Propositional Resolution and Prime Implicates
Generation

Nicolas Peltier
Laboratory of Informatics of Grenoble/CNRS
University Grenoble Alps

February 23, 2021

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 con-
sidered: ordered resolution (with selection functions), positive and neg-
ative resolution, semantic resolution and unit resolution (the latter re-
finement 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 as-
sumed to be finite, so that the results can be applied to formulas ob-
tained by grounding sets of first-order clauses (however, a total order-
ing 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) implica-
tes 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, knowl-
edge compilation, diagnosis, debugging etc.). We also show that im-
plicates 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.

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 : (∀ x y. (x ≠ y → ((x,y) ∈ atom-ordering ∨ (y,x) ∈ atom-ordering)))
and \( \text{atom-ordering-trans} : \forall x y z. (x, y) \in \text{atom-ordering} \rightarrow (y, z) \in \text{atom-ordering} \rightarrow (x, z) \in \text{atom-ordering} \) 

and \( \text{atom-ordering-irrefl} : \forall x y. (x, y) \in \text{atom-ordering} \rightarrow (y, x) \notin \text{atom-ordering} \)

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).

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

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

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

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

⟨proof⟩

fun positive :: 'at Literal ⇒ bool where 
(\( \text{positive} (\text{Pos } A) ) = \text{True} \mid \text{positive} (\text{Neg } A) = \text{False} \)

fun negative :: 'at Literal ⇒ bool where 
(\( \text{negative} (\text{Pos } A) ) = \text{False} \mid \text{negative} (\text{Neg } A) = \text{True} \)

type-synonym 'a Clause = 'a Literal set 

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.

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

fun atoms-formula :: 'at Formula ⇒ 'at set where 
\( \text{atoms-formula } S = \{ A. \exists C. C \in S \land A \in \text{atoms-clause}(C) \} \)
lemma atoms-formula-subset: \( S_1 \subseteq S_2 \implies \text{atoms-formula } S_1 \subseteq \text{atoms-formula } S_2 \)

(proof)

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

(proof)

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 literal-ordering A B

assumes literal-ordering B C

shows literal-ordering A C

(proof)

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 :: \( 'a \text{Interpretation} \Rightarrow 'a \text{Literal} \Rightarrow \text{bool} \) (infix \( | = 65 \))

where \( \text{validate-literal } I (\text{Pos } A) = (A \in I) \mid \)

\( \text{validate-literal } I (\text{Neg } A) = (A \notin I) \)

fun validate-clause :: \( 'a \text{Interpretation} \Rightarrow 'a \text{Clause} \Rightarrow \text{bool} \) (infix \( | = 65 \))

where \( \text{validate-clause } I C = (\exists L. (L \in C) \land (\text{validate-literal } I L)) \)

fun validate-formula :: \( 'a \text{Interpretation} \Rightarrow 'a \text{Formula} \Rightarrow \text{bool} \) (infix \( | = 65 \))

where \( \text{validate-formula } I S = (\forall C. (C \in S \implies (\text{validate-clause } I C))) \)
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
(proof)

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
(proof)

lemma entailment-implies-validity:
  assumes entails-formula S1 S2
  assumes validate-formula I S1
  shows validate-formula I S2
(proof)

lemma validity-implies-entailment:
  assumes ∀I. validate-formula I S1 ⟹ validate-formula I S2
  shows entails-formula S1 S2
(proof)

lemma entails-transitive:
  assumes entails-formula S1 S2
  assumes entails-formula S2 S3
  shows entails-formula S1 S3
(proof)

lemma equivalent-transitive:
  assumes equivalent S1 S2
  assumes equivalent S2 S3
  shows equivalent S1 S3
(proof)

lemma entailment-subset :
  assumes S2 ⊆ S1
shows entails-formula \( S_1 S_2 \)
(proof)

lemma entailed-formula-entails-implicates:
  assumes entails-formula \( S_1 S_2 \)
  assumes entails \( S_2 C \)
  shows entails \( S_1 C \)
(proof)

3 Inference Rules

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

type-synonym 'a BinaryRule = 'a Clause ⇒ 'a Clause ⇒ bool

definition less-restrictive :: 'at BinaryRule ⇒ 'at BinaryRule ⇒ bool
  where
    (less-restrictive \( R_1 R_2 \)) = \( (\forall P_1 P_2 C. (R_2 P_1 P_2 C) \rightarrow ((R_1 P_1 P_2 C) \lor (R_1 P_2 P_1 C))) \)

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 \( R S \) = \{ \( C \). \( \exists P_1 P_2. P_1 \in S \land P_2 \in S \land (R P_1 P_2 C) \) \}

fun add-all-deducible-clauses :: 'at BinaryRule ⇒ 'at Formula ⇒ 'at Formula
  where
    add-all-deducible-clauses \( R S \) = \( (S \cup \text{all-deducible-clauses} \, R \, S) \)

definition derived-clauses-are-finite :: 'at BinaryRule ⇒ bool
  where
    derived-clauses-are-finite \( R \) = \( (\forall P_1 P_2 C. (\text{finite} \, P_1 \rightarrow \text{finite} \, P_2 \rightarrow (R P_1 P_2 C) \rightarrow \text{finite} \, C)) \)

lemma less-restrictive-and-finite :
  assumes less-restrictive \( R_1 R_2 \)
  assumes derived-clauses-are-finite \( R_1 \)
  shows derived-clauses-are-finite \( R_2 \)
(proof)

We then define the unrestricted resolution rule and usual resolution refinements.

3.1 Unrestricted Resolution

definition resolvent :: 'at BinaryRule
  where
    (resolvent \( P_1 P_2 C \) ≡
      \( (\exists A. ((\text{Pos} \, A) \in P_1 \land (\text{Neg} \, A) \in P_2 \land (C = ( (P_1 - \{ \text{Pos} \, A\}) \cup (P_2 - \{ \text{Neg} \, A\}))) ) \)))
For technical convenience, 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)
⟨proof⟩

lemma resolvent-is-resolvent-upon :
  assumes resolvent P1 P2 C
  shows ∃ A. C = resolvent-upon P1 P2 A
⟨proof⟩

lemma resolvent-is-finite :
  shows derived-clauses-are-finite resolvent
⟨proof⟩

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
(ordered-resolvent P1 P2 C) ≡
  (∃ A. ((C = ( (P1 - { Pos A}) ∪ ( P2 - { Neg A }))) ∧
  (strictly-maximal-literal P1 (Pos A)) ∧ (strictly-maximal-literal P2 (Neg A))))

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

lemma resolution-and-max-literal :
  assumes R = resolvent-upon P1 P2 A
  assumes strictly-maximal-literal P1 (Pos A)
  assumes strictly-maximal-literal P2 (Neg A)
  assumes strictly-maximal-literal R M
  shows (atom M, A) ∈ atom-ordering
⟨proof⟩
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** \((\text{selected-part } Sel \ C) = \{ L. L \in C \land (\exists A \in Sel. L = (Neg A)) \}\)

**definition** \(\text{ordered-set-resolvent} :: \text{’at set } \Rightarrow \text{’at Clause } \Rightarrow \text{’at Clause } \Rightarrow \text{’at Clause } \Rightarrow \text{bool}\)

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

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

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

**lemma** ordered-set-resolvent-is-ordered-resolvent :

\[\text{assumes } \text{ordered-resolvent } P1 P2 C\]
\[\text{assumes } \text{selected-part } Sel \ P1 = \{\}\]
\[\text{assumes } \text{selected-part } Sel \ P2 = \{\}\]
\[\text{shows } \text{ordered-set-resolvent } Sel \ P1 P2 C\]

**lemma** ordered-resolvent-upon-is-resolvent :

\[\text{assumes } \text{strictly-maximal-literal } P1 (Pos A)\]
\[\text{assumes } \text{strictly-maximal-literal } P2 (Neg A)\]
\[\text{shows } \text{ordered-resolvent } P1 P2 (\text{resolvent-upon } P1 P2 A)\]

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** \(\text{validated-part} :: \text{’at set } \Rightarrow \text{’at Clause } \Rightarrow \text{’at Clause}\)

\[
\text{where} \ (\text{validated-part } I \ C) = \{ L. L \in C \land (\text{validate-literal } I \ L) \}
\]

**definition** \(\text{ordered-model-resolvent} ::\)

\[\text{’at Interpretation } \Rightarrow \text{’at Clause } \Rightarrow \text{’at Clause } \Rightarrow \text{’at Clause } \Rightarrow \text{bool}\]

\[
\text{where}
\]
\[
\text{(ordered-model-resolvent } I \ P1 P2 C) = \\
(\exists L. (C = (P1 - \{ L \}) \cup (P2 - \{ \text{complement } L \}))) \land \\
((\text{validated-part } I \ P1) = \{\} \land (\text{strictly-maximal-literal } P1 L))
\]

8
lemma ordered-model-resolvent-is-resolvent : less-restrictive resolvent (ordered-model-resolvent I)
⟨proof⟩

3.5 Unit Resolution
Resolution is applied only if one parent is unit (this restriction is incomplete).
definition Unit :: 'at Clause ⇒ bool
where (Unit C) = ((card C) = 1)
definition unit-resolvent :: 'at BinaryRule
where (unit-resolvent P1 P2 C) = ((∃ L. (C = (P1 - {L}) ∪ (P2 - {complement L})))
∧ L ∈ P1 ∧ (complement L) ∈ P2) ∧ Unit P1)
lemma unit-resolvent-is-resolvent : less-restrictive resolvent unit-resolvent
⟨proof⟩

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. (∃ A. L = Pos A) ∧ L ∈ C }
definition negative-part :: 'at Clause ⇒ 'at Clause
where (negative-part C) = { L. (∃ A. L = Neg A) ∧ L ∈ C }
lemma decomposition-clause-pos-neg :
  C = (negative-part C) ∪ (positive-part C)
⟨proof⟩
definition ordered-positive-resolvent :: 'at Clause ⇒ 'at Clause ⇒ 'at Clause ⇒ bool
where (ordered-positive-resolvent P1 P2 C) =
  (∃ L. (C = (P1 - {L}) ∪ (P2 - {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 (ordered-negative-resolvent P1 P2 C) =
\((\exists L. (C = (P1 - \{ L \} \cup (P2 - \{ \text{complement} \ L \}))) \land
((\text{positive-part} \ P1) = \{ \} \land (\text{strictly-maximal-literal} \ P1 \ L))
\land (\text{strictly-maximal-literal} (\text{positive-part} \ P2) (\text{complement} \ L)))\)

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

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

## 4 Redundancy Elimination Rules

We define the usual redundancy elimination rules.

**Definition** tautology :: 'a Clause \(\Rightarrow\) bool

where

\((\text{tautology} \ C) \equiv (\exists \ A. (\text{Pos} \ A \in C \land \text{Neg} \ A \in C))\)

**Definition** subsumes :: 'a Clause \(\Rightarrow\) 'a Clause \(\Rightarrow\) bool

where

\((\text{subsumes} \ C \ D) \equiv (C \subseteq D)\)

**Definition** redundant :: 'a Clause \(\Rightarrow\) 'a Formula \(\Rightarrow\) bool

where

\((\text{redundant} \ C \ S) = ((\text{tautology} \ C) \lor (\exists \ D. (D \in S \land \text{subsumes} \ D \ C)))\)

**Definition** strictly-redundant :: 'a Clause \(\Rightarrow\) 'a Formula \(\Rightarrow\) bool

where

\((\text{strictly-redundant} \ C \ S) = ((\text{tautology} \ C) \lor (\exists \ D. (D \in S \land (D \subset C))))\)

**Definition** simplify :: 'at Formula \(\Rightarrow\) 'at Formula

where

\((\text{simplify} \ S) = \{ C. C \in S \land \neg \text{strictly-redundant} \ C \ S \} \)

We first establish some basic syntactic properties.

**Lemma** tautology-monotonous : (tautology \ C) \(\Rightarrow\) (C \subseteq D) \(\Rightarrow\) (tautology \ D)

**Lemma** simplify-involutive:

\text{shows} \ simplify (simplify \ S) = (simplify \ S)

**Lemma** simplify-finite:

\text{assumes} \ all-fulfill \ finite \ S

\text{shows} \ all-fulfill \ finite (simplify \ S)

**Lemma** atoms-formula-simplify:

\text{shows} \ atoms-formula (simplify \ S) \subseteq atoms-formula \ S
lemma subsumption-preserves-redundancy:
assumes redundant \( C \subseteq S \)
assumes subsumes \( C \subseteq D \)
shows redundant \( D \subseteq S \)

lemma subsumption-and-max-literal:
assumes subsumes \( C_1 \subseteq C_2 \)
assumes strictly-maximal-literal \( C_1 \subseteq L_1 \)
assumes strictly-maximal-literal \( C_2 \subseteq L_2 \)
assumes \( A_1 = \text{atom} \ L_1 \)
assumes \( A_2 = \text{atom} \ L_2 \)
shows \((A_1 = A_2) \lor (A_1, A_2) \in \text{atom-ordering}\)

lemma superset-preserves-redundancy:
assumes redundant \( C \subseteq S \)
assumes \( S \subseteq S' \)
shows redundant \( C \subseteq S' \)

lemma superset-preserves-strict-redundancy:
assumes strictly-redundant \( C \subseteq S \)
assumes \( S \subseteq S' \)
shows strictly-redundant \( C \subseteq S' \)

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 \( I \subseteq C \)

lemma subsumption-and-semantics:
assumes subsumes \( C \subseteq D \)
assumes validate-clause \( I \subseteq C \)
shows validate-clause \( I \subseteq D \)

lemma redundancy-and-semantics:
assumes redundant \( C \subseteq S \)
assumes validate-formula \( I \subseteq S \)
shows validate-clause \( I \subseteq C \)

lemma redundancy-implies-entailment:
assumes redundant $C \ S$
shows entails $S \ C$
⟨proof⟩

lemma simplify-and-membership:
  assumes all-fulfill finite $S$
  assumes $T = \text{simplify} \ S$
  assumes $C \in S$
  shows redundant $C \ T$
⟨proof⟩

lemma simplify-preserves-redundancy:
  assumes all-fulfill finite $S$
  assumes redundant $C \ S$
  shows redundant $C (\text{simplify} \ S)$
⟨proof⟩

lemma simplify-preserves-strict-redundancy:
  assumes all-fulfill finite $S$
  assumes strictly-redundant $C \ S$
  shows strictly-redundant $C (\text{simplify} \ S)$
⟨proof⟩

lemma simplify-preserves-semantic:
  assumes $T = \text{simplify} \ S$
  assumes all-fulfill finite $S$
  shows validate-formula $I \ S \iff \text{validate-formula} \ I \ T$
⟨proof⟩

lemma simplify-preserves-equivalence:
  assumes $T = \text{simplify} \ S$
  assumes all-fulfill finite $S$
  shows equivalent $S \ T$
⟨proof⟩

After simplification, the formula contains no strictly redundant clause:

definition non-redundant :: ‘a at Formula ⇒ bool
  where non-redundant $S = (\forall C. (C \in S \rightarrow \neg\text{strictly-redundant} C \ S))$

lemma simplify-non-redundant:
  shows non-redundant (simplify $S$)
⟨proof⟩

lemma deducible-clause-preserve-redundancy:
  assumes redundant $C \ S$
  shows redundant $C (\text{add-all-deducible-clauses} \ R \ S)$
⟨proof⟩
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 ⟩

**lemma** inverse-clause-renaming : (rename-clause A (rename-clause A L)) = L
⟨ proof ⟩

**lemma** inverse-formula-renaming : rename-formula A (rename-formula A L) = L
⟨ proof ⟩

**lemma** renaming-preserves-cardinality :
   card (rename-clause A C) = card C
⟨ proof ⟩

**lemma** renaming-preserves-literal-order :
   assumes literal-ordering L1 L2
   shows literal-ordering (rename-literal A L1) (rename-literal A L2)
⟨ proof ⟩

**lemma** inverse-renaming-preserves-literal-order :
   assumes literal-ordering (rename-literal A L1) (rename-literal A L2)
   shows literal-ordering L1 L2
⟨ proof ⟩

**lemma** renaming-is-injective:
   assumes rename-literal A L1 = rename-literal A L2
   shows L1 = L2
⟨ proof ⟩

**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 ⟩

**lemma** renaming-and-selected-part :
   selected-part UNIV C = rename-clause Sel (validated-part Sel (rename-clause Sel C))


\begin{proof}

\textbf{lemma} renaming-preserves-tautology:
\begin{itemize}
  \item \textbf{assumes} tautology \( C \)
  \item \textbf{shows} tautology (rename-clause Sel \( C \))
\end{itemize}
\end{proof}

\begin{proof}

\textbf{lemma} rename-union : rename-clause Sel \(( C \cup D ) = rename-clause Sel C \cup rename-clause Sel D \)
\end{proof}

\begin{proof}

\textbf{lemma} renaming-set-minus-subset :
\begin{itemize}
  \item rename-clause Sel \(( C \setminus \{ L \} ) \subseteq rename-clause Sel \( C \setminus \{ rename-literal Sel L \} \)
\end{itemize}
\end{proof}

\begin{proof}

\textbf{definition} rename-interpretation :: 'at set => 'at Interpretation => 'at Interpretation
\begin{itemize}
  \item rename-interpretation Sel I = \{ A. (A \in I \land A \notin Sel) \} \cup \{ A. (A \notin I \land A \in Sel) \}
\end{itemize}
\end{proof}

\begin{proof}

\textbf{lemma} renaming-preserves-semantic :
\begin{itemize}
  \item \textbf{assumes} validate-literal \( I \) \( L \)
  \item \textbf{shows} validate-literal (rename-interpretation Sel \( I \)) (rename-literal Sel \( L \))
\end{itemize}
\end{proof}

\begin{proof}

\textbf{lemma} renaming-preserves-satisfiability:
\begin{itemize}
  \item \textbf{assumes} satisfiable \( S \)
  \item \textbf{shows} satisfiable (rename-formula Sel \( S \))
\end{itemize}
\end{proof}

\begin{proof}

\textbf{lemma} renaming-preserves-subsumption:
\begin{itemize}
  \item \textbf{assumes} subsumes \( C \) \( D \)
  \item \textbf{shows} subsumes (rename-clause Sel \( C \)) (rename-clause Sel \( D \))
\end{itemize}
\end{proof}

\section{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.

\textbf{definition} Sound :: 'at BinaryRule => bool
\begin{itemize}
  \item \textbf{where}
  \begin{itemize}
    \item (Sound Rule) \equiv \forall I \ P1 \ P2 \ C. (Rule \ P1 \ P2 \ C \rightarrow (validate-clause \ I \ P1) \rightarrow (validate-clause \ I \ P2))
  \end{itemize}
\end{itemize}
lemma soundness-and-entailment :
assumes Sound Rule
assumes Rule P1 P2 C
assumes P1 ∈ S
assumes P2 ∈ S
shows entails S C
⟨proof⟩

lemma all-deducible-sound:
assumes Sound R
shows entails-formula S (all-deducible-clauses R S)
⟨proof⟩

lemma add-all-deducible-sound:
assumes Sound R
shows entails-formula S (add-all-deducible-clauses R S)
⟨proof⟩

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
⟨proof⟩

We finally establish usual concrete soundness results.

theorem resolution-is-correct:
(Sound resolvent)
⟨proof⟩

theorem ordered-resolution-correct : Sound ordered-resolvent
⟨proof⟩

theorem ordered-model-resolution-correct : Sound (ordered-model-resolvent I)
⟨proof⟩

theorem ordered-positive-resolution-correct : Sound ordered-positive-resolvent
⟨proof⟩

theorem ordered-negative-resolution-correct : Sound ordered-negative-resolvent
⟨proof⟩

theorem unit-resolvent-correct : Sound unit-resolvent
⟨proof⟩
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) → 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
⟨proof⟩

**lemma** less-restrictive-complete:
**assumes** less-restrictive R1 R2
**assumes** Complete R2
**shows** Complete R1
⟨proof⟩

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** canonic-int-fun-ordered : 'at ⇒ ('at Formula ⇒ bool)
**where**
(canonic-int-fun-ordered A) =
(∀ S. (∃ C. (C ∈ S) ∧ (strictly-maximal-literal C (Pos A) ))
∧ (∀ B. (Pos B ∈ C → (B, A) ∈ atom-ordering → (¬(canonic-int-fun-ordered B) S)))
∧ (∀ B. (Neg B ∈ C → (B, A) ∈ atom-ordering → ((canonic-int-fun-ordered B) S))))
⟨proof⟩

definition canonic-int-ordered :: 'at Formula ⇒ 'at Interpretation
where
(canonic-int-ordered S) = { A. ((canonic-int-fun-ordered A) S) }

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

lemma int-validate-cl-with-pos-max :
  assumes strictly-maximal-literal C (Pos A)
  assumes C ∈ S
  shows validate-clause (canonic-int-ordered S) C
⟨proof⟩

lemma strictly-maximal-literal-exists :

∀ C. (((finite C) ∧ (card C) = n ∧ n ≠ 0 ∧ (¬ (tautology C)))
→ (∃ A. (strictly-maximal-literal C A)) (is ?P n)
⟨proof⟩

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 {} ∉ S
  assumes C ∈ S
  shows validate-clause (canonic-int-ordered S) C
⟨proof⟩

theorem ordered-resolution-is-complete :
  Complete ordered-resolvent
⟨proof⟩

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-set-resolvent Sel) S
  shows saturated-binary-rule ordered-resolvent (empty-selected-part Sel S)
⟨proof⟩

definition ordered-set-resolvent-upon :: 'at set ⇒ 'at Clause ⇒ 'at Clause ⇒ 'at Clause ⇒ bool
  where
  ordered-set-resolvent-upon Sel P1 P2 C A ≡ 
  ( ((C = ( P1 − { Pos A }) ∪ ( P2 − { Neg A })) )
  ∧ (strictly-maximal-literal P1 (Pos A)) ∧ ((selected-part Sel P1) = {}))
  ∧ ( ((strictly-maximal-literal P2 (Neg A)) ∧ (selected-part Sel P2) = {}))
  ∨ (strictly-maximal-literal (selected-part Sel P2) (Neg A))))

lemma ordered-set-resolvent-upon-is-resolvent:
  assumes ordered-set-resolvent-upon Sel P1 P2 C A
  shows ordered-set-resolvent Sel P1 P2 C
⟨proof⟩

lemma resolution-decreases-selected-part:
  assumes ordered-set-resolvent-upon Sel P1 P2 C A
  assumes Neg A ∈ P2
  assumes finite P1
  assumes finite P2
  assumes card (selected-part Sel P2) = Suc n
  shows card (selected-part Sel C) = n
⟨proof⟩

lemma canonic-int-validates-all-clauses-sel:
  assumes saturated-binary-rule (ordered-set-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⟩

theorem ordered-resolution-is-complete-ordered-sel:
  Complete (ordered-set-resolvent Sel)
⟨proof⟩

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.

lemma ordered-res-with-selection-is-model-res:
  assumes ordered-set-resolvent UNIV P1 P2 C

shows ordered-model-resolvent Sel (rename-clause Sel P1) (rename-clause Sel P2) (rename-clause Sel C) (proof)

theorem ordered-resolution-is-complete-model-resolution:
  Complete (ordered-model-resolvent Sel) (proof)

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 :
  validate-literal { } L = (∃ A. (L = Neg A)) (proof)

lemma universal-interpretation-validate :
  validate-literal UNIV L = (∃ A. (L = Pos A)) (proof)

lemma negative-part-lemma:
  (negative-part C) = (validated-part { } C) (proof)

lemma positive-part-lemma:
  (positive-part C) = (validated-part UNIV C) (proof)

lemma negative-resolvent-is-model-res:
  less-restrictive ordered-negative-resolvent (ordered-model-resolvent UNIV) (proof)

lemma positive-resolvent-is-model-res:
  less-restrictive ordered-positive-resolvent (ordered-model-resolvent { }) (proof)

theorem ordered-positive-resolvent-is-complete : Complete ordered-positive-resolvent (proof)

theorem ordered-negative-resolvent-is-complete: Complete ordered-negative-resolvent (proof)

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⟩

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 M)) }

We prove that all inference rules yield finite clauses.

theorem ordered-resolvent-is-finite : derived-clauses-are-finite ordered-resolvent
⟨proof⟩

theorem model-resolvent-is-finite : derived-clauses-are-finite (ordered-model-resolvent I)
⟨proof⟩

theorem positive-resolvent-is-finite : derived-clauses-are-finite ordered-positive-resolvent
theorem negative-resolvent-is-finite : derived-clauses-are-finite ordered-negative-resolvent
⟨proof⟩

theorem unit-resolvent-is-finite : derived-clauses-are-finite unit-resolvent
⟨proof⟩

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⟩

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⟩

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⟩

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⟩

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
  assumes derived-clauses-are-finite R
  shows ∃ M. strictly-redundant D (inferred-clause-sets R S M)
⟨proof⟩

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⟩

We deduce that the set of persistent clauses is saturated.

theorem persistent-clauses-are-saturated:
assumes derived-clauses-are-finite R
assumes all-fulfill finite S
shows saturated-binary-rule R (saturation R S)
⟨proof⟩

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⟩

end
end

9 Prime Implicates Generation

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

9.1 Implicates and Prime Implicates

We first introduce the definitions of implicates and prime implicates.

definition implicates :: 'at Formula ⇒ 'at Formula
where implicates S = { C. entails S C }
**Definition** prime-implicates :: 'at Formula ⇒ 'at 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

\[(\text{partial-evaluation } S C) = \{ \exists E. \exists D. D \in S \land E = D - C \land \lnot (\exists L. (L \in C) \land (\text{complement } L) \in D) \}\]

**lemma** partial-evaluation-is-saturated :

- **assumes** saturated-binary-rule resolvent S
- **shows** saturated-binary-rule ordered-resolvent (partial-evaluation S C)

**proof**

**lemma** evaluation-wrt-implicate-is-unsat :

- **assumes** entails S C
- **assumes** ¬tautology C
- **shows** ¬satisfiable (partial-evaluation S C)

**proof**

**lemma** entailment-and-implicates:

- **assumes** entails-formula S1 S2
- **shows** implicates S2 ⊆ implicates S1

**proof**

**lemma** equivalence-and-implicates:

- **assumes** equivalent S1 S2
- **shows** implicates S1 = implicates S2

**proof**

**lemma** equivalence-and-prime-implicates:

- **assumes** equivalent S1 S2
- **shows** prime-implicates S1 = prime-implicates S2

**proof**

**lemma** unrestricted-resolution-is-deductive-complete :

- **assumes** saturated-binary-rule resolvent S
- **assumes** all-fulfill finite S
assumes $C \in \text{implicates } S$
shows redundant $C S$

lemma prime-implicates-generation-correct :
assumes saturated-binary-rule resolvent $S$
assumes non-redundant $S$
assumes all-fulfill finite $S$
shows $S \subseteq \text{prime-implicates } S$

(\text{proof})

theorem prime-implicates-of-saturated-sets:
assumes saturated-binary-rule resolvent $S$
assumes all-fulfill finite $S$
assumes non-redundant $S$
shows $S = \text{prime-implicates } S$

(\text{proof})

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.

\text{definition all-resolvents-upon :: 'at Formula } \Rightarrow 'at \Rightarrow '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 $S C$

(\text{proof})

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

(\text{proof})

lemma atoms-formula-resolvents:
shows atoms-formula $(\text{all-resolvents-upon } S A) \subseteq \text{atoms-formula } S$

(\text{proof})
We define a partial saturation predicate that is restricted to a specific atom.

**definition** partial-saturation :: 'at Formula ⇒ 'at ⇒ 'at Formula ⇒ bool
where
  (partial-saturation S A R) = (∀ P1 P2. (P1 ∈ S → P2 ∈ S → (redundant (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⟩

We show that if \( R \) is a set of resolvents of a set of clauses \( S \) then the same holds for \( S \cup R \). 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 (∀x. ¬(tautology x)) S
  assumes P1 ∈ S ∪ R
  shows in-all-resolvents-upon S A P1
⟨proof⟩

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 partial-saturation S E1 S
  assumes partial-saturation S E2 (S ∪ R)
  assumes all-fulfill (∀x. ¬(tautology x)) S
  assumes all-fulfill (in-all-resolvents-upon S E2) R
  shows partial-saturation (S ∪ R) E1 (S ∪ R)
⟨proof⟩

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 ¬ tautology P1

25
assumes \( \lnot \text{tautology} P2 \)
assumes \( C = \text{resolvent-upon} P1 P2 E2 \)
assumes \( \lnot \text{subsumes} P1 C \)
assumes \( \lnot \text{subsumes} P2 C \)
shows \((\text{Neg} \ E2) \notin C \land (\text{Pos} \ E2) \notin C\)
 ⟨proof⟩

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

**Lemma ensures-partial-saturation**:  
asumes partial-saturation \( S \) \( E2 \) \((S \cup R)\)
asumes all-fulfill \((\lambda x. \lnot(\text{tautology} \ x)) \) \( S \)
asumes all-fulfill \((\text{in-all-resolvents-upon} \ S \ E2) \) \( R \)
asumes all-fulfill \((\lambda x. \lnot\text{redundant} \ x \ S)) \) \( R \)
shows partial-saturation \((S \cup R) \) \( E2 \) \((S \cup R)\)
⟨proof⟩

**Lemma resolvents-preserve-equivalence**:  
shows equivalent \( S \) \((S \cup (\text{all-resolvents-upon} \ S \ A))\)
⟨proof⟩

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** :: \((\text{nat} \Rightarrow \text{at}) \Rightarrow \text{at} \) \( \text{Formula} \Rightarrow \) \( \text{nat} \Rightarrow \text{at} \) \( \text{Formula} \)
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**:  
asumes partial-saturation \( S1 \) \( A \ S1 \)
asumes \( S2 \subseteq S1 \)
asumes all-fulfill \((\lambda x. \text{redundant} \ x \ S2) \) \( S1 \)
shows partial-saturation \( S2 \) \( A \ S2 \)
⟨proof⟩

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**:  
asumes \( \text{atoms-formula} \ S = \{ \ X. \exists I::\text{nat}. \ I < N \land X = (A \ I) \} \)
asumes all-fulfill finite \( S \)
shows \((\text{prime-implicates} \ S) = (\text{resolvents-sequence} \ A \ S \ N)\)
⟨proof⟩