Propositional Resolution and Prime Implicates Generation

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

October 13, 2025

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

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 *atom-ordering-trans*: $\forall x\ y\ z.\ (x,y) \in$ *atom-ordering* $\longrightarrow (y,z) \in$ *atom-ordering* $\longrightarrow (x,z) \in$ *atom-ordering*
and *atom-ordering-irrefl*: $\forall x\ y.\ (x,y) \in$ *atom-ordering* $\longrightarrow (y,x) \notin$ *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).

datatype *'a Literal* = *Pos 'a* | *Neg 'a*

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

fun *atom* :: *'a Literal* ⇒ *'a*
where
(*atom* (*Pos A*)) = *A* |
(*atom* (*Neg A*)) = *A*

fun *complement* :: *'a Literal* ⇒ *'a Literal*
where
(*complement* (*Pos A*)) = (*Neg A*) |
(*complement* (*Neg A*)) = (*Pos A*)

lemma *atom-property* : *A* = (*atom L*) ⟹ (*L* = (*Pos A*) ∨ *L* = (*Neg A*))
by (*metis atom.elims*)

fun *positive* :: *'at Literal* ⇒ *bool*
where
(*positive* (*Pos A*)) = *True* |
(*positive* (*Neg A*)) = *False*

fun *negative* :: *'at Literal* ⇒ *bool*
where
(*negative* (*Pos A*)) = *False* |
(*negative* (*Neg A*)) = *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 *atoms-clause C* = { *A*. ∃ *L*. *L* ∈ *C* ∧ *A* = *atom*(*L*) }

fun *atoms-formula* :: *'at Formula* ⇒ *'at set*
where *atoms-formula S* = { *A*. ∃ *C*. *C* ∈ *S* ∧ *A* ∈ *atoms-clause*(*C*) }

lemma *atoms-formula-subset*: $S1 \subseteq S2 \Longrightarrow$ *atoms-formula* $S1 \subseteq$ *atoms-formula* $S2$
by *auto*

lemma *atoms-formula-union*: *atoms-formula* $(S1 \cup S2) =$ *atoms-formula* $S1 \cup$ *atoms-formula* $S2$
by *auto*

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

definition *all-fulfill* :: $('at\ Clause \Rightarrow bool) \Rightarrow 'at\ Formula \Rightarrow bool$
where *all-fulfill* $P\ S = (\forall\, C.\ (C \in S \longrightarrow (P\ C)))$

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

fun *literal-ordering* :: $'at\ Literal \Rightarrow 'at\ Literal \Rightarrow bool$
where
$(literal\text{-}ordering\ L1\ L2) = ((atom\ L1, atom\ L2) \in atom\text{-}ordering)$

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

definition *strictly-maximal-literal* :: $'at\ Clause \Rightarrow 'at\ Literal \Rightarrow bool$
where
$(strictly\text{-}maximal\text{-}literal\ S\ A) \equiv (A \in S) \wedge (\forall B.\ (\ B \in S \wedge A \neq B) \longrightarrow (literal\text{-}ordering\ B\ A))$

2 Semantics

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

type-synonym $'a\ Interpretation = 'a\ set$

fun *validate-literal* :: $'at\ Interpretation \Rightarrow 'at\ Literal \Rightarrow bool$ (**infix** ‹⊨› *65*)
where
$(validate\text{-}literal\ I\ (Pos\ A)) = (A \in I)$ |
$(validate\text{-}literal\ I\ (Neg\ A)) = (A \notin I)$

fun *validate-clause* :: $'at\ Interpretation \Rightarrow 'at\ Clause \Rightarrow bool$ (**infix** ‹⊨› *65*)
where
$(validate\text{-}clause\ I\ C) = (\exists\, L.\ (L \in\ C) \wedge (validate\text{-}literal\ I\ L))$

fun *validate-formula* :: $'at\ Interpretation \Rightarrow 'at\ Formula \Rightarrow bool$ (**infix** ‹⊨› *65*)
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 *S2* ⊆ *S1*
shows *entails-formula S1 S2*
proof −
have ∀ *L La*. *L* ∉ *La* ∨ *entails La L*
by (*meson entails-member*)
thus *?thesis*
by (*meson assms entails-formula-def rev-subsetD*)
qed

lemma *entailed-formula-entails-implicates*:
assumes *entails-formula S1 S2*
assumes *entails S2 C*
shows *entails S1 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*) = (∀ *P1 P2 C*. (*R2 P1 P2 C*) ⟶ ((*R1 P1 P2 C*) ∨ (*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*. ∃ *P1 P2*. *P1* ∈ *S* ∧ *P2* ∈ *S* ∧ (*R P1 P2 C*) }

fun *add-all-deducible-clauses*:: *'at BinaryRule* ⇒ *'at Formula* ⇒ *'at Formula*
where *add-all-deducible-clauses R S* = (*S* ∪ *all-deducible-clauses R S*)

definition *derived-clauses-are-finite* :: *'at BinaryRule* ⇒ *bool*
where *derived-clauses-are-finite R* =
(∀ *P1 P2 C*. (*finite P1* ⟶ *finite P2* ⟶ (*R P1 P2 C*) ⟶ *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 refinements.

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
  (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 −
  obtain MA where M = (Pos MA) ∨ M = (Neg MA) using Literal.exhaust [of
M] by auto
  hence MA = (atom M) by auto
  from ‹strictly-maximal-literal R M› and ‹R = resolvent-upon P1 P2 A›
    have M ∈ P1 − { Pos A } ∨ M ∈ P2 − { Neg A }
    unfolding strictly-maximal-literal-def by auto
  hence (MA,A) ∈ atom-ordering
  proof
    assume M ∈ P1 − { Pos A }
    from ‹M ∈ P1 − { Pos A }› and ‹strictly-maximal-literal P1 (Pos A)›
      have literal-ordering M (Pos A)
      unfolding strictly-maximal-literal-def by auto
    from ‹M = Pos MA ∨ M = Neg MA› and ‹literal-ordering M (Pos A)›
    show (MA,A) ∈ atom-ordering by auto
  next
    assume M ∈ P2 − { Neg A }
    from ‹M ∈ P2 − { Neg A }› and ‹strictly-maximal-literal P2 (Neg A)›
    have literal-ordering M (Neg A) by (auto simp only: strictly-maximal-literal-def)
    from ‹M = Pos MA ∨ M = Neg MA› and ‹literal-ordering M (Neg A)›
    show (MA,A) ∈ atom-ordering by auto
  qed
  from this and ‹MA = atom M› show ?thesis by auto
qed
```

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 (selected-part Sel C) = { L. L ∈ C ∧ (∃ A ∈ Sel. L = (Neg A)) }

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

where
(*ordered-sel-resolvent 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-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* (*Pos A*)
assumes *strictly-maximal-literal P2* (*Neg A*)
shows *ordered-resolvent P1 P2* (*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* ∧ (*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*) =
(∃ *L*. (*C* = (*P1* − { *L* } ∪ (*P2* − { *complement L* }))) ∧
((*validated-part I P1*) = {} ∧ (*strictly-maximal-literal P1 L*))
∧ (*strictly-maximal-literal* (*validated-part I 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*

and ¬*resolvent P2 P1 C* **unfolding** *less-restrictive-def* **by** *auto*
from ‹*ordered-model-resolvent I P1 P2 C*› **obtain** *L*
where *strictly-maximal-literal P1 L*
and *strictly-maximal-literal* (*validated-part I P2*) (*complement L*)
and *C* = (*P1* − { *L* }) ∪ (*P2* − { *complement L* })
using *ordered-model-resolvent-def* [*of I P1 P2 C*] **by** *auto*
from ‹*strictly-maximal-literal P1 L*› **have** *L* ∈ *P1* **by** (*simp only*: *strictly-maximal-literal-def*)
from ‹*strictly-maximal-literal* (*validated-part I P2*) (*complement L*)› **have** (*complement L*) ∈ *P2*
by (*auto simp only*: *strictly-maximal-literal-def validated-part-def*)
obtain *A* **where** *L* = *Pos A* ∨ *L* = *Neg A* **using** *Literal.exhaust* [*of L*] **by** *auto*
from *this* **and** ‹*C* = (*P1* − { *L* }) ∪ (*P2* − { *complement L* })› **and** ‹*L* ∈ *P1*› **and** ‹(*complement L*) ∈ *P2*›
have *resolvent P1 P2 C* ∨ *resolvent P2 P1 C* **unfolding** *resolvent-def* **by** *auto*
from *this* **and** ‹¬*resolvent P2 P1 C*› **and** ‹¬*resolvent P1 P2 C*› **show** *False* **by** *auto*
qed

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 (*rule ccontr*)
assume ¬ *less-restrictive resolvent unit-resolvent*
then obtain *P1 P2 C* **where** *unit-resolvent P1 P2 C* **and** ¬*resolvent P1 P2 C*
and ¬*resolvent P2 P1 C* **unfolding** *less-restrictive-def* **by** *auto*
from ‹*unit-resolvent P1 P2 C*› **obtain** *L* **where** *L* ∈ *P1* **and** *complement L* ∈ *P2*
and *C* = (*P1* − { *L* }) ∪ (*P2* − { *complement L* })
using *unit-resolvent-def* [*of P1 P2 C*] **by** *auto*
obtain *A* **where** *L* = *Pos A* ∨ *L* = *Neg A* **using** *Literal.exhaust* [*of L*] **by** *auto*
from *this* **and** ‹*C* = (*P1* − { *L* }) ∪ (*P2* − { *complement L* })› **and** ‹*L* ∈ *P1*› **and** ‹*complement L* ∈ *P2*›
have *resolvent P1 P2 C* ∨ *resolvent P2 P1 C* **unfolding** *resolvent-def* **by** *auto*
from *this* **and** ‹¬*resolvent P2 P1 C*› **and** ‹¬*resolvent P1 P2 C*› **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
  $(positive\text{-}part\ C) = \{\ L.\ (\exists A.\ L = Pos\ A) \wedge L \in C\ \}$

definition *negative-part* :: *'at Clause* ⇒ *'at Clause*
where
  $(negative\text{-}part\ C) = \{\ L.\ (\exists A.\ L = Neg\ A) \wedge L \in C\ \}$

lemma *decomposition-clause-pos-neg* :
  $C = (negative\text{-}part\ C) \cup (positive\text{-}part\ C)$
proof
  show $C \subseteq (negative\text{-}part\ C) \cup (positive\text{-}part\ C)$
  proof
    fix x **assume** $x \in C$
    obtain A **where** $x = Pos\ A \vee x = Neg\ A$ **using** *Literal.exhaust* [*of x*] **by** *auto*
    show $x \in (negative\text{-}part\ C) \cup (positive\text{-}part\ C)$
    proof *cases*
      assume $x = Pos\ A$
      from *this* **and** ‹$x \in C$› **have** $x \in positive\text{-}part\ C$ **unfolding** *positive-part-def* **by** *auto*
      then show $x \in (negative\text{-}part\ C) \cup (positive\text{-}part\ C)$ **by** *auto*
    next
      assume $x \neq Pos\ A$
      from *this* **and** ‹$x = Pos\ A \vee x = Neg\ A$›**have** $x = Neg\ A$ **by** *auto*
      from *this* **and** ‹$x \in C$› **have** $x \in negative\text{-}part\ C$ **unfolding** *negative-part-def* **by** *auto*
      then show $x \in (negative\text{-}part\ C) \cup (positive\text{-}part\ C)$ **by** *auto*
    qed
  qed
next
  show $(negative\text{-}part\ C) \cup (positive\text{-}part\ C) \subseteq C$ **unfolding** *negative-part-def*
  and *positive-part-def* **by** *auto*
qed

definition *ordered-positive-resolvent* :: *'at Clause* ⇒ *'at Clause* ⇒ *'at Clause* ⇒ *bool*
where
  $(ordered\text{-}positive\text{-}resolvent\ P1\ P2\ C) =$
    $(\exists L.\ (C = (P1 - \{\ L\ \} \cup (P2 - \{\ complement\ L\ \}))) \wedge$
      $((negative\text{-}part\ P1) = \{\}\ \wedge (strictly\text{-}maximal\text{-}literal\ P1\ L))$
      $\wedge (strictly\text{-}maximal\text{-}literal\ (negative\text{-}part\ P2)\ (complement\ L)))$

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

(*ordered-negative-resolvent P1 P2 C*) =
(∃ *L*. (*C* = (*P1* − { *L* } ∪ (*P2* − { *complement L* }))) ∧
((*positive-part P1*) = {} ∧ (*strictly-maximal-literal P1 L*))
∧ (*strictly-maximal-literal* (*positive-part P2*) (*complement L*)))

lemma *positive-resolvent-is-resolvent* : *less-restrictive resolvent ordered-positive-resolvent*
proof (*rule ccontr*)
assume ¬ *less-restrictive resolvent ordered-positive-resolvent*
then obtain *P1 P2 C* **where** *ordered-positive-resolvent P1 P2 C* **and** ¬*resolvent P1 P2 C*
and ¬*resolvent P2 P1 C* **unfolding** *less-restrictive-def* **by** *auto*
from ‹*ordered-positive-resolvent P1 P2 C*› **obtain** *L*
where *strictly-maximal-literal P1 L*
and *strictly-maximal-literal* (*negative-part P2*)(*complement L*)
and *C* = (*P1* − { *L* }) ∪ (*P2* − { *complement L* })
using *ordered-positive-resolvent-def* [*of P1 P2 C*] **by** *auto*
from ‹*strictly-maximal-literal P1 L*› **have** *L* ∈ *P1* **unfolding** *strictly-maximal-literal-def* **by** *auto*
from ‹*strictly-maximal-literal* (*negative-part P2*) (*complement L*)› **have** (*complement L*) ∈ *P2*
unfolding *strictly-maximal-literal-def negative-part-def* **by** *auto*
obtain *A* **where** *L* = *Pos A* ∨ *L* = *Neg A* **using** *Literal.exhaust* [*of L*] **by** *auto*
from *this* **and** ‹*C* = (*P1* − { *L* }) ∪ (*P2* − { *complement L* })› **and** ‹*L* ∈ *P1*› **and** ‹(*complement L*) ∈ *P2*›
have *resolvent P1 P2 C* ∨ *resolvent P2 P1 C* **unfolding** *resolvent-def* **by** *auto*
from *this* **and** ‹¬(*resolvent P2 P1 C*)› **and** ‹¬(*resolvent P1 P2 C*)› **show** *False* **by** *auto*
qed

lemma *negative-resolvent-is-resolvent* : *less-restrictive resolvent ordered-negative-resolvent*
proof (*rule ccontr*)
assume ¬ *less-restrictive resolvent ordered-negative-resolvent*
then obtain *P1 P2 C* **where** (*ordered-negative-resolvent P1 P2 C*) **and** ¬(*resolvent P1 P2 C*)
and ¬(*resolvent P2 P1 C*) **unfolding** *less-restrictive-def* **by** *auto*
from ‹*ordered-negative-resolvent P1 P2 C*› **obtain** *L* **where** *strictly-maximal-literal P1 L*
and *strictly-maximal-literal* (*positive-part P2*)(*complement L*)
and *C* = (*P1* − { *L* }) ∪ (*P2* − { *complement L* })
using *ordered-negative-resolvent-def* [*of P1 P2 C*] **by** *auto*
from ‹*strictly-maximal-literal P1 L*› **have** *L* ∈ *P1* **unfolding** *strictly-maximal-literal-def* **by** *auto*
from ‹*strictly-maximal-literal* (*positive-part P2*) (*complement L*)› **have** (*complement L*) ∈ *P2*
unfolding *strictly-maximal-literal-def positive-part-def* **by** *auto*
obtain *A* **where** *L* = *Pos A* ∨ *L* = *Neg A* **using** *Literal.exhaust* [*of L*] **by** *auto*
from *this* **and** ‹*C* = (*P1* − { *L* }) ∪ (*P2* − { *complement L* })› **and** ‹*L* ∈ *P1*› **and** ‹(*complement L*) ∈ *P2*›
have *resolvent P1 P2 C* ∨ *resolvent P2 P1 C* **unfolding** *resolvent-def* **by** *auto*

from *this* **and** ‹¬*resolvent P2 P1 C*› **and** ‹¬*resolvent P1 P2 C*› **show** *False* **by** *auto*
qed

4 Redundancy Elimination Rules

We define the usual redundancy elimination rules.

definition *tautology* :: $'a$ *Clause* $\Rightarrow$ *bool*
where
$(tautology\ C) \equiv (\exists\ A.\ (Pos\ A \in C \land Neg\ A \in C))$

definition *subsumes* :: $'a$ *Clause* $\Rightarrow$ $'a$ *Clause* $\Rightarrow$ *bool*
where
$(subsumes\ C\ D)\ \equiv (C \subseteq D)$

definition *redundant* :: $'a$ *Clause* $\Rightarrow$ $'a$ *Formula* $\Rightarrow$ *bool*
where
$redundant\ C\ S = ((tautology\ C) \lor (\exists D.\ (D \in S \land subsumes\ D\ C)))$

definition *strictly-redundant* :: $'a$ *Clause* $\Rightarrow$ $'a$ *Formula* $\Rightarrow$ *bool*
where
$strictly\text{-}redundant\ C\ S = ((tautology\ C) \lor (\exists D.\ (D \in S \land (D \subset C))))$

definition *simplify* :: $'at$ *Formula* $\Rightarrow$ $'at$ *Formula*
where
$simplify\ S = \{\ C.\ C \in S \land \neg strictly\text{-}redundant\ C\ S\ \}$

We first establish some basic syntactic properties.

lemma *tautology-monotonous* : $(tautology\ C) \Longrightarrow (C \subseteq D) \Longrightarrow (tautology\ D)$
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 $atoms\text{-}formula\ (simplify\ S) \subseteq atoms\text{-}formula\ S$
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*)
from ‹*subsumes C1 C2*› **and** ‹*strictly-maximal-literal C1 L1*› **have** *L1* ∈ *C2*
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 subsumes 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 vali-
date-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 impI)+)
        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 tau-
tology-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
              unfolding redundant-def strictly-redundant-def by auto
              from ‹D ⊂ C› and ‹C ∈ S› and ‹all-fulfill finite S› have card D <
card 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* **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* :: *'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-onI 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 *A1* **where** *at1*: *L1* = (*Pos A1*) ∨ *L1* = (*Neg A1*) **using** *Literal.exhaust* [*of L1*] **by** *auto*
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** ($L \in C$) **and** *Lismax*: ($\forall B.$ ($B \in C \wedge L \neq B$) $\longrightarrow$ (*literal-ordering* *B L*))
unfolding *strictly-maximal-literal-def* **by** *auto*
from ‹$L \in C$› **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*
from ‹*rename-literal A L* ≠ *B*› **and** ‹*B* = *rename-literal A B′*›
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 ∈ selected-part UNIV C
    show x ∈ rename-clause Sel (validated-part Sel (rename-clause Sel C))
    proof −
      from ‹x ∈ selected-part UNIV C› obtain A where x = Neg A and x ∈ C
        unfolding selected-part-def by auto
      from ‹x ∈ C› have rename-literal Sel x ∈ rename-clause Sel C
        unfolding rename-clause-def by blast
      show x ∈ rename-clause Sel (validated-part Sel (rename-clause Sel C))
      proof cases
        assume A ∈ Sel
        from this and ‹x = Neg A› have rename-literal Sel x = Pos A
          unfolding rename-literal-def by auto
        from this and ‹A ∈ Sel› have validate-literal Sel (rename-literal Sel x) by
auto
        from this and ‹rename-literal Sel x ∈ rename-clause Sel C›
        have rename-literal Sel x ∈ validated-part Sel (rename-clause Sel C)
          unfolding validated-part-def by auto
        thus x ∈ rename-clause Sel (validated-part Sel (rename-clause Sel C))
          using inverse-renaming rename-clause-def by auto
      next
        assume A ∉ Sel
        from this and ‹x = Neg A› have rename-literal Sel x = Neg A
          unfolding rename-literal-def by auto
        from this and ‹A ∉ Sel› have validate-literal Sel (rename-literal Sel x) by
auto
        from this and ‹rename-literal Sel x ∈ rename-clause Sel C›
        have rename-literal Sel x ∈ validated-part Sel (rename-clause Sel C)
          unfolding  validated-part-def by auto
        thus x ∈ rename-clause Sel (validated-part Sel (rename-clause Sel C))
          using inverse-renaming rename-clause-def by auto
      qed
    qed
  qed
  next
 show rename-clause Sel (validated-part Sel (rename-clause Sel C)) ⊆ (selected-part
UNIV C)
  proof
    fix x
    assume x ∈ rename-clause Sel (validated-part Sel (rename-clause Sel C))
    from this obtain y where y ∈ validated-part Sel (rename-clause Sel C)
      and x = rename-literal Sel y
      unfolding rename-clause-def validated-part-def by auto
    from ‹y ∈ validated-part Sel (rename-clause Sel C)› have
     y ∈ rename-clause Sel C and validate-literal Sel y unfolding validated-part-def
by auto
   from ‹y ∈ rename-clause Sel C› obtain z where z ∈ C and y = rename-literal
Sel z
      unfolding rename-clause-def by auto
    obtain A where zA: z = Pos A ∨ z = Neg A using Literal.exhaust [of z] by
```

```
auto
    show x ∈ selected-part UNIV C
    proof cases
        assume A ∈ Sel
        from this and zA and ‹y = rename-literal Sel z› have y = complement z
          using rename-literal-def by auto
        from this and ‹A ∈ Sel› and zA and ‹validate-literal Sel y› have y = Pos
A
          and z = Neg A by auto
        from this and  ‹A ∈ Sel› and ‹x = rename-literal Sel y› have x = Neg A
          unfolding rename-literal-def by auto
        from this and ‹z ∈ C› and ‹z = Neg A› show x ∈ selected-part UNIV C
          unfolding selected-part-def by auto
    next
        assume A ∉ Sel
        from this and zA and ‹y = rename-literal Sel z› have y = z
          using rename-literal-def by auto
        from this and ‹A ∉ Sel› and zA and ‹validate-literal Sel y› have y = Neg
A
          and z = Neg A by auto
        from this and  ‹A ∉ Sel› and ‹x = rename-literal Sel y› have x = Neg A
          unfolding rename-literal-def by auto
        from this and ‹z ∈ C› and ‹z = Neg A› show x ∈ selected-part UNIV C
          unfolding selected-part-def by auto
    qed
  qed
qed

lemma renaming-preserves-tautology:
  assumes tautology C
  shows tautology (rename-clause Sel C)
proof −
 from assms obtain A where Pos A ∈ C and Neg A ∈ C unfolding tautology-def
by auto
  from ‹Pos A ∈ C› have rename-literal Sel (Pos A) ∈  rename-clause Sel C
    unfolding rename-clause-def by auto
  from ‹Neg A ∈ C› have rename-literal Sel (Neg A) ∈  rename-clause Sel C
    unfolding rename-clause-def by auto
  show ?thesis
  proof cases
    assume A ∈ Sel
    from this have rename-literal Sel (Pos A) = Neg A
      and rename-literal Sel (Neg A) = (Pos A)
      unfolding rename-literal-def by auto
    from ‹rename-literal Sel (Pos A) = (Neg A)› and  ‹rename-literal Sel (Neg A)
= (Pos A)›
      and ‹rename-literal Sel (Pos A) ∈  (rename-clause Sel C)›
      and  ‹rename-literal Sel (Neg A) ∈  (rename-clause Sel C)›
      show tautology (rename-clause Sel C) unfolding tautology-def by auto
```

next
assume *A* ∉ *Sel*
from *this* **have** *rename-literal Sel* (*Pos A*) = *Pos A* **and** *rename-literal Sel* (*Neg A*) = (*Neg A*)
unfolding *rename-literal-def* **by** *auto*
from ‹*rename-literal Sel* (*Pos A*) = *Pos A*› **and** ‹*rename-literal Sel* (*Neg A*) = (*Neg A*)›
and ‹*rename-literal Sel* (*Pos A*) ∈ *rename-clause Sel C*›
and ‹*rename-literal Sel* (*Neg A*) ∈ *rename-clause Sel C*›
show *tautology* (*rename-clause Sel C*) **unfolding** *tautology-def* **by** *auto*
qed
qed

lemma *rename-union* : *rename-clause Sel* (*C* ∪ *D*) = *rename-clause Sel C* ∪ *rename-clause Sel D*
unfolding *rename-clause-def* **by** *auto*

lemma *renaming-set-minus-subset* :
rename-clause Sel (*C* − { *L* }) ⊆ *rename-clause Sel C* − {*rename-literal Sel L* }
proof
fix *x* **assume** *x* ∈ *rename-clause Sel* (*C* − { *L* })
then obtain *y* **where** *y* ∈ *C* − { *L* } **and** *x* = *rename-literal Sel y*
unfolding *rename-clause-def* **by** *auto*
from ‹*y* ∈ *C* − { *L* }› **and** ‹*x* = *rename-literal Sel y*› **have** *x* ∈ *rename-clause Sel C*
unfolding *rename-clause-def* **by** *auto*
have *x* ≠ *rename-literal Sel L*
proof
assume *x* = *rename-literal Sel L*
hence *rename-literal Sel x* = *L* **using** *inverse-renaming* **by** *auto*
from *this* **and** ‹*x* = *rename-literal Sel y*› **have** *y* = *L* **using** *inverse-renaming* **by** *auto*
from *this* **and** ‹*y* ∈ *C* − { *L* }› **show** *False* **by** *auto*
qed
from ‹*x* ≠ *rename-literal Sel L*› **and** ‹*x* ∈ *rename-clause Sel C*›
show *x* ∈ (*rename-clause Sel C*) − {*rename-literal Sel L* } **by** *auto*
qed

lemma *renaming-set-minus* : *rename-clause Sel* (*C* − { *L* })
= (*rename-clause Sel C*) − {*rename-literal Sel L* }
proof
show *rename-clause Sel* (*C* − { *L* }) ⊆ (*rename-clause Sel C*) − {*rename-literal Sel L* }
using *renaming-set-minus-subset* **by** *auto*
next
show (*rename-clause Sel C*) − {*rename-literal Sel L* } ⊆ *rename-clause Sel* (*C* − { *L* })
proof −
have *rename-clause Sel* ((*rename-clause Sel C*) − { (*rename-literal Sel L*) })

```
    ⊆ (rename-clause Sel (rename-clause Sel C)) − {rename-literal Sel (rename-literal
Sel L) }
    using renaming-set-minus-subset by auto
  from this
    have rename-clause Sel ( (rename-clause Sel C) − { (rename-literal Sel L) })
⊆ (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) }))
            ⊆ (rename-clause Sel (C − {L })) using rename-clause-def by auto
  from this
    show (rename-clause Sel C) − { (rename-literal Sel L) } ⊆ rename-clause Sel
(C − {L })
    using inverse-renaming inverse-clause-renaming by auto
 qed
qed

definition rename-interpretation :: 'at set ⇒ 'at Interpretation ⇒ 'at Interpreta-
tion
where
  rename-interpretation Sel I = { A. (A ∈ I ∧ A ∉ Sel) } ∪ { A. (A ∉ I ∧ A ∈
Sel) }

lemma renaming-preserves-semantic :
  assumes validate-literal I L
  shows validate-literal (rename-interpretation Sel I) (rename-literal Sel L)
proof −
  let ?J = rename-interpretation Sel I
    obtain A where L = Pos A ∨ L = Neg A using Literal.exhaust [of L] by auto
    from ‹L = Pos A ∨ L = Neg A› have atom L = A by auto
    show ?thesis
    proof cases
      assume A ∈ Sel
      from this and ‹atom L = A› have rename-literal Sel L = complement 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 = complement 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 = 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
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-preserves-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 ccontr*)
  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*:
  (*Sound resolvent*)
proof (*rule ccontr*)
  assume ¬ (*Sound resolvent*)
  then obtain *I P1 P2 C* **where**
    resolvent P1 P2 C validate-clause I P1 validate-clause I P2 **and** ¬*validate-clause I C*
    unfolding *Sound-def* **by** *blast*
  from ‹*resolvent P1 P2 C*› **obtain** *A* **where**
    (*Pos A*) ∈ *P1* **and** (*Neg A*) ∈ *P2* **and** *C* = ((*P1* − { *Pos A*}) ∪ (*P2* − { *Neg A* }))
    unfolding *resolvent-def* **by** *auto*
  show *False*
  proof *cases*
    assume *A* ∈ *I*
    hence ¬*validate-literal I* (*Neg A*) **by** *auto*
    from ‹¬*validate-literal I* (*Neg A*)› **and** ‹*validate-clause I P2*›
      have *validate-clause I* (*P2* − { *Neg A* }) **by** *auto*
    from ‹*validate-clause I* (*P2* − { *Neg A* })› **and** ‹*C* = ((*P1* − { *Pos A*}) ∪ (*P2* − { *Neg A* }))›
      and ‹¬*validate-clause I C*› **show** *False* **by** *auto*
  next
    assume *A* ∉ *I*
    hence ¬*validate-literal I* (*Pos A*) **by** *auto*
    from ‹¬*validate-literal I* (*Pos A*)› **and** ‹*validate-clause I P1*›
      have *validate-clause I* (*P1* − { *Pos A* }) **by** *auto*
    from ‹*validate-clause I* (*P1* − { *Pos A* })› **and** ‹*C* = ((*P1* − { *Pos A*}) ∪ (*P2* − { *Neg A* }))›
      and ‹¬*validate-clause I C*›
      show *False* **by** *auto*
  qed
qed

theorem *ordered-resolution-correct* : *Sound ordered-resolvent*
using *resolution-is-correct* **and** *ordered-resolvent-is-resolvent less-restrictive-correct*
by *auto*

theorem *ordered-model-resolution-correct* : *Sound* (*ordered-model-resolvent I*)
using *resolution-is-correct ordered-model-resolvent-is-resolvent less-restrictive-correct*
by *auto*

theorem *ordered-positive-resolution-correct* : *Sound ordered-positive-resolvent*

using *less-restrictive-correct positive-resolvent-is-resolvent resolution-is-correct* **by** *auto*

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*) ⟶ *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*

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*) =
      (λ*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*)))))
by *auto*
termination apply (*relation atom-ordering*)
by *auto* (*simp add*: *atom-ordering-wf*)

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 *cases*
    assume *c1*: (∀ *B*. (*Pos B* ∈ *C* ⟶ (*B*, *A*) ∈ *atom-ordering*
              ⟶ (¬(*canonic-int-fun-ordered B*) *S*)))
    show *?thesis*
    proof *cases*
      assume *c2*: (∀ *B*. (*Neg B* ∈ *C* ⟶ (*B*, *A*) ∈ *atom-ordering*
              ⟶ ((*canonic-int-fun-ordered B*) *S*)))
      have ((*canonic-int-fun-ordered A*) *S*)
      proof (*rule ccontr*)
        assume ¬ ((*canonic-int-fun-ordered A*) *S*)
        from ‹¬ ((*canonic-int-fun-ordered A*) *S*)›
        have *e*: ¬ (∃ *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*))))
        by ((*simp only*:*canonic-int-fun-ordered.simps*[*of A*]), *blast*)

```
      from e and c1 and c2 and ‹(C ∈ S)›and ‹(strictly-maximal-literal C (Pos
A))›
       show False by blast
     qed
     from ‹((canonic-int-fun-ordered A) S)› have A ∈ (canonic-int-ordered S)
       unfolding canonic-int-ordered-def by blast
      from ‹A ∈ (canonic-int-ordered S)› and ‹(strictly-maximal-literal C (Pos
A))›
       show ?thesis
       unfolding strictly-maximal-literal-def by auto
   next
     assume not-c2: ¬( ∀ B. ( Neg B ∈ C ⟶ (B, A) ∈ atom-ordering
                  ⟶ ((canonic-int-fun-ordered B) S)))
      from not-c2 obtain B where Neg B ∈ C and ¬((canonic-int-fun-ordered
B) S)
     by blast
     from ‹¬ ((canonic-int-fun-ordered B) S)› have B ∉ (canonic-int-ordered S)
       unfolding canonic-int-ordered-def by blast
     with ‹Neg B ∈ C› show ?thesis by auto
   qed
  next
   assume not-c1: ¬(∀ B. ( Pos B ∈ C ⟶ (B, A) ∈ atom-ordering
                  ⟶ (¬(canonic-int-fun-ordered B) S)))
    from not-c1 obtain B where Pos B ∈ C and ((canonic-int-fun-ordered B)
S)
     by blast
   from ‹((canonic-int-fun-ordered B) S)› have B ∈ (canonic-int-ordered S)
     unfolding canonic-int-ordered-def by blast
   with ‹Pos B ∈ C› show ?thesis by auto
qed

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 (induction n)
   show (?P 0) by auto
   next
     fix n assume ?P n
     show ?P (Suc n)
     proof
          fix C
          show (finite C ∧ card C = Suc n ∧ Suc n ≠ 0 ∧ ¬ (tautology C))
            ⟶ (∃ A. (strictly-maximal-literal C A))
          proof
            assume finite C ∧ card C = Suc n ∧ Suc n ≠ 0 ∧ ¬(tautology C)
             hence (finite C) and (card C) = (Suc n) and (¬ (tautology C)) by
auto
```

have $C \neq \{\}$
proof
assume $C = \{\}$
from ‹*finite C*› **and** ‹$C = \{\}$› **have** *card C* = *0* **using** *card-0-eq* **by** *auto*
from ‹*card C* = *0*› **and** ‹*card C* = *Suc n*› **show** *False* **by** *auto*
qed
then obtain L **where** $L \in C$ **by** *auto*
from ‹¬*tautology C*› **have** ¬*tautology* ($C - \{ L \}$) **using** *tautology-monotonous*
by *auto*
from ‹$L \in C$› **and** ‹*finite C*› **have** *Suc* (*card* ($C - \{ L \}$)) = *card C*
using *card-Suc-Diff1* **by** *metis*
with ‹*card C* = *Suc n*› **have** *card* ($C - \{ L \}$) = n **by** *auto*

show $\exists A$. (*strictly-maximal-literal C A*)
proof *cases*
assume *card C* = *1*
from *this* **and** ‹*card C* = *Suc n*› **have** $n = 0$ **by** *auto*
from *this* **and** ‹*finite C*› **and** ‹*card* ($C - \{ L \}$) = n› **have** $C - \{ L \} = \{\}$
using *card-0-eq* **by** *auto*
from *this* **and** ‹$L \in C$› **show** *?thesis* **unfolding** *strictly-maximal-literal-def* **by** *auto*
next
assume *card C* ≠ *1*
from ‹*finite C*› **have** *finite* ($C - \{ L \}$) **by** *auto*
from ‹*Suc* (*card* ($C - \{ L \}$)) = *card C*› **and** ‹*card C* ≠ *1*›
and ‹(*card* ($C - \{ L \}$)) = n› **have** $n \neq 0$ **by** *auto*
from *this* **and** ‹*finite* ($C - \{ L \}$)› **and** ‹*card* ($C - \{ L \}$) = n›
and ‹¬*tautology* ($C - \{ L \}$)› **and** ‹*?P n*›
obtain A **where** *strictly-maximal-literal* ($C - \{ L \}$) A **by** *metis*
show $\exists M$. *strictly-maximal-literal C M*
proof *cases*
assume (*atom L*, *atom A*) ∈ *atom-ordering*
from *this* **have** *literal-ordering L A* **by** *auto*
from *this* **and** ‹*strictly-maximal-literal* ($C - \{ L \}$) A›
have *strictly-maximal-literal C A*
unfolding *strictly-maximal-literal-def* **by** *blast*
thus *?thesis* **by** *auto*
next
assume (*atom L*, *atom A*) ∉ *atom-ordering*
have *l-cases*: L = (*Pos* (*atom L*)) ∨ L = (*Neg* (*atom L*))
by ((*rule atom-property* [*of* (*atom L*)]), *auto*)
have *a-cases*: A = (*Pos* (*atom A*)) ∨ A = (*Neg* (*atom A*))
by ((*rule atom-property* [*of* (*atom A*)]), *auto*)
from *l-cases* **and** *a-cases* **and**‹(*strictly-maximal-literal* ($C - \{ L \}$) A)›
and ‹¬ (*tautology C*)› **and** ‹$L \in C$›

have *atom L* ≠ *atom A*
unfolding *strictly-maximal-literal-def* **and** *tautology-def* **by** *auto*
from *this* **and** ‹(*atom L*, *atom A*) ∉ *atom-ordering*› **and** *atom-ordering-total*
have (*atom A*,*atom L*) ∈ *atom-ordering* **by** *auto*
hence *literal-ordering A L* **by** *auto*
from *this* **and** ‹*L* ∈ *C*› **and** ‹*strictly-maximal-literal* (*C* − { *L* }) *A*›
and *literal-ordering-trans*
have *strictly-maximal-literal C L* **unfolding** *strictly-maximal-literal-def*
unfolding *strictly-maximal-literal-def* **by** *blast*
thus *?thesis* **by** *auto*
qed
qed
qed
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 {} ∉ *S*
assumes *C* ∈ *S*
shows *validate-clause* (*canonic-int-ordered S*) *C*
proof *cases*
assume (*tautology C*)
thus *?thesis* **using** *tautologies-are-valid* [*of C* (*canonic-int-ordered S*)] **by** *auto*
next
assume ¬*tautology C*
from ‹*all-fulfill finite S*› **and** ‹*C* ∈ *S*› **have** *finite C* **using** *all-fulfill-def* **by** *auto*
from ‹{} ∉ *S*› **and** ‹*C* ∈ *S*› **and** ‹*finite C*› **have** *card C* ≠ *0* **using** *card-0-eq* **by** *auto*
from ‹¬*tautology C*› **and** ‹*finite C*› **and** ‹*card C* ≠ *0*› **obtain** *L*
where *strictly-maximal-literal C L* **using** *strictly-maximal-literal-exists* **by** *blast*
obtain *A* **where** *A* = *atom L* **by** *auto*

have *inductive-lemma*:
∀ *C L*. ((*C* ∈ *S*) ⟶ (*strictly-maximal-literal C L*) ⟶ (*A* = (*atom L*))
⟶ (*validate-clause* (*canonic-int-ordered S*) *C*)) (**is** (*?Q A*))
proof ((*rule wf-induct* [*of atom-ordering ?Q A*]),(*rule atom-ordering-wf*))
next
fix *x*
assume *hyp-induct*: ∀ *y*. (*y*,*x*) ∈ *atom-ordering* ⟶ (*?Q y*)
show *?Q x*
proof (*rule*)+
fix *C L* **assume** *C* ∈ *S* *strictly-maximal-literal C 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
            from ‹(canonic-int-fun-ordered x) S›
              have (∃ 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
from ‹¬(canonic-int-fun-ordered AA) S› have AA ∉ canonic-int-ordered
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 ‹M ∈ (D − { Pos x })› and ‹strictly-maximal-literal D (Pos
x)›
      and ‹M = (Neg AA)›
   have (AA,x) ∈ atom-ordering unfolding strictly-maximal-literal-def
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
 from ‹(canonic-int-fun-ordered AA) S› have AA ∈ canonic-int-ordered
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 $M \in (C - \{\ Neg\ x\ \})$
from ‹¬*validate-clause*(*canonic-int-ordered S*) *C*› **and** ‹$M \in (C - \{$ *Neg x* $\})$›
and ‹*validate-literal* (*canonic-int-ordered S*) *M*› **show** *False* **by** *auto*
qed
qed
from ‹¬*validate-clause* (*canonic-int-ordered S*) *R*› **have** ¬*tautology R*
using *tautologies-are-valid* **by** *auto*
from ‹*ordered-resolvent D C R*› **and** ‹$D \in S$› **and** ‹$C \in S$›
and ‹*saturated-binary-rule ordered-resolvent S*›
have *redundant R S* **unfolding** *saturated-binary-rule-def* **by** *auto*
from *this* **and** ‹¬*tautology R*› **obtain** R' **where** $R' \in S$ **and** *subsumes* R' R
unfolding *redundant-def* **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 ‹*all-fulfill finite S*› **and** ‹$C \in S$› **have** *finite C* **using** *all-fulfill-def* **by** *auto*
from ‹*all-fulfill finite S*› **and** ‹$D \in S$› **have** *finite D* **using** *all-fulfill-def* **by** *auto*
from ‹*finite C*› **and** ‹*finite D*› **and** ‹(*resolvent D C R*)› **have** *finite R*
using *resolvent-is-finite* **unfolding** *derived-clauses-are-finite-def* **by** *blast*
from ‹*finite R*› **and** ‹*subsumes* R' R› **have** *finite* R' **unfolding** *subsumes-def*
using *finite-subset* **by** *auto*
from ‹$R' \in S$› **and** ‹$\{\} \notin S$› **and** ‹(*subsumes* R' R)› **have** $R' \neq \{\}$
unfolding *subsumes-def* **by** *auto*
from ‹*finite* R'› **and** ‹$R' \neq \{\}$› **have** *card* $R' \neq 0$ **using** *card-0-eq* **by** *auto*
from ‹*subsumes* R' R› **and** ‹¬*tautology R*› **have** ¬*tautology* R'
unfolding *subsumes-def*
using *tautology-monotonous* **by** *auto*
from ‹¬*tautology* R'› **and** ‹*finite* R'› **and** ‹*card* $R' \neq 0$› **obtain** LR'
where *strictly-maximal-literal* R' LR' **using** *strictly-maximal-literal-exists*

by *blast*
from ‹*finite R*› **and** ‹*finite* R'› **and** ‹*card* $R' \neq 0$› **and** ‹*subsumes* R' R›
have *card* $R \neq 0$
unfolding *subsumes-def* **by** *auto*
from ‹¬*tautology R*› **and** ‹*finite R*› **and** ‹*card* $R \neq 0$› **obtain** *LR*
where *strictly-maximal-literal R LR* **using** *strictly-maximal-literal-exists* **by** *blast*
obtain *AR* **and** AR' **where** *AR* = *atom LR* **and** AR' = *atom* LR' **by** *auto*
from ‹*subsumes* R' R› **and** ‹*AR* = *atom LR*› **and** ‹AR' = *atom* LR'›
and ‹(*strictly-maximal-literal R LR*)›

and ‹(*strictly-maximal-literal* R' LR')› **have** ($AR' = AR$) ∨ (AR',AR) ∈ *atom-ordering*
using *subsumption-and-max-literal* **by** *auto*
from ‹R = (*resolvent-upon* D C x)› **and** ‹AR = *atom* LR›
and ‹*strictly-maximal-literal* R LR›
and ‹*strictly-maximal-literal* D (*Pos* x)›
and ‹*strictly-maximal-literal* C L› **and** ‹L = (*Neg* x)›
have (AR,x) ∈ *atom-ordering* **using** *resolution-and-max-literal* **by** *auto*
from ‹(AR,x) ∈ *atom-ordering*› **and** ‹($AR' = AR$) ∨ (AR',AR) ∈ *atom-ordering*›
have (AR',x) ∈ *atom-ordering* **using** *atom-ordering-trans* **by** *auto*
from *this* **and** *hyp-induct* **and** ‹$R' \in S$› **and** ‹*strictly-maximal-literal* R' LR'›
and ‹AR' = *atom* LR'› **have** *validate-clause* (*canonic-int-ordered* S) R' **by** *auto*
from *this* **and** ‹*subsumes* R' R› **and** ‹¬*validate-clause* (*canonic-int-ordered* S) R›
show *False* **using** *subsumption-and-semantics* **by** *blast*
qed
qed
qed
qed
from *inductive-lemma* **and** ‹$C \in S$› **and** ‹*strictly-maximal-literal* C L› **and** ‹A = *atom* L› **show** *?thesis* **by** *blast*
qed

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

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* ∧ *subsumes D C*)
    unfolding *empty-selected-part-def redundant-def* **by** *auto*
    from *this* **and** ‹*tautology C* ∨ (∃ *D*. *D* ∈ *S* ∧ *subsumes D C*)›
      and ‹¬*redundant C* (*empty-selected-part Sel S*)›

obtain *D* **where** $D \in S$ **and** *subsumes D C* **and** $D \notin$ *empty-selected-part Sel S*
unfolding *redundant-def* **by** *auto*
from ‹$D \notin$ *empty-selected-part Sel S*› **and** ‹$D \in S$› **obtain** *B* **where** $B \in Sel$ **and** *Neg* $B \in D$
unfolding *empty-selected-part-def selected-part-def* **by** *auto*
from ‹*Neg* $B \in D$› *this* **and** ‹*subsumes D C*› **have** *Neg* $B \in C$ **unfolding** *subsumes-def* **by** *auto*
from *this* **and** ‹$C = ((P1 - \{ Pos\ A\}) \cup (P2 - \{ Neg\ A \}))$› **have** *Neg* $B \in (P1 \cup P2)$ **by** *auto*
from ‹*Neg* $B \in (P1 \cup P2)$› **and** ‹$P1 \in$ *empty-selected-part Sel S*›
and ‹$P2 \in$ *empty-selected-part Sel S*› **and** ‹$B \in Sel$› **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* ⇒ *'at* ⇒ *bool*
where
ordered-sel-resolvent-upon Sel P1 P2 C A ≡
$(((C = ((P1 - \{ Pos\ A\}) \cup (P2 - \{ Neg\ A \})))$
∧ (*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-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 ‹*finite P2*› **have** *finite* (*selected-part Sel P2*) **unfolding** *selected-part-def* **by** *auto*
from ‹*card* (*selected-part Sel P2*) = (*Suc n*)› **have** *card* (*selected-part Sel P2*) ≠ *0* **by** *auto*
from *this* **and** ‹*finite* (*selected-part Sel P2*)› **have** *selected-part Sel P2* ≠ {}
using *card-0-eq* **by** *auto*
from *this* **and** ‹*ordered-sel-resolvent-upon Sel P1 P2 C A*› **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 }›
  have selected-part Sel C = 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
qed

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
```

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 ‹{} ∉ *S*› **have** {} ∉ *?SE* **unfolding** *empty-selected-part-def* **by** *auto*
from ‹*selected-part Sel C* = {}› ‹*C* ∈ *S*› **have** *C* ∈ *?SE* **unfolding** *empty-selected-part-def*
by *auto*
from ‹*all-fulfill finite S*› **have** *all-fulfill finite ?SE*
unfolding *empty-selected-part-def all-fulfill-def* **by** *auto*
from *this* **and** ‹*saturated-binary-rule ordered-resolvent ?SE*› **and** ‹{} ∉ *?SE*› **and** ‹*C* ∈ *?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* ≠ {} **by** *auto*
show *validate-clause ?I C*
proof (*rule ccontr*)
assume ¬*validate-clause ?I C*
show *False*
proof (*cases*)
assume *tautology C*
from ‹*tautology C*› **and** ‹¬*validate-clause ?I C*› **show** *False*
using *tautologies-are-valid* **by** *auto*
next
assume ¬(*tautology C*)
hence ¬(*tautology* (*selected-part Sel C*))
unfolding *selected-part-def tautology-def* **by** *auto*
from ‹*selected-part Sel C* ≠ {}› **and** ‹*finite* (*selected-part Sel C*)›
have *card* (*selected-part Sel C*) ≠ *0* **by** *auto*
from *this* **and** ‹¬(*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* ∈ (*selected-part Sel C*)
and *L* ∈ *C* **unfolding** *strictly-maximal-literal-def selected-part-def* **by** *auto*
from *this* **and** ‹¬*validate-clause ?I C*› **have** ¬(*validate-literal ?I L*) **by** *auto*
from ‹*L* ∈ (*selected-part Sel C*)› **obtain** *A* **where** *L* = (*Neg A*) **and** *A* ∈ *Sel*
unfolding *selected-part-def* **by** *auto*

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*) *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 ∃ *R′*. *R′* ∈ *S* ∧ (*subsumes R′ ?R*)
then obtain *R′* **where** *R′* ∈ *S* **and** *subsumes R′ ?R* **by** *auto*
from ‹*finite C*›**and** ‹*finite D*› **have** *finite ?R* **by** *auto*
from *this* **have** *finite* (*selected-part Sel ?R*) **unfolding** *selected-part-def*
by *auto*
from ‹*subsumes R′ ?R*› **have** *selected-part Sel R′* ⊆ *selected-part Sel*
?R
unfolding *selected-part-def* **and** *subsumes-def* **by** *auto*
from *this* **and** ‹*finite* (*selected-part Sel ?R*)›
have *card* (*selected-part Sel R′*) ≤ *card* (*selected-part Sel ?R*)
using *card-mono* **by** *auto*
from *this* **and** ‹*card* (*selected-part Sel ?R*) = *m*› **and** ‹*n* = *Suc m*›
have *card* (*selected-part Sel R′*) < *n* **by** *auto*
from *this* **and** *ind-hyp* **and** ‹*R′* ∈ *S*› **have** *validate-clause ?I R′* **by**
auto
from *this* **and** ‹*subsumes R′ ?R*› **show** *validate-clause ?I ?R*
using *subsumption-and-semantics* [*of R′ ?R ?I*] **by** *auto*
qed
from *this* **obtain** *L′* **where** *L′* ∈ *?R* **and** *validate-literal ?I L′* **by** *auto*
have *L′* ∉ *D* − { *Pos A* }
proof
assume *L′* ∈ *D* − { *Pos A* }
from *this* **have** *L′* ∈ *D* **by** *auto*
let *?A′* = *atom L′*
have *L′* = (*Pos ?A′*) ∨ *L′* = (*Neg ?A′*) **using** *atom-property* [*of ?A′*
L′] **by** *auto*
thus *False*
proof
assume *L′* = (*Pos ?A′*)
from *this* **and** ‹*strictly-maximal-literal D* (*Pos A*)› **and** ‹*L′* ∈ *D* −
{ *Pos A* }›
have (*?A′*,*A*) ∈ *atom-ordering* **unfolding** *strictly-maximal-literal-def*
by *auto*
from *c1*
have *c1′*: *Pos ?A′* ∈ *D* ⟶ (*?A′*, *A*) ∈ *atom-ordering*
⟶ (¬(*canonic-int-fun-ordered ?A′*) *?SE*)
by *blast*
from ‹*L′* ∈ *D*› **and** ‹*L′* = *Pos ?A′*› **have** *Pos ?A′* ∈ *D* **by** *auto*
from *c1′* **and** ‹*Pos ?A′* ∈ *D*› **and** ‹(*?A′*,*A*) ∈ *atom-ordering*›
have ¬(*canonic-int-fun-ordered ?A′*) *?SE* **by** *blast*
from *this* **have** *?A′* ∉ *?I* **unfolding** *canonic-int-ordered-def* **by**
blast
from *this* **have** ¬(*validate-literal ?I* (*Pos ?A′*)) **by** *auto*
from *this* **and** ‹*L′* = *Pos ?A′*› **and** ‹*validate-literal ?I L′*› **show**
False **by** *auto*
next
assume *L′* = *Neg ?A′*
from *this* **and** ‹*strictly-maximal-literal D* (*Pos A*)› **and** ‹*L′* ∈ *D* −

```
{ Pos A }›
                have (?A',A) ∈ atom-ordering unfolding strictly-maximal-literal-def
by auto
                  from c2
                    have c2': Neg ?A' ∈ D ⟶ (?A', A) ∈ atom-ordering
                      ⟶ (canonic-int-fun-ordered ?A') ?SE
                    by blast
                  from ‹L' ∈ D› and ‹L' = (Neg ?A')› have Neg ?A' ∈ D by auto
                  from c2' and ‹Neg ?A' ∈ D› and ‹(?A',A) ∈ atom-ordering›
                  have (canonic-int-fun-ordered ?A') ?SE by blast
                    from this have ?A' ∈ ?I unfolding canonic-int-ordered-def by
blast
                  from this have ¬validate-literal ?I (Neg ?A') by auto
                    from this and ‹L' = Neg ?A'› and ‹validate-literal ?I L'› show
False by auto
                qed
              qed
            from this and ‹L' ∈ ?R› have L' ∈ C by auto
            from this and ‹validate-literal ?I L'› and ‹¬validate-clause ?I C› show
False by auto
          qed
        qed
      qed
    qed
  qed
  from ‹?P n› and ‹n = card (selected-part Sel C)› and ‹C ∈ S› show ?thesis by
auto
qed

theorem ordered-resolution-is-complete-ordered-sel :
  Complete (ordered-sel-resolvent Sel)
proof (rule ccontr)
  assume ¬Complete (ordered-sel-resolvent Sel)
  then obtain S where saturated-binary-rule (ordered-sel-resolvent Sel) S
    and all-fulfill finite S
    and {} ∉ S
    and ¬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 ¬(validate-formula ?I S)
    from this obtain C where C ∈ S and ¬(validate-clause ?I C) by auto
    from ‹saturated-binary-rule (ordered-sel-resolvent Sel) S› and ‹all-fulfill finite
S›
      and ‹{} ∉ S› and ‹C ∈ S› and ‹¬(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 ‹¬(satisfiable S)› show False
```

unfolding *satisfiable-def* **by** *blast*
qed

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-sel-resolvent UNIV P1 P2 C*
shows *ordered-model-resolvent Sel* (*rename-clause Sel P1*) (*rename-clause Sel P2*)
(*rename-clause Sel C*)
proof −
from *assms* **obtain** *A*
where *c-def*: *C* = ((*P1* − { *Pos A* }) ∪ (*P2* − { *Neg A* }))
and *selected-part UNIV P1* = {}
and *strictly-maximal-literal P1* (*Pos A*)
and *disj*: ((*strictly-maximal-literal P2* (*Neg A*)) ∧ (*selected-part UNIV P2*) = {})
∨ *strictly-maximal-literal* (*selected-part UNIV P2*) (*Neg A*)
unfolding *ordered-sel-resolvent-def* **by** *blast*
have *rename-clause Sel* ((*P1* − { *Pos A* }) ∪ (*P2* − { *Neg A* }))
= (*rename-clause Sel* (*P1* − { *Pos A* })) ∪ *rename-clause Sel* (*P2* − { (*Neg A*) })
using *rename-union* [*of Sel P1* − { *Pos A* } *P2* − { *Neg A* }] **by** *auto*
from *this* **and** *c-def* **have** *ren-c*: (*rename-clause Sel C*) =
(*rename-clause Sel* (*P1* − { *Pos A* })) ∪ *rename-clause Sel* (*P2* − { (*Neg A*) }) **by** *auto*
have *m1*: (*rename-clause Sel* (*P1* − { *Pos A* })) = (*rename-clause Sel P1*)
− { *rename-literal Sel* (*Pos A*) }
using *renaming-set-minus* **by** *auto*
have *m2*: *rename-clause Sel* (*P2* − { *Neg A* }) = (*rename-clause Sel P2*)
− { *rename-literal Sel* (*Neg A*) }
using *renaming-set-minus* **by** *auto*
from *m1* **and** *m2* **and** *ren-c* **have**
rc-def: (*rename-clause Sel C*) =
((*rename-clause Sel P1*) − { *rename-literal Sel* (*Pos A*) })
∪ ((*rename-clause Sel P2*) − { *rename-literal Sel* (*Neg A*) })
by *auto*
have ¬((*strictly-maximal-literal P2* (*Neg A*)) ∧ (*selected-part UNIV P2*) = {})
proof
assume (*strictly-maximal-literal P2* (*Neg A*)) ∧ (*selected-part UNIV P2*) = {}
from *this* **have** *strictly-maximal-literal P2* (*Neg A*) **and** *selected-part UNIV P2* = {} **by** *auto*
from ‹*strictly-maximal-literal P2* (*Neg A*)› **have** *Neg A* ∈ *P2*
unfolding *strictly-maximal-literal-def* **by** *auto*
from *this* **and** ‹*selected-part UNIV P2* = {}› **show** *False* **unfolding** *selected-part-def* **by** *auto*

qed
from *this* **and** *disj* **have** *strictly-maximal-literal* (*selected-part UNIV P2*) (*Neg A*) **by** *auto*
from *this* **have** *strictly-maximal-literal* (*rename-clause Sel* (*validated-part Sel* (*rename-clause Sel P2*))) (*Neg A*)
using *renaming-and-selected-part* **by** *auto*
from *this* **have**
strictly-maximal-literal (*rename-clause Sel* (*rename-clause Sel* (*validated-part Sel* (*rename-clause Sel P2*))))
(*rename-literal Sel* (*Neg A*)) **using** *renaming-preserves-strictly-maximal-literal* **by** *auto*
from *this* **have**
p1: *strictly-maximal-literal* (*validated-part Sel* (*rename-clause Sel P2*))
(*rename-literal Sel* (*Neg A*))
using *inverse-clause-renaming* **by** *auto*
from ‹*strictly-maximal-literal P1* (*Pos A*)›
have *p2*: *strictly-maximal-literal* (*rename-clause Sel P1*) (*rename-literal Sel* (*Pos A*))
using *renaming-preserves-strictly-maximal-literal* **by** *auto*
from ‹(*selected-part UNIV P1*) = {}› **have**
rename-clause Sel (*validated-part Sel* (*rename-clause Sel P1*)) = {}
using *renaming-and-selected-part* **by** *auto*
from *this* **have** *q*: *validated-part Sel* (*rename-clause Sel P1*) = {}
unfolding *rename-clause-def* **by** *auto*
have *r*: *rename-literal Sel* (*Neg A*) = *complement* (*rename-literal Sel* (*Pos A*))
unfolding *rename-literal-def* **by** *auto*
from *r* **and** *q* **and** *p1* **and** *p2* **and** *rc-def* **show**
ordered-model-resolvent Sel (*rename-clause Sel P1*) (*rename-clause Sel P2*)(*rename-clause Sel C*)
using *ordered-model-resolvent-def* [*of Sel rename-clause Sel P1 rename-clause Sel P2*
rename-clause Sel C] **by** *auto*
qed

theorem *ordered-resolution-is-complete-model-resolution*:
Complete (*ordered-model-resolvent Sel*)
proof (*rule ccontr*)
assume ¬*Complete* (*ordered-model-resolvent Sel*)
then obtain *S* **where** *saturated-binary-rule* (*ordered-model-resolvent Sel*) *S*
and {} ∉ *S* **and** *all-fulfill finite S* **and** ¬(*satisfiable S*) **unfolding** *Complete-def* **by** *auto*
let *?S′* = *rename-formula Sel S*
have {} ∉ *?S′*
proof
assume {} ∈ *?S′*
then obtain *V* **where** *V* ∈ *S* **and** *rename-clause Sel V* = {} **unfolding** *rename-formula-def* **by** *auto*
from ‹*rename-clause Sel V* = {}› **have** *V* = {} **unfolding** *rename-clause-def* **by** *auto*

from *this* and ‹*V* ∈ *S*› and ‹{} ∉ *S*› show *False* by *auto*
qed
from ‹*all-fulfill finite S*› have *all-fulfill finite ?S′*
unfolding *all-fulfill-def rename-formula-def rename-clause-def* by *auto*
have *saturated-binary-rule* (*ordered-sel-resolvent UNIV*) *?S′*
proof (*rule ccontr*)
assume ¬(*saturated-binary-rule* (*ordered-sel-resolvent UNIV*) *?S′*)
from *this* obtain *P1* and *P2* and *C* where *P1* ∈ *?S′* and *P2* ∈ *?S′*
and *ordered-sel-resolvent UNIV P1 P2 C* and ¬*tautology C*
and *not-subsumed*: ∀ *D*. (*D* ∈ *?S′* ⟶ ¬*subsumes D C*)
unfolding *saturated-binary-rule-def redundant-def* by *auto*
from ‹*ordered-sel-resolvent UNIV P1 P2 C*›
have *ord-ren*: *ordered-model-resolvent Sel* (*rename-clause Sel P1*) (*rename-clause Sel P2*)
(*rename-clause Sel C*)
using *ordered-res-with-selection-is-model-res* by *auto*
have ¬*tautology* (*rename-clause Sel C*)
using *renaming-preserves-tautology inverse-clause-renaming*
by (*metis* ‹¬ *tautology C*› *inverse-clause-renaming renaming-preserves-tautology*)
from ‹*P1* ∈ *?S′*› have *rename-clause Sel P1* ∈ *rename-formula Sel ?S′*
unfolding *rename-formula-def* by *auto*
hence *rename-clause Sel P1* ∈ *S* using *inverse-formula-renaming* by *auto*
from ‹*P2* ∈ *?S′*› have *rename-clause Sel P2* ∈ *rename-formula Sel ?S′*
unfolding *rename-formula-def* by *auto*
hence *rename-clause Sel P2* ∈ *S* using *inverse-formula-renaming* by *auto*
from ‹¬*tautology* (*rename-clause Sel C*)› and *ord-ren*
and ‹*saturated-binary-rule* (*ordered-model-resolvent Sel*) *S*›
and ‹*rename-clause Sel P1* ∈ *S*› and ‹*rename-clause Sel P2* ∈ *S*›
obtain *D′* where *D′* ∈ *S* and *subsumes D′* (*rename-clause Sel C*)
unfolding *saturated-binary-rule-def redundant-def* by *blast*
from ‹*subsumes D′* (*rename-clause Sel C*)›
have *subsumes* (*rename-clause Sel D′*) (*rename-clause Sel* (*rename-clause Sel C*))
using *renaming-preserves-subsumption* by *auto*
hence *subsumes* (*rename-clause Sel D′*) *C* using *inverse-clause-renaming* by *auto*
from ‹*D′* ∈ *S*› have *rename-clause Sel D′* ∈ *?S′* unfolding *rename-formula-def* by *auto*
from *this* and *not-subsumed* and ‹*subsumes* (*rename-clause Sel D′*) *C*› show *False* by *auto*
qed
from *this* and ‹{} ∉ *?S′*› and ‹*all-fulfill finite ?S′*› have *satisfiable ?S′*
using *ordered-resolution-is-complete-ordered-sel* unfolding *Complete-def* by *auto*
hence *satisfiable* (*rename-formula Sel ?S′*) using *renaming-preserves-satisfiability* by *auto*
from *this* and ‹¬*satisfiable S*› show *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* :
  validate-literal {} $L = (\exists A.\ (L = Neg\ A))$
by (*meson empty-iff validate-literal.elims*(*2*) *validate-literal.simps*(*2*))

lemma *universal-interpretation-validate* :
  validate-literal UNIV $L = (\exists A.\ (L = Pos\ A))$
by (*meson UNIV-I validate-literal.elims*(*2*) *validate-literal.simps*(*1*))

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

lemma *positive-part-lemma*:
  (*positive-part C*) = (*validated-part UNIV C*)
unfolding *positive-part-def validated-part-def* **using** *universal-interpretation-validate*
by *blast*

lemma *negative-resolvent-is-model-res*:
  less-restrictive ordered-negative-resolvent (*ordered-model-resolvent 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*:
  less-restrictive ordered-positive-resolvent (*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* (*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*

then have *card* (*positive-part* (*rename-clause I P1*)) ≤ *1* **unfolding** *Horn-def* **by** *auto*
from ‹*negative-part* (*rename-clause I P1*) = {}›
have *rename-clause I P1* = (*positive-part* (*rename-clause I P1*))
using *decomposition-clause-pos-neg* **by** *auto*
from *this* **and** ‹*card* (*positive-part* (*rename-clause I P1*)) ≤ *1*›
have *card* (*rename-clause I P1*) ≤ *1* **by** *auto*
from ‹*strictly-maximal-literal P1 L*› **have** *P1* ≠ {}
unfolding *strictly-maximal-literal-def* **by** *auto*
then have *rename-clause I P1* ≠ {} **unfolding** *rename-clause-def* **by** *auto*
from ‹*all-fulfill finite S*› **and** ‹*P1* ∈ *S*› **have** *finite P1* **unfolding** *all-fulfill-def* **by** *auto*
then have *finite* (*rename-clause I P1*) **unfolding** *rename-clause-def* **by** *auto*
from *this* **and** ‹*rename-clause I P1* ≠ {}› **have** *card*(*rename-clause I P1*) ≠ *0*

using *card-0-eq* **by** *auto*
from *this* **and** ‹*card* (*rename-clause I P1*) ≤ *1*› **have** *card* (*rename-clause I P1*) = *1* **by** *auto*
then have *card P1* = *1* **using** *renaming-preserves-cardinality* **by** *auto*
then have *Unit P1* **unfolding** *Unit-def* **using** *card-image* **by** *auto*
from *this* **and** ‹*L* ∈ *P1*› **and** ‹*complement L* ∈ *P2*› **and** *def-c* **have** *unit-resolvent P1 P2 C*
unfolding *unit-resolvent-def* **by** *auto*
from *this* **and** ‹¬(*redundant C S*)› **and** ‹*P1* ∈ *S*› **and** ‹*P2* ∈ *S*›
and ‹*saturated-binary-rule unit-resolvent S*›
show *False* **unfolding** *saturated-binary-rule-def* **by** *auto*
qed
from *this* **and** ‹*all-fulfill finite S*› **and** ‹{} ∉ *S*› **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 M*))
}

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 ccontr*)
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** ‹*all-fulfill finite* (*inferred-clause-sets R S N*)›
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*]
all-inferred-clause-sets-are-finite [*of 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 $D \in$ *inferred-clause-sets R S N*
assumes $D \notin$ *saturation R S*
assumes *all-fulfill finite S*
assumes *derived-clauses-are-finite R*
shows $\exists M$. *strictly-redundant D* (*inferred-clause-sets R S M*)
proof (*rule ccontr*)
assume *hyp*: $\neg(\exists M$. *strictly-redundant D* (*inferred-clause-sets R S M*))
{
fix *M*
have $D \in$ (*inferred-clause-sets R S* (*N+M*))
proof (*induction M*)
show $D \in$ *inferred-clause-sets R S* (*N+0*) **using** *assms*(*1*) **by** *auto*
next
fix *M* **assume** $D \in$ *inferred-clause-sets R S* (*N+M*)
from *this* **have** $D \in$ *add-all-deducible-clauses R* (*inferred-clause-sets R S* (*N+M*)) **by** *auto*
show $D \in$ (*inferred-clause-sets R S* (*N+*(*Suc M*)))
proof (*rule ccontr*)
assume $D \notin$ (*inferred-clause-sets R S* (*N+*(*Suc M*)))
from *this* **and** ‹$D \in$ *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*

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
qed
}
from *assms*(*2*) **and** *assms*(*1*) **have** ¬(∀ *M'*. (*M'* ≥ *N* ⟶ *D* ∈ *inferred-clause-sets R S M'*)) **by** *auto*
from *this* **obtain** *M'* **where** *M'* ≥ *N* **and** *D* ∉ *inferred-clause-sets R S M'* **by** *auto*
from ‹*M'* ≥ *N*› **obtain** *N'*:: *nat* **where** *N'* = *M'* − *N* **by** *auto*
have *D* ∈ *inferred-clause-sets R S* (*N*+(*M'*−*N*))
by (*simp add*: ‹⋀*M*. *D* ∈ *inferred-clause-sets R S* (*N* + *M*)›)
from *this* **and** ‹*D* ∉ *inferred-clause-sets R S M'*› **show** *False* **by** (*simp add*: ‹*N* ≤ *M'*›)
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 *?nat-order* = { (*x*::*nat*,*y*::*nat*). *x* < *y*}
{
fix *I* **have** ∀ *C N*. *finite C* ⟶ *card C* = *I*
⟶ (*redundant C* (*inferred-clause-sets R S N*)) ⟶ *redundant C* (*saturation R S*) (**is** *?P I*)
proof ((*rule wf-induct* [*of ?nat-order ?P I*]),(*simp add*:*wf*))
fix *I* **assume** *hyp-induct*: ∀ *J*. (*J*,*I*) ∈ *?nat-order* ⟶ (*?P J*)
show *?P I*
proof ((*rule allI*)+,(*rule impI*)+)
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 ¬*tautology C*
from *this* **and** ‹*redundant C* (*inferred-clause-sets R S N*)› **obtain** *D*
where *subsumes D C* **and** *D* ∈ *inferred-clause-sets R S N* **unfolding** *redundant-def* **by** *auto*

show *redundant C* (*saturation R S*)
proof (*cases*)
assume $D \in$ *saturation R S*
from *this* **and** ‹*subsumes D C*› **show** *redundant C* (*saturation R S*)
unfolding *redundant-def* **by** *auto*
next
assume $D \notin$ *saturation R S*
from *assms*(*2*) *assms*(*3*) **and** ‹$D \in$ *inferred-clause-sets R S N*› **and** ‹D $\notin$ *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** $D' \subset D$ **and** $D' \in$ *inferred-clause-sets R S M*
unfolding *strictly-redundant-def* **by** *auto*

from ‹$D' \subset D$› **and** ‹*subsumes D C*› **have** $D' \subset C$ **unfolding** *subsumes-def* **by** *auto*
from ‹$D' \subset C$› **and** ‹*finite C*› **have** *finite* D'
by (*meson psubset-imp-subset rev-finite-subset*)
from ‹$D' \subset C$› **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*) $\in$ *?nat-order* **by** *auto*
from ‹$D' \in$ *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*) $\in$ *?nat-order*› **have** *?P* (*card* D') **by** *force*
from *this* **and** ‹*finite* D'› **and** ‹*redundant* D' (*inferred-clause-sets R S M*)› **have**
redundant D' (*saturation R S*) **by** *auto*
show *redundant C* (*saturation R S*)
by (*meson* ‹$D' \subset C$› ‹*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*:
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** ‹*C* ∈ *add-all-deducible-clauses R* (*inferred-clause-sets R S ?N*)›
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 Suc ?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*)]
by *auto*

qed
}
from *this* **and** ‹*C* ∈ *inferred-clause-sets R S N*› **and** ‹¬ *entails S C*› **show** *False*
unfolding *entails-formula-def* **by** *auto*
qed
have *entails-formula* (*saturation R S*) *S*
proof (*rule ccontr*)
assume ¬ *entails-formula* (*saturation R S*) *S*
then obtain *C* **where** *C* ∈ *S* **and** ¬ *entails* (*saturation R S*) *C*
unfolding *entails-formula-def* **by** *auto*
from ‹*C* ∈ *S*› **have** *redundant C S* **unfolding** *redundant-def subsumes-def* **by** *auto*
from *assms*(*3*) **and** ‹*redundant C S*› **have** *redundant C* (*inferred-clause-sets R S 0*)
using *simplify-preserves-redundancy* **by** *auto*
from *assms*(*3*) **and** ‹*C* ∈ *S*› **have** *finite C* **unfolding** *all-fulfill-def* **by** *auto*
from ‹*redundant C* (*inferred-clause-sets R S 0*)› *assms*(*2*) *assms*(*3*) ‹*finite C*›

have *redundant C* (*saturation R S*)
using *persistent-clauses-subsume-redundant-clauses* [*of C R S 0*] **by** *auto*
from *this* **and** ‹¬ *entails* (*saturation R S*) *C*› **show** *False*
using *entails-formula-def redundancy-implies-entailment* **by** *auto*
qed
from ‹*entails-formula S* (*saturation R S*)› **and** ‹*entails-formula* (*saturation R S*) *S*›
show *?thesis*
unfolding *equivalent-def* **by** *auto*
qed

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
  (*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 L ∈ C and validate-literal ?J L 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-literal ?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
```

```
      from ‹D ∈ S› have D−C ∈ ?peval ∨  (∃ L. (L ∈ C) ∧ (complement L) ∈
D)
      by auto
      thus False
      proof
        assume ∃ L. (L ∈ C) ∧ (complement L) ∈ D
        then obtain L where L ∈ C and complement L ∈ D by auto
       obtain X where L = (Pos X) ∨ L = (Neg X) using Literal.exhaust [of L]
by auto
          from this  and ‹L ∈ C› and ‹¬(tautology C)› have validate-literal ?J
(complement L)
        unfolding tautology-def by auto
        from ‹(validate-literal ?J (complement L))› and ‹(complement L) ∈ D›
          and ‹¬(validate-clause ?J D)›
        show False by auto
      next
        assume D−C ∈ ?peval
        from ‹D−C ∈ ?peval› and ‹(validate-formula I ?peval)›
        have validate-clause I (D−C) using validate-formula.simps by blast
        from this obtain L where L ∈ D and L ∉ C and validate-literal I L 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 ‹validate-literal I L› and ‹L ∉
C›
        have validate-literal ?J L unfolding tautology-def by auto
        from ‹validate-literal ?J L› and ‹L ∈ D› and ‹¬(validate-clause ?J D)›
        show False by auto
      qed
    qed
    from ‹¬validate-clause ?J C› and ‹validate-formula ?J S› and ‹entails S C›
show False
    unfolding entails-def by auto
qed

lemma entailment-and-implicates:
  assumes entails-formula S1 S2
  shows implicates S2 ⊆ implicates S1
using assms entailed-formula-entails-implicates implicates-def by auto

lemma equivalence-and-implicates:
  assumes equivalent S1 S2
  shows implicates S1 = implicates S2
using assms entailment-and-implicates equivalent-def by blast

lemma equivalence-and-prime-implicates:
  assumes equivalent S1 S2
  shows prime-implicates S1 = prime-implicates S2
using 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
    from ‹entails S C› and ‹¬tautology C› have ¬satisfiable ?peval
    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* $x \lor (\exists y.\ (y \in (\textit{implicates } S)) \land (y \subset x))$›
obtain y **where** $y \in$ *implicates* S **and** $y \subset x$ **by** *auto*
from ‹$y \in$ *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 ‹$y \subset x$› **and** ‹$\neg$*tautology* x› **have** $\neg$*tautology* y **unfolding** *tautology-def* **by** *auto*
with ‹*redundant* y S› **obtain** z **where** $z \in S$ **and** $z \subseteq y$
unfolding *redundant-def subsumes-def* **by** *auto*
with ‹$y \subset x$› **have** $z \subset x$ **by** *auto*
with ‹$z \in S$› **show** *strictly-redundant* x S **using** *strictly-redundant-def* **by** *auto*
qed
with ‹*non-redundant* S› **and** ‹$x \in S$› **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** $S \subseteq$ *prime-implicates* S **using** *prime-implicates-generation-correct* **by** *auto*
show *prime-implicates* $S \subseteq S$
proof
fix x **assume** $x \in$ *prime-implicates* S
from *this* **have** $x \in$ *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 $x \in S$
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 ‹$S \subseteq$ *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 ‹$x \in$ *prime-implicates* S› **show** *False*
unfolding *prime-implicates-def simplify-def* **by** *auto*
qed
qed
qed

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 (*all-resolvents-upon S A*) = { *C*. ∃ *P1 P2*. *P1* ∈ *S* ∧ *P2* ∈ *S* ∧ *C* = (*resolvent-upon P1 P2 A*) }

lemma *resolvent-upon-correct*:
  assumes *P1* ∈ *S*
  assumes *P2* ∈ *S*
  assumes *C* = *resolvent-upon P1 P2 A*
  shows *entails S C*
proof *cases*
  assume *Pos A* ∈ *P1* ∧ *Neg A* ∈ *P2*
  with ‹*C* = *resolvent-upon P1 P2 A*› **have** *resolvent P1 P2 C*
    unfolding *resolvent-def* **by** *auto*
  with ‹*P1* ∈ *S*› **and** ‹*P2* ∈ *S*› **show** *?thesis*
    using *soundness-and-entailment resolution-is-correct* **by** *auto*
next
  assume ¬ (*Pos A* ∈ *P1* ∧ *Neg A* ∈ *P2*)
  with ‹*C* = *resolvent-upon P1 P2 A*› **have** *P1* ⊆ *C* ∨ *P2* ⊆ *C* **by** *auto*
  with ‹*P1* ∈ *S*› **and** ‹*P2* ∈ *S*› **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* ∪ (*all-resolvents-upon S A*))
using *assms* **unfolding** *all-fulfill-def all-resolvents-upon-def* **by** *auto*

lemma *atoms-formula-resolvents*:
  shows *atoms-formula* (*all-resolvents-upon S A*) ⊆ *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
  (*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* ($S \cup R$)
assumes C = *resolvent-upon P1 P2 A*
shows *redundant C* ($S \cup R$)
proof (*rule ccontr*)
assume ¬*redundant C* ($S \cup R$)
from ‹C = *resolvent-upon P1 P2 A*› **have** C = (*P1* − { *Pos A* }) ∪ (*P2* − { *Neg A* }) **by** *auto*
from ‹¬*redundant C* ($S \cup R$)› **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* ($S \cup R$)› **show** *False* **unfolding** *redundant-def* **by** *auto*
next
assume $A \neq B$
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* ($S \cup R$)› **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*

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 $A \neq B$
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
qed
with ‹*redundant P2 S*› **obtain** *Q2* **where** *Q2* ∈ *S* **and** *subsumes Q2 P2*
unfolding *redundant-def* **by** *auto*
let *?res* = (*Q1* − { *Pos A* }) ∪ (*Q2* − { *Neg A* })
have *?res* = *resolvent-upon Q1 Q2 A* **by** *auto*
from *this* **and** ‹*partial-saturation S A* (*S* ∪ *R*)› **and** ‹*Q1* ∈ *S*› **and** ‹*Q2* ∈ *S*›

have *redundant ?res* (*S* ∪ *R*)
unfolding *partial-saturation-def* **by** *auto*
from ‹*subsumes Q1 P1*› **and** ‹*subsumes Q2 P2*› **and** ‹*C* = (*P1* − { *Pos A* }) ∪ (*P2* − { *Neg A* })›
have *subsumes ?res C* **unfolding** *subsumes-def* **by** *auto*
with ‹*redundant ?res* (*S* ∪ *R*)› **and** ‹¬*redundant C* (*S* ∪ *R*)› **show** *False*
using *subsumption-preserves-redundancy* **by** *auto*
qed

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* (λ*x*. ¬(*tautology x*)) *S*
assumes *P1* ∈ *S* ∪ *R*
shows *in-all-resolvents-upon S A P1*
proof ((*cases P1* ∈ *R*),(*metis all-fulfill-def assms*(*1*)))
next
assume *P1* ∉ *R*
with ‹*P1* ∈ *S* ∪ *R*› **have** *P1* ∈ *S* **by** *auto*
with ‹(*all-fulfill* (λ*x*. ¬(*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
    from this have P1 = (P1 − { Pos A }) ∪ (P1 − { Neg A }) by auto
    with ‹P1 ∈ S› show ?thesis unfolding resolvent-def
      unfolding 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 partial-saturation S E1 S
  assumes partial-saturation S E2 (S ∪ R)
  assumes all-fulfill (λx. ¬(tautology x)) S
  assumes all-fulfill (in-all-resolvents-upon S E2) R
  shows partial-saturation (S ∪ R) E1 (S ∪ R)
proof (rule ccontr)
  assume ¬ partial-saturation (S ∪ R) E1 (S ∪ R)
  from this obtain P1 P2 C where P1 ∈ S ∪ R and P2 ∈ S ∪ R and C =
resolvent-upon P1 P2 E1
    and ¬ redundant C (S ∪ R)
    unfolding partial-saturation-def by auto
  from ‹C = resolvent-upon P1 P2 E1› have C = (P1 − { Pos E1 }) ∪ (P2 −
{ Neg E1 }) by auto
  from ‹P1 ∈ S ∪ R› and assms(4) and ‹(all-fulfill (λx. ¬(tautology x)) S )›
  have in-all-resolvents-upon S E2 P1 using every-clause-is-a-resolvent by auto
  then obtain P1-1 P1-2 where P1-1 ∈ S and P1-2 ∈ S and P1 = resol-
vent-upon P1-1 P1-2 E2
    using every-clause-is-a-resolvent unfolding in-all-resolvents-upon-def by blast
  from ‹P2 ∈ S ∪ R› and assms(4) and ‹(all-fulfill (λx. ¬(tautology x)) S )›
    have in-all-resolvents-upon S E2 P2 using every-clause-is-a-resolvent by auto
  then obtain P2-1 P2-2 where P2-1 ∈ S and P2-2 ∈ S and P2 = resol-
vent-upon P2-1 P2-2 E2
    using every-clause-is-a-resolvent unfolding in-all-resolvents-upon-def by blast
  let ?R1 = resolvent-upon P1-1 P2-1 E1
  from ‹partial-saturation S E1 S› and ‹P1-1 ∈ S› and ‹P2-1 ∈ S› have redun-
dant ?R1 S
    unfolding partial-saturation-def by auto
  let ?R2 = resolvent-upon P1-2 P2-2 E1
  from ‹partial-saturation S E1 S› and ‹P1-2 ∈ S› and ‹P2-2 ∈ S› have redun-
dant ?R2 S
    unfolding partial-saturation-def by auto
  let ?C = resolvent-upon ?R1 ?R2 E2
  from ‹C = resolvent-upon P1 P2 E1› and ‹P2 = resolvent-upon P2-1 P2-2 E2›

    and ‹P1 = resolvent-upon P1-1 P1-2 E2›
    have ?C = C by auto
  with ‹redundant ?R1 S› and ‹redundant ?R2 S› and ‹partial-saturation S E2
(S ∪ R)›
    and ‹¬ redundant C (S ∪ R)›
    show False using 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 ¬ *tautology P1*
 assumes ¬ *tautology P2*
 assumes *C* = *resolvent-upon P1 P2 E2*
 assumes ¬ *subsumes P1 C*
 assumes ¬ *subsumes P2 C*
 shows (*Neg E2*) ∉ *C* ∧ (*Pos E2*) ∉ *C*
proof
 from ‹*C* = *resolvent-upon P1 P2 E2*› **have** *C* = (*P1* − { *Pos E2* }) ∪ (*P2* − { *Neg E2* })
  by *auto*
 show (*Neg E2*) ∉ *C*
 proof
  assume *Neg E2* ∈ *C*
  from ‹*C* = *resolvent-upon P1 P2 E2*› **have** *C* = (*P1* − { *Pos E2* }) ∪ (*P2* − { *Neg E2* })
   by *auto*
  with ‹*Neg E2* ∈ *C*› **have** *Neg E2* ∈ *P1* **by** *auto*
  from ‹¬ *subsumes P1 C*› **and** ‹*C* = (*P1* − { *Pos E2* }) ∪ (*P2* − { *Neg E2* })› **have** *Pos E2* ∈ *P1*
   unfolding *subsumes-def* **by** *auto*
  from ‹*Neg E2* ∈ *P1*› **and** ‹*Pos E2* ∈ *P1*› **and** ‹¬ *tautology P1*› **show** *False*
   unfolding *tautology-def* **by** *auto*
 qed
 next show (*Pos E2*) ∉ *C*
 proof
  assume *Pos E2* ∈ *C*
  from ‹*C* = *resolvent-upon P1 P2 E2*› **have** *C* = (*P1* − { *Pos E2* }) ∪ (*P2* − { *Neg E2* })
   by *auto*
  with ‹*Pos E2* ∈ *C*› **have** *Pos E2* ∈ *P2* **by** *auto*
  from ‹¬ *subsumes P2 C*› **and** ‹*C* = (*P1* − { *Pos E2* }) ∪ (*P2* − { *Neg E2* })› **have** *Neg E2* ∈ *P2*
   unfolding *subsumes-def* **by** *auto*
  from ‹*Neg E2* ∈ *P2*› **and** ‹*Pos E2* ∈ *P2*› **and** ‹¬ *tautology P2*› **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 E2* (*S* ∪ *R*)
 assumes *all-fulfill* (λ*x*. ¬(*tautology x*)) *S*

assumes *all-fulfill* (*in-all-resolvents-upon S E2*) *R*
assumes *all-fulfill* (λ*x*. (¬*redundant x S*)) *R*
shows *partial-saturation* (*S* ∪ *R*) *E2* (*S* ∪ *R*)
proof (*rule ccontr*)
assume ¬ *partial-saturation* (*S* ∪ *R*) *E2* (*S* ∪ *R*)
from *this* **obtain** *P1 P2 C* **where** *P1* ∈ *S* ∪ *R* **and** *P2* ∈ *S* ∪ *R* **and** *C* = *resolvent-upon P1 P2 E2*
and ¬ *redundant C* (*S* ∪ *R*)
unfolding *partial-saturation-def* **by** *auto*
have *P1* ∈ *S*
proof (*rule ccontr*)
assume *P1* ∉ *S*
with ‹*P1* ∈ *S* ∪ *R*› **have** *P1* ∈ *R* **by** *auto*
with *assms*(*3*) **obtain** *P1-1* **and** *P1-2* **where** *P1-1* ∈ *S* **and** *P1-2* ∈ *S*
and *P1* = *resolvent-upon P1-1 P1-2 E2*
unfolding *all-fulfill-def in-all-resolvents-upon-def* **by** *auto*
from ‹*all-fulfill* (λ*x*. ¬(*tautology x*)) *S*› **and** ‹*P1-1* ∈ *S*› **and** ‹*P1-2* ∈ *S*›
have ¬ *tautology P1-1* **and** ¬ *tautology P1-2*
unfolding *all-fulfill-def* **by** *auto*
from ‹*all-fulfill* (λ*x*. (¬*redundant x S*)) *R*› **and** ‹*P1* ∈ *R*› **and** ‹*P1-1* ∈ *S*› **and** ‹*P1-2* ∈ *S*›
have ¬ *subsumes P1-1 P1* **and** ¬ *subsumes P1-2 P1*
unfolding *redundant-def all-fulfill-def* **by** *auto*
from ‹¬ *tautology P1-1*› ‹¬ *tautology P1-2*› ‹¬ *subsumes P1-1 P1*› **and** ‹¬ *subsumes P1-2 P1*›
and ‹*P1* = *resolvent-upon P1-1 P1-2 E2*›
have (*Neg E2*) ∉ *P1* ∧ (*Pos E2*) ∉ *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 ‹¬ *redundant C* (*S* ∪ *R*)› **and** ‹*P1* ∈ *S* ∪ *R*› **show** *False* **unfolding** *redundant-def*
by *auto*
qed
have *P2* ∈ *S*
proof (*rule ccontr*)
assume *P2* ∉ *S*
with ‹*P2* ∈ *S* ∪ *R*› **have** *P2* ∈ *R* **by** *auto*
with *assms*(*3*) **obtain** *P2-1* **and** *P2-2* **where** *P2-1* ∈ *S* **and** *P2-2* ∈ *S*
and *P2* = *resolvent-upon P2-1 P2-2 E2*
unfolding *all-fulfill-def in-all-resolvents-upon-def* **by** *auto*
from ‹(*all-fulfill* (λ*x*. ¬(*tautology x*)) *S*)› **and** ‹*P2-1* ∈ *S*› **and** ‹*P2-2* ∈ *S*›
have ¬ *tautology P2-1* **and** ¬ *tautology P2-2*
unfolding *all-fulfill-def* **by** *auto*
from ‹*all-fulfill* (λ*x*. (¬*redundant x S*)) *R*› **and** ‹*P2* ∈ *R*› **and** ‹*P2-1* ∈ *S*› **and** ‹*P2-2* ∈ *S*›
have ¬ *subsumes P2-1 P2* **and** ¬ *subsumes P2-2 P2*
unfolding *redundant-def all-fulfill-def* **by** *auto*
from ‹¬ *tautology P2-1*› ‹¬ *tautology P2-2*› ‹¬ *subsumes P2-1 P2*› **and** ‹¬

```
subsumes P2-2 P2›
      and ‹P2 = resolvent-upon P2-1 P2-2 E2›
      have (Neg E2) ∉ P2 ∧ (Pos E2) ∉ P2
      using resolvents-do-not-contain-atom [of P2-1 P2-2 P2 E2] by auto
     with ‹C = resolvent-upon P1 P2 E2› have subsumes P2 C unfolding sub-
sumes-def by auto
    with ‹¬ redundant C (S ∪ R)› and ‹P2 ∈ S ∪ R›
      show False unfolding redundant-def by auto
    qed
   from ‹P1 ∈ S› and ‹P2 ∈ S› and ‹partial-saturation S E2 (S ∪ R)›
    and ‹C = resolvent-upon P1 P2 E2› and ‹¬ redundant C (S ∪ R)›
    show False unfolding redundant-def partial-saturation-def by auto
qed

lemma resolvents-preserve-equivalence:
  shows equivalent S (S ∪ (all-resolvents-upon S A))
proof −
  have S ⊆ (S ∪ (all-resolvents-upon S A)) by auto
   then have entails-formula (S ∪ (all-resolvents-upon S A)) S using entail-
ment-subset by auto
  have entails-formula S (S ∪ (all-resolvents-upon S A))
  proof (rule ccontr)
    assume ¬entails-formula S (S ∪ (all-resolvents-upon S A))
    from this obtain C where C ∈ (all-resolvents-upon S A) and ¬entails S C
      unfolding entails-formula-def using entails-member by auto
    from ‹C ∈ (all-resolvents-upon S A)› obtain P1 P2
      where C = resolvent-upon P1 P2 A and P1 ∈ S and P2 ∈ S
      unfolding all-resolvents-upon-def by auto
   from ‹C = resolvent-upon P1 P2 A› and ‹P1 ∈ S› and ‹P2 ∈ S› have entails
S C
      using resolvent-upon-correct by auto
    with ‹¬entails S C› show False by auto
  qed
  from ‹entails-formula (S ∪ (all-resolvents-upon S A)) S›
    and ‹entails-formula S (S ∪ (all-resolvents-upon S A))›
    show ?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 :: (nat ⇒ 'at) ⇒ 'at Formula ⇒ nat ⇒ 'at Formula
 where
  (resolvents-sequence A S 0) = (simplify S) |
  (resolvents-sequence A S (Suc N)) =
    (simplify ((resolvents-sequence A S N)
      ∪ (all-resolvents-upon (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* ⊆ *S1*
  assumes *all-fulfill* (*λx. redundant x S2*) *S1*
  shows *partial-saturation S2 A S2*
proof (*rule ccontr*)
  assume ¬*partial-saturation S2 A S2*
  then obtain *P1 P2 C* **where** *P1* ∈ *S2 P2* ∈ *S2* **and** *C* = (*resolvent-upon P1 P2 A*)
    and ¬ *redundant C S2*
    unfolding *partial-saturation-def* **by** *auto*
  from ‹*P1* ∈ *S2*› **and** ‹*S2* ⊆ *S1*› **have** *P1* ∈ *S1* **by** *auto*
  from ‹*P2* ∈ *S2*› **and** ‹*S2* ⊆ *S1*› **have** *P2* ∈ *S1* **by** *auto*
  from ‹*P1* ∈ *S1*› **and** ‹*P2* ∈ *S1*› **and** ‹*partial-saturation S1 A S1*› **and** ‹*C* = *resolvent-upon P1 P2 A*›
    have *redundant C S1* **unfolding** *partial-saturation-def* **by** *auto*
  from ‹¬ *redundant C S2*› **have** ¬*tautology C* **unfolding** *redundant-def* **by** *auto*
  with ‹*redundant C S1*› **obtain** *D* **where** *D* ∈ *S1* **and** *D* ⊆ *C*
    unfolding *redundant-def subsumes-def* **by** *auto*
  from ‹*D* ∈ *S1*› **and** ‹*all-fulfill* (*λx. redundant x S2*) *S1*› **have** *redundant D S2*
    unfolding *all-fulfill-def* **by** *auto*
  from ‹¬ *tautology C*› **and** ‹*D* ⊆ *C*› **have** ¬ *tautology D* **unfolding** *tautology-def* **by** *auto*
  with ‹*redundant D S2*› **obtain** *E* **where** *E* ∈ *S2* **and** *E* ⊆ *D*
    unfolding *redundant-def subsumes-def* **by** *auto*
  from ‹*E* ⊆ *D*› **and** ‹*D* ⊆ *C*› **have** *E* ⊆ *C* **by** *auto*
  from ‹*E* ∈ *S2*› **and** ‹*E* ⊆ *C*› **and** ‹¬*redundant C S2*› **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*. ∃ *I*::*nat*. *I* < *N* ∧ *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 *?Final* = *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 *?equiv-init* = *λI*.(*equivalent S* (*resolvents-sequence A S I*))

let *?partial-saturation* = λI. ($\forall$ *J*::*nat*. (*J* < *I*
$\longrightarrow$ (*partial-saturation* (*resolvents-sequence* *A* *S* *I*) (*A* *J*) (*resolvents-sequence* *A* *S* *I*))))
let *?no-tautologies* = λI.(*all-fulfill* (λx. $\neg$(*tautology* *x*)) (*resolvents-sequence* *A* *S* *I*))
let *?atoms-init* = λI.(*atoms-formula* (*resolvents-sequence* *A* *S* *I*)
$\subseteq$ { *X*. $\exists$ *I*::*nat*. *I* < *N* $\wedge$ *X* = (*A* *I*)})
let *?non-redundant* = λI.(*non-redundant* (*resolvents-sequence* *A* *S* *I*))
let *?finite* =λI. (*all-fulfill finite* (*resolvents-sequence* *A* *S* *I*))

have $\forall$ *I*. (*I* $\leq$ *N* $\longrightarrow$ (*?equiv-init* *I*) $\wedge$ (*?partial-saturation* *I*) $\wedge$ (*?no-tautologies* *I*)
$\wedge$ (*?atoms-init* *I*) $\wedge$ (*?non-redundant* *I*) $\wedge$ (*?finite* *I*))

proof (*rule allI*)
fix *I*
show (*I* $\leq$ *N*
$\longrightarrow$ (*?equiv-init* *I*) $\wedge$ (*?partial-saturation* *I*) $\wedge$ (*?no-tautologies* *I*) $\wedge$ (*?atoms-init* *I*)
$\wedge$ (*?non-redundant* *I*) $\wedge$ (*?finite* *I*)) (**is** *I* $\leq$ *N* $\longrightarrow$ *?P* *I*)
proof (*induction* *I*)

We show that the properties are all satisfied by the initial clause set (after simplification).

show *0* $\leq$ *N* $\longrightarrow$ *?P* *0*
proof (*rule impI*)+
assume *0* $\leq$ *N*
let *?R* = *resolvents-sequence* *A* *S* *0*
from ‹*all-fulfill finite* *S*›
have *?equiv-init* *0* **using** *simplify-preserves-equivalence* **by** *auto*
moreover have *?no-tautologies* *0*
using *simplify-def strictly-redundant-def all-fulfill-def* **by** *auto*
moreover have *?partial-saturation* *0* **by** *auto*
moreover from ‹*all-fulfill finite* *S*› **have** *?finite* *0* **using** *simplify-finite* **by** *auto*
moreover have *atoms-formula* *?R* $\subseteq$ *atoms-formula* *S* **using** *atoms-formula-simplify* **by** *auto*
moreover with ‹*atoms-formula* *S* = { *X*. $\exists$ *I*::*nat*. *I* < *N* $\wedge$ *X* = (*A* *I*) }›
have *v*: *?atoms-init* *0* **unfolding** *simplify-def* **by** *auto*
moreover have *?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 (*Suc* *I*) $\leq$ *N* $\longrightarrow$ (*?P* (*Suc* *I*))
proof (*rule impI*)+

assume (*Suc I*) ≤ *N*
let *?Prec* = *resolvents-sequence A S I*
let *?R* = *resolvents-sequence A S* (*Suc I*)
from ‹*Suc I* ≤ *N*› **and** ‹*I* ≤ *N* ⟶ *?P I*›
have *?equiv-init I* **and** *?partial-saturation I* **and** *?no-tautologies I* **and** *?finite I*
and *?atoms-init I* **and** *?non-redundant I* **by** *auto*
have *equivalent ?Prec* (*?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*)))
using *resolvents-preserve-equivalence* **by** *auto*
from ‹*?finite I*› **have** *all-fulfill finite* (*?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*)))
using *all-resolvents-upon-is-finite* **by** *auto*
then have *all-fulfill finite* (*simplify* (*?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*))))
using *simplify-finite* **by** *auto*
then have *?finite* (*Suc I*) **by** *auto*
from ‹*all-fulfill finite* (*?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*)))›
have *equivalent* (*?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*))) *?R*
using *simplify-preserves-equivalence* **by** *auto*
from ‹*equivalent ?Prec* (*?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*)))›
and ‹*equivalent* (*?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*))) *?R*›
have *equivalent ?Prec ?R* **by** (*rule equivalent-transitive*)
from ‹*?equiv-init I*› **and** *this* **have** *?equiv-init* (*Suc I*) **by** (*rule equivalent-transitive*)
have *?no-tautologies* (*Suc I*) **using** *simplify-def strictly-redundant-def all-fulfill-def*
by *auto*
let *?Delta* = *?R* − *?Prec*
have *?R* ⊆ *?Prec* ∪ *?Delta* **by** *auto*
have *all-fulfill* (λ*x*. (*redundant x ?R*)) (*?Prec* ∪ *?Delta*)
proof (*rule ccontr*)
assume ¬*all-fulfill* (λ*x*. (*redundant x ?R*)) (*?Prec* ∪ *?Delta*)
then obtain *x* **where** ¬*redundant x ?R* **and** *x* ∈ *?Prec* ∪ *?Delta* **unfolding** *all-fulfill-def*
by *auto*
from ‹¬*redundant x ?R*› **have** ¬*x* ∈ *?R* **unfolding** *redundant-def subsumes-def* **by** *auto*
with ‹*x* ∈ *?Prec* ∪ *?Delta*› **have** *x* ∈ (*?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*)))
by *auto*
with ‹*all-fulfill finite* (*?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*)))›
have *redundant x* (*simplify* (*?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*))))
using *simplify-and-membership* **by** *blast*
with ‹¬*redundant x ?R*› **show** *False* **by** *auto*
qed
have *all-fulfill* (*in-all-resolvents-upon ?Prec* (*A I*)) *?Delta*
proof (*rule ccontr*)
assume ¬ (*all-fulfill* (*in-all-resolvents-upon ?Prec* (*A I*)) *?Delta*)
then obtain *C* **where** *C* ∈ *?Delta*

and ¬*in-all-resolvents-upon ?Prec* (*A I*) *C*
unfolding *all-fulfill-def* **by** *auto*
then obtain *C* **where** *C* ∈ *?Delta*
and *not-res*: ∀ *P1 P2*. ¬(*P1* ∈ *?Prec* ∧ *P2* ∈ *?Prec* ∧ *C* = *resolvent-upon P1 P2* (*A I*))
unfolding *all-fulfill-def in-all-resolvents-upon-def* **by** *blast*
from ‹*C* ∈ *?Delta*› **have** *C* ∈ *?R* **and** *C* ∉ *?Prec* **by** *auto*
then have *C* ∈ *simplify* (*?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*))) **by** *auto*
then have *C* ∈ *?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*)) **unfolding** *simplify-def* **by** *auto*
with ‹*C* ∉ *?Prec*› **have** *C* ∈ (*all-resolvents-upon ?Prec* (*A I*)) **by** *auto*
with *not-res* **show** *False* **unfolding** *all-resolvents-upon-def* **by** *auto*
qed
have *all-fulfill* (λ*x*. (¬*redundant x ?Prec*)) *?Delta*
proof (*rule ccontr*)
assume ¬*all-fulfill* (λ*x*. (¬*redundant x ?Prec*)) *?Delta*
then obtain *C* **where** *C* ∈ *?Delta* **and** *redundant*: *redundant C ?Prec*
unfolding *all-fulfill-def* **by** *auto*
from ‹*C* ∈ *?Delta*› **have** *C* ∈ *?R* **and** *C* ∉ *?Prec* **by** *auto*
show *False*
proof *cases*
assume *strictly-redundant C ?Prec*
then have *strictly-redundant C* (*?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*)))
unfolding *strictly-redundant-def* **by** *auto*
then have *C* ∉ *simplify* (*?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*)))
unfolding *simplify-def* **by** *auto*
then have *C* ∉ *?R* **by** *auto*
with ‹*C* ∈ *?R*› **show** *False* **by** *auto*
next assume ¬*strictly-redundant C ?Prec*
with *redundant* **have** *C* ∈ *?Prec*
unfolding *strictly-redundant-def redundant-def subsumes-def* **by** *auto*
with ‹*C* ∉ *?Prec*› **show** *False* **by** *auto*
qed
qed
have ∀ *J*::*nat*. (*J* < (*Suc I*)) ⟶ (*partial-saturation ?R* (*A J*) *?R*)
proof (*rule ccontr*)
assume ¬(∀ *J*::*nat*. (*J* < (*Suc I*)) ⟶ (*partial-saturation ?R* (*A J*) *?R*))
then obtain *J* **where** *J* < (*Suc I*) **and** ¬(*partial-saturation ?R* (*A J*) *?R*) **by** *auto*
from ‹¬(*partial-saturation ?R* (*A J*) *?R*)› **obtain** *P1 P2 C*
where *P1* ∈ *?R* **and** *P2* ∈ *?R* **and** *C* = *resolvent-upon P1 P2* (*A J*) **and** ¬ *redundant C ?R*
unfolding *partial-saturation-def* **by** *auto*
have *partial-saturation ?Prec* (*A I*) (*?Prec* ∪ *?Delta*)
proof (*rule ccontr*)
assume ¬*partial-saturation ?Prec* (*A I*) (*?Prec* ∪ *?Delta*)
then obtain *P1 P2 C* **where** *P1* ∈ *?Prec* **and** *P2* ∈ *?Prec*

and *C* = *resolvent-upon P1 P2* (*A I*) **and**
¬*redundant C* (*?Prec* ∪ *?Delta*) **unfolding** *partial-saturation-def* **by** *auto*
from ‹*C* = *resolvent-upon P1 P2* (*A I*)› **and** ‹*P1* ∈ *?Prec*› **and** ‹*P2* ∈ *?Prec*›
have *C* ∈ *?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*))
unfolding *all-resolvents-upon-def* **by** *auto*
from ‹*all-fulfill finite* (*?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*)))›
and *this* **have** *redundant C ?R*
using *simplify-and-membership* [*of ?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*)) *?R C*]
by *auto*
with ‹*?R* ⊆ *?Prec* ∪ *?Delta*› **have** *redundant C* (*?Prec* ∪ *?Delta*)
using *superset-preserves-redundancy* [*of C ?R* (*?Prec* ∪ *?Delta*)] **by** *auto*
with ‹¬*redundant C* (*?Prec* ∪ *?Delta*)› **show** *False* **by** *auto*
qed
show *False*
proof *cases*
assume *J* = *I*
from ‹*partial-saturation ?Prec* (*A I*) (*?Prec* ∪ *?Delta*)› **and** ‹*?no-tautologies I*›
and ‹(*all-fulfill* (*in-all-resolvents-upon ?Prec* (*A I*)) *?Delta*)›
and ‹*all-fulfill* (λ*x*. (¬*redundant x ?Prec*)) *?Delta*›
have *partial-saturation* (*?Prec* ∪ *?Delta*) (*A I*) (*?Prec* ∪ *?Delta*)
using *ensures-partial-saturation* [*of ?Prec* (*A I*) *?Delta*] **by** *auto*
with ‹*?R* ⊆ *?Prec* ∪ *?Delta*›
and ‹*all-fulfill* (λ*x*. (*redundant x ?R*)) (*?Prec* ∪ *?Delta*)›
have *partial-saturation ?R* (*A I*) *?R* **using** *redundancy-implies-partial-saturation*
by *auto*
with ‹*J* = *I*› **and** ‹¬(*partial-saturation ?R* (*A J*) *?R*)› **show** *False* **by** *auto*
next
assume *J* ≠ *I*
with ‹*J* < (*Suc I*)› **have** *J* < *I* **by** *auto*
with ‹*?partial-saturation I*›
have *partial-saturation ?Prec* (*A J*) *?Prec* **by** *auto*
with ‹*partial-saturation ?Prec* (*A I*) (*?Prec* ∪ *?Delta*)› **and** ‹*?no-tautologies I*›
and ‹(*all-fulfill* (*in-all-resolvents-upon ?Prec* (*A I*)) *?Delta*)›
and ‹*all-fulfill* (λ*x*. (¬*redundant x ?Prec*)) *?Delta*›
have *partial-saturation* (*?Prec* ∪ *?Delta*) (*A J*) (*?Prec* ∪ *?Delta*)
using *partial-saturation-is-preserved* [*of ?Prec A J A I ?Delta*] **by** *auto*
with ‹*?R* ⊆ *?Prec* ∪ *?Delta*›
and ‹*all-fulfill* (λ*x*. (*redundant x ?R*)) (*?Prec* ∪ *?Delta*)›
have *partial-saturation ?R* (*A J*) *?R*
using *redundancy-implies-partial-saturation* **by** *auto*
with ‹¬(*partial-saturation ?R* (*A J*) *?R*)› **show** *False* **by** *auto*
qed

qed
have *non-redundant ?R* **using** *simplify-non-redundant* **by** *auto*
from ‹*?atoms-init I*› **have** *atoms-formula* (*all-resolvents-upon ?Prec* (*A I*))
⊆ { *X*. ∃ *I*::*nat*. *I* < *N* ∧ *X* = (*A I*)}
using *atoms-formula-resolvents* [*of ?Prec A I*] **by** *auto*
with ‹*?atoms-init I*›
have *atoms-formula* (*?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*)))
⊆ { *X*. ∃ *I*::*nat*. *I* < *N* ∧ *X* = (*A I*)}
using *atoms-formula-union* [*of ?Prec all-resolvents-upon ?Prec* (*A I*)] **by** *auto*
from *this* **have** *atoms-formula ?R* ⊆ { *X*. ∃ *I*::*nat*. *I* < *N* ∧ *X* = (*A I*)}
using *atoms-formula-simplify* [*of ?Prec* ∪ (*all-resolvents-upon ?Prec* (*A I*))] **by** *auto*
from ‹*equivalent S* (*resolvents-sequence A S* (*Suc I*))›
and ‹(∀ *J*::*nat*. (*J* < (*Suc I*)
⟶ (*partial-saturation* (*resolvents-sequence A S* (*Suc I*)) (*A J*)
(*resolvents-sequence A S* (*Suc I*)))))›
and ‹(*all-fulfill* (λ*x*. ¬(*tautology x*)) (*resolvents-sequence A S* (*Suc I*)))›
and ‹(*all-fulfill finite* (*resolvents-sequence A S* (*Suc I*)))›
and ‹*non-redundant ?R*›
and ‹*atoms-formula* (*resolvents-sequence A S* (*Suc I*)) ⊆ { *X*. ∃ *I*::*nat*. *I* < *N* ∧ *X* = (*A I*)}›
show *?P* (*Suc I*) **by** *auto*
qed
qed
qed

Using the above invariants, we show that the final clause set is saturated.

from *this* **have** ∀ *J*. (*J* < *N* ⟶ *partial-saturation ?Final* (*A J*) *?Final*)
and *atoms-formula* (*resolvents-sequence A S N*) ⊆ { *X*. ∃ *I*::*nat*. *I* < *N* ∧ *X* = (*A I*)}
and *equivalent S ?Final*
and *non-redundant ?Final*
and *all-fulfill finite ?Final*
by *auto*
have *saturated-binary-rule resolvent ?Final*
proof (*rule ccontr*)
assume ¬ *saturated-binary-rule resolvent ?Final*
then obtain *P1 P2 C* **where** *P1* ∈ *?Final* **and** *P2* ∈ *?Final* **and** *resolvent P1 P2 C*
and ¬*redundant C ?Final*
unfolding *saturated-binary-rule-def* **by** *auto*
from ‹*resolvent P1 P2 C*› **obtain** *B* **where** *C* = *resolvent-upon P1 P2 B*
unfolding *resolvent-def* **by** *auto*
show *False*
proof *cases*
assume *B* ∈ (*atoms-formula ?Final*)
with ‹*atoms-formula ?Final* ⊆ { *X*. ∃ *I*::*nat*. *I* < *N* ∧ *X* = (*A I*) }›
obtain *I* **where** *B* = (*A I*) **and** *I* < *N*

by *auto*
from ‹$B = (A\ I)$› **and** ‹C = *resolvent-upon P1 P2 B*› **have** C = *resolvent-upon P1 P2* $(A\ I)$
by *auto*
from ‹$\forall J.\ (J < N \longrightarrow$ *partial-saturation ?Final* $(A\ J)$ *?Final*)› **and** ‹$B = (A\ I)$›**and** ‹$I < N$›
have *partial-saturation ?Final* $(A\ I)$ *?Final* **by** *auto*
with ‹C = *resolvent-upon P1 P2* $(A\ I)$›**and** ‹*P1* $\in$ *?Final*› **and** ‹*P2* $\in$ *?Final*›
have *redundant C ?Final* **unfolding** *partial-saturation-def* **by** *auto*
with ‹$\neg$*redundant C ?Final*› **show** *False* **by** *auto*
next
assume $B \notin$ *atoms-formula ?Final*
with ‹*P1* $\in$ *?Final*› **have** $B \notin$ *atoms-clause P1* **by** *auto*
then have *Pos B* $\notin$ *P1* **by** *auto*
with ‹C = *resolvent-upon P1 P2 B*› **have** *P1* $\subseteq C$ **by** *auto*
with ‹*P1* $\in$ *?Final*› **and** ‹$\neg$*redundant C ?Final*› **show** *False*
unfolding *redundant-def subsumes-def* **by** *auto*
qed
qed
with ‹*all-fulfill finite ?Final*› **and** ‹*non-redundant ?Final*›
have *prime-implicates ?Final* = *?Final*
using *prime-implicates-of-saturated-sets* [*of ?Final*] **by** *auto*
with ‹*equivalent S ?Final*› **show** *?thesis* **using** *equivalence-and-prime-implicates* **by** *auto*
qed

end
end