Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

April 20, 2016

Contents

theory *Prop-Resolution*
imports *Partial-Clausal-Logic List-More Wellfounded-More*

begin

Chapter 1

Resolution-based techniques

This chapter contains the formalisation of resolution and superposition.

1.1 Resolution

1.1.1 Simplification Rules

```
inductive simplify :: 'v clauses ⇒ 'v clauses ⇒ bool for N :: 'v clause set where
tautology-deletion:
  A + {#Pos P#} + {#Neg P#} ∈ N ⟹ simplify  N (N − {A + {#Pos P#} + {#Neg P#}})|
condensation:
  A + {#L#} + {#L#} ∈ N ⟹ simplify N (N − {A + {#L#} + {#L#}} ∪ {A + {#L#}}) |
subsumption:
  A ∈ N ⟹ A ⊂# B ⟹ B ∈ N ⟹ simplify N (N − {B})

lemma simplify-preserves-un-sat':
  fixes N N' :: 'v clauses
  assumes simplify N N'
  and total-over-m I N
  shows I ⊨s N' ⟶ I ⊨s N
  using assms
proof (induct rule: simplify.induct)
  case (tautology-deletion A P)
  then have I ⊨ A + {#Pos P#} + {#Neg P#}
    by (metis total-over-m-def total-over-set-literal-defined true-cls-singleton true-cls-union
      true-lit-def uminus-Neg union-commute)
  then show ?case by (metis Un-Diff-cancel2 true-clss-singleton true-clss-union)
next
  case (condensation A P)
  then show ?case by (metis Diff-insert-absorb Set.set-insert insertE true-cls-union true-clss-def
    true-clss-singleton true-clss-union)
next
  case (subsumption A B)
  have A ≠ B using subsumption.hyps(2) by auto
  then have I ⊨s N − {B} ⟹ I ⊨ A using ⟨A ∈ N⟩ by (simp add: true-clss-def)
  moreover have I ⊨ A ⟹ I ⊨ B using ⟨A <# B⟩ by auto
  ultimately show ?case by (metis insert-Diff-single true-clss-insert)
qed

lemma simplify-preserves-un-sat:
  fixes N N' :: 'v clauses
  assumes simplify N N'
  and total-over-m I N
  shows I ⊨s N ⟶ I ⊨s N'
  using assms apply (induct rule: simplify.induct)
  using true-clss-def by fastforce+

lemma simplify-preserves-un-sat'':
  fixes N N' :: 'v clauses
  assumes simplify N N'
  and total-over-m I N'
  shows I ⊨s N ⟶ I ⊨s N'
  using assms apply (induct rule: simplify.induct)
  using true-clss-def by fastforce+

lemma simplify-preserves-un-sat-eq:
  fixes N N' :: 'v clauses
  assumes simplify N N'
  and total-over-m I N
  shows I ⊨s N ⟷ I ⊨s N'
  using simplify-preserves-un-sat simplify-preserves-un-sat' assms by blast

lemma simplify-preserves-finite:
 assumes simplify ψ ψ'
 shows finite ψ ⟷ finite ψ'
 using assms by (induct rule: simplify.induct, auto simp add: remove-def)

lemma rtranclp-simplify-preserves-finite:
 assumes rtranclp simplify ψ ψ'
 shows finite ψ ⟷ finite ψ'
 using assms by (induct rule: rtranclp-induct) (auto simp add: simplify-preserves-finite)

lemma simplify-atms-of-ms:
  assumes simplify ψ ψ'
  shows atms-of-ms ψ' ⊆ atms-of-ms ψ
  using assms unfolding atms-of-ms-def
proof (induct rule: simplify.induct)
  case (tautology-deletion A P)
  then show ?case by auto
next
  case (condensation A P)
  moreover have A + {#P#} + {#P#} ∈ ψ ⟹ ∃ x∈ψ. atm-of P ∈ atm-of ‘ set-mset x
    by (metis Un-iff atms-of-def atms-of-plus atms-of-singleton insert-iff)
  ultimately show ?case by (auto simp add: atms-of-def)
next
  case (subsumption A P)
  then show ?case by auto
qed

lemma rtranclp-simplify-atms-of-ms:
  assumes rtranclp simplify ψ ψ'
  shows atms-of-ms ψ' ⊆ atms-of-ms ψ
  using assms apply (induct rule: rtranclp-induct)
   apply (fastforce intro: simplify-atms-of-ms)
  using simplify-atms-of-ms by blast

lemma factoring-imp-simplify:
  assumes {#L#} + {#L#} + C ∈ N
  shows ∃N'. simplify N N'
proof −
  have C + {#L#} + {#L#} ∈ N using assms by (simp add: add.commute union-lcomm)
  from condensation[OF this] show ?thesis by blast
qed
```

1.1.2 Unconstrained Resolution

```
type-synonym 'v uncon-state = 'v clauses
inductive uncon-res :: 'v uncon-state ⇒ 'v uncon-state ⇒ bool where
resolution:
  {#Pos p#} + C ∈ N ⟹ {#Neg p#} + D ∈ N ⟹ ({#Pos p#} + C, {#Neg p#} + D) ∉
already-used
   ⟹ uncon-res (N) (N ∪ {C + D}) |
factoring: {#L#} + {#L#} + C ∈ N ⟹ uncon-res N (N ∪{C + {#L#}})

lemma uncon-res-increasing:
  assumes uncon-res S S' and ψ ∈ S
  shows ψ ∈ S'
  using assms by (induct rule: uncon-res.induct) auto

lemma rtranclp-uncon-inference-increasing:
  assumes rtranclp uncon-res S S' and ψ ∈ S
  shows ψ ∈ S'
  using assms by (induct rule: rtranclp-induct) (auto simp add: uncon-res-increasing)
```

Subsumption

```
definition subsumes :: 'a literal multiset ⇒ 'a literal multiset ⇒ bool where
subsumes χ χ' ⟷
  (∀I. total-over-m I {χ'} ⟶ total-over-m I {χ})
  ∧ (∀I. total-over-m I {χ} ⟶ I ⊨ χ ⟶ I ⊨ χ')

lemma subsumes-refl[simp]:
  subsumes χ χ
  unfolding subsumes-def by auto


lemma subsumes-subsumption:
  assumes subsumes D χ
  and C ⊂# D and ¬tautology χ
  shows subsumes C χ unfolding subsumes-def
  using assms subsumption-total-over-m subsumption-chained unfolding subsumes-def
  by (blast intro!: subset-mset.less-imp-le)

lemma subsumes-tautology:
  assumes subsumes (C + {#Pos P#} + {#Neg P#}) χ
  shows tautology χ
  using assms unfolding subsumes-def by (simp add: tautology-def)
```

1.1.3 Inference Rule

```
type-synonym 'v state = 'v clauses × ('v clause × 'v clause) set
inductive inference-clause :: 'v state ⇒ 'v clause × ('v clause × 'v clause) set ⇒ bool
  (infix ⇒Res 100) where
```

```
resolution:
  {#Pos p#} + C ∈ N ⟹ {#Neg p#} + D ∈ N ⟹ ({#Pos p#} + C, {#Neg p#} + D) ∉
already-used
  ⟹ inference-clause (N, already-used) (C + D, already-used ∪ {({#Pos p#} + C, {#Neg p#} +
D)}) |
factoring: {#L#} + {#L#} + C ∈ N ⟹ inference-clause (N, already-used) (C + {#L#}, already-used)

inductive inference :: 'v state ⇒ 'v state ⇒ bool where
inference-step: inference-clause S (clause, already-used)
  ⟹ inference S (fst S ∪ {clause}, already-used)


abbreviation already-used-inv
  :: 'a literal multiset set × ('a literal multiset × 'a literal multiset) set ⇒ bool where
already-used-inv state ≡
  (∀ (A, B) ∈ snd state. ∃ p. Pos p ∈# A ∧ Neg p ∈# B ∧
         ((∃ χ ∈ fst state. subsumes χ ((A − {#Pos p#}) + (B − {#Neg p#})))
           ∨ tautology ((A − {#Pos p#}) + (B − {#Neg p#}))))

lemma inference-clause-preserves-already-used-inv:
  assumes inference-clause S S'
  and already-used-inv S
  shows already-used-inv (fst S ∪ {fst S'}, snd S')
  using assms apply (induct rule: inference-clause.induct)
  by fastforce+

lemma inference-preserves-already-used-inv:
  assumes inference S S'
  and already-used-inv S
  shows already-used-inv S'
  using assms
proof (induct rule: inference.induct)
  case (inference-step S clause already-used)
  then show ?case
    using inference-clause-preserves-already-used-inv[of S (clause, already-used)] by simp
qed

lemma rtranclp-inference-preserves-already-used-inv:
  assumes rtranclp inference S S'
  and already-used-inv S
  shows already-used-inv S'
  using assms apply (induct rule: rtranclp-induct, simp)
  using inference-preserves-already-used-inv unfolding tautology-def by fast

lemma subsumes-condensation:
  assumes subsumes (C + {#L#} + {#L#}) D
  shows subsumes (C + {#L#}) D
  using assms unfolding subsumes-def by simp

lemma simplify-preserves-already-used-inv:
  assumes simplify N N'
  and already-used-inv (N, already-used)
  shows already-used-inv (N', already-used)
  using assms
proof (induct rule: simplify.induct)
  case (condensation C L)
  then show ?case
    using subsumes-condensation by simp fast
next
  {
     fix a:: 'a and A :: 'a set and P
     have (∃x ∈ Set.remove a A. P x) ⟷ (∃x ∈ A. x ≠ a ∧ P x) by auto
  } note ex-member-remove = this
  {
    fix a a0 :: 'v clause and A :: 'v clauses and  y
    assume a ∈ A and a0 ⊂# a
    then have (∃x ∈ A. subsumes x y) ⟷ (subsumes a y  ∨ (∃x ∈ A. x ≠ a ∧ subsumes x y))
      by auto
  } note tt2 = this
  case (subsumption A B) note A = this(1) and AB = this(2) and B = this(3) and inv = this(4)
  show ?case
    proof (standard, standard)
      fix x a b
      assume x: x ∈ snd (N − {B}, already-used) and [simp]: x = (a, b)
      obtain p where p: Pos p ∈# a ∧ Neg p ∈# b and
        q: (∃χ∈N. subsumes χ (a − {#Pos p#} + (b − {#Neg p#})))
          ∨ tautology (a − {#Pos p#} + (b − {#Neg p#}))
        using inv x by fastforce
      consider (taut) tautology (a − {#Pos p#} + (b − {#Neg p#})) |
        (χ) χ where χ ∈ N subsumes χ (a − {#Pos p#} + (b − {#Neg p#}))
          ¬tautology (a − {#Pos p#} + (b − {#Neg p#}))
        using q by auto
      then show
        ∃p. Pos p ∈# a ∧ Neg p ∈# b
            ∧ ((∃χ∈fst (N − {B}, already-used). subsumes χ (a − {#Pos p#} + (b − {#Neg p#})))
                ∨ tautology (a − {#Pos p#} + (b − {#Neg p#})))
        proof cases
          case taut
          then show ?thesis using p by auto
        next
          case χ note H = this
          show ?thesis using p A AB B  subsumes-subsumption[OF - AB H(3)] H(1,2) by auto
        qed
    qed
next
  case (tautology-deletion C P)
  then show ?case apply clarify
  proof −
    fix a b
    assume C + {#Pos P#} + {#Neg P#} ∈ N
    assume already-used-inv (N, already-used)
    and (a, b) ∈ snd (N − {C + {#Pos P#} + {#Neg P#}}, already-used)
    then obtain p where
      Pos p ∈# a ∧ Neg p ∈# b ∧
        ((∃χ∈fst (N ∪ {C + {#Pos P#} + {#Neg P#}}, already-used).
              subsumes χ (a − {#Pos p#} + (b − {#Neg p#})))
          ∨ tautology (a − {#Pos p#} + (b − {#Neg p#})))
      by fastforce
    moreover have tautology (C + {#Pos P#} + {#Neg P#}) by auto
    ultimately show
      ∃p. Pos p ∈# a ∧ Neg p ∈# b
      ∧ ((∃χ∈fst (N − {C + {#Pos P#} + {#Neg P#}}, already-used).
        subsumes χ (a − {#Pos p#} + (b − {#Neg p#})))
      ∨ tautology (a − {#Pos p#} + (b − {#Neg p#})))
    by (metis (no-types) Diff-iff Un-insert-right empty-iff fst-conv insertE subsumes-tautology
      sup-bot.right-neutral)
  qed
qed


lemma
  factoring-satisfiable: I ⊨ {#L#} + {#L#} + C ⟷ I ⊨ {#L#} + C and
  resolution-satisfiable:
    consistent-interp I ⟹ I ⊨ {#Pos p#} + C ⟹ I ⊨ {#Neg p#} + D ⟹ I ⊨ C + D and
    factoring-same-vars: atms-of ({#L#} + {#L#} + C) = atms-of ({#L#} + C)
  unfolding true-cls-def consistent-interp-def by (fastforce split: if-split-asm)+

lemma inference-increasing:
  assumes inference S S′ and ψ ∈ fst S
  shows ψ ∈ fst S′
  using assms by (induct rule: inference.induct, auto)

lemma rtranclp-inference-increasing:
  assumes rtranclp inference S S′ and ψ ∈ fst S
  shows ψ ∈ fst S′
  using assms by (induct rule: rtranclp-induct, auto simp add: inference-increasing)

lemma inference-clause-already-used-increasing:
  assumes inference-clause S S′
  shows snd S ⊆ snd S′
  using assms by (induct rule:inference-clause.induct, auto)


lemma inference-already-used-increasing:
  assumes inference S S′
  shows snd S ⊆ snd S′
  using assms apply (induct rule:inference.induct)
  using inference-clause-already-used-increasing by fastforce

lemma inference-clause-preserves-un-sat:
  fixes N N′ :: ′v clauses
  assumes inference-clause T T′
  and  total-over-m I (fst T)
  and consistent: consistent-interp I
  shows I ⊨s fst T ⟷ I ⊨s fst T ∪ {fst T′}
  using assms apply (induct rule: inference-clause.induct)
  unfolding consistent-interp-def true-clss-def by auto force+


lemma inference-preserves-un-sat:
  fixes N N′ :: ′v clauses
  assumes inference T T′
  and  total-over-m I (fst T)
  and consistent: consistent-interp I
  shows I ⊨s fst T ⟷ I ⊨s fst T′
  using assms apply (induct rule: inference.induct)
  using inference-clause-preserves-un-sat by fastforce
```

```
lemma inference-clause-preserves-atms-of-ms:
  assumes inference-clause S S'
  shows atms-of-ms (fst (fst S ∪ {fst S'}, snd S')) ⊆ atms-of-ms (fst S)
  using assms apply (induct rule: inference-clause.induct)
   apply auto
     apply (metis Set.set-insert UnCI atms-of-ms-insert atms-of-plus)
    apply (metis Set.set-insert UnCI atms-of-ms-insert atms-of-plus)
   apply (simp add: in-m-in-literals union-assoc)
  unfolding atms-of-ms-def using assms by fastforce

lemma inference-preserves-atms-of-ms:
  fixes N N' :: 'v clauses
  assumes inference T T'
  shows atms-of-ms (fst T') ⊆ atms-of-ms (fst T)
  using assms apply (induct rule: inference.induct)
  using inference-clause-preserves-atms-of-ms by fastforce

lemma inference-preserves-total:
  fixes N N' :: 'v clauses
  assumes inference (N, already-used) (N', already-used')
  shows total-over-m I N ⟹ total-over-m I N'
    using assms inference-preserves-atms-of-ms unfolding total-over-m-def total-over-set-def
    by fastforce


lemma rtranclp-inference-preserves-total:
  assumes rtranclp inference T T'
  shows total-over-m I (fst T) ⟹ total-over-m I (fst T')
  using assms by (induct rule: rtranclp-induct, auto simp add: inference-preserves-total)

lemma rtranclp-inference-preserves-un-sat:
  assumes rtranclp inference N N'
  and  total-over-m I (fst N)
  and consistent: consistent-interp I
  shows I ⊨s fst N ⟷ I ⊨s fst N'
  using assms apply (induct rule: rtranclp-induct)
  apply (simp add: inference-preserves-un-sat)
  using inference-preserves-un-sat rtranclp-inference-preserves-total by blast

lemma inference-preserves-finite:
  assumes inference ψ ψ' and finite (fst ψ)
  shows finite (fst ψ')
  using assms by (induct rule: inference.induct, auto simp add: simplify-preserves-finite)


lemma inference-clause-preserves-finite-snd:
  assumes inference-clause ψ ψ' and finite (snd ψ)
  shows finite (snd ψ')
  using assms by (induct rule: inference-clause.induct, auto)


lemma inference-preserves-finite-snd:
  assumes inference ψ ψ' and finite (snd ψ)
  shows finite (snd ψ')
  using assms inference-clause-preserves-finite-snd by (induct rule: inference.induct, fastforce)
```

```
lemma rtranclp-inference-preserves-finite:
  assumes rtranclp inference ψ ψ' and finite (fst ψ)
  shows finite (fst ψ')
  using assms by (induct rule: rtranclp-induct)
    (auto simp add: simplify-preserves-finite inference-preserves-finite)

lemma consistent-interp-insert:
  assumes consistent-interp I
  and atm-of P ∉ atm-of ‘ I
  shows consistent-interp (insert P I)
proof –
  have P: insert P I = I ∪ {P} by auto
  show ?thesis unfolding P
  apply (rule consistent-interp-disjoint)
  using assms by (auto simp: image-iff)
qed

lemma simplify-clause-preserves-sat:
  assumes simp: simplify ψ ψ'
  and satisfiable ψ'
  shows satisfiable ψ
  using assms
proof induction
  case (tautology-deletion A P) note AP = this(1) and sat = this(2)
  let ?A' = A + {#Pos P#} + {#Neg P#}
  let ?ψ' = ψ – {?A'}
  obtain I where
    I: I ⊨s ?ψ' and
    cons: consistent-interp I  and
    tot: total-over-m I ?ψ'
    using sat unfolding satisfiable-def by auto
  { assume Pos P ∈ I ∨ Neg P ∈ I
    then have I ⊨ ?A' by auto
    then have I ⊨s ψ using I by (metis insert-Diff tautology-deletion.hyps true-clss-insert)
    then have ?case using cons tot by auto
  }
  moreover {
    assume Pos: Pos P ∉ I and Neg: Neg P ∉ I
    then have consistent-interp (I ∪ {Pos P}) using cons by simp
    moreover have I'A: I ∪ {Pos P} ⊨ ?A' by auto
    have {Pos P} ∪ I ⊨s ψ – {A + {#Pos P#} + {#Neg P#}}
      using ‹I ⊨s ψ – {A + {#Pos P#} + {#Neg P#}}› true-clss-union-increase' by blast
    then have I ∪ {Pos P} ⊨s ψ
      by (metis (no-types) Un-empty-right Un-insert-left Un-insert-right I'A insert-Diff
        sup-bot.left-neutral tautology-deletion.hyps true-clss-insert)
    ultimately have ?case using satisfiable-carac' by blast
  }
  ultimately show ?case by blast
next
  case (condensation A L) note AL = this(1) and sat = this(2)
  have f3: simplify ψ (ψ – {A + {#L#} + {#L#}} ∪ {A + {#L#}})
    using AL simplify.condensation by blast
  obtain LL :: 'a literal multiset set ⇒ 'a literal set where
    f4: LL (ψ – {A + {#L#} + {#L#}} ∪ {A + {#L#}}) ⊨s ψ – {A + {#L#} + {#L#}} ∪ {A
+ {#L#}}
```

```
      ∧ consistent-interp (LL (ψ − {A + {#L#} + {#L#}} ∪ {A + {#L#}}))
      ∧ total-over-m (LL (ψ − {A + {#L#} + {#L#}}
                    ∪ {A + {#L#}})) (ψ − {A + {#L#} + {#L#}} ∪ {A + {#L#}})
    using sat by (meson satisfiable-def)
  have f5: insert (A + {#L#} + {#L#}) (ψ − {A + {#L#} + {#L#}}) = ψ
    using AL by fastforce
  have atms-of (A + {#L#} + {#L#}) = atms-of ({#L#} + A)
    by simp
  then show ?case
    using f5 f4 f3 by (metis (no-types) add.commute satisfiable-def simplify-preserves-un-sat′
      total-over-m-insert total-over-m-union)
next
  case (subsumption A B) note A = this(1) and AB = this(2) and B = this(3) and sat = this(4)
  let ?ψ′ = ψ − {B}
  obtain I where I: I ⊨s ?ψ′ and cons: consistent-interp I  and tot: total-over-m I ?ψ′
    using sat unfolding satisfiable-def by auto
  have I ⊨ A using A I by (metis AB Diff-iff subset-mset.less-irrefl singletonD true-clss-def)
  then have I ⊨ B using AB subset-mset.less-imp-le true-cls-mono-leD by blast
  then have I ⊨s ψ using I by (metis insert-Diff-single true-clss-insert)
  then show ?case using cons satisfiable-carac′ by blast
qed

lemma simplify-preserves-unsat:
  assumes inference ψ ψ′
  shows satisfiable (fst ψ′) ⟶ satisfiable (fst ψ)
  using assms apply (induct rule: inference.induct)
  using satisfiable-decreasing by (metis fst-conv)+

lemma inference-preserves-unsat:
  assumes inference** S S′
  shows satisfiable (fst S′) ⟶  satisfiable (fst S)
  using assms apply (induct rule: rtranclp-induct)
  apply simp-all
  using simplify-preserves-unsat by blast

datatype ′v sem-tree = Node ′v ′v sem-tree ′v sem-tree | Leaf

fun sem-tree-size :: ′v sem-tree ⇒ nat where
sem-tree-size Leaf = 0 |
sem-tree-size (Node - ag ad) = 1 + sem-tree-size ag + sem-tree-size ad

lemma sem-tree-size[case-names bigger]:
  (⋀xs:: ′v sem-tree. (⋀ys:: ′v sem-tree. sem-tree-size ys < sem-tree-size xs ⟹ P ys) ⟹ P xs)
  ⟹ P xs
  by (fact Nat.measure-induct-rule)


fun partial-interps :: ′v sem-tree ⇒ ′v interp ⇒ ′v clauses ⇒ bool where
partial-interps Leaf I ψ = (∃χ. ¬ I ⊨ χ ∧ χ ∈ ψ ∧ total-over-m I {χ}) |
partial-interps (Node v ag ad) I ψ ⟷
  (partial-interps ag (I ∪ {Pos v}) ψ ∧ partial-interps ad (I∪ {Neg v}) ψ)


lemma simplify-preserve-partial-leaf:
  simplify N N′ ⟹ partial-interps Leaf I N ⟹ partial-interps Leaf I N′
  apply (induct rule: simplify.induct)
```

```
    using union-lcomm apply auto[1]
   apply (simp, metis atms-of-plus total-over-set-union true-cls-union)
  apply simp
  by (metis atms-of-ms-singleton mset-le-exists-conv subset-mset-def true-cls-mono-leD
    total-over-m-def total-over-m-sum)
```

```
lemma simplify-preserve-partial-tree:
  assumes simplify N N′
  and partial-interps t I N
  shows partial-interps t I N′
  using assms apply (induct t arbitrary: I, simp)
  using simplify-preserve-partial-leaf by metis
```

```
lemma inference-preserve-partial-tree:
  assumes inference S S′
  and partial-interps t I (fst S)
  shows partial-interps t I (fst S′)
  using assms apply (induct t arbitrary: I, simp-all)
  by (meson inference-increasing)
```

```
lemma rtranclp-inference-preserve-partial-tree:
  assumes rtranclp inference N N′
  and partial-interps t I (fst N)
  shows partial-interps t I (fst N′)
  using assms apply (induct rule: rtranclp-induct, auto)
  using inference-preserve-partial-tree by force
```

```
function build-sem-tree :: ′v :: linorder set  ⇒ ′v clauses ⇒ ′v sem-tree where
build-sem-tree atms ψ =
  (if atms = {} ∨ ¬ finite atms
  then Leaf
  else Node (Min atms) (build-sem-tree (Set.remove (Min atms) atms) ψ)
     (build-sem-tree (Set.remove (Min atms) atms) ψ))
by auto
termination
  apply (relation measure (λ(A, -).  card A), simp-all)
  apply (metis Min-in card-Diff1-less remove-def)+
done
declare build-sem-tree.induct[case-names tree]
```

```
lemma unsatisfiable-empty[simp]:
  ¬unsatisfiable {}
   unfolding satisfiable-def apply auto
  using consistent-interp-def unfolding total-over-m-def total-over-set-def atms-of-ms-def by blast
```

```
lemma partial-interps-build-sem-tree-atms-general:
  fixes ψ :: ′v :: linorder clauses and p :: ′v literal list
  assumes unsat: unsatisfiable ψ and finite ψ and consistent-interp I
  and finite atms
  and atms-of-ms ψ = atms ∪ atms-of-s I and atms ∩ atms-of-s I = {}
  shows partial-interps (build-sem-tree atms ψ) I ψ
  using assms
```

```
proof (induct arbitrary: I rule: build-sem-tree.induct)
 case (1 atms ψ Ia) note IH1 = this(1) and IH2 = this(2) and unsat = this(3) and finite = this(4)
   and cons = this(5)  and f = this(6) and un = this(7) and disj = this(8)
 {
   assume atms: atms = {}
   then have atmsIa: atms-of-ms ψ = atms-of-s Ia using un by auto
   then have total-over-m Ia ψ unfolding total-over-m-def atmsIa by auto
   then have χ: ∃ χ ∈ ψ. ¬ Ia ⊨ χ
     using unsat cons unfolding true-clss-def satisfiable-def by auto
   then have build-sem-tree atms ψ = Leaf using atms by auto
   moreover
     have tot: ⋀χ. χ ∈ ψ ⟹ total-over-m Ia {χ}
     unfolding total-over-m-def total-over-set-def atms-of-ms-def atms-of-s-def
     using atmsIa atms-of-ms-def by fastforce
   have partial-interps Leaf Ia ψ
     using χ tot by (auto simp add: total-over-m-def total-over-set-def atms-of-ms-def)

     ultimately have ?case by metis
 }
 moreover {
   assume atms: atms ≠ {}
   have build-sem-tree atms ψ = Node (Min atms) (build-sem-tree (Set.remove (Min atms) atms) ψ)
      (build-sem-tree (Set.remove (Min atms) atms) ψ)
     using build-sem-tree.simps[of atms ψ] f atms by metis

   have consistent-interp (Ia ∪ {Pos (Min atms)}) unfolding consistent-interp-def
     by (metis Int-iff Min-in Un-iff atm-of-uminus atms cons consistent-interp-def disj empty-iff
       f in-atms-of-s-decomp insert-iff literal.distinct(1) literal.exhaust-sel literal.sel(2)
       uminus-Neg uminus-Pos)
   moreover have atms-of-ms ψ = Set.remove (Min atms) atms ∪ atms-of-s (Ia ∪ {Pos (Min atms)})
     using Min-in atms f un by fastforce
   moreover have disj': Set.remove (Min atms) atms ∩ atms-of-s (Ia ∪ {Pos (Min atms)}) = {}
     by simp (metis disj disjoint-iff-not-equal member-remove)
   moreover have finite (Set.remove (Min atms) atms) using f by (simp add: remove-def)
   ultimately have subtree1: partial-interps (build-sem-tree (Set.remove (Min atms) atms) ψ)
      (Ia ∪ {Pos (Min atms)}) ψ
     using IH1[of Ia ∪ {Pos (Min (atms))}] atms f unsat finite by metis

   have consistent-interp (Ia ∪ {Neg (Min atms)}) unfolding consistent-interp-def
     by (metis Int-iff Min-in Un-iff atm-of-uminus atms cons consistent-interp-def disj empty-iff
       f in-atms-of-s-decomp insert-iff literal.distinct(1) literal.exhaust-sel literal.sel(2)
       uminus-Neg)
   moreover have atms-of-ms ψ = Set.remove (Min atms) atms ∪ atms-of-s (Ia ∪ {Neg (Min atms)})
      using ‹atms-of-ms ψ = Set.remove (Min atms) atms ∪ atms-of-s (Ia ∪ {Pos (Min atms)})› by
blast

   moreover have disj': Set.remove (Min atms) atms ∩ atms-of-s (Ia ∪ {Neg (Min atms)}) = {}
     using disj by auto
   moreover have finite (Set.remove (Min atms) atms) using f by (simp add: remove-def)
   ultimately have subtree2: partial-interps (build-sem-tree (Set.remove (Min atms) atms) ψ)
      (Ia ∪ {Neg (Min atms)}) ψ
     using IH2[of Ia ∪ {Neg (Min (atms))}]  atms f unsat finite by metis

   then have ?case
     using IH1 subtree1 subtree2 f local.finite unsat atms by simp
 }
```

```
  ultimately show ?case by metis
qed


lemma partial-interps-build-sem-tree-atms:
  fixes ψ :: 'v :: linorder clauses and p :: 'v literal list
  assumes unsat: unsatisfiable ψ and finite: finite ψ
  shows partial-interps (build-sem-tree (atms-of-ms ψ) ψ) {} ψ
proof –
  have consistent-interp {} unfolding consistent-interp-def by auto
  moreover have atms-of-ms ψ = atms-of-ms ψ ∪ atms-of-s {} unfolding atms-of-s-def by auto
  moreover have atms-of-ms ψ ∩ atms-of-s {} = {}  unfolding atms-of-s-def by auto
  moreover have finite (atms-of-ms ψ) unfolding atms-of-ms-def using finite by simp
  ultimately show partial-interps (build-sem-tree (atms-of-ms ψ) ψ) {} ψ
    using partial-interps-build-sem-tree-atms-general[of ψ {} atms-of-ms ψ] assms by metis
qed

lemma can-decrease-count:
  fixes ψ'' :: 'v clauses × ('v clause × 'v clause × 'v) set
  assumes count χ L = n
  and L ∈# χ and χ ∈ fst ψ
  shows ∃ψ' χ'. inference** ψ ψ' ∧ χ' ∈ fst ψ' ∧ (∀L. L ∈# χ ⟷ L ∈# χ')
               ∧ count χ' L = 1
               ∧ (∀φ. φ ∈ fst ψ ⟶ φ ∈ fst ψ')
               ∧ (I ⊨ χ ⟷ I ⊨ χ')
               ∧ (∀I'. total-over-m I' {χ} ⟶ total-over-m I' {χ'})
  using assms
proof (induct n arbitrary: χ ψ)
  case 0
  then show ?case by (simp add: not-in-iff[symmetric])
next
   case (Suc n χ)
   note IH = this(1) and count = this(2) and L = this(3) and χ = this(4)
   {
     assume n = 0
     then have inference** ψ ψ
     and χ ∈ fst ψ
     and ∀L. (L ∈# χ) ⟷ (L ∈# χ)
     and count χ L = (1::nat)
     and ∀φ. φ ∈ fst ψ ⟶ φ ∈ fst ψ
       by (auto simp add: count L χ)
     then have ?case by metis
   }
   moreover {
     assume n > 0
     then have ∃C. χ = C + {#L, L#}
        by (smt L Suc-eq-plus1-left add.left-commute add-diff-cancel-left' add-diff-cancel-right'
          count-greater-zero-iff count-single local.count multi-member-split plus-multiset.rep-eq)
     then obtain C where C: χ = C + {#L, L#} by metis
     let ?χ' = C +{#L#}
     let ?ψ' = (fst ψ ∪ {?χ'}, snd ψ)
     have φ: ∀φ ∈ fst ψ. (φ ∈ fst ψ ∨ φ ≠ ?χ') ⟷ φ ∈ fst ?ψ' unfolding C by auto
     have inf: inference ψ ?ψ'
       using C factoring χ prod.collapse union-commute inference-step by metis
     moreover have count': count ?χ' L = n using C count by auto
     moreover have Lχ': L ∈# ?χ' by auto
     moreover have χ′ψ′: ?χ′ ∈ fst ?ψ′ by auto
     ultimately obtain ψ′′ and χ′′
     where
       inference** ?ψ′ ψ′′ and
       α: χ′′ ∈ fst ψ′′ and
       ∀ La. (La ∈# ?χ′) ⟷ (La ∈# χ′′) and
       β: count χ′′ L = (1::nat) and
       φ′: ∀ φ. φ ∈ fst ?ψ′ ⟶ φ ∈ fst ψ′′ and
       Iχ: I ⊨ ?χ′ ⟷ I ⊨ χ′′ and
       tot: ∀ I′. total-over-m I′ {?χ′} ⟶ total-over-m I′ {χ′′}
       using IH[of ?χ′ ?ψ′] count′ Lχ′ χ′ψ′ by blast

     then have inference** ψ ψ′′
     and ∀ La. (La ∈# χ) ⟷ (La ∈# χ′′)
     using inf unfolding C by auto
     moreover have ∀ φ. φ ∈ fst ψ ⟶ φ ∈ fst ψ′′ using φ φ′ by metis
     moreover have I ⊨ χ ⟷ I ⊨ χ′′ using Iχ unfolding true-cls-def C by auto
     moreover have ∀ I′. total-over-m I′ {χ} ⟶ total-over-m I′ {χ′′}
       using tot unfolding C total-over-m-def by auto
     ultimately have ?case using φ φ′ α β by metis
  }
  ultimately show ?case by auto
qed

lemma can-decrease-tree-size:
  fixes ψ :: ′v state and tree :: ′v sem-tree
  assumes finite (fst ψ) and already-used-inv ψ
  and partial-interps tree I (fst ψ)
  shows ∃ (tree′:: ′v sem-tree) ψ′. inference** ψ ψ′ ∧ partial-interps tree′ I (fst ψ′)
           ∧ (sem-tree-size tree′ < sem-tree-size tree ∨ sem-tree-size tree = 0)
  using assms
proof (induct arbitrary: I rule: sem-tree-size)
  case (bigger xs I) note IH = this(1) and finite = this(2) and a-u-i = this(3) and part = this(4)

  {
    assume sem-tree-size xs = 0
    then have ?case using part by blast
  }

  moreover {
    assume sn0: sem-tree-size xs > 0
    obtain ag ad v where xs: xs = Node v ag ad using sn0 by (cases xs, auto)
    {
       assume sem-tree-size ag = 0 and sem-tree-size ad = 0
       then have ag: ag = Leaf and ad: ad = Leaf by (cases ag, auto) (cases ad, auto)

       then obtain χ χ′ where
        χ: ¬ I ∪ {Pos v} ⊨ χ and
        totχ: total-over-m (I ∪ {Pos v}) {χ} and
        χψ: χ ∈ fst ψ and
        χ′: ¬ I ∪ {Neg v} ⊨ χ′ and
        totχ′: total-over-m (I ∪ {Neg v}) {χ′} and
        χ′ψ: χ′ ∈ fst ψ
        using part unfolding xs by auto
       have Posv: Pos v ∉# χ using χ unfolding true-cls-def true-lit-def by auto
       have Negv: Neg v ∉# χ′ using χ′ unfolding true-cls-def true-lit-def by auto
```

```
{
  assume Negχ: Neg v ∉# χ
  have ¬ I ⊨ χ using χ Posv unfolding true-cls-def true-lit-def by auto
  moreover have total-over-m I {χ}
    using Posv Negχ atm-imp-pos-or-neg-lit totχ unfolding total-over-m-def total-over-set-def
    by fastforce
  ultimately have partial-interps Leaf I (fst ψ)
  and sem-tree-size Leaf < sem-tree-size xs
  and inference** ψ ψ
    unfolding xs by (auto simp add: χψ)
}
moreover {
  assume Posχ: Pos v ∉# χ'
  then have Iχ: ¬ I ⊨ χ' using χ' Posv unfolding true-cls-def true-lit-def by auto
  moreover have total-over-m I {χ'}
    using Negv Posχ atm-imp-pos-or-neg-lit totχ'
    unfolding total-over-m-def total-over-set-def by fastforce
  ultimately have partial-interps Leaf I (fst ψ) and
    sem-tree-size Leaf < sem-tree-size xs and
    inference** ψ ψ
    using χ'ψ Iχ unfolding xs by auto
}
moreover {
  assume neg: Neg v ∈# χ and pos: Pos v ∈# χ'
  then obtain ψ' χ2 where inf: rtranclp inference ψ ψ' and χ2incl: χ2 ∈ fst ψ'
    and χχ2-incl: ∀ L. L ∈# χ ⟷ L ∈# χ2
    and countχ2: count χ2 (Neg v) = 1
    and φ: ∀ φ::'v literal multiset. φ ∈ fst ψ ⟶ φ ∈ fst ψ'
    and Iχ: I ⊨ χ ⟷ I ⊨ χ2
    and tot-impχ: ∀ I'. total-over-m I' {χ} ⟶ total-over-m I' {χ2}
    using can-decrease-count[of χ Neg v count χ (Neg v) ψ I] χψ χ'ψ by auto

  have χ' ∈ fst ψ' by (simp add: χ'ψ φ)
  with pos
  obtain ψ'' χ2' where
  inf': inference** ψ' ψ''
  and χ2'-incl: χ2' ∈ fst ψ''
  and χ'χ2-incl: ∀ L::'v literal. (L ∈# χ') = (L ∈# χ2')
  and countχ2': count χ2' (Pos v) = (1::nat)
  and φ': ∀ φ::'v literal multiset. φ ∈ fst ψ' ⟶ φ ∈ fst ψ''
  and Iχ': I ⊨ χ' ⟷ I ⊨ χ2'
  and tot-impχ': ∀ I'. total-over-m I' {χ'} ⟶ total-over-m I' {χ2'}
  using can-decrease-count[of χ' Pos v  count χ' (Pos v) ψ' I] by auto

  obtain C where χ2: χ2 = C + {#Neg v#} and negC: Neg v ∉# C and posC: Pos v ∉# C
    proof –
      have ⋀m. Suc 0 – count m (Neg v) = count (χ2 – m) (Neg v)
        by (simp add: countχ2)
      then show ?thesis
        using that by (metis (no-types) One-nat-def Posv Suc-inject Suc-pred χχ2-incl
          count-diff count-single insert-DiffM2 mem-Collect-eq multi-member-skip neg
          not-gr0 set-mset-def union-commute)
    qed

  obtain C' where
    χ2': χ2' = C' + {#Pos v#} and
  posC': Pos v ∉# C' and
  negC': Neg v ∉# C'
  proof −
    assume a1: ⋀C'. ⟦χ2' = C' + {#Pos v#}; Pos v ∉# C'; Neg v ∉# C'⟧ ⟹ thesis
    have f2: ⋀n. (n::nat) − n = 0
      by simp
    have Neg v ∉# χ2' − {#Pos v#}
      using Negv χ'χ2-incl by (auto simp: not-in-iff)
    have count {#Pos v#} (Pos v) = 1
      by simp
    then show ?thesis
      by (metis χ'χ2-incl ‹Neg v ∉# χ2' − {#Pos v#}› a1 countχ2' count-diff f2
        insert-DiffM2 less-numeral-extra(3) mem-Collect-eq pos set-mset-def)
  qed

have already-used-inv ψ'
  using rtranclp-inference-preserves-already-used-inv[of ψ ψ'] a-u-i inf by blast
then have a-u-i-ψ'': already-used-inv ψ''
  using rtranclp-inference-preserves-already-used-inv a-u-i inf' unfolding tautology-def
  by simp

have totC: total-over-m I {C}
  using tot-impχ totχ tot-over-m-remove[of I Pos v C] negC posC unfolding χ2
  by (metis total-over-m-sum uminus-Neg uminus-of-uminus-id)
have totC': total-over-m I {C'}
  using tot-impχ' totχ' total-over-m-sum tot-over-m-remove[of I Neg v C'] negC' posC'
  unfolding χ2' by (metis total-over-m-sum uminus-Neg)
have ¬ I ⊨ C + C'
  using χ Iχ χ' Iχ' unfolding χ2 χ2' true-cls-def by auto
then have part-I-ψ''': partial-interps Leaf I (fst ψ'' ∪ {C + C'})
  using totC totC' by simp
    (metis ‹¬ I ⊨ C + C'› atms-of-ms-singleton total-over-m-def total-over-m-sum)
{
  assume ({#Pos v#} + C', {#Neg v#} + C) ∉ snd ψ''
  then have inf'': inference ψ'' (fst ψ'' ∪ {C + C'}, snd ψ'' ∪ {(χ2', χ2)})
    using add.commute φ' χ2incl ‹χ2' ∈ fst ψ''› unfolding χ2 χ2'
    by (metis prod.collapse inference-step resolution)
  have inference** ψ (fst ψ'' ∪ {C + C'}, snd ψ'' ∪ {(χ2', χ2)})
    using inf inf' inf'' rtranclp-trans by auto
  moreover have sem-tree-size Leaf < sem-tree-size xs unfolding xs by auto
  ultimately have ?case using part-I-ψ''' by (metis fst-conv)
}
moreover {
  assume a: ({#Pos v#} + C', {#Neg v#} + C) ∈ snd ψ''
  then have (∃χ ∈ fst ψ''. (∀I. total-over-m I {C+C'} ⟶ total-over-m I {χ})
          ∧ (∀I. total-over-m I {χ} ⟶ I ⊨ χ ⟶ I ⊨ C' + C))
        ∨ tautology (C' + C)
    proof −
      obtain p where p: Pos p ∈# ({#Pos v#} + C') and
      n: Neg p ∈# ({#Neg v#} + C) and
      decomp: ((∃χ∈fst ψ''.
                (∀I. total-over-m I {({#Pos v#} + C') − {#Pos p#}
                    + (({#Neg v#} + C) − {#Neg p#})}
                  ⟶ total-over-m I {χ})
                ∧ (∀I. total-over-m I {χ} ⟶ I ⊨ χ
                ⟶ I ⊨ ({#Pos v#} + C') − {#Pos p#} + (({#Neg v#} + C) − {#Neg p#}))
```

```
                    )
                ∨ tautology (({#Pos v#} + C') − {#Pos p#} + (({#Neg v#} + C) − {#Neg p#})))
            using a by (blast intro: allE[OF a-u-i-ψ''[unfolded subsumes-def Ball-def],
                of ({#Pos v#} + C', {#Neg v#} + C)])
          { assume p ≠ v
            then have Pos p ∈# C' ∧ Neg p ∈# C using p n by force
            then have ?thesis unfolding Bex-def by auto
          }
          moreover {
            assume p = v
           then have ?thesis using decomp by (metis add.commute add-diff-cancel-left')
          }
          ultimately show ?thesis by auto
        qed
      moreover {
        assume ∃χ ∈ fst ψ''. (∀I. total-over-m I {C+C'} ⟶ total-over-m I {χ})
          ∧ (∀I. total-over-m I {χ} ⟶ I ⊨ χ ⟶ I ⊨ C' + C)
        then obtain ϑ where ϑ: ϑ ∈ fst ψ'' and
          tot-ϑ-CC': ∀I. total-over-m I {C+C'} ⟶ total-over-m I {ϑ} and
          ϑ-inv: ∀I. total-over-m I {ϑ} ⟶ I ⊨ ϑ ⟶ I ⊨ C' + C by blast
        have partial-interps Leaf I (fst ψ'')
          using tot-ϑ-CC' ϑ ϑ-inv totC totC' ‹¬ I ⊨ C + C'› total-over-m-sum by fastforce
        moreover have sem-tree-size Leaf < sem-tree-size xs unfolding xs by auto
        ultimately have ?case by (metis inf inf' rtranclp-trans)
      }
      moreover {
        assume tautCC': tautology (C' + C)
        have total-over-m I {C'+C} using totC totC' total-over-m-sum by auto
        then have ¬tautology (C' + C)
          using ‹¬ I ⊨ C + C'› unfolding add.commute[of C C'] total-over-m-def
          unfolding tautology-def by auto
        then have False using tautCC'  unfolding tautology-def by auto
      }
      ultimately have ?case by auto
    }
    ultimately have ?case by auto
  }
  ultimately have ?case using part by (metis (no-types) sem-tree-size.simps(1))
}
moreover {
  assume size-ag: sem-tree-size ag > 0
  have sem-tree-size ag < sem-tree-size xs unfolding xs by auto
  moreover have partial-interps ag (I ∪ {Pos v}) (fst ψ)
    and partad: partial-interps ad (I ∪ {Neg v}) (fst ψ)
    using part partial-interps.simps(2) unfolding xs by metis+
  moreover have sem-tree-size ag < sem-tree-size xs ⟶ finite (fst ψ) ⟶ already-used-inv ψ
    ⟶ ( partial-interps ag (I ∪ {Pos v}) (fst ψ) ⟶
    (∃tree' ψ'. inference** ψ ψ' ∧ partial-interps tree' (I ∪ {Pos v}) (fst ψ')
      ∧ (sem-tree-size tree' < sem-tree-size ag ∨ sem-tree-size ag = 0)))
      using IH by auto
  ultimately obtain ψ' :: 'v state and  tree' :: 'v sem-tree  where
    inf: inference** ψ ψ'
    and part: partial-interps tree' (I ∪ {Pos v}) (fst ψ')
    and size: sem-tree-size tree' < sem-tree-size ag ∨ sem-tree-size ag = 0
    using finite part rtranclp.rtrancl-refl a-u-i by blast
```

```
      have partial-interps ad (I ∪ {Neg v}) (fst ψ')
        using rtranclp-inference-preserve-partial-tree inf partad by metis
      then have partial-interps (Node v tree' ad) I (fst ψ') using part by auto
      then have ?case using inf size size-ag part unfolding xs by fastforce
    }
    moreover {
      assume size-ad: sem-tree-size ad > 0
      have sem-tree-size ad < sem-tree-size xs unfolding xs by auto
      moreover have partag: partial-interps ag (I ∪ {Pos v}) (fst ψ) and
        partial-interps ad (I ∪ {Neg v}) (fst ψ)
        using part partial-interps.simps(2) unfolding xs by metis+
      moreover have sem-tree-size ad < sem-tree-size xs ⟶ finite (fst ψ) ⟶ already-used-inv ψ
        ⟶ ( partial-interps ad (I ∪ {Neg v}) (fst ψ)
        ⟶ (∃ tree' ψ'. inference** ψ ψ' ∧ partial-interps tree' (I ∪ {Neg v}) (fst ψ')
            ∧ (sem-tree-size tree' < sem-tree-size ad ∨ sem-tree-size ad = 0)))
        using IH by auto
      ultimately obtain ψ' :: 'v state and  tree' :: 'v sem-tree  where
        inf: inference** ψ ψ'
        and part: partial-interps tree' (I ∪ {Neg v}) (fst ψ')
        and size: sem-tree-size tree' < sem-tree-size ad ∨ sem-tree-size ad = 0
        using finite part  rtranclp.rtrancl-refl a-u-i by blast

      have partial-interps ag (I ∪ {Pos v}) (fst ψ')
        using rtranclp-inference-preserve-partial-tree inf partag by metis
      then have partial-interps (Node v ag tree') I (fst ψ') using part by auto
      then have ?case using inf size size-ad unfolding xs by fastforce
    }
    ultimately have ?case by auto
  }
  ultimately show ?case by auto
qed

lemma inference-completeness-inv:
  fixes ψ :: 'v ::linorder state
  assumes
    unsat: ¬satisfiable (fst ψ) and
    finite: finite (fst ψ) and
    a-u-v: already-used-inv ψ
  shows ∃ψ'. (inference** ψ ψ' ∧ {#} ∈ fst ψ')
proof –
  obtain tree where  partial-interps tree {} (fst ψ)
    using partial-interps-build-sem-tree-atms assms by metis
  then show ?thesis
    using unsat finite a-u-v
    proof (induct tree arbitrary: ψ rule: sem-tree-size)
      case (bigger tree ψ) note H = this
      {
        fix χ
        assume tree: tree = Leaf
        obtain χ where χ: ¬ {} ⊨ χ and totχ: total-over-m {} {χ} and χψ: χ ∈ fst ψ
          using H unfolding tree by auto
        moreover have {#} = χ
          using  totχ unfolding total-over-m-def total-over-set-def by fastforce
        moreover have inference** ψ ψ by auto
        ultimately have ?case by metis
      }
      moreover {
        fix v tree1 tree2
        assume tree: tree = Node v tree1 tree2
        obtain
          tree' ψ' where inf: inference** ψ ψ' and
          part': partial-interps tree' {} (fst ψ') and
          decrease: sem-tree-size tree' < sem-tree-size tree ∨ sem-tree-size tree = 0
          using can-decrease-tree-size[of ψ] H(2,4,5) unfolding tautology-def by meson
        have sem-tree-size tree' < sem-tree-size tree using decrease unfolding tree by auto
        moreover have finite (fst ψ') using rtranclp-inference-preserves-finite inf H(4) by metis
        moreover have unsatisfiable (fst ψ')
          using inference-preserves-unsat inf bigger.prems(2) by blast
        moreover have already-used-inv ψ'
          using H(5) inf rtranclp-inference-preserves-already-used-inv[of ψ ψ'] by auto
        ultimately have ?case using inf rtranclp-trans part' H(1) by fastforce
      }
      ultimately show ?case by (cases tree, auto)
   qed
qed

lemma inference-completeness:
  fixes ψ :: 'v ::linorder state
  assumes unsat: ¬satisfiable (fst ψ)
  and finite: finite (fst ψ)
  and snd ψ = {}
  shows ∃ψ'. (rtranclp inference ψ ψ' ∧ {#} ∈ fst ψ')
proof -
  have already-used-inv ψ unfolding assms by auto
  then show ?thesis using assms inference-completeness-inv by blast
qed

lemma inference-soundness:
  fixes ψ :: 'v ::linorder state
  assumes rtranclp inference ψ ψ' and {#} ∈ fst ψ'
  shows unsatisfiable (fst ψ)
  using assms by (meson rtranclp-inference-preserves-un-sat satisfiable-def true-cls-empty
    true-clss-def)

lemma inference-soundness-and-completeness:
fixes ψ :: 'v ::linorder state
assumes finite: finite (fst ψ)
and snd ψ = {}
shows (∃ψ'. (inference** ψ ψ' ∧ {#} ∈ fst ψ')) ⟷ unsatisfiable (fst ψ)
  using assms inference-completeness inference-soundness by metis
```

1.1.4 Lemma about the simplified state

```
abbreviation simplified ψ ≡ (no-step simplify ψ)

lemma simplified-count:
  assumes simp: simplified ψ and χ: χ ∈ ψ
  shows count χ L ≤ 1
proof -
  {
    let ?χ' = χ - {#L, L#}
    assume count χ L ≥ 2
    then have f1: count (χ − {#L, L#} + {#L, L#}) L = count χ L
      by simp
    then have L ∈# χ − {#L#}
      by (metis (no-types) add.left-neutral add-diff-cancel-left′ count-union diff-diff-add
        diff-single-trivial insert-DiffM mem-Collect-eq multi-member-this not-gr0 set-mset-def)
    then have χ′: ?χ′ + {#L#} + {#L#} = χ
      using f1 by (metis diff-diff-add diff-single-eq-union in-diffD)

    have ∃ψ′. simplify ψ ψ′
      by (metis (no-types, hide-lams) χ χ′ add.commute factoring-imp-simplify union-assoc)
    then have False using simp by auto
  }
  then show ?thesis by arith
qed

lemma simplified-no-both:
  assumes  simp: simplified ψ and χ: χ ∈ ψ
  shows ¬ (L ∈# χ ∧ −L ∈# χ)
proof (rule ccontr)
  assume ¬ ¬ (L ∈# χ ∧ − L ∈# χ)
  then have L ∈# χ ∧ − L ∈# χ by metis
  then obtain  χ′ where χ = χ′ + {#Pos (atm-of L)#}+ {#Neg (atm-of L)#}
    by (metis Neg-atm-of-iff Pos-atm-of-iff diff-union-swap insert-DiffM2 uminus-Neg uminus-Pos)
  then show False using χ simp tautology-deletion by fastforce
qed

lemma simplified-not-tautology:
  assumes simplified {ψ}
  shows ~tautology ψ
proof (rule ccontr)
  assume ~ ?thesis
  then obtain p where Pos p ∈# ψ ∧ Neg p ∈# ψ using tautology-decomp by metis
  then obtain χ where ψ = χ + {#Pos p#} + {#Neg p#}
    by (metis insert-noteq-member literal.distinct(1) multi-member-split)
  then have ~ simplified {ψ} by (auto intro: tautology-deletion)
  then show False using assms by auto
qed

lemma simplified-remove:
  assumes simplified {ψ}
  shows simplified {ψ − {#l#}}
proof (rule ccontr)
  assume ns: ¬ simplified {ψ − {#l#}}
  {
    assume l ∉# ψ
    then have ψ − {#l#} = ψ by simp
    then have False using ns assms by auto
  }
  moreover {
    assume lψ: l∈# ψ
    have A: ⋀A. A ∈ {ψ − {#l#}} ⟷ A + {#l#} ∈ {ψ} by (auto simp add: lψ)
    obtain l′ where l′: simplify {ψ − {#l#}} l′ using ns by metis
    then have ∃l′. simplify {ψ} l′
      proof (induction rule: simplify.induct)
        case (tautology-deletion A P)
        have {#Neg P#} + ({#Pos P#} + (A + {#l#})) ∈ {ψ}
```

```
          by (metis (no-types) A add.commute tautology-deletion.hyps union-lcomm)
        then show ?thesis
           by (metis simplify.tautology-deletion[of A+{#l#} P {ψ}] add.commute)
      next
        case (condensation A L)
        have A + {#L#} + {#L#} + {#l#} ∈ {ψ}
          using A condensation.hyps by blast
        then have {#L, L#} + (A + {#l#}) ∈ {ψ}
          by (metis (no-types) union-assoc union-commute)
        then show ?case
          using factoring-imp-simplify by blast
      next
        case (subsumption A B)
        then show ?case by blast
      qed
    then have False using assms(1) by blast
  }
  ultimately show False by auto
qed


lemma in-simplified-simplified:
  assumes simp: simplified ψ and incl: ψ′ ⊆ ψ
  shows simplified ψ′
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain ψ′′ where simplify ψ′ ψ′′  by metis
    then have ∃ l′. simplify ψ l′
      proof (induction rule: simplify.induct)
        case (tautology-deletion A P)
        then show ?thesis using simplify.tautology-deletion[of A P ψ] incl by blast
      next
        case (condensation A L)
        then show ?case using simplify.condensation[of A L ψ] incl by blast
      next
        case (subsumption A B)
        then show ?case using simplify.subsumption[of A ψ B] incl by auto
      qed
  then show False using assms(1) by blast
qed

lemma simplified-in:
  assumes simplified ψ
  and N ∈ ψ
  shows simplified {N}
  using assms by (metis Set.set-insert empty-subsetI in-simplified-simplified insert-mono)

lemma subsumes-imp-formula:
  assumes ψ ≤# φ
  shows {ψ} ⊨p φ
  unfolding true-clss-cls-def apply auto
  using assms true-cls-mono-leD by blast

lemma simplified-imp-distinct-mset-tauto:
  assumes simp: simplified ψ′
  shows distinct-mset-set ψ′ and  ∀ χ ∈ ψ′. ¬tautology χ
```

```
proof –
  show ∀ χ ∈ ψ'. ¬tautology χ
    using simp by (auto simp add: simplified-in simplified-not-tautology)

  show distinct-mset-set ψ'
    proof (rule ccontr)
      assume ¬?thesis
      then obtain χ where χ ∈ ψ' and ¬distinct-mset χ unfolding distinct-mset-set-def by auto
      then obtain L where count χ L ≥ 2
        unfolding distinct-mset-def
        by (meson count-greater-eq-one-iff le-antisym simp simplified-count)
      then show False by (metis Suc-1 ‹χ ∈ ψ'› not-less-eq-eq simp simplified-count)
    qed
qed

lemma simplified-no-more-full1-simplified:
  assumes simplified ψ
  shows ¬full1 simplify ψ ψ'
  using assms unfolding full1-def by (meson tranclpD)
```

1.1.5 Resolution and Invariants

```
inductive resolution :: 'v state ⇒ 'v state ⇒ bool where
full1-simp: full1 simplify N N' ⟹ resolution (N, already-used) (N', already-used) |
inferring: inference (N, already-used) (N', already-used') ⟹ simplified N
  ⟹ full simplify N' N'' ⟹ resolution (N, already-used) (N'', already-used')
```

Invariants

```
lemma resolution-finite:
  assumes resolution ψ ψ' and finite (fst ψ)
  shows finite (fst ψ')
  using assms by (induct rule: resolution.induct)
    (auto simp add: full1-def full-def  rtranclp-simplify-preserves-finite
      dest: tranclp-into-rtranclp inference-preserves-finite)

lemma rtranclp-resolution-finite:
  assumes resolution** ψ ψ' and finite (fst ψ)
  shows finite (fst ψ')
  using assms by (induct rule: rtranclp-induct, auto simp add: resolution-finite)

lemma resolution-finite-snd:
  assumes resolution ψ ψ' and finite (snd ψ)
  shows finite (snd ψ')
  using assms apply (induct rule: resolution.induct, auto simp add: inference-preserves-finite-snd)
  using inference-preserves-finite-snd snd-conv by metis

lemma rtranclp-resolution-finite-snd:
  assumes resolution** ψ ψ' and finite (snd ψ)
  shows finite (snd ψ')
  using assms by (induct rule: rtranclp-induct, auto simp add: resolution-finite-snd)

lemma resolution-always-simplified:
 assumes resolution ψ ψ'
 shows simplified (fst ψ')
 using assms by (induct rule: resolution.induct)
```

```
  (auto simp add: full1-def full-def)

lemma tranclp-resolution-always-simplified:
  assumes tranclp resolution ψ ψ'
  shows simplified (fst ψ')
  using assms by (induct rule: tranclp.induct, auto simp add: resolution-always-simplified)

lemma resolution-atms-of:
  assumes resolution ψ ψ' and finite (fst ψ)
  shows atms-of-ms (fst ψ') ⊆ atms-of-ms (fst ψ)
  using assms apply (induct rule: resolution.induct)
    apply(simp add: rtranclp-simplify-atms-of-ms tranclp-into-rtranclp full1-def )
  by (metis (no-types, lifting) contra-subsetD fst-conv full-def
    inference-preserves-atms-of-ms rtranclp-simplify-atms-of-ms subsetI)

lemma rtranclp-resolution-atms-of:
  assumes resolution** ψ ψ' and finite (fst ψ)
  shows atms-of-ms (fst ψ') ⊆ atms-of-ms (fst ψ)
  using assms apply (induct rule: rtranclp-induct)
  using resolution-atms-of rtranclp-resolution-finite by blast+

lemma resolution-include:
  assumes res: resolution ψ ψ' and finite: finite (fst ψ)
  shows fst ψ' ⊆ simple-clss (atms-of-ms (fst ψ))
proof −
  have finite': finite (fst ψ') using local.finite res resolution-finite by blast
  have simplified (fst ψ') using res finite' resolution-always-simplified by blast
  then have fst ψ' ⊆ simple-clss (atms-of-ms (fst ψ'))
    using simplified-in-simple-clss finite' simplified-imp-distinct-mset-tauto[of fst ψ'] by auto
  moreover have atms-of-ms (fst ψ') ⊆ atms-of-ms (fst ψ)
    using res finite resolution-atms-of[of ψ ψ'] by auto
  ultimately show ?thesis by (meson atms-of-ms-finite local.finite order.trans rev-finite-subset
    simple-clss-mono)
qed

lemma rtranclp-resolution-include:
  assumes res: tranclp resolution ψ ψ' and finite: finite (fst ψ)
  shows fst ψ' ⊆ simple-clss (atms-of-ms (fst ψ))
  using assms apply (induct rule: tranclp.induct)
    apply (simp add: resolution-include)
  by (meson simple-clss-mono order-class.le-trans resolution-include
    rtranclp-resolution-atms-of rtranclp-resolution-finite tranclp-into-rtranclp)

abbreviation already-used-all-simple
  :: ('a literal multiset × 'a literal multiset) set ⇒ 'a set ⇒ bool where
already-used-all-simple already-used vars ≡
(∀ (A, B) ∈ already-used. simplified {A} ∧ simplified {B} ∧ atms-of A ⊆ vars ∧ atms-of B ⊆ vars)

lemma already-used-all-simple-vars-incl:
  assumes vars ⊆ vars'
  shows already-used-all-simple a vars ⟹ already-used-all-simple a vars'
  using assms by fast

lemma inference-clause-preserves-already-used-all-simple:
  assumes inference-clause S S'
  and already-used-all-simple (snd S) vars
  and simplified (fst S)
  and atms-of-ms (fst S) ⊆ vars
  shows already-used-all-simple (snd (fst S ∪ {fst S'}, snd S')) vars
  using assms
proof (induct rule: inference-clause.induct)
  case (factoring L C N already-used)
  then show ?case by (simp add: simplified-in factoring-imp-simplify)
next
  case (resolution P C N D already-used) note H = this
  show ?case apply clarify
    proof −
      fix A B v
      assume (A, B) ∈ snd (fst (N, already-used)
        ∪ {fst (C + D, already-used ∪ {({#Pos P#} + C, {#Neg P#} + D)})},
          snd (C + D, already-used ∪ {({#Pos P#} + C, {#Neg P#} + D)}))
      then have (A, B) ∈ already-used ∨ (A, B) = ({#Pos P#} + C, {#Neg P#} + D) by auto
      moreover {
        assume (A, B) ∈ already-used
        then have simplified {A} ∧ simplified {B} ∧ atms-of A ⊆ vars ∧ atms-of B ⊆ vars
          using H(4) by auto
      }
      moreover {
        assume eq: (A, B) = ({#Pos P#} + C, {#Neg P#} + D)
        then have simplified {A} using simplified-in H(1,5) by auto
        moreover have simplified {B} using eq simplified-in H(2,5) by auto
        moreover have atms-of A ⊆ atms-of-ms N
          using eq H(1)
          using atms-of-atms-of-ms-mono[of A N] by auto
        moreover have atms-of B ⊆ atms-of-ms N
          using eq H(2) atms-of-atms-of-ms-mono[of B N] by auto
        ultimately have simplified {A} ∧ simplified {B} ∧ atms-of A ⊆ vars ∧ atms-of B ⊆ vars
          using H(6) by auto
      }
      ultimately show simplified {A} ∧ simplified {B} ∧ atms-of A ⊆ vars ∧ atms-of B ⊆ vars
        by fast
    qed
qed

lemma inference-preserves-already-used-all-simple:
  assumes inference S S'
  and already-used-all-simple (snd S) vars
  and simplified (fst S)
  and atms-of-ms (fst S) ⊆ vars
  shows already-used-all-simple (snd S') vars
  using assms
proof (induct rule: inference.induct)
  case (inference-step S clause already-used)
  then show ?case
    using inference-clause-preserves-already-used-all-simple[of S (clause, already-used) vars]
    by auto
qed

lemma already-used-all-simple-inv:
  assumes resolution S S'
  and already-used-all-simple (snd S) vars
  and atms-of-ms (fst S) ⊆ vars
  shows already-used-all-simple (snd S') vars
  using assms
proof (induct rule: resolution.induct)
  case (full1-simp N N')
  then show ?case by simp
next
  case (inferring N already-used N' already-used' N'')
  then show already-used-all-simple (snd (N'', already-used')) vars
    using inference-preserves-already-used-all-simple[of (N, already-used)] by simp
qed

lemma rtranclp-already-used-all-simple-inv:
  assumes resolution** S S'
  and already-used-all-simple (snd S) vars
  and atms-of-ms (fst S) ⊆ vars
  and finite (fst S)
  shows already-used-all-simple (snd S') vars
  using assms
proof (induct rule: rtranclp-induct)
  case base
  then show ?case by simp
next
  case (step S' S'') note infstar = this(1) and IH = this(3) and res = this(2) and
    already = this(4) and atms = this(5) and finite = this(6)
  have already-used-all-simple (snd S') vars using IH already atms finite by simp
  moreover have atms-of-ms (fst S') ⊆ atms-of-ms (fst S)
    by (simp add: infstar local.finite rtranclp-resolution-atms-of)
  then have atms-of-ms (fst S') ⊆ vars using atms by auto
  ultimately show ?case
    using already-used-all-simple-inv[OF res] by simp
qed

lemma inference-clause-simplified-already-used-subset:
  assumes inference-clause S S'
  and simplified (fst S)
  shows snd S ⊂ snd S'
  using assms apply (induct rule: inference-clause.induct, auto)
  using factoring-imp-simplify by blast

lemma inference-simplified-already-used-subset:
  assumes inference S S'
  and simplified (fst S)
  shows snd S ⊂ snd S'
  using assms apply (induct rule: inference.induct)
  by (metis inference-clause-simplified-already-used-subset snd-conv)

lemma resolution-simplified-already-used-subset:
  assumes resolution S S'
  and simplified (fst S)
  shows snd S ⊂ snd S'
  using assms apply (induct rule: resolution.induct, simp-all add: full1-def)
  apply (meson tranclpD)
  by (metis inference-simplified-already-used-subset fst-conv snd-conv)

lemma tranclp-resolution-simplified-already-used-subset:
  assumes tranclp resolution S S'
  and simplified (fst S)
  shows snd S ⊂ snd S'
  using assms apply (induct rule: tranclp.induct)
  using resolution-simplified-already-used-subset apply metis
  by (meson tranclp-resolution-always-simplified resolution-simplified-already-used-subset
    less-trans)

abbreviation already-used-top vars ≡ simple-clss vars × simple-clss vars

lemma already-used-all-simple-in-already-used-top:
  assumes already-used-all-simple s vars and finite vars
  shows s ⊆ already-used-top vars
proof
  fix x
  assume x-s: x ∈ s
  obtain A B where x: x = (A, B) by (cases x, auto)
  then have simplified {A} and atms-of A ⊆ vars using assms(1) x-s by fastforce+
  then have A: A ∈ simple-clss vars
    using simple-clss-mono[of atms-of A vars] x assms(2)
    simplified-imp-distinct-mset-tauto[of {A}]
    distinct-mset-not-tautology-implies-in-simple-clss by fast
  moreover have simplified {B} and atms-of B ⊆ vars using assms(1) x-s x by fast+
  then have B: B ∈ simple-clss vars
    using simplified-imp-distinct-mset-tauto[of {B}]
    distinct-mset-not-tautology-implies-in-simple-clss
    simple-clss-mono[of atms-of B vars] x assms(2) by fast
  ultimately show x ∈ simple-clss vars × simple-clss vars
    unfolding x by auto
qed

lemma already-used-top-finite:
  assumes finite vars
  shows finite (already-used-top vars)
  using simple-clss-finite assms by auto

lemma already-used-top-increasing:
  assumes var ⊆ var' and finite var'
  shows already-used-top var ⊆ already-used-top var'
  using assms simple-clss-mono by auto

lemma already-used-all-simple-finite:
  fixes s :: ('a literal multiset × 'a literal multiset) set and vars :: 'a set
  assumes already-used-all-simple s vars and finite vars
  shows finite s
  using assms already-used-all-simple-in-already-used-top[OF assms(1)]
  rev-finite-subset[OF already-used-top-finite[of vars]] by auto

abbreviation card-simple vars ψ ≡ card (already-used-top vars − ψ)

lemma resolution-card-simple-decreasing:
  assumes res: resolution ψ ψ'
  and a-u-s: already-used-all-simple (snd ψ) vars
  and finite-v: finite vars
  and finite-fst: finite (fst ψ)
  and finite-snd: finite (snd ψ)
  and simp: simplified (fst ψ)
  and atms-of-ms (fst ψ) ⊆ vars
  shows card-simple vars (snd ψ') < card-simple vars (snd ψ)
proof -
  let ?vars = vars
  let ?top = simple-clss ?vars × simple-clss ?vars
  have 1: card-simple vars (snd ψ) = card ?top − card (snd ψ)
    using card-Diff-subset finite-snd  already-used-all-simple-in-already-used-top[OF a-u-s]
    finite-v by metis
  have a-u-s': already-used-all-simple (snd ψ') vars
    using already-used-all-simple-inv res a-u-s assms(7) by blast
  have f: finite (snd ψ') using already-used-all-simple-finite a-u-s' finite-v by auto
  have 2: card-simple vars (snd ψ') = card ?top − card (snd ψ')
    using card-Diff-subset[OF f] already-used-all-simple-in-already-used-top[OF a-u-s' finite-v]
    by auto
  have card (already-used-top vars) ≥ card (snd ψ')
    using already-used-all-simple-in-already-used-top[OF a-u-s' finite-v]
    card-mono[of already-used-top vars snd ψ'] already-used-top-finite[OF finite-v] by metis
  then show ?thesis
    using  psubset-card-mono[OF f resolution-simplified-already-used-subset[OF res simp]]
    unfolding 1 2 by linarith
qed


lemma tranclp-resolution-card-simple-decreasing:
  assumes tranclp resolution ψ ψ' and finite-fst: finite (fst ψ)
  and already-used-all-simple (snd ψ) vars
  and atms-of-ms (fst ψ) ⊆ vars
  and finite-v: finite vars
  and finite-snd: finite (snd ψ)
  and simplified (fst ψ)
  shows card-simple vars (snd ψ') < card-simple vars (snd ψ)
  using assms
proof (induct rule: tranclp-induct)
  case (base ψ')
  then show ?case by (simp add: resolution-card-simple-decreasing)
next
  case (step ψ' ψ'') note res = this(1) and res' = this(2) and a-u-s = this(5) and
    atms = this(6) and f-v = this(7) and f-fst = this(4) and H = this
  then have card-simple vars (snd ψ') < card-simple vars (snd ψ) by auto
  moreover have a-u-s': already-used-all-simple (snd ψ') vars
    using rtranclp-already-used-all-simple-inv[OF tranclp-into-rtranclp[OF res] a-u-s atms f-fst] .
  have finite (fst ψ')
    by (meson finite-fst res rtranclp-resolution-finite tranclp-into-rtranclp)
  moreover have finite (snd ψ') using already-used-all-simple-finite[OF a-u-s' f-v] .
  moreover have simplified (fst ψ') using res tranclp-resolution-always-simplified by blast
  moreover have atms-of-ms (fst ψ') ⊆ vars
    by (meson atms f-fst order.trans res rtranclp-resolution-atms-of tranclp-into-rtranclp)
  ultimately show ?case
    using resolution-card-simple-decreasing[OF res' a-u-s' f-v] f-v
    less-trans[of card-simple vars (snd ψ'') card-simple vars (snd ψ')
      card-simple vars (snd ψ)]
    by blast
qed


lemma tranclp-resolution-card-simple-decreasing-2:
  assumes tranclp resolution ψ ψ'
  and finite-fst: finite (fst ψ)
  and empty-snd: snd ψ = {}
  and simplified (fst ψ)
  shows card-simple (atms-of-ms (fst ψ)) (snd ψ') < card-simple (atms-of-ms (fst ψ)) (snd ψ)
proof −
  let ?vars = atms-of-ms (fst ψ)
  have already-used-all-simple (snd ψ) ?vars unfolding empty-snd by auto
  moreover have atms-of-ms (fst ψ) ⊆ ?vars by auto
  moreover have finite-v: finite ?vars using finite-fst by auto
  moreover have finite-snd: finite (snd ψ) unfolding empty-snd by auto
  ultimately show ?thesis
    using assms(1,2,4) tranclp-resolution-card-simple-decreasing[of ψ ψ'] by presburger
qed
```

well-foundness if the relation

```
lemma wf-simplified-resolution:
  assumes f-vars: finite vars
  shows wf {(y:: 'v:: linorder state, x). (atms-of-ms (fst x) ⊆ vars ∧ simplified (fst x)
    ∧ finite (snd x) ∧ finite (fst x) ∧ already-used-all-simple (snd x) vars) ∧ resolution x y}
proof −
  {
    fix a b :: 'v::linorder state
    assume (b, a) ∈ {(y, x). (atms-of-ms (fst x) ⊆ vars ∧ simplified (fst x) ∧ finite (snd x)
      ∧ finite (fst x) ∧ already-used-all-simple (snd x) vars) ∧ resolution x y}
    then have
      atms-of-ms (fst a) ⊆ vars and
      simp: simplified (fst a) and
      finite (snd a) and
      finite (fst a) and
      a-u-v: already-used-all-simple (snd a) vars and
      res: resolution a b by auto
    have finite (already-used-top vars) using f-vars already-used-top-finite by blast
    moreover have already-used-top vars ⊆ already-used-top vars by auto
    moreover have snd b ⊆ already-used-top vars
      using already-used-all-simple-in-already-used-top[of snd b vars]
      a-u-v already-used-all-simple-inv[OF res] ‹finite (fst a)› ‹atms-of-ms (fst a) ⊆ vars› f-vars
      by presburger
    moreover have snd a ⊂ snd b using resolution-simplified-already-used-subset[OF res simp] .
    ultimately have finite (already-used-top vars) ∧ already-used-top vars ⊆ already-used-top vars
      ∧ snd b ⊆ already-used-top vars ∧ snd a ⊂ snd b by metis
  }
  then show ?thesis using wf-bounded-set[of {(y:: 'v:: linorder state, x).
    (atms-of-ms (fst x) ⊆ vars
    ∧ simplified (fst x) ∧ finite (snd x) ∧ finite (fst x)∧ already-used-all-simple (snd x) vars)
    ∧ resolution x y} λ-. already-used-top vars snd] by auto
qed

lemma wf-simplified-resolution':
  assumes f-vars: finite vars
  shows wf {(y:: 'v:: linorder state, x). (atms-of-ms (fst x) ⊆ vars ∧ ¬simplified (fst x)
    ∧ finite (snd x) ∧ finite (fst x) ∧ already-used-all-simple (snd x) vars) ∧ resolution x y}
  unfolding wf-def
   apply (simp add: resolution-always-simplified)
  by (metis (mono-tags, hide-lams) fst-conv resolution-always-simplified)
```

```
lemma wf-resolution:
  assumes f-vars: finite vars
  shows wf ({(y:: 'v:: linorder state, x). (atms-of-ms (fst x) ⊆ vars ∧ simplified (fst x)
        ∧ finite (snd x) ∧ finite (fst x) ∧ already-used-all-simple (snd x) vars) ∧ resolution x y}
    ∪ {(y, x). (atms-of-ms (fst x) ⊆ vars ∧ ¬ simplified (fst x) ∧ finite (snd x) ∧ finite (fst x)
       ∧ already-used-all-simple (snd x) vars) ∧ resolution x y}) (is wf (?R ∪ ?S))
proof −
  have Domain ?R Int Range ?S = {} using resolution-always-simplified by auto blast
  then show wf (?R ∪ ?S)
    using wf-simplified-resolution[OF f-vars] wf-simplified-resolution'[OF f-vars] wf-Un[of ?R ?S]
    by fast
qed

lemma rtrancp-simplify-already-used-inv:
  assumes simplify** S S'
  and already-used-inv (S, N)
  shows already-used-inv (S', N)
  using assms apply induction
  using simplify-preserves-already-used-inv by fast+

lemma full1-simplify-already-used-inv:
  assumes full1 simplify S S'
  and already-used-inv (S, N)
  shows already-used-inv (S', N)
  using assms tranclp-into-rtranclp[of simplify S S'] rtrancp-simplify-already-used-inv
  unfolding full1-def by fast

lemma full-simplify-already-used-inv:
  assumes full simplify S S'
  and already-used-inv (S, N)
  shows already-used-inv (S', N)
  using assms  rtrancp-simplify-already-used-inv unfolding full-def by fast
lemma resolution-already-used-inv:
  assumes resolution S S'
  and already-used-inv S
  shows already-used-inv S'
  using assms
proof induction
  case (full1-simp N N' already-used)
  then show ?case using full1-simplify-already-used-inv by fast
next
  case (inferring N already-used N' already-used' N'') note inf = this(1) and full = this(3) and
    a-u-v = this(4)
  then show ?case
    using inference-preserves-already-used-inv[OF inf a-u-v] full-simplify-already-used-inv full
    by fast
qed

lemma rtranclp-resolution-already-used-inv:
  assumes resolution** S S'
  and already-used-inv S
  shows already-used-inv S'
  using assms apply induction
  using resolution-already-used-inv by fast+
lemma rtanclp-simplify-preserves-unsat:
  assumes simplify** ψ ψ'
  shows satisfiable ψ' ⟶ satisfiable ψ
  using assms apply induction
  using simplify-clause-preserves-sat by blast+

lemma full1-simplify-preserves-unsat:
  assumes full1 simplify ψ ψ'
  shows satisfiable ψ' ⟶ satisfiable ψ
  using assms rtanclp-simplify-preserves-unsat[of ψ ψ'] tranclp-into-rtranclp
  unfolding full1-def by metis

lemma full-simplify-preserves-unsat:
  assumes full simplify ψ ψ'
  shows satisfiable ψ' ⟶ satisfiable ψ
  using assms rtanclp-simplify-preserves-unsat[of ψ ψ'] unfolding full-def by metis

lemma resolution-preserves-unsat:
  assumes resolution ψ ψ'
  shows satisfiable (fst ψ') ⟶ satisfiable (fst ψ)
  using assms apply (induct rule: resolution.induct)
  using full1-simplify-preserves-unsat apply (metis fst-conv)
  using full-simplify-preserves-unsat simplify-preserves-unsat by fastforce

lemma rtranclp-resolution-preserves-unsat:
  assumes resolution** ψ ψ'
  shows satisfiable (fst ψ') ⟶ satisfiable (fst ψ)
  using assms apply induction
  using resolution-preserves-unsat by fast+

lemma rtranclp-simplify-preserve-partial-tree:
  assumes simplify** N N'
  and partial-interps t I N
  shows partial-interps t I N'
  using assms apply (induction, simp)
  using simplify-preserve-partial-tree by metis

lemma full1-simplify-preserve-partial-tree:
  assumes full1 simplify N N'
  and partial-interps t I N
  shows partial-interps t I N'
  using assms rtranclp-simplify-preserve-partial-tree[of N N' t I] tranclp-into-rtranclp
  unfolding full1-def by fast

lemma full-simplify-preserve-partial-tree:
  assumes full simplify N N'
  and partial-interps t I N
  shows partial-interps t I N'
  using assms rtranclp-simplify-preserve-partial-tree[of N N' t I] tranclp-into-rtranclp
  unfolding full-def by fast

lemma resolution-preserve-partial-tree:
  assumes resolution S S'
  and partial-interps t I (fst S)
  shows partial-interps t I (fst S')
  using assms apply induction
```

```
    using full1-simplify-preserve-partial-tree fst-conv apply metis
  using full-simplify-preserve-partial-tree inference-preserve-partial-tree by fastforce

lemma rtranclp-resolution-preserve-partial-tree:
  assumes resolution** S S'
  and partial-interps t I (fst S)
  shows partial-interps t I (fst S')
  using assms apply induction
  using resolution-preserve-partial-tree by fast+
  thm nat-less-induct nat.induct

lemma nat-ge-induct[case-names 0 Suc]:
  assumes P 0
  and ⋀n. (⋀m. m<Suc n ⟹ P m) ⟹ P (Suc n)
  shows P n
  using assms apply (induct rule: nat-less-induct)
  by (rename-tac n, case-tac n) auto

lemma wf-always-more-step-False:
  assumes wf R
  shows (∀x. ∃z. (z, x)∈R) ⟹ False
 using assms unfolding wf-def by (meson Domain.DomainI assms wfE-min)

lemma finite-finite-mset-element-of-mset[simp]:
  assumes finite N
  shows finite {f φ L |φ L. φ ∈ N ∧ L ∈# φ ∧ P φ L}
  using assms
proof (induction N rule: finite-induct)
  case empty
  show ?case by auto
next
  case (insert x N) note finite = this(1) and IH = this(3)
  have {f φ L |φ L. (φ = x ∨ φ ∈ N) ∧ L ∈# φ ∧ P φ L} ⊆ {f x L | L. L ∈# x ∧ P x L}
    ∪ {f φ L |φ L. φ ∈ N ∧ L ∈# φ ∧ P φ L} by auto
  moreover have finite {f x L | L. L ∈# x} by auto
  ultimately show ?case using IH finite-subset by fastforce
qed


definition sum-count-ge-2 :: 'a multiset set ⇒ nat (Ξ) where
sum-count-ge-2 ≡ folding.F (λφ. op +(msetsum {#count φ L |L ∈# φ. 2 ≤ count φ L#})) 0


interpretation sum-count-ge-2:
  folding (λφ. op +(msetsum {#count φ L |L ∈# φ. 2 ≤ count φ L#})) 0
rewrites
  folding.F (λφ. op +(msetsum {#count φ L |L ∈# φ. 2 ≤ count φ L#})) 0 = sum-count-ge-2
proof −
  show folding (λφ. op + (msetsum (image-mset (count φ) {# L ∈# φ. 2 ≤ count φ L#})))
    by standard auto
  then interpret sum-count-ge-2:
    folding (λφ. op +(msetsum {#count φ L |L ∈# φ. 2 ≤ count φ L#})) 0 .
  show folding.F (λφ. op + (msetsum (image-mset (count φ) {# L ∈# φ. 2 ≤ count φ L#}))) 0
    = sum-count-ge-2 by (auto simp add: sum-count-ge-2-def)
qed
```

```
lemma finite-incl-le-setsum:
 finite (B::'a multiset set) ⟹ A ⊆ B ⟹ Ξ A ≤ Ξ B
proof (induction arbitrary:A rule: finite-induct)
  case empty
  then show ?case by simp
next
  case (insert a F) note finite = this(1) and aF = this(2) and IH = this(3) and AF = this(4)
  show ?case
    proof (cases a ∈ A)
      assume a ∉ A
      then have A ⊆ F using AF by auto
      then show ?case using IH[of A] by (simp add: aF local.finite)
    next
      assume aA: a ∈ A
      then have A − {a} ⊆ F using AF by auto
      then have Ξ (A − {a}) ≤ Ξ F using IH by blast
      then show ?case
        proof −
          obtain nn :: nat ⇒ nat ⇒ nat where
            ∀ x0 x1. (∃ v2. x0 = x1 + v2) = (x0 = x1 + nn x0 x1)
            by moura
          then have Ξ F = Ξ (A − {a}) + nn (Ξ F) (Ξ (A − {a}))
            by (meson ‹Ξ (A − {a}) ≤ Ξ F› le-iff-add)
          then show ?thesis
            by (metis (no-types) le-iff-add aA aF add.assoc finite.insertI finite-subset
              insert.prems local.finite sum-count-ge-2.insert sum-count-ge-2.remove)
        qed
    qed
qed

lemma simplify-finite-measure-decrease:
  simplify N N' ⟹ finite N ⟹ card N' + Ξ N' < card N + Ξ N
proof (induction rule: simplify.induct)
  case (tautology-deletion A P) note an = this(1) and fin = this(2)
  let ?N' = N − {A + {#Pos P#} + {#Neg P#}}
  have card ?N' < card N
    by (meson card-Diff1-less tautology-deletion.hyps tautology-deletion.prems)
  moreover have ?N' ⊆ N by auto
  then have sum-count-ge-2 ?N' ≤ sum-count-ge-2 N using finite-incl-le-setsum[OF fin] by blast
  ultimately show ?case by linarith
next
  case (condensation A L) note AN = this(1) and fin = this(2)
  let ?C' = A + {#L#}
  let ?C = A + {#L#} + {#L#}
  let ?N' = N − {?C} ∪ {?C'}
  have card ?N' ≤ card N
    using AN by (metis (no-types, lifting) Diff-subset Un-empty-right Un-insert-right card.remove
      card-insert-if card-mono fin finite-Diff order-refl)
  moreover have Ξ {?C'} < Ξ {?C}
    proof −
      have mset-decomp:
        {# La ∈# A. (L = La ⟶ La ∈# A) ∧ (L ≠ La ⟶ 2 ≤ count A La)#}
        = {# La ∈# A. L ≠ La ∧ 2 ≤ count A La#} +
          {# La ∈# A. L = La ∧ Suc 0 ≤ count A L#}
          by (auto simp: multiset-eq-iff ac-simps)
      have mset-decomp2: {# La ∈# A. L ≠ La ⟶ 2 ≤ count A La#} =
        {# La ∈# A. L ≠ La ∧ 2 ≤ count A La#} + replicate-mset (count A L) L
        by (auto simp: multiset-eq-iff)
      show ?thesis
        by (auto simp: mset-decomp mset-decomp2 filter-mset-eq ac-simps)
   qed
  have Ξ ?N' < Ξ N
    proof cases
      assume a1: ?C' ∈ N
      then show ?thesis
        proof −
          have f2: ⋀m M. insert (m::'a literal multiset) (M − {m}) = M ∪ {} ∨ m ∉ M
            using Un-empty-right insert-Diff by blast
          have f3: ⋀m M Ma. insert (m::'a literal multiset) M − insert m Ma = M − insert m Ma
            by simp
          then have f4: ⋀M m. M − {m::'a literal multiset} = M ∪ {} ∨ m ∈ M
            using Diff-insert-absorb Un-empty-right by fastforce
          have f5: insert (A + {#L#} + {#L#}) N = N
            using f3 f2 Un-empty-right condensation.hyps insert-iff by fastforce
          have ⋀m M. insert (m::'a literal multiset) M = M ∪ {} ∨ m ∉ M
            using f3 f2 Un-empty-right add.right-neutral insert-iff by fastforce
          then have Ξ (N − {A + {#L#} + {#L#}}) < Ξ N
            using f5 f4 by (metis Un-empty-right ‹Ξ {A + {#L#}} < Ξ {A + {#L#} + {#L#}}›
              add.right-neutral add-diff-cancel-left' add-gr-0 diff-less fin finite.emptyI not-le
              sum-count-ge-2.empty sum-count-ge-2.insert-remove trans-le-add2)
          then show ?thesis
            using f3 f2 a1 by (metis (no-types) Un-empty-right Un-insert-right condensation.hyps
              insert-iff multi-self-add-other-not-self)
        qed
    next
      assume ?C' ∉ N
      have mset-decomp:
        {# La ∈# A. (L = La ⟶ Suc 0 ≤ count A La) ∧ (L ≠ La ⟶ 2 ≤ count A La)#}
        = {# La ∈# A. L ≠ La ∧ 2 ≤ count A La#} +
          {# La ∈# A. L = La ∧ Suc 0 ≤ count A L#}
           by (auto simp: multiset-eq-iff ac-simps)
      have mset-decomp2: {# La ∈# A. L ≠ La ⟶ 2 ≤ count A La#} =
        {# La ∈# A. L ≠ La ∧ 2 ≤ count A La#} + replicate-mset (count A L) L
        by (auto simp: multiset-eq-iff)

      show ?thesis
        using ‹Ξ {A + {#L#}} < Ξ {A + {#L#} + {#L#}}› condensation.hyps fin
        sum-count-ge-2.remove[of - A + {#L#} + {#L#}] ‹?C' ∉ N›
        by (auto simp: mset-decomp mset-decomp2 filter-mset-eq)
    qed
  ultimately show ?case by linarith
next
  case (subsumption A B) note AN = this(1) and AB = this(2) and BN = this(3) and fin = this(4)
  have card (N − {B}) < card N using BN by (meson card-Diff1-less subsumption.prems)
  moreover have Ξ (N − {B}) ≤ Ξ N
    by (simp add: Diff-subset finite-incl-le-setsum subsumption.prems)
  ultimately show ?case by linarith
qed

lemma simplify-terminates:
  wf {(N', N). finite N ∧ simplify N N'}
  using assms apply (rule wfP-if-measure[of finite simplify λN. card N + Ξ N])
  using simplify-finite-measure-decrease by blast



lemma wf-terminates:
  assumes wf r
  shows ∃N'.(N', N)∈ r* ∧ (∀N''. (N'', N')∉ r)
proof −
  let ?P = λN. (∃N'.(N', N)∈ r* ∧ (∀N''. (N'', N')∉ r))
  have (∀x. (∀y. (y, x) ∈ r ⟶ ?P y) ⟶ ?P x)
    proof clarify
      fix x
      assume H: ∀y. (y, x) ∈ r ⟶ ?P y
      { assume ∃y. (y, x) ∈ r
        then obtain y where y: (y, x) ∈ r by blast
        then have ?P y using H by blast
        then have ?P x using y by (meson rtrancl.rtrancl-into-rtrancl)
      }
      moreover {
        assume ¬(∃y. (y, x) ∈ r)
        then have ?P x by auto
      }
      ultimately show ?P x by blast
    qed
  moreover have (∀x. (∀y. (y, x) ∈ r ⟶ ?P y) ⟶ ?P x) ⟶ All ?P
    using assms unfolding wf-def by (rule allE)
  ultimately have All ?P by blast
  then show ?P N by blast
qed

lemma rtranclp-simplify-terminates:
  assumes fin: finite N
  shows ∃N'. simplify** N N' ∧ simplified N'
proof −
  have H: {(N', N). finite N ∧ simplify N N'} = {(N', N). simplify N N' ∧ finite N} by auto
  then have wf: wf {(N', N). simplify N N' ∧ finite N}
    using simplify-terminates by (simp add: H)
  obtain N' where N': (N', N)∈ {(b, a). simplify a b ∧ finite a}* and
    more: (∀N''. (N'', N')∉ {(b, a). simplify a b ∧ finite a})
    using Prop-Resolution.wf-terminates[OF wf, of N] by blast
  have 1: simplify** N N'
    using N' by (induction rule: rtrancl.induct) auto
  then have finite N' using fin rtranclp-simplify-preserves-finite by blast
  then have 2: ∀N''. ¬simplify N' N'' using more by auto

  show ?thesis using 1 2 by blast
qed

lemma finite-simplified-full1-simp:
  assumes finite N
  shows simplified N ∨ (∃N'. full1 simplify N N')
  using rtranclp-simplify-terminates[OF assms] unfolding full1-def
  by (metis Nitpick.rtranclp-unfold)

lemma finite-simplified-full-simp:
  assumes finite N
  shows ∃ N'. full simplify N N'
  using rtranclp-simplify-terminates[OF assms] unfolding full-def by metis

lemma can-decrease-tree-size-resolution:
  fixes ψ :: 'v state and tree :: 'v sem-tree
  assumes finite (fst ψ) and already-used-inv ψ
  and partial-interps tree I (fst ψ)
  and simplified (fst ψ)
  shows ∃(tree':: 'v sem-tree) ψ'. resolution** ψ ψ' ∧ partial-interps tree' I (fst ψ')
    ∧ (sem-tree-size tree' < sem-tree-size tree ∨ sem-tree-size tree = 0)
  using assms
proof (induct arbitrary: I rule: sem-tree-size)
  case (bigger xs I) note IH = this(1) and finite = this(2) and a-u-i = this(3) and part = this(4)
    and simp = this(5)

  { assume sem-tree-size xs = 0
    then have ?case using part by blast
  }

  moreover {
    assume sn0: sem-tree-size xs > 0
    obtain ag ad v where xs: xs = Node v ag ad using sn0 by (cases xs, auto)
    {
       assume sem-tree-size ag = 0 ∧ sem-tree-size ad = 0
       then have ag: ag = Leaf and ad: ad = Leaf by (cases ag, auto, cases ad, auto)

       then obtain χ χ' where
         χ: ¬ I ∪ {Pos v} ⊨ χ and
         totχ: total-over-m (I ∪ {Pos v}) {χ} and
         χψ: χ ∈ fst ψ and
         χ': ¬ I ∪ {Neg v} ⊨ χ'  and
         totχ': total-over-m (I ∪ {Neg v}) {χ'} and χ'ψ: χ' ∈ fst ψ
         using part unfolding xs by auto
       have Posv: Pos v ∉# χ using χ unfolding true-cls-def true-lit-def by auto
       have Negv: Neg v ∉# χ' using χ' unfolding true-cls-def true-lit-def by auto
       {
         assume Negχ: Neg v ∉# χ
         then have ¬ I ⊨ χ using χ Posv unfolding true-cls-def true-lit-def by auto
         moreover have total-over-m I {χ}
           using Posv Negχ atm-imp-pos-or-neg-lit totχ unfolding total-over-m-def total-over-set-def
           by fastforce
         ultimately have partial-interps Leaf I (fst ψ)
           and sem-tree-size Leaf < sem-tree-size xs
           and resolution** ψ ψ
           unfolding xs by (auto simp add: χψ)
       }
       moreover {
          assume Posχ: Pos v ∉# χ'
          then have Iχ: ¬ I ⊨ χ' using χ' Posv unfolding true-cls-def true-lit-def by auto
          moreover have total-over-m I {χ'}
            using Negv Posχ atm-imp-pos-or-neg-lit totχ'
            unfolding total-over-m-def total-over-set-def by fastforce
          ultimately have  partial-interps Leaf I (fst ψ)
            and sem-tree-size Leaf < sem-tree-size xs
            and resolution** ψ ψ
            using χ'ψ Iχ unfolding xs by auto
```

```
}
moreover {
  assume neg: Neg v ∈# χ and pos: Pos v ∈# χ'
  have count χ (Neg v) = 1
    using simplified-count[OF simp χψ] neg
    by (simp add: dual-order.antisym)
  have count χ' (Pos v) = 1
    using simplified-count[OF simp χ'ψ] pos
    by (simp add: dual-order.antisym)

  obtain C where χC: χ = C + {#Neg v#} and negC: Neg v ∉# C and posC: Pos v ∉# C
    by (metis (no-types, lifting) One-nat-def Posv Suc-eq-plus1-left ‹count χ (Neg v) = 1›
      add-diff-cancel-left' count-diff count-greater-eq-one-iff count-single insert-DiffM
      insert-DiffM2 less-numeral-extra(3) multi-member-skip not-le not-less-eq-eq)

  obtain C' where
    χC': χ' = C' + {#Pos v#} and
    posC': Pos v ∉# C' and
    negC': Neg v ∉# C'
    by (metis (no-types, lifting) One-nat-def Negv Suc-eq-plus1-left ‹count χ' (Pos v) = 1›
      add-diff-cancel-left' count-diff count-greater-eq-one-iff count-single insert-DiffM
      insert-DiffM2 less-numeral-extra(3) multi-member-skip not-le not-less-eq-eq)

  have totC: total-over-m I {C}
    using totχ tot-over-m-remove[of I Pos v C] negC posC unfolding χC
    by (metis total-over-m-sum uminus-Neg uminus-of-uminus-id)
  have totC': total-over-m I {C'}
    using totχ' total-over-m-sum tot-over-m-remove[of I Neg v C'] negC' posC'
    unfolding χC' by (metis total-over-m-sum uminus-Neg)
  have ¬ I ⊨ C + C'
    using χ χ' χC χC' by auto
  then have part-I-ψ''': partial-interps Leaf I (fst ψ ∪ {C + C'})
    using totC totC' ‹¬ I ⊨ C + C'› by (metis Un-insert-right insertI1
      partial-interps.simps(1) total-over-m-sum)
  {
    assume ({#Pos v#} + C', {#Neg v#} + C) ∉ snd ψ
    then have inf'': inference ψ (fst ψ ∪ {C + C'}, snd ψ ∪ {(χ', χ)})
      by (metis χ'ψ χC χC' χψ add.commute inference-step prod.collapse resolution)
    obtain N' where full: full simplify (fst ψ ∪ {C + C'}) N'
      by (metis finite-simplified-full-simp fst-conv inf'' inference-preserves-finite
        local.finite)
    have resolution ψ (N', snd ψ ∪ {(χ', χ)})
      using resolution.intros(2)[OF - simp full, of snd ψ snd ψ ∪ {(χ', χ)}] inf''
      by (metis surjective-pairing)
    moreover have partial-interps Leaf I N'
      using full-simplify-preserve-partial-tree[OF full part-I-ψ'''] .
    moreover have sem-tree-size Leaf < sem-tree-size xs unfolding xs by auto
    ultimately have ?case
      by (metis (no-types) prod.sel(1) rtranclp.rtrancl-into-rtrancl rtranclp.rtrancl-refl)
  }
  moreover {
    assume a: ({#Pos v#} + C', {#Neg v#} + C) ∈ snd ψ
    then have (∃χ ∈ fst ψ. (∀I. total-over-m I {C+C'} ⟶ total-over-m I {χ})
        ∧ (∀I. total-over-m I {χ} ⟶ I ⊨ χ ⟶ I ⊨ C' + C)) ∨ tautology (C' + C)
      proof –
        obtain p where p: Pos p ∈# ({#Pos v#} + C') ∧ Neg p ∈# ({#Neg v#} + C)
```

```
                  ∧((∃χ∈fst ψ. (∀I. total-over-m I {({#Pos v#} + C′) − {#Pos p#} + (({#Neg v#}
+ C) − {#Neg p#})} ⟶ total-over-m I {χ}) ∧ (∀I. total-over-m I {χ} ⟶ I ⊨ χ ⟶ I ⊨ ({#Pos
v#} + C′) − {#Pos p#} + (({#Neg v#} + C) − {#Neg p#}))) ∨ tautology (({#Pos v#} + C′) −
{#Pos p#} + (({#Neg v#} + C) − {#Neg p#})))
                  using a by (blast intro: allE[OF a-u-i[unfolded subsumes-def Ball-def],
                    of ({#Pos v#} + C′, {#Neg v#} + C)])
                { assume p ≠ v
                  then have Pos p ∈# C′ ∧ Neg p ∈# C using p by force
                  then have ?thesis by auto
                }
                moreover {
                  assume p = v
                 then have ?thesis using p by (metis add.commute add-diff-cancel-left′)
                }
                ultimately show ?thesis by auto
              qed
            moreover {
              assume ∃χ ∈ fst ψ. (∀I. total-over-m I {C+C′} ⟶ total-over-m I {χ})
                ∧ (∀I. total-over-m I {χ} ⟶ I ⊨ χ ⟶ I ⊨ C′ + C)
              then obtain ϑ where
                ϑ: ϑ ∈ fst ψ and
                tot-ϑ-CC′: ∀I. total-over-m I {C+C′} ⟶ total-over-m I {ϑ} and
                ϑ-inv: ∀I. total-over-m I {ϑ} ⟶ I ⊨ ϑ ⟶ I ⊨ C′ + C by blast
              have partial-interps Leaf I (fst ψ)
                using tot-ϑ-CC′ ϑ ϑ-inv totC totC′ ‹¬ I ⊨ C + C′› total-over-m-sum by fastforce
              moreover have sem-tree-size Leaf < sem-tree-size xs unfolding xs by auto
              ultimately have ?case by blast
            }
            moreover {
              assume tautCC′: tautology (C′ + C)
              have total-over-m I {C′+C} using totC totC′ total-over-m-sum by auto
              then have ¬tautology (C′ + C)
                using ‹¬ I ⊨ C + C′› unfolding add.commute[of C C′] total-over-m-def
                unfolding tautology-def by auto
              then have False using tautCC′  unfolding tautology-def by auto
            }
            ultimately have ?case by auto
          }
          ultimately have ?case by auto
        }
        ultimately have ?case using part by (metis (no-types) sem-tree-size.simps(1))
      }
      moreover {
        assume size-ag: sem-tree-size ag > 0
        have sem-tree-size ag < sem-tree-size xs unfolding xs by auto
        moreover have partial-interps ag (I ∪ {Pos v}) (fst ψ)
        and partad: partial-interps ad (I ∪ {Neg v}) (fst ψ)
          using part partial-interps.simps(2) unfolding xs by metis+
        moreover
          have sem-tree-size ag < sem-tree-size xs ⟹ finite (fst ψ) ⟹ already-used-inv ψ
            ⟹ partial-interps ag (I ∪ {Pos v}) (fst ψ) ⟹ simplified (fst ψ)
            ⟹ ∃tree′ ψ′. resolution** ψ ψ′ ∧ partial-interps tree′ (I ∪ {Pos v}) (fst ψ′)
                ∧ (sem-tree-size tree′ < sem-tree-size ag ∨ sem-tree-size ag = 0)
            using IH[of ag I ∪ {Pos v}]  by auto
        ultimately obtain ψ′ :: ′v state and  tree′ :: ′v sem-tree  where
          inf: resolution** ψ ψ′
          and part: partial-interps tree′ (I ∪ {Pos v}) (fst ψ′)
          and size: sem-tree-size tree′ < sem-tree-size ag ∨ sem-tree-size ag = 0
          using finite part rtranclp.rtrancl-refl a-u-i simp by blast

        have partial-interps ad (I ∪ {Neg v}) (fst ψ′)
          using rtranclp-resolution-preserve-partial-tree inf partad by fast
        then have partial-interps (Node v tree′ ad) I (fst ψ′) using part by auto
        then have ?case using inf size size-ag part unfolding xs by fastforce
      }
      moreover {
        assume size-ad: sem-tree-size ad > 0
        have sem-tree-size ad < sem-tree-size xs unfolding xs by auto
        moreover
          have
            partag: partial-interps ag (I ∪ {Pos v}) (fst ψ) and
            partial-interps ad (I ∪ {Neg v}) (fst ψ)
            using part partial-interps.simps(2) unfolding xs by metis+
        moreover have sem-tree-size ad < sem-tree-size xs ⟶ finite (fst ψ) ⟶ already-used-inv ψ
          ⟶ ( partial-interps ad (I ∪ {Neg v}) (fst ψ) ⟶ simplified (fst ψ)
          ⟶ (∃ tree′ ψ′. resolution** ψ ψ′ ∧ partial-interps tree′ (I ∪ {Neg v}) (fst ψ′)
              ∧ (sem-tree-size tree′ < sem-tree-size ad ∨ sem-tree-size ad = 0)))
          using IH by blast
        ultimately obtain ψ′ :: ′v state and  tree′ :: ′v sem-tree  where
          inf: resolution** ψ ψ′
          and part: partial-interps tree′ (I ∪ {Neg v}) (fst ψ′)
          and size: sem-tree-size tree′ < sem-tree-size ad ∨ sem-tree-size ad = 0
          using finite part  rtranclp.rtrancl-refl a-u-i simp by blast

        have partial-interps ag (I ∪ {Pos v}) (fst ψ′)
          using rtranclp-resolution-preserve-partial-tree inf partag by fast
        then have partial-interps (Node v ag tree′) I (fst ψ′) using part by auto
        then have ?case using inf size size-ad unfolding xs by fastforce
      }
       ultimately have ?case by auto
    }
    ultimately show ?case by auto
  qed

  lemma resolution-completeness-inv:
    fixes ψ :: ′v ::linorder state
    assumes
      unsat: ¬satisfiable (fst ψ) and
      finite: finite (fst ψ) and
      a-u-v: already-used-inv ψ
    shows ∃ψ′. (resolution** ψ ψ′ ∧ {#} ∈ fst ψ′)
  proof −
    obtain tree where  partial-interps tree {} (fst ψ)
      using partial-interps-build-sem-tree-atms assms by metis
    then show ?thesis
      using unsat finite a-u-v
      proof (induct tree arbitrary: ψ rule: sem-tree-size)
        case (bigger tree ψ) note H = this
        {
          fix χ
          assume tree: tree = Leaf
          obtain χ where χ: ¬ {} ⊨ χ and totχ: total-over-m {} {χ} and χψ: χ ∈ fst ψ
            using H unfolding tree by auto
          moreover have {#} = χ
            using H atms-empty-iff-empty totχ
            unfolding true-cls-def total-over-m-def total-over-set-def by fastforce
          moreover have resolution** ψ ψ by auto
          ultimately have ?case by metis
        }
        moreover {
          fix v tree1 tree2
          assume tree: tree = Node v tree1 tree2
          obtain ψ0 where ψ0: resolution** ψ ψ0 and simp: simplified (fst ψ0)
            proof −
              { assume simplified (fst ψ)
              moreover have resolution** ψ ψ by auto
              ultimately have thesis using that by blast
              }
              moreover {
              assume ¬simplified (fst ψ)
              then have ∃ψ'. full1 simplify (fst ψ) ψ'
                by (metis Nitpick.rtranclp-unfold bigger.prems(3) full1-def
                  rtranclp-simplify-terminates)
              then obtain N where full1 simplify (fst ψ) N by metis
              then have resolution ψ (N, snd ψ)
                using resolution.intros(1)[of fst ψ N snd ψ] by auto
              moreover have simplified N
                using ‹full1 simplify (fst ψ) N› unfolding full1-def by blast
              ultimately have ?thesis using that by force
              }
              ultimately show ?thesis by auto
            qed


          have p: partial-interps tree {} (fst ψ0)
          and uns: unsatisfiable (fst ψ0)
          and f: finite (fst ψ0)
          and a-u-v: already-used-inv ψ0
               using ψ0 bigger.prems(1) rtranclp-resolution-preserve-partial-tree apply blast
              using ψ0 bigger.prems(2) rtranclp-resolution-preserves-unsat apply blast
             using ψ0 bigger.prems(3) rtranclp-resolution-finite apply blast
            using rtranclp-resolution-already-used-inv[OF ψ0 bigger.prems(4)] by blast
          obtain tree' ψ' where
            inf: resolution** ψ0 ψ' and
            part': partial-interps tree' {} (fst ψ') and
            decrease: sem-tree-size tree' < sem-tree-size tree ∨ sem-tree-size tree = 0
            using can-decrease-tree-size-resolution[OF f a-u-v p simp] unfolding tautology-def
            by meson
          have s: sem-tree-size tree' < sem-tree-size tree using decrease unfolding tree by auto
          have fin: finite (fst ψ')
            using f inf rtranclp-resolution-finite by blast
          have unsat: unsatisfiable (fst ψ')
            using rtranclp-resolution-preserves-unsat inf uns by metis
          have a-u-i': already-used-inv ψ'
            using a-u-v inf rtranclp-resolution-already-used-inv[of ψ0 ψ'] by auto
          have ?case
            using inf rtranclp-trans[of resolution] H(1)[OF s part' unsat fin a-u-i'] ψ0 by blast
        }
```

```
      ultimately show ?case by (cases tree, auto)
    qed
qed

lemma resolution-preserves-already-used-inv:
  assumes resolution S S'
  and already-used-inv S
  shows already-used-inv S'
  using assms
  apply (induct rule: resolution.induct)
   apply (rule full1-simplify-already-used-inv; simp)
  apply (rule full-simplify-already-used-inv, simp)
  apply (rule inference-preserves-already-used-inv, simp)
  apply blast
  done

lemma rtranclp-resolution-preserves-already-used-inv:
  assumes resolution** S S'
  and already-used-inv S
  shows already-used-inv S'
  using assms
  apply (induct rule: rtranclp-induct)
   apply simp
  using resolution-preserves-already-used-inv by fast

lemma resolution-completeness:
  fixes ψ :: 'v ::linorder state
  assumes unsat: ¬satisfiable (fst ψ)
  and finite: finite (fst ψ)
  and snd ψ = {}
  shows ∃ψ'. (resolution** ψ ψ' ∧ {#} ∈ fst ψ')
proof −
  have already-used-inv ψ unfolding assms by auto
  then show ?thesis using assms resolution-completeness-inv by blast
qed

lemma rtranclp-preserves-sat:
  assumes simplify** S S'
  and satisfiable S
  shows satisfiable S'
  using assms apply induction
   apply simp
  by (meson satisfiable-carac satisfiable-def simplify-preserves-un-sat-eq)

lemma resolution-preserves-sat:
  assumes resolution S S'
  and satisfiable (fst S)
  shows satisfiable (fst S')
  using assms apply (induction rule: resolution.induct)
   using rtranclp-preserves-sat tranclp-into-rtranclp unfolding full1-def apply fastforce
  by (metis fst-conv full-def inference-preserves-un-sat rtranclp-preserves-sat
    satisfiable-carac' satisfiable-def)

lemma rtranclp-resolution-preserves-sat:
  assumes resolution** S S'
  and satisfiable (fst S)
  shows satisfiable (fst S′)
  using assms apply (induction rule: rtranclp-induct)
   apply simp
  using resolution-preserves-sat by blast

lemma resolution-soundness:
  fixes ψ :: ′v ::linorder state
  assumes resolution** ψ ψ′ and {#} ∈ fst ψ′
  shows unsatisfiable (fst ψ)
  using assms by (meson rtranclp-resolution-preserves-sat satisfiable-def true-cls-empty
    true-clss-def)

lemma resolution-soundness-and-completeness:
fixes ψ :: ′v ::linorder state
assumes finite: finite (fst ψ)
and snd: snd ψ = {}
shows (∃ψ′. (resolution** ψ ψ′ ∧ {#} ∈ fst ψ′)) ⟷ unsatisfiable (fst ψ)
  using assms resolution-completeness resolution-soundness by metis

lemma simplified-falsity:
  assumes simp: simplified ψ
  and {#} ∈ ψ
  shows ψ = {{#}}
proof (rule ccontr)
  assume H: ¬ ?thesis
  then obtain χ where χ ∈ ψ and χ ≠ {#} using assms(2) by blast
  then have {#} ⊂# χ by (simp add: mset-less-empty-nonempty)
  then have simplify ψ (ψ − {χ})
    using simplify.subsumption[OF assms(2) ‹{#} ⊂# χ› ‹χ ∈ ψ›] by blast
  then show False using simp by blast
qed


lemma simplify-falsity-in-preserved:
  assumes simplify χs χs′
  and {#} ∈ χs
  shows {#} ∈ χs′
  using assms
  by induction auto

lemma rtranclp-simplify-falsity-in-preserved:
  assumes simplify** χs χs′
  and {#} ∈ χs
  shows {#} ∈ χs′
  using assms
  by induction (auto intro: simplify-falsity-in-preserved)

lemma resolution-falsity-get-falsity-alone:
  assumes finite (fst ψ)
  shows (∃ψ′. (resolution** ψ ψ′ ∧ {#} ∈ fst ψ′)) ⟷ (∃a-u-v. resolution** ψ ({{#}}, a-u-v))
    (is ?A ⟷ ?B)
proof
  assume ?B
  then show ?A by auto
next
  assume ?A
  then obtain χs a-u-v where χs: resolution** ψ (χs, a-u-v) and F: {#} ∈ χs by auto
  { assume simplified χs
    then have ?B using simplified-falsity[OF - F] χs by blast
  }
  moreover {
    assume ¬ simplified χs
    then obtain χs' where full1 simplify χs χs'
       by (metis χs assms finite-simplified-full1-simp fst-conv rtranclp-resolution-finite)
    then have {#} ∈ χs'
      unfolding full1-def by (meson F rtranclp-simplify-falsity-in-preserved
        tranclp-into-rtranclp)
    then have ?B
      by (metis χs ‹full1 simplify χs χs'› fst-conv full1-simp resolution-always-simplified
        rtranclp.rtrancl-into-rtrancl simplified-falsity)
  }
  ultimately show ?B by blast
qed

lemma resolution-soundness-and-completeness':
  fixes ψ :: 'v ::linorder state
  assumes
    finite: finite (fst ψ)and
    snd: snd ψ = {}
  shows (∃ a-u-v. (resolution** ψ ({{#}}, a-u-v))) ⟷ unsatisfiable (fst ψ)
    using assms resolution-completeness resolution-soundness resolution-falsity-get-falsity-alone
    by metis

end
theory Prop-Superposition
imports Partial-Clausal-Logic ../lib/Herbrand-Interpretation
begin
```

1.2 Superposition

```
no-notation Herbrand-Interpretation.true-cls (infix ⊨ 50)
notation Herbrand-Interpretation.true-cls (infix ⊨h 50)

no-notation Herbrand-Interpretation.true-clss (infix ⊨s 50)
notation Herbrand-Interpretation.true-clss (infix ⊨hs 50)

lemma herbrand-interp-iff-partial-interp-cls:
  S ⊨h C ⟷ {Pos P|P. P∈S} ∪ {Neg P|P. P∉S} ⊨ C
  unfolding Herbrand-Interpretation.true-cls-def Partial-Clausal-Logic.true-cls-def
  by auto

lemma herbrand-consistent-interp:
  consistent-interp ({Pos P|P. P∈S} ∪ {Neg P|P. P∉S})
  unfolding consistent-interp-def by auto

lemma herbrand-total-over-set:
  total-over-set ({Pos P|P. P∈S} ∪ {Neg P|P. P∉S}) T
  unfolding total-over-set-def by auto

lemma herbrand-total-over-m:
  total-over-m ({Pos P|P. P∈S} ∪ {Neg P|P. P∉S}) T
  unfolding total-over-m-def by (auto simp add: herbrand-total-over-set)

lemma herbrand-interp-iff-partial-interp-clss:
  S ⊨hs C ⟷ {Pos P|P. P∈S} ∪ {Neg P|P. P∉S} ⊨s C
  unfolding true-clss-def Ball-def herbrand-interp-iff-partial-interp-cls
  Partial-Clausal-Logic.true-clss-def by auto

definition clss-lt :: 'a::wellorder clauses ⇒ 'a clause ⇒ 'a clauses  where
clss-lt N C = {D ∈ N. D #⊂# C}

notation (latex output)
 clss-lt (-<^bsup>-<^esup>)

locale selection =
  fixes S :: 'a clause ⇒ 'a clause
  assumes
    S-selects-subseteq: ⋀C. S C ≤# C and
    S-selects-neg-lits: ⋀C L. L ∈# S C ⟹ is-neg L

locale ground-resolution-with-selection =
  selection S for S :: ('a :: wellorder) clause ⇒ 'a clause
begin

context
  fixes N :: 'a clause set
begin
```

We do not create an equivalent of δ, but we directly defined N_C by inlining the definition.

```
function
  production :: 'a clause ⇒ 'a interp
where
  production C =
   {A. C ∈ N ∧ C ≠ {#} ∧ Max (set-mset C) = Pos A ∧ count C (Pos A) ≤ 1
     ∧ ¬ (⋃D ∈ {D. D #⊂# C}. production D) ⊨h C ∧ S C = {#}}
  by auto
termination by (relation {(D, C). D #⊂# C}) (auto simp: wf-less-multiset)

declare production.simps[simp del]

definition interp :: 'a clause ⇒ 'a interp where
  interp C = (⋃D ∈ {D. D #⊂# C}. production D)

lemma production-unfold:
  production C = {A. C ∈ N ∧ C ≠ {#} ∧ Max (set-mset C) = Pos A∧ count C (Pos A) ≤ 1 ∧ ¬
interp C ⊨h C ∧ S C = {#}}
  unfolding interp-def by (rule production.simps)

abbreviation productive A ≡ (production A ≠ {})

abbreviation produces :: 'a clause ⇒ 'a ⇒ bool where
  produces C A ≡ production C = {A}

lemma producesD:
  produces C A ⟹ C ∈ N ∧ C ≠ {#} ∧ Pos A = Max (set-mset C) ∧ count C (Pos A) ≤ 1 ∧
    ¬ interp C ⊨h C ∧ S C = {#}
  unfolding production-unfold by auto
```

```
lemma produces C A ⟹ Pos A ∈# C
  by (simp add: Max-in-lits producesD)

lemma interp′-def-in-set:
  interp C = (⋃D ∈ {D ∈ N. D #⊂# C}. production D)
  unfolding interp-def apply auto
  unfolding production-unfold apply auto
  done

lemma production-iff-produces:
  produces D A ⟷ A ∈ production D
  unfolding production-unfold by auto

definition Interp :: ′a clause ⇒ ′a interp where
  Interp C = interp C ∪ production C

lemma
  assumes produces C P
  shows Interp C ⊨h C
  unfolding Interp-def assms using producesD[OF assms]
  by (metis Max-in-lits Un-insert-right insertI1 pos-literal-in-imp-true-cls)

definition INTERP :: ′a interp where
INTERP = (⋃D ∈N. production D)


lemma interp-subseteq-Interp[simp]: interp C ⊆ Interp C
  unfolding Interp-def by simp

lemma Interp-as-UNION: Interp C = (⋃D ∈ {D. D #⊆# C}. production D)
  unfolding Interp-def interp-def le-multiset-def by fast

lemma productive-not-empty: productive C ⟹ C ≠ {#}
  unfolding production-unfold by auto

lemma productive-imp-produces-Max-literal: productive C ⟹ produces C (atm-of (Max (set-mset C)))
  unfolding production-unfold by (auto simp del: atm-of-Max-lit)

lemma productive-imp-produces-Max-atom: productive C ⟹ produces C (Max (atms-of C))
  unfolding atms-of-def Max-atm-of-set-mset-commute[OF productive-not-empty]
  by (rule productive-imp-produces-Max-literal)

lemma produces-imp-Max-literal: produces C A ⟹ A = atm-of (Max (set-mset C))
  by (metis Max-singleton insert-not-empty productive-imp-produces-Max-literal)

lemma produces-imp-Max-atom: produces C A ⟹ A = Max (atms-of C)
  by (metis Max-singleton insert-not-empty productive-imp-produces-Max-atom)

lemma produces-imp-Pos-in-lits: produces C A ⟹ Pos A ∈# C
  by (auto intro: Max-in-lits dest!: producesD)

lemma productive-in-N: productive C ⟹ C ∈ N
  unfolding production-unfold by auto

lemma produces-imp-atms-leq: produces C A ⟹ B ∈ atms-of C ⟹ B ≤ A
```

```
  by (metis Max-ge finite-atms-of insert-not-empty productive-imp-produces-Max-atom
    singleton-inject)

lemma produces-imp-neg-notin-lits: produces C A ⟹ Neg A ∉# C
  by (rule pos-Max-imp-neg-notin) (auto dest: producesD)

lemma less-eq-imp-interp-subseteq-interp: C #⊆# D ⟹ interp C ⊆ interp D
  unfolding interp-def by auto (metis multiset-order.order.strict-trans2)

lemma less-eq-imp-interp-subseteq-Interp: C #⊆# D ⟹ interp C ⊆ Interp D
  unfolding Interp-def using less-eq-imp-interp-subseteq-interp by blast

lemma less-imp-production-subseteq-interp: C #⊂# D ⟹ production C ⊆ interp D
  unfolding interp-def by fast

lemma less-eq-imp-production-subseteq-Interp: C #⊆# D ⟹ production C ⊆ Interp D
  unfolding Interp-def using less-imp-production-subseteq-interp
  by (metis multiset-order.le-imp-less-or-eq le-supI1 sup-ge2)

lemma less-imp-Interp-subseteq-interp: C #⊂# D ⟹ Interp C ⊆ interp D
  unfolding Interp-def
  by (auto simp: less-eq-imp-interp-subseteq-interp less-imp-production-subseteq-interp)

lemma less-eq-imp-Interp-subseteq-Interp: C #⊆# D ⟹ Interp C ⊆ Interp D
  using less-imp-Interp-subseteq-interp
  unfolding Interp-def by (metis multiset-order.le-imp-less-or-eq le-supI2 subset-refl sup-commute)

lemma false-Interp-to-true-interp-imp-less-multiset: A ∉ Interp C ⟹ A ∈ interp D ⟹ C #⊂# D
  using less-eq-imp-interp-subseteq-Interp multiset-linorder.not-less by blast

lemma false-interp-to-true-interp-imp-less-multiset: A ∉ interp C ⟹ A ∈ interp D ⟹ C #⊂# D
  using less-eq-imp-interp-subseteq-interp multiset-linorder.not-less by blast

lemma false-Interp-to-true-Interp-imp-less-multiset: A ∉ Interp C ⟹ A ∈ Interp D ⟹ C #⊂# D
  using less-eq-imp-Interp-subseteq-Interp multiset-linorder.not-less by blast

lemma false-interp-to-true-Interp-imp-le-multiset: A ∉ interp C ⟹ A ∈ Interp D ⟹ C #⊆# D
  using less-imp-Interp-subseteq-interp multiset-linorder.not-less by blast

lemma interp-subseteq-INTERP: interp C ⊆ INTERP
  unfolding interp-def INTERP-def by (auto simp: production-unfold)

lemma production-subseteq-INTERP: production C ⊆ INTERP
  unfolding INTERP-def using production-unfold by blast

lemma Interp-subseteq-INTERP: Interp C ⊆ INTERP
  unfolding Interp-def by (auto intro!: interp-subseteq-INTERP production-subseteq-INTERP)
```

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book.

```
lemma produces-imp-in-interp:
  assumes a-in-c: Neg A ∈# C and d: produces D A
  shows A ∈ interp C
proof −
  from d have Max (set-mset D) = Pos A
    using production-unfold by blast
  then have D #⊂# {#Neg A#}
```

```
    by (auto intro: Max-pos-neg-less-multiset)
  moreover have {#Neg A#} #⊆# C
    by (rule less-eq-imp-le-multiset) (rule mset-le-single[OF a-in-c])
  ultimately show ?thesis
    using d by (blast dest: less-eq-imp-interp-subseteq-interp less-imp-production-subseteq-interp)
qed

lemma neg-notin-Interp-not-produce: Neg A ∈# C ⟹ A ∉ Interp D ⟹ C #⊆# D ⟹ ¬ produces
D'' A
  by (auto dest: produces-imp-in-interp less-eq-imp-interp-subseteq-Interp)

lemma in-production-imp-produces: A ∈ production C ⟹ produces C A
  by (metis insert-absorb productive-imp-produces-Max-atom singleton-insert-inj-eq')

lemma not-produces-imp-notin-production: ¬ produces C A ⟹ A ∉ production C
  by (metis in-production-imp-produces)

lemma not-produces-imp-notin-interp: (⋀D. ¬ produces D A) ⟹ A ∉ interp C
  unfolding interp-def by (fast intro!: in-production-imp-produces)
```

The results below corresponds to Lemma 3.4.

Nitpicking: If $D = D'$ and D is productive, $I^D \subseteq I_{D'}$ does not hold.

```
lemma true-Interp-imp-general:
  assumes
    c-le-d: C #⊆# D and
    d-lt-d': D #⊂# D' and
    c-at-d: Interp D ⊨h C and
    subs: interp D' ⊆ (⋃C ∈ CC. production C)
  shows (⋃C ∈ CC. production C) ⊨h C
proof (cases ∃A. Pos A ∈# C ∧ A ∈ Interp D)
  case True
  then obtain A where a-in-c: Pos A ∈# C and a-at-d: A ∈ Interp D
    by blast
  from a-at-d have A ∈ interp D'
    using d-lt-d' less-imp-Interp-subseteq-interp by blast
  then show ?thesis
    using subs a-in-c by (blast dest: contra-subsetD)
next
  case False
  then obtain A where a-in-c: Neg A ∈# C and A ∉ Interp D
    using c-at-d unfolding true-cls-def by blast
  then have ⋀D''. ¬ produces D'' A
    using c-le-d neg-notin-Interp-not-produce by simp
  then show ?thesis
    using a-in-c subs not-produces-imp-notin-production by auto
qed

lemma true-Interp-imp-interp: C #⊆# D ⟹ D #⊂# D' ⟹ Interp D ⊨h C ⟹ interp D' ⊨h C
  using interp-def true-Interp-imp-general by simp

lemma true-Interp-imp-Interp: C #⊆# D ⟹ D #⊂# D' ⟹ Interp D ⊨h C ⟹ Interp D' ⊨h C
  using Interp-as-UNION interp-subseteq-Interp true-Interp-imp-general by simp

lemma true-Interp-imp-INTERP: C #⊆# D ⟹ Interp D ⊨h C ⟹ INTERP ⊨h C
  using INTERP-def interp-subseteq-INTERP
```

```
    true-Interp-imp-general[OF - less-multiset-right-total]
  by simp

lemma true-interp-imp-general:
  assumes
    c-le-d: C #⊆# D and
    d-lt-d': D #⊂# D' and
    c-at-d: interp D ⊨h C and
    subs: interp D' ⊆ (⋃C ∈ CC. production C)
  shows (⋃C ∈ CC. production C) ⊨h C
proof (cases ∃A. Pos A ∈# C ∧ A ∈ interp D)
  case True
  then obtain A where a-in-c: Pos A ∈# C and a-at-d: A ∈ interp D
    by blast
  from a-at-d have A ∈ interp D'
    using d-lt-d' less-eq-imp-interp-subseteq-interp[OF multiset-order.less-imp-le] by blast
  then show ?thesis
    using subs a-in-c by (blast dest: contra-subsetD)
next
  case False
  then obtain A where a-in-c: Neg A ∈# C and A ∉ interp D
    using c-at-d unfolding true-cls-def by blast
  then have ⋀D''. ¬ produces D'' A
    using c-le-d by (auto dest: produces-imp-in-interp less-eq-imp-interp-subseteq-interp)
  then show ?thesis
    using a-in-c subs not-produces-imp-notin-production by auto
qed
```

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book. Here the strict maximality is important

```
lemma true-interp-imp-interp: C #⊆# D ⟹ D #⊂# D' ⟹ interp D ⊨h C ⟹ interp D' ⊨h C
  using interp-def true-interp-imp-general by simp

lemma true-interp-imp-Interp: C #⊆# D ⟹ D #⊂# D' ⟹ interp D ⊨h C ⟹ Interp D' ⊨h C
  using Interp-as-UNION interp-subseteq-Interp[of D'] true-interp-imp-general by simp

lemma true-interp-imp-INTERP: C #⊆# D ⟹ interp D ⊨h C ⟹ INTERP ⊨h C
  using INTERP-def interp-subseteq-INTERP
    true-interp-imp-general[OF - less-multiset-right-total]
  by simp

lemma productive-imp-false-interp: productive C ⟹ ¬ interp C ⊨h C
  unfolding production-unfold by auto
```

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book. Here the strict maximality is important

```
lemma cls-gt-double-pos-no-production:
  assumes D: {#Pos P, Pos P#} #⊂# C
  shows ¬produces C P
proof -
  let ?D = {#Pos P, Pos P#}
  note D' = D[unfolded less-multiset_HO]
  consider
    (P) count C (Pos P) ≥ 2
  | (Q) Q where Q > Pos P and Q ∈# C
```

```
    using HOL.spec[OF HOL.conjunct2[OF D'], of Pos P] by (auto split: if-split-asm)
  then show ?thesis
    proof cases
      case Q
      have Q ∈ set-mset C
        using Q(2) by (auto split: if-split-asm)
      then have Max (set-mset C) > Pos P
        using Q(1) Max-gr-iff by blast
      then show ?thesis
        unfolding production-unfold by auto
    next
      case P
      then show ?thesis
        unfolding production-unfold by auto
    qed
qed
```

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book.

```
lemma
  assumes D: C+{#Neg P#} #⊂# D
  shows production D ≠ {P}
proof −
  note D' = D[unfolded less-multiset_HO]
  consider
    (P) Neg P ∈# D
  | (Q) Q where Q > Neg P and count D Q > count (C + {#Neg P#}) Q
    using HOL.spec[OF HOL.conjunct2[OF D'], of Neg P] count-greater-zero-iff by fastforce
  then show ?thesis
    proof cases
      case Q
      have Q ∈ set-mset D
        using Q(2) gr-implies-not0 by fastforce
      then have Max (set-mset D) > Neg P
        using Q(1) Max-gr-iff by blast
      then have Max (set-mset D) > Pos P
        using less-trans[of Pos P Neg P Max (set-mset D)] by auto
      then show ?thesis
        unfolding production-unfold by auto
    next
      case P
      then have Max (set-mset D) > Pos P
        by (meson Max-ge finite-set-mset le-less-trans linorder-not-le pos-less-neg)
      then show ?thesis
        unfolding production-unfold by auto
    qed
qed

lemma in-interp-is-produced:
  assumes P ∈ INTERP
  shows ∃D. D +{#Pos P#} ∈ N ∧ produces (D +{#Pos P#}) P
  using assms unfolding INTERP-def UN-iff production-iff-produces Ball-def
  by (metis ground-resolution-with-selection.produces-imp-Pos-in-lits insert-DiffM2
    ground-resolution-with-selection-axioms not-produces-imp-notin-production)


end
```

```
end

abbreviation MMax M ≡ Max (set-mset M)
```

1.2.1 We can now define the rules of the calculus

```
inductive superposition-rules :: 'a clause ⇒ 'a clause ⇒ 'a clause ⇒ bool where
factoring: superposition-rules (C + {#Pos P#} + {#Pos P#}) B (C + {#Pos P#}) |
superposition-l: superposition-rules (C₁ + {#Pos P#}) (C₂ + {#Neg P#}) (C₁+ C₂)

inductive superposition :: 'a clauses ⇒ 'a clauses ⇒ bool where
superposition: A ∈ N ⟹ B ∈ N ⟹ superposition-rules A B C
  ⟹ superposition N (N ∪ {C})

definition abstract-red :: 'a::wellorder clause ⇒ 'a clauses ⇒ bool where
abstract-red C N = (clss-lt N C ⊨p C)

lemma less-multiset[iff]: M < N ⟷ M #⊂# N
  unfolding less-multiset-def by auto

lemma less-eq-multiset[iff]: M ≤ N ⟷ M #⊆# N
  unfolding less-eq-multiset-def by auto

lemma herbrand-true-clss-true-clss-cls-herbrand-true-clss:
  assumes
    AB: A ⊨hs B and
    BC: B ⊨p C
  shows A ⊨h C
proof −
  let ?I = {Pos P |P. P ∈ A} ∪ {Neg P |P. P ∉ A}
  have B: ?I ⊨s B using AB
    by (auto simp add: herbrand-interp-iff-partial-interp-clss)

  have IH: ⋀I. total-over-set I (atms-of C) ⟹ total-over-m I B ⟹ consistent-interp I
    ⟹ I ⊨s B ⟹ I ⊨ C using BC
    by (auto simp add: true-clss-cls-def)
  show ?thesis
    unfolding herbrand-interp-iff-partial-interp-cls
    by (auto intro: IH[of ?I] simp add: herbrand-total-over-set herbrand-total-over-m
      herbrand-consistent-interp B)
qed

lemma abstract-red-subset-mset-abstract-red:
  assumes
    abstr: abstract-red C N and
    c-lt-d: C ⊆# D
  shows abstract-red D N
proof −
  have {D ∈ N. D #⊂# C} ⊆ {D' ∈ N. D' #⊂# D}
    using c-lt-d less-eq-imp-le-multiset by fastforce
  then show ?thesis
    using abstr unfolding abstract-red-def clss-lt-def
    by (metis (no-types, lifting) c-lt-d subset-mset.diff-add true-clss-cls-mono-r'
      true-clss-cls-subset)
qed
```

```
lemma true-clss-cls-extended:
  assumes
    A ⊨p B and
    tot: total-over-m I A and
    cons: consistent-interp I and
    I-A: I ⊨s A
  shows I ⊨ B
proof −
  let ?I = I ∪ {Pos P|P. P ∈ atms-of B ∧ P ∉ atms-of-s I}
  have consistent-interp ?I
    using cons unfolding consistent-interp-def atms-of-s-def atms-of-def
      apply (auto 1 5 simp add: image-iff)
    by (metis atm-of-uminus literal.sel(1))
  moreover have total-over-m ?I (A ∪ {B})
    proof −
      obtain aa :: 'a set ⇒ 'a literal set ⇒ 'a where
        f2: ∀x0 x1. (∃v2. v2 ∈ x0 ∧ Pos v2 ∉ x1 ∧ Neg v2 ∉ x1)
          ⟷ (aa x0 x1 ∈ x0 ∧ Pos (aa x0 x1) ∉ x1 ∧ Neg (aa x0 x1) ∉ x1)
        by moura
      have ∀a. a ∉ atms-of-ms A ∨ Pos a ∈ I ∨ Neg a ∈ I
        using tot by (simp add: total-over-m-def total-over-set-def)
      then have aa (atms-of-ms A ∪ atms-of-ms {B}) (I ∪ {Pos a |a. a ∈ atms-of B ∧ a ∉ atms-of-s
I})
        ∉ atms-of-ms A ∪ atms-of-ms {B} ∨ Pos (aa (atms-of-ms A ∪ atms-of-ms {B})
          (I ∪ {Pos a |a. a ∈ atms-of B ∧ a ∉ atms-of-s I})) ∈ I
            ∪ {Pos a |a. a ∈ atms-of B ∧ a ∉ atms-of-s I}
          ∨ Neg (aa (atms-of-ms A ∪ atms-of-ms {B})
            (I ∪ {Pos a |a. a ∈ atms-of B ∧ a ∉ atms-of-s I})) ∈ I
            ∪ {Pos a |a. a ∈ atms-of B ∧ a ∉ atms-of-s I}
        by auto
      then have total-over-set (I ∪ {Pos a |a. a ∈ atms-of B ∧ a ∉ atms-of-s I})
        (atms-of-ms A ∪ atms-of-ms {B})
        using f2 by (meson total-over-set-def)
      then show ?thesis
        by (simp add: total-over-m-def)
    qed
  moreover have ?I ⊨s A
    using I-A by auto
  ultimately have ?I ⊨ B
    using ‹A⊨pB› unfolding true-clss-cls-def by auto
  then show ?thesis
oops
lemma
  assumes
    CP: ¬ clss-lt N ({#C#} + {#E#}) ⊨p {#C#} + {#Neg P#} and
     clss-lt N ({#C#} + {#E#}) ⊨p {#E#} + {#Pos P#} ∨ clss-lt N ({#C#} + {#E#}) ⊨p
{#C#} + {#Neg P#}
  shows clss-lt N ({#C#} + {#E#}) ⊨p {#E#} + {#Pos P#}

oops

locale ground-ordered-resolution-with-redundancy =
  ground-resolution-with-selection +
  fixes redundant :: 'a::wellorder clause ⇒ 'a clauses ⇒ bool
  assumes
```

```
    redundant-iff-abstract: redundant A N ⟷ abstract-red A N
begin
definition saturated :: 'a clauses ⇒ bool where
saturated N ⟷ (∀ A B C. A ∈ N ⟶ B ∈ N ⟶ ¬redundant A N ⟶ ¬redundant B N
  ⟶ superposition-rules A B C ⟶ redundant C N ∨ C ∈ N)

lemma
  assumes
    saturated: saturated N and
    finite: finite N and
    empty: {#} ∉ N
  shows INTERP N ⊨hs N
proof (rule ccontr)
  let ?N𝓘 = INTERP N
  assume ¬ ?thesis
  then have not-empty: {E∈N. ¬?N𝓘 ⊨h E} ≠ {}
    unfolding true-clss-def Ball-def by auto
  def D ≡ Min {E∈N. ¬?N𝓘 ⊨h E}
  have [simp]: D ∈ N
    unfolding D-def
    by (metis (mono-tags, lifting) Min-in not-empty finite mem-Collect-eq rev-finite-subset subsetI)
  have not-d-interp: ¬?N𝓘 ⊨h D
    unfolding D-def
    by (metis (mono-tags, lifting) Min-in finite mem-Collect-eq not-empty rev-finite-subset subsetI)
  have cls-not-D: ⋀E. E ∈ N ⟹ E ≠ D ⟹ ¬?N𝓘 ⊨h E ⟹ D ≤ E
    using finite D-def by (auto simp del: less-eq-multiset)
  obtain C L where D: D = C + {#L#} and LSD: L ∈# S D ∨ (S D = {#} ∧ Max (set-mset D)
= L)
    proof (cases S D = {#})
      case False
      then obtain L where L ∈# S D
        using Max-in-lits by blast
      moreover
        then have L ∈# D
          using S-selects-subseteq[of D] by auto
        then have D = (D − {#L#}) + {#L#}
          by auto
      ultimately show ?thesis using that by blast
    next
      let ?L = MMax D
      case True
      moreover
        have ?L ∈# D
          by (metis (no-types, lifting) Max-in-lits ‹D ∈ N› empty)
        then have D = (D − {#?L#}) + {#?L#}
          by auto
      ultimately show ?thesis using that by blast
    qed
  have red: ¬redundant D N
    proof (rule ccontr)
      assume red[simplified]: ~~redundant D N
      have ∀E < D. E ∈ N ⟶ ?N𝓘 ⊨h E
        using cls-not-D not-le by fastforce
      then have ?N𝓘 ⊨hs clss-lt N D
        unfolding clss-lt-def true-clss-def Ball-def by blast
      then show False
        using  red not-d-interp unfolding abstract-red-def redundant-iff-abstract
        using herbrand-true-clss-true-clss-cls-herbrand-true-clss by fast
    qed

  consider
    (L) P where L = Pos P and S D = {#} and Max (set-mset D) = Pos P
  | (Lneg) P where L = Neg P
    using LSD S-selects-neg-lits[of L D] by (cases L) auto
  then show False
    proof cases
      case L note P = this(1) and S = this(2) and max = this(3)
      have count D L > 1
        proof (rule ccontr)
          assume ~ ?thesis
          then have count: count D L = 1
            unfolding D by (auto simp: not-in-iff)
          have ¬?N𝓘⊨h D
            using not-d-interp true-interp-imp-INTERP ground-resolution-with-selection-axioms
              by blast
          then have produces N D P
            using not-empty empty finite ‹D ∈ N› count L
              true-interp-imp-INTERP unfolding production-iff-produces unfolding production-unfold
            by (auto simp add: max not-empty)
          then have INTERP N ⊨h D
            unfolding D
            by (metis pos-literal-in-imp-true-cls produces-imp-Pos-in-lits
              production-subseteq-INTERP singletonI subsetCE)
          then show False
            using not-d-interp by blast
        qed
      then have Pos P ∈# C
        by (simp add: P D)
      then obtain C' where C':D = C' + {#Pos P#} + {#Pos P#}
        unfolding D by (metis (full-types) P insert-DiffM2)
      have sup: superposition-rules D D (D − {#L#})
        unfolding C' L by (auto simp add: superposition-rules.simps)
      have C' + {#Pos P#}  #⊂# C' + {#Pos P#} + {#Pos P#}
        by auto
      moreover have ¬?N𝓘 ⊨h (D − {#L#})
        using not-d-interp unfolding C' L by auto
      ultimately have C' + {#Pos P#} ∉ N
        by (metis (no-types, lifting) C' P add-diff-cancel-right' cls-not-D less-multiset
          multi-self-add-other-not-self not-le)
      have D − {#L#} #⊂# D
        unfolding C' L by auto
      have c'-p-p: C' + {#Pos P#} + {#Pos P#} − {#Pos P#} = C' + {#Pos P#}
        by auto
      have redundant (C' + {#Pos P#}) N
        using saturated red sup ‹D ∈ N›‹C' + {#Pos P#} ∉ N›  unfolding saturated-def C' L c'-p-p
        by blast
      moreover have C' + {#Pos P#}  ⊆# C' + {#Pos P#} + {#Pos P#}
        by auto
      ultimately show False
        using red unfolding C' redundant-iff-abstract by (blast dest:
          abstract-red-subset-mset-abstract-red)
    next
```

```
case Lneg note L = this(1)
have P ∈ ?N𝓘
  using not-d-interp unfolding D true-cls-def L by (auto split: if-split-asm)
then obtain E where
  DPN: E + {#Pos P#} ∈ N and
  prod: production N (E + {#Pos P#}) = {P}
  using in-interp-is-produced by blast
have sup-EC: superposition-rules (E + {#Pos P#}) (C + {#Neg P#}) (E + C)
  using superposition-l by fast
then have superposition N (N ∪ {E+C})
  using DPN ‹D ∈ N› unfolding D L by (auto simp add: superposition.simps)
have
  PMax: Pos P = MMax (E + {#Pos P#}) and
  count (E + {#Pos P#}) (Pos P) ≤ 1 and
  S (E + {#Pos P#}) = {#} and
  ¬interp N (E + {#Pos P#}) ⊨h E + {#Pos P#}
  using prod unfolding production-unfold by auto
have Neg P ∉# E
  using prod produces-imp-neg-notin-lits by force
then have ⋀y. y ∈# (E + {#Pos P#})
  ⟹ count (E + {#Pos P#}) (Neg P) < count (C + {#Neg P#}) (Neg P)
  using count-greater-zero-iff by fastforce
moreover have  ⋀y. y ∈# (E + {#Pos P#}) ⟹ y < Neg P
  using PMax by (metis DPN Max-less-iff empty finite-set-mset pos-less-neg
    set-mset-eq-empty-iff)
moreover have E + {#Pos P#} ≠ C + {#Neg P#}
  using prod produces-imp-neg-notin-lits by force
ultimately have E + {#Pos P#} #⊂# C + {#Neg P#}
  unfolding less-multiset_HO by (metis count-greater-zero-iff less-iff-Suc-add zero-less-Suc)
have ce-lt-d: C + E #⊂# D
unfolding D L by (simp add: ‹⋀y. y ∈# E + {#Pos P#} ⟹ y < Neg P› ex-gt-imp-less-multiset)
have ?N𝓘 ⊨h E + {#Pos P#}
  using ‹P ∈ ?N𝓘› by blast
have ?N𝓘 ⊨h C+E ∨ C+E ∉ N
  using ce-lt-d cls-not-D unfolding D-def by fastforce
have Pos P ∉# C+E
  using D ‹P ∈ ground-resolution-with-selection.INTERP S N›
    ‹count (E + {#Pos P#}) (Pos P) ≤ 1› multi-member-skip not-d-interp
    by (auto simp: not-in-iff)
then have ⋀y. y ∈# C+E
  ⟹ count (C+E) (Pos P) < count (E + {#Pos P#}) (Pos P)
  using set-mset-def by fastforce

have ¬redundant (C + E) N
  proof (rule ccontr)
    assume red'[simplified]: ¬ ?thesis
    have abs: clss-lt N (C + E) ⊨p C + E
      using redundant-iff-abstract red' unfolding abstract-red-def by auto
    have clss-lt N (C + E) ⊨p E + {#Pos P#} ∨ clss-lt N (C + E) ⊨p C + {#Neg P#}
      proof clarify
        assume CP: ¬ clss-lt N (C + E) ⊨p C + {#Neg P#}
        { fix I
          assume
            total-over-m I (clss-lt N (C + E) ∪ {E + {#Pos P#}}) and
            consistent-interp I and
            I ⊨s clss-lt N (C + E)
```

```
                  then have I ⊨ C + E
                    using  abs sorry
                  moreover have ¬ I ⊨ C + {#Neg P#}
                    using CP unfolding true-clss-cls-def
                    sorry
                  ultimately have  I ⊨ E + {#Pos P#} by auto
              }
              then show clss-lt N (C + E) ⊨p E + {#Pos P#}
                unfolding true-clss-cls-def by auto
            qed
          moreover have clss-lt N (C + E) ⊆ clss-lt N (C + {#Neg P#})
            using ce-lt-d mult-less-trans unfolding clss-lt-def D L by force
          ultimately have redundant (C + {#Neg P#}) N ∨ clss-lt N (C + E) ⊨p E + {#Pos P#}
            unfolding redundant-iff-abstract abstract-red-def using true-clss-cls-subset by blast
          show False sorry
        qed
      moreover have ¬ redundant (E + {#Pos P#}) N
        sorry
      ultimately have CEN: C + E ∈ N
        using ‹D∈N› ‹E + {#Pos P#}∈N› saturated sup-EC red unfolding saturated-def D L
        by (metis union-commute)
      have CED: C + E ≠ D
        using D ce-lt-d by auto
      have interp: ¬ INTERP N ⊨h C + E
      sorry
      show False
         using cls-not-D[OF CEN CED interp] ce-lt-d unfolding INTERP-def less-eq-multiset-def by
auto
  qed
qed

end

lemma tautology-is-redundant:
  assumes tautology C
  shows abstract-red C N
  using assms unfolding abstract-red-def true-clss-cls-def tautology-def by auto

lemma subsumed-is-redundant:
  assumes AB: A ⊂# B
  and AN: A ∈ N
  shows abstract-red B N
proof –
  have A ∈ clss-lt N B using AN AB unfolding clss-lt-def
    by (auto dest: less-eq-imp-le-multiset simp add: multiset-order.dual-order.order-iff-strict)
  then show ?thesis
    using AB unfolding abstract-red-def true-clss-cls-def Partial-Clausal-Logic.true-clss-def
    by blast
qed

inductive redundant :: 'a clause ⇒ 'a clauses ⇒ bool where
subsumption: A ∈ N ⟹ A ⊂# B ⟹ redundant B N

lemma redundant-is-redundancy-criterion:
  fixes A :: 'a :: wellorder clause and N :: 'a :: wellorder clauses
  assumes redundant A N
  shows  abstract-red A N
  using assms
proof (induction rule: redundant.induct)
  case (subsumption A B N)
  then show ?case
    using subsumed-is-redundant[of A N B] unfolding abstract-red-def clss-lt-def by auto
qed

lemma redundant-mono:
  redundant A N ⟹ A ⊆# B ⟹  redundant B N
  apply (induction rule: redundant.induct)
  by (meson subset-mset.less-le-trans subsumption)

locale truc =
    selection S for S :: nat clause ⇒ nat clause
begin

end

end
```