they are more restrictive than unrestricted resolution.

3.2 Ordered Resolution

In the first refinement, resolution is only allowed on maximal literals.

definition ordered-resolvent :: 'at Clause ⇒ 'at Clause ⇒ 'at Clause ⇒ bool
where  
\[(\text{ordered-resolvent } P_1 P_2 C) \equiv (\exists A. ((C = (P_1 - \{\text{Pos } A\}) \cup (P_2 - \{\text{Neg } A\})) \land (\text{strictly-maximal-literal } P_1 (\text{Pos } A)) \land (\text{strictly-maximal-literal } P_2 (\text{Neg } A))))\]

We now show that the maximal literal of the resolvent is always smaller than those of the premises.

**Lemma:** \(\text{resolution-and-max-literal} :\)
- **Assumes:** \(R = \text{resolvent-upon } P_1 P_2 A\)
- **Assumes:** \(\text{strictly-maximal-literal } P_1 (\text{Pos } A)\)
- **Assumes:** \(\text{strictly-maximal-literal } P_2 (\text{Neg } A)\)
- **Shows:** \((\text{atom } M, A) \in \text{atom-ordering}\)

**Proof:**
- Obtain \(MA\) where \(M = (\text{Pos } MA) \lor M = (\text{Neg } MA)\) using \(\text{Literal.exhaust}\) \([\text{of } M']\) by \(\text{auto}\)
  - Hence \(MA = \text{atom } M\) by \(\text{auto}\)
- From \((\text{strictly-maximal-literal } R M)\) and \((R = \text{resolvent-upon } P_1 P_2 A)\)
  - Have \(M \in P_1 - \{\text{Pos } A\} \lor M \in P_2 - \{\text{Neg } A\}\)
  - Unfolding \(\text{strictly-maximal-literal-def}\) by \(\text{auto}\)
  - Hence \((MA, A) \in \text{atom-ordering}\) by \(\text{auto}\)

Next
- Assume \(M \in P_2 - \{\text{Neg } A\}\)
- From \((M \in P_2 - \{\text{Neg } A\})\) and \((\text{strictly-maximal-literal } P_2 (\text{Neg } A)\)
  - Have \(\text{literal-ordering } M (\text{Neg } A)\) by \(\text{auto simp only: strictly-maximal-literal-def}\)
  - From \((M = \text{Pos } MA \lor M = \text{Neg } MA)\) and \(\text{litorder } M (\text{Pos } A)\)
  - Show \((MA, A) \in \text{atom-ordering}\) by \(\text{auto}\)

qed
- From this and \(MA = \text{atom } M\) show \(\text{thesis}\) by \(\text{auto}\)

**3.3 Ordered Resolution with Selection**

In the next restriction strategy, some negative literals are selected with highest priority for applying the resolution rule, regardless of the ordering. Relaxed ordering restrictions also apply.

**Definition:** \((\text{selected-part } \text{Sel } C) = \{ \text{L, } L \in C \land (\exists A \in \text{Sel. } L = (\text{Neg } A)) \}\)

**Definition:** \(\text{ordered-set-resolvent} :: \text{‘at set } \Rightarrow \text{‘at Clause } \Rightarrow \text{‘at Clause } \Rightarrow \text{‘at Clause } \Rightarrow \text{bool}\)
where

\[(\text{ordered-sel-resolvent } \text{Sel } P1 P2 C) \equiv\]

\[(\exists A. ((C = (\{ P1 \} - \{ \text{Pos } A \}) \cup (\{ P2 \} - \{ \text{Neg } A \}))) \land
   (\text{strictly-maximal-literal } P1 \text{ (Pos } A) \land ((\text{selected-part Sel } P1) = \{\}) \land
   ((\text{strictly-maximal-literal } P2 \text{ (Neg } A) \land ((\text{selected-part Sel } P2) = \{\}) \lor
   (\text{strictly-maximal-literal (selected-part Sel } P2 \text{ (Neg } A))))))\]

lemma ordered-resolvent-is-resolvent : less-restrictive resolvent ordered-resolvent
using less-restrictive-def ordered-resolvent-def resolvent-upon-is-resolvent strictly-maximal-literal-def
by auto

The next lemma states that ordered resolution with selection coincides with ordered resolution if the selected part is empty.

lemma ordered-sel-resolvent-is-ordered-resolvent :
assumes ordered-resolvent P1 P2 C
assumes selected-part Sel P1 = \{\}
assumes selected-part Sel P2 = \{\}
shows ordered-sel-resolvent Sel P1 P2 C
using assms ordered-resolvent-def ordered-sel-resolvent-def by auto

lemma ordered-resolvent-upon-is-resolvent :
assumes strictly-maximal-literal P1 \text{ (Pos } A)
assumes strictly-maximal-literal P2 \text{ (Neg } A)
shows ordered-resolvent P1 P2 \text{ (resolvent-upon } P1 P2 A)
using assms ordered-resolvent-def by auto

3.4 Semantic Resolution

In this strategy, resolution is applied only if one parent is false in some (fixed) interpretation. Note that ordering restrictions still apply, although they are relaxed.

definition validated-part :: 'at set ⇒ 'at Clause ⇒ 'at Clause
where (validated-part I C) = \{ L. L ∈ C \land (validate-literal I L) \}

definition ordered-model-resolvent ::
'at Interpretation ⇒ 'at Clause ⇒ 'at Clause ⇒ 'at Clause ⇒ bool
where
(ordered-model-resolvent I P1 P2 C) =
(\exists L. (C = (\{ P1 \} - \{ L \} \cup (\{ P2 \} - \{ \text{complement } L \}))) \land
((\text{validated-part } I \text{ P1}) = \{\}) \land (\text{strictly-maximal-literal } P1 L) \land
(\text{strictly-maximal-literal (validated-part } I \text{ P2) (complement } L)))

lemma ordered-model-resolvent-is-resolvent : less-restrictive resolvent (ordered-model-resolvent I)
proof (rule ccontr)
assume ¬ less-restrictive resolvent (ordered-model-resolvent I)
then obtain P1 P2 C where ordered-model-resolvent I P1 P2 C and ¬resolvent P1 P2 C

9
and \( \neg \text{resolvent } P_2 \ P_1 \ C \) unfolding less-restrictive-def by auto

from \( \langle \text{ordered-model-resolvent } I \ P_1 \ P_2 \ C \rangle \) obtain \( L \)

where strictly-maximal-literal \( P_1 \ L \)

and strictly-maximal-literal \( \langle \text{validated-part } I \ P_2 \rangle \) (complement \( L \))

and \( C = (P_1 - \{ L \}) \cup (P_2 - \{ \text{complement } L \}) \)

using ordered-model-resolvent-def \{of \( I \ P_1 \ P_2 \ C \)\} by auto

from \( \langle \text{strictly-maximal-literal } P_1 \ L \rangle \) have \( L \in P_1 \) by (simp only: strictly-maximal-literal-def)

from \( \langle \text{strictly-maximal-literal } (\text{validated-part } I \ P_2) \rangle \) (complement \( L \)) have (complement \( L \)) \( \in P_2 \) by (auto simp only: strictly-maximal-literal-def validated-part-def)

obtain \( A \) where \( L = \text{Pos } A \lor L = \text{Neg } A \) using Literal.exhaust [of \( L \)] by auto

from this and \( \langle C = (P_1 - \{ L \}) \cup (P_2 - \{ \text{complement } L \}) \rangle \) and \( \langle L \in P_1 \rangle \) and \( \langle \text{complement } L \rangle \in P_2 \) have \( \text{resolvent } P_1 \ P_2 \ C \lor \text{resolvent } P_2 \ P_1 \ C \) unfolding resolvent-def by auto

from this and \( \langle \neg \text{resolvent } P_2 \ P_1 \ C \rangle \) and \( \langle \neg \text{resolvent } P_1 \ P_2 \ C \rangle \) show False by auto

qed

3.5 Unit Resolution

Resolution is applied only if one parent is unit (this restriction is incomplete).

definition Unit : 'a at Clause \Rightarrow bool

where \( \langle \text{Unit } C \rangle = ((\text{card } C) = 1) \)

definition unit-resolvent : 'a at BinaryRule

where \( \langle \text{unit-resolvent } P_1 \ P_2 \ C \rangle = ((\exists \ L. \ (C = (P_1 - \{ L \}) \cup (P_2 - \{ \text{complement } L \}))) \land \langle L \in P_1 \rangle \land \langle \text{complement } L \rangle \in P_2 \land \text{Unit } P_1 \)

lemma unit-resolvent-is-resolvent : less-restrictive resolvent unit-resolvent

proof (rule ccontr)

assume \( \neg \text{less-restrictive resolvent } \text{unit-resolvent} \)

then obtain \( P_1 \ P_2 \ C \) where unit-resolvent \( P_1 \ P_2 \ C \) and \( \neg \text{resolvent } P_1 \ P_2 \ C \) and \( \neg \text{resolvent } P_2 \ P_1 \ C \) unfolding less-restrictive-def by auto

from \( \langle \text{unit-resolvent } P_1 \ P_2 \ C \rangle \) obtain \( L \) where \( L \in P_1 \) and complement \( L \) \( \in P_2 \)

and \( C = (P_1 - \{ L \}) \cup (P_2 - \{ \text{complement } L \}) \)

using unit-resolvent-def [of \( P_1 \ P_2 \ C \)] by auto

obtain \( A \) where \( L = \text{Pos } A \lor L = \text{Neg } A \) using Literal.exhaust [of \( L \)] by auto

from this and \( \langle C = (P_1 - \{ L \}) \cup (P_2 - \{ \text{complement } L \} ) \rangle \) and \( \langle L \in P_1 \rangle \) and \( \langle \text{complement } L \rangle \in P_2 \) have \( \text{resolvent } P_1 \ P_2 \ C \lor \text{resolvent } P_2 \ P_1 \ C \) unfolding resolvent-def by auto

from this and \( \langle \neg \text{resolvent } P_2 \ P_1 \ C \rangle \) and \( \langle \neg \text{resolvent } P_1 \ P_2 \ C \rangle \) show False by auto

qed
3.6 Positive and Negative Resolution

Resolution is applied only if one parent is positive (resp. negative). Again, relaxed ordering restrictions apply.

definition positive-part :: 'at Clause ⇒ 'at Clause
where
(positive-part C) = \{ L. (\exists A. L = Pos A) ∧ L ∈ C \}

definition negative-part :: 'at Clause ⇒ 'at Clause
where
(negative-part C) = \{ L. (\exists A. L = Neg A) ∧ L ∈ C \}

lemma decomposition-clause-pos-neg :
C = (negative-part C) ∪ (positive-part C)

proof
show C ⊆ (negative-part C) ∪ (positive-part C)
proof
fix x assume x ∈ C
obtain A where x = Pos A ∨ x = Neg A using Literal.exhaust \{ x \} by auto
show x ∈ (negative-part C) ∪ (positive-part C)
proof cases
assume x = Pos A
from this and \{ x ∈ C \} have x ∈ positive-part C unfolding positive-part-def by auto
then show x ∈ (negative-part C) ∪ (positive-part C) by auto
qed
next
assume x ≠ Pos A
from this and \{ x = Pos A ∨ x = Neg A \} have x = Neg A by auto
from this and \{ x ∈ C \} have x ∈ negative-part C unfolding negative-part-def by auto
then show x ∈ (negative-part C) ∪ (positive-part C) by auto
qed
qed

next
show (negative-part C) ∪ (positive-part C) ⊆ C unfolding negative-part-def
and positive-part-def by auto
qed

definition ordered-positive-resolvent :: 'at Clause ⇒ 'at Clause ⇒ 'at Clause ⇒ bool

where
(ordered-positive-resolvent P1 P2 C) =
(\exists L. (C = (P1 − \{ L \} ∪ (P2 − \{ \text{complement} L \}))) ∧
((negative-part P1) = \{\} ∧ (strictly-maximal-literal P1 L)) ∧ (strictly-maximal-literal (negative-part P2) (complement L)))

definition ordered-negative-resolvent :: 'at Clause ⇒ 'at Clause ⇒ 'at Clause ⇒ bool

where
\[
(\text{ordered-negative-resolvent } P_1 P_2 C) = \\
(\exists L. (C = (P_1 - \{ L \} \cup (P_2 - \{ \text{complement } L \}))) \land \\
((\text{positive-part } P_1) = \{ \} \land (\text{strictly-maximal-literal } P_1 L)) \\
\land (\text{strictly-maximal-literal } (\text{positive-part } P_2) (\text{complement } L)))
\]

**lemma positive-resolvent-is-resolvent**: less-restrictive resolvent ordered-positive-resolvent

**proof** (rule ccontr)

assume \( \neg \text{less-restrictive resolvent ordered-positive-resolvent} \)

then obtain \( P_1 P_2 C \) where ordered-positive-resolvent \( P_1 P_2 C \) and \( \neg \text{resolvent } P_1 P_2 C \)

and \( \neg \text{resolvent } P_2 P_1 C \) unfolding less-restrictive-def by auto

from ordered-positive-resolvent \( P_1 P_2 C \) obtain \( L \)

where strictly-maximal-literal \( P_1 L \)

and strictly-maximal-literal (negative-part \( P_2) \) (complement \( L \))

and \( C = (P_1 - \{ L \}) \cup (P_2 - \{ \text{complement } L \}) \)

using ordered-positive-resolvent-def [of \( P_1 P_2 C \) by auto

from strictly-maximal-literal \( P_1 L \) have \( L \in P_1 \) unfolding strictly-maximal-literal-def by auto

from strictly-maximal-literal (negative-part \( P_2) \) (complement \( L \)) have \( \text{complement } L \in P_2 \)

unfolding strictly-maximal-literal-def negative-part-def by auto

obtain \( A \) where \( L = \text{Pos } A \lor L = \text{Neg } A \) using Literal.exhaust [of \( L \) by auto

from this and \( C = (P_1 - \{ L \}) \cup (P_2 - \{ \text{complement } L \}) \) and \( \text{complement } L \in P_1 \)

and \( \text{complement } L \in P_2 \)

have resolvent \( P_1 P_2 C \lor \text{resolvent } P_2 P_1 C \) unfolding resolvent-def by auto

from this and \( \neg (\text{resolvent } P_2 P_1 C) \) and \( \neg (\text{resolvent } P_1 P_2 C) \) show False by auto

qed

**lemma negative-resolvent-is-resolvent**: less-restrictive resolvent ordered-negative-resolvent

**proof** (rule ccontr)

assume \( \neg \text{less-restrictive resolvent ordered-negative-resolvent} \)

then obtain \( P_1 P_2 C \) where (ordered-negative-resolvent \( P_1 P_2 C \) and \( \neg (\text{resolvent } P_1 P_2 C) \)

and \( \neg (\text{resolvent } P_2 P_1 C) \) unfolding less-restrictive-def by auto

from ordered-negative-resolvent \( P_1 P_2 C \) obtain \( L \) where strictly-maximal-literal \( P_1 L \)

and \( C = (P_1 - \{ L \}) \cup (P_2 - \{ \text{complement } L \}) \)

using ordered-negative-resolvent-def [of \( P_1 P_2 C \) by auto

from strictly-maximal-literal \( P_1 L \) have \( L \in P_1 \) unfolding strictly-maximal-literal-def by auto

from strictly-maximal-literal (positive-part \( P_2) \) (complement \( L \)) have \( \text{complement } L \in P_2 \)

unfolding strictly-maximal-literal-def positive-part-def by auto

obtain \( A \) where \( L = \text{Pos } A \lor L = \text{Neg } A \) using Literal.exhaust [of \( L \) by auto

from this and \( C = (P_1 - \{ L \}) \cup (P_2 - \{ \text{complement } L \}) \) and \( \text{complement } L \in P_1 \)

and \( \text{complement } L \in P_2 \)

have resolvent \( P_1 P_2 C \lor \text{resolvent } P_2 P_1 C \) unfolding resolvent-def by auto

\[12\]
from this and \(\langle \neg \text{resolvent } P2 \ P1 \ C \rangle\) and \(\langle \neg \text{resolvent } P1 \ P2 \ C \rangle\) show False by auto
qed

4 Redundancy Elimination Rules

We define the usual redundancy elimination rules.

definition tautology :: 'a Clause ⇒ bool
where
(tautology C) ≡ (\(\exists\ A.\ (\text{Pos } A \in C \land \text{Neg } A \in C)\))

definition subsumes :: 'a Clause ⇒ 'a Clause ⇒ bool
where
(subsumes C D) ≡ (C ⊆ D)

definition redundant :: 'a Clause ⇒ 'a Formula ⇒ bool
where
redundant C S = ((tautology C) ∨ (\(\exists\ D.\ (D \in S \land \text{subsumes } D C)\)))

definition strictly-redundant :: 'a Clause ⇒ 'a Formula ⇒ bool
where
strictly-redundant C S = ((tautology C) ∨ (\(\exists\ D.\ (D \in S \land (D \subset C)\))))

definition simplify :: 'at Formula ⇒ 'at Formula
where
simplify S = { C. C ∈ S \& ¬strictly-redundant C S }  

We first establish some basic syntactic properties.

lemma tautology-monotonous : (tautology C) ⇒ (C ⊆ D) ⇒ (tautology D)
unfolding tautology-def by auto

lemma simplify-involutive:
  shows simplify (simplify S) = (simplify S)
proof –
  show ?thesis unfolding simplify-def strictly-redundant-def by auto
qed

lemma simplify-finite:
  assumes all-fulfill finite S
  shows all-fulfill finite (simplify S)
using assms all-fulfill-def simplify-def by auto

lemma atoms-formula-simplify:
  shows atoms-formula (simplify S) ⊆ atoms-formula S
unfolding simplify-def using atoms-formula-subset by auto

lemma subsumption-preserves-redundancy :
  assumes redundant C S
assumes subsumes C D
shows redundant D S
using assms tautology-monotonous unfolding redundant-def subsumes-def by blast

lemma subsumption-and-max-literal :
assumes subsumes C1 C2
assumes strictly-maximal-literal C1 L1
assumes strictly-maximal-literal C2 L2
assumes A1 = atom L1
assumes A2 = atom L2
shows (A1 = A2) ∨ (A1, A2) ∈ atom-ordering
proof –
  from ⟨A1 = atom L1⟩ have L1 = (Pos A1) ∨ L1 = (Neg A1) by (rule atom-property)
  from ⟨A2 = atom L2⟩ have L2 = (Pos A2) ∨ L2 = (Neg A2) by (rule atom-property)
  unfolding strictly-maximal-literal-def subsumes-def by auto
  from ⟨strictly-maximal-literal C2 L2⟩ and ⟨L1 ∈ C2⟩ have L1 = L2 ∨ literal-ordering L1 L2
  unfolding strictly-maximal-literal-def by auto
thus ?thesis
proof
  assume L1 = L2
  from ⟨l1 = l2⟩ and ⟨A1 = atom L1⟩ and ⟨A2 = atom L2⟩ show ?thesis by auto
next
  assume literal-ordering L1 L2
  from ⟨literal-ordering L1 L2⟩ and ⟨l1 = (Pos A1) ∨ l1 = (Neg A1)⟩
  and ⟨l2 = (Pos A2) ∨ l2 = (Neg A2)⟩
  show ?thesis by auto
qed
qed

lemma superset-preserves-redundancy:
assumes redundant C S
assumes S ⊆ S’
shows redundant C S’
using assms unfolding redundant-def by blast

lemma superset-preserves-strict-redundancy:
assumes strictly-redundant C S
assumes S ⊆ SS
shows strictly-redundant C SS
using assms unfolding strictly-redundant-def by blast

The following lemmas relate the above notions with that of semantic entailment and thus establish the soundness of redundancy elimination rules.

lemma tautologies-are-valid :
assumes tautology C
shows validate-clause I C
by (meson assms tautology-def validate-clause.simps validate-literal.simps(1)
validate-literal.simps(2))

lemma subsumption-and-semantics :
  assumes subsames C D
  assumes validate-clause I C
  shows validate-clause I D
using assms unfolding subsumes-def by auto

lemma redundancy-and-semantics :
  assumes redundant C S
  assumes validate-formula I S
  shows validate-clause I C
by
(meson assms redundant-def subsumption-and-semantics tautologies-are-valid validate-formula.elims)

lemma redundancy-implies-entailment:
  assumes redundant C S
  shows entails S C
using assms entails-def redundancy-and-semantics by auto

lemma simplify-and-membership :
  assumes all-fulfill finite S
  assumes T = simplify S
  assumes C ∈ S
  shows redundant C T
proof −
{fix n
  have ∀ C. card C ≤ n ⟹ C ∈ S ⟹ redundant C T (is ?P n)
proof (induction n)
  show ?P 0
  proof ((rule allI),(rule impl)+)
    fix C assume card C ≤ 0 and C ∈ S
    from ⊢ card C ≤ 0 and C ∈ S and (all-fulfill finite S) have C = {} using card-0-eq
    unfolding all-fulfill-def by auto
    then have ∼ strictly-redundant C S unfolding strictly-redundant-def
    tautology-def by auto
    from this and (C ∈ S) and (T = simplify S) have C ∈ T using simplify-def
    by auto
    from this show redundant C T unfolding redundant-def subsumes-def by auto
  qed
  next
  fix n assume ?P n
  show ?P (Suc n)
proof ((rule allI),(rule impI)+)
fix C assume card C ≤ (Suc n) and C ∈ S
show redundant C T
proof (rule ccontr)
assume ¬redundant C T
from this have C ∉ T unfolding redundant-def subsumes-def by auto
from this and (T = simplify S) and (C ∈ S) have strictly-redundant C
S unfolding simplify-def strictly-redundant-def by auto
from this and (¬redundant C T) obtain D where D ∈ S and D ⊂ C
C unfolding redundant-def strictly-redundant-def by auto
from (D ⊂ C) and (C ∈ S) and (all-fulfill finite S) have card D < card C
C unfolding all-fulfill-def
using psubset-card mono by auto
from this and (card C ≤ (Suc n)) have card D ≤ n by auto
from this and (?P n) and (D ∈ S) have redundant D T by auto
show False proof cases
assume tautology D
from this and (D ⊂ C) have tautology C unfolding tautology-def by auto
then have redundant C T unfolding redundant-def by auto
from this and (¬redundant C T) show False by auto
next
assume ¬tautology D
from this and (¬redundant D T) obtain E where E ∈ T and E ⊆ D
unfolding redundant-def subsumes-def by auto
from this and (D ⊂ C) have E ⊆ C by auto
from this and (E ∈ T) and (¬redundant C T) show False unfolding redundant-def and subsumes-def by auto
qed
qed
qed
qed
}
from this and (C ∈ S) show ?thesis by auto
qed

lemma simplify-preserves-redundancy:
assumes all-fulfill finite S
assumes redundant C S
shows redundant C (simplify S)
by (meson assms redundant-def simplify-and-membership subsumption-preserves-redundancy)

lemma simplify-preserves-strict-redundancy:
assumes all-fulfill finite S
assumes strictly-redundant C S
shows strictly-redundant C (simplify S)
proof ((cases tautology C),(auto simp add: strictly-redundant-def)[1])
next
  assume ¬tautology C
  from this and assms(2) obtain D where D ⊆ C and D ∈ S unfolding strictly-redundant-def by auto
  from ⟨D ∈ S⟩ have redundant D S unfolding redundant-def subsumes-def by auto
  from assms(1) this have redundant D (simplify S) using simplify-preserves-redundancy
  unfolding subsumes-def by auto
  from ¬tautology C; and ⟨D ⊆ C⟩ have ¬tautology D unfolding tautology-def
  by auto
  from this and ⟨redundant D (simplify S)⟩ obtain E where E ∈ simplify S
  and subsumes E D unfolding redundant-def by auto
  from ⟨subsumes E D⟩ and ⟨D ⊆ C⟩ have E ⊆ C unfolding subsumes-def by auto
  from this and ⟨E ∈ simplify S; show strictly-redundant C (simplify S) unfolding strictly-redundant-def by auto
qed

lemma simplify-preserves-semantic :
  assumes T = simplify S
  assumes all-fulfill finite S
  shows validate-formula I S ←→ validate-formula I T
by (metis (mono-tags, lifting) assms mem-Collect-eq redundancy-and-semantics simplify-and-membership simplify-def validate-formula.simps)

lemma simplify-preserves-equivalence :
  assumes T = simplify S
  assumes all-fulfill finite S
  shows equivalent S T
using assms equivalent-def simplify-preserves-semantic validity-implies-entailment
by auto

After simplification, the formula contains no strictly redundant clause:
definition non-redundant :: 'a at Formula ⇒ bool
  where non-redundant S = (∀ C. (C ∈ S −→ ¬strictly-redundant C S))

lemma simplify-non-redundant:
  shows non-redundant (simplify S)
by (simp add: non-redundant-def simplify-def strictly-redundant-def)

lemma deducible-clause-preserve-redundancy:
  assumes redundant C S
  shows redundant C (add-all-deducible-clauses R S)
using assms superset-preserves-redundancy by fastforce
5 Renaming

A renaming is a function changing the sign of some literals. We show that this operation preserves most of the previous syntactic and semantic notions.

definition rename-literal :: 'at set ⇒ 'at Literal ⇒ 'at Literal
  where rename-literal A L = (if ((atom L) ∈ A) then (complement L) else L)

definition rename-clause :: 'at set ⇒ 'at Clause ⇒ 'at Clause
  where rename-clause A C = { L. ∃ LL. LL ∈ C ∧ L = (rename-literal A LL)}

definition rename-formula :: 'at set ⇒ 'at Formula ⇒ 'at Formula
  where rename-formula A S = { C. ∃ CC. CC ∈ S ∧ C = (rename-clause A CC)}

lemma inverse-renaming : (rename-literal A (rename-literal A L)) = L
  proof
    obtain A where at: L = (Pos A) ∨ L = (Neg A) using Literal.exhaust [of L ]
    by auto
    from at show ?thesis unfolding rename-literal-def by auto
  qed

lemma inverse-clause-renaming : (rename-clause A (rename-clause A L)) = L
  proof
    show ?thesis using inverse-renaming unfolding rename-clause-def by auto
  qed

lemma inverse-formula-renaming : rename-formula A (rename-formula A L) = L
  proof
    show ?thesis using inverse-clause-renaming unfolding rename-formula-def by auto
  qed

lemma renaming-preserves-cardinality :
  card (rename-clause A C) = card C
  proof
    have im: rename-clause A C = (rename-literal A) C unfolding rename-clause-def
    by auto
    have inj-on (rename-literal A) C by (metis inj-on1 inverse-renaming)
    from this and im show ?thesis using card-image by auto
  qed

lemma renaming-preserves-literal-order :
  assumes literal-ordering L1 L2
  shows literal-ordering (rename-literal A L1) (rename-literal A L2)
  proof
    obtain A2 where at2: L2 = (Pos A2) ∨ L2 = (Neg A2) using Literal.exhaust [of L2 ] by auto
    from assms and at1 and at2 show ?thesis unfolding rename-literal-def by
auto
qed

lemma inverse-renaming-preserves-literal-order :
  assumes literal-ordering (rename-literal A L1) (rename-literal A L2)
  shows literal-ordering L1 L2
by (metis assms inverse-renaming renaming-preserves-literal-order)

lemma renaming-is-injective :
  assumes rename-literal A L1 = rename-literal A L2
  shows L1 = L2
by (metis (no-types) assms inverse-renaming)

lemma renaming-preserves-strictly-maximal-literal :
  assumes strictly-maximal-literal C L
  shows strictly-maximal-literal (rename-clause A C) (rename-literal A L)
proof -
  from assms have (L ∈ C) and Lismax: (∀ B. (B ∈ C ∧ L ≠ B) → (literal-ordering B L))
  unfolding strictly-maximal-literal-def by auto
  from ⟨L ∈ C⟩ have (rename-literal A L) ∈ (rename-clause A C)
  unfolding rename-literal-def and rename-clause-def by auto
  have ∀ B. (B ∈ rename-clause A C → rename-literal A L ≠ B
    →→ literal-ordering B (rename-literal A L))
  proof (rule)+
    fix B assume B ∈ rename-clause A C and rename-literal A L ≠ B
    from ⟨B ∈ rename-clause A C⟩ obtain B′ where B′ ∈ C and B = rename-literal A B′
    unfolding rename-clause-def by auto
    have rename-literal A L ≠ rename-literal A B′ by auto
    hence L ≠ B′ by auto
    from this and ⟨B′ ∈ C⟩ and Lismax have literal-ordering B′ L by auto
    from this and ⟨B = (rename-literal A B′)⟩
    show literal-ordering B (rename-literal A L) using renaming-preserves-literal-order
  by auto
  qed
  from this and (rename-literal A L) ∈ (rename-clause A C): show thesis
  unfolding strictly-maximal-literal-def by auto
qed

lemma renaming-and-selected-part :
  selected-part UNIV C = rename-clause Sel (validated-part Sel (rename-clause Sel C))
proof
  show selected-part UNIV C ⊆ rename-clause Sel (validated-part Sel (rename-clause Sel C))
  proof
fix $x$ assume $x \in \text{selected-part } \text{UNIV } C$

show $x \in \text{rename-clause } \text{Sel} \left( \text{validated-part } \text{Sel} \left( \text{rename-clause } C \right) \right)$

proof

from $\langle x \in \text{selected-part } \text{UNIV } C \rangle$ obtain $A$ where $x = \text{Neg } A$ and $x \in C$

unfolding \text{selected-part-def} by auto

from $\langle x \in C \rangle$ have $\text{rename-literal } \text{Sel} \left( x \in \text{rename-clause } C \right)$

unfolding \text{rename-clause-def} by blast

show $x \in \text{rename-clause } \text{Sel} \left( \text{validated-part } \text{Sel} \left( \text{rename-clause } C \right) \right)$

proof cases

assume $A \in \text{Sel}$

from this and $\langle x = \text{Neg } A \rangle$ have $\text{rename-literal } \text{Sel} \left( x \in \text{rename-clause } C \right)$

unfolding \text{rename-literal-def} by auto

from this and $(A \in \text{Sel})$ have $\text{validate-literal } \text{Sel} \left( \text{rename-literal } \text{Sel} \left( x \right) \right)$ by auto

def

from this and $(A \notin \text{Sel})$ have $\text{rename-literal } \text{Sel} \left( x \in \text{rename-clause } C \right)$

unfolding \text{rename-literal-def} by auto

from this and $(A \notin \text{Sel})$ have $\text{validate-literal } \text{Sel} \left( \text{rename-literal } \text{Sel} \left( x \right) \right)$ by auto

next

assume $A \notin \text{Sel}$

from this and $\langle x = \text{Neg } A \rangle$ have $\text{rename-literal } \text{Sel} \left( x \in \text{rename-clause } C \right)$

unfolding \text{rename-literal-def} by auto

from this and $(A \notin \text{Sel})$ have $\text{validate-literal } \text{Sel} \left( \text{rename-literal } \text{Sel} \left( x \right) \right)$ by auto

qed

qed

next

show $\text{rename-clause } \text{Sel} \left( \text{validated-part } \text{Sel} \left( \text{rename-clause } C \right) \right) \subseteq \left( \text{selected-part } \text{UNIV } C \right)$

proof

fix $x$

assume $x \in \text{rename-clause } \text{Sel} \left( \text{validated-part } \text{Sel} \left( \text{rename-clause } C \right) \right)$

from this obtain $y$ where $y \in \text{validated-part } \text{Sel} \left( \text{rename-clause } C \right)$

and $x = \text{rename-literal } \text{Sel} \left( y \right)$

unfolding \text{rename-clause-def} \text{validated-part-def} by auto

from $\langle y \in \text{validated-part } \text{Sel} \left( \text{rename-clause } C \right) \rangle$ have $y \in \text{rename-clause } C$ and $\text{validate-literal } \text{Sel} \left( y \right)$ unfolding \text{validated-part-def} by auto

from $\langle y \in \text{rename-clause } C \rangle$ obtain $z$ where $z \in C$ and $y = \text{rename-literal } \text{Sel} \left( z \right)$

unfolding \text{rename-clause-def} by auto

obtain $A$ where $zA: z = \text{Pos } A \lor \text{Neg } A$ using \text{Literal.exhaust} [of $z$] by
auto

show \( x \in \text{selected-part \( UNIV \) C} \)
proof cases
  assume \( A \in \text{Sel} \)
  from this and \( zA \) and \( (y = \text{rename-literal Sel } z) \) have \( y = \text{complement } z \)
  using rename-literal-def by auto
  from this and \( (A \in \text{Sel} \) and \( zA \) and \( (\text{validate-literal Sel } y) \) have \( y = \text{Pos} \)\( A \)
and \( z = \text{Neg } A \) by auto
  from this and \( (A \in \text{Sel} \) and \( (x = \text{rename-literal Sel } y) \) have \( x = \text{Neg} \)\( A \)
  unfolding rename-literal-def by auto
  from this and \( (z \in C) \) and \( (z = \text{Neg } A) \) show \( x \in \text{selected-part \( UNIV \) C} \)
  unfolding selected-part-def by auto
next
  assume \( A \notin \text{Sel} \)
  from this and \( zA \) and \( (y = \text{rename-literal Sel } z) \) have \( y = z \)
  using rename-literal-def by auto
  from this and \( (A \notin \text{Sel} \) and \( zA \) and \( (\text{validate-literal Sel } y) \) have \( y = \text{Neg} \)\( A \)
and \( z = \text{Neg } A \) by auto
  from this and \( (A \notin \text{Sel} \) and \( (x = \text{rename-literal Sel } y) \) have \( x = \text{Neg} \)\( A \)
  unfolding rename-literal-def by auto
  from this and \( (z \in C) \) and \( (z = \text{Neg } A) \) show \( x \in \text{selected-part \( UNIV \) C} \)
  unfolding selected-part-def by auto
qed
qed

lemma renaming-preserves-tautology:
  assumes tautology \( C \)
  shows tautology \( (\text{rename-clause Sel } C) \)
proof –
  from assms obtain \( A \) where \( \text{Pos } A \in C \) and \( \text{Neg } A \in C \) unfolding tautology-def
  by auto
  from \( \langle \text{Pos } A \in C \rangle \) have \( \text{rename-literal Sel } (\text{Pos } A) \in \text{rename-clause Sel } C \)
  unfolding rename-clause-def by auto
  from \( \langle \text{Neg } A \in C \rangle \) have \( \text{rename-literal Sel } (\text{Neg } A) \in \text{rename-clause Sel } C \)
  unfolding rename-clause-def by auto
  show \( \text{thesis} \)
proof cases
  assume \( A \in \text{Sel} \)
  from this have \( \text{rename-literal Sel } (\text{Pos } A) = \text{Neg } A \)
  and \( \text{rename-literal Sel } (\text{Neg } A) = (\text{Pos } A) \)
  unfolding rename-literal-def by auto
  from \( \langle \text{rename-literal Sel } (\text{Pos } A) = (\text{Neg } A) \rangle \) and \( \langle \text{rename-literal Sel } (\text{Neg } A) \rangle \) = \( (\text{Pos } A) \)
  and \( \langle \text{rename-literal Sel } (\text{Pos } A) \rangle \in \langle \text{rename-clause Sel } C \rangle \)
  and \( \langle \text{rename-literal Sel } (\text{Neg } A) \rangle \in \langle \text{rename-clause Sel } C \rangle \)
  show tautology \( (\text{rename-clause Sel } C) \) unfolding tautology-def by auto

21
next
  assume $A \notin \text{Sel}$
  from this have rename-literal $\text{Sel} (\text{Pos } A) = \text{Pos } A$ and rename-literal $\text{Sel} (\text{Neg } A) = (\text{Neg } A)$
    unfolding rename-literal-def by auto
  from $\text{rename-literal } \text{Sel} (\text{Pos } A) = \text{Pos } A$ and $\text{rename-literal } \text{Sel} (\text{Neg } A) = (\text{Neg } A)$
    and $\text{rename-literal } \text{Sel} (\text{Pos } A) \in \text{rename-clause } \text{Sel } C$
    and $\text{rename-literal } \text{Sel} (\text{Neg } A) \in \text{rename-clause } \text{Sel } C$
    show tautology (rename-clause $\text{Sel } C$) unfolding tautology-def by auto
qeda
\( \subseteq (\text{rename-clause } \text{Sel} (\text{rename-clause } \text{Sel } \text{C})) - \{ \text{rename-literal } \text{Sel} (\text{rename-literal } \text{Sel } \text{L}) \} \)

using renaming-set-minus-subset by auto
from this
have rename-clause Sel ( (rename-clause Sel C) - { (rename-literal Sel L) })
\( \subseteq (C - \{L\}) \)
using inverse-renaming inverse-clause-renaming by auto
from this
have rename-clause Sel (rename-clause Sel ( (rename-clause Sel C) - { (rename-literal Sel L) }))
\( \subseteq (\text{rename-clause } \text{Sel} (C - \{L\})) \)
using rename-clause-def by auto
from this
have rename-clause Sel (rename-clause Sel C) - { (rename-literal Sel L) } \( \subseteq \) rename-clause Sel (C - {L})
using inverse-renaming inverse-clause-renaming by auto
qed
qed

definition rename-interpretation :: \`at set \Rightarrow \`at Interpretation \Rightarrow \`at Interpretation
where
\text{rename-interpretation } \text{Sel } \text{I} = \{ \text{A. } (\text{A} \in \text{I} \wedge \text{A} \notin \text{Sel}) \} \cup \{ \text{A. } (\text{A} \notin \text{I} \wedge \text{A} \in \text{Sel}) \} \)

lemma renaming-preserves-semantic :
assumes validate-literal \text{I} \text{L}
shows validate-literal (\text{rename-interpretation } \text{Sel } \text{I}) (\text{rename-literal Sel } \text{L})
proof −
let \(?J = \text{rename-interpretation } \text{Sel } \text{I}\)
obtain \text{A} where \text{L} = \text{Pos } \text{A} \lor \text{L} = \text{Neg } \text{A} using \text{Literal.exhaust [of L]} by auto
from \(\text{L} = \text{Pos } \text{A} \lor \text{L} = \text{Neg } \text{A}\) have \text{atom } \text{L} = \text{A} by auto
show \(?\text{thesis}\)
proof cases
assume \text{A} \in \text{Sel}
from this and \(\text{atom } \text{L} = \text{A}\) have \text{rename-literal Sel } \text{L} = \text{complement } \text{L}
unfolding rename-literal-def by auto
show \(?\text{thesis}\)
proof cases
assume \text{L} = \text{Pos } \text{A}
from this and \(\text{validate-literal } \text{I} \text{L}\) have \text{A} \in \text{I} by auto
from this and \(\text{A} \in \text{Sel}\) have \text{A} \notin \(?\text{J}\) unfolding rename-interpretation-def
by blast
from this and \(\text{L} = \text{Pos } \text{A}\) and \(\text{rename-literal Sel } \text{L} = \text{complement } \text{L}\)
show \(?\text{thesis}\) by auto
next
assume \text{L} \neq \text{Pos } \text{A}
from this and \(\text{L} = \text{Pos } \text{A} \lor \text{L} = \text{Neg } \text{A}\) have \text{L} = \text{Neg } \text{A} by auto
from this and \(\text{validate-literal } \text{I} \text{L}\) have \text{A} \notin \text{I} by auto
from this and \(\text{A} \in \text{Sel}\) have \text{A} \notin \(?\text{J}\) unfolding rename-interpretation-def
by blast

23
from this and ⟨L = Neg A⟩ and ⟨rename-literal Sel L = complement L⟩
show ?thesis by auto
qed

next
assume A ∉ Sel
from this and ⟨atom L = A⟩ have rename-literal Sel L = L
  unfolding rename-literal-def by auto
show ?thesis
proof cases
  assume L = Pos A
  from this and ⟨validate-literal I L⟩ have A ∈ I by auto
  from this and ⟨A ∉ Sel⟩ have A ∈ ?J unfolding rename-interpretation-def
  by blast
  from this and ⟨L = Pos A⟩ and ⟨rename-literal Sel L = L⟩ show ?thesis
by auto
  next
    assume L ≠ Pos A
    from this and ⟨L = Pos A ∨ L = Neg A⟩ have L = Neg A by auto
    from this and ⟨validate-literal I L⟩ have A ∉ I by auto
    from this and ⟨A ∉ Sel⟩ have A ∉ ?J unfolding rename-interpretation-def
    by blast
    from this and ⟨L = Neg A⟩ and ⟨rename-literal Sel L = L⟩ show ?thesis
by auto
qed
qed

lemma renaming-preserves-satisfiability:
  assumes satisfiable S
  shows satisfiable (rename-formula Sel S)
proof –
  from assms obtain I where validate-formula I S unfolding satisfiable-def by auto
  let ?J = rename-interpretation Sel I
  have validate-formula ?J (rename-formula Sel S)
  proof (rule ccontr)
    assume ¬validate-formula ?J (rename-formula Sel S)
    then obtain C where C ∈ S and ¬(validate-clause ?J (rename-clause Sel C))
    unfolding rename-formula-def by auto
    from ⟨C ∈ S⟩ and ⟨validate-formula I S⟩ obtain L where L ∈ C
    and validate-literal I L by auto
    from ⟨validate-literal I L⟩ have validate-literal ?J (rename-literal Sel L)
      using renaming-preservation-semantic by auto
    from this and ⟨L ∈ C⟩ and ¬validate-clause ?J (rename-clause Sel C) show False
    unfolding rename-clause-def by auto
  qed
  from this show ?thesis unfolding satisfiable-def by auto
qed

lemma renaming-preserves-subsumption:
  assumes subsumes C D
  shows subsumes (rename-clause Sel C) (rename-clause Sel D)
  using assms unfolding subsumes-def rename-clause-def by auto

6 Soundness

In this section we prove that all the rules introduced in the previous section
are sound. We first introduce an abstract notion of soundness.

definition Sound :: 'at BinaryRule ⇒ bool
where
  (Sound Rule) ≡ ∀ I P1 P2 C. (Rule P1 P2 C → (validate-clause I P1) →
  (validate-clause I P2) → (validate-clause I C))

lemma soundness-and-entailment :
  assumes Sound Rule
  assumes Rule P1 P2 C
  assumes P1 ∈ S
  assumes P2 ∈ S
  shows entails S C
  using Sound-def assms entails-def by auto

lemma all-deducible-sound:
  assumes Sound R
  shows entails-formula S (all-deducible-clauses R S)
  proof (rule contr)
    assume ¬entails-formula S (all-deducible-clauses R S)
    then obtain C where C ∈ all-deducible-clauses R S and ¬ entails S C
      unfolding entails-formula-def by auto
    from ⟨C ∈ all-deducible-clauses R S⟩ obtain P1 P2 where R P1 P2 C and P1 ∈ S and P2 ∈ S
      by auto
    from ⟨R P1 P2 C; and assms(1) and (P1 ∈ S) and (P2 ∈ S) and (¬ entails S C)⟩
      show False using soundness-and-entailment by auto
  qed

lemma add-all-deducible-sound:
  assumes Sound R
  shows entails-formula S (add-all-deducible-clauses R S)
  by (metis UnE add-all-deducible-clauses.simps all-deducible-sound assms entails-formula-def entails-member)

If a rule is more restrictive than a sound rule then it is necessarily sound.

lemma less-restrictive-correct:
assumes less-restrictive \( R1 \) \( R2 \)
assumes Sound \( R1 \)
shows Sound \( R2 \)
using assms unfolding less-restrictive-def Sound-def by blast

We finally establish usual concrete soundness results.

**Theorem resolution-is-correct:**

\((\text{Sound resolvent})\)

**Proof (rule ccontr)**

assume \( \neg (\text{Sound resolvent}) \)
then obtain \( I \ P1 \ P2 \ C \) where

\( \text{resolvent } P1 \ P2 \ C \) validate-clause \( I \ P1 \) validate-clause \( I \ P2 \) and \( \neg \text{validate-clause } I \ C \)

unfolding Sound-def by blast
from \( \langle \text{resolvent } P1 \ P2 \ C \rangle \) obtain \( A \) where

\( (\text{Pos } A) \in P1 \) and \( (\text{Neg } A) \in P2 \) and \( C = ( (P1 - \{\text{Pos } A\}) \cup (P2 - \{\text{Neg } A\}) \) 

unfolding resolvent-def by auto
show False
proof cases
assume \( A \in I \)

hence \( \neg \text{validate-literal } I (\text{Neg } A) \) by auto
from \( \langle \neg \text{validate-literal } I (\text{Neg } A) \rangle \) and \( \langle \text{validate-clause } I \ P2 \rangle \) have \( \text{validate-clause } I (P2 - \{\text{Neg } A\}) \) by auto
from \( \langle \text{validate-clause } I (P2 - \{\text{Neg } A\}) \rangle \) and \( \langle (P1 - \{\text{Pos } A\}) \cup (P2 - \{\text{Neg } A\}) \rangle \) show False by auto

next
assume \( A \notin I \)

hence \( \neg \text{validate-literal } I (\text{Pos } A) \) by auto
from \( \langle \neg \text{validate-literal } I (\text{Pos } A) \rangle \) and \( \langle \text{validate-clause } I \ P1 \rangle \) have \( \text{validate-clause } I (P1 - \{\text{Pos } A\}) \) by auto
from \( \langle \text{validate-clause } I (P1 - \{\text{Pos } A\}) \rangle \) and \( \langle (P1 - \{\text{Pos } A\}) \cup (P2 - \{\text{Neg } A\}) \rangle \) show False by auto
qed

**Theorem ordered-resolution-correct**

\( \text{Sound ordered-resolvent} \)

using resolution-is-correct and ordered-resolvent-is-resolvent less-restrictive-correct by auto

**Theorem ordered-model-resolution-correct**

\( \text{Sound } \langle \text{ordered-model-resolvent } I \rangle \)

using resolution-is-correct ordered-model-resolvent-is-resolvent less-restrictive-correct by auto

**Theorem ordered-positive-resolution-correct**

\( \text{Sound ordered-positive-resolvent} \)

using less-restrictive-correct positive-resolvent-is-resolvent resolution-is-correct by
theorem ordered-negative-resolution-correct : Sound ordered-negative-resolvent using less-restrictive-correct negative-resolvent-is-resolvent resolution-is-correct by auto

theorem unit-resolvent-correct : Sound unit-resolvent using less-restrictive-correct resolution-is-correct unit-resolvent-is-resolvent by auto

7 Refutational Completeness

In this section we establish the refutational completeness of the previous inference rules (under adequate restrictions for the unit resolution rule). Completeness is proven w.r.t. redundancy elimination rules, i.e., we show that every saturated unsatisfiable clause set contains the empty clause.

We first introduce an abstract notion of saturation.

definition saturated-binary-rule :: 'a BinaryRule ⇒ 'a Formula ⇒ bool where
(saturated-binary-rule Rule S) ≡ (∀ P1 P2 C. (((P1 ∈ S) ∧ (P2 ∈ S) ∧ (Rule P1 P2 C))) −→ redundant C S)

definition Complete :: 'at BinaryRule ⇒ bool where
(Complete Rule) = (∀ S. ((saturated-binary-rule Rule S) −→ (all-fulfill finite S) −→ (S S ∈ (S) −→ satisfiable S))

If a set of clauses is saturated under some rule then it is necessarily saturated under more restrictive rules, which entails that if a rule is less restrictive than a complete rule then it is also complete.

lemma less-restrictive-saturated:
assumes less-restrictive R1 R2
assumes saturated-binary-rule R1 S
shows saturated-binary-rule R2 S
using assms unfolding less-restrictive-def Complete-def saturated-binary-rule-def by blast

lemma less-restrictive-complete:
assumes less-restrictive R1 R2
assumes Complete R2
shows Complete R1
using assms less-restrictive-saturated Complete-def by auto

27
7.1 Ordered Resolution

We define a function associating every set of clauses $S$ with a “canonic” interpretation constructed from $S$. If $S$ is saturated under ordered resolution and does not contain the empty clause then the interpretation is a model of $S$. The interpretation is defined by mean of an auxiliary function that maps every atom to a function indicating whether the atom occurs in the interpretation corresponding to a given clause set. The auxiliary function is defined by induction on the set of atoms.

function \texttt{canonic-int-fun-ordered} :: 'at \Rightarrow ('at Formula \Rightarrow bool)
where
\begin{align*}
\texttt{canonic-int-fun-ordered} \ A = \\
\quad (\lambda S. (\exists C. (C \in S) \land (\text{strictly-maximal-literal} \ C \ (\text{Pos} \ A)) \\
\quad \land (\forall B. (\text{Pos} \ B \in C \rightarrow (B, A) \in \text{atom-ordering} \rightarrow (\neg (\text{canonic-int-fun-ordered} \ B) \ S)))) \\
\quad \land (\forall B. (\text{Neg} \ B \in C \rightarrow (B, A) \in \text{atom-ordering} \rightarrow ((\text{canonic-int-fun-ordered} \ B) \ S))))
\end{align*}
by auto
termination apply (relation atom-ordering)
by auto
(simp add: atom-ordering-af)

definition \texttt{canonic-int-ordered} :: 'at Formula \Rightarrow 'at Interpretation
where
\begin{align*}
\texttt{canonic-int-ordered} \ S = \{ A. ((\text{canonic-int-fun-ordered} \ A) \ S) \}
\end{align*}

We first prove that the canonic interpretation validates every clause having a positive strictly maximal literal

lemma \texttt{int-validate-cl-with-pos-max} :
assumes \texttt{strictly-maximal-literal} \ C \ (\text{Pos} \ A)
assumes \texttt{C} \in \texttt{S}
shows \texttt{validate-clause} \ (\texttt{canonic-int-ordered} \ S) \ C

proof cases
assume \texttt{c1}: (\forall B. (\text{Pos} \ B \in C \rightarrow (B, A) \in \text{atom-ordering} \\
\rightarrow (\neg (\text{canonic-int-fun-ordered} \ B) \ S)))
show \ ?thesis
proof cases
assume \texttt{c2}: (\forall B. (\text{Neg} \ B \in C \rightarrow (B, A) \in \text{atom-ordering} \\
\rightarrow ((\text{canonic-int-fun-ordered} \ B) \ S)))
have ((\text{canonic-int-fun-ordered} \ A) \ S)
proof (rule ccontr)
assume \texttt{e} : (\forall C. (C \in S) \land (\text{strictly-maximal-literal} \ C \ (\text{Pos} \ A)) \\
\land (\forall B. (\text{Pos} \ B \in C \rightarrow (B, A) \in \text{atom-ordering} \rightarrow (\neg (\text{canonic-int-fun-ordered} \ B) \ S))) \\
\land (\forall B. (\text{Neg} \ B \in C \rightarrow (B, A) \in \text{atom-ordering} \rightarrow ((\text{canonic-int-fun-ordered} \ B) \ S))))
by ((simp only:canonic-int-fun-ordered.simps[of A]), blast)
from \( e \) and \( c_1 \) and \( c_2 \) and \( \langle (C \in S) \land ((\text{strictly-maximal-literal } C \in \text{Pos } A)) \rangle \)

\[ \text{show } \text{False by blast } \]

\[ \text{qed} \]

from \( \langle (\text{canonic-int-fun-ordered } A \in \text{canonic-int-ordered } S) \rangle \)
have \( A \in (\text{canonic-int-ordered } S) \)

\[ \text{unfolding canonic-int-ordered-def by blast } \]

show \( ?\text{thesis} \)

\[ \text{unfolding strictly-maximal-literal-def by auto } \]

next

assume \( \neg c_2 \)

\[ \neg (\forall B. (\text{Neg } B \in C \rightarrow (B, A) \in \text{atom-ordering} \rightarrow ((\text{canonic-int-fun-ordered } B) \in S))) \]

from \( \neg c_2 \) obtain \( B \) where \( \text{Neg } B \in C \) and \( \neg ((\text{canonic-int-fun-ordered } B) \in S) \)

\[ \text{by blast } \]

from \( (\neg (\text{canonic-int-fun-ordered } B) \in S) \)
have \( B \notin (\text{canonic-int-ordered } S) \)

\[ \text{unfolding canonic-int-ordered-def by blast } \]

with \( (\text{Neg } B \in C) \)
show \( ?\text{thesis} \) by auto

\[ \text{qed} \]

next

assume \( \neg c_1 \):

\[ \neg (\forall B. (\text{Pos } B \in C \rightarrow (B, A) \in \text{atom-ordering} \rightarrow ((\neg (\text{canonic-int-fun-ordered } B) \in S))) \]

from \( \neg c_1 \) obtain \( B \) where \( \text{Pos } B \in C \) and \( (\text{canonic-int-fun-ordered } B) \in S) \)

\[ \text{by blast } \]

from \( ((\text{canonic-int-fun-ordered } B) \in S) \)
have \( B \in (\text{canonic-int-ordered } S) \)

\[ \text{unfolding canonic-int-ordered-def by blast } \]

with \( (\text{Pos } B \in C) \)
show \( ?\text{thesis} \) by auto

\[ \text{qed} \]

lemma \( \text{strictly-maximal-literal-exists} : \)

\[ \forall C. ((\text{finite } C) \land (\text{card } C) = n \land n \neq 0 \land \neg (\text{tautology } C))) \rightarrow (\exists A. (\text{strictly-maximal-literal } C \in A)) \] is \( ?P \ n) \]

proof (induction \( n \))

show \( (?P \ 0) \) by auto

next

fix \( n \) assume \( ?P \ n \)

show \( ?P \ (\text{Suc } n) \)

proof

fix \( C \)

show \( (\text{finite } C \land \text{card } C = \text{Suc } n \land \text{Suc } n \neq 0 \land \neg (\text{tautology } C)) \rightarrow (\exists A. (\text{strictly-maximal-literal } C \in A)) \)

proof

assume \( \text{finite } C \land \text{card } C = \text{Suc } n \land \text{Suc } n \neq 0 \land \neg (\text{tautology } C) \)

hence \( (\text{finite } C) \land (\text{card } C) = (\text{Suc } n) \land \neg (\text{tautology } C) \) by

\[ \text{auto} \]
have $C \neq \{\}$
proof
  assume $C = \{\}$
  from $\langle \text{finite } C \rangle$ and $\langle C = \{\} \rangle$ have $\text{card } C = 0$ using $\text{card-0-eq}$ by auto
qed
then obtain $L$ where $L \in C$ by auto
from $\langle \neg \text{tautology } C \rangle$ have $\neg \text{tautology } (C - \{ L \})$ using $\text{tautology-monotonous}$
  by auto
from $\langle L \in C \rangle$ and $\langle \text{finite } C \rangle$ have $\text{Suc } (\text{card } (C - \{ L \})) = \text{card } C$ using $\text{card-Suc-Diff1}$ by metis
with $\langle \text{card } C = \text{Suc } n \rangle$ have $\text{card } (C - \{ L \}) = n$ by auto
show $\exists A. (\text{strictly-maximal-literal } C A)$
proof cases
  assume $\text{card } C = 1$
  from this and $\langle \text{card } C = \text{Suc } n \rangle$ have $n = 0$ by auto
  from this and $\langle \text{finite } C \rangle$ and $\langle \text{card } (C - \{ L \}) = n \rangle$ have $C - \{ L \} = \{\}$
    using $\text{card-0-eq}$ by auto
from this and $\langle L \in C \rangle$ show $?\text{thesis}$ unfolding $\text{strictly-maximal-literal-def}$
  by auto
next
assume $\text{card } C \neq 1$
from $\langle \text{finite } C \rangle$ have $\text{finite } (C - \{ L \})$ by auto
from $\langle \text{Suc } (\text{card } (C - \{ L \})) = \text{card } C; \text{and } \langle \text{card } C \neq 1 \rangle \rangle$
  and $\langle \text{card } (C - \{ L \}) = n \rangle$ have $n \neq 0$ by auto
from this and $\langle \text{finite } (C - \{ L \}); \text{and } \langle \text{card } (C - \{ L \}) = n \rangle \rangle$
  and $\langle \neg \text{tautology } (C - \{ L \}); \text{and } \langle ?P n \rangle \rangle$ obtain $A$ where $\text{strictly-maximal-literal } (C - \{ L \}) A$ by metis
show $\exists M. \text{strictly-maximal-literal } C M$
proof cases
  assume $\langle \text{atom } L, \text{atom } A \rangle \in \text{atom-ordering}$
    from this have $\text{literal-ordering } L A$ by auto
    from this and $\langle \text{strictly-maximal-literal } (C - \{ L \}) A \rangle$
      have $\text{strictly-maximal-literal } C A$
    unfolding $\text{strictly-maximal-literal-def}$ by blast
    thus $\langle \text{thesis }$ by auto
next
assume $\langle \text{atom } L, \text{atom } A \rangle \notin \text{atom-ordering}$
    have $\langle \text{atom-property } [\text{of } (\text{atom } L)]; \text{auto} \rangle$
      by $\langle \text{rule atom-property } [\text{of } (\text{atom } L)]; \text{auto} \rangle$
    have $\langle \text{atom-property } [\text{of } (\text{atom } A)]; \text{auto} \rangle$
      by $\langle \text{rule atom-property } [\text{of } (\text{atom } A)]; \text{auto} \rangle$
    from l-cases and a-cases and $\langle \text{strictly-maximal-literal } (C - \{ L \}) A \rangle$
      and $\langle \neg \text{tautology } C; \text{and } \langle L \in C \rangle \rangle$
have atom $L \neq atom A$

unfolding strictly-maximal-literal-def and tautology-def by auto

from this and $(atom L, atom A) \notin atom-ordering$ and

atom-ordering-total

have $(atom A, atom L) \in atom-ordering$ by auto

hence literal-ordering $A \lessdot L$ by auto

from this and $⟨L \in C⟩$ and $(strictly-maximal-literal (C - \{L\}) A)$ and

literal-ordering-trans

have $(strictly-maximal-literal C \lessdot L)$ unfolding strictly-maximal-literal-def

thus $\negthesis$ by blast

qed

qed

We then deduce that all clauses are validated.

**Lemma** canonic-int-validates-all-clauses:

assumes saturated-binary-rule ordered-resolvent $S$

assumes all-fulfill finite $S$

assumes $\{\} \notin S$

assumes $C \in S$

shows validate-clause $(canonic-int-ordered S) C$

**Proof** cases

assume $(tautology C)$

thus $\negthesis$ using tautologies-are-valid $[of \ C \ (canonic-int-ordered S)]$ by auto

next

assume $\neg tautology C$

from $(all-fulfill finite S) \land (C \in S)$ have finite $C$ using all-fulfill-def by auto

from $\{\} \notin S$ and $(C \in S) \land (finite C)$ have $card C \neq 0$ using card-0-eq by auto

from $(\neg tautology C) \land (finite C)$ and $(\neg card C \neq 0)$ obtain $L$

where $(strictly-maximal-literal C \lessdot L)$ using strictly-maximal-literal-exists by blast

obtain $A$ where $A = (atom L)$ by auto

have inductive-lemma:

$\forall C. ((C \in S) \rightarrow (strictly-maximal-literal C \lessdot L)) \rightarrow (A = (atom L))$

$\rightarrow (validate-clause (canonic-int-ordered S) C))$ (is $(\neg Q A)$)

**Proof** ($(rule wf-induct [of atom-ordering $\neg Q A$]),(rule atom-ordering-wf))

next

fix $x$

assume hyp-induct: $\forall y. (y,x) \in atom-ordering \rightarrow (\neg Q y)$

show $\neg Q x$

**Proof** (rule)+

fix $C \lessdot L$ assume $C \in S$ strictly-maximal-literal $C \lessdot L \ x = (atom L)$
show validate-clause (canonic-int-ordered S) C
proof cases
  assume L = Pos x
  from ⟨L = Pos x⟩ and ⟨strictly-maximal-literal C L⟩ and ⟨C ∈ S⟩
  show validate-clause (canonic-int-ordered S) C
  using int-validate-cl-with-pos-max by auto
next
  assume L ≠ Pos x
  have L = (Neg x) using ⟨L ≠ Pos x⟩ ⟨x = atom L⟩ atom-property by fastforce
show (validate-clause (canonic-int-ordered S) C)
proof (rule ccontr)
  assume ¬ (validate-clause (canonic-int-ordered S) C)
  from ⟨L = (Neg x)⟩ and ⟨(strictly-maximal-literal C L)⟩
  and ⟨¬ (validate-clause (canonic-int-ordered S) C)⟩
  have x ∈ canonic-int-ordered S unfolding strictly-maximal-literal-def
by auto
from ⟨x ∈ canonic-int-ordered S⟩ have ⟨canonic-int-fun-ordered x⟩ S
  unfolding canonic-int-ordered-def by blast
and ⟨∃ C. (C ∈ S) ∧ (strictly-maximal-literal C (Pos x))⟩
  ∧ ⟨∀ B. (Pos B ∈ C → (B, x) ∈ atom-ordering → (¬(canonic-int-fun-ordered B) S))⟩
  ∧ ⟨∀ B. (Neg B ∈ C → (B, x) ∈ atom-ordering → ((canonic-int-fun-ordered B) S))⟩
by (simp only: canonic-int-fun-ordered.simps [of x])
then obtain D
where ⟨D ∈ S⟩ and ⟨strictly-maximal-literal D (Pos x)⟩
and a: ⟨∀ B. (Pos B ∈ D → (B, x) ∈ atom-ordering → (¬(canonic-int-fun-ordered B) S))⟩
and b: ⟨∀ B. (Neg B ∈ D → (B, x) ∈ atom-ordering → ((canonic-int-fun-ordered B) S))⟩
by blast
obtain R where R = (resolvent-upon D C x) by auto
from ⟨R = resolvent-upon D C x⟩ and ⟨strictly-maximal-literal D (Pos x)⟩
and ⟨strictly-maximal-literal C L⟩ and ⟨L = (Neg x)⟩ have resolvent D C R
unfolding strictly-maximal-literal-def using resolvent-upon-is-resolvent
by auto
from ⟨R = resolvent-upon D C x⟩ and ⟨strictly-maximal-literal D (Pos x)⟩
and ⟨strictly-maximal-literal C L⟩ and ⟨L = Neg x⟩
have ordered-resolvent D C R
using ordered-resolvent-upon-is-resolvent by auto
have ¬ validate-clause (canonic-int-ordered S) R
proof
assume validate-clause (canonic-int-ordered S) R
from validate-clause (canonic-int-ordered S) R obtain M
  where (M ∈ R) and validate-literal (canonic-int-ordered S) M
  by auto
from (M ∈ R) and (R = resolvent-upon D C x)
  have (M ∈ (D − { Pos x })) ∨ (M ∈ (C − { Neg x })) by auto
thus False

proof
  assume M ∈ (D − { Pos x })
  show False
  proof cases
    assume ∃ AA. M = (Pos AA)
    from this obtain AA where M = Pos AA by auto
    from (M ∈ D − { Pos x }) and (strictly-maximal-literal D (Pos x))
      and (M = Pos AA)
    have (AA, x) ∈ atom-ordering unfolding strictly-maximal-literal-def
      by auto
    from a and (AA, x) ∈ atom-ordering; and (M = (Pos AA)); and
      (M ∈ (D − { Pos x }))
    have ¬(canonic-int-fun-ordered AA) S by blast
    S
    unfolding canonic-int-ordered-def by blast
    from (AA ∉ canonic-int-ordered S) and (M = Pos AA)
      and (validate-literal (canonic-int-ordered S) M)
    show False by auto
next
  assume ¬(∃ AA. M = (Pos AA))
    obtain AA where M = (Pos AA) ∨ M = (Neg AA) using
    Literal.exhaust [of M] by auto
    from this and ¬(∃ AA. M = (Pos AA)); have M = (Neg AA) by auto
    from b and (AA, x) ∈ atom-ordering; and (M = (Neg AA)); and
      (M ∈ (D − { Pos x }))
    have (canonic-int-fun-ordered AA) S by blast
    S
    unfolding canonic-int-ordered-def by blast
    from AA ∈ canonic-int-ordered S; and (M = (Neg AA))
      and (validate-literal (canonic-int-ordered S) M) show False by auto
qed
next
  assume M ∈ (C − { Neg x })
\[
\text{from } \langle \neg \text{validate-clause}(\text{canonic-int-ordered } S) \rangle \text{ C and } \langle M \in (C - \{\text{Neg } x \}) \rangle:
\]
\[
\quad \text{and } \langle \text{validate-literal } (\text{canonic-int-ordered } S) \rangle \text{ M; show } \text{False by auto qed}
\]
\[
\quad \text{qed from } \langle \neg \text{validate-clause } (\text{canonic-int-ordered } S) \rangle \text{ R; have } \neg \text{tautology } R
\]
\[
\quad \text{using } \text{tautologies-are-valid by auto from } \langle \text{ordered-resolvent } D C R \rangle \text{ and } \langle D \in S \rangle \text{ and } \langle C \in S \rangle
\]
\[
\quad \text{and } \langle \text{saturated-binary-rule ordered-resolvent } S \rangle
\]
\[
\quad \text{have redundant } R S \text{ unfolding saturated-binary-rule-def by auto from this and } \langle \neg \text{tautology } R \rangle \text{ obtain } R' \text{ where } R' \in S \text{ and subsumes R' R}
\]
\[
\quad \text{unfolding redundant-def by auto from } \langle R = \text{resolvent-upon } D C x \rangle \text{ and } \langle \text{strictly-maximal-literal } D \rangle (\text{Pos } x)
\]
\[
\quad \text{and } \langle \text{strictly-maximal-literal } C L \rangle \text{ and } \langle L = \langle \text{Neg } x \rangle \rangle
\]
\[
\quad \text{have resolvent } D C R \text{ unfolding strictly-maximal-literal-def}
\]
\[
\quad \text{using } \text{resolvent-upon-is-resolvent by auto from } \langle \text{all-fulfill finite } S \rangle \text{ and } \langle C \in S \rangle \text{ have finite } C \text{ using all-fulfill-def by auto from } \langle \text{all-fulfill finite } S \rangle \text{ and } \langle D \in S \rangle \text{ have finite } D \text{ using all-fulfill-def by auto}
\]
\[
\quad \text{from } \langle \text{finite } C \rangle \text{ and } \langle \text{finite } D \rangle \text{ and } \langle \neg \text{tautology } D C R \rangle \text{ have finite } R
\]
\[
\quad \text{using } \text{resolvent-is-finite unfolding derived-clauses-are-finite-def by blast from } \langle \text{finite } R \rangle \text{ and } \langle \text{subsumes } R' R \rangle \text{ have finite } R' \text{ unfolding subsumes-def using } \text{finite-subset by auto from } \langle R' \in S \rangle \text{ and } \langle \{\} \notin S \rangle \text{ and } \langle \text{subsumes } R' R \rangle \text{ have } R' \neq \{\}
\]
\[
\quad \text{unfolding subsumes-def by auto from } \langle \text{finite } R' \rangle \text{ and } \langle R' \neq \{\} \rangle \text{ have card } R' \neq 0 \text{ using card-0-eq by auto}
\]
\[
\quad \text{from } \langle \text{subsumes } R' R \rangle \text{ and } \langle \neg \text{tautology } R \rangle \text{ have } \neg \text{tautology } R'
\]
\[
\quad \text{unfolding subsumes-def using } \text{tautology-monotous by auto from } \langle \neg \text{tautology } R' \rangle \text{ and } \langle \text{finite } R' \rangle \text{ and } \langle \text{card } R' \neq 0 \rangle \text{ obtain LR'}
\]
\[
\quad \text{where strictly-maximal-literal } R' LR' \text{ using strictly-maximal-literal-exists by blast from } \langle \text{finite } R \rangle \text{ and } \langle \text{finite } R' \rangle \text{ and } \langle \text{card } R' \neq 0 \rangle \text{ and } \langle \text{subsumes } R' R \rangle \text{ have card } R \neq 0 \text{ unfolding subsumes-def by auto from } \langle \neg \text{tautology } R \rangle \text{ and } \langle \text{finite } R \rangle \text{ and } \langle \text{card } R \neq 0 \rangle \text{ obtain LR}
\]
\[
\quad \text{where strictly-maximal-literal } R LR \text{ using strictly-maximal-literal-exists by blast obtain } AR \text{ and } AR' \text{ where } AR = \text{atom } LR \text{ and } AR' = \text{atom } LR' \text{ by auto from } \langle \text{subsumes } R' R \rangle \text{ and } \langle \text{AR} = \text{atom } LR \rangle \text{ and } \langle \text{AR}' = \text{atom } LR' \rangle \text{ and } \langle \text{strictly-maximal-literal } R LR \rangle
\]
\[
\quad \text{and } \langle \text{strictly-maximal-literal } R' LR' \rangle \text{ have } \langle \text{AR}' = \text{AR} \rangle \lor (\text{AR}',\text{AR})
\]

34
\( \in \text{atom-ordering} \)

using subsumption-and-max-literal by auto
from \( \langle R = (\text{resolvent-upon } D C x) \rangle \) and \( \langle AR = \text{atom } LR \rangle \)
and \( \langle \text{strictly-maximal-literal } LR \rangle \)
and \( \langle \text{strictly-maximal-literal } D (\text{Pos } x) \rangle \)
and \( \langle \text{strictly-maximal-literal } C L \rangle \)
and \( \langle L = (\text{Neg } x) \rangle \)

have \( (AR,x) \in \text{atom-ordering} \) using resolution-and-max-literal by auto
from \( \langle (AR,x) \in \text{atom-ordering} \rangle \) and \( \langle (AR' = AR) \rangle \)

have \( (AR',x) \in \text{atom-ordering} \) using atom-ordering-trans by auto
from this and hyp-induct and \( \langle R' \in S \rangle \) and \( \langle \text{strictly-maximal-literal } R' \rangle \)
\( R' \) by auto
from this and \( \langle \text{subsumes } R' R \rangle \) and \( \langle \neg\text{validate-clause } (\text{canonic-int-ordered } S) \rangle \)
\( S \) \( R \)

show False using subsumption-and-semantics by blast
qed
qed
qed
from inductive-lemma and \( \langle C \in S \rangle \) and \( \langle \text{strictly-maximal-literal } C L \rangle \) and \( \langle A = \text{atom } L \rangle \)
show ?thesis by blast
qed

theorem ordered-resolution-is-complete :
Complete ordered-resolvent
proof (rule ccontr)
assume \( \neg \text{Complete ordered-resolvent} \)
then obtain \( S \) where saturated-binary-rule ordered-resolvent \( S \)
and \( \text{all-fulfill finite } S \) and \( \langle \emptyset \notin S \text{ and } \neg\text{satisfiable } S \rangle \) unfolding Complete-def by auto
have validate-formula (canonic-int-ordered \( S \)) \( S \)
proof (rule ccontr)
assume \( \neg\text{validate-formula } (\text{canonic-int-ordered } S) \)
from this obtain \( C \) where \( C \in S \) and \( \neg\text{validate-clause } (\text{canonic-int-ordered } S) \)
\( C \) by auto
from (saturated-binary-rule ordered-resolvent \( S \)) and \( \langle \text{all-fulfill finite } S \rangle \) and \( \langle \emptyset \notin S \rangle \)
and \( \langle C \in S \rangle \) and \( \langle \neg\text{validate-clause } (\text{canonic-int-ordered } S) \rangle \)
\( C \)
show False using canonic-int-validates-all-clauses by auto
qed
from (validate-formula (canonic-int-ordered \( S \)) \( S \)) and \( \langle \neg\text{satisfiable } S \rangle \) show False
unfolding satisfiable-def by blast
qed

35
7.2 Ordered Resolution with Selection

We now consider the case where some negative literals are considered with highest priority. The proof reuses the canonic interpretation defined in the previous section. The interpretation is constructed using only clauses with no selected literals. By the previous result, all such clauses must be satisfied. We then show that the property carries over to the clauses with non empty selected part.

**definition** empty-selected-part Sel S = \{ C : C ∈ S ∧ (selected-part Sel C) = {} \}

**lemma** saturated-ordered-sel-res-empty.sel :

assumes saturated-binary-rule (ordered-sel-resolvent Sel) S

shows saturated-binary-rule ordered-resolvent (empty-selected-part Sel S)

**proof** −

show ?thesis

proof (rule ccontr)

assume ¬saturated-binary-rule ordered-resolvent (empty-selected-part Sel S)

then obtain P1 and P2 and C

where P1 ∈ empty-selected-part Sel S and P2 ∈ empty-selected-part Sel S

and ordered-resolvent P1 P2 C

and ¬redundant C (empty-selected-part Sel S)

unfolding saturated-binary-rule-def by auto

from ordered-resolvent P1 P2 C obtain A where C = ( (P1 - { Pos A}) ∪ (P2 - { Neg A}))

and strictly-maximal-literal P1 (Pos A) and strictly-maximal-literal P2 (Neg A)

unfolding ordered-resolvent-def by auto

from (P1 ∈ empty-selected-part Sel S) have selected-part Sel P1 = {}

unfolding empty-selected-part-def by auto

from (P2 ∈ empty-selected-part Sel S) have selected-part Sel P2 = {}

unfolding empty-selected-part-def by auto

from (C = ( (P1 - { Pos A}) ∪ (P2 - { Neg A}))) and (strictly-maximal-literal P1 (Pos A))

and (strictly-maximal-literal P2 (Neg A)) and (selected-part Sel P1) = {} and (selected-part Sel P2) = {}

have ordered-sel-resolvent Sel P1 P2 C unfolding ordered-sel-resolvent-def by auto

from (saturated-binary-rule (ordered-sel-resolvent Sel) S)

have ∀ P1 P2 C. (P1 ∈ S ∧ P2 ∈ S ∧ (ordered-sel-resolvent Sel P1 P2 C)) → redundant C S

unfolding saturated-binary-rule-def by auto

from this and (P1 ∈ (empty-selected-part Sel S) and (P2 ∈ (empty-selected-part Sel S))

and (ordered-sel-resolvent Sel P1 P2 C) have tautology C ∨ (∃ D. D ∈ S ∧ subsuces D C)

unfolding empty-selected-part-def redundant-def by auto

from this and (tautology C ∨ (∃ D. D ∈ S ∧ subsuces D C))

and (¬redundant C (empty-selected-part Sel S))

36
obtain $D$ where $D \in S$ and subsumes $D \in C$ and $D \notin \text{empty-selected-part}$ $Sel \ S$

unfolding redundant-def by auto

from $\langle D \notin \text{empty-selected-part} \text{ Sel S} \rangle$ and $\langle D \in S \rangle$ obtain $B$ where $B \in Sel$ and $Neg B \in D$

unfolding empty-selected-part-def selected-part-def by auto

from $\langle Neg B \in D \rangle$ this and $\langle \text{subsumes} D \in C \rangle$ have $Neg B \in C$ unfolding subsumes-def by auto

from $\langle this \rangle$ and $\langle C = ( (P1 - \{Pos A\}) \cup (P2 - \{Neg A\}) ) \rangle$ have $Neg B \in (P1 \cup P2)$ by auto

from $\langle Neg B \in (P1 \cup P2) \rangle$ and $\langle P1 \in \text{empty-selected-part} \text{ Sel S} \rangle$

and $\langle P2 \in \text{empty-selected-part} \text{ Sel S} \rangle$ and $\langle B \in Sel \rangle$ show False

unfolding empty-selected-part-def selected-part-def by auto

qed

qed
definition ordered-sel-resolvent-upon :: 'at set ⇒ 'at Clause ⇒ 'at Clause ⇒ 'at Clause ⇒ bool

where

ordered-sel-resolvent-upon Sel P1 P2 C A ≡

$(((C = ( (P1 - \{Pos A\}) \cup (P2 - \{Neg A\}) ))$

$\land (\text{strictly-maximal-literal} P1 (Pos A)) \land (\text{selected-part Sel P1}) = \{\})$

$\land ( ((\text{strictly-maximal-literal} P2 (Neg A)) \land (\text{selected-part Sel P2}) = \{\})$

$\lor (\text{strictly-maximal-literal} (\text{selected-part Sel P2}) (Neg A)))))$

lemma ordered-sel-resolvent-upon-is-resolvent:

assumes ordered-sel-resolvent-upon Sel P1 P2 C A

shows ordered-sel-resolvent Sel P1 P2 C

using assms unfolding ordered-sel-resolvent-upon-def and ordered-sel-resolvent-def by auto

lemma resolution-decreases-selected-part:

assumes ordered-sel-resolvent-upon Sel P1 P2 C A

assumes $Neg A \in P2$

assumes finite P1

assumes finite P2

assumes card (selected-part Sel P2) = Suc n

shows card (selected-part Sel C) = n

proof −

from $\langle \text{finite P2} \rangle$ have finite (selected-part Sel P2) unfolding selected-part-def by auto

from $\langle \text{card (selected-part Sel P2)} = (\text{Suc n}) \rangle$ have $\text{card (selected-part Sel P2)} \neq 0$ by auto

from $\langle this \text{ and } \text{finite (selected-part Sel P2)} \rangle$ have selected-part Sel P2 $\neq \{\}$

using card-0-eq by auto

from $\langle this \text{ and } \text{ordered-sel-resolvent-upon Sel P1 P2 C A} \rangle$ have

$C = (P1 - \{Pos A\}) \cup (P2 - \{Neg A\})$

and selected-part Sel P1 = $\{\}$ and strictly-maximal-literal (selected-part Sel P2) (Neg A)
unfolding ordered-sel-resolvent-upon-def by auto
from (strictly-maximal-literal (selected-part Sel P2) (Neg A))
  have Neg A ∈ selected-part Sel P2
unfolding strictly-maximal-literal-def by auto
from this have A ∈ Sel unfolding selected-part-def by auto
from (selected-part Sel P1 = {}) have selected-part Sel (P1 - { Pos A}) = {}
  unfolding selected-part-def by auto
from ⟨Neg A ∈ (selected-part Sel P2): have selected-part Sel (P2 - { Neg A}) = (selected-part Sel P2) - { Neg A }⟩
  unfolding selected-part-def by auto
from ⟨C = ( (P1 - { Pos A}) ∪ (P2 - { Neg A })): have selected-part Sel C
  = (selected-part Sel (P1 - { Pos A})) ∪ (selected-part Sel (P2 - { Neg A })): unfolding selected-part-def by auto
from this and ⟨selected-part Sel (P1 - { Pos A}) = {}⟩
  and ⟨selected-part Sel (P2 - { Neg A }) = selected-part Sel P2 - { Neg A }⟩ by auto
from ⟨Neg A ∈ P2; and (A ∈ Sel): have Neg A ∈ selected-part Sel P2 unfolding selected-part-def by auto
from this and ⟨selected-part Sel C = (selected-part Sel P2) - { Neg A }⟩
  and ⟨finite (selected-part Sel P2): have card (selected-part Sel C) = card (selected-part Sel P2) - 1 by auto
from this and ⟨card (selected-part Sel P2) = Suc n: show ?thesis by auto⟩

lemma canonic-int-validates-all-clauses-sel:
  assumes saturated-binary-rule (ordered-sel-resolvent Sel) S
  assumes all-fulfill finite S
  assumes {} /∈ S
  assumes C ∈ S
  shows validate-clause (canonic-int-ordered (empty-selected-part Sel S)) C
proof -
  let ?nat-order = { (x::nat,y::nat). x < y }
  let ?SE = empty-selected-part Sel S
  let ?I = canonic-int-ordered ?SE
  obtain n where n = card (selected-part Sel C) by auto
  have ∀ C. card (selected-part Sel C) = n → C ∈ S → validate-clause ?I C
(is ?P n)
proof ((rule wf-induct [of ?nat-order ?P n]), (simp add:wf))
next
  fix n assume ind-hyp: ∀ m. (m,n) ∈ ?nat-order → (?P m)
  show (?P n)
proof (rule+)
  fix C assume card (selected-part Sel C) = n and C ∈ S
  from (all-fulfill finite S) and ⟨C ∈ S⟩ have finite C unfolding all-fulfill-def by auto
  from this have finite (selected-part Sel C) unfolding selected-part-def by auto
  show validate-clause ?I C

38
proof (rule nat.exhaust [of n])
  assume n = 0
  from this and (card (selected-part Sel C) = n) and (finite (selected-part Sel C))
  have selected-part Sel C = {} by auto
  from (saturated-binary-rule (ordered-sel-resolvent Sel) S)
  have saturated-binary-rule ordered-resolvent ?SE
    using saturated-ordered-sel-res-empty-sel by auto
  from {} \notin S have {} \notin ?SE unfolding empty-selected-part-def by auto
  from (selected-part Sel C = {}) \langle C \in S \rangle have C \in ?SE unfolding empty-selected-part-def
    by auto
  from (all-fulfill finite S) have all-fulfill finite ?SE
    unfolding empty-selected-part-def by auto
  from this and (saturated-binary-rule ordered-resolvent ?SE) and ({} \notin ?SE) and (C \in ?SE)
  show validate-clause ?I C using canonic-int-validates-all-clauses by auto
next
fix m assume n = Suc m
from this and (card (selected-part Sel C) = n) have selected-part Sel C \neq {} by auto
show validate-clause ?I C proof (rule ccontr)
  assume \neg validate-clause ?I C
  show False
  proof (cases)
    assume \neg tautology C
    from \langle tautology C \rangle and (\neg validate-clause ?I C) show False
      using tautologies-are-valid by auto
next
assume \neg (tautology C)
  hence (\neg (tautology (selected-part Sel C)))
    unfolding selected-part-def tautology-def by auto
  from (selected-part Sel C \neq {}) and (finite (selected-part Sel C))
    have card (selected-part Sel C) \neq 0 by auto
  from this and (\neg (tautology (selected-part Sel C))) and (finite (selected-part Sel C))
  obtain L where strictly-maximal-literal (selected-part Sel C) L
    using strictly-maximal-literal-exists [of card (selected-part Sel C)] by blast
  from (strictly-maximal-literal (selected-part Sel C) L) have L \in (selected-part Sel C)
    and L \in C unfolding strictly-maximal-literal-def selected-part-def by auto
  from this and (\neg validate-clause ?I C) have (\neg (validate-literal ?I L)) by auto
  from (L \in (selected-part Sel C)) obtain A where L = (Neg A) and A \in Sel
    unfolding selected-part-def by auto

39
from (~ (validate-literal ?I L)) and ⟨ L = (Neg A) ⟩ have A ∈ ?I by auto
from this have ((canonic-int-fun-ordered A) ?SE) unfolding canonic-int-ordered-def
  by blast
have (( ∃ C. (C ∈ ?SE) ∧ (strictly-maximal-literal C (Pos A)) ∧ (∀ B. (Pos B ∈ C → (B, A) ∈ atom-ordering → (~ (canonic-int-fun-ordered B) ?SE)))) ∧ (∀ B. (Neg B ∈ C → (B, A) ∈ atom-ordering → ((canonic-int-fun-ordered B) ?SE))))) (is ?exp)
proof (rule ccontr)
  assume ~ ?exp
  from this have ~((canonic-int-fun-ordered A) ?SE)
  by ((simp only: canonic-int-fun-ordered.simps [of A]), blast)
from this and ((canonic-int-fun-ordered A) ?SE) show False by blast
qed
then obtain D where
  D ∈ ?SE and strictly-maximal-literal D (Pos A)
and c1: (∀ B. (Pos B ∈ D → (B, A) ∈ atom-ordering → (~ (canonic-int-fun-ordered B) ?SE)))
and c2: (∀ B. (Neg B ∈ D → (B, A) ∈ atom-ordering → ((canonic-int-fun-ordered B) ?SE)))
by blast
from ⟨ D ∈ ?SE ⟩ have (selected-part Sel D) = {} and D ∈ S
  unfolding empty-selected-part-def by auto
from ⟨ D ∈ ?SE ⟩ and ⟨ all-fulfill finite S ⟩ have finite D
  unfolding empty-selected-part-def all-fulfill-def by auto
let ?R = (D − {Pos A}) ∪ (C − {Neg A})
from (strictly-maximal-literal D (Pos A))
  and (strictly-maximal-literal (selected-part Sel C) L)
and ⟨ L = (Neg A) ⟩ and ((selected-part Sel D) = {}) have (ordered-sel-resolvent-upon Sel D C ?R A)
  unfolding ordered-sel-resolvent-upon-def by auto
from this have ordered-sel-resolvent Sel D C ?R by (rule ordered-sel-resolvent-upon-is-resolvent)
from ⟨ ordered-sel-resolvent-upon Sel D C ?R A ⟩ : ((card (selected-part Sel C)) = n)
  and ⟨ n = Suc m ⟩ and ⟨ L ∈ C ⟩ and ⟨ L = (Neg A) ⟩ and ⟨ finite D ⟩
  and ⟨ finite C ⟩
  have card (selected-part Sel ?R) = m
  using resolution-decreases-selected-part by auto
from (ordered-sel-resolvent Sel D C ?R) and ⟨ D ∈ S ⟩ and ⟨ C ∈ S ⟩
  and (saturated-binary-rule (ordered-sel-resolvent Sel C) S)
  have redundant ?R S unfolding saturated-binary-rule-def by auto
hence tautology ?R ∨ (∃ RR. (RR ∈ S ∧ (subsumes RR ?R)))
  unfolding redundant-def by auto
hence validate-clause ?I ?R proof
  assume tautology ?R
  thus validate-clause ?I ?R by (rule tautologies-are-valid)
next
assume $\exists R', R' \in S \land (\text{subsumes } R' \ ?R)$
then obtain $R'$ where $R' \in S$ and $\text{subsumes } R' \ ?R$ by auto
from $\langle \text{finite } C \text{ and } \text{finite } D \rangle$ have finite $?R$ by auto
from this have finite (selected-part Sel $?R$) unfolding selected-part-def
by auto
from $\langle \text{subsumes } R' \ ?R \rangle$ have selected-part Sel $R' \subseteq$ selected-part Sel $?R$
unfolding selected-part-def and subsumes-def by auto
from this and $\langle \text{finite } (\text{selected-part Sel } ?R) \rangle$
have card (selected-part Sel $R'$) $\leq$ card (selected-part Sel $\ ?R$)
using card-mono by auto
from this and $\langle \text{card } (\text{selected-part Sel } ?R) = m \rangle$ and $(n = \text{Suc } m)$
have card (selected-part Sel $R'$) $<$ $n$ by auto
from this and $\langle \text{ind-hyp } \text{and } \langle R' \in S \rangle \text{ have validate-clause } ?I \ R' \rangle$
proof
assume $L' \in D - \{ \text{Pos } A \}$
from this have $L' \in D$ by auto
let $?A' = \text{atom } L'$
have $L' = (\text{Pos } ?A') \lor L' = (\text{Neg } ?A')$ using atom-property $[\text{of } ?A' \langle L' \rangle$ by auto
thus False
proof
assume $L' = (\text{Pos } ?A')$
from this and $\langle \text{strictly-maximal-literal } D \ (\text{Pos } A) \rangle$ and $\langle L' \in D - \{ \text{Pos } A \} \rangle$
have $\langle ?A', A \rangle \in \text{atom-ordering}$ unfolding strictly-maximal-literal-def
by auto
from $c1$
have $c1': \text{Pos } ?A' \in D \rightarrow (\langle ?A', A \rangle \in \text{atom-ordering})$
$\rightarrow (\langle \text{canonic-int-fun-ordered } ?A' \rangle \ ?SE \rangle$ by blast
from $\langle L' \in D \rangle$ and $\langle L' = \text{Pos } ?A' \rangle$ have $\text{Pos } ?A' \in D$ by auto
from $c1'$ and $\langle \text{Pos } ?A' \in D \rangle$ and $\langle \langle ?A', A \rangle \in \text{atom-ordering} \rangle$
have $\langle \text{canonic-int-fun-ordered } ?A' \rangle$ ?SE by blast
from this have $?A' \notin \ ?I$ unfolding canonic-int-ordered-def by blast
from this have $\langle \text{validate-literal } ?I \ (\text{Pos } ?A') \rangle$ by auto
from this and $\langle L' = \text{Pos } ?A' \rangle$ and $\langle \text{validate-literal } ?I \ L' \rangle$ show False by auto
next
assume $L' = \text{Neg } ?A'$
from this and $\langle \text{strictly-maximal-literal } D \ (\text{Pos } A) \rangle$ and $\langle L' \in D - \{ \text{Pos } A \} \rangle$

41
{ Pos A }

by auto

from c2

have c2': Neg ?A' ∈ D → (\(\neg A', A\)) ∈ atom-ordering

→ (canonic-int-fun-ordered ?A') ?SE

by blast

from \(\langle L' \rangle \in D\) and \(\langle L' = (\neg A') \rangle\) have Neg ?A' ∈ D by auto

from c2' and \(\langle \neg ?A' \in D, \langle \langle \neg ?A', A \rangle \rangle \in atom-ordering \rangle\)

have (canonic-int-fun-ordered ?A') ?SE by blast

from this have ?A' ∈ ?I unfolding canonic-int-ordered-def by blast

from this have \(\neg validate-lite ?I \langle \neg ?A' \rangle\) by auto

from this and \(\langle L' = \neg ?A' \rangle\) and \(\neg validate-lite ?I L'\) show False by auto

qed

qed

qed

from \(\langle P n \rangle\) and \(\langle n = card (selected-part Sel C)\rangle\) and \(\langle C \in S \rangle\) show ?thesis by auto

qed

theorem ordered-resolution-is-complete-ordered-sel :

Complete (ordered-sel-resolvent Sel)

proof (rule ccontr)

assume \(\neg Complete (ordered-sel-resolvent Sel)\)

then obtain S where saturated-binary-rule (ordered-sel-resolvent Sel) S

and all-fulfill finite S

and \(\{} \notin S\)

and \(\neg satisfiable S\) unfolding Complete-def by auto

let \(?SE = empty-selected-part Sel S\)

let \(?I = canonic-int-ordered ?SE\)

have validate-formula ?I S

proof (rule ccontr)

assume \(\neg(validate-formula ?I S)\)

from this obtain C where C ∈ S and \(\neg(validate-clause ?I C)\) by auto

from (saturated-binary-rule (ordered-sel-resolvent Sel) S) and \(\langle all-fulfill finite S \rangle\)

and \(\{} \notin S\) and \(\langle C \in S \rangle\) and \(\neg(validate-clause ?I C)\)

show False using canonic-int-validates-all-clauses-sel [of Sel S C] by auto

qed

from (validate-formula ?I S) and \(\neg(satisfiable S)\) show False

42
unfolding \textit{satisfiable-def} by blast

\section{7.3 Semantic Resolution}

We show that under some particular renaming, model resolution simulates
ordered resolution where all negative literals are selected, which immediately
entails the refutational completeness of model resolution.

\textbf{lemma} ordered-res-with-selection-is-model-res:
\begin{itemize}
\item \textbf{assumes} ordered-sel-resolvent UNIV P1 P2 C
\item \textbf{shows} ordered-model-resolvent Sel (rename-clause Sel P1) (rename-clause Sel P2)
\end{itemize}
\begin{quote}
(\text{rename-clause Sel C})
\end{quote}
\begin{itemize}
\item \textbf{proof} –
\item from \texttt{assms obtain} A
\item \textbf{where} \textit{c-def}: \texttt{C} = \((\text{P1} - \{ \text{Pos A } \}) \cup (\text{P2} - \{ \text{Neg A } \}))
\item and \texttt{selected-part UNIV P1} = \{
\item and \texttt{strictly-maximal-literal P1} \texttt{(Pos A)}
\item and \texttt{disj}: \((\text{strictly-maximal-literal P2} \texttt{ (Neg A))} \land \texttt{(selected-part UNIV P2) = \{}}
\item or \texttt{(strictly-maximal-literal (selected-part UNIV P2) (Neg A))}
\item unfolding \texttt{ordered-sel-resolvent-def} by blast
\item have \texttt{rename-clause Sel ((P1 - \{ Pos A \}) \cup (P2 - \{ Neg A \}))}
\item = \((\text{rename-clause Sel (P1 - \{ Pos A \})) \cup \text{rename-clause Sel (P2 - \{ (Neg A) \})})
\item using \texttt{rename-union} \texttt{[of Sel P1 - \{ Pos A \} P2 - \{ Neg A \]] by auto}
\item from \texttt{this and c-def} have \texttt{ren-c: (rename-clause Sel C) =}
\item \((\text{rename-clause Sel (P1 - \{ Pos A \})) \cup \text{rename-clause Sel (P2 - \{ (Neg A) \})})
\item by auto
\item have \texttt{m1: (rename-clause Sel (P1 - \{ Pos A \})) = (rename-clause Sel P1)}
\item = \{ \text{rename-literal Sel (Pos A)} \}
\item using \texttt{renaming-set-minus by auto}
\item have \texttt{m2: rename-clause Sel (P2 - \{ Neg A \}) = (rename-clause Sel P2)}
\item = \{ \text{rename-literal Sel (Neg A)} \}
\item using \texttt{renaming-set-minus by auto}
\item from \texttt{m1 and m2 and ren-c have}
\item \texttt{rc-def: (rename-clause Sel C) =}
\item \(((\text{rename-clause Sel P1}) - \{ \text{rename-literal Sel (Pos A)} \})
\item \cup ((\text{rename-clause Sel P2}) - \{ \text{rename-literal Sel (Neg A)} \}))
\item by auto
\item have \texttt{\neg((strictly-maximal-literal P2 (Neg A)) \land \texttt{(selected-part UNIV P2) = \{}}
\item proof
\item assume \texttt{(strictly-maximal-literal P2 (Neg A)) \land \texttt{(selected-part UNIV P2) = \{}}
\item from \texttt{this have strictly-maximal-literal P2 (Neg A) and selected-part UNIV P2}
\item = \{ \} by auto
\item from \texttt{strictly-maximal-literal P2 (Neg A): have Neg A \in P2}
\item unfolding \texttt{strictly-maximal-literal-def by auto}
\item from \texttt{this and (selected-part UNIV P2) = \{\}, show False unfolding selected-part-def}
\item by auto
\end{itemize}
qed

\textbf{theorem} ordered-resolution-is-complete-model-resolution:
Complete \((\text{ordered-model-resolvent } \text{Sel})\)
\textbf{proof} (rule ccontr)
assume \(\neg\text{Complete } (\text{ordered-model-resolvent } \text{Sel})\)
then obtain \(S\) where \(\text{saturated-binary-rule } (\text{ordered-model-resolvent } \text{Sel}) \ S\)
and \(\{\} \notin S\) and \(\text{all-fulfill finite } S\) and \(\neg(\text{satisfiable } S)\) unfolding \text{Complete-def}
by auto
let \(?S' = \text{rename-formula } \text{Sel } S\)
have \(\{\} \notin ?S'\)
\textbf{proof}
assume \(\{\} \in ?S'\)
then obtain \(V\) where \(V \in S\) and \(\text{rename-clause } \text{Sel } V = \{\}\) unfolding \text{rename-formula-def} by auto
from \(\text{rename-clause } \text{Sel } V = \{\}\) have \(V = \{\}\) unfolding \text{rename-clause-def} by auto
from this and \( \langle V \in S \rangle \) and \( \{\} \notin S \) show \text{False by auto} 

qed

from \( \text{all-fulfill finite } S \) have \text{all-fulfill finite } ?S'

unfolding \text{all-fulfill-def rename-formula-def rename-clause-def} by auto

have \text{saturated-binary-rule (ordered-set-resolvent UNIV) } ?S'

proof (rule contr)

assume \( (\neg \text{saturated-binary-rule (ordered-set-resolvent UNIV) } ?S') \)

from this obtain \( P_1 \) and \( P_2 \) and \( C \) where

\( P_1 \in ?S' \) and \( P_2 \in ?S' \) and \( \text{ordered-set-resolvent UNIV } P_1 \ P_2 \ C \) and \( \neg \text{tautology } C \)

unfolding \text{saturated-binary-rule-def redundant-def} by auto

have \( \text{saturated-binary-rule } \text{UNIV } P_1 \ P_2 \ C \)

have \text{ord-ren: ordered-model-resolvent Sel (rename-clause Sel } P_1 \text{) (rename-clause Sel } P_2 \text{)}

\( (\text{rename-clause Sel } C) \)

using \text{ordered-res-with-selection-is-model-res} by auto

have \( \neg \text{tautology } (\text{rename-clause Sel } C) \)

using \text{renaming-preserves-tautology inverse-clause-renaming}

by \( (\text{metis } (\text{\neg \text{tautology } C}) \ \text{inverse-clause-renaming renaming-preserves-tautology}) \)

from \( \langle P_1 \in ?S' \rangle \) have \text{rename-clause Sel } P_1 \in \text{rename-formula Sel } ?S'

unfolding \text{rename-formula-def} by auto

hence \text{rename-clause Sel } P_1 \in S using \text{inverse-formula-renaming} by auto

from \( \langle P_2 \in ?S' \rangle \) have \text{rename-clause Sel } P_2 \in \text{rename-formula Sel } ?S'

unfolding \text{rename-formula-def} by auto

hence \text{rename-clause Sel } P_2 \in S using \text{inverse-formula-renaming} by auto

from \( (\neg \text{tautology } (\text{rename-clause Sel } C)) \) and \text{ord-ren}

and \( (\text{saturated-binary-rule (ordered-model-resolvent Sel) } S) \)

and \( (\text{rename-clause Sel } P_1 \in S) \) and \( (\text{rename-clause Sel } P_2 \in S) \)

obtain \( D' \) where \( D' \in S \) and \text{subsumes } D' \text{ (rename-clause Sel } C) \)

unfolding \text{saturated-binary-rule-def redundant-def} by blast

from \( \langle \text{subsumes } D' \rangle \text{ (rename-clause Sel } C) \)

have \text{subsumes (rename-clause Sel } D') \text{ (rename-clause Sel (rename-clause Sel } C))

using \text{renaming-preserves-subsumption} by auto

hence \text{subsumes (rename-clause Sel } D') \text{ C using inverse-clause-renaming} by auto

from \( \langle D' \in S \rangle \) have \text{rename-clause Sel } D' \in ?S' unfolding \text{rename-formula-def}

by auto

from this and \text{not-subsumed} and \text{ (subsumes (rename-clause Sel } D') \text{ C) show False by auto}

qed

from this and \( \{\} \notin ?S' \) and \text{all-fulfill finite } ?S' \text{ have satisfiable } ?S'

using \text{ordered-resolution-is-complete-ordered-set unfolding} \text{Complete-def} by auto

hence \text{satisfiable (rename-formula Sel } ?S') \text{ using renaming-preserves-satisfiability}

by auto

from this and \( (\neg \text{satisfiable } S) \) show \text{False using inverse-formula-renaming} by auto

qed
7.4 Positive and Negative Resolution

We show that positive and negative resolution simulate model resolution with some specific interpretation. Then completeness follows from previous results.

**Lemma** empty-interpretation-validate:
\[
\text{validate-literal } \{\} L = (\exists A. (L = \neg A))
\]
by (meson empty-iff validate-literal.elims(2) validate-literal.simps(2))

**Lemma** universal-interpretation-validate:
\[
\text{validate-literal } \text{UNIV } L = (\exists A. (L = \text{Pos } A))
\]
by (meson UNIV-I validate-literal.elims(2) validate-literal.simps(1))

**Lemma** negative-part-lemma:
\[
(\text{negative-part } C) = (\text{validated-part } \{\} C)
\]
unfolding negative-part-def validated-part-def using empty-interpretation-validate
by blast

**Lemma** positive-part-lemma:
\[
(\text{positive-part } C) = (\text{validated-part } \text{UNIV } C)
\]
unfolding positive-part-def validated-part-def using universal-interpretation-validate
by blast

**Lemma** negative-resolvent-is-model-res:
\[
\text{less-restrictive ordered-negative-resolvent } (\text{ordered-model-resolvent } \text{UNIV})
\]
unfolding ordered-negative-resolvent-def ordered-model-resolvent-def less-restrictive-def
using positive-part-lemma by auto

**Lemma** positive-resolvent-is-model-res:
\[
\text{less-restrictive ordered-positive-resolvent } (\text{ordered-model-resolvent } \{\})
\]
unfolding ordered-positive-resolvent-def ordered-model-resolvent-def less-restrictive-def
using negative-part-lemma by auto

**Theorem** ordered-positive-resolvent-is-complete: Complete ordered-positive-resolvent
using ordered-resolution-is-complete-model-resolution less-restrictive-complete positive-resolvent-is-model-res
by auto

**Theorem** ordered-negative-resolvent-is-complete: Complete ordered-negative-resolvent
using ordered-resolution-is-complete-model-resolution less-restrictive-complete negative-resolvent-is-model-res
by auto

7.5 Unit Resolution and Horn Renamable Clauses

Unit resolution is complete if the considered clause set can be transformed into a Horn clause set by renaming. This result is proven by showing that unit resolution simulates semantic resolution for Horn-renamable clauses (for
some specific interpretation).

**definition** Horn :: 'at Clause ⇒ bool
where (Horn C) = ((card (positive-part C)) ≤ 1)

**definition** Horn-renamable-formula :: 'at Formula ⇒ bool
where Horn-renamable-formula S = (∃I. (all-fulfill Horn (rename-formula I S)))

**theorem** unit-resolvent-complete-for-Horn-renamable-set:
assumes saturated-binary-rule unit-resolvent S
assumes all-fulfill finite S
assumes {} /∈ S
assumes Horn-renamable-formula S
shows satisfiable S

**proof**

from (Horn-renamable-formula S) obtain I where all-fulfill Horn (rename-formula I S)

unfolding Horn-renamable-formula-def by auto
have saturated-binary-rule (ordered-model-resolvent I) S

proof (rule ccontr)
assume ¬saturated-binary-rule (ordered-model-resolvent I) S
then obtain P1 P2 C where ordered-model-resolvent I P1 P2 C and P1 ∈ S and P2 ∈ S
and ¬redundant C S
unfolding saturated-binary-rule-def by auto
from (ordered-model-resolvent I P1 P2 C) obtain L
where def-c: C = ((P1 - \{L\}) ∪ (P2 - \{complement L\}))
and strictly-maximal-literal P1 L and validated-part I P1 = {}
and strictly-maximal-literal (validated-part I P2) (complement L)
unfolding ordered-model-resolvent-def by auto
from strictly-maximal-literal P1 L have L ∈ P1
unfolding strictly-maximal-literal-def by auto
from strictly-maximal-literal (validated-part I P2) (complement L) have complement L ∈ P2
unfolding strictly-maximal-literal-def validated-part-def by auto
have selected-part UNIV (rename-clause I P1)
  = rename-clause I (validated-part I (rename-clause I P1))
using renaming-and-selected-part [of rename-clause I P1 I] by auto
then have selected-part UNIV (rename-clause I P1) = rename-clause I (validated-part I P1)
using inverse-clause-renaming by auto
from this and (validated-part I P1 = {}): have selected-part UNIV (rename-clause I P1) = {}
unfolding rename-clause-def by auto
then have negative-part (rename-clause I P1) = {}
unfolding selected-part-def negative-part-def by auto
from (all-fulfill Horn (rename-formula I S)) and (P1 ∈ S) have Horn (rename-clause I P1)
unfolding all-fulfill-def and rename-formula-def by auto

47
then have \( \text{card} \ (\text{positive-part} \ (\text{rename-clause} \ I \ P1)) \leq 1 \) unfolding Horn-def
by auto
from \( \text{negative-part} \ (\text{rename-clause} \ I \ P1) = \{\} \),
have rename-clause \( I \ P1 = (\text{positive-part} \ (\text{rename-clause} \ I \ P1)) \)
using decomposition-clause-pos-neg by auto
from this and \( \langle \text{card} \ (\text{positive-part} \ (\text{rename-clause} \ I \ P1)) \leq 1 \rangle \),
have \( \text{card} \ (\text{rename-clause} \ I \ P1) \leq 1 \) by auto
from strictly-maximal-literal \( P1 \ L \) have \( P1 \neq \{\} \)
unfolding strictly-maximal-literal-def by auto
then have rename-clause \( I \ P1 \neq \{\} \) unfolding rename-clause-def by auto
from \( \langle \text{all-fulfill finite} \ S \rangle \) and \( \langle P1 \in S \rangle \) have finite \( P1 \) unfolding all-fulfill-def
by auto
then have finite \( (\text{rename-clause} \ I \ P1) \) unfolding rename-clause-def by auto
from this and \( \langle \text{card} \ (\text{rename-clause} \ I \ P1) \neq 0 \rangle \) have \( \text{card} \ (\text{rename-clause} \ I \ P1) = 1 \) by auto
then have \( \text{card} \ P1 = 1 \) using renaming-preserves-cardinality by auto
then have Unit \( P1 \) unfolding Unit-def using card-image by auto
from this and \( \langle L \in P1 \rangle \) and \( \langle \text{complement} \ L \in P2 \rangle \) and def-c have unit-resolvent \( P1 \ P2 \ C \)
unfolding unit-resolvent-def by auto
from this and \( \langle \neg (\text{redundant} \ C \ S) \rangle \) and \( \langle P1 \in S \rangle \) and \( \langle P2 \in S \rangle \)
and (saturated-binary-rule unit-resolvent S)
show False unfolding saturated-binary-rule-def by auto
qed
from this and \( \langle \text{all-fulfill finite} \ S \rangle \) and \( \langle \{\} \notin S \rangle \) show thesis
using ordered-resolution-is-complete-model-resolution unfolding Complete-def
by auto
qed

8 Computation of Saturated Clause Sets

We now provide a concrete (rather straightforward) procedure for computing saturated clause sets. Starting from the initial set, we define a sequence of clause sets, where each set is obtained from the previous one by applying the resolution rule in a systematic way, followed by redundancy elimination rules. The algorithm is generic, in the sense that it applies to any binary inference rule.

fun inferred-clause-sets :: 'at BinaryRule ⇒ 'at Formula ⇒ nat ⇒ 'at Formula
where
(inferred-clause-sets R S 0) = (simplify S) |
(inferred-clause-sets R S (Suc N)) =
(simplify (add-all-deducible-clauses R (inferred-clause-sets R S N)))

The saturated set is constructed by considering the set of persistent clauses, i.e., the clauses that are generated and never deleted.
fun saturation :: 'at BinaryRule ⇒ 'at Formula ⇒ 'at Formula
where
saturation R S = { C. ∃ N. (∀ M. (M ≥ N −→ C ∈ inferred-clause-sets R S)) }

We prove that all inference rules yield finite clauses.

theorem ordered-resolvent-is-finite: derived-clauses-are-finite ordered-resolvent
using less-restrictive-and-finite ordered-resolvent-is-resolvent resolvent-is-finite
by auto

theorem model-resolvent-is-finite: derived-clauses-are-finite (ordered-model-resolvent I)
using less-restrictive-and-finite ordered-model-resolvent-is-resolvent resolvent-is-finite
by auto

theorem positive-resolvent-is-finite: derived-clauses-are-finite ordered-positive-resolvent
using less-restrictive-and-finite positive-resolvent-is-resolvent resolvent-is-finite
by auto

theorem negative-resolvent-is-finite: derived-clauses-are-finite ordered-negative-resolvent
using less-restrictive-and-finite negative-resolvent-is-resolvent resolvent-is-finite
by auto

theorem unit-resolvent-is-finite: derived-clauses-are-finite unit-resolvent
using less-restrictive-and-finite unit-resolvent-is-resolvent resolvent-is-finite
by auto

lemma all-deducible-clauses-are-finite:
  assumes derived-clauses-are-finite R
  assumes all-fulfill finite S
  shows all-fulfill finite (all-deducible-clauses R S)
proof (rule contr)
  assume ¬ all-fulfill finite (all-deducible-clauses R S)
  from this obtain C where C ∈ all-deducible-clauses R S and ¬ finite C
  unfolding all-fulfill-def by auto
  from ⟨ C ∈ all-deducible-clauses R S ⟩ have ∃ P1 P2. R P1 P2 C ∧ P1 ∈ S ∧ P2 ∈ S by auto
  then obtain P1 P2 where R P1 P2 C and P1 ∈ S and P2 ∈ S by auto
  from ⟨ P1 ∈ S ⟩ and ⟨ all-fulfill finite S ⟩ have finite P1 unfolding all-fulfill-def
  by auto
  from ⟨ P2 ∈ S ⟩ and ⟨ all-fulfill finite S ⟩ have finite P2 unfolding all-fulfill-def
  by auto
  from ⟨ finite P1 ⟩ and ⟨ finite P2 ⟩ and ⟨ derived-clauses-are-finite R ⟩ and ⟨ R P1 P2 C ⟩ and ¬ finite C, show False
  unfolding derived-clauses-are-finite-def by metis
qed

This entails that all the clauses occurring in the sets in the sequence are finite.
lemma all-inferred-clause-sets-are-finite:
assumes derived-clauses-are-finite R
assumes all-fulfill finite S
shows all-fulfill finite (inferred-clause-sets R S N)
proof (induction N)
  from assms show all-fulfill finite (inferred-clause-sets R S 0)
    using simplify-finite by auto
next
  fix N assume all-fulfill finite (inferred-clause-sets R S N)
  then have all-fulfill finite (all-deducible-clauses R (inferred-clause-sets R S N))
    using assms(1) all-deducible-clauses-are-finite [of R inferred-clause-sets R S N]
      by auto
  from this and have all-fulfill finite (add-all-deducible-clauses R (inferred-clause-sets R S N))
    using all-fulfill-def by auto
  then show all-fulfill finite (inferred-clause-sets R S (Suc N))
    using simplify-finite by auto
qed

lemma add-all-deducible-clauses-finite:
assumes derived-clauses-are-finite R
assumes all-fulfill finite S
shows all-fulfill finite (add-all-deducible-clauses R (inferred-clause-sets R S N))
proof
  from assms have all-fulfill finite (all-deducible-clauses R (inferred-clause-sets R S N))
    using all-deducible-clauses-are-finite [of R inferred-clause-sets R S N] by metis
  then show all-fulfill finite (add-all-deducible-clauses R (inferred-clause-sets R S N))
    using assms all-fulfill-def all-inferred-clause-sets-are-finite [of R S N] by auto
qed

We show that the set of redundant clauses can only increase.

lemma sequence-of-inferred-clause-sets-is-monotonous:
assumes derived-clauses-are-finite R
assumes all-fulfill finite S
shows ∀ C. redundant C (inferred-clause-sets R S N) → redundant C (inferred-clause-sets R S (N+M::nat))
proof ((induction M), auto simp del: inferred-clause-sets.simps)
  fix M C assume ind-hyp: ∀ C. redundant C (inferred-clause-sets R S N) → redundant C (inferred-clause-sets R S (N+M::nat))
  assume redundant C (inferred-clause-sets R S N)
  from this and ind-hyp have redundant C (inferred-clause-sets R S (N+M)) by auto
  then have redundant C (add-all-deducible-clauses R (inferred-clause-sets R S (N+M)))
using deducible-clause-preserve-redundancy by auto
then have all-fulfill finite (add-all-deducible-clauses R (inferred-clause-sets R S (N+M)))
  using assms add-all-deducible-clauses-finite [of R S N+M] by auto
from ⟨redundant C (inferred-clause-sets R S N) ⟩ and ind-hyp
  have redundant C (inferred-clause-sets R S (N+M)) by auto
from ⟨redundant C (inferred-clause-sets R S (N+M)) ⟩
  have redundant C (add-all-deducible-clauses R (inferred-clause-sets R S (N+M)))
  using deducible-clause-preserve-redundancy by blast
from this and ⟨all-fulfill finite (add-all-deducible-clauses R (inferred-clause-sets R S (N+M))) ⟩
  have redundant C (simplify (add-all-deducible-clauses R (inferred-clause-sets R S (N+M))))
  using simplify-preserves-redundancy by auto
thus redundant C (inferred-clause-sets R S (Suc (N + M))) by auto
qed

We show that non-persistent clauses are strictly redundant in some element of the sequence.

lemma non-persistent-clauses-are-redundant:
  assumes D ∈ inferred-clause-sets R S N
  assumes D /∈ saturation R S
  assumes all-fulfill finite S
  shows ∃ M. strictly-redundant D (inferred-clause-sets R S M)
proof (rule ccontr)
assume hyp: ¬(∃ M. strictly-redundant D (inferred-clause-sets R S M))
  { fix M
    have D ∈ (inferred-clause-sets R S (N+M))
    proof (induction M)
      show D ∈ inferred-clause-sets R S (N+0) using assms(1) by auto
    next
      fix M assume D ∈ inferred-clause-sets R S (N+M)
      from this have D ∈ add-all-deducible-clauses R (inferred-clause-sets R S (N+M)) by auto
      show D ∈ (inferred-clause-sets R S (N+(Suc M)))
      proof (rule ccontr)
        assume D /∈ (inferred-clause-sets R S (N+(Suc M)))
        from this and ⟨D ∈ add-all-deducible-clauses R (inferred-clause-sets R S (N+M)) ⟩
          have strictly-redundant D (add-all-deducible-clauses R (inferred-clause-sets R S (N+M)))
          using simplify-def by auto
        then have all-fulfill finite (add-all-deducible-clauses R (inferred-clause-sets R S (N+M)))
        using assms(4) assms(3) add-all-deducible-clauses-finite [of R S N+M] by auto
      qed
  }
from this and strictly-redundant \( D \) (add-all-deducible-clauses \( R \) (inferred-clause-sets \( R S (N+M))\))

have strictly-redundant \( D \) (inferred-clause-sets \( R S (N+(Suc M))\))

using simplify-preserves-strict-redundancy by auto

from this and hyp show False by blast
qed

This entails that the clauses that are redundant in some set in the sequence are also redundant in the set of persistent clauses.

lemma persistent-clauses-subsume-redundant-clauses:

assumes redundant \( C \) (inferred-clause-sets \( R S N)\)

assumes all-fulfill finite \( S \)

assumes derived-clauses-are-finite \( R \)

assumes finite \( C \)

shows redundant \( C \) (saturation \( R S)\)

proof

let \( ?\text{n}-order = \{ (x::nat,y::nat). x < y \} \)

{ 
fix \( I \) have \( \forall C N. \) finite \( C \) \( \rightarrow \) card \( C = I \)

\( \rightarrow \) (redundant \( C \) (inferred-clause-sets \( R S N)) \) \( \rightarrow \) redundant \( C \) (saturation \( R S)\) (is \( ?P I \))

proof

((rule wf-induct [of \( ?\text{n}-order \) ?\( P I \)])(simp add:wf))

fix \( I \) assume hyp-induct: \( \forall J. (J,I) \in ?\text{n}-order \rightarrow (?P J)\)

show \( ?P I \)

proof

((rule allI)+(rule_impl)+)

fix \( C N \) assume finite \( C \) card \( C = I \) redundant \( C \) (inferred-clause-sets \( R S N)\)

show redundant \( C \) (saturation \( R S)\)

proof (cases)

assume tautology \( C \)

then show redundant \( C \) (saturation \( R S)\) unfolding redundant-def by auto

next

assume \( \neg\text{tautology} \) \( C \)

from this and redundant \( C \) (inferred-clause-sets \( R S N))\) obtain \( D \)

where subsumes \( D \) \( C \) and \( D \) \( \in \) inferred-clause-sets \( R S N\) unfolding
redundant-def by auto
show redundant C (saturation R S)
proof (cases)
assume D ∈ saturation R S
from this and ⟨subsumes D C⟩ show redundant C (saturation R S)
unfolding redundant-def by auto
next
assume D /∈ saturation R S
from assms (2) assms (3) and ⟨D ∈ inferred-clause-sets R S N⟩ and ⟨D /∈ saturation R S⟩
obtain M where strictly-redundant D (inferred-clause-sets R S M) using

non-persistent-clauses-are-redundant [of D R S] by auto
from ⟨subsumes D C⟩ and (¬tautology C) have ¬tautology D
unfolding subsumes-def tautology-def by auto
from (strictly-redundant D (inferred-clause-sets R S M)) and (¬tautology D)
obtain D’ where D’ ⊂ D and D’ ∈ inferred-clause-sets R S M
unfolding strictly-redundant-def by auto

from ⟨D’ ⊂ D⟩ and ⟨subsumes D C⟩ have D’ ⊂ C unfolding subsumes-def by auto
from ⟨D’ ⊂ C⟩ and ⟨finite C⟩ have finite D’
by (meson psubset-imp-subset rev-finite-subset)
from ⟨D’ ⊂ C⟩ and ⟨finite C⟩ have card D’ < card C
unfolding all-fulfill-def
using psubset-card-mono by auto
from this and ⟨card C = I⟩ have (card D’,I) ∈ ?nat-order by auto
from ⟨D’ ∈ inferred-clause-sets R S M⟩ have redundant D’ (inferred-clause-sets R S M)
unfolding redundant-def subsumes-def by auto
from hyp-induct and ⟨(card D’,I) ∈ ?nat-order⟩ have ?P (card D’) by force
from this and ⟨finite D’⟩ have (redundant D’ (inferred-clause-sets R S M))
have redundant D’ (saturation R S) by auto
show redundant C (saturation R S)
by (meson D’ ⊂ C) (redundant D’ (saturation R S));
psubset-imp-subset subsumes-def subsumption-preserves-redundancy)
qed
qed
qed
qed
}
then show redundant C (saturation R S) using assms(1) assms(4) by blast
qed

We deduce that the set of persistent clauses is saturated.

theorem persistent-clauses-are-saturated:

53
assumes derived-clauses-are-finite R
assumes all-fulfill finite S
shows saturated-binary-rule R (saturation R S)
proof (rule ccontr)
  let ?S = saturation R S
  assume ~saturated-binary-rule R ?S
  then obtain P1 P2 C where R P1 P2 C and P1 ∈ ?S and P2 ∈ ?S and
                  ~redundant C ?S
            unfolding saturated-binary-rule-def by blast
from ⟨P1 ∈ ?S⟩ obtain N1 where i: ∀ M. (M ≥ N1 → P1 ∈ (inferred-clause-sets R S M))
            by auto
from ⟨P2 ∈ ?S⟩ obtain N2 where ii: ∀ M. (M ≥ N2 → P2 ∈ (inferred-clause-sets R S M))
            by auto
let ?N = max N1 N2
have ?N ≥ N1 and ?N ≥ N2 by auto
from this and i have P1 ∈ inferred-clause-sets R S ?N by metis
from ⟨?N ≥ N2⟩ and ii have P2 ∈ inferred-clause-sets R S ?N by metis
from ⟨R P1 P2 C⟩ and ⟨P1 ∈ inferred-clause-sets R S ?N⟩ and ⟨P2 ∈ inferred-clause-sets R S ?N⟩
  have C ∈ all-deducible-clauses R (inferred-clause-sets R S ?N) by auto
from this have C ∈ add-all-deducible-clauses R (inferred-clause-sets R S ?N)
  by auto
from assms have all-fulfill finite (inferred-clause-sets R S ?N)
  using all-inferred-clause-sets-are-finite [of R S ?N] by auto
from assms have all-fulfill finite (add-all-deducible-clauses R (inferred-clause-sets R S ?N))
  using add-all-deducible-clauses-finite by auto
from this and i have redundant C (inferred-clause-sets R S (Suc ?N))
  using simplify-and-membership
  [of add-all-deducible-clauses R (inferred-clause-sets R S ?N)]
  inferred-clause-sets R S (Suc ?N) C
    by auto
have finite P1
  using ⟨P1 ∈ inferred-clause-sets R S (max N1 N2)⟩
  ⟨all-fulfill finite (inferred-clause-sets R S (max N1 N2))⟩ all-fulfill-def by auto

have finite P2
  using ⟨P2 ∈ inferred-clause-sets R S (max N1 N2)⟩
  ⟨all-fulfill finite (inferred-clause-sets R S (max N1 N2))⟩ all-fulfill-def by auto

from ⟨R P1 P2 C⟩ and ⟨finite P1⟩ and ⟨finite P2⟩ and ⟨derived-clauses-are-finite R⟩ have finite C
  unfolding derived-clauses-are-finite-def by blast
from assms this and ⟨redundant C (inferred-clause-sets R S (Suc ?N))⟩
  have redundant C (saturation R S)
  using persistent-clauses-subsume-redundant-clauses [of C R S ?N]
by auto
thus False using (¬redundant C ?S; by auto
qed

Finally, we show that the computed saturated set is equivalent to the initial
formula.

theorem saturation-is-correct:
  assumes Sound R
  assumes derived-clauses-are-finite R
  assumes all-fulfill finite S
  shows equivalent S (saturation R S)
proof –
  have entails-formula S (saturation R S)
  proof (rule ccontr)
    assume ¬ entails-formula S (saturation R S)
    then obtain C where C ∈ saturation R S and ¬ entails S C
    unfolding entails-formula-def by auto
    from :C ∈ saturation R S; obtain N where C ∈ inferred-clause-sets R S N
    by auto
    { fix N
      have entails-formula S (inferred-clause-sets R S N)
      proof (induction N)
        show entails-formula S (inferred-clause-sets R S 0)
        using assms(3) simplify-preserves-semantic validity-implies-entailment by
        auto
        next
        fix N assume entails-formula S (inferred-clause-sets R S N)
        from assms(1) have entails-formula (inferred-clause-sets R S N)
          (add-all-deducible-clauses R (inferred-clause-sets R S N))
          using add-all-deducible-sound by auto
        from this and (entails-formula S (inferred-clause-sets R S N))
        have entails-formula S (add-all-deducible-clauses R (inferred-clause-sets R S N))
          using entails-transitive
          [of S inferred-clause-sets R S N add-all-deducible-clauses R (inferred-clause-sets R S N)]
          by auto
        have inferred-clause-sets R S (Suc N) ⊆ add-all-deducible-clauses R
          (inferred-clause-sets R S N)
          using simplify-def by auto
        then have entails-formula (add-all-deducible-clauses R (inferred-clause-sets R S N))
          (inferred-clause-sets R S (Suc N)) using entailment-subset by auto
        from this and (entails-formula S (add-all-deducible-clauses R (inferred-clause-sets R S N)))
        show entails-formula S (inferred-clause-sets R S (Suc N))
        using entails-transitive [of S add-all-deducible-clauses R (inferred-clause-sets R S N)]
  }
We show that the unrestricted resolution rule is deductive complete, i.e.
that it is able to generate all (prime) implicates of any given clause set.

theory Prime-Implicates

imports Propositional-Resolution

begin

context propositional-atoms

begin

9 Prime Implicates Generation


9.1 Implicates and Prime Implicates

We first introduce the definitions of implicates and prime implicates.

definition implicates :: 'a Formula ⇒ 'a Formula
where implicates S = { C. entails S C }

definition prime-implicates :: 'a Formula ⇒ 'a Formula
where prime-implicates S = simplify (implicates S)

9.2 Generation of Prime Implicates

We introduce a function simplifying a given clause set by evaluating some literals to false. We show that this partial evaluation operation preserves saturatedness and that if the considered set of literals is an implicate of the initial clause set then the partial evaluation yields a clause set that is unsatisfiable. Then the proof follows from refutational completeness: since the partially evaluated set is unsatisfiable and saturated it must contain the empty clause, and therefore the initial clause set necessarily contains a clause subsuming the implicate.

fun partial-evaluation :: 'a Formula ⇒ 'a Literal set ⇒ 'a Formula
where

(partial-evaluation S C) = { E. ∃ D. D ∈ S ∧ E = D¬C ∧ ¬(∃ L. (L ∈ C) ∧ (complement L) ∈ D)}

lemma partial-evaluation-is-saturated :
assumes saturated-binary-rule resolvent S
shows saturated-binary-rule ordered-resolvent (partial-evaluation S C)
proof (rule ccontr)
let ?peval = partial-evaluation S C
assume ¬saturated-binary-rule ordered-resolvent ?peval
from this obtain P1 and P2 and R where P1 ∈ ?peval and P2 ∈ ?peval
and ordered-resolvent P1 P2 R and ¬(tautology R)
and not-subsumed: ¬(∃ D. ((D ∈ (partial-evaluation S C)) ∧ (subsumes D R)))

unfolding saturated-binary-rule-def and redundant-def by auto
from :P1 ∈ ?peval obtain PP1 where PP1 ∈ S and P1 = PP1 − C
and i: ¬(∃ L. (L ∈ C) ∧ (complement L) ∈ PP1) by auto
from :P2 ∈ ?peval obtain PP2 where PP2 ∈ S and P2 = PP2 − C
and ii: ¬(∃ L. (L ∈ C) ∧ (complement L) ∈ PP2) by auto
from :ordered-resolvent P1 P2 R); obtain A where
r-def: R = (P1 − { Pos A }) ∪ (P2 − { Neg A }) and (Pos A) ∈ P1 and
(Neg A) ∈ P2

unfolding ordered-resolvent-def strictly-maximal-literal-def by auto
let ?RR = (PP1 − { Pos A }) ∪ (PP2 − { Neg A }).
from :P1 = PP1 − C; and (Pos A) ∈ P1; have (Pos A) ∈ PP1 by auto
from :P2 = PP2 − C; and (Neg A) ∈ P2; have (Neg A) ∈ PP2 by auto
from r-def and :P1 = PP1 − C; and :P2 = PP2 − C; have R = ?RR −
C by auto
from ⟨(Pos A) ∈ PP1⟩ and ⟨(Neg A) ∈ PP2⟩
  have resolvent PP1 PP2 ?RR unfolding resolvent-def by auto
with ⟨PP1 ∈ S⟩ and ⟨PP2 ∈ S⟩ and ⟨(saturated-binary-rule resolvent S)⟩
  have tautology ?RR ∨ (∃D. (D ∈ S ∧ (subsumes D ?RR)))
unfolding saturated-binary-rule-def redundant-def by auto
thus False
proof
  assume tautology ?RR
  with ⟨R = ?RR − C⟩ and (¬tautology R)
  obtain X where X ∈ C and complement X ∈ PP1 ∪ PP2
  unfolding tautology-def by auto
from ⟨X ∈ C⟩ and ⟨(complement X ∈ PP1 ∪ PP2)⟩ and i and ii
  show False by auto
next
  assume ∃D. ((D ∈ S) ∧ (subsumes D ?RR))
from this obtain D where D ∈ S and subsumes D ?RR
by auto
from ⟨subsumes D ?RR⟩ and ⟨R = ?RR − C⟩
  have subsumes (D − C) R unfolding subsumes-def by auto
from ⟨D ∈ S⟩ and ii and i and ⟨(subsumes D ?RR)⟩ have D − C ∈ ?peval
  unfolding subsumes-def by auto
with ⟨subsumes (D − C) R⟩ and not-subsumed show False by auto
qed
qed

lemma evaluation-wrt-implicate-is-unsat :
  assumes entails S C
  assumes (¬tautology C)
  shows (¬satisfiable (partial-evaluation S C))
proof
  let ?peval = partial-evaluation S C
  assume satisfiable ?peval
  then obtain I where validate-formula I ?peval unfolding satisfiable-def by auto
  let ?J = (I − {X. (Pos X) ∈ C}) ∪ {Y. (Neg Y) ∈ C}
  have (¬validate-clause ?J C)
  proof
    assume validate-clause ?J C
    then obtain L where validate-formula ?J L unfolding validate-clause-def by auto
    obtain X where L = (Pos X) ∨ L = (Neg X) using Literal.exhaust [of L]
    by auto
    from ⟨L = (Pos X) ∨ L = (Neg X)⟩ and ⟨L ∈ C⟩ and (¬tautology C) and
    ⟨validate-clausal ?J L⟩
    show False unfolding tautology-def by auto
  qed
  have validate-formula ?J S
  proof (rule ccontr)
    assume (¬(validate formula ?J S))
    then obtain D where D ∈ S and (¬(validate-clause ?J D)) by auto

58
\[ \langle D \in S \rangle \text{ have } D - C \in \text{ ?peval } \lor (\exists L. (L \in C) \land (\text{complement } L) \in D) \]

by auto
thus False
proof

assume \( \exists L. (L \in C) \land (\text{complement } L) \in D \)
then obtain \( L \) where \( L \in C \) and \( \text{complement } L \in D \) by auto
obtain \( X \) where \( L = (\text{Pos } X) \lor L = (\text{Neg } X) \) using \text{Literal.exhaust} \[ \text{of } L \]
by auto

from this and \( (L \in C) \) and \( \neg (\text{tautology } C) \) have \( \text{validate-literal } ?J \) \( \langle \text{complement } L \rangle \)
unfolding \text{tautology-def} by auto
from \( \langle \text{validate-literal } ?J \ (\text{complement } L) \rangle \) and \( \langle \text{complement } L \rangle \in D \)
and \( \neg (\text{validate-clause } ?J \ (\text{complement } L)) \)
show False by auto

next
assume \( D - C \in \text{ ?peval} \)
from \( \langle D - C \in \text{ ?peval} \rangle \) and \( \langle \text{validate-formula } I \ (\text{complement } L) \rangle \)
have \( \text{validate-clause } I \ (D - C) \) using \text{validate-formula.simps} by blast
from this obtain \( L \) where \( L \in D \) and \( L \notin C \) and \( \text{validate-literal } I \ L \) by auto
obtain \( X \) where \( L = (\text{Pos } X) \lor L = (\text{Neg } X) \) using \text{Literal.exhaust} \[ \text{of } L \]
by auto

from \( \langle L = (\text{Pos } X) \lor L = (\text{Neg } X) \rangle \) and \( \langle \text{validate-literal } I \ L \rangle \) and \( \langle L \notin C \rangle \)
have \( \text{validate-literal } ?J \ L \) unfolding \text{tautology-def} by auto
from \( \langle \neg (\text{validate-clause } ?J \ (\text{complement } L)) \rangle \)
show False by auto
qed

qed

lemma entailment-and-implicates:
assumes \( \text{entails-formula } S1 \subseteq S2 \)
shows \( \text{implicates } S2 \subseteq \text{implicates } S1 \)
using \text{assms entailed-formula-entails-implicates implicates-def} by auto

lemma equivalence-and-implicates:
assumes \( \text{equivalent } S1 \subseteq S2 \)
shows \( \text{implicates } S1 = \text{implicates } S2 \)
using \text{assms entailment-and-implicates equivalent-def} by blast

lemma equivalence-and-prime-implicates:
assumes \( \text{equivalent } S1 \subseteq S2 \)
shows \( \text{prime-implicates } S1 = \text{prime-implicates } S2 \)
using \text{assms equivalence-and-implicates prime-implicates-def} by auto
lemma unrestricted-resolution-is-deductive-complete :
assumes saturated-binary-rule resolvent S
assumes all-fulfill finite S
assumes C ∈ implicates S
shows redundant C S

proof ((cases tautology C),(simp add: redundant-def))
next
assume ¬ tautology C
have ∃ D. (D ∈ S) ∧ (subsumes D C)
proof ¬
let ?peval = partial-evaluation S C
from (saturated-binary-rule resolvent S)
  have saturated-binary-rule ordered-resolvent ?peval
using partial-evaluation-is-saturated by auto
from (C ∈ implicates S) have entails S C unfolding implicates-def by auto
using evaluation-wrt-implicate-is-unsat by auto
from (all-fulfill finite S) have all-fulfill finite ?peval unfolding all-fulfill-def by auto
from (¬satisfiable ?peval) and (saturated-binary-rule ordered-resolvent ?peval)
  and (all-fulfill finite ?peval)
  have {} ∈ ?peval using Complete-def ordered-resolution-is-complete by blast
then show ?thesis unfolding subsumes-def by auto
qed
then show ?thesis unfolding redundant-def by auto
qed

lemma prime-implicates-generation-correct :
assumes saturated-binary-rule resolvent S
assumes non-redundant S
assumes all-fulfill finite S
shows S ⊆ prime-implicates S

proof
fix x assume x ∈ S
show x ∈ prime-implicates S
proof (rule ccontr)
  assume ¬ x ∈ prime-implicates S
  from (x ∈ S) have entails S x unfolding entails-def implicates-def by auto
  then have x ∈ implicates S unfolding implicates-def by auto
  with (¬ x ∈ (prime-implicates S)) have strictly-redundant x (implicates S)
  unfolding prime-implicates-def simplify-def by auto
  from this have tautology x ∨ (∃ y. (y ∈ (implicates S)) ∧ (y ⊂ x)) unfolding strictly-redundant-def by auto
  then have strictly-redundant x S
  proof ((cases tautology x),(simp add: strictly-redundant-def))
  next
  assume ¬tautology x
  with (tautology x ∨ (∃ y. (y ∈ (implicates S)) ∧ (y ⊂ x)))
    obtain y where y ∈ implicates S and y ⊂ x by auto

60
\begin{verbatim}
from (y ∈ implicates S) and (saturated-binary-rule resolvent S) and (all-fulfill finite S)
have redundant y S using unrestricted-resolution-is-deductive-complete by auto
from (y ⊂ x) and (¬tautology x) have ¬tautology y unfolding tautology-def by auto
with (redundant y S) obtain z where z ∈ S and z ⊆ y
unfolding redundant-def subsumes-def by auto
with (y ⊂ x) have z ⊂ x by auto
with (z ∈ S) show strictly-redundant x S using strictly-redundant-def by auto
qed
with (non-redundant S) and (x ∈ S) show False unfolding non-redundant-def by auto
qed
qed

theorem prime-implicates-of-saturated-sets:
assumes saturated-binary-rule resolvent S
assumes all-fulfill finite S
assumes non-redundant S
shows S = prime-implicates S
proof
from assms show S ⊆ prime-implicates S using prime-implicates-generation-correct by auto
show prime-implicates S ⊆ S
proof
fix x assume x ∈ prime-implicates S
from this have x ∈ implicates S unfolding prime-implicates-def simplify-def by auto
with assms have redundant x S
using unrestricted-resolution-is-deductive-complete by auto
show x ∈ S
proof (rule ccontr)
assume x \notin S
with (redundant x S) have strictly-redundant x S
unfolding redundant-def strictly-redundant-def subsumes-def by auto
with (S ⊆ prime-implicates S) have strictly-redundant x (prime-implicates S)
unfolding strictly-redundant-def by auto
then have strictly-redundant x (implicates S)
unfolding strictly-redundant-def prime-implicates-def simplify-def by auto
with (x ∈ prime-implicates S) show False
unfolding prime-implicates-def simplify-def by auto
qed
qed
qed
\end{verbatim}
9.3 Incremental Prime Implicates Computation

We show that it is possible to compute the set of prime implicates incrementally i.e., to fix an ordering among atoms, and to compute the set of resolvents upon each atom one by one, without backtracking (in the sense that if the resolvents upon a given atom are generated at some step $i$ then no resolvents upon the same atom are generated at step $i < j$. This feature is critical in practice for the efficiency of prime implicates generation algorithms.

We first introduce a function computing all resolvents upon a given atom.

**definition** all-resolvents-upon :: 'at Formula ⇒ 'at ⇒ 'at Formula

where \( (\text{all-resolvents-upon } S A) = \{ C. \exists P1 P2. P1 \in S \land P2 \in S \land C = (\text{resolvent-upon } P1 P2 A) \} \)

**lemma** resolvent-upon-correct:
- assumes $P1 \in S$
- assumes $P2 \in S$
- assumes $C = \text{resolvent-upon } P1 P2 A$
- shows entails $S C$

**proof** cases
- assume $\text{Pos } A \in P1 \land \text{Neg } A \in P2$
  - with $\langle C = \text{resolvent-upon } P1 P2 A \rangle$
    - unfolding resolvent-def by auto
  - with $\langle P1 \in S \rangle$ and $\langle P2 \in S \rangle$ show ?thesis
    - using soundness-and-entailment resolution-is-correct by auto
- next
  - assume $\neg (\text{Pos } A \in P1 \land \text{Neg } A \in P2)$
  - with $\langle C = \text{resolvent-upon } P1 P2 A \rangle$
    - have $P1 \subseteq C \lor P2 \subseteq C$ by auto
  - with $\langle P1 \in S \rangle$ and $\langle P2 \in S \rangle$ have redundant $C S$
    - unfolding redundant-def subsumes-def by auto
  - then show ?thesis using redundancy-implies-entailment by auto
qed

**lemma** all-resolvents-upon-is-finite:
- assumes all-fulfill finite $S$
- shows all-fulfill finite $(S \cup (\text{all-resolvents-upon } S A))$

**using** assms unfolding all-fulfill-def all-resolvents-upon-def by auto

**lemma** atoms-formula-resolvents:
- shows atoms-formula $(\text{all-resolvents-upon } S A) \subseteq \text{atoms-formula } S$
- unfolding all-resolvents-upon-def by auto

We define a partial saturation predicate that is restricted to a specific atom.

**definition** partial-saturation :: 'at Formula ⇒ 'at ⇒ 'at Formula ⇒ bool

where
\[
(\text{partial-saturation } S A R) = (\forall P1 P2. (P1 \in S \rightarrow P2 \in S \rightarrow (\text{redundant} (\text{resolvent-upon } P1 P2 A) R)))
\]
We show that the resolvent of two redundant clauses in a partially saturated set is itself redundant.

**Lemma** resolvent-upon-and-partial-saturation:

assumes redundant P1 S
assumes redundant P2 S
assumes partial-saturation S A (S ∪ R)
assumes C = resolvent-upon P1 P2 A
shows redundant C (S ∪ R)

**Proof** (rule condr)

assume ¬redundant C (S ∪ R)
from ⟨C = resolvent-upon P1 P2 A, have C = (P1 − {Pos A}) ∪ (P2 − {Neg A}) ⟩ by auto
from ⟨¬redundant C (S ∪ R), have ¬tautology C unfolding redundant-def by auto ⟩
have ¬(tautology P1)
proof
assume tautology P1
then obtain B where Pos B ∈ P1 and Neg B ∈ P1 unfolding tautology-def by auto
show False
proof cases
assume A = B
with ⟨Neg B ∈ P1⟩ and ⟨C = (P1 − {Pos A}) ∪ (P2 − {Neg A})⟩ have subsumes P2 C
unfolding subsumes-def using Literal.distinct by blast
with ⟨redundant P2 S⟩ have redundant C S
using subsumption-preserves-redundancy by auto
with ⟨¬redundant C (S ∪ R), show False unfolding redundant-def by auto⟩
next
assume A ≠ B
with ⟨C = (P1 − {Pos A}) ∪ (P2 − {Neg A})⟩ and ⟨Pos B ∈ P1⟩ and ⟨Neg B ∈ P1⟩ have Pos B ∈ C and Neg B ∈ C by auto
with ⟨¬redundant C (S ∪ R), show False unfolding tautology-def redundant-def by auto⟩
qed

qed

with ⟨redundant P1 S⟩ obtain Q1 where Q1 ∈ S and subsumes Q1 P1
unfolding redundant-def by auto
have ¬(tautology P2)
proof
assume tautology P2
then obtain B where Pos B ∈ P2 and Neg B ∈ P2 unfolding tautology-def by auto
show False
proof cases
assume A = B
with ⟨Pos B ∈ P2⟩ and ⟨C = (P1 − {Pos A}) ∪ (P2 − {Neg A})⟩ have subsumes P1 C

63
unfolding subsumes-def using Literal, distinct by blast
with ⟨redundant P1 S⟩ have redundant C S
using subsumption-preserves-redundancy by auto
with ⟨¬redundant C (S ∪ R)⟩ show False unfolding redundant-def by auto
next
assume A ≠ B
with ⟨C = (P1 - { Pos A }) ∪ (P2 - { Neg A })⟩ and ⟨Pos B ∈ P2⟩ and ⟨Neg B ∈ P2⟩
have Pos B ∈ C and Neg B ∈ C by auto
with ⟨¬redundant C (S ∪ R)⟩ show False unfolding tautology-def redundant-def by auto
qed
definition in-all-resolvents-upon:: 'at Formula ⇒ 'at ⇒ 'at Clause ⇒ bool
where
in-all-resolvents-upon S A C = (∃ P1 P2. (P1 ∈ S ∧ P2 ∈ S ∧ C = resolvent-upon P1 P2 A))

lemma every-clause-is-a-resolvent:
assumes all-fulfill (in-all-resolvents-upon S A) R
assumes all-fulfill (λx. ¬(tautology x)) S
assumes P1 ∈ S ∪ R
shows in-all-resolvents-upon S A P1
proof ((cases P1 ∈ R).(metis all-fulfill-assms(1)))
next
assume P1 /∈ R
with ⟨P1 ∈ S ∪ R⟩ have P1 ∈ S by auto
with ⟨(all-fulfill (λx. ¬(tautology x)) S) ⟩ have ¬ tautology P1
unfolding all-fulfill-def by auto
from ⟨¬ tautology P1⟩ have Neg A /∈ P1 ∨ Pos A /∈ P1 unfolding tautology-def by auto

We show that if R is a set of resolvents of a set of clauses S then the same holds for S ∪ R. For the clauses in S, the premises are identical to the resolvent and the inference is thus redundant (this trick is useful to simplify proofs).

definition in-all-resolvents-upon:: 'at Formula ⇒ 'at ⇒ 'at Clause ⇒ bool
where
in-all-resolvents-upon S A C = (∃ P1 P2. (P1 ∈ S ∧ P2 ∈ S ∧ C = resolvent-upon P1 P2 A))

lemma every-clause-is-a-resolvent:
assumes all-fulfill (in-all-resolvents-upon S A) R
assumes all-fulfill (λx. ¬(tautology x)) S
assumes P1 ∈ S ∪ R
shows in-all-resolvents-upon S A P1
proof ((cases P1 ∈ R).(metis all-fulfill-assms(1)))
next
assume P1 /∈ R
with ⟨P1 ∈ S ∪ R⟩ have P1 ∈ S by auto
with ⟨(all-fulfill (λx. ¬(tautology x)) S) ⟩ have ¬ tautology P1
unfolding all-fulfill-def by auto
from ⟨¬ tautology P1⟩ have Neg A /∈ P1 ∨ Pos A /∈ P1 unfolding tautology-def by auto

64
from this have $P_1 = (P_1 - \{ \text{Pos } A \}) \cup (P_1 - \{ \text{Neg } A \})$ by auto
with $(P_1 \in S)$ show $\text{thesis unfolding resolvent-def}$
unfolding $\text{in-all-resolvents-upon-def}$ by auto

qed

We show that if a formula is partially saturated then it stays so when new resolvents are added in the set.

**lemma partial-saturation-is-preserved :**

assumes $\text{partial-saturation } S \ E_1 \ S$
assumes $\text{partial-saturation } S \ E_2 \ (S \cup R)$
assumes $\text{all-fulfill } (\lambda x. \neg(\text{tautology } x)) \ S$
assumes $\text{all-fulfill in-all-resolvents-upon } S \ E_2 \ R$
shows $\text{partial-saturation } (S \cup R) \ E_1 \ (S \cup R)$

**proof (rule ccontr)**

assume $\neg \text{partial-saturation } (S \cup R) \ E_1 \ (S \cup R)$
from this obtain $P_1 \ P_2 \ C$ where $P_1 \in S \cup R$ and $P_2 \in S \cup R$ and $C = \text{resolvent-upon } P_1 \ P_2 \ E_1$
and $\neg \text{redundant } C \ (S \cup R)$

unfolding $\text{partial-saturation-def}$ by auto
from $(C = \text{resolvent-upon } P_1 \ P_2 \ E_1)$ have $C = (P_1 - \{ \text{Pos } E_1 \}) \cup (P_2 - \{ \text{Neg } E_1 \})$ by auto
from $(P_1 \in S \cup R \land \text{assms} (4))$ and $(\text{all-fulfill } (\lambda x. \neg(\text{tautology } x)) \ S)$
have $\text{in-all-resolvents-upon } S \ E_2 \ P_1$ using $\text{every-clause-is-a-resolvent}$ by auto
then obtain $P_1-1 \ P_1-2$ where $P_1-1 \in S$ and $P_1-2 \in S$ and $P_1 = \text{resolvent-upon } P_1-1 \ P_1-2 \ E_2$

using $\text{every-clause-is-a-resolvent unfolding in-all-resolvents-upon-def}$ by blast
from $(P_2 \in S \cup R \land \text{assms} (4))$ and $(\text{all-fulfill } (\lambda x. \neg(\text{tautology } x)) \ S)$
have $\text{in-all-resolvents-upon } S \ E_2 \ P_2$ using $\text{every-clause-is-a-resolvent}$ by auto
then obtain $P_2-1 \ P_2-2$ where $P_2-1 \in S$ and $P_2-2 \in S$ and $P_2 = \text{resolvent-upon } P_2-1 \ P_2-2 \ E_2$

using $\text{every-clause-is-a-resolvent unfolding in-all-resolvents-upon-def}$ by blast
let $?R_1 = \text{resolvent-upon } P_1-1 \ P_2-1 \ E_1$
from $(\text{partial-saturation } S \ E_1 \ S) \land (P_1-1 \in S) \land (P_2-1 \in S)$ have redundant $?R_1 \ S$

unfolding $\text{partial-saturation-def}$ by auto
let $?R_2 = \text{resolvent-upon } P_1-2 \ P_2-2 \ E_1$
from $(\text{partial-saturation } S \ E_1 \ S) \land (P_1-2 \in S) \land (P_2-2 \in S)$ have redundant $?R_2 \ S$

unfolding $\text{partial-saturation-def}$ by auto
let $?C = \text{resolvent-upon } ?R_1 \ ?R_2 \ E_2$
from $(C = \text{resolvent-upon } P_1 \ P_2 \ E_1) \land (P_2 = \text{resolvent-upon } P_2-1 \ P_2-2 \ E_2)$
and $(P_1 = \text{resolvent-upon } P_1-1 \ P_1-2 \ E_2)$
have $?C = C$ by auto
with $(\text{redundant } ?R_1 \ S) \land (\text{redundant } ?R_2 \ S) \land (\text{partial-saturation } S \ E_2 \ (S \cup R))$
and $(\neg \text{redundant } C \ (S \cup R))$
show False using $\text{resolvent-upon-and-partial-saturation}$ by auto

qed
The next lemma shows that the clauses inferred by applying the resolution rule upon a given atom contain no occurrence of this atom, unless the inference is redundant.

**Lemma** resolvents-do-not-contain-atom:
- Assumes \( \neg \text{tautology } P_1 \)
- Assumes \( \neg \text{tautology } P_2 \)
- Assumes \( C = \text{resolvent-upon } P_1 P_2 E_2 \)
- Assumes \( \neg \text{subsumes } P_1 C \)
- Assumes \( \neg \text{subsumes } P_2 C \)
- Shows \( (\neg E_2) \notin C \land (\text{Pos } E_2) \notin C \)

**Proof**

From \((C = \text{resolvent-upon } P_1 P_2 E_2)\) have \( C = (P_1 - \{\text{Pos } E_2\}) \cup (P_2 - \{\neg E_2\}) \)

by auto

Show \((\neg E_2) \notin C\)

Proof

Assume \(\neg E_2 \in C\)

From \((C = \text{resolvent-upon } P_1 P_2 E_2)\) have \( C = (P_1 - \{\text{Pos } E_2\}) \cup (P_2 - \{\neg E_2\}) \)

by auto

With \((\neg E_2 \in C)\) have \(\neg E_2 \in P_1\) by auto

From \((\neg \text{subsumes } P_1 C) \land (C = (P_1 - \{\text{Pos } E_2\}) \cup (P_2 - \{\neg E_2\}))\) have \(\text{Pos } E_2 \in P_1\)

unfolding subsumes-def by auto

From \((\neg E_2 \in P_1) \land (\text{Pos } E_2 \in P_1) \land (\neg \text{tautology } P_1)\) show False

unfolding tautology-def by auto

QEd

Next show \((\text{Pos } E_2) \notin C\)

Proof

Assume \(\text{Pos } E_2 \in C\)

From \((C = \text{resolvent-upon } P_1 P_2 E_2)\) have \( C = (P_1 - \{\text{Pos } E_2\}) \cup (P_2 - \{\neg E_2\}) \)

by auto

With \((\text{Pos } E_2 \in C)\) have \(\text{Pos } E_2 \in P_2\) by auto

From \((\neg \text{subsumes } P_2 C) \land (C = (P_1 - \{\text{Pos } E_2\}) \cup (P_2 - \{\neg E_2\}))\) have \(\neg E_2 \in P_2\)

unfolding subsumes-def by auto

From \((\neg E_2 \in P_2) \land (\text{Pos } E_2 \in P_2) \land (\neg \text{tautology } P_2)\) show False

unfolding tautology-def by auto

QEd

QEd

The next lemma shows that partial saturation can be ensured by computing all (non-redundant) resolvents upon the considered atom.

**Lemma** ensures-partial-saturation:
- Assumes partial-saturation \( S E_2 (S \cup R) \)
- Assumes all-fulfill \((\lambda x. \neg(\text{tautology } x)) S\)
- Assumes all-fulfill \((\text{in-all-resolvents-upon } S E_2) R\)
- Assumes all-fulfill \((\lambda x. \neg(\text{redundant } x S)) R\)
shows partial-saturation \((S \cup R) E2 (S \cup R)\)

**proof (rule ccontr)**

**assume** \(\neg\) partial-saturation \((S \cup R) E2 (S \cup R)\)

**from this obtain** \(P1 P2 C\) **where** \(P1 \in S \cup R\) and \(P2 \in S \cup R\) and \(C = \) resolvent-upon \(P1 P2 E2\)

and \(\neg\) redundant \(C (S \cup R)\)

**unfolding** partial-saturation-def by auto

**have** \(P1 \in S\)

**proof (rule ccontr)**

**assume** \(P1 /\in S\)

**with** \(P1 \in S \cup R\) have \(P1 \in R\) by auto

**with assms(3) obtain** \(P1-1\) and \(P1-2\) **where** \(P1-1 \in S\) and \(P1-2 \in S\)

and \(P1 = \) resolvent-upon \(P1-1 P1-2 E2\)

**unfolding** all-fulfill-def in-all-resolvents-upon-def by auto

**from** \((\text{all-fulfill} (\lambda x. \neg(\text{tautology} x))) S\) \(\text{and}\) \((P1-1 \in S)\) \(\text{and}\) \((P1-2 \in S)\)

**have** \(\neg\) tautology \(P1-1\) and \(\neg\) tautology \(P1-2\)

**unfolding** all-fulfill-def by auto

**from** \((\text{all-fulfill} (\lambda x. \neg(\text{redundant} x S))) R\) \(\text{and}\) \((P1 \in R)\) \(\text{and}\) \((P1-1 \in S)\) \(\text{and}\) \((P1-2 \in S)\)

**have** \(\neg\) subsumes \(P1-1 P1\) and \(\neg\) subsumes \(P1-2 P1\)

**unfolding** redundant-def all-fulfill-def by auto

**from** \((\neg\) tautology \(P1-1)\) \(\neg\) tautology \(P1-2)\) \(\neg\) subsumes \(P1-1 P1\) and \(\neg\) subsumes \(P1-2 P1\)

and \(P1 = \) resolvent-upon \(P1-1 P1-2 E2\)

**have** \((\neg E2) /\in P1\) and \((\text{Pos} E2) /\in P1\)

**using** resolvents-do-not-contain-atom [of \(P1-1 P1-2 P1 E2\)] by auto

**with** \((C = \) resolvent-upon \(P1 P2 E2\)) have subsumes \(P1 C\) unfolding subsumes-def by auto

**with** \((\neg\) redundant \(C (S \cup R)\) and \((P1 \in S \cup R)\) show False unfolding redundant-def by auto

**qed**

**have** \(P2 \in S\)

**proof (rule ccontr)**

**assume** \(P2 /\in S\)

**with** \(P2 \in S \cup R\) have \(P2 \in R\) by auto

**with assms(3) obtain** \(P2-1\) and \(P2-2\) **where** \(P2-1 \in S\) and \(P2-2 \in S\)

and \(P2 = \) resolvent-upon \(P2-1 P2-2 E2\)

**unfolding** all-fulfill-def in-all-resolvents-upon-def by auto

**from** \((\text{all-fulfill} (\lambda x. \neg(\text{tautology} x))) S\) \(\text{and}\) \((P2-1 \in S)\) \(\text{and}\) \((P2-2 \in S)\)

**have** \(\neg\) tautology \(P2-1\) and \(\neg\) tautology \(P2-2\)

**unfolding** all-fulfill-def by auto

**from** \((\neg\) tautology \(P2-1)\) \(\neg\) tautology \(P2-2)\) \(\neg\) subsumes \(P2-1 P2\) and \(\neg\) subsumes \(P2-2 P2\)

and \((P2 = \) resolvent-upon \(P2-1 P2-2 E2\)\).

67
have \((\neg E2) \notin P2 \land (\text{Pos } E2) \notin P2\)

using resolvents-do-not-contain-atom [of \(P2-1 \ P2-2 \ P2 \ E2\)] by auto

with \(\langle C = \text{resolvent-upon } P1 \ P2 \ E2; \ \text{have subsumes } P2 \ C \ \text{unfolding } \text{subsumes-def}\rangle\) by auto

with \(\langle \neg \text{redundant } C (S \cup R); \ \text{and } (P2 \in S \cup R)\rangle\) by auto

show False unfolding redundant-def by auto

qed

lemma resolvents-preserve-equivalence:
shows equivalent \(S \ (S \cup (\text{all-resolvents-upon } S \ A))\)

proof –

have \(S \subseteq (S \cup (\text{all-resolvents-upon } S \ A))\) by auto

then have entails-formula \((S \cup (\text{all-resolvents-upon } S \ A)) \ S\) using entailment-subset by auto

have entails-formula \(S \ (S \cup (\text{all-resolvents-upon } S \ A))\)

proof (rule ccontr)

assume \(\neg\text{entails-formula } S \ (S \cup (\text{all-resolvents-upon } S \ A))\)

from this obtain \(C \ \text{where } C \in (\text{all-resolvents-upon } S \ A)\ \text{and } \neg\text{entails } S \ C\)

unfolding entails-formula-def using entails-member by auto

from \(\langle C = \text{resolvent-upon } P1 \ P2 \ A; \ \text{and } P1 \in S \ \text{and } P2 \in S \rangle\) obtain \(P1 \ P2\)

where \(C = \text{resolvent-upon } P1 \ P2 \ A\) and \(P1 \in S \ \text{and } P2 \in S\)

unfolding all-resolvents-upon-def by auto

from \(\langle C = \text{resolvent-upon } P1 \ P2 \ A; \ \text{and } P1 \in S \ \text{and } P2 \in S \rangle\) have entails \(S \ C\)

using resolvent-upon-correct by auto

with \(\langle \neg\text{entails } S \ C\rangle\) show False by auto

qed

from \(\langle\text{entails-formula } (S \cup (\text{all-resolvents-upon } S \ A)); \ \text{and } \langle\text{entails-formula } (S \cup (\text{all-resolvents-upon } S \ A))\rangle\) show \(\exists\text{thesis unfolding equivalent-def by auto}\)

qed

Given a sequence of atoms, we define a sequence of clauses obtained by resolving upon each atom successively. Simplification rules are applied at each iteration step.

fun resolvents-sequence :: \(\text{nat} \Rightarrow '\text{at } \text{Formula} \Rightarrow \text{nat} \Rightarrow '\text{at } \text{Formula}\)

where
\(((\text{resolvents-sequence } A \ S \ 0) = (\text{simplify } S)); \ |
\ (\text{resolvents-sequence } A \ S \ (\text{Suc } N)) =
\ (\text{simplify } ((\text{resolvents-sequence } A \ S \ N)
\ \cup (\text{all-resolvents-upon } (\text{resolvents-sequence } A \ S \ N) \ (A \ N))))\)

The following lemma states that partial saturation is preserved by simplification.

lemma redundancy-implies-partial-saturation:
assumes partial-saturation \( S1 \) \( A \) \( S1 \)
assumes \( S2 \subseteq S1 \)
assumes all-fulfill (\( \lambda x. \) redundant \( x \) \( S2 \)) \( S1 \)
shows partial-saturation \( S2 \) \( A \) \( S2 \)

proof (rule ccontr)
assume \( \neg \) partial-saturation \( S2 \) \( A \) \( S2 \)
then obtain \( P1 \) \( P2 \) \( C \) where \( P1 \in S2 \) \( P2 \in S2 \) and \( C = (\text{resolvent-upon} \ P1 \ P2 \ A) \)
and \( \neg \) redundant \( C \) \( S2 \)

unfolding partial-saturation-def by auto
from \( \langle P1 \in S2 \rangle \) and \( \langle S2 \subseteq S1 \rangle \) have \( P1 \in S1 \) by auto
from \( \langle P2 \in S2 \rangle \) and \( \langle S2 \subseteq S1 \rangle \) have \( P2 \in S1 \) by auto
from \( \langle P1 \in S1 \rangle \) and \( \langle P2 \in S1 \rangle \) and (partial-saturation \( S1 \) \( A \) \( S1 \)); and \( \langle C = \text{resolvent-upon} \ P1 \ P2 \ A \rangle \)

have redundant \( C \) \( S1 \) unfolding partial-saturation-def by auto
from \( \langle \neg \) redundant \( C \) \( S2 \rangle \) have \( \neg \)tautology \( C \) unfolding redundant-def by auto
with \( \langle \neg \) redundant \( C \) \( S1 \rangle \) obtain \( D \) where \( D \in S1 \) and \( D \subseteq C \)
unfolding redundant-def subsumes-def by auto
from \( \langle D \in S1 \rangle \) and \( \langle \text{all-fulfill} \ (\lambda x. \) redundant \( x \) \( S2 \)) \( S1 \rangle \) have redundant \( D \) \( S2 \)
unfolding all-fulfill-def by auto
from \( \langle \neg \) tautology \( C \rangle \) and \( \langle D \subseteq C \rangle \) have \( \neg \) tautology \( D \) unfolding tautology-def by auto

unfolding redundant-def subsumes-def by auto

have redundant \( D \) \( S2 \) unfolding redundant-def subsumes-def by auto
from \( \langle E \subseteq D \rangle \) and \( \langle D \subseteq C \rangle \) have \( E \subseteq C \) by auto
from \( \langle E \in S2 \rangle \) and \( \langle E \subseteq C \rangle \) and \( \langle \neg \) redundant \( C \) \( S2 \rangle \) show False
unfolding redundant-def subsumes-def by auto

qed

The next theorem finally states that the implicate generation algorithm is sound and complete in the sense that the final clause set in the sequence is exactly the set of prime implicates of the considered clause set.

theorem incremental-prime-implication-generation:
assumes atoms-formula \( S \) = \( \{ X. \exists I :: \text{nat}. I < N \land X = (A I) \} \)
assumes all-fulfill finite \( S \)
shows (prime-implicates \( S \)) = (resolvents-sequence \( A \) \( S \) \( N \))

proof –

We define a set of invariants and show that they are satisfied by all sets in the above sequence. For the last set in the sequence, the invariants ensure that the clause set is saturated, which entails the desired property.

let \( ?\text{Final} = \text{resolvents-sequence} \ A \ S \ N \)

We define some properties and show by induction that they are satisfied by all the clause sets in the constructed sequence

let \( ?\text{equiv-init} = \lambda I. (\text{equivalent} \ S \ (\text{resolvents-sequence} \ A \ S \ I)) \)
let \( ?\text{partial-saturation} = \lambda I. (\forall J :: \text{nat}. (J < I \rightarrow (\text{partial-saturation} \ (\text{resolvents-sequence} \ A \ S \ I) \ (A J) \ (\text{resolvents-sequence} \ A \ S \ I)))) \)

69
let \( ?\text{no-tautologies} = \lambda I. (\text{all-fulfill} (\lambda x. \neg(\text{tautology} x)) (\text{resolvents-sequence} A S I)) \)
let \( ?\text{atoms-init} = \lambda I. (\text{atoms-formula} (\text{resolvents-sequence} A S I) \subseteq \{ X. \exists I::\text{nat}. \, I < N \land X = (A I)\}) \)
let \( ?\text{non-redundant} = \lambda I. (\text{non-redundant} (\text{resolvents-sequence} A S I)) \)
let \( ?\text{finite} = \lambda I. (\text{all-fulfill finite} (\text{resolvents-sequence} A S I)) \)

have \( \forall I. (I \leq N \longrightarrow (\text{equiv-init} I) \land (\text{partial-saturation} I) \land (\text{no-tautologies} I) \land (\text{atoms-init} I) \land (\text{non-redundant} I) \land (\text{finite} I)) \)

proof (rule allI)
fix \( I \)
show \( (I \leq N \longrightarrow ?P I) \)
proof (rule impI)+
assume \( 0 \leq N \)
let \( ?R = \text{resolvents-sequence} A S 0 \)
from \( \text{all-fulfill finite} S \)
have \( ?\text{equiv-init} 0 \) using simplify-preserves-equivalence by auto
moreover have \( ?\text{no-tautologies} 0 \) using simplify-def strictly-redundant-def all-fulfill-def by auto
moreover have \( ?\text{partial-saturation} 0 \) by auto
moreover from \( \text{all-fulfill finite} S \) have \( ?\text{finite} 0 \) using simplify-finite by auto
moreover have \( \text{atoms-formula} ?R \subseteq \text{atoms-formula} S \) using atoms-formula-simplify by auto
moreover with \( \text{atoms-formula} S = \{ X. \exists I::\text{nat}. \, I < N \land X = (A I)\} \)
| have \( v::\text{nat} \) unfolding simplify-def by auto
moreover have \( ?\text{non-redundant} 0 \) using simplify-non-redundant by auto
ultimately show \( ?P 0 \) by auto
qed

We then show that the properties are preserved by induction.

next
fix \( I \) assume \( I \leq N \longrightarrow ?P I \)
show \( (\text{Suc} I) \leq N \longrightarrow (?P (\text{Suc} I)) \)
proof (rule impI)+
assume \( (\text{Suc} I) \leq N \)
let \( ?\text{Rec} = \text{resolvents-sequence} A S I \)
let \( ?R = \text{resolvents-sequence} A S (\text{Suc} I) \)
from $\langle \text{Suc } I \leq N \rangle$ and $\langle I \leq N \rightarrow ?P I \rangle$
have $?\text{equiv-init } I$ and $?\text{partial-saturation } I$ and $?\text{no-tautologies } I$ and $?\text{finite } I$
and $?\text{atoms-init } I$ and $?\text{non-redundant } I$ by auto
have equivalent $?\text{Prec } (?\text{Prec } \cup (\text{all-resolvents-upon } ?\text{Prec } (A I)))$
using resolvents-preserve-equivalence by auto
from $?\text{finite } I$ have all-fulfill finite $?(?\text{Prec } \cup (\text{all-resolvents-upon } ?\text{Prec } (A I)))$
using all-resolvents-upon-is-finite by auto
then have all-fulfill finite $(?\text{Prec } \cup (\text{all-resolvents-upon } ?\text{Prec } (A I)))$
proof (rule ccontr)
assume $\neg$all-fulfill finite $(?\text{Prec } \cup (\text{all-resolvents-upon } ?\text{Prec } (A I)))$
then obtain $x$ where $x \in ?\text{Prec } \cup ?\text{Delta}$ unfolding all-fulfill-def
by auto
from $\neg$redundant $x$ $?R$ have $\neg x \in ?R$ unfolding redundant-def
subsames-def by auto
with $x \in ?\text{Prec } \cup ?\text{Delta}$ have $x \in (?\text{Prec } \cup (\text{all-resolvents-upon } ?\text{Prec } (A I)))$
by auto
with all-fulfill finite $(?\text{Prec } \cup (\text{all-resolvents-upon } ?\text{Prec } (A I)))$
have redundant $x$ $(?\text{Prec } \cup (\text{all-resolvents-upon } ?\text{Prec } (A I)))$
using simplify-and-membership by blast
with $\neg$redundant $x$ $?R$ show False by auto
qed
have all-fulfill (in-all-resolvents-upon $?\text{Prec } (A I)$) $?\text{Delta}$
proof (rule ccontr)
assume $\neg$ (all-fulfill (in-all-resolvents-upon $?\text{Prec } (A I)$) $?\text{Delta}$)
then obtain $C$ where $C \in ?\text{Delta}$
and $\neg$in-all-resolvents-upon $?\text{Prec } (A I)$ $C$
unfolding all-fulfill-def by auto
then obtain $C$ where $C \in ?\text{Delta}$
and not-res: $\forall P1 P2. \neg(P1 \in ?\text{Prec } \land P2 \in ?\text{Prec } \land C = \text{resolvent-upon}$
P1 P2 (A I))

unfolding all-fulfill-def in-all-resolvents-upon-def by blast
from \((C \in ?Delta)\) have \(C \in ?R\) and \(C \notin ?Prec\) by auto
then have \(C \in \text{simplify} \ (\text{?Prec} \cup \text{all-resolvents-upon} \ ?Prec\ (A I))\) by auto
then have \(C \in \text{?Prec} \cup \text{(all-resolvents-upon} \ ?Prec\ (A I))\) unfolding simplify-def by auto
with \((C \notin ?Prec)\) have \(C \in \text{(all-resolvents-upon} \ ?Prec\ (A I))\) by auto
with not-res show False unfolding all-resolvents-upon-def by auto
qed

have all-fulfill \((\lambda x. \ (\neg \text{redundant} \ x \ ?Prec))\) ?Delta
proof (rule ccontr)
assume \(\neg\text{all-fulfill} \ (\lambda x. \ (\neg \text{redundant} \ x \ ?Prec))\) ?Delta
then obtain \(C\) where \(C \in ?Delta\) and \(\text{redundant \ redundant} \ ?Prec\) unfolding all-fulfil-def by auto
from \((C \in ?Delta)\) have \(\text{C \in \ ?R\ and \ C \notin \ ?Prec\ by auto}\)
show False proof cases
assume \(\text{strictly-redundant} \ ?Prec\)
then have \(\text{strictly-redundant} \ ?Prec\ ((\text{?Prec} \cup \text{all-resolvents-upon} \ ?Prec\ (A I)))\) unfolding strictly-redundant-def by auto
then have \(\text{C \notin \ ?R\ by auto}\)
with \((C \in ?R)\) show False by auto
next assume \(\neg\text{strictly-redundant} \ ?Prec\)
with redundant have \(C \in ?Prec\)
unfolding strictly-redundant-def redundant-def subsumes-def by auto
with \((C \notin ?Prec)\) show False by auto
qed

have \(\forall \ J::\text{nat.} \ (J < (\text{Suc} \ I)) \rightarrow (\text{partial-saturation} \ ?R\ (A J) \ ?R)\)
proof (rule ccontr)
assume \(\neg(\forall \ J::\text{nat.} \ (J < (\text{Suc} \ I)) \rightarrow (\text{partial-saturation} \ ?R\ (A J) \ ?R))\)
then obtain \(J\) where \(J < (\text{Suc} \ I)\) and \(\neg(\text{partial-saturation} \ ?R\ (A J) \ ?R)\) by auto
from \(\neg(\text{partial-saturation} \ ?R\ (A J) ?R)\) obtain \(P1 \ P2 \ C\)
where \(\text{P1 \in \ ?R\ and \ P2 \in ?R\ and \ C \in \text{resolvent-upon} \ P1 \ P2\ (A J)}\)
and \(\neg\text{redundant} \ ?R\)
unfolding partial-saturation-def by auto
have partial-saturation ?Prec (A I) (?Prec \cup ?Delta)
proof (rule ccontr)
assume \(\neg\text{partial-saturation} \ ?Prec\ (A I) \ (\text{?Prec} \cup \text{?Delta})\)
then obtain \(P1 \ P2 \ C\) where \(P1 \in ?Prec\) and \(P2 \in ?Prec\)
and \(C \in \text{resolvent-upon} \ P1 \ P2\ (A I)\) and
\(\neg\text{redundant} \ ?Prec\ (\text{?Prec} \cup \text{?Delta)}\) unfolding partial-saturation-def by auto
from \(C \in \text{resolvent-upon} \ P1 \ P2\ (A I)\) and \(P1 \in ?Prec\) and \(P2 \in ?Prec\)
Using the above invariants, we show that the final clause set is saturated.

\textbf{proof (rule ccontr)}
\begin{align*}
&\text{assume } \neg \text{saturated-binary-rule resolvent } ?\text{Final} \\
&\text{then obtain } P1 P2 C \text{ where } P1 \in ?\text{Final and } P2 \in ?\text{Final and resolvent } P1 P2 C \\
&\quad \text{and } \neg \text{redundant } C ?\text{Final} \\
&\text{unfolding saturated-binary-rule-def by auto} \\
&\text{from resolvent } P1 P2 C \text{ obtain } B \text{ where } C = \text{resolvent-upon } P1 P2 B \\
&\text{unfolding resolvent-def by auto} \\
&\text{show False} \\
&\text{proof cases} \\
&\text{assume } B \in (\text{atoms-formula } ?\text{Final}) \\
&\quad \text{with } (\text{atoms-formula } ?\text{Final} \subseteq \{ X. \exists I:\text{nat. } I < N \land X = (A I) \}) \\
&\quad \text{obtain } I \text{ where } B = (A I) \text{ and } I < N \\
&\quad \text{by auto} \\
&\text{from } (B = (A I) \text{ and } (C = \text{resolvent-upon } P1 P2 B) \text{ have } C = \text{resolvent-upon } P1 P2 (A I)) \\
&\quad \text{by auto} \\
\end{align*}

next assume B /∈ atoms-formula ?Final with ⟨P1 ∈ ?Final⟩ have B /∈ atoms-clause P1 by auto then have Pos B /∈ P1 by auto with ⟨C = resolvent-upon P1 P2 B⟩ have P1 ⊆ C by auto with ⟨P1 ∈ ?Final⟩ and ⟨¬redundant C ?Final⟩ show False unfolding redundant-def subsumes-def by auto

ded

ded


with ⟨equivalent S ?Final⟩ show ?thesis using equivalence-and-prime-implicates by auto

qed

end