Technical University of Denmark

Formalization of Bachmair and Ganzinger's Ordered Resolution Prover

Schlichtkrull, Anders; Blanchette, Jasmin Christian; Traytel, Dmitriy; Waldmann, Uwe

Published in:
Archive of Formal Proofs

Publication date:
2018

Document Version
Publisher's PDF, also known as Version of record

Formalization of Bachmair and Ganzinger's Ordered Resolution Prover

Anders Schlichtkrull, Jasmin Christian Blanchette, Dmitriy Traytel, and Uwe Waldmann

January 22, 2018

Abstract

This Isabelle/HOL formalization covers Sections 2 to 4 of Bachmair and Ganzinger's "Resolution Theorem Proving" chapter in the *Handbook of Automated Reasoning*. This includes soundness and completeness of unordered and ordered variants of ground resolution with and without literal selection, the standard redundancy criterion, a general framework for refutational theorem proving, and soundness and completeness of an abstract first-order prover.

Contents

1 Introduction

Bachmair and Ganzinger's "Resolution Theorem Proving" chapter in the *Handbook of Automated Reasoning* is the standard reference on the topic. It defines a general framework for propositional and first-order resolution-based theorem proving. Resolution forms the basis for superposition, the calculus implemented in many popular automatic theorem provers.

This Isabelle/HOL formalization covers Sections 2.1, 2.2, 2.4, 2.5, 3, 4.1, 4.2, and 4.3 of Bachmair and Ganzinger's chapter. Section 2 focuses on preliminaries. Section 3 introduces unordered and ordered variants of ground resolution with and without literal selection and proves them refutationally complete. Section 4.1 presents a framework for theorem provers based on refutation and saturation. Finally, Section 4.2 generalizes the refutational completeness argument and introduces the standard redundancy criterion, which can be used in conjunction with ordered resolution. Section 4.3 lifts the result to a first-order prover, specified as a calculus. Figure 1 shows the corresponding Isabelle theory structure.

2 Map Function on Two Parallel Lists

theory *Map2*
 imports *Main*
begin

This theory defines a map function that applies a (curried) binary function elementwise to two parallel lists.

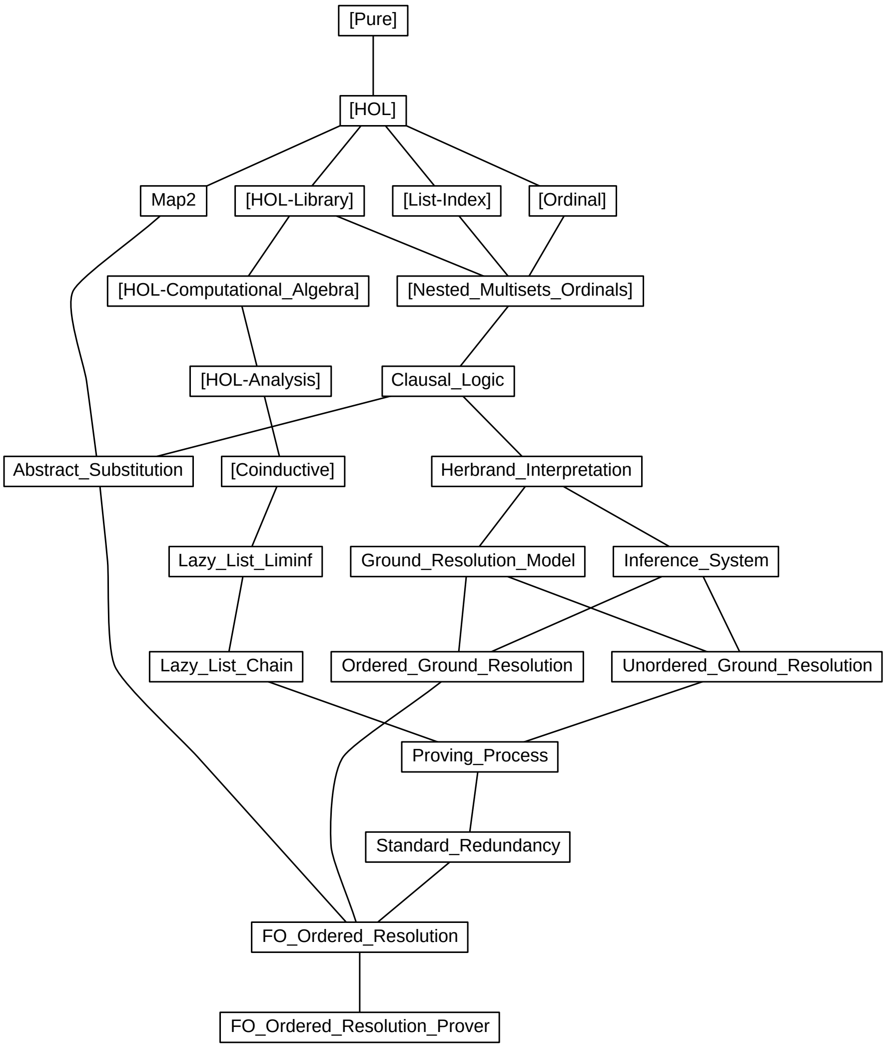

Figure 1: Theory dependency graph

The definition is taken from https://www.isa-afp.org/browser_info/current/AFP/Jinja/Listn.html.

abbreviation *map2* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow 'c\ list$ **where**
$map2\ f\ xs\ ys \equiv map\ (case\_prod\ f)\ (zip\ xs\ ys)$

lemma *map2_empty_iff* [*simp*]: $map2\ f\ xs\ ys = [] \longleftrightarrow xs = [] \lor ys = []$
by (*metis Nil_is_map_conv list.exhaust list.simps*(*3*) *zip.simps*(*1*) *zip_Cons_Cons zip_Nil*)

lemma *image_map2*: $length\ t = length\ s \Longrightarrow g\ `\ set\ (map2\ f\ t\ s) = set\ (map2\ (\lambda a\ b.\ g\ (f\ a\ b))\ t\ s)$
by *auto*

lemma *map2_tl*: $length\ t = length\ s \Longrightarrow map2\ f\ (tl\ t)\ (tl\ s) = tl\ (map2\ f\ t\ s)$
by (*metis* (*no_types*, *lifting*) *hd_Cons_tl list.sel*(*3*) *map2_empty_iff map_tl tl_Nil zip_Cons_Cons*)

lemma *map_zip_assoc*:
$map\ f\ (zip\ (zip\ xs\ ys)\ zs) = map\ (\lambda(x,\ y,\ z).\ f\ ((x,\ y),\ z))\ (zip\ xs\ (zip\ ys\ zs))$
by (*induct zs arbitrary*: *xs ys*) (*auto simp add*: *zip.simps*(*2*) *split*: *list.splits*)

lemma *set_map2_ex*:
assumes $length\ t = length\ s$
shows $set\ (map2\ f\ s\ t) = \{x.\ \exists\, i < length\ t.\ x = f\ (s\ !\ i)\ (t\ !\ i)\}$
proof (*rule*; *rule*)
fix x
assume $x \in set\ (map2\ f\ s\ t)$
then obtain i **where** *i_p*: $i < length\ (map2\ f\ s\ t) \land x = map2\ f\ s\ t\ !\ i$
by (*metis in_set_conv_nth*)
from *i_p* **have** $i < length\ t$
by *auto*
moreover from *this i_p* **have** $x = f\ (s\ !\ i)\ (t\ !\ i)$
using *assms* **by** *auto*
ultimately show $x \in \{x.\ \exists\, i < length\ t.\ x = f\ (s\ !\ i)\ (t\ !\ i)\}$
using *assms* **by** *auto*
next
fix x
assume $x \in \{x.\ \exists\, i < length\ t.\ x = f\ (s\ !\ i)\ (t\ !\ i)\}$
then obtain i **where** *i_p*: $i < length\ t \land x = f\ (s\ !\ i)\ (t\ !\ i)$
by *auto*
then have $i < length\ (map2\ f\ s\ t)$
using *assms* **by** *auto*
moreover from *i_p* **have** $x = map2\ f\ s\ t\ !\ i$
using *assms* **by** *auto*
ultimately show $x \in set\ (map2\ f\ s\ t)$
by (*metis in_set_conv_nth*)
qed

end

3 Liminf of Lazy Lists

theory *Lazy_List_Liminf*
imports *Coinductive.Coinductive_List*
begin

Lazy lists, as defined in the *Archive of Formal Proofs*, provide finite and infinite lists in one type, defined coinductively. The present theory introduces the concept of the union of all elements of a lazy list of sets and the limit of such a lazy list. The definitions are stated more generally in terms of lattices. The basis for this theory is Section 4.1 ("Theorem Proving Processes") of Bachmair and Ganzinger's chapter.

definition *Sup_llist* :: $'a\ set\ llist \Rightarrow 'a\ set$ **where**
$Sup\_llist\ Xs = (\bigcup i \in \{i.\ enat\ i < llength\ Xs\}.\ lnth\ Xs\ i)$

lemma *lnth_subset_Sup_llist*: $enat\ i < llength\ xs \Longrightarrow lnth\ xs\ i \subseteq Sup\_llist\ xs$
unfolding *Sup_llist_def* **by** *auto*

lemma Sup_llist_LNil[simp]: Sup_llist LNil $= \{\}$
  unfolding Sup_llist_def **by** auto

lemma Sup_llist_LCons[simp]: Sup_llist (LCons X Xs) $=$ X $\cup$ Sup_llist Xs
  unfolding Sup_llist_def
proof (intro subset_antisym subsetI)
  fix x
  assume $x \in (\bigcup i \in \{i.$ enat $i <$ llength (LCons X Xs)$\}.$ lnth (LCons X Xs) $i)$
  then obtain i **where** len: enat $i <$ llength (LCons X Xs) **and** nth: $x \in$ lnth (LCons X Xs) i
    by blast
  from nth **have** $x \in X \vee i > 0 \wedge x \in$ lnth Xs $(i - 1)$
    by (metis lnth_LCons′ neq0_conv)
  then have $x \in X \vee (\exists i.$ enat $i <$ llength Xs $\wedge\ x \in$ lnth Xs $i)$
    by (metis len Suc_pred′ eSuc_enat iless_Suc_eq less_irrefl llength_LCons not_less order_trans)
  then show $x \in X \cup (\bigcup i \in \{i.$ enat $i <$ llength Xs$\}.$ lnth Xs $i)$
    by blast
qed ((auto)[], metis i0_lb lnth_0 zero_enat_def, metis Suc_ile_eq lnth_Suc_LCons)

lemma lhd_subset_Sup_llist: $\neg$ lnull Xs $\Longrightarrow$ lhd Xs $\subseteq$ Sup_llist Xs
  by (cases Xs) simp_all

definition Sup_upto_llist :: ′a set llist $\Rightarrow$ nat $\Rightarrow$ ′a set **where**
  Sup_upto_llist Xs $j = (\bigcup i \in \{i.$ enat $i <$ llength Xs $\wedge\ i \le j\}.$ lnth Xs $i)$

lemma Sup_upto_llist_mono: $j \le k \Longrightarrow$ Sup_upto_llist Xs $j \subseteq$ Sup_upto_llist Xs k
  unfolding Sup_upto_llist_def **by** auto

lemma Sup_upto_llist_subset_Sup_llist: $j \le k \Longrightarrow$ Sup_upto_llist Xs $j \subseteq$ Sup_llist Xs
  unfolding Sup_llist_def Sup_upto_llist_def **by** auto

lemma elem_Sup_llist_imp_Sup_upto_llist: $x \in$ Sup_llist Xs $\Longrightarrow \exists j.\ x \in$ Sup_upto_llist Xs j
  unfolding Sup_llist_def Sup_upto_llist_def **by** blast

lemma finite_Sup_llist_imp_Sup_upto_llist:
  assumes finite X **and** $X \subseteq$ Sup_llist Xs
  shows $\exists k.\ X \subseteq$ Sup_upto_llist Xs k
  using assms
proof induct
  case (insert x X)
  then have x: $x \in$ Sup_llist Xs **and** X: $X \subseteq$ Sup_llist Xs
    by simp+
  from x **obtain** k **where** k: $x \in$ Sup_upto_llist Xs k
    using elem_Sup_llist_imp_Sup_upto_llist **by** fast
  from X **obtain** k' **where** k': $X \subseteq$ Sup_upto_llist Xs k'
    using insert.hyps(3) **by** fast
  have insert x X $\subseteq$ Sup_upto_llist Xs (max k k')
    using k k'
    by (metis insert_absorb insert_subset Sup_upto_llist_mono max.cobounded2 max.commute
      order.trans)
  then show ?case
    by fast
qed simp

definition Liminf_llist :: ′a set llist $\Rightarrow$ ′a set **where**
  Liminf_llist Xs $=$
   $(\bigcup i \in \{i.$ enat $i <$ llength Xs$\}.\ \bigcap j \in \{j.\ i \le j \wedge$ enat $j <$ llength Xs$\}.$ lnth Xs $j)$

lemma Liminf_llist_subset_Sup_llist: Liminf_llist Xs $\subseteq$ Sup_llist Xs
  unfolding Liminf_llist_def Sup_llist_def **by** fast

lemma Liminf_llist_LNil[simp]: Liminf_llist LNil $= \{\}$
  unfolding Liminf_llist_def **by** simp

```
lemma Liminf_llist_LCons:
  Liminf_llist (LCons X Xs) = (if lnull Xs then X else Liminf_llist Xs) (is ?lhs = ?rhs)
proof (cases lnull Xs)
  case nnull: False
  show ?thesis
  proof
    {
      fix x
      assume ∃ i. enat i ≤ llength Xs
        ∧ (∀ j. i ≤ j ∧ enat j ≤ llength Xs ⟶ x ∈ lnth (LCons X Xs) j)
      then have ∃ i. enat (Suc i) ≤ llength Xs
        ∧ (∀ j. Suc i ≤ j ∧ enat j ≤ llength Xs ⟶ x ∈ lnth (LCons X Xs) j)
        by (cases llength Xs,
            metis not_lnull_conv[THEN iffD1, OF nnull] Suc_le_D eSuc_enat eSuc_ile_mono
              llength_LCons not_less_eq_eq zero_enat_def zero_le,
            metis Suc_leD enat_ord_code(3))
      then have ∃ i. enat i < llength Xs ∧ (∀ j. i ≤ j ∧ enat j < llength Xs ⟶ x ∈ lnth Xs j)
        by (metis Suc_ile_eq Suc_n_not_le_n lift_Suc_mono_le lnth_Suc_LCons nat_le_linear)
    }
    then show ?lhs ⊆ ?rhs
      by (simp add: Liminf_llist_def nnull) (rule subsetI, simp)

    {
      fix x
      assume ∃ i. enat i < llength Xs ∧ (∀ j. i ≤ j ∧ enat j < llength Xs ⟶ x ∈ lnth Xs j)
      then obtain i where
        i: enat i < llength Xs and
        j: ∀ j. i ≤ j ∧ enat j < llength Xs ⟶ x ∈ lnth Xs j
        by blast

      have enat (Suc i) ≤ llength Xs
        using i by (simp add: Suc_ile_eq)
      moreover have ∀ j. Suc i ≤ j ∧ enat j ≤ llength Xs ⟶ x ∈ lnth (LCons X Xs) j
        using Suc_ile_eq Suc_le_D j by force
      ultimately have ∃ i. enat i ≤ llength Xs ∧ (∀ j. i ≤ j ∧ enat j ≤ llength Xs ⟶
        x ∈ lnth (LCons X Xs) j)
        by blast
    }
    then show ?rhs ⊆ ?lhs
      by (simp add: Liminf_llist_def nnull) (rule subsetI, simp)
  qed
qed (simp add: Liminf_llist_def enat_0_iff(1))

lemma lfinite_Liminf_llist: lfinite Xs ⟹ Liminf_llist Xs = (if lnull Xs then {} else llast Xs)
proof (induction rule: lfinite_induct)
  case (LCons xs)
  then obtain y ys where
    xs: xs = LCons y ys
    by (meson not_lnull_conv)
  show ?case
    unfolding xs by (simp add: Liminf_llist_LCons LCons.IH[unfolded xs, simplified] llast_LCons)
qed (simp add: Liminf_llist_def)

lemma Liminf_llist_ltl: ¬ lnull (ltl Xs) ⟹ Liminf_llist Xs = Liminf_llist (ltl Xs)
  by (metis Liminf_llist_LCons lhd_LCons_ltl lnull_ltlI)

end
```

4 Relational Chains over Lazy Lists

```
theory Lazy_List_Chain
  imports HOL−Library.BNF_Corec Lazy_List_Liminf
begin
```

A chain is a lazy lists of elements such that all pairs of consecutive elements are related by a given relation. A full chain is either an infinite chain or a finite chain that cannot be extended. The inspiration for this theory is Section 4.1 ("Theorem Proving Processes") of Bachmair and Ganzinger's chapter.

4.1 Chains

```
coinductive chain :: ('a ⇒ 'a ⇒ bool) ⇒ 'a llist ⇒ bool for R :: 'a ⇒ 'a ⇒ bool where
 chain_singleton: chain R (LCons x LNil)
| chain_cons: chain R xs ⟹ R x (lhd xs) ⟹ chain R (LCons x xs)

lemma
  chain_LNil[simp]: ¬ chain R LNil and
  chain_not_lnull: chain R xs ⟹ ¬ lnull xs
  by (auto elim: chain.cases)

lemma chain_lappend:
  assumes
    r_xs: chain R xs and
    r_ys: chain R ys and
    mid: R (llast xs) (lhd ys)
  shows chain R (lappend xs ys)
proof (cases lfinite xs)
  case True
  then show ?thesis
    using r_xs mid
  proof (induct rule: lfinite.induct)
    case (lfinite_LConsI xs x)
    note fin = this(1) and ih = this(2) and r_xxs = this(3) and mid = this(4)
    show ?case
    proof (cases xs = LNil)
      case True
      then show ?thesis
        using r_ys mid by simp (rule chain_cons)
    next
      case xs_nnil: False
      have r_xs: chain R xs
        by (metis chain.simps ltl_simps(2) r_xxs xs_nnil)
      have mid': R (llast xs) (lhd ys)
        by (metis llast_LCons lnull_def mid xs_nnil)
      have start: R x (lhd (lappend xs ys))
        by (metis (no_types) chain.simps lhd_LCons lhd_lappend chain_not_lnull ltl_simps(2) r_xxs
           xs_nnil)
      show ?thesis
        unfolding lappend_code(2) using ih[OF r_xs mid'] start by (rule chain_cons)
    qed
  qed simp
qed (simp add: r_xs lappend_inf)

lemma chain_length_pos: chain R xs ⟹ llength xs > 0
  by (cases xs) simp+

lemma chain_ldropn:
  assumes chain R xs and enat n < llength xs
  shows chain R (ldropn n xs)
  using assms
  by (induct n arbitrary: xs, simp,
     metis chain.cases ldrop_eSuc_ltl ldropn_LNil ldropn_eq_LNil ltl_simps(2) not_less)

lemma chain_lnth_rel:
  assumes
    chain: chain R xs and
    len: enat (Suc j) < llength xs
  shows R (lnth xs j) (lnth xs (Suc j))
```

```
proof −
  define ys where ys = ldropn j xs
  have llength ys > 1
    unfolding ys_def using len
    by (metis One_nat_def funpow_swap1 ldropn_0 ldropn_def ldropn_eq_LNil ldropn_ltl not_less
        one_enat_def)
  obtain y0 y1 ys′ where
    ys: ys = LCons y0 (LCons y1 ys′)
    unfolding ys_def by (metis Suc_ile_eq ldropn_Suc_conv_ldropn len less_imp_not_less not_less)
  have chain R ys
    unfolding ys_def using Suc_ile_eq chain chain_ldropn len less_imp_le by blast
  then have R y0 y1
    unfolding ys by (auto elim: chain.cases)
  then show ?thesis
    using ys_def unfolding ys by (metis ldropn_Suc_conv_ldropn ldropn_eq_LConsD llist.inject)
qed

lemma infinite_chain_lnth_rel:
  assumes ¬ lfinite c and chain r c
  shows r (lnth c i) (lnth c (Suc i))
  using assms chain_lnth_rel lfinite_conv_llength_enat by force

lemma lnth_rel_chain:
  assumes
    ¬ lnull xs and
    ∀ j. enat (j + 1) < llength xs ⟶ R (lnth xs j) (lnth xs (j + 1))
  shows chain R xs
  using assms
proof (coinduction arbitrary: xs rule: chain.coinduct)
  case chain
  note nnul = this(1) and nth_chain = this(2)

  show ?case
  proof (cases lnull (ltl xs))
    case True
    have xs = LCons (lhd xs) LNil
      using nnul True by (simp add: llist.expand)
    then show ?thesis
      by blast
  next
    case nnul′: False
    moreover have xs = LCons (lhd xs) (ltl xs)
      using nnul by simp
    moreover have
      ∀ j. enat (j + 1) < llength (ltl xs) ⟶ R (lnth (ltl xs) j) (lnth (ltl xs) (j + 1))
      using nnul nth_chain
      by (metis Suc_eq_plus1 ldrop_eSuc_ltl ldropn_Suc_conv_ldropn ldropn_eq_LConsD lnth_ltl)
    moreover have R (lhd xs) (lhd (ltl xs))
      using nnul′ nnul nth_chain[rule_format, of 0, simplified]
      by (metis ldropn_0 ldropn_Suc_conv_ldropn ldropn_eq_LConsD lhd_LCons_ltl lhd_conv_lnth
          lnth_Suc_LCons ltl_simps(2))
    ultimately show ?thesis
      by blast
  qed
qed

lemma chain_lmap:
  assumes ∀ x y. R x y ⟶ R′ (f x) (f y) and chain R xs
  shows chain R′ (lmap f xs)
  using assms
proof (coinduction arbitrary: xs)
  case chain
  then have (∃ y. xs = LCons y LNil) ∨ (∃ ys x. xs = LCons x ys ∧ chain R ys ∧ R x (lhd ys))
```

```
    using chain.simps[of R xs] by auto
  then show ?case
  proof
    assume ∃ ys x. xs = LCons x ys ∧ chain R ys ∧ R x (lhd ys)
    then have ∃ ys x. lmap f xs = LCons x ys ∧
      (∃ xs. ys = lmap f xs ∧ (∀ x y. R x y ⟶ R' (f x) (f y)) ∧ chain R xs) ∧ R' x (lhd ys)
      using chain
      by (metis (no_types) lhd_LCons llist.distinct(1) llist.exhaust_sel llist.map_sel(1)
          lmap_eq_LNil chain_not_lnull ltl_lmap ltl_simps(2))
    then show ?thesis
      by auto
  qed auto
qed

lemma chain_mono:
  assumes ∀ x y. R x y ⟶ R' x y and chain R xs
  shows chain R' xs
  using assms by (rule chain_lmap[of _ _ λx. x, unfolded llist.map_ident])

lemma lfinite_chain_imp_rtranclp_lhd_llast: lfinite xs ⟹ chain R xs ⟹ R** (lhd xs) (llast xs)
proof (induct rule: lfinite.induct)
  case (lfinite_LConsI xs x)
  note fin_xs = this(1) and ih = this(2) and r_x_xs = this(3)
  show ?case
  proof (cases xs = LNil)
    case xs_nnil: False
    then have r_xs: chain R xs
      using r_x_xs by (blast elim: chain.cases)
    then show ?thesis
      using ih[OF r_xs] xs_nnil r_x_xs
      by (metis chain.cases converse_rtranclp_into_rtranclp lhd_LCons llast_LCons chain_not_lnull
          ltl_simps(2))
  qed simp
qed simp

lemma tranclp_imp_exists_finite_chain_list:
  R++ x y ⟹ ∃ xs. xs ≠ [] ∧ tl xs ≠ [] ∧ chain R (llist_of xs) ∧ hd xs = x ∧ last xs = y
proof (induct rule: tranclp.induct)
  case (r_into_trancl x y)
  note r_xy = this

  define xs where
    xs = [x, y]

  have xs ≠ [] and tl xs ≠ [] and chain R (llist_of xs) and hd xs = x and last xs = y
    unfolding xs_def using r_xy by (auto intro: chain.intros)
  then show ?case
    by blast
next
  case (trancl_into_trancl x y z)
  note rstar_xy = this(1) and ih = this(2) and r_yz = this(3)

  obtain xs where
    xs: xs ≠ [] tl xs ≠ [] chain R (llist_of xs) hd xs = x last xs = y
    using ih by blast
  define ys where
    ys = xs @ [z]

  have ys ≠ [] and tl ys ≠ [] and chain R (llist_of ys) and hd ys = x and last ys = z
    unfolding ys_def using xs r_yz
    by (auto simp: lappend_llist_of_llist_of[symmetric] intro: chain_singleton chain_lappend)
  then show ?case
    by blast
```

```
qed

inductive-cases chain_consE: chain R (LCons x xs)
inductive-cases chain_nontrivE: chain R (LCons x (LCons y xs))

primrec prepend where
  prepend [] ys = ys
| prepend (x # xs) ys = LCons x (prepend xs ys)

lemma prepend_butlast:
  xs ≠ [] ⟹ ¬ lnull ys ⟹ last xs = lhd ys ⟹ prepend (butlast xs) ys = prepend xs (ltl ys)
  by (induct xs) auto

lemma lnull_prepend[simp]: lnull (prepend xs ys) = (xs = [] ∧ lnull ys)
  by (induct xs) auto

lemma lhd_prepend[simp]: lhd (prepend xs ys) = (if xs ≠ [] then hd xs else lhd ys)
  by (induct xs) auto

lemma prepend_LNil[simp]: prepend xs LNil = llist_of xs
  by (induct xs) auto

lemma lfinite_prepend[simp]: lfinite (prepend xs ys) ⟷ lfinite ys
  by (induct xs) auto

lemma llength_prepend[simp]: llength (prepend xs ys) = length xs + llength ys
  by (induct xs) (auto simp: enat_0 iadd_Suc eSuc_enat[symmetric])

lemma llast_prepend[simp]: ¬ lnull ys ⟹ llast (prepend xs ys) = llast ys
  by (induct xs) (auto simp: llast_LCons)

lemma prepend_prepend: prepend xs (prepend ys zs) = prepend (xs @ ys) zs
  by (induct xs) auto

lemma chain_prepend:
  chain R (llist_of zs) ⟹ last zs = lhd xs ⟹ chain R xs ⟹ chain R (prepend zs (ltl xs))
  by (induct zs; cases xs)
    (auto split: if_splits simp: lnull_def[symmetric] intro!: chain_cons elim!: chain_consE)

lemma lmap_prepend[simp]: lmap f (prepend xs ys) = prepend (map f xs) (lmap f ys)
  by (induct xs) auto

lemma lset_prepend[simp]: lset (prepend xs ys) = set xs ∪ lset ys
  by (induct xs) auto

lemma prepend_LCons: prepend xs (LCons y ys) = prepend (xs @ [y]) ys
  by (induct xs) auto

lemma lnth_prepend:
  lnth (prepend xs ys) i = (if i < length xs then nth xs i else lnth ys (i − length xs))
  by (induct xs arbitrary: i) (auto simp: lnth_LCons' nth_Cons')

theorem lfinite_less_induct[consumes 1, case_names less]:
  assumes fin: lfinite xs
    and step: ⋀xs. lfinite xs ⟹ (⋀zs. llength zs < llength xs ⟹ P zs) ⟹ P xs
  shows P xs
using fin proof (induct the_enat (llength xs) arbitrary: xs rule: less_induct)
  case (less xs)
  show ?case
    using less(2) by (intro step[OF less(2)] less(1))
      (auto dest!: lfinite_llength_enat simp: eSuc_enat elim!: less_enatE llength_eq_enat_lfiniteD)
qed
```

```
theorem lfinite_prepend_induct[consumes 1, case_names LNil prepend]:
  assumes lfinite xs
    and LNil: P LNil
    and prepend: ⋀xs. lfinite xs ⟹ (⋀zs. (∃ys. xs = prepend ys zs ∧ ys ≠ []) ⟹ P zs) ⟹ P xs
  shows P xs
using assms(1) proof (induct xs rule: lfinite_less_induct)
  case (less xs)
  from less(1) show ?case
    by (cases xs)
      (force simp: LNil neq_Nil_conv dest: lfinite_llength_enat intro!: prepend[of LCons _ _] intro: less)+
qed

coinductive emb :: 'a llist ⇒ 'a llist ⇒ bool where
  emb LNil xs
| emb xs ys ⟹ emb (LCons x xs) (prepend zs (LCons x ys))

inductive prepend_cong1 for X where
  prepend_cong1_base: X xs ⟹ prepend_cong1 X xs
| prepend_cong1_prepend: prepend_cong1 X ys ⟹ prepend_cong1 X (prepend xs ys)

lemma emb_prepend_coinduct[rotated, case_names emb]:
  assumes (⋀x1 x2. X x1 x2 ⟹
    (∃xs. x1 = LNil ∧ x2 = xs)
     ∨ (∃xs ys x zs. x1 = LCons x xs ∧ x2 = prepend zs (LCons x ys)
       ∧ (prepend_cong1 (X xs) ys ∨ emb xs ys))) (is ⋀x1 x2. X x1 x2 ⟹ ?bisim x1 x2)
  shows X x1 x2 ⟹ emb x1 x2
proof (erule emb.coinduct[OF prepend_cong1_base])
  fix xs zs
  assume prepend_cong1 (X xs) zs
  then show ?bisim xs zs
    by (induct zs rule: prepend_cong1.induct) (erule assms, force simp: prepend_prepend)
qed

context
begin

private coinductive chain' for R where
  chain' R (LCons x LNil)
| chain R (llist_of zs) ⟹ zs ≠ [] ⟹ tl zs ≠ [] ⟹ ¬ lnull xs ⟹ last zs = lhd xs ⟹
    ys = ltl xs ⟹ chain' R xs ⟹ chain' R (prepend zs ys)

private lemma chain_imp_chain': chain R xs ⟹ chain' R xs
proof (coinduction arbitrary: xs rule: chain'.coinduct)
  case chain'
  then show ?case
  proof (cases rule: chain.cases)
    case (chain_cons zs z)
    then show ?thesis
      by (intro disjI2) (force intro: chain.intros exI[of _ [z, lhd zs]] exI[of _ zs]
          elim: chain.cases)
  qed simp
qed

private inductive-cases chain'_LConsE: chain' R (LCons x xs)

private lemma chain'_stepD1:
  assumes chain' R (LCons x (LCons y xs))
  shows chain' R (LCons y xs)
proof (cases xs)
  case [simp]: (LCons z zs)
  with assms show ?thesis
  proof (cases rule: chain'.cases)
    case (2 as ys xs)
```

```
    then show ?thesis
    proof (cases tl (tl as))
      case Nil
      with 2 show ?thesis by (auto simp: neq_Nil_conv)
    next
      case (Cons b bs)
      with 2 have chain′ R (prepend (y # b # bs) xs)
        by (intro chain′.intros)
          (auto simp: chain_cons not_lnull_conv neq_Nil_conv elim: chain_nontrivE)
      with 2 Cons show ?thesis
        by (auto simp: neq_Nil_conv)
    qed
  qed
qed (simp only: chain′.intros(1))

private lemma chain′_stepD2: chain′ R (LCons x (LCons y xs)) ⟹ R x y
  by (erule chain′.cases) (auto simp: neq_Nil_conv elim!: chain_nontrivE split: if_splits)

private lemma chain′_imp_chain: chain′ R xs ⟹ chain R xs
proof (coinduction arbitrary: xs rule: chain.coinduct)
  case chain
  then show ?case
  proof (cases rule: chain′.cases)
    case (2 ys zs xs)
    then show ?thesis
    proof (cases ltl zs)
      case LNil
      with chain 2 show ?thesis
        by (auto 0 4 simp: neq_Nil_conv not_lnull_conv elim: chain′_stepD1 chain′_stepD2)
    next
      case (LCons b bs)
      with chain 2 show ?thesis
        unfolding neq_Nil_conv not_lnull_conv
        by (elim exE) (auto elim: chain′_stepD1 chain_nontrivE)
    qed
  qed simp
qed

private lemma chain_chain′: chain = chain′
  unfolding fun_eq_iff by (metis chain_imp_chain′ chain′_imp_chain)

lemma chain_prepend_coinduct[case_names chain]:
  X x ⟹ (⋀x. X x ⟹
    (∃ z. x = LCons z LNil) ∨
    (∃ xs zs. x = prepend zs (ltl xs) ∧ zs ≠ [] ∧ tl zs ≠ [] ∧ ¬ lnull xs ∧ last zs = lhd xs ∧
      (X xs ∨ chain R xs) ∧ chain R (llist_of zs))) ⟹ chain R x
  by (subst chain_chain′, erule chain′.coinduct) (auto simp: chain_chain′)

end

context
  fixes R :: ′a ⇒ ′a ⇒ bool
begin

private definition pick where
  pick x y = (SOME xs. xs ≠ [] ∧ tl xs ≠ [] ∧ chain R (llist_of xs) ∧ hd xs = x ∧ last xs = y)

private lemma pick[simp]:
  assumes R⁺⁺ x y
  shows pick x y ≠ [] tl (pick x y) ≠ [] chain R (llist_of (pick x y))
    hd (pick x y) = x last (pick x y) = y
  unfolding pick_def using tranclp_imp_exists_finite_chain_list[THEN someI_ex, OF assms] by auto
```

```
private lemma butlast_pick[simp]: R⁺⁺ x y ⟹ butlast (pick x y) ≠ []
  by (cases pick x y; cases tl (pick x y)) (auto dest: pick(2))

private friend-of-corec prepend where
  prepend xs ys = (case xs of [] ⇒
    (case ys of LNil ⇒ LNil | LCons x xs ⇒ LCons x xs) | x # xs' ⇒ LCons x (prepend xs' ys))
  by (simp split: list.splits llist.splits) transfer_prover

private corec wit where
  wit xs = (case xs of LCons x (LCons y xs) ⇒
     let zs = pick x y in LCons (hd zs) (prepend (butlast (tl zs)) (wit (LCons y xs))) | _ ⇒ xs)

private lemma
  wit_LNil[simp]: wit LNil = LNil and
  wit_lsingleton[simp]: wit (LCons x LNil) = LCons x LNil and
  wit_LCons2: wit (LCons x (LCons y xs)) =
     (let zs = pick x y in LCons (hd zs) (prepend (butlast (tl zs)) (wit (LCons y xs))))
  by (subst wit.code; auto)+

private lemma wit_LCons: wit (LCons x xs) = (case xs of LNil ⇒ LCons x LNil | LCons y xs ⇒
     (let zs = pick x y in LCons (hd zs) (prepend (butlast (tl zs)) (wit (LCons y xs)))))
  by (subst wit.code; auto split: llist.splits)+

private lemma lnull_wit[simp]: lnull (wit xs) ⟷ lnull xs
  by (subst wit.code) (auto split: llist.splits simp: Let_def)

private lemma lhd_wit[simp]: chain R⁺⁺ xs ⟹ lhd (wit xs) = lhd xs
  by (erule chain.cases; subst wit.code) (auto split: llist.splits simp: Let_def)

private lemma butlast_alt: butlast xs = (if tl xs = [] then [] else hd xs # butlast (tl xs))
  by (cases xs) auto

private lemma wit_alt:
  chain R⁺⁺ xs ⟹ wit xs = (case xs of LCons x (LCons y xs) ⇒
     prepend (pick x y) (ltl (wit (LCons y xs))) | _ ⇒ xs)
  by (auto split: llist.splits simp: prepend_butlast[symmetric] wit_LCons2 Let_def
    prepend.simps(2)[symmetric] butlast_alt[of pick _ _]
    simp del: prepend.simps elim!: chain_nontrivE)

private lemma wit_alt2:
  chain R⁺⁺ xs ⟹ wit xs = (case xs of LCons x (LCons y xs) ⇒
     prepend (butlast (pick x y)) (wit (LCons y xs)) | _ ⇒ xs)
  by (auto split: llist.splits simp: wit_LCons2 Let_def
    prepend.simps(2)[symmetric] butlast_alt[of pick _ _]
    simp del: prepend.simps elim!: chain_nontrivE)

private lemma LNil_eq_iff_lnull: LNil = xs ⟷ lnull xs
  by (cases xs) auto

private lemma lfinite_wit[simp]:
  assumes chain R⁺⁺ xs
  shows lfinite (wit xs) ⟷ lfinite xs
proof
  assume lfinite (wit xs)
  from this assms show lfinite xs
  proof (induct wit xs arbitrary: xs rule: lfinite_prepend_induct)
    case (prepend zs)
    then show ?case
    proof (cases zs)
      case [simp]: (LCons x xs)
      then show ?thesis
      proof (cases xs)
        case [simp]: LCons
```

```
        with prepend show ?thesis
          by (subst (asm) (2) wit_alt2) (force split: llist.splits elim!: chain_nontrivE)+
      qed simp
    qed simp
  qed (simp add: LNil_eq_iff_lnull)
next
  assume lfinite xs
  then show lfinite (wit xs)
  proof (induct xs rule: lfinite.induct)
    case (lfinite_LConsI xs x)
    then show ?case
      by (cases xs) (auto simp: wit_LCons Let_def)
  qed simp
qed

private lemma llast_wit[simp]:
  assumes chain R^++ xs
  shows llast (wit xs) = llast xs
proof (cases lfinite xs)
  case True
  from this assms show ?thesis
  proof (induct rule: lfinite.induct)
    case (lfinite_LConsI xs x)
    then show ?case
      by (cases xs) (auto simp: wit_LCons2 llast_LCons elim: chain_nontrivE)
  qed auto
qed (auto simp: llast_linfinite assms)

lemma emb_wit[simp]: chain R^++ xs ⟹ emb xs (wit xs)
proof (coinduction arbitrary: xs rule: emb_prepend_coinduct)
  case (emb xs)
  then show ?case
  proof (cases rule: chain.cases)
    case (chain_cons zs z)
    then show ?thesis
      by (subst (2) wit.code)
        (auto split: llist.splits intro!: exI[of _ []] exI[of _ _ :: _ llist]
          prepend_cong1_prepend[OF prepend_cong1_base])
  qed (auto intro!: exI[of _ LNil] exI[of _ []] emb.intros)
qed

lemma chain_tranclp_imp_exists_chain:
  chain R^++ xs ⟹
   ∃ys. chain R ys ∧ emb xs ys ∧ (lfinite ys ⟷ lfinite xs) ∧ lhd ys = lhd xs
     ∧ llast ys = llast xs
proof (intro exI[of _ wit xs] conjI, coinduction arbitrary: xs rule: chain_prepend_coinduct)
  case chain
  then show ?case
    by (subst (1 2) wit_alt; assumption?) (erule chain.cases; force split: llist.splits)
qed auto

inductive-cases emb_LConsE: emb (LCons z zs) ys
inductive-cases emb_LNil2E: emb xs LNil

lemma emb_lset_mono[rotated]: x ∈ lset xs ⟹ emb xs ys ⟹ x ∈ lset ys
  by (induct x xs arbitrary: ys rule: llist.set_induct) (auto elim!: emb_LConsE)

lemma emb_Ball_lset_antimono:
  assumes emb Xs Ys
  shows ∀Y ∈ lset Ys. x ∈ Y ⟹ ∀X ∈ lset Xs. x ∈ X
  using emb_lset_mono[OF assms] by blast

lemma emb_lfinite_antimono[rotated]: lfinite ys ⟹ emb xs ys ⟹ lfinite xs
```

```
  by (induct ys arbitrary: xs rule: lfinite_prepend_induct)
    (force elim!: emb_LNil2E simp: LNil_eq_iff_lnull prepend_LCons elim: emb.cases)+

lemma emb_Liminf_llist_mono_aux:
  assumes emb Xs Ys and ¬ lfinite Xs and ¬ lfinite Ys and ∀ j≥i. x ∈ lnth Ys j
  shows ∀ j≥i. x ∈ lnth Xs j
using assms proof (induct i arbitrary: Xs Ys rule: less_induct)
  case (less i)
  then show ?case
  proof (cases i)
    case 0
    then show ?thesis
      using emb_Ball_lset_antimono[OF less(2), of x] less(5)
      unfolding Ball_def in_lset_conv_lnth simp_thms
        not_lfinite_llength[OF less(3)] not_lfinite_llength[OF less(4)] enat_ord_code subset_eq
      by blast
  next
    case [simp]: (Suc nat)
    from less(2,3) obtain xs as b bs where
      [simp]: Xs = LCons b xs Ys = prepend as (LCons b bs) and emb xs bs
      by (auto elim: emb.cases)
    have IH: ∀ k≥j. x ∈ lnth xs k if ∀ k≥j. x ∈ lnth bs k j < i for j
      using that less(1)[OF _ ‹emb xs bs›] less(3,4) by auto
    from less(5) have ∀ k≥i − length as − 1. x ∈ lnth xs k
      by (intro IH allI)
        (drule spec[of _ _ + length as + 1], auto simp: lnth_prepend lnth_LCons')
    then show ?thesis
      by (auto simp: lnth_LCons')
  qed
qed

lemma emb_Liminf_llist_infinite:
  assumes emb Xs Ys and ¬ lfinite Xs
  shows Liminf_llist Ys ⊆ Liminf_llist Xs
proof −
  from assms have ¬ lfinite Ys
    using emb_lfinite_antimono by blast
  with assms show ?thesis
    unfolding Liminf_llist_def by (auto simp: not_lfinite_llength dest: emb_Liminf_llist_mono_aux)
qed

lemma emb_lmap: emb xs ys ⟹ emb (lmap f xs) (lmap f ys)
proof (coinduction arbitrary: xs ys rule: emb.coinduct)
  case emb
  show ?case
  proof (cases xs)
    case xs: (LCons x xs')

    obtain ysa0 and zs0 where
      ys: ys = prepend zs0 (LCons x ysa0) and
      emb': emb xs' ysa0
      using emb_LConsE[OF emb[unfolded xs]] by metis

    let ?xa = f x
    let ?xsa = lmap f xs'
    let ?zs = map f zs0
    let ?ysa = lmap f ysa0

    have lmap f xs = LCons ?xa ?xsa
      unfolding xs by simp
    moreover have lmap f ys = prepend ?zs (LCons ?xa ?ysa)
      unfolding ys by simp
    moreover have ∃ xsa ysa. ?xsa = lmap f xsa ∧ ?ysa = lmap f ysa ∧ emb xsa ysa
```

using *emb'* **by** *blast*
ultimately show *?thesis*
by *blast*
qed *simp*
qed

end

lemma *chain_inf_llist_if_infinite_chain_function*:
assumes ∀ *i*. *r* (*f* (*Suc i*)) (*f i*)
shows ¬ *lfinite* (*inf_llist f*) ∧ *chain* r^{-1-1} (*inf_llist f*)
using *assms* **by** (*simp add*: *lnth_rel_chain*)

lemma *infinite_chain_function_iff_infinite_chain_llist*:
(∃ *f*. ∀ *i*. *r* (*f* (*Suc i*)) (*f i*)) ⟷ (∃ *c*. ¬ *lfinite c* ∧ *chain* r^{-1-1} *c*)
using *chain_inf_llist_if_infinite_chain_function infinite_chain_lnth_rel* **by** *blast*

lemma *wfP_iff_no_infinite_down_chain_llist*: *wfP r* ⟷ (∄ *c*. ¬ *lfinite c* ∧ *chain* r^{-1-1} *c*)
proof −
have *wfP r* ⟷ *wf* {(*x*, *y*). *r x y*}
unfolding *wfP_def* **by** *auto*
also have ... ⟷ (∄ *f*. ∀ *i*. (*f* (*Suc i*), *f i*) ∈ {(*x*, *y*). *r x y*})
using *wf_iff_no_infinite_down_chain* **by** *blast*
also have ... ⟷ (∄ *f*. ∀ *i*. *r* (*f* (*Suc i*)) (*f i*))
by *auto*
also have ... ⟷ (∄ *c*. ¬*lfinite c* ∧ *chain* r^{-1-1} *c*)
using *infinite_chain_function_iff_infinite_chain_llist* **by** *blast*
finally show *?thesis*
by *auto*
qed

4.2 Full Chains

coinductive *full_chain* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a llist* ⇒ *bool* **for** *R* :: *'a* ⇒ *'a* ⇒ *bool* **where**
full_chain_singleton: (∀ *y*. ¬ *R x y*) ⟹ *full_chain R* (*LCons x LNil*)
| *full_chain_cons*: *full_chain R xs* ⟹ *R x* (*lhd xs*) ⟹ *full_chain R* (*LCons x xs*)

lemma
full_chain_LNil[*simp*]: ¬ *full_chain R LNil* **and**
full_chain_not_lnull: *full_chain R xs* ⟹ ¬ *lnull xs*
by (*auto elim*: *full_chain.cases*)

lemma *full_chain_ldropn*:
assumes *full*: *full_chain R xs* **and** *enat n* < *llength xs*
shows *full_chain R* (*ldropn n xs*)
using *assms*
by (*induct n arbitrary*: *xs*, *simp*,
metis full_chain.cases ldrop_eSuc_ltl ldropn_LNil ldropn_eq_LNil ltl_simps(*2*) *not_less*)

lemma *full_chain_iff_chain*:
full_chain R xs ⟷ *chain R xs* ∧ (*lfinite xs* ⟶ (∀ *y*. ¬ *R* (*llast xs*) *y*))
proof (*intro iffI conjI impI allI*; (*elim conjE*)*?*)
assume *full*: *full_chain R xs*

show *chain*: *chain R xs*
using *full* **by** (*coinduction arbitrary*: *xs*) (*auto elim*: *full_chain.cases*)

{
fix *y*
assume *lfinite xs*
then obtain *n* **where**
suc_n: *Suc n* = *llength xs*
by (*metis chain chain_length_pos lessE less_enatE lfinite_conv_llength_enat*)

```
    have full_chain R (ldropn n xs)
      by (rule full_chain_ldropn[OF full]) (use suc_n Suc_ile_eq in force)
    moreover have ldropn n xs = LCons (llast xs) LNil
      using suc_n by (metis enat_le_plus_same(2) enat_ord_simps(2) gen_llength_def
          ldropn_Suc_conv_ldropn ldropn_all lessI llast_ldropn llast_singleton llength_code)
    ultimately show ¬ R (llast xs) y
      by (auto elim: full_chain.cases)
  }
next
  assume
    chain R xs and
    lfinite xs ⟶ (∀ y. ¬ R (llast xs) y)
  then show full_chain R xs
    by (coinduction arbitrary: xs) (erule chain.cases, simp, metis lfinite_LConsI llast_LCons)
qed

lemma full_chain_imp_chain: full_chain R xs ⟹ chain R xs
  using full_chain_iff_chain by blast

lemma full_chain_length_pos: full_chain R xs ⟹ llength xs > 0
  by (fact chain_length_pos[OF full_chain_imp_chain])

lemma full_chain_lnth_rel:
  full_chain R xs ⟹ enat (Suc j) < llength xs ⟹ R (lnth xs j) (lnth xs (Suc j))
  by (fact chain_lnth_rel[OF full_chain_imp_chain])

inductive-cases full_chain_consE: full_chain R (LCons x xs)
inductive-cases full_chain_nontrivE: full_chain R (LCons x (LCons y xs))

lemma full_chain_tranclp_imp_exists_full_chain:
  assumes full: full_chain R⁺⁺ xs
  shows ∃ ys. full_chain R ys ∧ emb xs ys ∧ lfinite ys = lfinite xs ∧ lhd ys = lhd xs
    ∧ llast ys = llast xs
proof −
  obtain ys where ys:
    chain R ys emb xs ys lfinite ys = lfinite xs lhd ys = lhd xs llast ys = llast xs
    using full_chain_imp_chain[OF full] chain_tranclp_imp_exists_chain by blast
  have full_chain R ys
    using ys(1,3,5) full unfolding full_chain_iff_chain by auto
  then show ?thesis
    using ys(2−5) by auto
qed

end
```

5 Clausal Logic

```
theory Clausal_Logic
  imports Nested_Multisets_Ordinals.Multiset_More
begin
```

Resolution operates of clauses, which are disjunctions of literals. The material formalized here corresponds roughly to Sections 2.1 (“Formulas and Clauses”) of Bachmair and Ganzinger, excluding the formula and term syntax.

5.1 Literals

Literals consist of a polarity (positive or negative) and an atom, of type $'a$.

```
datatype 'a literal =
  is_pos: Pos (atm_of: 'a)
| Neg (atm_of: 'a)
```

abbreviation *is_neg* :: *'a literal* ⇒ *bool* **where**
is_neg L ≡ ¬ *is_pos L*

lemma *Pos_atm_of_iff*[*simp*]: *Pos* (*atm_of L*) = *L* ⟷ *is_pos L*
by (*cases L*) *simp+*

lemma *Neg_atm_of_iff*[*simp*]: *Neg* (*atm_of L*) = *L* ⟷ *is_neg L*
by (*cases L*) *simp+*

lemma *set_literal_atm_of*: *set_literal L* = {*atm_of L*}
by (*cases L*) *simp+*

lemma *ex_lit_cases*: (∃ *L*. *P L*) ⟷ (∃ *A*. *P* (*Pos A*) ∨ *P* (*Neg A*))
by (*metis literal.exhaust*)

instantiation *literal* :: (*type*) *uminus*
begin

definition *uminus_literal* :: *'a literal* ⇒ *'a literal* **where**
uminus L = (*if is_pos L then Neg else Pos*) (*atm_of L*)

instance ..

end

lemma
uminus_Pos[*simp*]: − *Pos A* = *Neg A* **and**
uminus_Neg[*simp*]: − *Neg A* = *Pos A*
unfolding *uminus_literal_def* **by** *simp_all*

lemma *atm_of_uminus*[*simp*]: *atm_of* (−*L*) = *atm_of L*
by (*case_tac L*, *auto*)

lemma *uminus_of_uminus_id*[*simp*]: − (− (*x* :: *'v literal*)) = *x*
by (*simp add*: *uminus_literal_def*)

lemma *uminus_not_id*[*simp*]: *x* ≠ − (*x*:: *'v literal*)
by (*case_tac x*) *auto*

lemma *uminus_not_id'*[*simp*]: − *x* ≠ (*x*:: *'v literal*)
by (*case_tac x*, *auto*)

lemma *uminus_eq_inj*[*iff*]: − (*a*::*'v literal*) = − *b* ⟷ *a* = *b*
by (*case_tac a*; *case_tac b*) *auto+*

lemma *uminus_lit_swap*: (*a*::*'a literal*) = − *b* ⟷ − *a* = *b*
by *auto*

lemma *is_pos_neg_not_is_pos*: *is_pos* (− *L*) ⟷ ¬ *is_pos L*
by (*cases L*) *auto*

instantiation *literal* :: (*preorder*) *preorder*
begin

definition *less_literal* :: *'a literal* ⇒ *'a literal* ⇒ *bool* **where**
less_literal L M ⟷ *atm_of L* < *atm_of M* ∨ *atm_of L* ≤ *atm_of M* ∧ *is_neg L* < *is_neg M*

definition *less_eq_literal* :: *'a literal* ⇒ *'a literal* ⇒ *bool* **where**
less_eq_literal L M ⟷ *atm_of L* < *atm_of M* ∨ *atm_of L* ≤ *atm_of M* ∧ *is_neg L* ≤ *is_neg M*

instance
apply *intro_classes*
unfolding *less_literal_def less_eq_literal_def* **by** (*auto intro*: *order_trans simp*: *less_le_not_le*)

end

instantiation *literal* :: (*order*) *order*
begin

instance
  by *intro_classes* (*auto simp*: *less_eq_literal_def intro*: *literal.expand*)

end

lemma *pos_less_neg*[*simp*]: $Pos\ A < Neg\ A$
  unfolding *less_literal_def* **by** *simp*

lemma *pos_less_pos_iff*[*simp*]: $Pos\ A < Pos\ B \longleftrightarrow A < B$
  unfolding *less_literal_def* **by** *simp*

lemma *pos_less_neg_iff*[*simp*]: $Pos\ A < Neg\ B \longleftrightarrow A \leq B$
  unfolding *less_literal_def* **by** (*auto simp*: *less_le_not_le*)

lemma *neg_less_pos_iff*[*simp*]: $Neg\ A < Pos\ B \longleftrightarrow A < B$
  unfolding *less_literal_def* **by** *simp*

lemma *neg_less_neg_iff*[*simp*]: $Neg\ A < Neg\ B \longleftrightarrow A < B$
  unfolding *less_literal_def* **by** *simp*

lemma *pos_le_neg*[*simp*]: $Pos\ A \leq Neg\ A$
  unfolding *less_eq_literal_def* **by** *simp*

lemma *pos_le_pos_iff*[*simp*]: $Pos\ A \leq Pos\ B \longleftrightarrow A \leq B$
  unfolding *less_eq_literal_def* **by** (*auto simp*: *less_le_not_le*)

lemma *pos_le_neg_iff*[*simp*]: $Pos\ A \leq Neg\ B \longleftrightarrow A \leq B$
  unfolding *less_eq_literal_def* **by** (*auto simp*: *less_imp_le*)

lemma *neg_le_pos_iff*[*simp*]: $Neg\ A \leq Pos\ B \longleftrightarrow A < B$
  unfolding *less_eq_literal_def* **by** *simp*

lemma *neg_le_neg_iff*[*simp*]: $Neg\ A \leq Neg\ B \longleftrightarrow A \leq B$
  unfolding *less_eq_literal_def* **by** (*auto simp*: *less_imp_le*)

lemma *leq_imp_less_eq_atm_of*: $L \leq M \Longrightarrow atm\_of\ L \leq atm\_of\ M$
  unfolding *less_eq_literal_def* **using** *less_imp_le* **by** *blast*

instantiation *literal* :: (*linorder*) *linorder*
begin

instance
  apply *intro_classes*
  unfolding *less_eq_literal_def less_literal_def* **by** *auto*

end

instantiation *literal* :: (*wellorder*) *wellorder*
begin

instance
proof *intro_classes*
  fix P :: $'a\ literal \Rightarrow bool$ **and** L :: $'a\ literal$
  assume *ih*: $\bigwedge L.\ (\bigwedge M.\ M < L \Longrightarrow P\ M) \Longrightarrow P\ L$
  have $\bigwedge x.\ (\bigwedge y.\ y < x \Longrightarrow P\ (Pos\ y) \wedge P\ (Neg\ y)) \Longrightarrow P\ (Pos\ x) \wedge P\ (Neg\ x)$
    by (*rule conjI*[*OF ih ih*])
      (*auto simp*: *less_literal_def atm_of_def split*: *literal.splits intro*: *ih*)

```
  then have ⋀A. P (Pos A) ∧ P (Neg A)
    by (rule less_induct) blast
  then show P L
    by (cases L) simp+
qed

end
```

5.2 Clauses

Clauses are (finite) multisets of literals.

```
type-synonym 'a clause = 'a literal multiset

abbreviation map_clause :: ('a ⇒ 'b) ⇒ 'a clause ⇒ 'b clause where
  map_clause f ≡ image_mset (map_literal f)

abbreviation rel_clause :: ('a ⇒ 'b ⇒ bool) ⇒ 'a clause ⇒ 'b clause ⇒ bool where
  rel_clause R ≡ rel_mset (rel_literal R)

abbreviation poss :: 'a multiset ⇒ 'a clause where poss AA ≡ {#Pos A. A ∈# AA#}
abbreviation negs :: 'a multiset ⇒ 'a clause where negs AA ≡ {#Neg A. A ∈# AA#}

lemma Max_in_lits: C ≠ {#} ⟹ Max_mset C ∈# C
  by simp

lemma Max_atm_of_set_mset_commute: C ≠ {#} ⟹ Max (atm_of ` set_mset C) = atm_of (Max_mset C)
  by (rule mono_Max_commute[symmetric]) (auto simp: mono_def less_eq_literal_def)

lemma Max_pos_neg_less_multiset:
  assumes max: Max_mset C = Pos A and neg: Neg A ∈# D
  shows C < D
proof −
  have Max_mset C < Neg A
    using max by simp
  then show ?thesis
    using neg by (metis (no_types) Max_less_iff empty_iff ex_gt_imp_less_multiset finite_set_mset)
qed

lemma pos_Max_imp_neg_notin: Max_mset C = Pos A ⟹ Neg A ∉# C
  using Max_pos_neg_less_multiset by blast

lemma less_eq_Max_lit: C ≠ {#} ⟹ C ≤ D ⟹ Max_mset C ≤ Max_mset D
proof (unfold less_eq_multiset_HO)
  assume
    ne: C ≠ {#} and
    ex_gt: ∀x. count D x < count C x ⟶ (∃y > x. count C y < count D y)
  from ne have Max_mset C ∈# C
    by (fast intro: Max_in_lits)
  then have ∃l. l ∈# D ∧ ¬ l < Max_mset C
    using ex_gt by (metis count_greater_zero_iff count_inI less_not_sym)
  then have ¬ Max_mset D < Max_mset C
    by (metis Max.coboundedI[OF finite_set_mset] le_less_trans)
  then show ?thesis
    by simp
qed

definition atms_of :: 'a clause ⇒ 'a set where
  atms_of C = atm_of ` set_mset C

lemma atms_of_empty[simp]: atms_of {#} = {}
  unfolding atms_of_def by simp

lemma atms_of_singleton[simp]: atms_of {#L#} = {atm_of L}
```

unfolding *atms_of_def* **by** *auto*

lemma *atms_of_add_mset*[*simp*]: *atms_of* (*add_mset a A*) = *insert* (*atm_of a*) (*atms_of A*)
unfolding *atms_of_def* **by** *auto*

lemma *atms_of_union_mset*[*simp*]: *atms_of* ($A \cup\# B$) = *atms_of A* $\cup$ *atms_of B*
unfolding *atms_of_def* **by** *auto*

lemma *finite_atms_of*[*iff*]: *finite* (*atms_of C*)
by (*simp add*: *atms_of_def*)

lemma *atm_of_lit_in_atms_of*: $L \in\# C \Longrightarrow$ *atm_of L* $\in$ *atms_of C*
by (*simp add*: *atms_of_def*)

lemma *atms_of_plus*[*simp*]: *atms_of* ($C + D$) = *atms_of C* $\cup$ *atms_of D*
unfolding *atms_of_def* **by** *auto*

lemma *in_atms_of_minusD*: $x \in$ *atms_of* ($A - B$) $\Longrightarrow x \in$ *atms_of A*
by (*auto simp*: *atms_of_def dest*: *in_diffD*)

lemma *pos_lit_in_atms_of*: *Pos A* $\in\# C \Longrightarrow A \in$ *atms_of C*
unfolding *atms_of_def* **by** *force*

lemma *neg_lit_in_atms_of*: *Neg A* $\in\# C \Longrightarrow A \in$ *atms_of C*
unfolding *atms_of_def* **by** *force*

lemma *atm_imp_pos_or_neg_lit*: $A \in$ *atms_of C* $\Longrightarrow$ *Pos A* $\in\# C \vee$ *Neg A* $\in\# C$
unfolding *atms_of_def image_def mem_Collect_eq* **by** (*metis Neg_atm_of_iff Pos_atm_of_iff*)

lemma *atm_iff_pos_or_neg_lit*: $A \in$ *atms_of L* $\longleftrightarrow$ *Pos A* $\in\# L \vee$ *Neg A* $\in\# L$
by (*auto intro*: *pos_lit_in_atms_of neg_lit_in_atms_of dest*: *atm_imp_pos_or_neg_lit*)

lemma *atm_of_eq_atm_of*: *atm_of L* = *atm_of L'* $\longleftrightarrow$ ($L = L' \vee L = -L'$)
by (*cases L*; *cases L'*) *auto*

lemma *atm_of_in_atm_of_set_iff_in_set_or_uminus_in_set*: *atm_of L* $\in$ *atm_of* ‘ $I \longleftrightarrow (L \in I \vee -L \in I)$
by (*auto intro*: *rev_image_eqI simp*: *atm_of_eq_atm_of*)

lemma *lits_subseteq_imp_atms_subseteq*: *set_mset C* $\subseteq$ *set_mset D* $\Longrightarrow$ *atms_of C* $\subseteq$ *atms_of D*
unfolding *atms_of_def* **by** *blast*

lemma *atms_empty_iff_empty*[*iff*]: *atms_of C* = {} $\longleftrightarrow C = \{\#\}$
unfolding *atms_of_def image_def Collect_empty_eq* **by** *auto*

lemma
atms_of_poss[*simp*]: *atms_of* (*poss AA*) = *set_mset AA* **and**
atms_of_negs[*simp*]: *atms_of* (*negs AA*) = *set_mset AA*
unfolding *atms_of_def image_def* **by** *auto*

lemma *less_eq_Max_atms_of*: $C \neq \{\#\} \Longrightarrow C \leq D \Longrightarrow$ *Max* (*atms_of C*) $\leq$ *Max* (*atms_of D*)
unfolding *atms_of_def*
by (*metis Max_atm_of_set_mset_commute leq_imp_less_eq_atm_of less_eq_Max_lit*
less_eq_multiset_empty_right)

lemma *le_multiset_Max_in_imp_Max*:
Max (*atms_of D*) = $A \Longrightarrow C \leq D \Longrightarrow A \in$ *atms_of C* $\Longrightarrow$ *Max* (*atms_of C*) = A
by (*metis Max.coboundedI*[*OF finite_atms_of*] *atms_of_def empty_iff eq_iff image_subsetI*
less_eq_Max_atms_of set_mset_empty subset_Compl_self_eq)

lemma *atm_of_Max_lit*[*simp*]: $C \neq \{\#\} \Longrightarrow$ *atm_of* (*Max_mset C*) = *Max* (*atms_of C*)
unfolding *atms_of_def Max_atm_of_set_mset_commute* **..**

lemma *Max_lit_eq_pos_or_neg_Max_atm*:

$C \neq \{\#\} \Longrightarrow Max\_mset\ C = Pos\ (Max\ (atms\_of\ C)) \lor Max\_mset\ C = Neg\ (Max\ (atms\_of\ C))$
by (*metis Neg_atm_of_iff Pos_atm_of_iff atm_of_Max_lit*)

lemma *atms_less_imp_lit_less_pos*: $(\bigwedge B.\ B \in atms\_of\ C \Longrightarrow B < A) \Longrightarrow L \in\#\ C \Longrightarrow L < Pos\ A$
unfolding *atms_of_def less_literal_def* **by** *force*

lemma *atms_less_eq_imp_lit_less_eq_neg*: $(\bigwedge B.\ B \in atms\_of\ C \Longrightarrow B \leq A) \Longrightarrow L \in\#\ C \Longrightarrow L \leq Neg\ A$
unfolding *less_eq_literal_def* **by** (*simp add*: *atm_of_lit_in_atms_of*)

end

6 Herbrand Intepretation

theory *Herbrand_Interpretation*
imports *Clausal_Logic*
begin

The material formalized here corresponds roughly to Sections 2.2 ("Herbrand Interpretations") of Bachmair and Ganzinger, excluding the formula and term syntax.

A Herbrand interpretation is a set of ground atoms that are to be considered true.

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

definition *true_lit* :: $'a\ interp \Rightarrow 'a\ literal \Rightarrow bool$ (**infix** $\models l$ *50*) **where**
$I \models l\ L \longleftrightarrow (if\ is\_pos\ L\ then\ (\lambda P.\ P)\ else\ Not)\ (atm\_of\ L \in I)$

lemma *true_lit_simps*[*simp*]:
$I \models l\ Pos\ A \longleftrightarrow A \in I$
$I \models l\ Neg\ A \longleftrightarrow A \notin I$
unfolding *true_lit_def* **by** *auto*

lemma *true_lit_iff*[*iff*]: $I \models l\ L \longleftrightarrow (\exists A.\ L = Pos\ A \land A \in I \lor L = Neg\ A \land A \notin I)$
by (*cases L*) *simp+*

definition *true_cls* :: $'a\ interp \Rightarrow 'a\ clause \Rightarrow bool$ (**infix** $\models$ *50*) **where**
$I \models C \longleftrightarrow (\exists L \in\#\ C.\ I \models l\ L)$

lemma *true_cls_empty*[*iff*]: $\neg\ I \models \{\#\}$
unfolding *true_cls_def* **by** *simp*

lemma *true_cls_singleton*[*iff*]: $I \models \{\#L\#\} \longleftrightarrow I \models l\ L$
unfolding *true_cls_def* **by** *simp*

lemma *true_cls_add_mset*[*iff*]: $I \models add\_mset\ C\ D \longleftrightarrow I \models l\ C \lor I \models D$
unfolding *true_cls_def* **by** *auto*

lemma *true_cls_union*[*iff*]: $I \models C + D \longleftrightarrow I \models C \lor I \models D$
unfolding *true_cls_def* **by** *auto*

lemma *true_cls_mono*: $set\_mset\ C \subseteq set\_mset\ D \Longrightarrow I \models C \Longrightarrow I \models D$
unfolding *true_cls_def subset_eq* **by** *metis*

lemma
assumes $I \subseteq J$
shows
false_to_true_imp_ex_pos: $\neg\ I \models C \Longrightarrow J \models C \Longrightarrow \exists A \in J.\ Pos\ A \in\#\ C$ **and**
true_to_false_imp_ex_neg: $I \models C \Longrightarrow \neg\ J \models C \Longrightarrow \exists A \in J.\ Neg\ A \in\#\ C$
using *assms* **unfolding** *subset_iff true_cls_def* **by** (*metis literal.collapse true_lit_simps*)+

lemma *true_cls_replicate_mset*[*iff*]: $I \models replicate\_mset\ n\ L \longleftrightarrow n \neq 0 \land I \models l\ L$
by (*simp add*: *true_cls_def*)

lemma *pos_literal_in_imp_true_cls*[*intro*]: $Pos\ A \in\#\ C \Longrightarrow A \in I \Longrightarrow I \models C$

using *true_cls_def* **by** *blast*

lemma *neg_literal_notin_imp_true_cls*[*intro*]: $Neg\ A \in\#\ C \Longrightarrow A \notin I \Longrightarrow I \models C$
using *true_cls_def* **by** *blast*

lemma *pos_neg_in_imp_true*: $Pos\ A \in\#\ C \Longrightarrow Neg\ A \in\#\ C \Longrightarrow I \models C$
using *true_cls_def* **by** *blast*

definition *true_clss* :: $'a\ interp \Rightarrow 'a\ clause\ set \Rightarrow bool$ (**infix** $\models s$ *50*) **where**
$I \models s\ CC \longleftrightarrow (\forall C \in CC.\ I \models C)$

lemma *true_clss_empty*[*iff*]: $I \models s\ \{\}$
by (*simp add*: *true_clss_def*)

lemma *true_clss_singleton*[*iff*]: $I \models s\ \{C\} \longleftrightarrow I \models C$
unfolding *true_clss_def* **by** *blast*

lemma *true_clss_insert*[*iff*]: $I \models s\ insert\ C\ DD \longleftrightarrow I \models C \land I \models s\ DD$
unfolding *true_clss_def* **by** *blast*

lemma *true_clss_union*[*iff*]: $I \models s\ CC \cup DD \longleftrightarrow I \models s\ CC \land I \models s\ DD$
unfolding *true_clss_def* **by** *blast*

lemma *true_clss_mono*: $DD \subseteq CC \Longrightarrow I \models s\ CC \Longrightarrow I \models s\ DD$
by (*simp add*: *set_mp true_clss_def*)

abbreviation *satisfiable* :: $'a\ clause\ set \Rightarrow bool$ **where**
$satisfiable\ CC \equiv \exists I.\ I \models s\ CC$

definition *true_cls_mset* :: $'a\ interp \Rightarrow 'a\ clause\ multiset \Rightarrow bool$ (**infix** $\models m$ *50*) **where**
$I \models m\ CC \longleftrightarrow (\forall C \in\#\ CC.\ I \models C)$

lemma *true_cls_mset_empty*[*iff*]: $I \models m\ \{\#\}$
unfolding *true_cls_mset_def* **by** *auto*

lemma *true_cls_mset_singleton*[*iff*]: $I \models m\ \{\#C\#\} \longleftrightarrow I \models C$
by (*simp add*: *true_cls_mset_def*)

lemma *true_cls_mset_union*[*iff*]: $I \models m\ CC + DD \longleftrightarrow I \models m\ CC \land I \models m\ DD$
unfolding *true_cls_mset_def* **by** *auto*

lemma *true_cls_mset_add_mset*[*iff*]: $I \models m\ add\_mset\ C\ CC \longleftrightarrow I \models C \land I \models m\ CC$
unfolding *true_cls_mset_def* **by** *auto*

lemma *true_cls_mset_image_mset*[*iff*]: $I \models m\ image\_mset\ f\ A \longleftrightarrow (\forall x \in\#\ A.\ I \models f\ x)$
unfolding *true_cls_mset_def* **by** *auto*

lemma *true_cls_mset_mono*: $set\_mset\ DD \subseteq set\_mset\ CC \Longrightarrow I \models m\ CC \Longrightarrow I \models m\ DD$
unfolding *true_cls_mset_def subset_iff* **by** *auto*

lemma *true_clss_set_mset*[*iff*]: $I \models s\ set\_mset\ CC \longleftrightarrow I \models m\ CC$
unfolding *true_clss_def true_cls_mset_def* **by** *auto*

lemma *true_cls_mset_true_cls*: $I \models m\ CC \Longrightarrow C \in\#\ CC \Longrightarrow I \models C$
using *true_cls_mset_def* **by** *auto*

end

7 Abstract Substitutions

theory *Abstract_Substitution*
imports *Clausal_Logic Map2*
begin

Atoms and substitutions are abstracted away behind some locales, to avoid having a direct dependency on the IsaFoR library.

Conventions: $'s$ substitutions, $'a$ atoms.

7.1 Library

```
lemma f_Suc_decr_eventually_const:
  fixes f :: nat ⇒ nat
  assumes leq: ∀ i. f (Suc i) ≤ f i
  shows ∃ l. ∀ l' ≥ l. f l' = f (Suc l')
proof (rule ccontr)
  assume a: ∄ l. ∀ l' ≥ l. f l' = f (Suc l')
  have ∀ i. ∃ i'. i' > i ∧ f i' < f i
  proof
    fix i
    from a have ∃ l' ≥ i. f l' ≠ f (Suc l')
      by auto
    then obtain l' where
      l'_p: l' ≥ i ∧ f l' ≠ f (Suc l')
      by metis
    then have f l' > f (Suc l')
      using leq le_eq_less_or_eq by auto
    moreover have f i ≥ f l'
      using leq l'_p by (induction l' arbitrary: i) (blast intro: lift_Suc_antimono_le)+
    ultimately show ∃ i' > i. f i' < f i
      using l'_p less_le_trans by blast
  qed
  then obtain g_sm :: nat ⇒ nat where
    g_sm_p: ∀ i. g_sm i > i ∧ f (g_sm i) < f i
    by metis

  define c :: nat ⇒ nat where
    ⋀n. c n = (g_sm ^^ n) 0

  have f (c i) > f (c (Suc i)) for i
    by (induction i) (auto simp: c_def g_sm_p)
  then have ∀ i. (f ∘ c) i > (f ∘ c) (Suc i)
    by auto
  then have ∃ fc :: nat ⇒ nat. ∀ i. fc i > fc (Suc i)
    by metis
  then show False
    using wf_less_than by (simp add: wf_iff_no_infinite_down_chain)
qed
```

7.2 Substitution Operators

```
locale substitution_ops =
  fixes
    subst_atm :: 'a ⇒ 's ⇒ 'a and
    id_subst :: 's and
    comp_subst :: 's ⇒ 's ⇒ 's
begin

abbreviation subst_atm_abbrev :: 'a ⇒ 's ⇒ 'a (infixl ·a 67) where
  subst_atm_abbrev ≡ subst_atm

abbreviation comp_subst_abbrev :: 's ⇒ 's ⇒ 's (infixl ⊙ 67) where
  comp_subst_abbrev ≡ comp_subst

definition comp_substs :: 's list ⇒ 's list ⇒ 's list (infixl ⊙s 67) where
  σs ⊙s τs = map2 comp_subst σs τs

definition subst_atms :: 'a set ⇒ 's ⇒ 'a set (infixl ·as 67) where
```

$AA \cdot as\ \sigma = (\lambda A.\ A \cdot a\ \sigma)$ ‘ AA

definition *subst_atmss* :: $'a\ set\ set \Rightarrow 's \Rightarrow 'a\ set\ set$ (**infixl** $\cdot ass\ 67$) **where**
$AAA \cdot ass\ \sigma = (\lambda AA.\ AA \cdot as\ \sigma)$ ‘ AAA

definition *subst_atm_list* :: $'a\ list \Rightarrow 's \Rightarrow 'a\ list$ (**infixl** $\cdot al\ 67$) **where**
$As \cdot al\ \sigma = map\ (\lambda A.\ A \cdot a\ \sigma)\ As$

definition *subst_atm_mset* :: $'a\ multiset \Rightarrow 's \Rightarrow 'a\ multiset$ (**infixl** $\cdot am\ 67$) **where**
$AA \cdot am\ \sigma = image\_mset\ (\lambda A.\ A \cdot a\ \sigma)\ AA$

definition
subst_atm_mset_list :: $'a\ multiset\ list \Rightarrow 's \Rightarrow 'a\ multiset\ list$ (**infixl** $\cdot aml\ 67$)
where
$AAA \cdot aml\ \sigma = map\ (\lambda AA.\ AA \cdot am\ \sigma)\ AAA$

definition
subst_atm_mset_lists :: $'a\ multiset\ list \Rightarrow 's\ list \Rightarrow 'a\ multiset\ list$ (**infixl** $\cdot\cdot aml\ 67$)
where
$AAs \cdot\cdot aml\ \sigma s = map2\ (op\ \cdot am)\ AAs\ \sigma s$

definition *subst_lit* :: $'a\ literal \Rightarrow 's \Rightarrow 'a\ literal$ (**infixl** $\cdot l\ 67$) **where**
$L \cdot l\ \sigma = map\_literal\ (\lambda A.\ A \cdot a\ \sigma)\ L$

lemma *atm_of_subst_lit*[*simp*]: $atm\_of\ (L \cdot l\ \sigma) = atm\_of\ L \cdot a\ \sigma$
unfolding *subst_lit_def* **by** (*cases L*) *simp+*

definition *subst_cls* :: $'a\ clause \Rightarrow 's \Rightarrow 'a\ clause$ (**infixl** $\cdot\ 67$) **where**
$AA \cdot \sigma = image\_mset\ (\lambda A.\ A \cdot l\ \sigma)\ AA$

definition *subst_clss* :: $'a\ clause\ set \Rightarrow 's \Rightarrow 'a\ clause\ set$ (**infixl** $\cdot cs\ 67$) **where**
$AA \cdot cs\ \sigma = (\lambda A.\ A \cdot \sigma)$ ‘ AA

definition *subst_cls_list* :: $'a\ clause\ list \Rightarrow 's \Rightarrow 'a\ clause\ list$ (**infixl** $\cdot cl\ 67$) **where**
$Cs \cdot cl\ \sigma = map\ (\lambda A.\ A \cdot \sigma)\ Cs$

definition *subst_cls_lists* :: $'a\ clause\ list \Rightarrow 's\ list \Rightarrow 'a\ clause\ list$ (**infixl** $\cdot\cdot cl\ 67$) **where**
$Cs \cdot\cdot cl\ \sigma s = map2\ (op\ \cdot)\ Cs\ \sigma s$

definition *subst_cls_mset* :: $'a\ clause\ multiset \Rightarrow 's \Rightarrow 'a\ clause\ multiset$ (**infixl** $\cdot cm\ 67$) **where**
$CC \cdot cm\ \sigma = image\_mset\ (\lambda A.\ A \cdot \sigma)\ CC$

lemma *subst_cls_add_mset*[*simp*]: $add\_mset\ L\ C \cdot \sigma = add\_mset\ (L \cdot l\ \sigma)\ (C \cdot \sigma)$
unfolding *subst_cls_def* **by** *simp*

lemma *subst_cls_mset_add_mset*[*simp*]: $add\_mset\ C\ CC \cdot cm\ \sigma = add\_mset\ (C \cdot \sigma)\ (CC \cdot cm\ \sigma)$
unfolding *subst_cls_mset_def* **by** *simp*

definition *generalizes_atm* :: $'a \Rightarrow 'a \Rightarrow bool$ **where**
$generalizes\_atm\ A\ B \longleftrightarrow (\exists \sigma.\ A \cdot a\ \sigma = B)$

definition *strictly_generalizes_atm* :: $'a \Rightarrow 'a \Rightarrow bool$ **where**
$strictly\_generalizes\_atm\ A\ B \longleftrightarrow generalizes\_atm\ A\ B \wedge \neg\ generalizes\_atm\ B\ A$

definition *generalizes_lit* :: $'a\ literal \Rightarrow 'a\ literal \Rightarrow bool$ **where**
$generalizes\_lit\ L\ M \longleftrightarrow (\exists \sigma.\ L \cdot l\ \sigma = M)$

definition *strictly_generalizes_lit* :: $'a\ literal \Rightarrow 'a\ literal \Rightarrow bool$ **where**
$strictly\_generalizes\_lit\ L\ M \longleftrightarrow generalizes\_lit\ L\ M \wedge \neg\ generalizes\_lit\ M\ L$

definition *generalizes_cls* :: $'a\ clause \Rightarrow 'a\ clause \Rightarrow bool$ **where**
$generalizes\_cls\ C\ D \longleftrightarrow (\exists \sigma.\ C \cdot \sigma = D)$

definition *strictly_generalizes_cls* :: $'a$ *clause* $\Rightarrow$ $'a$ *clause* $\Rightarrow$ *bool* **where**
strictly_generalizes_cls C $D \longleftrightarrow$ *generalizes_cls* C D $\wedge$ $\neg$ *generalizes_cls* D C

definition *subsumes* :: $'a$ *clause* $\Rightarrow$ $'a$ *clause* $\Rightarrow$ *bool* **where**
subsumes C $D \longleftrightarrow (\exists\sigma.\ C \cdot \sigma \subseteq\#\ D)$

definition *strictly_subsumes* :: $'a$ *clause* $\Rightarrow$ $'a$ *clause* $\Rightarrow$ *bool* **where**
strictly_subsumes C $D \longleftrightarrow$ *subsumes* C D $\wedge$ $\neg$ *subsumes* D C

definition *variants* :: $'a$ *clause* $\Rightarrow$ $'a$ *clause* $\Rightarrow$ *bool* **where**
variants C $D \longleftrightarrow$ *generalizes_cls* C D $\wedge$ *generalizes_cls* D C

definition *is_renaming* :: $'s \Rightarrow$ *bool* **where**
is_renaming $\sigma \longleftrightarrow (\exists\tau.\ \sigma \odot \tau = id\_subst)$

definition *is_renaming_list* :: $'s$ *list* $\Rightarrow$ *bool* **where**
is_renaming_list $\sigma s \longleftrightarrow (\forall\sigma \in set\ \sigma s.\ is\_renaming\ \sigma)$

definition *inv_renaming* :: $'s \Rightarrow 's$ **where**
inv_renaming $\sigma = (SOME\ \tau.\ \sigma \odot \tau = id\_subst)$

definition *is_ground_atm* :: $'a \Rightarrow$ *bool* **where**
is_ground_atm $A \longleftrightarrow (\forall\sigma.\ A = A \cdot a\ \sigma)$

definition *is_ground_atms* :: $'a$ *set* $\Rightarrow$ *bool* **where**
is_ground_atms $AA = (\forall A \in AA.\ is\_ground\_atm\ A)$

definition *is_ground_atm_list* :: $'a$ *list* $\Rightarrow$ *bool* **where**
is_ground_atm_list $As \longleftrightarrow (\forall A \in set\ As.\ is\_ground\_atm\ A)$

definition *is_ground_atm_mset* :: $'a$ *multiset* $\Rightarrow$ *bool* **where**
is_ground_atm_mset $AA \longleftrightarrow (\forall A.\ A \in\#\ AA \longrightarrow is\_ground\_atm\ A)$

definition *is_ground_lit* :: $'a$ *literal* $\Rightarrow$ *bool* **where**
is_ground_lit $L \longleftrightarrow$ *is_ground_atm* (*atm_of* L)

definition *is_ground_cls* :: $'a$ *clause* $\Rightarrow$ *bool* **where**
is_ground_cls $C \longleftrightarrow (\forall L.\ L \in\#\ C \longrightarrow is\_ground\_lit\ L)$

definition *is_ground_clss* :: $'a$ *clause set* $\Rightarrow$ *bool* **where**
is_ground_clss $CC \longleftrightarrow (\forall C \in CC.\ is\_ground\_cls\ C)$

definition *is_ground_cls_list* :: $'a$ *clause list* $\Rightarrow$ *bool* **where**
is_ground_cls_list $CC \longleftrightarrow (\forall C \in set\ CC.\ is\_ground\_cls\ C)$

definition *is_ground_subst* :: $'s \Rightarrow$ *bool* **where**
is_ground_subst $\sigma \longleftrightarrow (\forall A.\ is\_ground\_atm\ (A \cdot a\ \sigma))$

definition *is_ground_subst_list* :: $'s$ *list* $\Rightarrow$ *bool* **where**
is_ground_subst_list $\sigma s \longleftrightarrow (\forall\sigma \in set\ \sigma s.\ is\_ground\_subst\ \sigma)$

definition *grounding_of_cls* :: $'a$ *clause* $\Rightarrow$ $'a$ *clause set* **where**
grounding_of_cls $C = \{C \cdot \sigma \mid \sigma.\ is\_ground\_subst\ \sigma\}$

definition *grounding_of_clss* :: $'a$ *clause set* $\Rightarrow$ $'a$ *clause set* **where**
grounding_of_clss $CC = (\bigcup C \in CC.\ grounding\_of\_cls\ C)$

definition *is_unifier* :: $'s \Rightarrow 'a$ *set* $\Rightarrow$ *bool* **where**
is_unifier σ $AA \longleftrightarrow card\ (AA \cdot as\ \sigma) \leq 1$

definition *is_unifiers* :: $'s \Rightarrow 'a$ *set set* $\Rightarrow$ *bool* **where**
is_unifiers σ $AAA \longleftrightarrow (\forall AA \in AAA.\ is\_unifier\ \sigma\ AA)$

definition *is_mgu* :: $'s \Rightarrow 'a\ set\ set \Rightarrow bool$ **where**
is_mgu σ $AAA \longleftrightarrow$ *is_unifiers* $\sigma\ AAA \land (\forall \tau.$ *is_unifiers* $\tau\ AAA \longrightarrow (\exists \gamma.\ \tau = \sigma \odot \gamma))$

definition *var_disjoint* :: $'a\ clause\ list \Rightarrow bool$ **where**
var_disjoint $Cs \longleftrightarrow$
$(\forall \sigma s.\ length\ \sigma s = length\ Cs \longrightarrow (\exists \tau.\ \forall i < length\ Cs.\ \forall S.\ S \subseteq\#\ Cs\ !\ i \longrightarrow S \cdot \sigma s\ !\ i = S \cdot \tau))$

end

7.3 Substitution Lemmas

locale *substitution* = *substitution_ops subst_atm id_subst comp_subst*
for
subst_atm :: $'a \Rightarrow 's \Rightarrow 'a$ **and**
id_subst :: $'s$ **and**
comp_subst :: $'s \Rightarrow 's \Rightarrow 's$ +
fixes
atm_of_atms :: $'a\ list \Rightarrow 'a$ **and**
renamings_apart :: $'a\ clause\ list \Rightarrow 's\ list$
assumes
subst_atm_id_subst[*simp*]: $A \cdot a\ id\_subst = A$ **and**
subst_atm_comp_subst[*simp*]: $A \cdot a\ (\tau \odot \sigma) = (A \cdot a\ \tau) \cdot a\ \sigma$ **and**
subst_ext: $(\bigwedge A.\ A \cdot a\ \sigma = A \cdot a\ \tau) \Longrightarrow \sigma = \tau$ **and**
make_ground_subst: *is_ground_cls* $(C \cdot \sigma) \Longrightarrow \exists \tau.$ *is_ground_subst* $\tau \land C \cdot \tau = C \cdot \sigma$ **and**
renames_apart:
$\bigwedge Cs.\ length\ (renamings\_apart\ Cs) = length\ Cs\ \land$
$(\forall \varrho \in set\ (renamings\_apart\ Cs).\ is\_renaming\ \varrho)\ \land$
$var\_disjoint\ (Cs \cdot\cdot cl\ (renamings\_apart\ Cs))$ **and**
atm_of_atms_subst:
$\bigwedge As\ Bs.\ atm\_of\_atms\ As \cdot a\ \sigma = atm\_of\_atms\ Bs \longleftrightarrow map\ (\lambda A.\ A \cdot a\ \sigma)\ As = Bs$ **and**
wf_strictly_generalizes_atm: *wfP strictly_generalizes_atm*
begin

lemma *subst_ext_iff*: $\sigma = \tau \longleftrightarrow (\forall A.\ A \cdot a\ \sigma = A \cdot a\ \tau)$
by (*blast intro*: *subst_ext*)

7.3.1 Identity Substitution

lemma *id_subst_comp_subst*[*simp*]: $id\_subst \odot \sigma = \sigma$
by (*rule subst_ext*) *simp*

lemma *comp_subst_id_subst*[*simp*]: $\sigma \odot id\_subst = \sigma$
by (*rule subst_ext*) *simp*

lemma *id_subst_comp_substs*[*simp*]: $replicate\ (length\ \sigma s)\ id\_subst \odot s\ \sigma s = \sigma s$
using *comp_substs_def* **by** (*induction* σs) *auto*

lemma *comp_substs_id_subst*[*simp*]: $\sigma s \odot s\ replicate\ (length\ \sigma s)\ id\_subst = \sigma s$
using *comp_substs_def* **by** (*induction* σs) *auto*

lemma *subst_atms_id_subst*[*simp*]: $AA \cdot as\ id\_subst = AA$
unfolding *subst_atms_def* **by** *simp*

lemma *subst_atmss_id_subst*[*simp*]: $AAA \cdot ass\ id\_subst = AAA$
unfolding *subst_atmss_def* **by** *simp*

lemma *subst_atm_list_id_subst*[*simp*]: $As \cdot al\ id\_subst = As$
unfolding *subst_atm_list_def* **by** *auto*

lemma *subst_atm_mset_id_subst*[*simp*]: $AA \cdot am\ id\_subst = AA$
unfolding *subst_atm_mset_def* **by** *simp*

lemma *subst_atm_mset_list_id_subst*[*simp*]: $AAs \cdot aml\ id\_subst = AAs$

unfolding *subst_atm_mset_list_def* **by** *simp*

lemma *subst_atm_mset_lists_id_subst*[*simp*]: AAs ·· *aml replicate* (*length* AAs) *id_subst* = AAs
unfolding *subst_atm_mset_lists_def* **by** (*induct* AAs) *auto*

lemma *subst_lit_id_subst*[*simp*]: L ·*l id_subst* = L
unfolding *subst_lit_def* **by** (*simp add*: *literal.map_ident*)

lemma *subst_cls_id_subst*[*simp*]: C · *id_subst* = C
unfolding *subst_cls_def* **by** *simp*

lemma *subst_clss_id_subst*[*simp*]: CC ·*cs id_subst* = CC
unfolding *subst_clss_def* **by** *simp*

lemma *subst_cls_list_id_subst*[*simp*]: Cs ·*cl id_subst* = Cs
unfolding *subst_cls_list_def* **by** *simp*

lemma *subst_cls_lists_id_subst*[*simp*]: Cs ··*cl replicate* (*length* Cs) *id_subst* = Cs
unfolding *subst_cls_lists_def* **by** (*induct* Cs) *auto*

lemma *subst_cls_mset_id_subst*[*simp*]: CC ·*cm id_subst* = CC
unfolding *subst_cls_mset_def* **by** *simp*

7.3.2 Associativity of Composition

lemma *comp_subst_assoc*[*simp*]: $\sigma \odot (\tau \odot \gamma) = \sigma \odot \tau \odot \gamma$
by (*rule subst_ext*) *simp*

7.3.3 Compatibility of Substitution and Composition

lemma *subst_atms_comp_subst*[*simp*]: AA ·*as* $(\tau \odot \sigma)$ = AA ·*as* τ ·*as* σ
unfolding *subst_atms_def* **by** *auto*

lemma *subst_atmss_comp_subst*[*simp*]: AAA ·*ass* $(\tau \odot \sigma)$ = AAA ·*ass* τ ·*ass* σ
unfolding *subst_atmss_def* **by** *auto*

lemma *subst_atm_list_comp_subst*[*simp*]: As ·*al* $(\tau \odot \sigma)$ = As ·*al* τ ·*al* σ
unfolding *subst_atm_list_def* **by** *auto*

lemma *subst_atm_mset_comp_subst*[*simp*]: AA ·*am* $(\tau \odot \sigma)$ = AA ·*am* τ ·*am* σ
unfolding *subst_atm_mset_def* **by** *auto*

lemma *subst_atm_mset_list_comp_subst*[*simp*]: AAs ·*aml* $(\tau \odot \sigma)$ = (AAs ·*aml* τ) ·*aml* σ
unfolding *subst_atm_mset_list_def* **by** *auto*

lemma *subst_atm_mset_lists_comp_substs*[*simp*]: AAs ··*aml* (τs ⊙*s* σs) = AAs ··*aml* τs ··*aml* σs
unfolding *subst_atm_mset_lists_def comp_substs_def map_zip_map map_zip_map2 map_zip_assoc*
by (*simp add*: *split_def*)

lemma *subst_lit_comp_subst*[*simp*]: L ·*l* $(\tau \odot \sigma)$ = L ·*l* τ ·*l* σ
unfolding *subst_lit_def* **by** (*auto simp*: *literal.map_comp o_def*)

lemma *subst_cls_comp_subst*[*simp*]: $C \cdot (\tau \odot \sigma) = C \cdot \tau \cdot \sigma$
unfolding *subst_cls_def* **by** *auto*

lemma *subst_clsscomp_subst*[*simp*]: CC ·*cs* $(\tau \odot \sigma)$ = CC ·*cs* τ ·*cs* σ
unfolding *subst_clss_def* **by** *auto*

lemma *subst_cls_list_comp_subst*[*simp*]: Cs ·*cl* $(\tau \odot \sigma)$ = Cs ·*cl* τ ·*cl* σ
unfolding *subst_cls_list_def* **by** *auto*

lemma *subst_cls_lists_comp_substs*[*simp*]: Cs ··*cl* (τs ⊙*s* σs) = Cs ··*cl* τs ··*cl* σs
unfolding *subst_cls_lists_def comp_substs_def map_zip_map map_zip_map2 map_zip_assoc*
by (*simp add*: *split_def*)

lemma *subst_cls_mset_comp_subst*[*simp*]: $CC \cdot cm\ (\tau \odot \sigma) = CC \cdot cm\ \tau \cdot cm\ \sigma$
  unfolding *subst_cls_mset_def* **by** *auto*

7.3.4 "Commutativity" of Membership and Substitution

lemma *Melem_subst_atm_mset*[*simp*]: $A \in\#\ AA \cdot am\ \sigma \longleftrightarrow (\exists B.\ B \in\#\ AA \wedge A = B \cdot a\ \sigma)$
  unfolding *subst_atm_mset_def* **by** *auto*

lemma *Melem_subst_cls*[*simp*]: $L \in\#\ C \cdot \sigma \longleftrightarrow (\exists M.\ M \in\#\ C \wedge L = M \cdot l\ \sigma)$
  unfolding *subst_cls_def* **by** *auto*

lemma *Melem_subst_cls_mset*[*simp*]: $AA \in\#\ CC \cdot cm\ \sigma \longleftrightarrow (\exists BB.\ BB \in\#\ CC \wedge AA = BB \cdot \sigma)$
  unfolding *subst_cls_mset_def* **by** *auto*

7.3.5 Signs and Substitutions

lemma *subst_lit_is_neg*[*simp*]: *is_neg* $(L \cdot l\ \sigma) =$ *is_neg* L
  unfolding *subst_lit_def* **by** *auto*

lemma *subst_lit_is_pos*[*simp*]: *is_pos* $(L \cdot l\ \sigma) =$ *is_pos* L
  unfolding *subst_lit_def* **by** *auto*

lemma *subst_minus*[*simp*]: $(-\ L) \cdot l\ \mu = -\ (L\ \cdot l\ \mu)$
  by (*simp add*: *literal.map_sel subst_lit_def uminus_literal_def*)

7.3.6 Substitution on Literal(s)

lemma *eql_neg_lit_eql_atm*[*simp*]: $(Neg\ A' \cdot l\ \eta) = Neg\ A \longleftrightarrow A' \cdot a\ \eta = A$
  by (*simp add*: *subst_lit_def*)

lemma *eql_pos_lit_eql_atm*[*simp*]: $(Pos\ A' \cdot l\ \eta) = Pos\ A \longleftrightarrow A' \cdot a\ \eta = A$
  by (*simp add*: *subst_lit_def*)

lemma *subst_cls_negs*[*simp*]: $(negs\ AA) \cdot \sigma = negs\ (AA \cdot am\ \sigma)$
  unfolding *subst_cls_def subst_lit_def subst_atm_mset_def* **by** *auto*

lemma *subst_cls_poss*[*simp*]: $(poss\ AA) \cdot \sigma = poss\ (AA \cdot am\ \sigma)$
  unfolding *subst_cls_def subst_lit_def subst_atm_mset_def* **by** *auto*

lemma *atms_of_subst_atms*: *atms_of* $C \cdot as\ \sigma =$ *atms_of* $(C \cdot \sigma)$
proof −
  have *atms_of* $(C \cdot \sigma) =$ *set_mset* (*image_mset atm_of* (*image_mset* (*map_literal* $(\lambda A.\ A \cdot a\ \sigma)$) C)))
    unfolding *subst_cls_def subst_atms_def subst_lit_def atms_of_def* **by** *auto*
  also have ... = *set_mset* (*image_mset* $(\lambda A.\ A \cdot a\ \sigma)$ (*image_mset atm_of* C))
    by *simp* (*meson literal.map_sel*)
  finally show *atms_of* $C \cdot as\ \sigma =$ *atms_of* $(C \cdot \sigma)$
    unfolding *subst_atms_def atms_of_def* **by** *auto*
qed

lemma *in_image_Neg_is_neg*[*simp*]: $L \cdot l\ \sigma \in Neg$ ‘ $AA \Longrightarrow$ *is_neg* L
  by (*metis bex_imageD literal.disc*(*2*) *literal.map_disc_iff subst_lit_def*)

lemma *subst_lit_in_negs_subst_is_neg*: $L \cdot l\ \sigma \in\# (negs\ AA) \cdot \tau \Longrightarrow$ *is_neg* L
  by *simp*

lemma *subst_lit_in_negs_is_neg*: $L \cdot l\ \sigma \in\#\ negs\ AA \Longrightarrow$ *is_neg* L
  by *simp*

7.3.7 Substitution on Empty

lemma *subst_atms_empty*[*simp*]: $\{\} \cdot as\ \sigma = \{\}$
  unfolding *subst_atms_def* **by** *auto*

lemma *subst_atmss_empty*[*simp*]: $\{\} \cdot ass\ \sigma = \{\}$

unfolding *subst_atmss_def* **by** *auto*

lemma *comp_substs_empty_iff* [*simp*]: $\sigma s \odot s\ \eta s = [] \longleftrightarrow \sigma s = [] \lor \eta s = []$
using *comp_substs_def map2_empty_iff* **by** *auto*

lemma *subst_atm_list_empty*[*simp*]: $[] \cdot al\ \sigma = []$
unfolding *subst_atm_list_def* **by** *auto*

lemma *subst_atm_mset_empty*[*simp*]: $\{\#\} \cdot am\ \sigma = \{\#\}$
unfolding *subst_atm_mset_def* **by** *auto*

lemma *subst_atm_mset_list_empty*[*simp*]: $[] \cdot aml\ \sigma = []$
unfolding *subst_atm_mset_list_def* **by** *auto*

lemma *subst_atm_mset_lists_empty*[*simp*]: $[] \cdot\cdot aml\ \sigma s = []$
unfolding *subst_atm_mset_lists_def* **by** *auto*

lemma *subst_cls_empty*[*simp*]: $\{\#\} \cdot \sigma = \{\#\}$
unfolding *subst_cls_def* **by** *auto*

lemma *subst_clss_empty*[*simp*]: $\{\} \cdot cs\ \sigma = \{\}$
unfolding *subst_clss_def* **by** *auto*

lemma *subst_cls_list_empty*[*simp*]: $[] \cdot cl\ \sigma = []$
unfolding *subst_cls_list_def* **by** *auto*

lemma *subst_cls_lists_empty*[*simp*]: $[] \cdot\cdot cl\ \sigma s = []$
unfolding *subst_cls_lists_def* **by** *auto*

lemma *subst_scls_mset_empty*[*simp*]: $\{\#\} \cdot cm\ \sigma = \{\#\}$
unfolding *subst_cls_mset_def* **by** *auto*

lemma *subst_atms_empty_iff* [*simp*]: $AA \cdot as\ \eta = \{\} \longleftrightarrow AA = \{\}$
unfolding *subst_atms_def* **by** *auto*

lemma *subst_atmss_empty_iff* [*simp*]: $AAA \cdot ass\ \eta = \{\} \longleftrightarrow AAA = \{\}$
unfolding *subst_atmss_def* **by** *auto*

lemma *subst_atm_list_empty_iff* [*simp*]: $As \cdot al\ \eta = [] \longleftrightarrow As = []$
unfolding *subst_atm_list_def* **by** *auto*

lemma *subst_atm_mset_empty_iff* [*simp*]: $AA \cdot am\ \eta = \{\#\} \longleftrightarrow AA = \{\#\}$
unfolding *subst_atm_mset_def* **by** *auto*

lemma *subst_atm_mset_list_empty_iff* [*simp*]: $AAs \cdot aml\ \eta = [] \longleftrightarrow AAs = []$
unfolding *subst_atm_mset_list_def* **by** *auto*

lemma *subst_atm_mset_lists_empty_iff* [*simp*]: $AAs \cdot\cdot aml\ \eta s = [] \longleftrightarrow (AAs = [] \lor \eta s = [])$
using *map2_empty_iff subst_atm_mset_lists_def* **by** *auto*

lemma *subst_cls_empty_iff* [*simp*]: $C \cdot \eta = \{\#\} \longleftrightarrow C = \{\#\}$
unfolding *subst_cls_def* **by** *auto*

lemma *subst_clss_empty_iff* [*simp*]: $CC \cdot cs\ \eta = \{\} \longleftrightarrow CC = \{\}$
unfolding *subst_clss_def* **by** *auto*

lemma *subst_cls_list_empty_iff* [*simp*]: $Cs \cdot cl\ \eta = [] \longleftrightarrow Cs = []$
unfolding *subst_cls_list_def* **by** *auto*

lemma *subst_cls_lists_empty_iff* [*simp*]: $Cs \cdot\cdot cl\ \eta s = [] \longleftrightarrow (Cs = [] \lor \eta s = [])$
using *map2_empty_iff subst_cls_lists_def* **by** *auto*

lemma *subst_cls_mset_empty_iff* [*simp*]: $CC \cdot cm\ \eta = \{\#\} \longleftrightarrow CC = \{\#\}$

unfolding *subst_cls_mset_def* **by** *auto*

7.3.8 Substitution on a Union

lemma *subst_atms_union*[*simp*]: $(AA \cup BB) \cdot as\ \sigma = AA \cdot as\ \sigma \cup BB \cdot as\ \sigma$
unfolding *subst_atms_def* **by** *auto*

lemma *subst_atmss_union*[*simp*]: $(AAA \cup BBB) \cdot ass\ \sigma = AAA \cdot ass\ \sigma \cup BBB \cdot ass\ \sigma$
unfolding *subst_atmss_def* **by** *auto*

lemma *subst_atm_list_append*[*simp*]: $(As\ @\ Bs) \cdot al\ \sigma = As \cdot al\ \sigma\ @\ Bs \cdot al\ \sigma$
unfolding *subst_atm_list_def* **by** *auto*

lemma *subst_atm_mset_union*[*simp*]: $(AA + BB) \cdot am\ \sigma = AA \cdot am\ \sigma + BB \cdot am\ \sigma$
unfolding *subst_atm_mset_def* **by** *auto*

lemma *subst_atm_mset_list_append*[*simp*]: $(AAs\ @\ BBs) \cdot aml\ \sigma = AAs \cdot aml\ \sigma\ @\ BBs \cdot aml\ \sigma$
unfolding *subst_atm_mset_list_def* **by** *auto*

lemma *subst_cls_union*[*simp*]: $(C + D) \cdot \sigma = C \cdot \sigma + D \cdot \sigma$
unfolding *subst_cls_def* **by** *auto*

lemma *subst_clss_union*[*simp*]: $(CC \cup DD) \cdot cs\ \sigma = CC \cdot cs\ \sigma \cup DD \cdot cs\ \sigma$
unfolding *subst_clss_def* **by** *auto*

lemma *subst_cls_list_append*[*simp*]: $(Cs\ @\ Ds) \cdot cl\ \sigma = Cs \cdot cl\ \sigma\ @\ Ds \cdot cl\ \sigma$
unfolding *subst_cls_list_def* **by** *auto*

lemma *subst_cls_mset_union*[*simp*]: $(CC + DD) \cdot cm\ \sigma = CC \cdot cm\ \sigma + DD \cdot cm\ \sigma$
unfolding *subst_cls_mset_def* **by** *auto*

7.3.9 Substitution on a Singleton

lemma *subst_atms_single*[*simp*]: $\{A\} \cdot as\ \sigma = \{A \cdot a\ \sigma\}$
unfolding *subst_atms_def* **by** *auto*

lemma *subst_atmss_single*[*simp*]: $\{AA\} \cdot ass\ \sigma = \{AA \cdot as\ \sigma\}$
unfolding *subst_atmss_def* **by** *auto*

lemma *subst_atm_list_single*[*simp*]: $[A] \cdot al\ \sigma = [A \cdot a\ \sigma]$
unfolding *subst_atm_list_def* **by** *auto*

lemma *subst_atm_mset_single*[*simp*]: $\{\#A\#\} \cdot am\ \sigma = \{\#A \cdot a\ \sigma\#\}$
unfolding *subst_atm_mset_def* **by** *auto*

lemma *subst_atm_mset_list*[*simp*]: $[AA] \cdot aml\ \sigma = [AA \cdot am\ \sigma]$
unfolding *subst_atm_mset_list_def* **by** *auto*

lemma *subst_cls_single*[*simp*]: $\{\#L\#\} \cdot \sigma = \{\#L \cdot l\ \sigma\#\}$
by *simp*

lemma *subst_clss_single*[*simp*]: $\{C\} \cdot cs\ \sigma = \{C \cdot \sigma\}$
unfolding *subst_clss_def* **by** *auto*

lemma *subst_cls_list_single*[*simp*]: $[C] \cdot cl\ \sigma = [C \cdot \sigma]$
unfolding *subst_cls_list_def* **by** *auto*

lemma *subst_cls_mset_single*[*simp*]: $\{\#C\#\} \cdot cm\ \sigma = \{\#C \cdot \sigma\#\}$
by *simp*

7.3.10 Substitution on *op*

lemma *subst_atm_list_Cons*[*simp*]: $(A\ \#\ As) \cdot al\ \sigma = A \cdot a\ \sigma\ \#\ As \cdot al\ \sigma$
unfolding *subst_atm_list_def* **by** *auto*

lemma *subst_atm_mset_list_Cons*[*simp*]: $(A \# As) \cdot aml\ \sigma = A \cdot am\ \sigma \# As \cdot aml\ \sigma$
unfolding *subst_atm_mset_list_def* **by** *auto*

lemma *subst_atm_mset_lists_Cons*[*simp*]: $(C \# Cs) \cdot\cdot aml\ (\sigma \# \sigma s) = C \cdot am\ \sigma \# Cs \cdot\cdot aml\ \sigma s$
unfolding *subst_atm_mset_lists_def* **by** *auto*

lemma *subst_cls_list_Cons*[*simp*]: $(C \# Cs) \cdot cl\ \sigma = C \cdot \sigma \# Cs \cdot cl\ \sigma$
unfolding *subst_cls_list_def* **by** *auto*

lemma *subst_cls_lists_Cons*[*simp*]: $(C \# Cs) \cdot\cdot cl\ (\sigma \# \sigma s) = C \cdot \sigma \# Cs \cdot\cdot cl\ \sigma s$
unfolding *subst_cls_lists_def* **by** *auto*

7.3.11 Substitution on *tl*

lemma *subst_atm_list_tl*[*simp*]: $tl\ (As \cdot al\ \eta) = tl\ As \cdot al\ \eta$
by (*induction As*) *auto*

lemma *subst_atm_mset_list_tl*[*simp*]: $tl\ (AAs \cdot aml\ \eta) = tl\ AAs \cdot aml\ \eta$
by (*induction AAs*) *auto*

7.3.12 Substitution on *op* !

lemma *comp_substs_nth*[*simp*]:
$length\ \tau s = length\ \sigma s \Longrightarrow i < length\ \tau s \Longrightarrow (\tau s \odot s\ \sigma s) ! i = (\tau s ! i) \odot (\sigma s ! i)$
by (*simp add*: *comp_substs_def*)

lemma *subst_atm_list_nth*[*simp*]: $i < length\ As \Longrightarrow (As \cdot al\ \tau) ! i = As ! i \cdot a\ \tau$
unfolding *subst_atm_list_def* **using** *less_Suc_eq_0_disj nth_map* **by** *force*

lemma *subst_atm_mset_list_nth*[*simp*]: $i < length\ AAs \Longrightarrow (AAs \cdot aml\ \eta) ! i = (AAs ! i) \cdot am\ \eta$
unfolding *subst_atm_mset_list_def* **by** *auto*

lemma *subst_atm_mset_lists_nth*[*simp*]:
$length\ AAs = length\ \sigma s \Longrightarrow i < length\ AAs \Longrightarrow (AAs \cdot\cdot aml\ \sigma s) ! i = (AAs ! i) \cdot am\ (\sigma s ! i)$
unfolding *subst_atm_mset_lists_def* **by** *auto*

lemma *subst_cls_list_nth*[*simp*]: $i < length\ Cs \Longrightarrow (Cs \cdot cl\ \tau) ! i = (Cs ! i) \cdot \tau$
unfolding *subst_cls_list_def* **using** *less_Suc_eq_0_disj nth_map* **by** (*induction Cs*) *auto*

lemma *subst_cls_lists_nth*[*simp*]:
$length\ Cs = length\ \sigma s \Longrightarrow i < length\ Cs \Longrightarrow (Cs \cdot\cdot cl\ \sigma s) ! i = (Cs ! i) \cdot (\sigma s ! i)$
unfolding *subst_cls_lists_def* **by** *auto*

7.3.13 Substitution on Various Other Functions

lemma *subst_clss_image*[*simp*]: $image\ f\ X \cdot cs\ \sigma = \{f\ x \cdot \sigma \mid x.\ x \in X\}$
unfolding *subst_clss_def* **by** *auto*

lemma *subst_cls_mset_image_mset*[*simp*]: $image\_mset\ f\ X \cdot cm\ \sigma = \{\# f\ x \cdot \sigma.\ x \in\# X \#\}$
unfolding *subst_cls_mset_def* **by** *auto*

lemma *mset_subst_atm_list_subst_atm_mset*[*simp*]: $mset\ (As \cdot al\ \sigma) = mset\ (As) \cdot am\ \sigma$
unfolding *subst_atm_list_def subst_atm_mset_def* **by** *auto*

lemma *mset_subst_cls_list_subst_cls_mset*: $mset\ (Cs \cdot cl\ \sigma) = (mset\ Cs) \cdot cm\ \sigma$
unfolding *subst_cls_mset_def subst_cls_list_def* **by** *auto*

lemma *sum_list_subst_cls_list_subst_cls*[*simp*]: $sum\_list\ (Cs \cdot cl\ \eta) = sum\_list\ Cs \cdot \eta$
unfolding *subst_cls_list_def* **by** (*induction Cs*) *auto*

lemma *set_mset_subst_cls_mset_subst_clss*: $set\_mset\ (CC \cdot cm\ \mu) = (set\_mset\ CC) \cdot cs\ \mu$
by (*simp add*: *subst_cls_mset_def subst_clss_def*)

lemma *Neg_Melem_subst_atm_subst_cls*[*simp*]: *Neg A* ∈# *C* ⟹ *Neg* (*A* ·*a* σ) ∈# *C* · σ
  by (*metis Melem_subst_cls eql_neg_lit_eql_atm*)

lemma *Pos_Melem_subst_atm_subst_cls*[*simp*]: *Pos A* ∈# *C* ⟹ *Pos* (*A* ·*a* σ) ∈# *C* · σ
  by (*metis Melem_subst_cls eql_pos_lit_eql_atm*)

lemma *in_atms_of_subst*[*simp*]: *B* ∈ *atms_of C* ⟹ *B* ·*a* σ ∈ *atms_of* (*C* · σ)
  by (*metis atms_of_subst_atms image_iff subst_atms_def*)

7.3.14 Renamings

lemma *is_renaming_id_subst*[*simp*]: *is_renaming id_subst*
  unfolding *is_renaming_def* **by** *simp*

lemma *is_renamingD*: *is_renaming* σ ⟹ (∀ *A1 A2*. *A1* ·*a* σ = *A2* ·*a* σ ⟷ *A1* = *A2*)
  by (*metis is_renaming_def subst_atm_comp_subst subst_atm_id_subst*)

lemma *inv_renaming_cancel_r*[*simp*]: *is_renaming r* ⟹ *r* ⊙ *inv_renaming r* = *id_subst*
  unfolding *inv_renaming_def is_renaming_def* **by** (*metis* (*mono_tags*) *someI_ex*)

lemma *inv_renaming_cancel_r_list*[*simp*]:
  is_renaming_list rs ⟹ *rs* ⊙*s map inv_renaming rs* = *replicate* (*length rs*) *id_subst*
  unfolding *is_renaming_list_def* **by** (*induction rs*) (*auto simp add*: *comp_substs_def*)

lemma *Nil_comp_substs*[*simp*]: [] ⊙*s s* = []
  unfolding *comp_substs_def* **by** *auto*

lemma *comp_substs_Nil*[*simp*]: *s* ⊙*s* [] = []
  unfolding *comp_substs_def* **by** *auto*

lemma *is_renaming_idempotent_id_subst*: *is_renaming r* ⟹ *r* ⊙ *r* = *r* ⟹ *r* = *id_subst*
  by (*metis comp_subst_assoc comp_subst_id_subst inv_renaming_cancel_r*)

lemma *is_renaming_left_id_subst_right_id_subst*:
  is_renaming r ⟹ *s* ⊙ *r* = *id_subst* ⟹ *r* ⊙ *s* = *id_subst*
  by (*metis comp_subst_assoc comp_subst_id_subst is_renaming_def*)

lemma *is_renaming_closure*: *is_renaming r1* ⟹ *is_renaming r2* ⟹ *is_renaming* (*r1* ⊙ *r2*)
  unfolding *is_renaming_def* **by** (*metis comp_subst_assoc comp_subst_id_subst*)

lemma *is_renaming_inv_renaming_cancel_atm*[*simp*]: *is_renaming* ϱ ⟹ *A* ·*a* ϱ ·*a inv_renaming* ϱ = *A*
  by (*metis inv_renaming_cancel_r subst_atm_comp_subst subst_atm_id_subst*)

lemma *is_renaming_inv_renaming_cancel_atms*[*simp*]: *is_renaming* ϱ ⟹ *AA* ·*as* ϱ ·*as inv_renaming* ϱ = *AA*
  by (*metis inv_renaming_cancel_r subst_atms_comp_subst subst_atms_id_subst*)

lemma *is_renaming_inv_renaming_cancel_atmss*[*simp*]: *is_renaming* ϱ ⟹ *AAA* ·*ass* ϱ ·*ass inv_renaming* ϱ = *AAA*
  by (*metis inv_renaming_cancel_r subst_atmss_comp_subst subst_atmss_id_subst*)

lemma *is_renaming_inv_renaming_cancel_atm_list*[*simp*]: *is_renaming* ϱ ⟹ *As* ·*al* ϱ ·*al inv_renaming* ϱ = *As*
  by (*metis inv_renaming_cancel_r subst_atm_list_comp_subst subst_atm_list_id_subst*)

lemma *is_renaming_inv_renaming_cancel_atm_mset*[*simp*]: *is_renaming* ϱ ⟹ *AA* ·*am* ϱ ·*am inv_renaming* ϱ = *AA*
  by (*metis inv_renaming_cancel_r subst_atm_mset_comp_subst subst_atm_mset_id_subst*)

lemma *is_renaming_inv_renaming_cancel_atm_mset_list*[*simp*]: *is_renaming* ϱ ⟹ (*AAs* ·*aml* ϱ) ·*aml inv_renaming* ϱ = *AAs*
  by (*metis inv_renaming_cancel_r subst_atm_mset_list_comp_subst subst_atm_mset_list_id_subst*)

lemma *is_renaming_list_inv_renaming_cancel_atm_mset_lists*[*simp*]:
  length AAs = *length* ϱ*s* ⟹ *is_renaming_list* ϱ*s* ⟹ *AAs* ··*aml* ϱ*s* ··*aml map inv_renaming* ϱ*s* = *AAs*
  by (*metis inv_renaming_cancel_r_list subst_atm_mset_lists_comp_substs subst_atm_mset_lists_id_subst*)

lemma *is_renaming_inv_renaming_cancel_lit*[*simp*]: *is_renaming* ϱ ⟹ (*L* ·*l* ϱ) ·*l inv_renaming* ϱ = *L*

by (*metis inv_renaming_cancel_r subst_lit_comp_subst subst_lit_id_subst*)

lemma *is_renaming_inv_renaming_cancel_cls*[*simp*]: *is_renaming* $\varrho \Longrightarrow C \cdot \varrho \cdot$ *inv_renaming* $\varrho = C$
by (*metis inv_renaming_cancel_r subst_cls_comp_subst subst_cls_id_subst*)

lemma *is_renaming_inv_renaming_cancel_clss*[*simp*]: *is_renaming* $\varrho \Longrightarrow CC \cdot cs\ \varrho \cdot cs$ *inv_renaming* $\varrho = CC$
by (*metis inv_renaming_cancel_r subst_clss_id_subst subst_clsscomp_subst*)

lemma *is_renaming_inv_renaming_cancel_cls_list*[*simp*]: *is_renaming* $\varrho \Longrightarrow Cs \cdot cl\ \varrho \cdot cl$ *inv_renaming* $\varrho = Cs$
by (*metis inv_renaming_cancel_r subst_cls_list_comp_subst subst_cls_list_id_subst*)

lemma *is_renaming_list_inv_renaming_cancel_cls_list*[*simp*]:
length $Cs =$ *length* $\varrho s \Longrightarrow$ *is_renaming_list* $\varrho s \Longrightarrow Cs \cdot\cdot cl\ \varrho s \cdot\cdot cl$ *map inv_renaming* $\varrho s = Cs$
by (*metis inv_renaming_cancel_r_list subst_cls_lists_comp_substs subst_cls_lists_id_subst*)

lemma *is_renaming_inv_renaming_cancel_cls_mset*[*simp*]: *is_renaming* $\varrho \Longrightarrow CC \cdot cm\ \varrho \cdot cm$ *inv_renaming* $\varrho = CC$
by (*metis inv_renaming_cancel_r subst_cls_mset_comp_subst subst_cls_mset_id_subst*)

7.3.15 Monotonicity

lemma *subst_cls_mono*: *set_mset* $C \subseteq$ *set_mset* $D \Longrightarrow$ *set_mset* $(C \cdot \sigma) \subseteq$ *set_mset* $(D \cdot \sigma)$
by *force*

lemma *subst_cls_mono_mset*: $C \subseteq\# D \Longrightarrow C \cdot \sigma \subseteq\# D \cdot \sigma$
unfolding *subst_clss_def* **by** (*metis mset_subset_eq_exists_conv subst_cls_union*)

lemma *subst_subset_mono*: $D \subset\# C \Longrightarrow D \cdot \sigma \subset\# C \cdot \sigma$
unfolding *subst_cls_def* **by** (*simp add*: *image_mset_subset_mono*)

7.3.16 Size after Substitution

lemma *size_subst*[*simp*]: *size* $(D \cdot \sigma) =$ *size* D
unfolding *subst_cls_def* **by** *auto*

lemma *subst_atm_list_length*[*simp*]: *length* $(As \cdot al\ \sigma) =$ *length* As
unfolding *subst_atm_list_def* **by** *auto*

lemma *length_subst_atm_mset_list*[*simp*]: *length* $(AAs \cdot aml\ \eta) =$ *length* AAs
unfolding *subst_atm_mset_list_def* **by** *auto*

lemma *subst_atm_mset_lists_length*[*simp*]: *length* $(AAs \cdot\cdot aml\ \sigma s) =$ *min* (*length* AAs) (*length* σs)
unfolding *subst_atm_mset_lists_def* **by** *auto*

lemma *subst_cls_list_length*[*simp*]: *length* $(Cs \cdot cl\ \sigma) =$ *length* Cs
unfolding *subst_cls_list_def* **by** *auto*

lemma *comp_substs_length*[*simp*]: *length* $(\tau s \odot s\ \sigma s) =$ *min* (*length* τs) (*length* σs)
unfolding *comp_substs_def* **by** *auto*

lemma *subst_cls_lists_length*[*simp*]: *length* $(Cs \cdot\cdot cl\ \sigma s) =$ *min* (*length* Cs) (*length* σs)
unfolding *subst_cls_lists_def* **by** *auto*

7.3.17 Variable Disjointness

lemma *var_disjoint_clauses*:
assumes *var_disjoint Cs*
shows $\forall \sigma s.$ *length* $\sigma s =$ *length* $Cs \longrightarrow (\exists \tau.\ Cs \cdot\cdot cl\ \sigma s = Cs \cdot cl\ \tau)$
proof *clarify*
fix σs :: *'s list*
assume *a*: *length* $\sigma s =$ *length* Cs
then obtain τ **where** $\forall i <$ *length* $Cs.\ \forall S.\ S \subseteq\# Cs\ !\ i \longrightarrow S \cdot \sigma s\ !\ i = S \cdot \tau$
using *assms* **unfolding** *var_disjoint_def* **by** *blast*
then have $\forall i <$ *length* $Cs.\ (Cs\ !\ i) \cdot \sigma s\ !\ i = (Cs\ !\ i) \cdot \tau$
by *auto*

then have $Cs \cdot\cdot cl\ \sigma s = Cs \cdot cl\ \tau$
  using *a* **by** (*simp add*: *nth_equalityI*)
then show $\exists\tau.\ Cs \cdot\cdot cl\ \sigma s = Cs \cdot cl\ \tau$
  by *auto*
qed

7.3.18 Ground Expressions and Substitutions

lemma *ex_ground_subst*: $\exists\sigma.$ *is_ground_subst* σ
  using *make_ground_subst*[*of* {#}]
  by (*simp add*: *is_ground_cls_def*)

lemma *is_ground_cls_list_Cons*[*simp*]:
  is_ground_cls_list (*C* # *Cs*) = (*is_ground_cls C* ∧ *is_ground_cls_list Cs*)
  unfolding *is_ground_cls_list_def* **by** *auto*

Ground union **lemma** *is_ground_atms_union*[*simp*]: *is_ground_atms* (*AA* ∪ *BB*) ⟷ *is_ground_atms AA* ∧ *is_ground_atms BB*
  unfolding *is_ground_atms_def* **by** *auto*

lemma *is_ground_atm_mset_union*[*simp*]:
  is_ground_atm_mset (*AA* + *BB*) ⟷ *is_ground_atm_mset AA* ∧ *is_ground_atm_mset BB*
  unfolding *is_ground_atm_mset_def* **by** *auto*

lemma *is_ground_cls_union*[*simp*]: *is_ground_cls* (*C* + *D*) ⟷ *is_ground_cls C* ∧ *is_ground_cls D*
  unfolding *is_ground_cls_def* **by** *auto*

lemma *is_ground_clss_union*[*simp*]:
  is_ground_clss (*CC* ∪ *DD*) ⟷ *is_ground_clss CC* ∧ *is_ground_clss DD*
  unfolding *is_ground_clss_def* **by** *auto*

lemma *is_ground_cls_list_is_ground_cls_sum_list*[*simp*]:
  is_ground_cls_list Cs ⟹ *is_ground_cls* (*sum_list Cs*)
  by (*meson in_mset_sum_list2 is_ground_cls_def is_ground_cls_list_def*)

Ground mono **lemma** *is_ground_cls_mono*: *C* ⊆# *D* ⟹ *is_ground_cls D* ⟹ *is_ground_cls C*
  unfolding *is_ground_cls_def* **by** (*metis set_mset_mono subsetD*)

lemma *is_ground_clss_mono*: *CC* ⊆ *DD* ⟹ *is_ground_clss DD* ⟹ *is_ground_clss CC*
  unfolding *is_ground_clss_def* **by** *blast*

lemma *grounding_of_clss_mono*: *CC* ⊆ *DD* ⟹ *grounding_of_clss CC* ⊆ *grounding_of_clss DD*
  using *grounding_of_clss_def* **by** *auto*

lemma *sum_list_subseteq_mset_is_ground_cls_list*[*simp*]:
  sum_list Cs ⊆# *sum_list Ds* ⟹ *is_ground_cls_list Ds* ⟹ *is_ground_cls_list Cs*
  by (*meson in_mset_sum_list is_ground_cls_def is_ground_cls_list_is_ground_cls_sum_list*
    is_ground_cls_mono is_ground_cls_list_def)

Substituting on ground expression preserves ground **lemma** *is_ground_comp_subst*[*simp*]: *is_ground_subst* σ ⟹ *is_ground_subst* ($\tau \odot \sigma$)
  unfolding *is_ground_subst_def is_ground_atm_def* **by** *auto*

lemma *ground_subst_ground_atm*[*simp*]: *is_ground_subst* σ ⟹ *is_ground_atm* ($A \cdot a\ \sigma$)
  by (*simp add*: *is_ground_subst_def*)

lemma *ground_subst_ground_lit*[*simp*]: *is_ground_subst* σ ⟹ *is_ground_lit* ($L \cdot l\ \sigma$)
  unfolding *is_ground_lit_def subst_lit_def* **by** (*cases L*) *auto*

lemma *ground_subst_ground_cls*[*simp*]: *is_ground_subst* σ ⟹ *is_ground_cls* ($C \cdot \sigma$)
  unfolding *is_ground_cls_def* **by** *auto*

lemma *ground_subst_ground_clss*[*simp*]: *is_ground_subst* σ ⟹ *is_ground_clss* ($CC \cdot cs\ \sigma$)

unfolding *is_ground_clss_def subst_clss_def* **by** *auto*

lemma *ground_subst_ground_cls_list*[*simp*]: *is_ground_subst* $\sigma \Longrightarrow$ *is_ground_cls_list* (*Cs* $\cdot$*cl* σ)
unfolding *is_ground_cls_list_def subst_cls_list_def* **by** *auto*

lemma *ground_subst_ground_cls_lists*[*simp*]:
$\forall \sigma \in$ *set* σs. *is_ground_subst* $\sigma \Longrightarrow$ *is_ground_cls_list* (*Cs* $\cdot\cdot$*cl* σs)
unfolding *is_ground_cls_list_def subst_cls_lists_def* **by** (*auto simp*: *set_zip*)

Substituting on ground expression has no effect **lemma** *is_ground_subst_atm*[*simp*]: *is_ground_atm A* $\Longrightarrow A \cdot$*a* $\sigma = A$
unfolding *is_ground_atm_def* **by** *simp*

lemma *is_ground_subst_atms*[*simp*]: *is_ground_atms AA* $\Longrightarrow AA \cdot$*as* $\sigma = AA$
unfolding *is_ground_atms_def subst_atms_def image_def* **by** *auto*

lemma *is_ground_subst_atm_mset*[*simp*]: *is_ground_atm_mset AA* $\Longrightarrow AA \cdot$*am* $\sigma = AA$
unfolding *is_ground_atm_mset_def subst_atm_mset_def* **by** *auto*

lemma *is_ground_subst_atm_list*[*simp*]: *is_ground_atm_list As* $\Longrightarrow As \cdot$*al* $\sigma = As$
unfolding *is_ground_atm_list_def subst_atm_list_def* **by** (*auto intro*: *nth_equalityI*)

lemma *is_ground_subst_atm_list_member*[*simp*]:
is_ground_atm_list As $\Longrightarrow i <$ *length As* $\Longrightarrow As \mathbin{!} i \cdot$*a* $\sigma = As \mathbin{!} i$
unfolding *is_ground_atm_list_def* **by** *auto*

lemma *is_ground_subst_lit*[*simp*]: *is_ground_lit L* $\Longrightarrow L \cdot$*l* $\sigma = L$
unfolding *is_ground_lit_def subst_lit_def* **by** (*cases L*) *simp_all*

lemma *is_ground_subst_cls*[*simp*]: *is_ground_cls C* $\Longrightarrow C \cdot \sigma = C$
unfolding *is_ground_cls_def subst_cls_def* **by** *simp*

lemma *is_ground_subst_clss*[*simp*]: *is_ground_clss CC* $\Longrightarrow CC \cdot$*cs* $\sigma = CC$
unfolding *is_ground_clss_def subst_clss_def image_def* **by** *auto*

lemma *is_ground_subst_cls_lists*[*simp*]:
assumes *length P* = *length Cs* **and** *is_ground_cls_list Cs*
shows *Cs* $\cdot\cdot$*cl* $P = Cs$
using *assms* **by** (*metis is_ground_cls_list_def is_ground_subst_cls min.idem nth_equalityI nth_mem subst_cls_lists_nth subst_cls_lists_length*)

lemma *is_ground_subst_lit_iff*: *is_ground_lit L* $\longleftrightarrow (\forall \sigma.\ L = L \cdot$*l* $\sigma)$
using *is_ground_atm_def is_ground_lit_def subst_lit_def* **by** (*cases L*) *auto*

lemma *is_ground_subst_cls_iff*: *is_ground_cls C* $\longleftrightarrow (\forall \sigma.\ C = C \cdot \sigma)$
by (*metis ex_ground_subst ground_subst_ground_cls is_ground_subst_cls*)

Members of ground expressions are ground **lemma** *is_ground_cls_as_atms*: *is_ground_cls C* $\longleftrightarrow (\forall A \in$ *atms_of C*. *is_ground_atm A*)
by (*auto simp*: *atms_of_def is_ground_cls_def is_ground_lit_def*)

lemma *is_ground_cls_imp_is_ground_lit*: $L \in\#\ C \Longrightarrow$ *is_ground_cls C* $\Longrightarrow$ *is_ground_lit L*
by (*simp add*: *is_ground_cls_def*)

lemma *is_ground_cls_imp_is_ground_atm*: $A \in$ *atms_of C* $\Longrightarrow$ *is_ground_cls C* $\Longrightarrow$ *is_ground_atm A*
by (*simp add*: *is_ground_cls_as_atms*)

lemma *is_ground_cls_is_ground_atms_atms_of*[*simp*]: *is_ground_cls C* $\Longrightarrow$ *is_ground_atms* (*atms_of C*)
by (*simp add*: *is_ground_cls_imp_is_ground_atm is_ground_atms_def*)

lemma *grounding_ground*: $C \in$ *grounding_of_clss M* $\Longrightarrow$ *is_ground_cls C*
unfolding *grounding_of_clss_def grounding_of_cls_def* **by** *auto*

lemma *in_subset_eq_grounding_of_clss_is_ground_cls*[*simp*]:
  C $\in$ *CC* $\Longrightarrow$ *CC* $\subseteq$ *grounding_of_clss DD* $\Longrightarrow$ *is_ground_cls C*
  unfolding *grounding_of_clss_def grounding_of_cls_def* **by** *auto*

lemma *is_ground_cls_empty*[*simp*]: *is_ground_cls* {#}
  unfolding *is_ground_cls_def* **by** *simp*

lemma *grounding_of_cls_ground*: *is_ground_cls C* $\Longrightarrow$ *grounding_of_cls C* = {*C*}
  unfolding *grounding_of_cls_def* **by** (*simp add*: *ex_ground_subst*)

lemma *grounding_of_cls_empty*[*simp*]: *grounding_of_cls* {#} = {{#}}
  by (*simp add*: *grounding_of_cls_ground*)

7.3.19 Subsumption

lemma *subsumes_empty_left*[*simp*]: *subsumes* {#} *C*
  unfolding *subsumes_def subst_cls_def* **by** *simp*

lemma *strictly_subsumes_empty_left*[*simp*]: *strictly_subsumes* {#} *C* $\longleftrightarrow$ *C* $\neq$ {#}
  unfolding *strictly_subsumes_def subsumes_def subst_cls_def* **by** *simp*

7.3.20 Unifiers

lemma *card_le_one_alt*: *finite X* $\Longrightarrow$ *card X* $\leq$ *1* $\longleftrightarrow$ *X* = {} $\vee$ ($\exists$ *x*. *X* = {*x*})
  by (*induct rule*: *finite_induct*) *auto*

lemma *is_unifier_subst_atm_eqI*:
  assumes *finite AA*
  shows *is_unifier* σ *AA* $\Longrightarrow$ *A* $\in$ *AA* $\Longrightarrow$ *B* $\in$ *AA* $\Longrightarrow$ *A* $\cdot a$ σ = *B* $\cdot a$ σ
  unfolding *is_unifier_def subst_atms_def card_le_one_alt*[*OF finite_imageI*[*OF assms*]]
  by (*metis equals0D imageI insert_iff*)

lemma *is_unifier_alt*:
  assumes *finite AA*
  shows *is_unifier* σ *AA* $\longleftrightarrow$ ($\forall$ *A* $\in$ *AA*. $\forall$ *B* $\in$ *AA*. *A* $\cdot a$ σ = *B* $\cdot a$ σ)
  unfolding *is_unifier_def subst_atms_def card_le_one_alt*[*OF finite_imageI*[*OF assms*(*1*)]]
  by (*rule iffI*, *metis empty_iff insert_iff insert_image*, *blast*)

lemma *is_unifiers_subst_atm_eqI*:
  assumes *finite AA is_unifiers* σ *AAA AA* $\in$ *AAA A* $\in$ *AA B* $\in$ *AA*
  shows *A* $\cdot a$ σ = *B* $\cdot a$ σ
  by (*metis assms is_unifiers_def is_unifier_subst_atm_eqI*)

theorem *is_unifiers_comp*:
  is_unifiers σ (*set_mset* ‘ *set* (*map2 add_mset As Bs*) $\cdot ass$ η) $\longleftrightarrow$
   is_unifiers ($\eta \odot \sigma$) (*set_mset* ‘ *set* (*map2 add_mset As Bs*))
  unfolding *is_unifiers_def is_unifier_def subst_atmss_def* **by** *auto*

7.3.21 Most General Unifier

lemma *is_mgu_is_unifiers*: *is_mgu* σ *AAA* $\Longrightarrow$ *is_unifiers* σ *AAA*
  using *is_mgu_def* **by** *blast*

lemma *is_mgu_is_most_general*: *is_mgu* σ *AAA* $\Longrightarrow$ *is_unifiers* τ *AAA* $\Longrightarrow$ $\exists\gamma$. $\tau = \sigma \odot \gamma$
  using *is_mgu_def* **by** *blast*

lemma *is_unifiers_is_unifier*: *is_unifiers* σ *AAA* $\Longrightarrow$ *AA* $\in$ *AAA* $\Longrightarrow$ *is_unifier* σ *AA*
  using *is_unifiers_def* **by** *simp*

7.3.22 Generalization and Subsumption

lemma *variants_iff_subsumes*: *variants C D* $\longleftrightarrow$ *subsumes C D* $\wedge$ *subsumes D C*
proof
  assume *variants C D*
  then show *subsumes C D* $\wedge$ *subsumes D C*

```
    unfolding variants_def generalizes_cls_def subsumes_def by (metis subset_mset.order.refl)
next
  assume sub: subsumes C D ∧ subsumes D C
  then have size C = size D
    unfolding subsumes_def by (metis antisym size_mset_mono size_subst)
  then show variants C D
    using sub unfolding subsumes_def variants_def generalizes_cls_def
    by (metis leD mset_subset_size size_mset_mono size_subst
        subset_mset.order.not_eq_order_implies_strict)
qed

lemma wf_strictly_generalizes_cls: wfP strictly_generalizes_cls
proof −
  {
    assume ∃ C_at. ∀ i. strictly_generalizes_cls (C_at (Suc i)) (C_at i)
    then obtain C_at :: nat ⇒ 'a clause where
      sg_C: ⋀i. strictly_generalizes_cls (C_at (Suc i)) (C_at i)
      by blast

    define n :: nat where
      n = size (C_at 0)

    have sz_C: size (C_at i) = n for i
    proof (induct i)
      case (Suc i)
      then show ?case
        using sg_C[of i] unfolding strictly_generalizes_cls_def generalizes_cls_def subst_cls_def
        by (metis size_image_mset)
    qed (simp add: n_def)

    obtain σ_at :: nat ⇒ 's where
      C_σ: ⋀i. image_mset (λL. L ·l σ_at i) (C_at (Suc i)) = C_at i
      using sg_C[unfolded strictly_generalizes_cls_def generalizes_cls_def subst_cls_def] by metis

    define Ls_at :: nat ⇒ 'a literal list where
      Ls_at = rec_nat (SOME Ls. mset Ls = C_at 0)
        (λi Lsi. SOME Ls. mset Ls = C_at (Suc i) ∧ map (λL. L ·l σ_at i) Ls = Lsi)

    have
      Ls_at_0: Ls_at 0 = (SOME Ls. mset Ls = C_at 0) and
      Ls_at_Suc: ⋀i. Ls_at (Suc i) =
        (SOME Ls. mset Ls = C_at (Suc i) ∧ map (λL. L ·l σ_at i) Ls = Ls_at i)
      unfolding Ls_at_def by simp+

    have mset_Lt_at_0: mset (Ls_at 0) = C_at 0
      unfolding Ls_at_0 by (rule someI_ex) (metis list_of_mset_exi)

    have mset (Ls_at (Suc i)) = C_at (Suc i) ∧ map (λL. L ·l σ_at i) (Ls_at (Suc i)) = Ls_at i
      for i
    proof (induct i)
      case 0
      then show ?case
        by (simp add: Ls_at_Suc, rule someI_ex,
            metis C_σ image_mset_of_subset_list mset_Lt_at_0)
    next
      case Suc
      then show ?case
        by (subst (1 2) Ls_at_Suc) (rule someI_ex, metis C_σ image_mset_of_subset_list)
    qed
    note mset_Ls = this[THEN conjunct1] and Ls_σ = this[THEN conjunct2]

    have len_Ls: ⋀i. length (Ls_at i) = n
      by (metis mset_Ls mset_Lt_at_0 not0_implies_Suc size_mset sz_C)
```

```
    have is_pos_Ls: ⋀i j. j < n ⟹ is_pos (Ls_at (Suc i) ! j) ⟷ is_pos (Ls_at i ! j)
      using Ls_σ len_Ls by (metis literal.map_disc_iff nth_map subst_lit_def)

    have Ls_τ_strict_lit: ⋀i τ. map (λL. L ·l τ) (Ls_at i) ≠ Ls_at (Suc i)
      by (metis C_σ mset_Ls Ls_σ mset_map sg_C generalizes_cls_def strictly_generalizes_cls_def
          subst_cls_def)

    have Ls_τ_strict_tm:
      map ((λt. t ·a τ) ∘ atm_of) (Ls_at i) ≠ map atm_of (Ls_at (Suc i)) for i τ
    proof −
      obtain j :: nat where
        j_lt: j < n and
        j_τ: Ls_at i ! j ·l τ ≠ Ls_at (Suc i) ! j
        using Ls_τ_strict_lit[of τ i] len_Ls
        by (metis (no_types, lifting) length_map list_eq_iff_nth_eq nth_map)

      have atm_of (Ls_at i ! j) ·a τ ≠ atm_of (Ls_at (Suc i) ! j)
        using j_τ is_pos_Ls[OF j_lt]
        by (metis (mono_guards) literal.expand literal.map_disc_iff literal.map_sel subst_lit_def)
      then show ?thesis
        using j_lt len_Ls by (metis nth_map o_apply)
    qed

    define tm_at :: nat ⇒ 'a where
      ⋀i. tm_at i = atm_of_atms (map atm_of (Ls_at i))

    have ⋀i. generalizes_atm (tm_at (Suc i)) (tm_at i)
      unfolding tm_at_def generalizes_atm_def atm_of_atms_subst
      using Ls_σ[THEN arg_cong, of map atm_of] by (auto simp: comp_def)
    moreover have ⋀i. ¬ generalizes_atm (tm_at i) (tm_at (Suc i))
      unfolding tm_at_def generalizes_atm_def atm_of_atms_subst by (simp add: Ls_τ_strict_tm)
    ultimately have ⋀i. strictly_generalizes_atm (tm_at (Suc i)) (tm_at i)
      unfolding strictly_generalizes_atm_def by blast
    then have False
      using wf_strictly_generalizes_atm[unfolded wfP_def wf_iff_no_infinite_down_chain] by blast
  }
  then show wfP (strictly_generalizes_cls :: 'a clause ⇒ _ ⇒ _)
    unfolding wfP_def by (blast intro: wf_iff_no_infinite_down_chain[THEN iffD2])
qed

lemma strict_subset_subst_strictly_subsumes:
  assumes cη_sub: C · η ⊂# D
  shows strictly_subsumes C D
  by (metis cη_sub leD mset_subset_size size_mset_mono size_subst strictly_subsumes_def
      subset_mset.dual_order.strict_implies_order substitution_ops.subsumes_def)

lemma subsumes_trans: subsumes C D ⟹ subsumes D E ⟹ subsumes C E
  unfolding subsumes_def
  by (metis (no_types) subset_mset.order.trans subst_cls_comp_subst subst_cls_mono_mset)

lemma subset_strictly_subsumes: C ⊂# D ⟹ strictly_subsumes C D
  using strict_subset_subst_strictly_subsumes[of C id_subst] by auto

lemma strictly_subsumes_neq: strictly_subsumes D' D ⟹ D' ≠ D · σ
  unfolding strictly_subsumes_def subsumes_def by blast

lemma strictly_subsumes_has_minimum:
  assumes CC ≠ {}
  shows ∃ C ∈ CC. ∀ D ∈ CC. ¬ strictly_subsumes D C
proof (rule ccontr)
  assume ¬ (∃ C ∈ CC. ∀ D∈CC. ¬ strictly_subsumes D C)
  then have ∀ C ∈ CC. ∃ D ∈ CC. strictly_subsumes D C
```

```
    by blast
  then obtain f where
    f_p: ∀ C ∈ CC. f C ∈ CC ∧ strictly_subsumes (f C) C
    by metis
  from assms obtain C where
    C_p: C ∈ CC
    by auto

  define c :: nat ⇒ 'a clause where
    ⋀n. c n = (f ^^ n) C

  have incc: c i ∈ CC for i
    by (induction i) (auto simp: c_def f_p C_p)
  have ps: ∀ i. strictly_subsumes (c (Suc i)) (c i)
    using incc f_p unfolding c_def by auto
  have ∀ i. size (c i) ≥ size (c (Suc i))
    using ps unfolding strictly_subsumes_def subsumes_def by (metis size_mset_mono size_subst)
  then have lte: ∀ i. (size ∘ c) i ≥ (size ∘ c) (Suc i)
    unfolding comp_def .
  then have ∃ l. ∀ l' ≥ l. size (c l') = size (c (Suc l'))
    using f_Suc_decr_eventually_const comp_def by auto
  then obtain l where
    l_p: ∀ l' ≥ l. size (c l') = size (c (Suc l'))
    by metis
  then have ∀ l' ≥ l. strictly_generalizes_cls (c (Suc l')) (c l')
    using ps unfolding strictly_generalizes_cls_def generalizes_cls_def
    by (metis size_subst less_irrefl strictly_subsumes_def mset_subset_size
        subset_mset_def subsumes_def strictly_subsumes_neq)
  then have ∀ i. strictly_generalizes_cls (c (Suc i + l)) (c (i + l))
    unfolding strictly_generalizes_cls_def generalizes_cls_def by auto
  then have ∃ f. ∀ i. strictly_generalizes_cls (f (Suc i)) (f i)
    by (rule exI[of _ λx. c (x + l)])
  then show False
    using wf_strictly_generalizes_cls
      wf_iff_no_infinite_down_chain[of {(x, y). strictly_generalizes_cls x y}]
    unfolding wfP_def by auto
qed

end
```

7.4 Most General Unifiers

```
locale mgu = substitution subst_atm id_subst comp_subst atm_of_atms renamings_apart
  for
    subst_atm :: 'a ⇒ 's ⇒ 'a and
    id_subst :: 's and
    comp_subst :: 's ⇒ 's ⇒ 's and
    atm_of_atms :: 'a list ⇒ 'a and
    renamings_apart :: 'a literal multiset list ⇒ 's list +
  fixes
    mgu :: 'a set set ⇒ 's option
  assumes
    mgu_sound: finite AAA ⟹ (∀ AA ∈ AAA. finite AA) ⟹ mgu AAA = Some σ ⟹ is_mgu σ AAA and
    mgu_complete:
      finite AAA ⟹ (∀ AA ∈ AAA. finite AA) ⟹ is_unifiers σ AAA ⟹ ∃ τ. mgu AAA = Some τ
begin

lemmas is_unifiers_mgu = mgu_sound[unfolded is_mgu_def, THEN conjunct1]
lemmas is_mgu_most_general = mgu_sound[unfolded is_mgu_def, THEN conjunct2]

lemma mgu_unifier:
  assumes
    aslen: length As = n and
    aaslen: length AAs = n and
```

mgu: *Some* σ = *mgu* (*set_mset* ‘ *set* (*map2 add_mset As AAs*)) **and**
i_lt: $i < n$ **and**
a_in: $A \in\# AAs \, ! \, i$
shows $A \cdot a \; \sigma = As \, ! \, i \cdot a \; \sigma$
proof −
from *mgu* **have** *is_mgu* σ (*set_mset* ‘ *set* (*map2 add_mset As AAs*))
using *mgu_sound* **by** *auto*
then have *is_unifiers* σ (*set_mset* ‘ *set* (*map2 add_mset As AAs*))
using *is_mgu_is_unifiers* **by** *auto*
then have *is_unifier* σ (*set_mset* (*add_mset* (*As* ! *i*) (*AAs* ! *i*)))
using *i_lt aslen aaslen* **unfolding** *is_unifiers_def is_unifier_def*
by *simp* (*metis length_zip min.idem nth_mem nth_zip prod.case set_mset_add_mset_insert*)
then show *?thesis*
using *aslen aaslen a_in is_unifier_subst_atm_eqI*
by (*metis finite_set_mset insertCI set_mset_add_mset_insert*)
qed

end

end

8 Refutational Inference Systems

theory *Inference_System*
imports *Herbrand_Interpretation*
begin

This theory gathers results from Section 2.4 (“Refutational Theorem Proving”), 3 (“Standard Resolution”), and 4.2 (“Counterexample-Reducing Inference Systems”) of Bachmair and Ganzinger’s chapter.

8.1 Preliminaries

Inferences have one distinguished main premise, any number of side premises, and a conclusion.

datatype *′a inference* =
Infer (*side_prems_of*: *′a clause multiset*) (*main_prem_of*: *′a clause*) (*concl_of*: *′a clause*)

abbreviation *prems_of* :: *′a inference* ⇒ *′a clause multiset* **where**
prems_of $\gamma \equiv$ *side_prems_of* γ + {#*main_prem_of* γ#}

abbreviation *concls_of* :: *′a inference set* ⇒ *′a clause set* **where**
concls_of $\Gamma \equiv$ *concl_of* ‘ Γ

definition *infer_from* :: *′a clause set* ⇒ *′a inference* ⇒ *bool* **where**
infer_from CC $\gamma \longleftrightarrow$ *set_mset* (*prems_of* γ) $\subseteq$ *CC*

locale *inference_system* =
fixes Γ :: *′a inference set*
begin

definition *inferences_from* :: *′a clause set* ⇒ *′a inference set* **where**
inferences_from CC = $\{\gamma.\ \gamma \in \Gamma \wedge$ *infer_from CC* $\gamma\}$

definition *inferences_between* :: *′a clause set* ⇒ *′a clause* ⇒ *′a inference set* **where**
inferences_between CC C = $\{\gamma.\ \gamma \in \Gamma \wedge$ *infer_from* $(CC \cup \{C\})\ \gamma \wedge C \in\#$ *prems_of* $\gamma\}$

lemma *inferences_from_mono*: $CC \subseteq DD \Longrightarrow$ *inferences_from CC* $\subseteq$ *inferences_from DD*
unfolding *inferences_from_def infer_from_def* **by** *fast*

definition *saturated* :: *′a clause set* ⇒ *bool* **where**
saturated N $\longleftrightarrow$ *concls_of* (*inferences_from N*) $\subseteq N$

```
lemma saturatedD:
  assumes
    satur: saturated N and
    inf: Infer CC D E ∈ Γ and
    cc_subs_n: set_mset CC ⊆ N and
    d_in_n: D ∈ N
  shows E ∈ N
proof −
  have Infer CC D E ∈ inferences_from N
    unfolding inferences_from_def infer_from_def using inf cc_subs_n d_in_n by simp
  then have E ∈ concls_of (inferences_from N)
    unfolding image_iff by (metis inference.sel(3))
  then show E ∈ N
    using satur unfolding saturated_def by blast
qed

end
```

Satisfiability preservation is a weaker requirement than soundness.

```
locale sat_preserving_inference_system = inference_system +
  assumes Γ_sat_preserving: satisfiable N ⟹ satisfiable (N ∪ concls_of (inferences_from N))

locale sound_inference_system = inference_system +
  assumes Γ_sound: Infer CC D E ∈ Γ ⟹ I ⊨m CC ⟹ I ⊨ D ⟹ I ⊨ E
begin

lemma Γ_sat_preserving:
  assumes sat_n: satisfiable N
  shows satisfiable (N ∪ concls_of (inferences_from N))
proof −
  obtain I where i: I ⊨s N
    using sat_n by blast
  then have ⋀CC D E. Infer CC D E ∈ Γ ⟹ set_mset CC ⊆ N ⟹ D ∈ N ⟹ I ⊨ E
    using Γ_sound unfolding true_clss_def true_cls_mset_def by (simp add: subset_eq)
  then have ⋀γ. γ ∈ Γ ⟹ infer_from N γ ⟹ I ⊨ concl_of γ
    unfolding infer_from_def by (case_tac γ) clarsimp
  then have I ⊨s concls_of (inferences_from N)
    unfolding inferences_from_def image_def true_clss_def infer_from_def by blast
  then have I ⊨s N ∪ concls_of (inferences_from N)
    using i by simp
  then show ?thesis
    by blast
qed

sublocale sat_preserving_inference_system
  by unfold_locales (erule Γ_sat_preserving)

end

locale reductive_inference_system = inference_system Γ for Γ :: ('a :: wellorder) inference set +
  assumes Γ_reductive: γ ∈ Γ ⟹ concl_of γ < main_prem_of γ
```

8.2 Refutational Completeness

Refutational completeness can be established once and for all for counterexample-reducing inference systems. The material formalized here draws from both the general framework of Section 4.2 and the concrete instances of Section 3.

```
locale counterex_reducing_inference_system =
  inference_system Γ for Γ :: ('a :: wellorder) inference set +
  fixes I_of :: 'a clause set ⇒ 'a interp
  assumes Γ_counterex_reducing:
    {#} ∉ N ⟹ D ∈ N ⟹ ¬ I_of N ⊨ D ⟹ (⋀C. C ∈ N ⟹ ¬ I_of N ⊨ C ⟹ D ≤ C) ⟹
      ∃ CC E. set_mset CC ⊆ N ∧ I_of N ⊨m CC ∧ Infer CC D E ∈ Γ ∧ ¬ I_of N ⊨ E ∧ E < D
```

begin

lemma *ex_min_counterex*:
  fixes N :: ($'a$ :: *wellorder*) *clause set*
  assumes $\neg\ I \models s\ N$
  shows $\exists\, C \in N.\ \neg\ I \models C \wedge (\forall\, D \in N.\ D < C \longrightarrow I \models D)$
proof −
  obtain C **where** $C \in N$ **and** $\neg\ I \models C$
    using *assms* **unfolding** *true_clss_def* **by** *auto*
  then have *c_in*: $C \in \{C \in N.\ \neg\ I \models C\}$
    by *blast*
  show *?thesis*
    using *wf_eq_minimal*[*THEN iffD1*, *rule_format*, *OF wf_less_multiset c_in*] **by** *blast*
qed

theorem *saturated_model*:
  assumes
    satur: *saturated* N **and**
    ec_ni_n: $\{\#\} \notin N$
  shows *I_of* $N \models s\ N$
proof −
  have *ec_ni_n*: $\{\#\} \notin N$
    using *ec_ni_n* **by** *auto*

  {
    assume $\neg$ *I_of* $N \models s\ N$
    then obtain D **where**
      d_in_n: $D \in N$ **and**
      d_cex: $\neg$ *I_of* $N \models D$ **and**
      d_min: $\bigwedge C.\ C \in N \Longrightarrow C < D \Longrightarrow$ *I_of* $N \models C$
      by (*meson ex_min_counterex*)
    then obtain $CC\ E$ **where**
      cc_subs_n: *set_mset* $CC \subseteq N$ **and**
      inf_e: *Infer* $CC\ D\ E \in \Gamma$ **and**
      e_cex: $\neg$ *I_of* $N \models E$ **and**
      e_lt_d: $E < D$
      using Γ*_counterex_reducing*[*OF ec_ni_n*] *not_less* **by** *metis*
    from *cc_subs_n inf_e* **have** $E \in N$
      using *d_in_n satur* **by** (*blast dest*: *saturatedD*)
    then have *False*
      using *e_cex e_lt_d d_min not_less* **by** *blast*
  }
  then show *?thesis*
    by *satx*
qed

Cf. Corollary 3.10:

corollary *saturated_complete*: *saturated* $N \Longrightarrow \neg$ *satisfiable* $N \Longrightarrow \{\#\} \in N$
  using *saturated_model* **by** *blast*

end

8.3 Compactness

Bachmair and Ganzinger claim that compactness follows from refutational completeness but leave the proof to the readers' imagination. Our proof relies on an inductive definition of saturation in terms of a base set of clauses.

context *inference_system*
begin

inductive-set *saturate* :: $'a$ *clause set* $\Rightarrow$ $'a$ *clause set* **for** CC :: $'a$ *clause set* **where**

base: $C \in CC \Longrightarrow C \in \mathit{saturate}\ CC$
| *step*: $\mathit{Infer}\ CC'\ D\ E \in \Gamma \Longrightarrow (\bigwedge C'.\ C' \in\#\ CC' \Longrightarrow C' \in \mathit{saturate}\ CC) \Longrightarrow D \in \mathit{saturate}\ CC \Longrightarrow$
$E \in \mathit{saturate}\ CC$

lemma *saturate_mono*: $C \in \mathit{saturate}\ CC \Longrightarrow CC \subseteq DD \Longrightarrow C \in \mathit{saturate}\ DD$
by (*induct rule*: *saturate.induct*) (*auto intro*: *saturate.intros*)

lemma *saturated_saturate*[*simp*, *intro*]: *saturated* (*saturate* N)
unfolding *saturated_def inferences_from_def infer_from_def image_def*
by *clarify* (*rename_tac* x, *case_tac* x, *auto elim*!: *saturate.step*)

lemma *saturate_finite*: $C \in \mathit{saturate}\ CC \Longrightarrow \exists\, DD.\ DD \subseteq CC \land \mathit{finite}\ DD \land C \in \mathit{saturate}\ DD$
proof (*induct rule*: *saturate.induct*)
case (*base* C)
then have $\{C\} \subseteq CC$ **and** *finite* $\{C\}$ **and** $C \in \mathit{saturate}\ \{C\}$
by (*auto intro*: *saturate.intros*)
then show *?case*
by *blast*
next
case (*step* CC' D E)
obtain *DD_of* **where**
$\bigwedge C.\ C \in\#\ CC' \Longrightarrow \mathit{DD\_of}\ C \subseteq CC \land \mathit{finite}\ (\mathit{DD\_of}\ C) \land C \in \mathit{saturate}\ (\mathit{DD\_of}\ C)$
using *step*(*3*) **by** *metis*
then have
$(\bigcup C \in \mathit{set\_mset}\ CC'.\ \mathit{DD\_of}\ C) \subseteq CC$
$\mathit{finite}\ (\bigcup C \in \mathit{set\_mset}\ CC'.\ \mathit{DD\_of}\ C) \land \mathit{set\_mset}\ CC' \subseteq \mathit{saturate}\ (\bigcup C \in \mathit{set\_mset}\ CC'.\ \mathit{DD\_of}\ C)$
by (*auto intro*: *saturate_mono*)
then obtain DD **where**
d_sub: $DD \subseteq CC$ **and** *d_fin*: *finite* DD **and** *in_sat_d*: $\mathit{set\_mset}\ CC' \subseteq \mathit{saturate}\ DD$
by *blast*
obtain EE **where**
e_sub: $EE \subseteq CC$ **and** *e_fin*: *finite* EE **and** *in_sat_ee*: $D \in \mathit{saturate}\ EE$
using *step*(*5*) **by** *blast*
have $DD \cup EE \subseteq CC$
using *d_sub e_sub step*(*1*) **by** *fast*
moreover have *finite* $(DD \cup EE)$
using *d_fin e_fin* **by** *fast*
moreover have $E \in \mathit{saturate}\ (DD \cup EE)$
using *in_sat_d in_sat_ee step.hyps*(*1*)
by (*blast intro*: *inference_system.saturate.step saturate_mono*)
ultimately show *?case*
by *blast*
qed

end

context *sound_inference_system*
begin

theorem *saturate_sound*: $C \in \mathit{saturate}\ CC \Longrightarrow I \models s\ CC \Longrightarrow I \models C$
by (*induct rule*: *saturate.induct*) (*auto simp*: *true_cls_mset_def true_clss_def* Γ*_sound*)

end

context *sat_preserving_inference_system*
begin

This result surely holds, but we have yet to prove it. The challenge is: Every time a new clause is introduced, we also get a new interpretation (by the definition of *sat_preserving_inference_system*). But the interpretation we want here is then the one that exists "at the limit". Maybe we can use compactness to prove it.

theorem *saturate_sat_preserving*: *satisfiable* $CC \Longrightarrow$ *satisfiable* (*saturate* CC)
oops

end

locale *sound_counterex_reducing_inference_system* =
counterex_reducing_inference_system + *sound_inference_system*
begin

Compactness of clausal logic is stated as Theorem 3.12 for the case of unordered ground resolution. The proof below is a generalization to any sound counterexample-reducing inference system. The actual theorem will become available once the locale has been instantiated with a concrete inference system.

theorem *clausal_logic_compact*:
fixes N :: ($'a$ :: *wellorder*) *clause set*
shows $\neg$ *satisfiable* $N \longleftrightarrow (\exists\, DD \subseteq N.$ *finite* $DD \wedge \neg$ *satisfiable* $DD)$
proof
assume $\neg$ *satisfiable* N
then have $\{\#\} \in$ *saturate* N
using *saturated_complete saturated_saturate saturate.base* **unfolding** *true_clss_def* **by** *meson*
then have $\exists\, DD \subseteq N.$ *finite* $DD \wedge \{\#\} \in$ *saturate* DD
using *saturate_finite* **by** *fastforce*
then show $\exists\, DD \subseteq N.$ *finite* $DD \wedge \neg$ *satisfiable* DD
using *saturate_sound* **by** *auto*
next
assume $\exists\, DD \subseteq N.$ *finite* $DD \wedge \neg$ *satisfiable* DD
then show $\neg$ *satisfiable* N
by (*blast intro*: *true_clss_mono*)
qed

end

end

9 Candidate Models for Ground Resolution

theory *Ground_Resolution_Model*
imports *Herbrand_Interpretation*
begin

The proofs of refutational completeness for the two resolution inference systems presented in Section 3 ("Standard Resolution") of Bachmair and Ganzinger's chapter share mostly the same candidate model construction. The literal selection capability needed for the second system is ignored by the first one, by taking $\lambda\_.\ \{\}$ as instantiation for the S parameter.

locale *selection* =
fixes S :: $'a$ *clause* $\Rightarrow$ $'a$ *clause*
assumes
S_selects_subseteq: $S\ C \subseteq\#\ C$ **and**
S_selects_neg_lits: $L \in\#\ S\ C \Longrightarrow$ *is_neg* L

locale *ground_resolution_with_selection* = *selection* S
for S :: ($'a$ :: *wellorder*) *clause* $\Rightarrow$ $'a$ *clause*
begin

The following commands corresponds to Definition 3.14, which generalizes Definition 3.1. *production* C is denoted ε_C in the chapter; *interp* C is denoted I_C; *Interp* C is denoted I^C; and *Interp_N* is denoted I_N. The mutually recursive definition from the chapter is massaged to simplify the termination argument. The *production_unfold* lemma below gives the intended characterization.

context
fixes N :: $'a$ *clause set*
begin

function *production* :: $'a$ *clause* $\Rightarrow$ $'a$ *interp* **where**
production C =
$\{A.\ C \in N \wedge C \neq \{\#\} \wedge$ *Max_mset* $C =$ *Pos* $A \wedge \neg (\bigcup D \in \{D.\ D < C\}.$ *production* $D) \models C \wedge S\ C = \{\#\}\}$

by *auto*
termination by (*rule termination*[*OF wf*, *simplified*])

declare *production.simps* [*simp del*]

definition *interp* :: *'a clause* ⇒ *'a interp* **where**
interp C = ($\bigcup$*D* ∈ {*D*. *D* < *C*}. *production D*)

lemma *production_unfold*:
production C = {*A*. *C* ∈ *N* ∧ *C* ≠ {#} ∧ *Max_mset C* = *Pos A* ∧ ¬ *interp C* ⊨ *C* ∧ *S C* = {#}}
unfolding *interp_def* **by** (*rule production.simps*)

abbreviation *productive* :: *'a clause* ⇒ *bool* **where**
productive C ≡ *production C* ≠ {}

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

lemma *producesD*: *produces C A* ⟹ *C* ∈ *N* ∧ *C* ≠ {#} ∧ *Pos A* = *Max_mset C* ∧ ¬ *interp C* ⊨ *C* ∧ *S C* = {#}
unfolding *production_unfold* **by** *auto*

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

lemma *interp_subseteq_Interp*[*simp*]: *interp C* ⊆ *Interp C*
by (*simp add*: *Interp_def*)

lemma *Interp_as_UNION*: *Interp C* = ($\bigcup$*D* ∈ {*D*. *D* ≤ *C*}. *production D*)
unfolding *Interp_def interp_def less_eq_multiset_def* **by** *fast*

lemma *productive_not_empty*: *productive C* ⟹ *C* ≠ {#}
unfolding *production_unfold* **by** *simp*

lemma *productive_imp_produces_Max_literal*: *productive C* ⟹ *produces C* (*atm_of* (*Max_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_mset C*)
using *productive_imp_produces_Max_literal* **by** *auto*

lemma *produces_imp_Max_atom*: *produces C A* ⟹ *A* = *Max* (*atms_of C*)
using *producesD produces_imp_Max_literal* **by** *auto*

lemma *produces_imp_Pos_in_lits*: *produces C A* ⟹ *Pos A* ∈# *C*
by (*simp add*: *producesD*)

lemma *productive_in_N*: *productive C* ⟹ *C* ∈ *N*
unfolding *production_unfold* **by** *simp*

lemma *produces_imp_atms_leq*: *produces C A* ⟹ *B* ∈ *atms_of C* ⟹ *B* ≤ *A*
using *Max.coboundedI produces_imp_Max_atom* **by** *blast*

lemma *produces_imp_neg_notin_lits*: *produces C A* ⟹ ¬ *Neg A* ∈# *C*
by (*simp add*: *pos_Max_imp_neg_notin producesD*)

lemma *less_eq_imp_interp_subseteq_interp*: *C* ≤ *D* ⟹ *interp C* ⊆ *interp D*
unfolding *interp_def* **by** *auto* (*metis 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*: $C < D \Longrightarrow$ *production* $C \subseteq$ *interp* D
unfolding *interp_def* **by** *fast*

lemma *less_eq_imp_production_subseteq_Interp*: $C \leq D \Longrightarrow$ *production* $C \subseteq$ *Interp* D
unfolding *Interp_def* **using** *less_imp_production_subseteq_interp*
by (*metis le_imp_less_or_eq le_supI1 sup_ge2*)

lemma *less_imp_Interp_subseteq_interp*: $C < D \Longrightarrow$ *Interp* $C \subseteq$ *interp* D
by (*simp add*: *Interp_def less_eq_imp_interp_subseteq_interp less_imp_production_subseteq_interp*)

lemma *less_eq_imp_Interp_subseteq_Interp*: $C \leq D \Longrightarrow$ *Interp* $C \subseteq$ *Interp* D
using *Interp_def less_eq_imp_interp_subseteq_Interp less_eq_imp_production_subseteq_Interp* **by** *auto*

lemma *not_Interp_to_interp_imp_less*: $A \notin$ *Interp* $C \Longrightarrow A \in$ *interp* $D \Longrightarrow C < D$
using *less_eq_imp_interp_subseteq_Interp not_less* **by** *blast*

lemma *not_interp_to_interp_imp_less*: $A \notin$ *interp* $C \Longrightarrow A \in$ *interp* $D \Longrightarrow C < D$
using *less_eq_imp_interp_subseteq_interp not_less* **by** *blast*

lemma *not_Interp_to_Interp_imp_less*: $A \notin$ *Interp* $C \Longrightarrow A \in$ *Interp* $D \Longrightarrow C < D$
using *less_eq_imp_Interp_subseteq_Interp not_less* **by** *blast*

lemma *not_interp_to_Interp_imp_le*: $A \notin$ *interp* $C \Longrightarrow A \in$ *Interp* $D \Longrightarrow C \leq D$
using *less_imp_Interp_subseteq_interp not_less* **by** *blast*

definition *INTERP* :: *'a interp* **where**
INTERP $= (\bigcup C \in N.$ *production* $C)$

lemma *interp_subseteq_INTERP*: *interp* $C \subseteq$ *INTERP*
unfolding *interp_def INTERP_def* **by** (*auto simp*: *production_unfold*)

lemma *production_subseteq_INTERP*: *production* $C \subseteq$ *INTERP*
unfolding *INTERP_def* **using** *production_unfold* **by** *blast*

lemma *Interp_subseteq_INTERP*: *Interp* $C \subseteq$ *INTERP*
by (*simp add*: *Interp_def interp_subseteq_INTERP production_subseteq_INTERP*)

lemma *produces_imp_in_interp*:
assumes *a_in_c*: *Neg* $A \in\#$ C **and** *d*: *produces* D A
shows $A \in$ *interp* C
by (*metis Interp_def Max_pos_neg_less_multiset UnCI a_in_c d*
not_interp_to_Interp_imp_le not_less producesD singletonI)

lemma *neg_notin_Interp_not_produce*: *Neg* $A \in\#$ $C \Longrightarrow A \notin$ *Interp* $D \Longrightarrow C \leq D \Longrightarrow \neg$ *produces* D'' A
using *less_eq_imp_interp_subseteq_Interp produces_imp_in_interp* **by** *blast*

lemma *in_production_imp_produces*: $A \in$ *production* $C \Longrightarrow$ *produces* C A
using *productive_imp_produces_Max_atom* **by** *fastforce*

lemma *not_produces_imp_notin_production*: $\neg$ *produces* C $A \Longrightarrow A \notin$ *production* C
using *in_production_imp_produces* **by** *blast*

lemma *not_produces_imp_notin_interp*: $(\bigwedge D. \neg$ *produces* D $A) \Longrightarrow A \notin$ *interp* C
unfolding *interp_def* **by** (*fast intro*!: *in_production_imp_produces*)

The results below corresponds to Lemma 3.4.

lemma *Interp_imp_general*:
assumes
c_le_d: $C \leq D$ **and**
d_lt_d': $D < D'$ **and**
c_at_d: *Interp* $D \models C$ **and**
subs: *interp* $D' \subseteq (\bigcup C \in CC.$ *production* $C)$

shows $(\bigcup C \in CC.\ \mathit{production}\ C) \models C$
proof (*cases* $\exists A.\ \mathit{Pos}\ A \in\#\ C \wedge A \in \mathit{Interp}\ D$)
case *True*
then obtain A **where** *a_in_c*: $\mathit{Pos}\ A \in\#\ C$ **and** *a_at_d*: $A \in \mathit{Interp}\ D$
by *blast*
from *a_at_d* **have** $A \in \mathit{interp}\ D'$
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*: $\mathit{Neg}\ A \in\#\ C$ **and** $A \notin \mathit{Interp}\ D$
using *c_at_d* **unfolding** *true_cls_def* **by** *blast*
then have $\bigwedge D''.\ \neg\ \mathit{produces}\ D''\ A$
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 *Interp_imp_interp*: $C \leq D \Longrightarrow D < D' \Longrightarrow \mathit{Interp}\ D \models C \Longrightarrow \mathit{interp}\ D' \models C$
using *interp_def Interp_imp_general* **by** *simp*

lemma *Interp_imp_Interp*: $C \leq D \Longrightarrow D \leq D' \Longrightarrow \mathit{Interp}\ D \models C \Longrightarrow \mathit{Interp}\ D' \models C$
using *Interp_as_UNION interp_subseteq_Interp Interp_imp_general* **by** (*metis antisym_conv2*)

lemma *Interp_imp_INTERP*: $C \leq D \Longrightarrow \mathit{Interp}\ D \models C \Longrightarrow \mathit{INTERP} \models C$
using *INTERP_def interp_subseteq_INTERP Interp_imp_general*[*OF _ le_multiset_right_total*] **by** *simp*

lemma *interp_imp_general*:
assumes
c_le_d: $C \leq D$ **and**
d_le_d′: $D \leq D'$ **and**
c_at_d: $\mathit{interp}\ D \models C$ **and**
subs: $\mathit{interp}\ D' \subseteq (\bigcup C \in CC.\ \mathit{production}\ C)$
shows $(\bigcup C \in CC.\ \mathit{production}\ C) \models C$
proof (*cases* $\exists A.\ \mathit{Pos}\ A \in\#\ C \wedge A \in \mathit{interp}\ D$)
case *True*
then obtain A **where** *a_in_c*: $\mathit{Pos}\ A \in\#\ C$ **and** *a_at_d*: $A \in \mathit{interp}\ D$
by *blast*
from *a_at_d* **have** $A \in \mathit{interp}\ D'$
using *d_le_d′ less_eq_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*: $\mathit{Neg}\ A \in\#\ C$ **and** $A \notin \mathit{interp}\ D$
using *c_at_d* **unfolding** *true_cls_def* **by** *blast*
then have $\bigwedge D''.\ \neg\ \mathit{produces}\ D''\ A$
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

lemma *interp_imp_interp*: $C \leq D \Longrightarrow D \leq D' \Longrightarrow \mathit{interp}\ D \models C \Longrightarrow \mathit{interp}\ D' \models C$
using *interp_def interp_imp_general* **by** *simp*

lemma *interp_imp_Interp*: $C \leq D \Longrightarrow D \leq D' \Longrightarrow \mathit{interp}\ D \models C \Longrightarrow \mathit{Interp}\ D' \models C$
using *Interp_as_UNION interp_subseteq_Interp*[*of D′*] *interp_imp_general* **by** *simp*

lemma *interp_imp_INTERP*: $C \leq D \Longrightarrow \mathit{interp}\ D \models C \Longrightarrow \mathit{INTERP} \models C$
using *INTERP_def interp_subseteq_INTERP interp_imp_general linear* **by** *metis*

lemma *productive_imp_not_interp*: $\mathit{productive}\ C \Longrightarrow \neg\ \mathit{interp}\ C \models C$

unfolding *production_unfold* **by** *simp*

This corresponds to Lemma 3.3:

lemma *productive_imp_Interp*:
assumes *productive C*
shows *Interp C* $\models$ *C*
proof –
obtain *A* **where** *a*: *produces C A*
using *assms productive_imp_produces_Max_atom* **by** *blast*
then have *a_in_c*: *Pos A* $\in\#$ *C*
by (*rule produces_imp_Pos_in_lits*)
moreover have *A* $\in$ *Interp C*
using *a less_eq_imp_production_subseteq_Interp* **by** *blast*
ultimately show *?thesis*
by *fast*
qed

lemma *productive_imp_INTERP*: *productive C* $\Longrightarrow$ *INTERP* $\models$ *C*
by (*fast intro*: *productive_imp_Interp Interp_imp_INTERP*)

This corresponds to Lemma 3.5:

lemma *max_pos_imp_Interp*:
assumes *C* $\in$ *N* **and** *C* $\neq$ {#} **and** *Max_mset C* = *Pos A* **and** *S C* = {#}
shows *Interp C* $\models$ *C*
proof (*cases productive C*)
case *True*
then show *?thesis*
by (*fast intro*: *productive_imp_Interp*)
next
case *False*
then have *interp C* $\models$ *C*
using *assms* **unfolding** *production_unfold* **by** *simp*
then show *?thesis*
unfolding *Interp_def* **using** *False* **by** *auto*
qed

The following results correspond to Lemma 3.6:

lemma *max_atm_imp_Interp*:
assumes
c_in_n: *C* $\in$ *N* **and**
pos_in: *Pos A* $\in\#$ *C* **and**
max_atm: *A* = *Max* (*atms_of C*) **and**
s_c_e: *S C* = {#}
shows *Interp C* $\models$ *C*
proof (*cases Neg A* $\in\#$ *C*)
case *True*
then show *?thesis*
using *pos_in pos_neg_in_imp_true* **by** *metis*
next
case *False*
moreover have *ne*: *C* $\neq$ {#}
using *pos_in* **by** *auto*
ultimately have *Max_mset C* = *Pos A*
using *max_atm* **using** *Max_in_lits Max_lit_eq_pos_or_neg_Max_atm* **by** *metis*
then show *?thesis*
using *ne c_in_n s_c_e* **by** (*blast intro*: *max_pos_imp_Interp*)
qed

lemma *not_Interp_imp_general*:
assumes
d'_le_d: *D'* $\leq$ *D* **and**
in_n_or_max_gt: *D'* $\in$ *N* $\wedge$ *S D'* = {#} $\vee$ *Max* (*atms_of D'*) < *Max* (*atms_of D*) **and**
d'_at_d: $\neg$ *Interp D* $\models$ *D'* **and**

d_lt_c: $D < C$ **and**
subs: *interp* $C \subseteq (\bigcup C \in CC.$ *production* $C)$
shows $\neg (\bigcup C \in CC.$ *production* $C) \models D'$
proof –
{
assume *cc_blw_d'*: $(\bigcup C \in CC.$ *production* $C) \models D'$
have *Interp* $D \subseteq (\bigcup C \in CC.$ *production* $C)$
using *less_imp_Interp_subseteq_interp d_lt_c subs* **by** *blast*
then obtain A **where** *a_in_d'*: *Pos* $A \in\#$ D' **and** *a_blw_cc*: $A \in (\bigcup C \in CC.$ *production* $C)$
using *cc_blw_d' d'_at_d false_to_true_imp_ex_pos* **by** *metis*
from *a_in_d'* **have** *a_at_d*: $A \notin$ *Interp* D
using *d'_at_d* **by** *fast*
from *a_blw_cc* **obtain** C' **where** *prod_c'*: *production* $C' = \{A\}$
by (*fast intro*!: *in_production_imp_produces*)
have *max_c'*: *Max* (*atms_of* C') $= A$
using *prod_c' productive_imp_produces_Max_atom* **by** *force*
have *leq_dc'*: $D \leq C'$
using *a_at_d d'_at_d prod_c'* **by** (*auto simp*: *Interp_def intro*: *not_interp_to_Interp_imp_le*)
then have $D' \leq C'$
using *d'_le_d order_trans* **by** *blast*
then have *max_d'*: *Max* (*atms_of* D') $= A$
using *a_in_d' max_c'* **by** (*fast intro*: *pos_lit_in_atms_of le_multiset_Max_in_imp_Max*)

{
assume $D' \in N \wedge S\ D' = \{\#\}$
then have *Interp* $D' \models D'$
using *a_in_d' max_d'* **by** (*blast intro*: *max_atm_imp_Interp*)
then have *Interp* $D \models D'$
using *d'_le_d* **by** (*auto intro*: *Interp_imp_Interp simp*: *less_eq_multiset_def*)
then have *False*
using *d'_at_d* **by** *satx*
}
moreover
{
assume *Max* (*atms_of* D') $<$ *Max* (*atms_of* D)
then have *False*
using *max_d' leq_dc' max_c' d'_le_d*
by (*metis le_imp_less_or_eq le_multiset_empty_right less_eq_Max_atms_of less_imp_not_less*)
}
ultimately have *False*
using *in_n_or_max_gt* **by** *satx*
}
then show *?thesis*
by *satx*
qed

lemma *not_Interp_imp_not_interp*:
$D' \leq D \Longrightarrow D' \in N \wedge S\ D' = \{\#\} \vee$ *Max* (*atms_of* D') $<$ *Max* (*atms_of* D) $\Longrightarrow \neg$ *Interp* $D \models D' \Longrightarrow$
$D < C \Longrightarrow \neg$ *interp* $C \models D'$
using *interp_def not_Interp_imp_general* **by** *simp*

lemma *not_Interp_imp_not_Interp*:
$D' \leq D \Longrightarrow D' \in N \wedge S\ D' = \{\#\} \vee$ *Max* (*atms_of* D') $<$ *Max* (*atms_of* D) $\Longrightarrow \neg$ *Interp* $D \models D' \Longrightarrow$
$D < C \Longrightarrow \neg$ *Interp* $C \models D'$
using *Interp_as_UNION interp_subseteq_Interp not_Interp_imp_general* **by** *metis*

lemma *not_Interp_imp_not_INTERP*:
$D' \leq D \Longrightarrow D' \in N \wedge S\ D' = \{\#\} \vee$ *Max* (*atms_of* D') $<$ *Max* (*atms_of* D) $\Longrightarrow \neg$ *Interp* $D \models D' \Longrightarrow$
$\neg$ *INTERP* $\models D'$
using *INTERP_def interp_subseteq_INTERP not_Interp_imp_general*[*OF* _ _ _ *le_multiset_right_total*]
by *simp*

Lemma 3.7 is a problem child. It is stated below but not proved; instead, a counterexample is displayed. This is not much of a problem, because it is not invoked in the rest of the chapter.

lemma
  assumes $D \in N$ **and** $\bigwedge D'. D' < D \Longrightarrow \mathit{Interp}\ D' \models C$
  shows $\mathit{interp}\ D \models C$
  oops

lemma
  assumes d: $D = \{\#\}$ **and** n: $N = \{D, C\}$ **and** c: $C = \{\#\mathit{Pos}\ A\#\}$
  shows $D \in N$ **and** $\bigwedge D'. D' < D \Longrightarrow \mathit{Interp}\ D' \models C$ **and** $\neg\ \mathit{interp}\ D \models C$
  using n **unfolding** $d\ c\ \mathit{interp\_def}$ **by** *auto*

end

end

end

10 Ground Unordered Resolution Calculus

theory *Unordered_Ground_Resolution*
  imports *Inference_System Ground_Resolution_Model*
begin

Unordered ground resolution is one of the two inference systems studied in Section 3 ("Standard Resolution") of Bachmair and Ganzinger's chapter.

10.1 Inference Rule

Unordered ground resolution consists of a single rule, called *unord_resolve* below, which is sound and counterexample-reducing.

locale *ground_resolution_without_selection*
begin

sublocale *ground_resolution_with_selection* **where** $S = \lambda\_. \{\#\}$
  by *unfold_locales auto*

inductive *unord_resolve* :: $'a\ \mathit{clause} \Rightarrow 'a\ \mathit{clause} \Rightarrow 'a\ \mathit{clause} \Rightarrow \mathit{bool}$ **where**
  $\mathit{unord\_resolve}\ (C + \mathit{replicate\_mset}\ (\mathit{Suc}\ n)\ (\mathit{Pos}\ A))\ (\mathit{add\_mset}\ (\mathit{Neg}\ A)\ D)\ (C + D)$

lemma *unord_resolve_sound*: $\mathit{unord\_resolve}\ C\ D\ E \Longrightarrow I \models C \Longrightarrow I \models D \Longrightarrow I \models E$
  using *unord_resolve.cases* **by** *fastforce*

The following result corresponds to Theorem 3.8, except that the conclusion is strengthened slightly to make it fit better with the counterexample-reducing inference system framework.

theorem *unord_resolve_counterex_reducing*:
  assumes
    ec_ni_n: $\{\#\} \notin N$ **and**
    c_in_n: $C \in N$ **and**
    c_cex: $\neg\ \mathit{INTERP}\ N \models C$ **and**
    c_min: $\bigwedge D.\ D \in N \Longrightarrow \neg\ \mathit{INTERP}\ N \models D \Longrightarrow C \le D$
  obtains $D\ E$ **where**
    $D \in N$
    $\mathit{INTERP}\ N \models D$
    $\mathit{productive}\ N\ D$
    $\mathit{unord\_resolve}\ D\ C\ E$
    $\neg\ \mathit{INTERP}\ N \models E$
    $E < C$
proof −
  have *c_ne*: $C \ne \{\#\}$
    using *c_in_n ec_ni_n* **by** *blast*
  have $\exists A.\ A \in \mathit{atms\_of}\ C \wedge A = \mathit{Max}\ (\mathit{atms\_of}\ C)$
    using *c_ne* **by** (*blast intro*: *Max_in_lits atm_of_Max_lit atm_of_lit_in_atms_of*)

```
then have ∃ A. Neg A ∈# C
  using c_ne c_in_n c_cex c_min Max_in_lits Max_lit_eq_pos_or_neg_Max_atm  max_pos_imp_Interp
    Interp_imp_INTERP by metis
then obtain A where neg_a_in_c: Neg A ∈# C
  by blast
then obtain C' where c: C = add_mset (Neg A) C'
  using insert_DiffM by metis
have A ∈ INTERP N
  using neg_a_in_c c_cex[unfolded true_cls_def] by fast
then obtain D where d0: produces N D A
  unfolding INTERP_def by (metis UN_E not_produces_imp_notin_production)
have prod_d: productive N D
  unfolding d0 by simp
then have d_in_n: D ∈ N
  using productive_in_N by fast
have d_true: INTERP N ⊨ D
  using prod_d productive_imp_INTERP by blast

obtain D' AAA where
  d: D = D' + AAA and
  d': D' = {#L ∈# D. L ≠ Pos A#} and
  aa: AAA = {#L ∈# D. L = Pos A#}
  using multiset_partition union_commute by metis
have d'_subs: set_mset D' ⊆ set_mset D
  unfolding d' by auto
have ¬ Neg A ∈# D
  using d0 by (blast dest: produces_imp_neg_notin_lits)
then have neg_a_ni_d': ¬ Neg A ∈# D'
  using d'_subs by auto
have a_ni_d': A ∉ atms_of D'
  using d' neg_a_ni_d' by (auto dest: atm_imp_pos_or_neg_lit)
have ∃ n. AAA = replicate_mset (Suc n) (Pos A)
  using aa d0 not0_implies_Suc produces_imp_Pos_in_lits[of N]
  by (simp add: filter_eq_replicate_mset del: replicate_mset_Suc)
then have res_e: unord_resolve D C (D' + C')
  unfolding c d by (fastforce intro: unord_resolve.intros)

have d'_le_d: D' ≤ D
  unfolding d by simp
have a_max_d: A = Max (atms_of D)
  using d0 productive_imp_produces_Max_atom by auto
then have D' ≠ {#} ⟹ Max (atms_of D') ≤ A
  using d'_le_d by (blast intro: less_eq_Max_atms_of)
moreover have D' ≠ {#} ⟹ Max (atms_of D') ≠ A
  using a_ni_d' Max_in by (blast intro: atms_empty_iff_empty[THEN iffD1])
ultimately have max_d'_lt_a: D' ≠ {#} ⟹ Max (atms_of D') < A
  using dual_order.strict_iff_order by blast

have ¬ interp N D ⊨ D
  using d0 productive_imp_not_interp by blast
then have ¬ Interp N D ⊨ D'
  unfolding d0 d' Interp_def true_cls_def by (auto simp: true_lit_def simp del: not_gr_zero)
then have ¬ INTERP N ⊨ D'
  using a_max_d d'_le_d max_d'_lt_a not_Interp_imp_not_INTERP by blast
moreover have ¬ INTERP N ⊨ C'
  using c_cex unfolding c by simp
ultimately have e_cex: ¬ INTERP N ⊨ D' + C'
  by simp

have ⋀B. B ∈ atms_of D' ⟹ B ≤ A
  using d0 d'_subs contra_subsetD lits_subseteq_imp_atms_subseteq produces_imp_atms_leq by metis
then have ⋀L. L ∈# D' ⟹ L < Neg A
  using neg_a_ni_d' antisym_conv1 atms_less_eq_imp_lit_less_eq_neg by metis
```

```
  then have lt_cex: D' + C' < C
    by (force intro: add.commute simp: c less_multiset_DM intro: exI[of _ {#Neg A#}])

  from d_in_n d_true prod_d res_e e_cex lt_cex show ?thesis ..
qed
```

10.2 Inference System

Lemma 3.9 and Corollary 3.10 are subsumed in the counterexample-reducing inference system framework, which is instantiated below.

```
definition unord_Γ :: 'a inference set where
  unord_Γ = {Infer {#C#} D E | C D E. unord_resolve C D E}

sublocale unord_Γ_sound_counterex_reducing?:
  sound_counterex_reducing_inference_system unord_Γ INTERP
proof unfold_locales
  fix D E and N :: ('b :: wellorder) clause set
  assume {#} ∉ N and D ∈ N and ¬ INTERP N ⊨ D and ⋀C. C ∈ N ⟹ ¬ INTERP N ⊨ C ⟹ D ≤ C
  then obtain C E where
    c_in_n: C ∈ N and
    c_true: INTERP N ⊨ C and
    res_e: unord_resolve C D E and
    e_cex: ¬ INTERP N ⊨ E and
    e_lt_d: E < D
    using unord_resolve_counterex_reducing by (metis (no_types))
  from c_in_n have set_mset {#C#} ⊆ N
    by auto
  moreover have Infer {#C#} D E ∈ unord_Γ
    unfolding unord_Γ_def using res_e by blast
  ultimately show
    ∃CC E. set_mset CC ⊆ N ∧ INTERP N ⊨m CC ∧ Infer CC D E ∈ unord_Γ ∧ ¬ INTERP N ⊨ E ∧ E < D
    using c_in_n c_true e_cex e_lt_d by blast
next
  fix CC D E and I :: 'b interp
  assume Infer CC D E ∈ unord_Γ and I ⊨m CC and I ⊨ D
  then show I ⊨ E
    by (clarsimp simp: unord_Γ_def true_cls_mset_def) (erule unord_resolve_sound, auto)
qed

lemmas clausal_logic_compact = unord_Γ_sound_counterex_reducing.clausal_logic_compact

end
```

Theorem 3.12, compactness of clausal logic, has finally been derived for a concrete inference system:

```
lemmas clausal_logic_compact = ground_resolution_without_selection.clausal_logic_compact

end
```

11 Ground Ordered Resolution Calculus with Selection

```
theory Ordered_Ground_Resolution
  imports Inference_System Ground_Resolution_Model
begin
```

Ordered ground resolution with selection is the second inference system studied in Section 3 (“Standard Resolution”) of Bachmair and Ganzinger’s chapter.

11.1 Inference Rule

Ordered ground resolution consists of a single rule, called *ord_resolve* below. Like *unord_resolve*, the rule is sound and counterexample-reducing. In addition, it is reductive.

context *ground_resolution_with_selection*
begin

The following inductive definition corresponds to Figure 2.

definition *maximal_wrt* :: $'a \Rightarrow 'a$ *literal multiset* $\Rightarrow$ *bool* **where**
  maximal_wrt $A\ DA \equiv A = Max\ (atms\_of\ DA)$

definition *strictly_maximal_wrt* :: $'a \Rightarrow 'a$ *literal multiset* $\Rightarrow$ *bool* **where**
  strictly_maximal_wrt $A\ CA \longleftrightarrow (\forall B \in atms\_of\ CA.\ B < A)$

inductive *eligible* :: $'a$ *list* $\Rightarrow 'a$ *clause* $\Rightarrow$ *bool* **where**
  eligible: $(S\ DA = negs\ (mset\ As)) \vee (S\ DA = \{\#\} \wedge length\ As = 1 \wedge maximal\_wrt\ (As\ !\ 0)\ DA) \Longrightarrow$
    eligible As DA

lemma $(S\ DA = negs\ (mset\ As) \vee S\ DA = \{\#\} \wedge length\ As = 1 \wedge maximal\_wrt\ (As\ !\ 0)\ DA) \longleftrightarrow$
    eligible As DA
  using *eligible.intros ground_resolution_with_selection.eligible.cases ground_resolution_with_selection_axioms* **by** *blast*

inductive
  ord_resolve :: $'a$ *clause list* $\Rightarrow 'a$ *clause* $\Rightarrow 'a$ *multiset list* $\Rightarrow 'a$ *list* $\Rightarrow 'a$ *clause* $\Rightarrow$ *bool*
where
  ord_resolve:
    $length\ CAs = n \Longrightarrow$
    $length\ Cs = n \Longrightarrow$
    $length\ AAs = n \Longrightarrow$
    $length\ As = n \Longrightarrow$
    $n \neq 0 \Longrightarrow$
    $(\forall i < n.\ CAs\ !\ i = Cs\ !\ i + poss\ (AAs\ !\ i)) \Longrightarrow$
    $(\forall i < n.\ AAs\ !\ i \neq \{\#\}) \Longrightarrow$
    $(\forall i < n.\ \forall A \in\# AAs\ !\ i.\ A = As\ !\ i) \Longrightarrow$
    $eligible\ As\ (D + negs\ (mset\ As)) \Longrightarrow$
    $(\forall i < n.\ strictly\_maximal\_wrt\ (As\ !\ i)\ (Cs\ !\ i)) \Longrightarrow$
    $(\forall i < n.\ S\ (CAs\ !\ i) = \{\#\}) \Longrightarrow$
    $ord\_resolve\ CAs\ (D + negs\ (mset\ As))\ AAs\ As\ (\bigcup\#\ mset\ Cs + D)$

lemma *ord_resolve_sound*:
  assumes
    res_e: *ord_resolve CAs DA AAs As E* **and**
    cc_true: $I \models m\ mset\ CAs$ **and**
    d_true: $I \models DA$
  shows $I \models E$
  using *res_e*
proof (*cases rule*: *ord_resolve.cases*)
  case (*ord_resolve n Cs D*)
  note $DA = this(1)$ **and** $e = this(2)$ **and** $cas\_len = this(3)$ **and** $cs\_len = this(4)$ **and**
    $as\_len = this(6)$ **and** $cas = this(8)$ **and** $aas\_ne = this(9)$ **and** $a\_eq = this(10)$

  show *?thesis*
  proof (*cases* $\forall A \in set\ As.\ A \in I$)
    case *True*
    then have $\neg I \models negs\ (mset\ As)$
      unfolding *true_cls_def* **by** *fastforce*
    then have $I \models D$
      using *d_true DA* **by** *fast*
    then show *?thesis*
      unfolding *e* **by** *blast*
  next
    case *False*
    then obtain *i* **where**
      a_in_aa: $i < n$ **and**
      a_false: $As\ !\ i \notin I$
      using *cas_len as_len* **by** (*metis in_set_conv_nth*)

```
    have ¬ I ⊨ poss (AAs ! i)
      using a_false a_eq aas_ne a_in_aa unfolding true_cls_def by auto
    moreover have I ⊨ CAs ! i
      using a_in_aa cc_true unfolding true_cls_mset_def using cas_len by auto
    ultimately have I ⊨ Cs ! i
      using cas a_in_aa by auto
    then show ?thesis
      using a_in_aa cs_len unfolding e true_cls_def
      by (meson in_Union_mset_iff nth_mem_mset union_iff)
  qed
qed

lemma filter_neg_atm_of_S: {#Neg (atm_of L). L ∈# S C#} = S C
  by (simp add: S_selects_neg_lits)
```

This corresponds to Lemma 3.13:

```
lemma ord_resolve_reductive:
  assumes ord_resolve CAs DA AAs As E
  shows E < DA
  using assms
proof (cases rule: ord_resolve.cases)
  case (ord_resolve n Cs D)
  note DA = this(1) and e = this(2) and cas_len = this(3) and cs_len = this(4) and
    ai_len = this(6) and nz = this(7) and cas = this(8) and maxim = this(12)

  show ?thesis
  proof (cases ⋃# mset Cs = {#})
    case True
    have negs (mset As) ≠ {#}
       using nz ai_len by auto
    then show ?thesis
       unfolding True e DA by auto
  next
    case False

    define max_A_of_Cs where max_A_of_Cs = Max (atms_of (⋃# mset Cs))

    have
      mc_in: max_A_of_Cs ∈ atms_of (⋃# mset Cs) and
      mc_max: ⋀B. B ∈ atms_of (⋃# mset Cs) ⟹ B ≤ max_A_of_Cs
      using max_A_of_Cs_def False by auto

    then have ∃ C_max ∈ set Cs. max_A_of_Cs ∈ atms_of (C_max)
      by (metis atm_imp_pos_or_neg_lit in_Union_mset_iff neg_lit_in_atms_of pos_lit_in_atms_of
          set_mset_mset)
    then obtain max_i where
      cm_in_cas: max_i < length CAs and
      mc_in_cm: max_A_of_Cs ∈ atms_of (Cs ! max_i)
      using in_set_conv_nth[of _ CAs] by (metis cas_len cs_len in_set_conv_nth)

    define CA_max where CA_max = CAs ! max_i
    define A_max where A_max = As ! max_i
    define C_max where C_max = Cs ! max_i

    have mc_lt_ma: max_A_of_Cs < A_max
      using maxim cm_in_cas mc_in_cm cas_len unfolding strictly_maximal_wrt_def A_max_def by auto

    then have ucas_ne_neg_aa: (⋃# mset Cs) ≠ negs (mset As)
      using mc_in mc_max mc_lt_ma cm_in_cas cas_len ai_len unfolding A_max_def
      by (metis atms_of_negs nth_mem set_mset_mset leD)
    moreover have ucas_lt_ma: ∀ B ∈ atms_of (⋃# mset Cs). B < A_max
      using mc_max mc_lt_ma by fastforce
    moreover have ¬ Neg A_max ∈# (⋃# mset Cs)
```

using *ucas_lt_ma neg_lit_in_atms_of*[*of A_max* $\bigcup\#$ *mset Cs*] **by** *auto*
moreover have *Neg A_max* $\in\#$ *negs* (*mset As*)
using *cm_in_cas cas_len ai_len A_max_def* **by** *auto*
ultimately have ($\bigcup\#$ *mset Cs*) $<$ *negs* (*mset As*)
unfolding $\mathit{less\_multiset}_{HO}$
by (*metis* (*no_types*) *atms_less_eq_imp_lit_less_eq_neg count_greater_zero_iff count_inI le_imp_less_or_eq less_imp_not_less not_le*)
then show *?thesis*
unfolding *e DA* **by** *auto*
qed
qed

This corresponds to Theorem 3.15:

theorem *ord_resolve_counterex_reducing*:
assumes
ec_ni_n: $\{\#\} \notin N$ **and**
d_in_n: $DA \in N$ **and**
d_cex: $\neg$ *INTERP N* $\models$ *DA* **and**
d_min: $\bigwedge C.\ C \in N \Longrightarrow \neg$ *INTERP N* $\models C \Longrightarrow DA \leq C$
obtains *CAs AAs As E* **where**
set CAs $\subseteq N$
INTERP N $\models m$ *mset CAs*
$\bigwedge CA.\ CA \in$ *set CAs* $\Longrightarrow$ *productive N CA*
ord_resolve CAs DA AAs As E
$\neg$ *INTERP N* $\models E$
$E < DA$
proof $-$
have *d_ne*: $DA \neq \{\#\}$
using *d_in_n ec_ni_n* **by** *blast*
have $\exists$ *As. As* $\neq [] \wedge$ *negs* (*mset As*) $\leq\#$ *DA* $\wedge$ *eligible As DA*
proof (*cases S DA* $= \{\#\}$)
assume *s_d_e*: *S DA* $= \{\#\}$

define *A* **where** *A* = *Max* (*atms_of DA*)
define *As* **where** *As* = [*A*]
define *D* **where** *D* = *DA*$-\{\#$*Neg A* $\#\}$

have *na_in_d*: *Neg A* $\in\#$ *DA*
unfolding *A_def* **using** *s_d_e d_ne d_in_n d_cex d_min*
by (*metis Max_in_lits Max_lit_eq_pos_or_neg_Max_atm max_pos_imp_Interp Interp_imp_INTERP*)
then have *das*: *DA* = *D* + *negs* (*mset As*) **unfolding** *D_def As_def* **by** *auto*
moreover from *na_in_d* **have** *negs* (*mset As*) $\subseteq\#$ *DA*
by (*simp add*: *As_def*)
moreover have *As* ! *0* = *Max* (*atms_of* (*D* + *negs* (*mset As*)))
using *A_def As_def das* **by** *auto*
then have *eligible As DA*
using *eligible s_d_e As_def das maximal_wrt_def* **by** *auto*
ultimately show *?thesis*
using *As_def* **by** *blast*
next
assume *s_d_e*: *S DA* $\neq \{\#\}$

define *As* :: *'a list* **where**
As = *list_of_mset* $\{\#$*atm_of L. L* $\in\#$ *S DA*$\#\}$
define *D* :: *'a clause* **where**
D = *DA* $-$ *negs* $\{\#$*atm_of L. L* $\in\#$ *S DA*$\#\}$

have *As* $\neq []$ **unfolding** *As_def* **using** *s_d_e*
by (*metis image_mset_is_empty_iff list_of_mset_empty*)
moreover have *da_sub_as*: *negs* $\{\#$*atm_of L. L* $\in\#$ *S DA*$\#\}$ $\subseteq\#$ *DA*
using *S_selects_subseteq* **by** (*auto simp*: *filter_neg_atm_of_S*)
then have *negs* (*mset As*) $\subseteq\#$ *DA*
unfolding *As_def* **by** *auto*

```
    moreover have das: DA = D + negs (mset As)
      using da_sub_as unfolding D_def As_def by auto
    moreover have S DA = negs {#atm_of L. L ∈# S DA#}
      by (auto simp: filter_neg_atm_of_S)
    then have S DA = negs (mset As)
      unfolding As_def by auto
    then have eligible As DA
      unfolding das using eligible by auto
    ultimately show ?thesis
      by blast
  qed
  then obtain As :: 'a list where
    as_ne: As ≠ [] and
    negs_as_le_d: negs (mset As) ≤# DA and
    s_d: eligible As DA
    by blast

  define D :: 'a clause where
    D = DA − negs (mset As)

  have set As ⊆ INTERP N
    using d_cex negs_as_le_d by force
  then have prod_ex: ∀ A ∈ set As. ∃ D. produces N D A
    unfolding INTERP_def
    by (metis (no_types, lifting) INTERP_def subsetCE UN_E not_produces_imp_notin_production)
  then have ⋀A. ∃ D. produces N D A ⟶ A ∈ set As
    using ec_ni_n by (auto intro: productive_in_N)
  then have ⋀A. ∃ D. produces N D A ⟷ A ∈ set As
    using prod_ex by blast
  then obtain CA_of where c_of0: ⋀A. produces N (CA_of A) A ⟷ A ∈ set As
    by metis
  then have prod_c0: ∀ A ∈ set As. produces N (CA_of A) A
    by blast

  define C_of where
    ⋀A. C_of A = {#L ∈# CA_of A. L ≠ Pos A#}
  define Aj_of where
    ⋀A. Aj_of A = image_mset atm_of {#L ∈# CA_of A. L = Pos A#}

  have pospos: ⋀LL A. {#Pos (atm_of x). x ∈# {#L ∈# LL. L = Pos A#}#} = {#L ∈# LL. L = Pos A#}
    by (metis (mono_tags, lifting) image_filter_cong literal.sel(1) multiset.map_ident)
  have ca_of_c_of_aj_of: ⋀A. CA_of A = C_of A + poss (Aj_of A)
    using pospos[of _ CA_of _] by (simp add: C_of_def Aj_of_def add.commute multiset_partition)

  define n :: nat where
    n = length As
  define Cs :: 'a clause list where
    Cs = map C_of As
  define AAs :: 'a multiset list where
    AAs = map Aj_of As
  define CAs :: 'a literal multiset list where
    CAs = map CA_of As

  have m_nz: ⋀A. A ∈ set As ⟹ Aj_of A ≠ {#}
    unfolding Aj_of_def using prod_c0 produces_imp_Pos_in_lits
    by (metis (full_types) filter_mset_empty_conv image_mset_is_empty_iff)

  have prod_c: productive N CA if ca_in: CA ∈ set CAs for CA
  proof −
    obtain i where i_p: i < length CAs CAs ! i = CA
      using ca_in by (meson in_set_conv_nth)
    have production N (CA_of (As ! i)) = {As ! i}
      using i_p CAs_def prod_c0 by auto
```

```
    then show productive N CA
      using i_p CAs_def by auto
  qed
  then have cs_subs_n: set CAs ⊆ N
    using productive_in_N by auto
  have cs_true: INTERP N ⊨m mset CAs
    unfolding true_cls_mset_def using prod_c productive_imp_INTERP by auto

  have ⋀A. A ∈ set As ⟹ ¬ Neg A ∈# CA_of A
    using prod_c0 produces_imp_neg_notin_lits by auto
  then have a_ni_c': ⋀A. A ∈ set As ⟹ A ∉ atms_of (C_of A)
    unfolding C_of_def using atm_imp_pos_or_neg_lit by force
  have c'_le_c: ⋀A. C_of A ≤ CA_of A
    unfolding C_of_def by (auto intro: subset_eq_imp_le_multiset)
  have a_max_c: ⋀A. A ∈ set As ⟹ A = Max (atms_of (CA_of A))
    using prod_c0 productive_imp_produces_Max_atom[of N] by auto
  then have ⋀A. A ∈ set As ⟹ C_of A ≠ {#} ⟹ Max (atms_of (C_of A)) ≤ A
    using c'_le_c by (metis less_eq_Max_atms_of)
  moreover have ⋀A. A ∈ set As ⟹ C_of A ≠ {#} ⟹ Max (atms_of (C_of A)) ≠ A
    using a_ni_c' Max_in by (metis (no_types) atms_empty_iff_empty finite_atms_of)
  ultimately have max_c'_lt_a: ⋀A. A ∈ set As ⟹ C_of A ≠ {#} ⟹ Max (atms_of (C_of A)) < A
    by (metis order.strict_iff_order)

  have le_cs_as: length CAs = length As
    unfolding CAs_def by simp

  have length CAs = n
    by (simp add: le_cs_as n_def)
  moreover have length Cs = n
    by (simp add: Cs_def n_def)
  moreover have length AAs = n
    by (simp add: AAs_def n_def)
  moreover have length As = n
    using n_def by auto
  moreover have n ≠ 0
    by (simp add: as_ne n_def)
  moreover have  ∀i. i < length AAs ⟶ (∀A ∈# AAs ! i. A = As ! i)
    using AAs_def Aj_of_def by auto

  have ⋀x B. production N (CA_of x) = {x} ⟹ B ∈# CA_of x ⟹ B ≠ Pos x ⟹ atm_of B < x
    by (metis atm_of_lit_in_atms_of insert_not_empty le_imp_less_or_eq Pos_atm_of_iff
        Neg_atm_of_iff pos_neg_in_imp_true produces_imp_Pos_in_lits produces_imp_atms_leq
        productive_imp_not_interp)
  then have ⋀B A. A∈set As ⟹ B ∈# CA_of A ⟹ B ≠ Pos A ⟹ atm_of B < A
    using prod_c0 by auto
  have ∀i. i < length AAs ⟶ AAs ! i ≠ {#}
    unfolding AAs_def using m_nz by simp
  have ∀i < n. CAs ! i = Cs ! i + poss (AAs ! i)
    unfolding CAs_def Cs_def AAs_def using ca_of_c_of_aj_of by (simp add: n_def)

  moreover have ∀i < n. AAs ! i ≠ {#}
    using ‹∀i < length AAs. AAs ! i ≠ {#}› calculation(3) by blast
  moreover have ∀i < n. ∀A ∈# AAs ! i. A = As ! i
    by (simp add: ‹∀i < length AAs. ∀A ∈# AAs ! i. A = As ! i› calculation(3))
  moreover have eligible As DA
    using s_d by auto
  then have eligible As (D + negs (mset As))
    using D_def negs_as_le_d by auto
  moreover have ⋀i. i < length AAs ⟹ strictly_maximal_wrt (As ! i) ((Cs ! i))
    by (simp add: C_of_def Cs_def ‹⋀x B. ⟦production N (CA_of x) = {x}; B ∈# CA_of x; B ≠ Pos x⟧ ⟹ atm_of
B < x› atms_of_def calculation(3) n_def prod_c0 strictly_maximal_wrt_def)

  have ∀i < n. strictly_maximal_wrt (As ! i) (Cs ! i)
```

by (*simp add*: ‹⋀*i*. *i* < *length AAs* ⟹ *strictly_maximal_wrt* (*As* ! *i*) (*Cs* ! *i*)› *calculation*(*3*))
moreover have ∀ *CA* ∈ *set CAs*. *S CA* = {#}
using *prod_c producesD productive_imp_produces_Max_literal* **by** *blast*
have ∀ *CA*∈*set CAs*. *S CA* = {#}
using ‹∀ *CA*∈*set CAs*. *S CA* = {#}› **by** *simp*
then have ∀ *i* < *n*. *S* (*CAs* ! *i*) = {#}
using ‹*length CAs* = *n*› *nth_mem* **by** *blast*
ultimately have *res_e*: *ord_resolve CAs* (*D* + *negs* (*mset As*)) *AAs As* (⋃# *mset Cs* + *D*)
using *ord_resolve* **by** *auto*

have ⋀*A*. *A* ∈ *set As* ⟹ ¬ *interp N* (*CA_of A*) ⊨ *CA_of A*
by (*simp add*: *prod_c0 producesD*)
then have ⋀*A*. *A* ∈ *set As* ⟹ ¬ *Interp N* (*CA_of A*) ⊨ *C_of A*
unfolding *prod_c0 C_of_def Interp_def true_cls_def* **using** *true_lit_def not_gr_zero prod_c0*
by *auto*
then have *c′_at_n*: ⋀*A*. *A* ∈ *set As* ⟹ ¬ *INTERP N* ⊨ *C_of A*
using *a_max_c c′_le_c max_c′_lt_a not_Interp_imp_not_INTERP* **unfolding** *true_cls_def*
by (*metis true_cls_def true_cls_empty*)

have ¬ *INTERP N* ⊨ ⋃# *mset Cs*
unfolding *Cs_def true_cls_def* **by** (*auto dest*!: *c′_at_n*)
moreover have ¬ *INTERP N* ⊨ *D*
using *d_cex* **by** (*metis D_def add_diff_cancel_right′ negs_as_le_d subset_mset.add_diff_assoc2 true_cls_def union_iff*)
ultimately have *e_cex*: ¬ *INTERP N* ⊨ ⋃# *mset Cs* + *D*
by *simp*

have *set CAs* ⊆ *N*
by (*simp add*: *cs_subs_n*)
moreover have *INTERP N* ⊨*m mset CAs*
by (*simp add*: *cs_true*)
moreover have ⋀*CA*. *CA* ∈ *set CAs* ⟹ *productive N CA*
by (*simp add*: *prod_c*)
moreover have *ord_resolve CAs DA AAs As* (⋃# *mset Cs* + *D*)
using *D_def negs_as_le_d res_e* **by** *auto*
moreover have ¬ *INTERP N* ⊨ ⋃# *mset Cs* + *D*
using *e_cex* **by** *simp*
moreover have (⋃# *mset Cs* + *D*) < *DA*
using *calculation*(*4*) *ord_resolve_reductive* **by** *auto*
ultimately show *thesis*
..
qed

lemma *ord_resolve_atms_of_concl_subset*:
assumes *ord_resolve CAs DA AAs As E*
shows *atms_of E* ⊆ (⋃ *C* ∈ *set CAs*. *atms_of C*) ∪ *atms_of DA*
using *assms*
proof (*cases rule*: *ord_resolve.cases*)
case (*ord_resolve n Cs D*)
note *DA* = *this*(*1*) **and** *e* = *this*(*2*) **and** *cas_len* = *this*(*3*) **and** *cs_len* = *this*(*4*) **and** *cas* = *this*(*8*)

have ∀ *i* < *n*. *set_mset* (*Cs* ! *i*) ⊆ *set_mset* (*CAs* ! *i*)
using *cas* **by** *auto*
then have ∀ *i* < *n*. *Cs* ! *i* ⊆# ⋃# *mset CAs*
by (*metis cas cas_len mset_subset_eq_add_left nth_mem_mset sum_mset.remove union_assoc*)
then have ∀ *C* ∈ *set Cs*. *C* ⊆# ⋃# *mset CAs*
using *cs_len in_set_conv_nth*[*of _ Cs*] **by** *auto*
then have *set_mset* (⋃# *mset Cs*) ⊆ *set_mset* (⋃# *mset CAs*)
by *auto* (*meson in_mset_sum_list2 mset_subset_eqD*)
then have *atms_of* (⋃# *mset Cs*) ⊆ *atms_of* (⋃# *mset CAs*)
by (*meson lits_subseteq_imp_atms_subseteq mset_subset_eqD subsetI*)
moreover have *atms_of* (⋃# *mset CAs*) = (⋃ *CA* ∈ *set CAs*. *atms_of CA*)
by (*intro set_eqI iffI*, *simp_all*,

metis in_mset_sum_list2 atm_imp_pos_or_neg_lit neg_lit_in_atms_of pos_lit_in_atms_of,
metis in_mset_sum_list atm_imp_pos_or_neg_lit neg_lit_in_atms_of pos_lit_in_atms_of)
ultimately have *atms_of* ($\bigcup\#$ *mset Cs*) $\subseteq$ ($\bigcup$ *CA* $\in$ *set CAs*. *atms_of CA*)
by *auto*
moreover have *atms_of D* $\subseteq$ *atms_of DA*
using *DA* **by** *auto*
ultimately show *?thesis*
unfolding *e* **by** *auto*
qed

11.2 Inference System

Theorem 3.16 is subsumed in the counterexample-reducing inference system framework, which is instantiated below. Unlike its unordered cousin, ordered resolution is additionally a reductive inference system.

definition *ord_Γ* :: *'a inference set* **where**
ord_Γ = {*Infer* (*mset CAs*) *DA E* | *CAs DA AAs As E*. *ord_resolve CAs DA AAs As E*}

sublocale *ord_Γ_sound_counterex_reducing?*:
sound_counterex_reducing_inference_system ground_resolution_with_selection.ord_Γ S
ground_resolution_with_selection.INTERP S +
reductive_inference_system ground_resolution_with_selection.ord_Γ S
proof *unfold_locales*
fix *DA* :: *'a clause* **and** *N* :: *'a clause set*
assume {#} $\notin$ *N* **and** *DA* $\in$ *N* **and** $\neg$ *INTERP N* $\models$ *DA* **and** $\bigwedge$*C*. *C* $\in$ *N* $\Longrightarrow$ $\neg$ *INTERP N* $\models$ *C* $\Longrightarrow$ *DA* $\leq$ *C*
then obtain *CAs AAs As E* **where**
dd_sset_n: *set CAs* $\subseteq$ *N* **and**
dd_true: *INTERP N* $\models$*m mset CAs* **and**
res_e: *ord_resolve CAs DA AAs As E* **and**
e_cex: $\neg$ *INTERP N* $\models$ *E* **and**
e_lt_c: *E* < *DA*
using *ord_resolve_counterex_reducing*[*of N DA thesis*] **by** *auto*

have *Infer* (*mset CAs*) *DA E* $\in$ *ord_Γ*
using *res_e* **unfolding** *ord_Γ_def* **by** (*metis* (*mono_tags*, *lifting*) *mem_Collect_eq*)
then show $\exists$ *CC E*. *set_mset CC* $\subseteq$ *N* $\wedge$ *INTERP N* $\models$*m CC* $\wedge$ *Infer CC DA E* $\in$ *ord_Γ*
$\wedge$ $\neg$ *INTERP N* $\models$ *E* $\wedge$ *E* < *DA*
using *dd_sset_n dd_true e_cex e_lt_c* **by** (*metis set_mset_mset*)
qed (*auto simp*: *ord_Γ_def intro*: *ord_resolve_sound ord_resolve_reductive*)

lemmas *clausal_logic_compact* = *ord_Γ_sound_counterex_reducing.clausal_logic_compact*

end

A second proof of Theorem 3.12, compactness of clausal logic:

lemmas *clausal_logic_compact* = *ground_resolution_with_selection.clausal_logic_compact*

end

12 Theorem Proving Processes

theory *Proving_Process*
imports *Unordered_Ground_Resolution Lazy_List_Chain*
begin

This material corresponds to Section 4.1 ("Theorem Proving Processes") of Bachmair and Ganzinger's chapter.

The locale assumptions below capture conditions R1 to R3 of Definition 4.1. *Rf* denotes $\mathcal{R}_{\mathcal{F}}$; *Ri* denotes $\mathcal{R}_{\mathcal{I}}$.

locale *redundancy_criterion* = *inference_system* +
fixes

  Rf :: *'a clause set* ⇒ *'a clause set* **and**
  Ri :: *'a clause set* ⇒ *'a inference set*
assumes
  Ri_subset_Γ: *Ri N* ⊆ Γ **and**
  Rf_mono: *N* ⊆ *N'* ⟹ *Rf N* ⊆ *Rf N'* **and**
  Ri_mono: *N* ⊆ *N'* ⟹ *Ri N* ⊆ *Ri N'* **and**
  Rf_indep: *N'* ⊆ *Rf N* ⟹ *Rf N* ⊆ *Rf* (*N* − *N'*) **and**
  Ri_indep: *N'* ⊆ *Rf N* ⟹ *Ri N* ⊆ *Ri* (*N* − *N'*) **and**
  Rf_sat: *satisfiable* (*N* − *Rf N*) ⟹ *satisfiable N*
begin

definition *saturated_upto* :: *'a clause set* ⇒ *bool* **where**
saturated_upto N ⟷ *inferences_from* (*N* − *Rf N*) ⊆ *Ri N*

inductive *derive* :: *'a clause set* ⇒ *'a clause set* ⇒ *bool* (**infix** ▷ *50*) **where**
deduction_deletion: *N* − *M* ⊆ *concls_of* (*inferences_from M*) ⟹ *M* − *N* ⊆ *Rf N* ⟹ *M* ▷ *N*

lemma *derive_subset*: *M* ▷ *N* ⟹ *N* ⊆ *M* ∪ *concls_of* (*inferences_from M*)
by (*meson Diff_subset_conv derive.cases*)

end

locale *sat_preserving_redundancy_criterion* =
sat_preserving_inference_system Γ :: (*'a* :: *wellorder*) *inference set* + *redundancy_criterion*
begin

lemma *deriv_sat_preserving*:
assumes
  deriv: *chain* (*op* ▷) *Ns* **and**
  sat_n0: *satisfiable* (*lhd Ns*)
shows *satisfiable* (*Sup_llist Ns*)
proof −
  have *ns0*: *lnth Ns 0* = *lhd Ns*
    using *deriv* **by** (*metis chain_not_lnull lhd_conv_lnth*)
  have *len_ns*: *llength Ns* > *0*
    using *deriv* **by** (*case_tac Ns*) *simp+*
  {
    fix *DD*
    assume *fin*: *finite DD* **and** *sset_lun*: *DD* ⊆ *Sup_llist Ns*
    then obtain *k* **where** *dd_sset*: *DD* ⊆ *Sup_upto_llist Ns k*
      using *finite_Sup_llist_imp_Sup_upto_llist* **by** *blast*
    have *satisfiable* (*Sup_upto_llist Ns k*)
    proof (*induct k*)
      case *0*
      then show *?case*
        using *len_ns ns0 sat_n0* **unfolding** *Sup_upto_llist_def true_clss_def* **by** *auto*
    next
      case (*Suc k*)
      show *?case*
      proof (*cases enat* (*Suc k*) ≥ *llength Ns*)
        case *True*
        then have *Sup_upto_llist Ns k* = *Sup_upto_llist Ns* (*Suc k*)
          unfolding *Sup_upto_llist_def* **using** *le_Suc_eq not_less* **by** *blast*
        then show *?thesis*
          using *Suc* **by** *simp*
      next
        case *False*
        then have *lnth Ns k* ▷ *lnth Ns* (*Suc k*)
          using *deriv* **by** (*auto simp*: *chain_lnth_rel*)
        then have *lnth Ns* (*Suc k*) ⊆ *lnth Ns k* ∪ *concls_of* (*inferences_from* (*lnth Ns k*))
          by (*rule derive_subset*)
        moreover have *lnth Ns k* ⊆ *Sup_upto_llist Ns k*
          unfolding *Sup_upto_llist_def* **using** *False Suc_ile_eq linear* **by** *blast*

```
      ultimately have lnth Ns (Suc k)
        ⊆ Sup_upto_llist Ns k ∪ concls_of (inferences_from (Sup_upto_llist Ns k))
        by clarsimp (metis UnCI UnE image_Un inferences_from_mono le_iff_sup)
      moreover have Sup_upto_llist Ns (Suc k) = Sup_upto_llist Ns k ∪ lnth Ns (Suc k)
        unfolding Sup_upto_llist_def using False by (force elim: le_SucE)
      moreover have
        satisfiable (Sup_upto_llist Ns k ∪ concls_of (inferences_from (Sup_upto_llist Ns k)))
        using Suc Γ_sat_preserving unfolding sat_preserving_inference_system_def by simp
      ultimately show ?thesis
        by (metis le_iff_sup true_clss_union)
    qed
  qed
  then have satisfiable DD
    using dd_sset unfolding Sup_upto_llist_def by (blast intro: true_clss_mono)
 }
 then show ?thesis
   using ground_resolution_without_selection.clausal_logic_compact[THEN iffD1] by metis
qed
```

This corresponds to Lemma 4.2:

```
lemma
 assumes deriv: chain (op ▷) Ns
 shows
   Rf_Sup_subset_Rf_Liminf: Rf (Sup_llist Ns) ⊆ Rf (Liminf_llist Ns) and
   Ri_Sup_subset_Ri_Liminf: Ri (Sup_llist Ns) ⊆ Ri (Liminf_llist Ns) and
   sat_deriv_Liminf_iff: satisfiable (Liminf_llist Ns) ⟷ satisfiable (lhd Ns)
proof −
 {
  fix C i j
  assume
    c_in: C ∈ lnth Ns i and
    c_ni: C ∉ Rf (Sup_llist Ns) and
    j: j ≥ i and
    j': enat j < llength Ns
  from c_ni have c_ni': ⋀i. enat i < llength Ns ⟹ C ∉ Rf (lnth Ns i)
    using Rf_mono lnth_subset_Sup_llist Sup_llist_def by (blast dest: contra_subsetD)
  have C ∈ lnth Ns j
  using j j'
  proof (induct j)
    case 0
    then show ?case
      using c_in by blast
  next
    case (Suc k)
    then show ?case
    proof (cases i < Suc k)
      case True
      have i ≤ k
        using True by linarith
      moreover have enat k < llength Ns
        using Suc.prems(2) Suc_ile_eq by (blast intro: dual_order.strict_implies_order)
      ultimately have c_in_k: C ∈ lnth Ns k
        using Suc.hyps by blast
      have rel: lnth Ns k ▷ lnth Ns (Suc k)
        using Suc.prems deriv by (auto simp: chain_lnth_rel)
      then show ?thesis
        using c_in_k c_ni' Suc.prems(2) by cases auto
    next
      case False
      then show ?thesis
        using Suc c_in by auto
    qed
  qed
```

```
  }
  then have lu_ll: Sup_llist Ns − Rf (Sup_llist Ns) ⊆ Liminf_llist Ns
    unfolding Sup_llist_def Liminf_llist_def by blast
  have rf: Rf (Sup_llist Ns − Rf (Sup_llist Ns)) ⊆ Rf (Liminf_llist Ns)
    using lu_ll Rf_mono by simp
  have ri: Ri (Sup_llist Ns − Rf (Sup_llist Ns)) ⊆ Ri (Liminf_llist Ns)
    using lu_ll Ri_mono by simp
  show Rf (Sup_llist Ns) ⊆ Rf (Liminf_llist Ns)
    using rf Rf_indep by blast
  show Ri (Sup_llist Ns) ⊆ Ri (Liminf_llist Ns)
    using ri Ri_indep by blast

  show satisfiable (Liminf_llist Ns) ⟷ satisfiable (lhd Ns)
  proof
    assume satisfiable (lhd Ns)
    then have satisfiable (Sup_llist Ns)
      using deriv deriv_sat_preserving by simp
    then show satisfiable (Liminf_llist Ns)
      using true_clss_mono[OF Liminf_llist_subset_Sup_llist] by blast
  next
    assume satisfiable (Liminf_llist Ns)
    then have satisfiable (Sup_llist Ns − Rf (Sup_llist Ns))
      using true_clss_mono[OF lu_ll] by blast
    then have satisfiable (Sup_llist Ns)
      using Rf_sat by blast
    then show satisfiable (lhd Ns)
      using deriv true_clss_mono lhd_subset_Sup_llist chain_not_lnull by metis
  qed
qed

lemma
  assumes chain (op ▷) Ns
  shows
    Rf_Liminf_eq_Rf_Sup: Rf (Liminf_llist Ns) = Rf (Sup_llist Ns) and
    Ri_Liminf_eq_Ri_Sup: Ri (Liminf_llist Ns) = Ri (Sup_llist Ns)
  using assms
  by (auto simp: Rf_Sup_subset_Rf_Liminf Rf_mono Ri_Sup_subset_Ri_Liminf Ri_mono
      Liminf_llist_subset_Sup_llist subset_antisym)

end
```

The assumption below corresponds to condition R4 of Definition 4.1.

```
locale effective_redundancy_criterion = redundancy_criterion +
  assumes Ri_effective: γ ∈ Γ ⟹ concl_of γ ∈ N ∪ Rf N ⟹ γ ∈ Ri N
begin

definition fair_clss_seq :: 'a clause set llist ⇒ bool where
  fair_clss_seq Ns ⟷ (let N' = Liminf_llist Ns − Rf (Liminf_llist Ns) in
     concls_of (inferences_from N' − Ri N') ⊆ Sup_llist Ns ∪ Rf (Sup_llist Ns))

end

locale sat_preserving_effective_redundancy_criterion =
  sat_preserving_inference_system Γ :: ('a :: wellorder) inference set +
  effective_redundancy_criterion
begin

sublocale sat_preserving_redundancy_criterion
  ..
```

The result below corresponds to Theorem 4.3.

```
theorem fair_derive_saturated_upto:
  assumes
```

deriv: *chain* (*op* ▷) *Ns* **and**
fair: *fair_clss_seq Ns*
shows *saturated_upto* (*Liminf_llist Ns*)
unfolding *saturated_upto_def*
proof
fix γ
let *?N′* = *Liminf_llist Ns* − *Rf* (*Liminf_llist Ns*)
assume γ: γ ∈ *inferences_from ?N′*
show γ ∈ *Ri* (*Liminf_llist Ns*)
proof (*cases* γ ∈ *Ri ?N′*)
case *True*
then show *?thesis*
using *Ri_mono* **by** *blast*
next
case *False*
have *concls_of* (*inferences_from ?N′* − *Ri ?N′*) ⊆ *Sup_llist Ns* ∪ *Rf* (*Sup_llist Ns*)
using *fair* **unfolding** *fair_clss_seq_def Let_def* **.**
then have *concl_of* γ ∈ *Sup_llist Ns* ∪ *Rf* (*Sup_llist Ns*)
using *False* γ **by** *auto*
moreover
{
assume *concl_of* γ ∈ *Sup_llist Ns*
then have γ ∈ *Ri* (*Sup_llist Ns*)
using γ *Ri_effective inferences_from_def* **by** *blast*
then have γ ∈ *Ri* (*Liminf_llist Ns*)
using *deriv Ri_Sup_subset_Ri_Liminf* **by** *fast*
}
moreover
{
assume *concl_of* γ ∈ *Rf* (*Sup_llist Ns*)
then have *concl_of* γ ∈ *Rf* (*Liminf_llist Ns*)
using *deriv Rf_Sup_subset_Rf_Liminf* **by** *blast*
then have γ ∈ *Ri* (*Liminf_llist Ns*)
using γ *Ri_effective inferences_from_def* **by** *auto*
}
ultimately show γ ∈ *Ri* (*Liminf_llist Ns*)
by *blast*
qed
qed

end

This corresponds to the trivial redundancy criterion defined on page 36 of Section 4.1.

locale *trivial_redundancy_criterion* = *inference_system*
begin

definition *Rf* :: *′a clause set* ⇒ *′a clause set* **where**
Rf _ = {}

definition *Ri* :: *′a clause set* ⇒ *′a inference set* **where**
Ri N = {γ. γ ∈ Γ ∧ *concl_of* γ ∈ *N*}

sublocale *effective_redundancy_criterion* Γ *Rf Ri*
by *unfold_locales* (*auto simp*: *Rf_def Ri_def*)

lemma *saturated_upto_iff*: *saturated_upto N* ⟷ *concls_of* (*inferences_from N*) ⊆ *N*
unfolding *saturated_upto_def inferences_from_def Rf_def Ri_def* **by** *auto*

end

The following lemmas corresponds to the standard extension of a redundancy criterion defined on page 38 of Section 4.1.

lemma *redundancy_criterion_standard_extension*:

assumes $\Gamma \subseteq \Gamma'$ **and** *redundancy_criterion* Γ *Rf Ri*
shows *redundancy_criterion* Γ' *Rf* $(\lambda N.\ Ri\ N \cup (\Gamma' - \Gamma))$
using *assms* **unfolding** *redundancy_criterion_def* **by** (*intro conjI*) ((*auto simp*: *rev_subsetD*)[5], *sat*)

lemma *redundancy_criterion_standard_extension_saturated_upto_iff*:
assumes $\Gamma \subseteq \Gamma'$ **and** *redundancy_criterion* Γ *Rf Ri*
shows *redundancy_criterion.saturated_upto* Γ *Rf Ri M* $\longleftrightarrow$
redundancy_criterion.saturated_upto Γ' *Rf* $(\lambda N.\ Ri\ N \cup (\Gamma' - \Gamma))$ *M*
using *assms redundancy_criterion.saturated_upto_def redundancy_criterion.saturated_upto_def*
redundancy_criterion_standard_extension
unfolding *inference_system.inferences_from_def* **by** *blast*

lemma *redundancy_criterion_standard_extension_effective*:
assumes $\Gamma \subseteq \Gamma'$ **and** *effective_redundancy_criterion* Γ *Rf Ri*
shows *effective_redundancy_criterion* Γ' *Rf* $(\lambda N.\ Ri\ N \cup (\Gamma' - \Gamma))$
using *assms redundancy_criterion_standard_extension*[*of* Γ]
unfolding *effective_redundancy_criterion_def effective_redundancy_criterion_axioms_def* **by** *auto*

lemma *redundancy_criterion_standard_extension_fair_iff*:
assumes $\Gamma \subseteq \Gamma'$ **and** *effective_redundancy_criterion* Γ *Rf Ri*
shows *effective_redundancy_criterion.fair_clss_seq* Γ' *Rf* $(\lambda N.\ Ri\ N \cup (\Gamma' - \Gamma))$ *Ns* $\longleftrightarrow$
effective_redundancy_criterion.fair_clss_seq Γ *Rf Ri Ns*
using *assms redundancy_criterion_standard_extension_effective*[*of* Γ Γ' *Rf Ri*]
effective_redundancy_criterion.fair_clss_seq_def[*of* Γ *Rf Ri Ns*]
effective_redundancy_criterion.fair_clss_seq_def[*of* Γ' *Rf* $(\lambda N.\ Ri\ N \cup (\Gamma' - \Gamma))$ *Ns*]
unfolding *inference_system.inferences_from_def Let_def* **by** *auto*

theorem *redundancy_criterion_standard_extension_fair_derive_saturated_upto*:
assumes
subs: $\Gamma \subseteq \Gamma'$ **and**
red: *redundancy_criterion* Γ *Rf Ri* **and**
red': *sat_preserving_effective_redundancy_criterion* Γ' *Rf* $(\lambda N.\ Ri\ N \cup (\Gamma' - \Gamma))$ **and**
deriv: *chain* (*redundancy_criterion.derive* Γ' *Rf*) *Ns* **and**
fair: *effective_redundancy_criterion.fair_clss_seq* Γ' *Rf* $(\lambda N.\ Ri\ N \cup (\Gamma' - \Gamma))$ *Ns*
shows *redundancy_criterion.saturated_upto* Γ *Rf Ri* (*Liminf_llist Ns*)
proof −
have *redundancy_criterion.saturated_upto* Γ' *Rf* $(\lambda N.\ Ri\ N \cup (\Gamma' - \Gamma))$ (*Liminf_llist Ns*)
by (*rule sat_preserving_effective_redundancy_criterion.fair_derive_saturated_upto*
[*OF red' deriv fair*])
then show *?thesis*
by (*rule redundancy_criterion_standard_extension_saturated_upto_iff*[*THEN iffD2*, *OF subs red*])
qed

end

13 The Standard Redundancy Criterion

theory *Standard_Redundancy*
imports *Proving_Process*
begin

This material is based on Section 4.2.2 ("The Standard Redundancy Criterion") of Bachmair and Ganzinger's chapter.

locale *standard_redundancy_criterion* =
inference_system Γ **for** Γ :: (*'a* :: *wellorder*) *inference set*
begin

abbreviation *redundant_infer* :: *'a clause set* $\Rightarrow$ *'a inference* $\Rightarrow$ *bool* **where**
redundant_infer $N\ \gamma \equiv$
$\exists DD.$ *set_mset DD* $\subseteq N \wedge (\forall I.\ I \models m\ DD +$ *side_prems_of* $\gamma \longrightarrow I \models$ *concl_of* $\gamma)$
$\wedge\ (\forall D.\ D \in\#\ DD \longrightarrow D <$ *main_prem_of* $\gamma)$

definition Rf :: $'a\ clause\ set \Rightarrow 'a\ clause\ set$ **where**
$Rf\ N = \{C.\ \exists DD.\ set\_mset\ DD \subseteq N \wedge (\forall I.\ I \models m\ DD \longrightarrow I \models C) \wedge (\forall D.\ D \in\#\ DD \longrightarrow D < C)\}$

definition Ri :: $'a\ clause\ set \Rightarrow 'a\ inference\ set$ **where**
$Ri\ N = \{\gamma \in \Gamma.\ redundant\_infer\ N\ \gamma\}$

lemma *tautology_redundant*:
assumes $Pos\ A \in\#\ C$
assumes $Neg\ A \in\#\ C$
shows $C \in Rf\ N$
proof −
have $set\_mset\ \{\#\} \subseteq N \wedge (\forall I.\ I \models m\ \{\#\} \longrightarrow I \models C) \wedge (\forall D.\ D \in\#\ \{\#\} \longrightarrow D < C)$
using *assms* **by** *auto*
then show $C \in Rf\ N$
unfolding *Rf_def* **by** *blast*
qed

lemma *contradiction_Rf*: $\{\#\} \in N \Longrightarrow Rf\ N = UNIV - \{\{\#\}\}$
unfolding *Rf_def* **by** *force*

The following results correspond to Lemma 4.5. The lemma *wlog_non_Rf* generalizes the core of the argument.

lemma *Rf_mono*: $N \subseteq N' \Longrightarrow Rf\ N \subseteq Rf\ N'$
unfolding *Rf_def* **by** *auto*

lemma *wlog_non_Rf*:
assumes *ex*: $\exists DD.\ set\_mset\ DD \subseteq N \wedge (\forall I.\ I \models m\ DD + CC \longrightarrow I \models E) \wedge (\forall D'.\ D' \in\#\ DD \longrightarrow D' < D)$
shows $\exists DD.\ set\_mset\ DD \subseteq N - Rf\ N \wedge (\forall I.\ I \models m\ DD + CC \longrightarrow I \models E) \wedge (\forall D'.\ D' \in\#\ DD \longrightarrow D' < D)$
proof −
from *ex* **obtain** $DD0$ **where**
dd0: $DD0 \in \{DD.\ set\_mset\ DD \subseteq N \wedge (\forall I.\ I \models m\ DD + CC \longrightarrow I \models E) \wedge (\forall D'.\ D' \in\#\ DD \longrightarrow D' < D)\}$
by *blast*
have $\exists DD.\ set\_mset\ DD \subseteq N \wedge (\forall I.\ I \models m\ DD + CC \longrightarrow I \models E) \wedge (\forall D'.\ D' \in\#\ DD \longrightarrow D' < D) \wedge$
$(\forall DD'.\ set\_mset\ DD' \subseteq N \wedge (\forall I.\ I \models m\ DD' + CC \longrightarrow I \models E) \wedge (\forall D'.\ D' \in\#\ DD' \longrightarrow D' < D) \longrightarrow$
$DD \leq DD')$
using *wf_eq_minimal*[*THEN iffD1*, *rule_format*, *OF wf_less_multiset dd0*]
unfolding *not_le*[*symmetric*] **by** *blast*
then obtain DD **where**
dd_subs_n: $set\_mset\ DD \subseteq N$ **and**
ddcc_imp_e: $\forall I.\ I \models m\ DD + CC \longrightarrow I \models E$ **and**
dd_lt_d: $\forall D'.\ D' \in\#\ DD \longrightarrow D' < D$ **and**
d_min: $\forall DD'.\ set\_mset\ DD' \subseteq N \wedge (\forall I.\ I \models m\ DD' + CC \longrightarrow I \models E) \wedge (\forall D'.\ D' \in\#\ DD' \longrightarrow D' < D) \longrightarrow$
$DD \leq DD'$
by *blast*

have $\forall Da.\ Da \in\#\ DD \longrightarrow Da \notin Rf\ N$
proof *clarify*
fix Da
assume
da_in_dd: $Da \in\#\ DD$ **and**
da_rf: $Da \in Rf\ N$

from *da_rf* **obtain** DD' **where**
dd'_subs_n: $set\_mset\ DD' \subseteq N$ **and**
dd'_imp_da: $\forall I.\ I \models m\ DD' \longrightarrow I \models Da$ **and**
dd'_lt_da: $\forall D'.\ D' \in\#\ DD' \longrightarrow D' < Da$
unfolding *Rf_def* **by** *blast*

define DDa **where**
$DDa = DD - \{\#Da\#\} + DD'$

have $set\_mset\ DDa \subseteq N$
unfolding *DDa_def* **using** *dd_subs_n dd'_subs_n*

```
      by (meson contra_subsetD in_diffD subsetI union_iff)
    moreover have ∀ I. I ⊨m DDa + CC ⟶ I ⊨ E
      using dd'_imp_da ddcc_imp_e da_in_dd unfolding DDa_def true_cls_mset_def
      by (metis in_remove1_mset_neq union_iff)
    moreover have ∀ D'. D' ∈# DDa ⟶ D' < D
      using dd_lt_d dd'_lt_da da_in_dd unfolding DDa_def
      by (metis insert_DiffM2 order.strict_trans union_iff)
    moreover have DDa < DD
      unfolding DDa_def
      by (meson da_in_dd dd'_lt_da mset_lt_single_right_iff single_subset_iff union_le_diff_plus)
    ultimately show False
      using d_min unfolding less_eq_multiset_def by (auto intro!: antisym)
  qed
  then show ?thesis
    using dd_subs_n ddcc_imp_e dd_lt_d by auto
qed

lemma Rf_imp_ex_non_Rf:
  assumes C ∈ Rf N
  shows ∃ CC. set_mset CC ⊆ N − Rf N ∧ (∀ I. I ⊨m CC ⟶ I ⊨ C) ∧ (∀ C'. C' ∈# CC ⟶ C' < C)
  using assms by (auto simp: Rf_def intro: wlog_non_Rf[of _ {#}, simplified])

lemma Rf_subs_Rf_diff_Rf: Rf N ⊆ Rf (N − Rf N)
proof
  fix C
  assume c_rf: C ∈ Rf N
  then obtain CC where
    cc_subs: set_mset CC ⊆ N − Rf N and
    cc_imp_c: ∀ I. I ⊨m CC ⟶ I ⊨ C and
    cc_lt_c: ∀ C'. C' ∈# CC ⟶ C' < C
    using Rf_imp_ex_non_Rf by blast
  have ∀ D. D ∈# CC ⟶ D ∉ Rf N
    using cc_subs by (simp add: subset_iff)
  then have cc_nr:
    ⋀C DD. C ∈# CC ⟹ set_mset DD ⊆ N ⟹ ∀ I. I ⊨m DD ⟶ I ⊨ C ⟹ ∃ D. D ∈# DD ∧ ~ D < C
      unfolding Rf_def by auto metis
  have set_mset CC ⊆ N
    using cc_subs by auto
  then have set_mset CC ⊆
    N − {C. ∃ DD. set_mset DD ⊆ N ∧ (∀ I. I ⊨m DD ⟶ I ⊨ C) ∧ (∀ D. D ∈# DD ⟶ D < C)}
    using cc_nr by auto
  then show C ∈ Rf (N − Rf N)
    using cc_imp_c cc_lt_c unfolding Rf_def by auto
qed

lemma Rf_eq_Rf_diff_Rf: Rf N = Rf (N − Rf N)
  by (metis Diff_subset Rf_mono Rf_subs_Rf_diff_Rf subset_antisym)
```

The following results correspond to Lemma 4.6.

```
lemma Ri_mono: N ⊆ N' ⟹ Ri N ⊆ Ri N'
  unfolding Ri_def by auto

lemma Ri_subs_Ri_diff_Rf: Ri N ⊆ Ri (N − Rf N)
proof
  fix γ
  assume γ_ri: γ ∈ Ri N
  then obtain CC D E where γ: γ = Infer CC D E
    by (cases γ)
  have cc: CC = side_prems_of γ and d: D = main_prem_of γ and e: E = concl_of γ
    unfolding γ by simp_all
  obtain DD where
    set_mset DD ⊆ N and ∀ I. I ⊨m DD + CC ⟶ I ⊨ E and ∀ C. C ∈# DD ⟶ C < D
    using γ_ri unfolding Ri_def cc d e by blast
```

```
  then obtain DD' where
    set_mset DD' ⊆ N − Rf N and ∀ I. I ⊨m DD' + CC ⟶ I ⊨ E and ∀ D'. D' ∈# DD' ⟶ D' < D
    using wlog_non_Rf by atomize_elim blast
  then show γ ∈ Ri (N − Rf N)
    using γ_ri unfolding Ri_def d cc e by blast
qed

lemma Ri_eq_Ri_diff_Rf: Ri N = Ri (N − Rf N)
  by (metis Diff_subset Ri_mono Ri_subs_Ri_diff_Rf subset_antisym)

lemma Ri_subset_Γ: Ri N ⊆ Γ
  unfolding Ri_def by blast

lemma Rf_indep: N' ⊆ Rf N ⟹ Rf N ⊆ Rf (N − N')
  by (metis Diff_cancel Diff_eq_empty_iff Diff_mono Rf_eq_Rf_diff_Rf Rf_mono)

lemma Ri_indep: N' ⊆ Rf N ⟹ Ri N ⊆ Ri (N − N')
  by (metis Diff_mono Ri_eq_Ri_diff_Rf Ri_mono order_refl)

lemma Rf_model:
  assumes I ⊨s N − Rf N
  shows I ⊨s N
proof −
  have I ⊨s Rf (N − Rf N)
    unfolding true_clss_def
    by (subst Rf_def, simp add: true_cls_mset_def, metis assms subset_eq true_clss_def)
  then have I ⊨s Rf N
    using Rf_subs_Rf_diff_Rf true_clss_mono by blast
  then show ?thesis
    using assms by (metis Un_Diff_cancel true_clss_union)
qed

lemma Rf_sat: satisfiable (N − Rf N) ⟹ satisfiable N
  by (metis Rf_model)
```

The following corresponds to Theorem 4.7:

```
sublocale redundancy_criterion Γ Rf Ri
  by unfold_locales (rule Ri_subset_Γ, (elim Rf_mono Ri_mono Rf_indep Ri_indep Rf_sat)+)

end

locale standard_redundancy_criterion_reductive =
  standard_redundancy_criterion + reductive_inference_system
begin
```

The following corresponds to Theorem 4.8:

```
lemma Ri_effective:
  assumes
    in_γ: γ ∈ Γ and
    concl_of_in_n_un_rf_n: concl_of γ ∈ N ∪ Rf N
  shows γ ∈ Ri N
proof −
  obtain CC D E where
    γ: γ = Infer CC D E
    by (cases γ)
  then have cc: CC = side_prems_of γ and d: D = main_prem_of γ and e: E = concl_of γ
    unfolding γ by simp_all
  note e_in_n_un_rf_n = concl_of_in_n_un_rf_n[folded e]

  {
    assume E ∈ N
    moreover have E < D
      using Γ_reductive e d in_γ by auto
```

```
    ultimately have
      set_mset {#E#} ⊆ N and ∀ I. I ⊨m {#E#} + CC ⟶ I ⊨ E and ∀ D'. D' ∈# {#E#} ⟶ D' < D
      by simp_all
    then have redundant_infer N γ
      using cc d e by blast
  }
  moreover
  {
    assume E ∈ Rf N
    then obtain DD where
      dd_sset: set_mset DD ⊆ N and
      dd_imp_e: ∀ I. I ⊨m DD ⟶ I ⊨ E and
      dd_lt_e: ∀ C'. C' ∈# DD ⟶ C' < E
      unfolding Rf_def by blast
    from dd_lt_e have ∀ Da. Da ∈# DD ⟶ Da < D
      using d e in_γ Γ_reductive less_trans by blast
    then have redundant_infer N γ
      using dd_sset dd_imp_e cc d e by blast
  }
  ultimately show γ ∈ Ri N
    using in_γ e_in_n_un_rf_n unfolding Ri_def by blast
qed

sublocale effective_redundancy_criterion Γ Rf Ri
  unfolding effective_redundancy_criterion_def
  by (intro conjI redundancy_criterion_axioms, unfold_locales, rule Ri_effective)

lemma contradiction_Rf: {#} ∈ N ⟹ Ri N = Γ
  unfolding Ri_def using Γ_reductive le_multiset_empty_right
  by (force intro: exI[of _ {#{#}#}] le_multiset_empty_left)

end

locale standard_redundancy_criterion_counterex_reducing =
  standard_redundancy_criterion + counterex_reducing_inference_system
begin
```

The following result corresponds to Theorem 4.9.

```
lemma saturated_upto_complete_if:
  assumes
    satur: saturated_upto N and
    unsat: ¬ satisfiable N
  shows {#} ∈ N
proof (rule ccontr)
  assume ec_ni_n: {#} ∉ N

  define M where
    M = N − Rf N

  have ec_ni_m: {#} ∉ M
    unfolding M_def using ec_ni_n by fast

  have I_of M ⊨s M
  proof (rule ccontr)
    assume ¬ I_of M ⊨s M
    then obtain D where
      d_in_m: D ∈ M and
      d_cex: ¬ I_of M ⊨ D and
      d_min: ⋀C. C ∈ M ⟹ C < D ⟹ I_of M ⊨ C
      using ex_min_counterex by meson
    then obtain γ CC E where
      γ: γ = Infer CC D E and
      cc_subs_m: set_mset CC ⊆ M and
```

cc_true: *I_of M* $\models$*m CC* **and**
γ_in: $\gamma \in \Gamma$ **and**
e_cex: $\neg$ *I_of M* $\models$ *E* **and**
e_lt_d: $E < D$
using *Γ_counterex_reducing*[*OF ec_ni_m*] *not_less* **by** *metis*
have *cc*: CC = *side_prems_of* γ **and** *d*: D = *main_prem_of* γ **and** *e*: E = *concl_of* γ
unfolding γ **by** *simp_all*
have $\gamma \in$ *Ri N*
by (*rule set_mp*[*OF satur*[*unfolded saturated_upto_def inferences_from_def infer_from_def*]])
(*simp add*: *γ_in d_in_m cc_subs_m cc*[*symmetric*] *d*[*symmetric*] *M_def*[*symmetric*])
then have $\gamma \in$ *Ri M*
unfolding *M_def* **using** *Ri_indep* **by** *fast*
then obtain *DD* **where**
dd_subs_m: *set_mset DD* $\subseteq$ *M* **and**
dd_cc_imp_d: $\forall$ *I*. *I* $\models$*m DD* + *CC* $\longrightarrow$ *I* $\models$ *E* **and**
dd_lt_d: $\forall$ *C*. *C* $\in$# *DD* $\longrightarrow$ $C < D$
unfolding *Ri_def cc d e* **by** *blast*
from *dd_subs_m dd_lt_d* **have** *I_of M* $\models$*m DD*
using *d_min* **unfolding** *true_cls_mset_def* **by** (*metis contra_subsetD*)
then have *I_of M* $\models$ *E*
using *dd_cc_imp_d cc_true* **by** *auto*
then show *False*
using *e_cex* **by** *auto*
qed
then have *I_of M* $\models$*s N*
using *M_def Rf_model* **by** *blast*
then show *False*
using *unsat* **by** *blast*
qed

theorem *saturated_upto_complete*:
assumes *saturated_upto N*
shows $\neg$ *satisfiable N* $\longleftrightarrow$ {#} $\in$ *N*
using *assms saturated_upto_complete_if true_clss_def* **by** *auto*

end

end

14 First-Order Ordered Resolution Calculus with Selection

theory *FO_Ordered_Resolution*
imports *Abstract_Substitution Ordered_Ground_Resolution Standard_Redundancy*
begin

This material is based on Section 4.3 ("A Simple Resolution Prover for First-Order Clauses") of Bachmair and Ganzinger's chapter. Specifically, it formalizes the ordered resolution calculus for first-order standard clauses presented in Figure 4 and its related lemmas and theorems, including soundness and Lemma 4.12 (the lifting lemma).

The following corresponds to pages 41–42 of Section 4.3, until Figure 5 and its explanation.

locale *FO_resolution* = *mgu subst_atm id_subst comp_subst atm_of_atms renamings_apart mgu*
for
subst_atm :: *'a* :: *wellorder* $\Rightarrow$ *'s* $\Rightarrow$ *'a* **and**
id_subst :: *'s* **and**
comp_subst :: *'s* $\Rightarrow$ *'s* $\Rightarrow$ *'s* **and**
renamings_apart :: *'a literal multiset list* $\Rightarrow$ *'s list* **and**
atm_of_atms :: *'a list* $\Rightarrow$ *'a* **and**
mgu :: *'a set set* $\Rightarrow$ *'s option* +
fixes
less_atm :: *'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool*
assumes
less_atm_stable: *less_atm A B* $\Longrightarrow$ *less_atm* ($A \cdot a\ \sigma$) ($B \cdot a\ \sigma$)

begin

14.1 Library

lemma *Bex_cartesian_product*: $(\exists xy \in A \times B.\ P\ xy) \equiv (\exists x \in A.\ \exists y \in B.\ P\ (x,\ y))$
  by *simp*

lemma *length_sorted_list_of_multiset*[*simp*]: *length* (*sorted_list_of_multiset A*) = *size A*
  by (*metis mset_sorted_list_of_multiset size_mset*)

lemma *eql_map_neg_lit_eql_atm*:
  assumes $map\ (\lambda L.\ L \cdot l\ \eta)\ (map\ Neg\ As') = map\ Neg\ As$
  shows $As' \cdot al\ \eta = As$
  using *assms* **by** (*induction As' arbitrary*: *As*) *auto*

lemma *instance_list*:
  assumes $negs\ (mset\ As) = SDA' \cdot \eta$
  shows $\exists As'.\ negs\ (mset\ As') = SDA' \wedge As' \cdot al\ \eta = As$
proof –
  from *assms* **have** *negL*: $\forall L \in\#\ SDA'.\ is\_neg\ L$
    using *Melem_subst_cls subst_lit_in_negs_is_neg* **by** *metis*

  from *assms* **have** $\{\#L \cdot l\ \eta.\ L \in\#\ SDA'\#\} = mset\ (map\ Neg\ As)$
    using *subst_cls_def* **by** *auto*
  then have $\exists NAs'.\ map\ (\lambda L.\ L \cdot l\ \eta)\ NAs' = map\ Neg\ As \wedge mset\ NAs' = SDA'$
    using *image_mset_of_subset_list*[*of* $\lambda L.\ L \cdot l\ \eta\ SDA'$ *map Neg As*] **by** *auto*
  then obtain As' **where** As'_p:
    $map\ (\lambda L.\ L \cdot l\ \eta)\ (map\ Neg\ As') = map\ Neg\ As \wedge mset\ (map\ Neg\ As') = SDA'$
    by (*metis* (*no_types*, *lifting*) *Neg_atm_of_iff negL ex_map_conv set_mset_mset*)

  have $negs\ (mset\ As') = SDA'$
    using As'_p **by** *auto*
  moreover have $map\ (\lambda L.\ L \cdot l\ \eta)\ (map\ Neg\ As') = map\ Neg\ As$
    using As'_p **by** *auto*
  then have $As' \cdot al\ \eta = As$
    using *eql_map_neg_lit_eql_atm* **by** *auto*
  ultimately show *?thesis*
    by *blast*
qed

14.2 First-Order Logic

inductive *true_fo_cls* :: *'a interp* ⇒ *'a clause* ⇒ *bool* (**infix** ⊨*fo 50*) **where**
  true_fo_cls: $(\bigwedge\sigma.\ is\_ground\_subst\ \sigma \Longrightarrow I \models C \cdot \sigma) \Longrightarrow I \models fo\ C$

lemma *true_fo_cls_inst*: $I \models fo\ C \Longrightarrow is\_ground\_subst\ \sigma \Longrightarrow I \models C \cdot \sigma$
  by (*rule true_fo_cls.induct*)

inductive *true_fo_cls_mset* :: *'a interp* ⇒ *'a clause multiset* ⇒ *bool* (**infix** ⊨*fom 50*) **where**
  true_fo_cls_mset: $(\bigwedge\sigma.\ is\_ground\_subst\ \sigma \Longrightarrow I \models m\ CC \cdot cm\ \sigma) \Longrightarrow I \models fom\ CC$

lemma *true_fo_cls_mset_inst*: $I \models fom\ C \Longrightarrow is\_ground\_subst\ \sigma \Longrightarrow I \models m\ C \cdot cm\ \sigma$
  by (*rule true_fo_cls_mset.induct*)

lemma *true_fo_cls_mset_def2*: $I \models fom\ CC \longleftrightarrow (\forall C \in\#\ CC.\ I \models fo\ C)$
  unfolding *true_fo_cls_mset.simps true_fo_cls.simps true_cls_mset_def* **by** *force*

context
  fixes *S* :: *'a clause* ⇒ *'a clause*
begin

14.3 Calculus

The following corresponds to Figure 4.

definition *maximal_wrt* :: $'a \Rightarrow 'a$ *literal multiset* $\Rightarrow$ *bool* **where**
maximal_wrt A $C \longleftrightarrow (\forall B \in$ *atms_of* $C.\ \neg$ *less_atm* A $B)$

definition *strictly_maximal_wrt* :: $'a \Rightarrow 'a$ *literal multiset* $\Rightarrow$ *bool* **where**
strictly_maximal_wrt A $C \equiv \forall B \in$ *atms_of* $C.\ A \neq B \wedge \neg$ *less_atm* A B

lemma *strictly_maximal_wrt_maximal_wrt*: *strictly_maximal_wrt* A $C \Longrightarrow$ *maximal_wrt* A C
unfolding *maximal_wrt_def strictly_maximal_wrt_def* **by** *auto*

inductive *eligible* :: $'s \Rightarrow 'a$ *list* $\Rightarrow 'a$ *clause* $\Rightarrow$ *bool* **where**
eligible:
$S\ DA = negs\ (mset\ As) \vee S\ DA = \{\#\} \wedge length\ As = 1 \wedge maximal\_wrt\ (As\ !\ 0 \cdot a\ \sigma)\ (DA \cdot \sigma) \Longrightarrow$
$eligible\ \sigma\ As\ DA$

inductive
ord_resolve
:: $'a$ *clause list* $\Rightarrow 'a$ *clause* $\Rightarrow 'a$ *multiset list* $\Rightarrow 'a$ *list* $\Rightarrow 's \Rightarrow 'a$ *clause* $\Rightarrow$ *bool*
where
ord_resolve:
$length\ CAs = n \Longrightarrow$
$length\ Cs = n \Longrightarrow$
$length\ AAs = n \Longrightarrow$
$length\ As = n \Longrightarrow$
$n \neq 0 \Longrightarrow$
$(\forall i < n.\ CAs\ !\ i = Cs\ !\ i + poss\ (AAs\ !\ i)) \Longrightarrow$
$(\forall i < n.\ AAs\ !\ i \neq \{\#\}) \Longrightarrow$
$Some\ \sigma = mgu\ (set\_mset\ `\ set\ (map2\ add\_mset\ As\ AAs)) \Longrightarrow$
$eligible\ \sigma\ As\ (D + negs\ (mset\ As)) \Longrightarrow$
$(\forall i < n.\ strictly\_maximal\_wrt\ (As\ !\ i \cdot a\ \sigma)\ (Cs\ !\ i \cdot \sigma)) \Longrightarrow$
$(\forall i < n.\ S\ (CAs\ !\ i) = \{\#\}) \Longrightarrow$
$ord\_resolve\ CAs\ (D + negs\ (mset\ As))\ AAs\ As\ \sigma\ (((\bigcup\#\ mset\ Cs) + D) \cdot \sigma)$

inductive
ord_resolve_rename
:: $'a$ *clause list* $\Rightarrow 'a$ *clause* $\Rightarrow 'a$ *multiset list* $\Rightarrow 'a$ *list* $\Rightarrow 's \Rightarrow 'a$ *clause* $\Rightarrow$ *bool*
where
ord_resolve_rename:
$length\ CAs = n \Longrightarrow$
$length\ AAs = n \Longrightarrow$
$length\ As = n \Longrightarrow$
$(\forall i < n.\ poss\ (AAs\ !\ i) \subseteq\#\ CAs\ !\ i) \Longrightarrow$
$negs\ (mset\ As) \subseteq\#\ DA \Longrightarrow$
$\varrho = hd\ (renamings\_apart\ (DA\ \#\ CAs)) \Longrightarrow$
$\varrho s = tl\ (renamings\_apart\ (DA\ \#\ CAs)) \Longrightarrow$
$ord\_resolve\ (CAs \cdot\cdot cl\ \varrho s)\ (DA \cdot \varrho)\ (AAs \cdot\cdot aml\ \varrho s)\ (As \cdot al\ \varrho)\ \sigma\ E \Longrightarrow$
$ord\_resolve\_rename\ CAs\ DA\ AAs\ As\ \sigma\ E$

lemma *ord_resolve_empty_main_prem*: $\neg$ *ord_resolve* $Cs\ \{\#\}\ AAs\ As\ \sigma\ E$
by (*simp add*: *ord_resolve.simps*)

lemma *ord_resolve_rename_empty_main_prem*: $\neg$ *ord_resolve_rename* $Cs\ \{\#\}\ AAs\ As\ \sigma\ E$
by (*simp add*: *ord_resolve_empty_main_prem ord_resolve_rename.simps*)

14.4 Soundness

Soundness is not discussed in the chapter, but it is an important property. The following lemma is used to prove soundness. It is also used to prove Lemma 4.10, which is used to prove completeness.

lemma *ord_resolve_ground_inst_sound*:
assumes

res_e: *ord_resolve CAs DA AAs As σ E* **and**
cc_inst_true: *I* ⊨*m mset CAs* ·*cm σ* ·*cm η* **and**
d_inst_true: *I* ⊨ *DA* · *σ* · *η* **and**
ground_subst_η: *is_ground_subst η*
shows *I* ⊨ *E* · *η*
using *res_e*
proof (*cases rule*: *ord_resolve.cases*)
case (*ord_resolve n Cs D*)
note *da* = *this*(*1*) **and** *e* = *this*(*2*) **and** *cas_len* = *this*(*3*) **and** *cs_len* = *this*(*4*) **and**
aas_len = *this*(*5*) **and** *as_len* = *this*(*6*) **and** *cas* = *this*(*8*) **and** *mgu* = *this*(*10*) **and**
len = *this*(*1*)

have *len*: *length CAs* = *length As*
using *as_len cas_len* **by** *auto*
have *is_ground_subst* (*σ* ⊙ *η*)
using *ground_subst_η* **by** (*rule is_ground_comp_subst*)
then have *cc_true*: *I* ⊨*m mset CAs* ·*cm σ* ·*cm η* **and** *d_true*: *I* ⊨ *DA* · *σ* · *η*
using *cc_inst_true d_inst_true* **by** *auto*

from *mgu* **have** *unif*: ∀ *i* < *n*. ∀ *A*∈#*AAs* ! *i*. *A* ·*a σ* = *As* ! *i* ·*a σ*
using *mgu_unifier as_len aas_len* **by** *blast*

show *I* ⊨ *E* · *η*
proof (*cases* ∀ *A* ∈ *set As*. *A* ·*a σ* ·*a η* ∈ *I*)
case *True*
then have ¬ *I* ⊨ *negs* (*mset As*) · *σ* · *η*
unfolding *true_cls_def*[*of I*] **by** *auto*
then have *I* ⊨ *D* · *σ* · *η*
using *d_true da* **by** *auto*
then show *?thesis*
unfolding *e* **by** *auto*
next
case *False*
then obtain *i* **where** *a_in_aa*: *i* < *length CAs* **and** *a_false*: (*As* ! *i*) ·*a σ* ·*a η* ∉ *I*
using *da len* **by** (*metis in_set_conv_nth*)
define *C* **where** *C* ≡ *Cs* ! *i*
define *BB* **where** *BB* ≡ *AAs* ! *i*
have *c_cf′*: *C* ⊆# ⋃# *mset CAs*
unfolding *C_def* **using** *a_in_aa cas cas_len*
by (*metis less_subset_eq_Union_mset mset_subset_eq_add_left subset_mset.order.trans*)
have *c_in_cc*: *C* + *poss BB* ∈# *mset CAs*
using *C_def BB_def a_in_aa cas_len in_set_conv_nth cas* **by** *fastforce*
{
fix *B*
assume *B* ∈# *BB*
then have *B* ·*a σ* = (*As* ! *i*) ·*a σ*
using *unif a_in_aa cas_len* **unfolding** *BB_def* **by** *auto*
}
then have ¬ *I* ⊨ *poss BB* · *σ* · *η*
using *a_false* **by** (*auto simp*: *true_cls_def*)
moreover have *I* ⊨ (*C* + *poss BB*) · *σ* · *η*
using *c_in_cc cc_true true_cls_mset_true_cls*[*of I mset CAs* ·*cm σ* ·*cm η*] **by** *force*
ultimately have *I* ⊨ *C* · *σ* · *η*
by *simp*
then show *?thesis*
unfolding *e subst_cls_union* **using** *c_cf′ C_def a_in_aa cas_len cs_len*
by (*metis* (*no_types*, *lifting*) *mset_subset_eq_add_left nth_mem_mset set_mset_mono sum_mset.remove true_cls_mono subst_cls_mono*)
qed
qed

lemma *ord_resolve_sound*:
assumes

```
    res_e: ord_resolve CAs DA AAs As σ E and
    cc_d_true: I ⊨fom mset CAs + {#DA#}
  shows I ⊨fo E
proof (rule true_fo_cls, use res_e in ‹cases rule: ord_resolve.cases›)
  fix η
  assume ground_subst_η: is_ground_subst η
  case (ord_resolve n Cs D)
  note da = this(1) and e = this(2) and cas_len = this(3) and cs_len = this(4)
    and aas_len = this(5) and as_len = this(6) and cas = this(8) and mgu = this(10)

  have is_ground_subst (σ ⊙ η)
    using ground_subst_η by (rule is_ground_comp_subst)
  then have cas_true: I ⊨m mset CAs ·cm σ ·cm η and da_true: I ⊨ DA · σ · η
    using true_fo_cls_mset_inst[OF cc_d_true, of σ ⊙ η] by auto
  show I ⊨ E · η
    using ord_resolve_ground_inst_sound[OF res_e cas_true da_true] ground_subst_η by auto
qed

lemma subst_sound: I ⊨fo C ⟹ I ⊨fo (C · ϱ)
  by (metis is_ground_comp_subst subst_cls_comp_subst true_fo_cls true_fo_cls_inst)

lemma true_fo_cls_mset_true_fo_cls: I ⊨fom CC ⟹ C ∈# CC ⟹ I ⊨fo C
  using true_fo_cls_mset_def2 by auto

lemma subst_sound_scl:
  assumes
    len: length P = length CAs and
    true_cas: I ⊨fom mset CAs
  shows I ⊨fom mset (CAs ··cl P)
proof −
  from true_cas have ∀ CA. CA∈# mset CAs ⟶ I ⊨fo CA
    using true_fo_cls_mset_true_fo_cls by auto
  then have ∀ i < length CAs. I ⊨fo CAs ! i
    using in_set_conv_nth by auto
  then have true_cp: ∀ i < length CAs. I ⊨fo CAs ! i · P ! i
    using subst_sound len by auto

  {
    fix CA
    assume CA ∈# mset (CAs ··cl P)
    then obtain i where
      i_x: i < length (CAs ··cl P) CA = (CAs ··cl P) ! i
      by (metis in_mset_conv_nth)
    then have I ⊨fo CA
      using true_cp unfolding subst_cls_lists_def by (simp add: len)
  }
  then show ?thesis
    unfolding true_fo_cls_mset_def2 by auto
qed
```

This is a lemma needed to prove Lemma 4.11.

```
lemma ord_resolve_rename_ground_inst_sound:
  assumes
    ord_resolve_rename CAs DA AAs As σ E and
    ϱs = tl (renamings_apart (DA # CAs)) and
    ϱ = hd (renamings_apart (DA # CAs)) and
    I ⊨m (mset (CAs ··cl ϱs)) ·cm σ ·cm η and
    I ⊨ DA · ϱ · σ · η and
    is_ground_subst η
  shows I ⊨ E · η
  using assms by (cases rule: ord_resolve_rename.cases) (fast intro: ord_resolve_ground_inst_sound)

lemma ord_resolve_rename_sound:
```

assumes
res_e: *ord_resolve_rename CAs DA AAs As* σ *E* **and**
cc_d_true: $I \models$*fom* (*mset CAs*) + {#*DA*#}
shows $I \models$*fo E*
using *res_e*
proof (*cases rule*: *ord_resolve_rename.cases*)
case (*ord_resolve_rename n* ϱ ϱ*s*)
note ϱ*s* = *this*(*7*) **and** *res* = *this*(*8*)
have *len*: *length* ϱ*s* = *length CAs*
using ϱ*s renames_apart* **by** *auto*
have $I \models$*fom mset* (*CAs* $\cdot\cdot$*cl* ϱ*s*) + {#*DA* $\cdot$ ϱ#}
using *subst_sound_scl*[*OF len*, *of I*] *subst_sound cc_d_true* **by** (*simp add*: *true_fo_cls_mset_def2*)
then show $I \models$*fo E*
using *ord_resolve_sound*[*OF res*] **by** *simp*
qed

14.5 Other Basic Properties

lemma *ord_resolve_unique*:
assumes
ord_resolve CAs DA AAs As σ *E* **and**
ord_resolve CAs DA AAs As σ' E'
shows $\sigma = \sigma' \wedge E = E'$
using *assms*
proof (*cases rule*: *ord_resolve.cases*[*case_product ord_resolve.cases*], *intro conjI*)
case (*ord_resolve_ord_resolve CAs n Cs AAs As* σ'' *DA* CAs' n' Cs' AAs' As' σ''' DA')
note *res* = *this*(*1*−*17*) **and** res' = *this*(*18*−*34*)

show σ: $\sigma = \sigma'$
using *res*(*3*−*5*,*14*) res'(*3*−*5*,*14*) **by** (*metis option.inject*)

have *Cs* = Cs'
using *res*(*1*,*3*,*7*,*8*,*12*) res'(*1*,*3*,*7*,*8*,*12*) **by** (*metis add_right_imp_eq nth_equalityI*)
moreover have *DA* = DA'
using *res*(*2*,*4*) res'(*2*,*4*) **by** *fastforce*
ultimately show $E = E'$
using *res*(*5*,*6*) res'(*5*,*6*) σ **by** *blast*
qed

lemma *ord_resolve_rename_unique*:
assumes
ord_resolve_rename CAs DA AAs As σ *E* **and**
ord_resolve_rename CAs DA AAs As σ' E'
shows $\sigma = \sigma' \wedge E = E'$
using *assms* **unfolding** *ord_resolve_rename.simps* **using** *ord_resolve_unique* **by** *meson*

lemma *ord_resolve_max_side_prems*: *ord_resolve CAs DA AAs As* σ *E* $\Longrightarrow$ *length CAs* $\leq$ *size DA*
by (*auto elim*!: *ord_resolve.cases*)

lemma *ord_resolve_rename_max_side_prems*:
ord_resolve_rename CAs DA AAs As σ *E* $\Longrightarrow$ *length CAs* $\leq$ *size DA*
by (*elim ord_resolve_rename.cases*, *drule ord_resolve_max_side_prems*, *simp add*: *renames_apart*)

14.6 Inference System

definition *ord_FO_Γ* :: *'a inference set* **where**
ord_FO_Γ = {*Infer* (*mset CAs*) *DA E* | *CAs DA AAs As* σ *E*. *ord_resolve_rename CAs DA AAs As* σ *E*}

interpretation *ord_FO_resolution*: *inference_system ord_FO_Γ* **.**

lemma *exists_compose*: $\exists x.\ P\ (f\ x) \Longrightarrow \exists y.\ P\ y$
by *meson*

lemma *finite_ord_FO_resolution_inferences_between*:

```
  assumes fin_cc: finite CC
  shows finite (ord_FO_resolution.inferences_between CC C)
proof -
  let ?CCC = CC ∪ {C}

  define all_AA where all_AA = (⋃D ∈ ?CCC. atms_of D)
  define max_ary where max_ary = Max (size ` ?CCC)
  define CAS where CAS = {CAs. CAs ∈ lists ?CCC ∧ length CAs ≤ max_ary}
  define AS where AS = {As. As ∈ lists all_AA ∧ length As ≤ max_ary}
  define AAS where AAS = {AAs. AAs ∈ lists (mset ` AS) ∧ length AAs ≤ max_ary}

  note defs = all_AA_def max_ary_def CAS_def AS_def AAS_def

  let ?infer_of =
    λCAs DA AAs As. Infer (mset CAs) DA (THE E. ∃σ. ord_resolve_rename CAs DA AAs As σ E)

  let ?Z = {γ | CAs DA AAs As σ E γ. γ = Infer (mset CAs) DA E
    ∧ ord_resolve_rename CAs DA AAs As σ E ∧ infer_from ?CCC γ ∧ C ∈# prems_of γ}
  let ?Y = {Infer (mset CAs) DA E | CAs DA AAs As σ E.
    ord_resolve_rename CAs DA AAs As σ E ∧ set CAs ∪ {DA} ⊆ ?CCC}
  let ?X = {?infer_of CAs DA AAs As | CAs DA AAs As. CAs ∈ CAS ∧ DA ∈ ?CCC ∧ AAs ∈ AAS ∧ As ∈ AS}
  let ?W = CAS × ?CCC × AAS × AS

  have fin_w: finite ?W
    unfolding defs using fin_cc by (simp add: finite_lists_length_le lists_eq_set)

  have ?Z ⊆ ?Y
    by (force simp: infer_from_def)
  also have ... ⊆ ?X
  proof -
    {
      fix CAs DA AAs As σ E
      assume
        res_e: ord_resolve_rename CAs DA AAs As σ E and
        da_in: DA ∈ ?CCC and
        cas_sub: set CAs ⊆ ?CCC

      have E = (THE E. ∃σ. ord_resolve_rename CAs DA AAs As σ E)
        ∧ CAs ∈ CAS ∧ AAs ∈ AAS ∧ As ∈ AS (is ?e ∧ ?cas ∧ ?aas ∧ ?as)
      proof (intro conjI)
        show ?e
          using res_e ord_resolve_rename_unique by (blast intro: the_equality[symmetric])
      next
        show ?cas
          unfolding CAS_def max_ary_def using cas_sub
            ord_resolve_rename_max_side_prems[OF res_e] da_in fin_cc
          by (auto simp add: Max_ge_iff)
      next
        show ?aas
          using res_e
        proof (cases rule: ord_resolve_rename.cases)
          case (ord_resolve_rename n ϱ ϱs)
          note len_cas = this(1) and len_aas = this(2) and len_as = this(3) and
            aas_sub = this(4) and as_sub = this(5) and res_e' = this(8)

          show ?thesis
            unfolding AAS_def
          proof (clarify, intro conjI)
            show AAs ∈ lists (mset ` AS)
              unfolding AS_def image_def
            proof clarsimp
              fix AA
              assume AA ∈ set AAs
```

```
      then obtain i where
        i_lt: i < n and
        aa: AA = AAs ! i
        by (metis in_set_conv_nth len_aas)

      have casi_in: CAs ! i ∈ ?CCC
        using i_lt len_cas cas_sub nth_mem by blast

      have pos_aa_sub: poss AA ⊆# CAs ! i
        using aa aas_sub i_lt by blast
      then have set_mset AA ⊆ atms_of (CAs ! i)
        by (metis atms_of_poss lits_subseteq_imp_atms_subseteq set_mset_mono)
      also have aa_sub: ... ⊆ all_AA
        unfolding all_AA_def using casi_in by force
      finally have aa_sub: set_mset AA ⊆ all_AA
        .

      have size AA = size (poss AA)
        by simp
      also have ... ≤ size (CAs ! i)
        by (rule size_mset_mono[OF pos_aa_sub])
      also have ... ≤ max_ary
        unfolding max_ary_def using fin_cc casi_in by auto
      finally have sz_aa: size AA ≤ max_ary
        .

      let ?As' = sorted_list_of_multiset AA

      have ?As' ∈ lists all_AA
        using aa_sub by auto
      moreover have length ?As' ≤ max_ary
        using sz_aa by simp
      moreover have AA = mset ?As'
        by simp
      ultimately show ∃xa. xa ∈ lists all_AA ∧ length xa ≤ max_ary ∧ AA = mset xa
        by blast
    qed
  next
    have length AAs = length As
      unfolding len_aas len_as ..
    also have ... ≤ size DA
      using as_sub size_mset_mono by fastforce
    also have ... ≤ max_ary
      unfolding max_ary_def using fin_cc da_in by auto
    finally show length AAs ≤ max_ary
      .
  qed
 qed
next
 show ?as
  unfolding AS_def
 proof (clarify, intro conjI)
  have set As ⊆ atms_of DA
    using res_e[simplified ord_resolve_rename.simps]
    by (metis atms_of_negs lits_subseteq_imp_atms_subseteq set_mset_mono set_mset_mset)
  also have ... ⊆ all_AA
    unfolding all_AA_def using da_in by blast
  finally show As ∈ lists all_AA
    unfolding lists_eq_set by simp
 next
  have length As ≤ size DA
    using res_e[simplified ord_resolve_rename.simps]
      ord_resolve_rename_max_side_prems[OF res_e] by auto
```

also have *size DA* $\leq$ *max_ary*
unfolding *max_ary_def* **using** *fin_cc da_in* **by** *auto*
finally show *length As* $\leq$ *max_ary*
.
qed
qed
}
then show *?thesis*
by *simp fast*
qed
also have ... $\subseteq$ (λ(*CAs*, *DA*, *AAs*, *As*). *?infer_of CAs DA AAs As*) ' *?W*
unfolding *image_def Bex_cartesian_product* **by** *fast*
finally show *?thesis*
unfolding *inference_system.inferences_between_def ord_FO_Γ_def mem_Collect_eq*
by (*fast intro*: *rev_finite_subset*[*OF finite_imageI*[*OF fin_w*]])
qed

lemma *ord_FO_resolution_inferences_between_empty_empty*:
ord_FO_resolution.inferences_between {} {#} = {}
unfolding *ord_FO_resolution.inferences_between_def inference_system.inferences_between_def*
infer_from_def ord_FO_Γ_def
using *ord_resolve_rename_empty_main_prem* **by** *auto*

14.7 Lifting

The following corresponds to the passage between Lemmas 4.11 and 4.12.

context
fixes *M* :: *'a clause set*
assumes *select*: *selection S*
begin

interpretation *selection*
by (*rule select*)

definition *S_M* :: *'a literal multiset* $\Rightarrow$ *'a literal multiset* **where**
S_M C =
(*if C* $\in$ *grounding_of_clss M then*
(*SOME C'*. $\exists$ *D* σ. *D* $\in$ *M* $\wedge$ *C* = *D* $\cdot$ σ $\wedge$ *C'* = *S D* $\cdot$ σ $\wedge$ *is_ground_subst* σ)
else
S C)

lemma *S_M_grounding_of_clss*:
assumes *C* $\in$ *grounding_of_clss M*
obtains *D* σ **where**
D $\in$ *M* $\wedge$ *C* = *D* $\cdot$ σ $\wedge$ *S_M C* = *S D* $\cdot$ σ $\wedge$ *is_ground_subst* σ
proof (*atomize_elim*, *unfold S_M_def eqTrueI*[*OF assms*] *if_True*, *rule someI_ex*)
from *assms* **show** $\exists$ *C' D* σ. *D* $\in$ *M* $\wedge$ *C* = *D* $\cdot$ σ $\wedge$ *C'* = *S D* $\cdot$ σ $\wedge$ *is_ground_subst* σ
by (*auto simp*: *grounding_of_clss_def grounding_of_cls_def*)
qed

lemma *S_M_not_grounding_of_clss*: *C* $\notin$ *grounding_of_clss M* $\Longrightarrow$ *S_M C* = *S C*
unfolding *S_M_def* **by** *simp*

lemma *S_M_selects_subseteq*: *S_M C* $\subseteq$# *C*
by (*metis S_M_grounding_of_clss S_M_not_grounding_of_clss S_selects_subseteq subst_cls_mono_mset*)

lemma *S_M_selects_neg_lits*: *L* $\in$# *S_M C* $\Longrightarrow$ *is_neg L*
by (*metis Melem_subst_cls S_M_grounding_of_clss S_M_not_grounding_of_clss S_selects_neg_lits*
subst_lit_is_neg)

end

end

The following corresponds to Lemma 4.12:

lemma *map2_add_mset_map*:
  assumes *length AAs′* = *n* **and** *length As′* = *n*
  shows *map2 add_mset* (*As′* ·*al* η) (*AAs′* ·*aml* η) = *map2 add_mset As′ AAs′* ·*aml* η
  using *assms*
proof (*induction n arbitrary*: *AAs′ As′*)
  case (*Suc n*)
  then have *map2 add_mset* (*tl* (*As′* ·*al* η)) (*tl* (*AAs′* ·*aml* η)) = *map2 add_mset* (*tl As′*) (*tl AAs′*) ·*aml* η
    by *simp*
  moreover
  have *Succ*: *length* (*As′* ·*al* η) = *Suc n length* (*AAs′* ·*aml* η) = *Suc n*
    using *Suc*(*3*) *Suc*(*2*) **by** *auto*
  then have *length* (*tl* (*As′* ·*al* η)) = *n length* (*tl* (*AAs′* ·*aml* η)) = *n*
    by *auto*
  then have *length* (*map2 add_mset* (*tl* (*As′* ·*al* η)) (*tl* (*AAs′* ·*aml* η))) = *n*
    length (*map2 add_mset* (*tl As′*) (*tl AAs′*) ·*aml* η) = *n*
    using *Suc*(*2*,*3*) **by** *auto*
  ultimately have ∀ *i* < *n*. *tl* (*map2 add_mset* ((*As′* ·*al* η)) ((*AAs′* ·*aml* η))) ! *i* =
    tl (*map2 add_mset* (*As′*) (*AAs′*) ·*aml* η) ! *i*
    using *Suc*(*2*,*3*) *Succ* **by** (*simp add*: *map2_tl map_tl subst_atm_mset_list_def del*: *subst_atm_list_tl*)
  moreover have *nn*: *length* (*map2 add_mset* ((*As′* ·*al* η)) ((*AAs′* ·*aml* η))) = *Suc n*
    length (*map2 add_mset* (*As′*) (*AAs′*) ·*aml* η) = *Suc n*
    using *Succ Suc* **by** *auto*
  ultimately have ∀ *i*. *i* < *Suc n* ⟶ *i* > *0* ⟶
    map2 add_mset (*As′* ·*al* η) (*AAs′* ·*aml* η) ! *i* = (*map2 add_mset As′ AAs′* ·*aml* η) ! *i*
    by (*auto simp*: *subst_atm_mset_list_def gr0_conv_Suc subst_atm_mset_def*)
  moreover have *add_mset* (*hd As′* ·*a* η) (*hd AAs′* ·*am* η) = *add_mset* (*hd As′*) (*hd AAs′*) ·*am* η
    unfolding *subst_atm_mset_def* **by** *auto*
  then have (*map2 add_mset* (*As′* ·*al* η) (*AAs′* ·*aml* η)) ! *0* = (*map2 add_mset* (*As′*) (*AAs′*) ·*aml* η) ! *0*
    using *Suc* **by** (*simp add*: *Succ*(*2*) *subst_atm_mset_def*)
  ultimately have ∀ *i* < *Suc n*. (*map2 add_mset* (*As′* ·*al* η) (*AAs′* ·*aml* η)) ! *i* =
    (*map2 add_mset* (*As′*) (*AAs′*) ·*aml* η) ! *i*
    using *Suc* **by** *auto*
  then show *?case*
    using *nn list_eq_iff_nth_eq* **by** *metis*
qed *auto*

lemma *maximal_wrt_subst*: *maximal_wrt* (*A* ·*a* σ) (*C* · σ) ⟹ *maximal_wrt A C*
  unfolding *maximal_wrt_def* **using** *in_atms_of_subst less_atm_stable* **by** *blast*

lemma *strictly_maximal_wrt_subst*: *strictly_maximal_wrt* (*A* ·*a* σ) (*C* · σ) ⟹ *strictly_maximal_wrt A C*
  unfolding *strictly_maximal_wrt_def* **using** *in_atms_of_subst less_atm_stable* **by** *blast*

lemma *ground_resolvent_subset*:
  assumes
    gr_cas: *is_ground_cls_list CAs* **and**
    gr_da: *is_ground_cls DA* **and**
    res_e: *ord_resolve S CAs DA AAs As* σ *E*
  shows *E* ⊆# (⋃# *mset CAs*) + *DA*
  using *res_e*
proof (*cases rule*: *ord_resolve.cases*)
  case (*ord_resolve n Cs D*)
  note *da* = *this*(*1*) **and** *e* = *this*(*2*) **and** *cas_len* = *this*(*3*) **and** *cs_len* = *this*(*4*)
    and *aas_len* = *this*(*5*) **and** *as_len* = *this*(*6*) **and** *cas* = *this*(*8*) **and** *mgu* = *this*(*10*)
  then have *cs_sub_cas*: ⋃# *mset Cs* ⊆# ⋃# *mset CAs*
    using *subseteq_list_Union_mset cas_len cs_len* **by** *force*
  then have *cs_sub_cas*: ⋃# *mset Cs* ⊆# ⋃# *mset CAs*
    using *subseteq_list_Union_mset cas_len cs_len* **by** *force*
  then have *gr_cs*: *is_ground_cls_list Cs*
    using *gr_cas* **by** *simp*
  have *d_sub_da*: *D* ⊆# *DA*
    by (*simp add*: *da*)
  then have *gr_d*: *is_ground_cls D*

using *gr_da is_ground_cls_mono* **by** *auto*

have *is_ground_cls* $(\bigcup\# \ mset \ Cs + D)$
using *gr_cs gr_d* **by** *auto*
with *e* **have** $E = (\bigcup\# \ mset \ Cs + D)$
by *auto*
then show *?thesis*
using *cs_sub_cas d_sub_da* **by** (*auto simp*: *subset_mset.add_mono*)
qed

lemma *ord_resolve_obtain_clauses*:
assumes
res_e: *ord_resolve* (*S_M S M*) *CAs DA AAs As* σ *E* **and**
select: *selection S* **and**
grounding: $\{DA\} \cup set \ CAs \subseteq grounding\_of\_clss \ M$ **and**
n: *length CAs* = *n* **and**
d: *DA* = *D* + *negs* (*mset As*) **and**
c: $(\forall i < n. \ CAs \ ! \ i = Cs \ ! \ i + poss \ (AAs \ ! \ i))$ *length Cs* = *n length AAs* = *n*
obtains $DA'' \ \eta'' \ CAs'' \ \eta s'' \ As'' \ AAs'' \ D'' \ Cs''$ **where**
$length \ CAs'' = n$
$length \ \eta s'' = n$
$DA'' \in M$
$DA'' \cdot \eta'' = DA$
$S \ DA'' \cdot \eta'' = S\_M \ S \ M \ DA$
$\forall CA'' \in set \ CAs''. \ CA'' \in M$
$CAs'' \cdot\cdot cl \ \eta s'' = CAs$
$map \ S \ CAs'' \cdot\cdot cl \ \eta s'' = map \ (S\_M \ S \ M) \ CAs$
$is\_ground\_subst \ \eta''$
$is\_ground\_subst\_list \ \eta s''$
$As'' \ \cdot al \ \eta'' = As$
$AAs'' \cdot\cdot aml \ \eta s'' = AAs$
$length \ As'' = n$
$D'' \cdot \eta'' = D$
$DA'' = D'' + (negs \ (mset \ As''))$
$S\_M \ S \ M \ (D + negs \ (mset \ As)) \neq \{\#\} \Longrightarrow negs \ (mset \ As'') = S \ DA''$
$length \ Cs'' = n$
$Cs'' \cdot\cdot cl \ \eta s'' = Cs$
$\forall i < n. \ CAs'' \ ! \ i = Cs'' \ ! \ i + poss \ (AAs'' \ ! \ i)$
$length \ AAs'' = n$
using *res_e*
proof (*cases rule*: *ord_resolve.cases*)
case (*ord_resolve n_twin Cs_twins D_twin*)
note *da* = *this*(*1*) **and** *e* = *this*(*2*) **and** *cas* = *this*(*8*) **and** *mgu* = *this*(*10*) **and** *eligible* = *this*(*11*)
from *ord_resolve* **have** *n_twin* = *n D_twin* = *D*
using *n d* **by** *auto*
moreover have *Cs_twins* = *Cs*
using *c cas n calculation*(*1*) ‹*length Cs_twins* = *n_twin*› **by** (*auto simp add*: *nth_equalityI*)
ultimately
have *nz*: $n \neq 0$ **and** *cs_len*: *length Cs* = *n* **and** *aas_len*: *length AAs* = *n* **and** *as_len*: *length As* = *n*
and *da*: *DA* = *D* + *negs* (*mset As*) **and** *eligible*: *eligible* (*S_M S M*) σ *As* (*D* + *negs* (*mset As*))
and *cas*: $\forall i<n. \ CAs \ ! \ i = Cs \ ! \ i + poss \ (AAs \ ! \ i)$
using *ord_resolve* **by** *force+*

note *n* = ‹$n \neq 0$› ‹*length CAs* = *n*› ‹*length Cs* = *n*› ‹*length AAs* = *n*› ‹*length As* = *n*›

interpret *S*: *selection S* **by** (*rule select*)

— Obtain FO side premises
have $\forall CA \in set \ CAs. \ \exists CA'' \ \eta c''. \ CA'' \in M \land CA'' \cdot \eta c'' = CA \land S \ CA'' \cdot \eta c'' = S\_M \ S \ M \ CA \land is\_ground\_subst \ \eta c''$
using *grounding S_M_grounding_of_clss select* **by** (*metis* (*no_types*) *le_supE subset_iff*)
then have $\forall i < n. \ \exists CA'' \ \eta c''. \ CA'' \in M \land CA'' \cdot \eta c'' = (CAs \ ! \ i) \land S \ CA'' \cdot \eta c'' = S\_M \ S \ M \ (CAs \ ! \ i) \land is\_ground\_subst \ \eta c''$

using n **by** *force*
then obtain $\eta s''f$ $CAs''f$ **where** *f_p*:
$\forall i < n.\ CAs''f\ i \in M$
$\forall i < n.\ (CAs''f\ i) \cdot (\eta s''f\ i) = (CAs\ !\ i)$
$\forall i < n.\ S\ (CAs''f\ i)\ \cdot (\eta s''f\ i) = S\_M\ S\ M\ (CAs\ !\ i)$
$\forall i < n.$ *is_ground_subst* $(\eta s''f\ i)$
using n **by** (*metis* (*no_types*))

define $\eta s''$ **where**
$\eta s'' = map\ \eta s''f\ [0\ ..{<}n]$
define CAs'' **where**
$CAs'' = map\ CAs''f\ [0\ ..{<}n]$

have $length\ \eta s'' = n\ length\ CAs'' = n$
unfolding $\eta s''\_def$ CAs''_def **by** *auto*
note $n = \langle length\ \eta s'' = n\rangle\ \langle length\ CAs'' = n\rangle\ n$

— The properties we need of the FO side premises
have CAs''_in_M: $\forall CA'' \in set\ CAs''.\ CA'' \in M$
unfolding CAs''_def **using** *f_p*(*1*) **by** *auto*
have CAs''_to_CAs: $CAs''\ \cdot\cdot cl\ \eta s'' = CAs$
unfolding CAs''_def $\eta s''\_def$ **using** *f_p*(*2*) **by** (*auto simp*: n *intro*: *nth_equalityI*)
have $SCAs''\_to\_SMCAs$: $(map\ S\ CAs'')\ \cdot\cdot cl\ \eta s'' = map\ (S\_M\ S\ M)\ CAs$
unfolding CAs''_def $\eta s''\_def$ **using** *f_p*(*3*) n **by** (*force intro*: *nth_equalityI*)
have *sub_ground*: $\forall \eta c'' \in set\ \eta s''.$ *is_ground_subst* $\eta c''$
unfolding $\eta s''\_def$ **using** *f_p* n **by** *force*
then have *is_ground_subst_list* $\eta s''$
using n **unfolding** *is_ground_subst_list_def* **by** *auto*

— Split side premises CAs” into Cs” and AAs”
obtain AAs'' Cs'' **where** $AAs''\_Cs''\_p$:
$AAs''\ \cdot\cdot aml\ \eta s'' = AAs\ length\ Cs'' = n\ Cs''\ \cdot\cdot cl\ \eta s'' = Cs$
$\forall i < n.\ CAs''\ !\ i = Cs''\ !\ i + poss\ (AAs''\ !\ i)\ length\ AAs'' = n$
proof $-$
have $\forall i < n.\ \exists AA''.\ AA''\ \cdot am\ \eta s''\ !\ i = AAs\ !\ i \wedge poss\ AA'' \subseteq\#\ CAs''\ !\ i$
proof (*rule*, *rule*)
fix i
assume $i < n$
have $CAs''\ !\ i \cdot \eta s''\ !\ i = CAs\ !\ i$
using $\langle i < n\rangle\ \langle CAs''\ \cdot\cdot cl\ \eta s'' = CAs\rangle\ n$ **by** *force*
moreover have $poss\ (AAs\ !\ i) \subseteq\#\ CAs\ !i$
using $\langle i < n\rangle$ *cas* **by** *auto*
ultimately obtain $poss\_AA''$ **where**
nn: $poss\_AA'' \cdot \eta s''\ !\ i = poss\ (AAs\ !\ i) \wedge poss\_AA'' \subseteq\#\ CAs''\ !\ i$
using *cas image_mset_of_subset* **unfolding** *subst_cls_def* **by** *metis*
then have *l*: $\forall L \in\#\ poss\_AA''.$ *is_pos* L
unfolding *subst_cls_def* **by** (*metis Melem_subst_cls imageE literal.disc*(*1*)
literal.map_disc_iff set_image_mset subst_cls_def subst_lit_def)

define AA'' **where**
$AA'' = image\_mset\ atm\_of\ poss\_AA''$

have *na*: $poss\ AA'' = poss\_AA''$
using *l* **unfolding** AA''_def **by** *auto*
then have $AA''\ \cdot am\ \eta s''\ !\ i = AAs\ !\ i$
using *nn* **by** (*metis* (*mono_tags*) *literal.inject*(*1*) *multiset.inj_map_strong subst_cls_poss*)
moreover have $poss\ AA'' \subseteq\#\ CAs''\ !\ i$
using *na nn* **by** *auto*
ultimately show $\exists AA'.\ AA'\ \cdot am\ \eta s''\ !\ i = AAs\ !\ i \wedge poss\ AA' \subseteq\#\ CAs''\ !\ i$
by *blast*
qed
then obtain $AAs''f$ **where**
$AAs''f\_p$: $\forall i < n.\ AAs''f\ i\ \cdot am\ \eta s''\ !\ i = AAs\ !\ i \wedge (poss\ (AAs''f\ i)) \subseteq\#\ CAs''\ !\ i$

by *metis*

define AAs'' **where** $AAs'' = map\ AAs''f\ [0\ ..{<}n]$

then have $length\ AAs'' = n$
by *auto*
note $n = n$ $\langle length\ AAs'' = n\rangle$

from AAs''_def **have** $\forall i < n.\ AAs''\ !\ i\ \cdot am\ \eta s''\ !\ i = AAs\ !\ i$
using $AAs''f\_p$ **by** *auto*
then have AAs'_AAs: $AAs''\ \cdot\cdot aml\ \eta s'' = AAs$
using n **by** ($auto\ intro$: $nth\_equalityI$)

from AAs''_def **have** AAs''_in_CAs'': $\forall i < n.\ poss\ (AAs''\ !\ i) \subseteq\#\ CAs''\ !\ i$
using $AAs''f\_p$ **by** *auto*

define Cs'' **where**
$Cs'' = map2\ (op\ -)\ CAs''\ (map\ poss\ AAs'')$

have $length\ Cs'' = n$
using $Cs''\_def\ n$ **by** *auto*
note $n = n$ $\langle length\ Cs'' = n\rangle$

have $\forall i < n.\ CAs''\ !\ i = Cs''\ !\ i + poss\ (AAs''\ !\ i)$
using $AAs''\_in\_CAs''\ Cs''\_def\ n$ **by** *auto*
then have $Cs''\ \cdot\cdot cl\ \eta s'' = Cs$
using $\langle CAs''\ \cdot\cdot cl\ \eta s'' = CAs\rangle\ AAs'\_AAs\ cas\ n$ **by** ($auto\ intro$: $nth\_equalityI$)

show *?thesis*
using *that*
$\langle AAs''\ \cdot\cdot aml\ \eta s'' = AAs\rangle\ \langle Cs''\ \cdot\cdot cl\ \eta s'' = Cs\rangle\ \langle\forall i < n.\ CAs''\ !\ i = Cs''\ !\ i + poss\ (AAs''\ !\ i)\rangle$
$\langle length\ AAs'' = n\rangle\ \langle length\ Cs'' = n\rangle$
by *blast*
qed

— Obtain FO main premise
have $\exists DA''\ \eta''.\ DA'' \in M \land DA = DA'' \cdot \eta'' \land S\ DA'' \cdot \eta'' = S\_M\ S\ M\ DA \land is\_ground\_subst\ \eta''$
using $grounding\ S\_M\_grounding\_of\_clss\ select$ **by** ($metis\ le\_supE\ singletonI\ subsetCE$)
then obtain $DA''\ \eta''$ **where**
$DA''\_\eta''\_p$: $DA'' \in M \land DA = DA'' \cdot \eta'' \land S\ DA'' \cdot \eta'' = S\_M\ S\ M\ DA \land is\_ground\_subst\ \eta''$
by *auto*
— The properties we need of the FO main premise
have DA''_in_M: $DA'' \in M$
using $DA''\_\eta''\_p$ **by** *auto*
have DA''_to_DA: $DA'' \cdot \eta'' = DA$
using $DA''\_\eta''\_p$ **by** *auto*
have SDA''_to_SMDA: $S\ DA'' \cdot \eta'' = S\_M\ S\ M\ DA$
using $DA''\_\eta''\_p$ **by** *auto*
have $is\_ground\_subst\ \eta''$
using $DA''\_\eta''\_p$ **by** *auto*

— Split main premise DA” into D” and As”
obtain $D''\ As''$ **where** $D''As''\_p$:
$As''\ \cdot al\ \eta'' = As\ length\ As'' = n\ D'' \cdot \eta'' = D\ DA'' = D'' + (negs\ (mset\ As''))$
$S\_M\ S\ M\ (D + negs\ (mset\ As)) \neq \{\#\} \Longrightarrow negs\ (mset\ As'') = S\ DA''$
proof −
{
assume a: $S\_M\ S\ M\ (D + negs\ (mset\ As)) = \{\#\} \land length\ As = (Suc\ 0)$
$\land\ maximal\_wrt\ (As\ !\ 0\ \cdot a\ \sigma)\ ((D + negs\ (mset\ As)) \cdot \sigma)$
then have as: $mset\ As = \{\#As\ !\ 0\#\}$
by ($auto\ intro$: $nth\_equalityI$)
then have $negs\ (mset\ As) = \{\#Neg\ (As\ !\ 0)\#\}$
by ($simp\ add$: $\langle mset\ As = \{\#As\ !\ 0\#\}\rangle$)

```
    then have DA = D + {#Neg (As ! 0)#}
      using da by auto
    then obtain L where L ∈# DA'' ∧ L ·l η'' = Neg (As ! 0)
      using DA''_to_DA by (metis Melem_subst_cls mset_subset_eq_add_right single_subset_iff)
    then have Neg (atm_of L) ∈# DA'' ∧ Neg (atm_of L) ·l η'' = Neg (As ! 0)
      by (metis Neg_atm_of_iff literal.sel(2) subst_lit_is_pos)
    then have [atm_of L] ·al η'' = As ∧ negs (mset [atm_of L]) ⊆# DA''
      using as subst_lit_def by auto
    then have ∃As'. As' ·al η'' = As ∧ negs (mset As') ⊆# DA''
      ∧ (S_M S M (D + negs (mset As)) ≠ {#} ⟶ negs (mset As') = S DA'')
      using a by blast
  }
  moreover
  {
    assume S_M S M (D + negs (mset As)) = negs (mset As)
    then have negs (mset As) = S DA'' · η''
      using da ‹S DA'' · η'' = S_M S M DA› by auto
    then have ∃As'. negs (mset As') = S DA'' ∧ As' ·al η'' = As
      using instance_list[of As S DA'' η''] S.S_selects_neg_lits by auto
    then have ∃As'. As' ·al η'' = As ∧ negs (mset As') ⊆# DA''
      ∧ (S_M S M (D + negs (mset As)) ≠ {#} ⟶ negs (mset As') = S DA'')
      using S.S_selects_subseteq by auto
  }
  ultimately have ∃As''. As'' ·al η'' = As ∧ (negs (mset As'')) ⊆# DA''
    ∧ (S_M S M (D + negs (mset As)) ≠ {#} ⟶ negs (mset As'') = S DA'')
    using eligible unfolding eligible.simps by auto
  then obtain As'' where
    As'_p: As'' ·al η'' = As ∧ negs (mset As'') ⊆# DA''
    ∧ (S_M S M (D + negs (mset As)) ≠ {#} ⟶ negs (mset As'') = S DA'')
    by blast
  then have length As'' = n
    using as_len by auto
  note n = n this

  have As'' ·al η'' = As
    using As'_p by auto

  define D'' where
    D'' = DA'' − negs (mset As'')
  then have DA'' = D'' + negs (mset As'')
    using As'_p by auto
  then have D'' · η'' = D
    using DA''_to_DA da As'_p by auto

  have S_M S M (D + negs (mset As)) ≠ {#} ⟹ negs (mset As'') = S DA''
    using As'_p by blast
  then show ?thesis
    using that ‹As'' ·al η'' = As› ‹D'' · η''= D› ‹DA'' = D'' +  (negs (mset As''))› ‹length As'' = n›
    by metis
qed

show ?thesis
  using that[OF n(2,1) DA''_in_M  DA''_to_DA SDA''_to_SMDA CAs''_in_M CAs''_to_CAs SCAs''_to_SMCAs
      ‹is_ground_subst η''› ‹is_ground_subst_list ηs''› ‹As''  ·al η'' = As›
      ‹AAs'' ··aml ηs'' = AAs›
      ‹length As'' = n›
      ‹D'' · η'' = D›
      ‹DA'' = D'' + (negs (mset As''))›
      ‹S_M S M (D + negs (mset As)) ≠ {#} ⟹ negs (mset As'') = S DA''›
      ‹length Cs'' = n›
      ‹Cs'' ··cl ηs'' = Cs›
      ‹∀i < n. CAs'' ! i = Cs'' ! i + poss (AAs'' ! i)›
      ‹length AAs'' = n›]
```

by *auto*
qed

lemma
assumes $Pos\ A \in\#\ C$
shows $A \in atms\_of\ C$
using *assms*
by (*simp add*: *atm_iff_pos_or_neg_lit*)

lemma *ord_resolve_rename_lifting*:
assumes
sel_stable: $\bigwedge\varrho\ C.\ is\_renaming\ \varrho \Longrightarrow S\ (C \cdot \varrho) = S\ C \cdot \varrho$ **and**
res_e: $ord\_resolve\ (S\_M\ S\ M)\ CAs\ DA\ AAs\ As\ \sigma\ E$ **and**
select: *selection S* **and**
grounding: $\{DA\} \cup set\ CAs \subseteq grounding\_of\_clss\ M$
obtains $\eta s\ \eta\ \eta 2\ CAs''\ DA''\ AAs''\ As''\ E''\ \tau$ **where**
$is\_ground\_subst\ \eta$
$is\_ground\_subst\_list\ \eta s$
$is\_ground\_subst\ \eta 2$
$ord\_resolve\_rename\ S\ CAs''\ DA''\ AAs''\ As''\ \tau\ E''$

$CAs''\ \cdot\cdot cl\ \eta s = CAs\ DA'' \cdot \eta = DA\ E'' \cdot \eta 2 = E$
$\{DA''\} \cup set\ CAs'' \subseteq M$
using *res_e*
proof (*cases rule*: *ord_resolve.cases*)
case (*ord_resolve n Cs D*)
note $da = this(1)$ **and** $e = this(2)$ **and** $cas\_len = this(3)$ **and** $cs\_len = this(4)$ **and**
$aas\_len = this(5)$ **and** $as\_len = this(6)$ **and** $nz = this(7)$ **and** $cas = this(8)$ **and**
$aas\_not\_empt = this(9)$ **and** $mgu = this(10)$ **and** $eligible = this(11)$ **and** $str\_max = this(12)$ **and**
$sel\_empt = this(13)$

have *sel_ren_list_inv*:
$\bigwedge\varrho s\ Cs.\ length\ \varrho s = length\ Cs \Longrightarrow is\_renaming\_list\ \varrho s \Longrightarrow map\ S\ (Cs\ \cdot\cdot cl\ \varrho s) = map\ S\ Cs\ \cdot\cdot cl\ \varrho s$
using *sel_stable* **unfolding** *is_renaming_list_def* **by** (*auto intro*: *nth_equalityI*)

note $n = \langle n \neq 0\rangle\ \langle length\ CAs = n\rangle\ \langle length\ Cs = n\rangle\ \langle length\ AAs = n\rangle\ \langle length\ As = n\rangle$

interpret *S*: *selection S* **by** (*rule select*)

obtain $DA''\ \eta''\ CAs''\ \eta s''\ As''\ AAs''\ D''\ Cs''$ **where** as'':
$length\ CAs'' = n$
$length\ \eta s'' = n$
$DA'' \in M$
$DA'' \cdot \eta'' = DA$
$S\ DA'' \cdot \eta'' = S\_M\ S\ M\ DA$
$\forall\ CA'' \in set\ CAs''.\ CA'' \in M$
$CAs''\ \cdot\cdot cl\ \eta s'' = CAs$
$map\ S\ CAs''\ \cdot\cdot cl\ \eta s'' = map\ (S\_M\ S\ M)\ CAs$
$is\_ground\_subst\ \eta''$
$is\_ground\_subst\_list\ \eta s''$
$As''\ \cdot al\ \eta'' = As$
$AAs''\ \cdot\cdot aml\ \eta s'' = AAs$
$length\ As'' = n$
$D'' \cdot \eta'' = D$
$DA'' = D'' + (negs\ (mset\ As''))$
$S\_M\ S\ M\ (D + negs\ (mset\ As)) \neq \{\#\} \Longrightarrow negs\ (mset\ As'') = S\ DA''$
$length\ Cs'' = n$
$Cs''\ \cdot\cdot cl\ \eta s'' = Cs$
$\forall\ i < n.\ CAs''\ !\ i = Cs''\ !\ i + poss\ (AAs''\ !\ i)$
$length\ AAs'' = n$
using *ord_resolve_obtain_clauses*[*of S M CAs DA*, *OF res_e select grounding n*(*2*) $\langle DA = D + negs\ (mset\ As)\rangle$
$\langle\forall\ i{<}n.\ CAs\ !\ i = Cs\ !\ i + poss\ (AAs\ !\ i)\rangle\ \langle length\ Cs = n\rangle\ \langle length\ AAs = n\rangle$, *of thesis*] **by** *blast*

note n = ⟨*length* CAs'' = n⟩ ⟨*length* $\eta s''$ = n⟩ ⟨*length* As'' = n⟩ ⟨*length* AAs'' = n⟩ ⟨*length* Cs'' = n⟩ n

have *length* (*renamings_apart* (DA'' # CAs'')) = *Suc n*
using *n renames_apart* **by** *auto*

note n = *this n*

define ϱ **where**
ϱ = *hd* (*renamings_apart* (DA'' # CAs''))
define ϱs **where**
ϱs = *tl* (*renamings_apart* (DA'' # CAs''))
define DA' **where**
$DA' = DA'' \cdot \varrho$
define D' **where**
$D' = D'' \cdot \varrho$
define As' **where**
$As' = As''\ \cdot al\ \varrho$
define CAs' **where**
$CAs' = CAs''\ \cdot\cdot cl\ \varrho s$
define Cs' **where**
$Cs' = Cs''\ \cdot\cdot cl\ \varrho s$
define AAs' **where**
$AAs' = AAs''\ \cdot\cdot aml\ \varrho s$
define η' **where**
η' = *inv_renaming* $\varrho \odot \eta''$
define $\eta s'$ **where**
$\eta s'$ = *map inv_renaming* $\varrho s \odot s\ \eta s''$

have *renames_DA''*: *is_renaming* ϱ
using *renames_apart* **unfolding** *ϱ_def*
by (*metis length_greater_0_conv list.exhaust_sel list.set_intros*(*1*) *list.simps*(*3*))

have *renames_CAs''*: *is_renaming_list* ϱs
using *renames_apart* **unfolding** *ϱs_def*
by (*metis is_renaming_list_def length_greater_0_conv list.set_sel*(*2*) *list.simps*(*3*))

have *length* $\varrho s = n$
unfolding *ϱs_def* **using** *n* **by** *auto*
note n = n ⟨*length* $\varrho s = n$⟩
have *length* $As' = n$
unfolding *As'_def* **using** *n* **by** *auto*
have *length* $CAs' = n$
using *as''*(*1*) *n* **unfolding** *CAs'_def* **by** *auto*
have *length* $Cs' = n$
unfolding *Cs'_def* **using** *n* **by** *auto*
have *length* $AAs' = n$
unfolding *AAs'_def* **using** *n* **by** *auto*
have *length* $\eta s' = n$
using *as''*(*2*) *n* **unfolding** *ηs'_def* **by** *auto*
note n = ⟨*length* $CAs' = n$⟩ ⟨*length* $\eta s' = n$⟩ ⟨*length* $As' = n$⟩ ⟨*length* $AAs' = n$⟩ ⟨*length* $Cs' = n$⟩ n

have *DA'_DA*: $DA' \cdot \eta' = DA$
using *as''*(*4*) **unfolding** *η'_def DA'_def* **using** *renames_DA''* **by** *simp*
have *D'_D*: $D' \cdot \eta' = D$
using *as''*(*14*) **unfolding** *η'_def D'_def* **using** *renames_DA''* **by** *simp*
have *As'_As*: $As'\ \cdot al\ \eta' = As$
using *as''*(*11*) **unfolding** *η'_def As'_def* **using** *renames_DA''* **by** *auto*
have *S* $DA' \cdot \eta'$ = *S_M S M DA*
using *as''*(*5*) **unfolding** *η'_def DA'_def* **using** *renames_DA'' sel_stable* **by** *auto*
have *CAs'_CAs*: $CAs'\ \cdot\cdot cl\ \eta s' = CAs$
using *as''*(*7*) **unfolding** *CAs'_def ηs'_def* **using** *renames_apart renames_CAs'' n* **by** *auto*
have *Cs'_Cs*: $Cs'\ \cdot\cdot cl\ \eta s' = Cs$
using *as''*(*18*) **unfolding** *Cs'_def ηs'_def* **using** *renames_apart renames_CAs'' n* **by** *auto*

have AAs'_AAs: $AAs' \cdot\cdot aml\ \eta s' = AAs$
  using $as''(12)$ **unfolding** $\eta s'\_def\ AAs'\_def$ **using** $renames\_CAs''$ **using** n **by** $auto$
have $map\ S\ CAs' \cdot\cdot cl\ \eta s' = map\ (S\_M\ S\ M)\ CAs$
  unfolding $CAs'\_def\ \eta s'\_def$ **using** $as''(8)\ n\ renames\_CAs''\ sel\_ren\_list\_inv$ **by** $auto$

have DA'_split: $DA' = D' + negs\ (mset\ As')$
  using $as''(15)\ DA'\_def\ D'\_def\ As'\_def$ **by** $auto$
then have D'_subset_DA': $D' \subseteq\#\ DA'$
  by $auto$
from DA'_split **have** $negs\_As'\_subset\_DA'$: $negs\ (mset\ As') \subseteq\#\ DA'$
  by $auto$

have CAs'_split: $\forall\, i{<}n.\ CAs'\ !\ i = Cs'\ !\ i + poss\ (AAs'\ !\ i)$
  using $as''(19)\ CAs'\_def\ Cs'\_def\ AAs'\_def\ n$ **by** $auto$
then have $\forall\, i{<}n.\ Cs'\ !\ i \subseteq\#\ CAs'\ !\ i$
  by $auto$
from CAs'_split **have** $poss\_AAs'\_subset\_CAs'$: $\forall\, i{<}n.\ poss\ (AAs'\ !\ i) \subseteq\#\ CAs'\ !\ i$
  by $auto$
then have $AAs'\_in\_atms\_of\_CAs'$: $\forall\, i < n.\ \forall\, A{\in}\#AAs'\ !\ i.\ A \in atms\_of\ (CAs'\ !\ i)$
  by $(auto\ simp\ add{:}\ atm\_iff\_pos\_or\_neg\_lit)$

have as':
  $S\_M\ S\ M\ (D + negs\ (mset\ As)) \neq \{\#\} \Longrightarrow negs\ (mset\ As') = S\ DA'$
proof $-$
  assume a: $S\_M\ S\ M\ (D + negs\ (mset\ As)) \neq \{\#\}$
  then have $negs\ (mset\ As'') \cdot \varrho = S\ DA'' \cdot \varrho$
    using $as''(16)$ **unfolding** ϱ_def **by** $metis$
  then show $negs\ (mset\ As') = S\ DA'$
    using $As'\_def\ DA'\_def$ **using** $sel\_stable[of\ \ \varrho\ DA'']\ renames\_DA''$ **by** $auto$
qed

have vd: $var\_disjoint\ (DA'\ \#\ CAs')$
  unfolding $DA'\_def\ CAs'\_def$ **using** $renames\_apart[of\ DA''\ \#\ CAs'']$
  unfolding $\varrho\_def\ \ \varrho s\_def$
  by $(metis\ length\_greater\_0\_conv\ list.exhaust\_sel\ n(6)\ substitution.subst\_cls\_lists\_Cons$
      $substitution\_axioms\ zero\_less\_Suc)$

— Introduce ground substitution
from $vd\ DA'\_DA\ CAs'\_CAs$ **have** $\exists \eta.\ \forall\, i < Suc\ n.\ \forall\, S.\ S \subseteq\#\ (DA'\ \#\ CAs')\ !\ i \longrightarrow S \cdot (\eta'\#\eta s')\ !\ i = S \cdot \eta$
  unfolding $var\_disjoint\_def$ **using** n **by** $auto$
then obtain η **where** η_p: $\forall\, i < Suc\ n.\ \forall\, S.\ S \subseteq\#\ (DA'\ \#\ CAs')\ !\ i \longrightarrow S \cdot (\eta'\#\eta s')\ !\ i = S \cdot \eta$
  by $auto$
have η_p_lit: $\forall\, i < Suc\ n.\ \forall\, L.\ L \in\#\ (DA'\ \#\ CAs')\ !\ i \longrightarrow L \cdot l\ (\eta'\#\eta s')\ !\ i = L \cdot l\ \eta$
proof $(rule, rule, rule, rule)$
  fix $i :: nat$ **and** $L :: 'a\ literal$
  assume a:
    $i < Suc\ n$
    $L \in\#\ (DA'\ \#\ CAs')\ !\ i$
  then have $\forall\, S.\ S \subseteq\#\ (DA'\ \#\ CAs')\ !\ i \longrightarrow S \cdot (\eta'\ \#\ \eta s')\ !\ i = S \cdot \eta$
    using η_p **by** $auto$
  then have $\{\#\ L\ \#\} \cdot (\eta'\ \#\ \eta s')\ !\ i = \{\#\ L\ \#\} \cdot \eta$
    using a **by** $(meson\ single\_subset\_iff)$
  then show $L \cdot l\ (\eta'\ \#\ \eta s')\ !\ i = L \cdot l\ \eta$ **by** $auto$
qed
have η_p_atm: $\forall\, i < Suc\ n.\ \forall\, A.\ A \in atms\_of\ ((DA'\ \#\ CAs')\ !\ i) \longrightarrow A \cdot a\ (\eta'\#\eta s')\ !\ i = A \cdot a\ \eta$
proof $(rule, rule, rule, rule)$
  fix $i :: nat$ **and** $A :: 'a$
  assume a:
    $i < Suc\ n$
    $A \in atms\_of\ ((DA'\ \#\ CAs')\ !\ i)$
  then obtain L **where** L_p: $atm\_of\ L = A \wedge L \in\#\ (DA'\ \#\ CAs')\ !\ i$
    unfolding $atms\_of\_def$ **by** $auto$
  then have $L \cdot l\ (\eta'\#\eta s')\ !\ i = L \cdot l\ \eta$

```
    using η_p_lit a by auto
  then show A ·a (η' # ηs') ! i = A ·a η
    using L_p unfolding subst_lit_def by (cases L) auto
qed

have DA'_DA: DA' · η = DA
  using DA'_DA η_p by auto
have D' · η = D using η_p D'_D n D'_subset_DA' by auto
have As' ·al η = As
proof (rule nth_equalityI)
  show length (As' ·al η) = length As
    using n by auto
next
  show ∀i<length (As' ·al η). (As' ·al η) ! i = As ! i
  proof (rule, rule)
    fix i :: nat
    assume a: i < length (As' ·al η)
    have A_eq: ∀A. A ∈ atms_of DA' ⟶ A ·a η' = A ·a η
      using η_p_atm n by force
    have As' ! i ∈ atms_of DA'
      using negs_As'_subset_DA' unfolding atms_of_def
      using a n by force
    then have As' ! i ·a η' = As' ! i ·a η
       using A_eq by simp
    then show (As' ·al η) ! i = As ! i
      using As'_As ⟨length As' = n⟩ a by auto
  qed
qed

have S DA' · η = S_M S M DA
  using ⟨S DA' · η' = S_M S M DA⟩ η_p S.S_selects_subseteq by auto

from η_p have η_p_CAs': ∀i < n. (CAs' ! i) · (ηs' ! i) = (CAs' ! i) · η
  using n by auto
then have CAs' ··cl ηs' = CAs' ·cl η
  using n by (auto intro: nth_equalityI)
then have CAs'_η_fo_CAs: CAs' ·cl η = CAs
  using CAs'_CAs η_p n by auto

from η_p have ∀i < n. S (CAs' ! i) · ηs' ! i = S (CAs' ! i) · η
  using S.S_selects_subseteq n by auto
then have map S CAs' ··cl ηs' = map S CAs' ·cl η
  using n by (auto intro: nth_equalityI)
then have SCAs'_η_fo_SMCAs: map S CAs' ·cl η = map (S_M S M) CAs
  using ⟨map S CAs' ··cl ηs' = map (S_M S M) CAs⟩ by auto

have Cs' ·cl η = Cs
proof (rule nth_equalityI)
  show length (Cs' ·cl η) = length Cs
    using n by auto
next
  show ∀i<length (Cs' ·cl η). (Cs' ·cl η) ! i = Cs ! i
  proof (rule, rule)
    fix i :: nat
    assume i < length (Cs' ·cl η)
    then have a: i < n
      using n by force
    have (Cs' ··cl ηs') ! i = Cs ! i
      using Cs'_Cs a n by force
    moreover
    have η_p_CAs': ∀S. S ⊆# CAs' ! i ⟶ S · ηs' ! i = S · η
      using η_p a by force
    have Cs' ! i · ηs' ! i = (Cs' ·cl η) ! i
```

```
      using η_p_CAs' ‹∀i<n. Cs' ! i ⊆# CAs' ! i› a n by force
    then have (Cs' ··cl ηs') ! i = (Cs' ·cl η) ! i
      using a n by force
    ultimately show (Cs' ·cl η) ! i = Cs ! i
      by auto
    thm Cs'_Cs η_p ‹∀i<n. Cs' ! i ⊆# CAs' ! i› a
  qed
qed

have AAs'_AAs: AAs' ·aml η = AAs
proof (rule nth_equalityI)
  show length (AAs' ·aml η) = length AAs
    using n by auto
next
  show ∀i<length (AAs' ·aml η). (AAs' ·aml η) ! i = AAs ! i
  proof (rule, rule)
    fix i :: nat
    assume a: i < length (AAs' ·aml η)
    then have i < n
      using n by force
    then have ∀A. A ∈ atms_of ((DA' # CAs') ! Suc i) ⟶ A ·a (η' # ηs') ! Suc i = A ·a η
      using η_p_atm n by force
    then have A_eq: ∀A. A ∈ atms_of (CAs' ! i) ⟶ A ·a ηs' ! i = A ·a η
      by auto
    have AAs_CAs': ∀A ∈# AAs' ! i. A ∈ atms_of (CAs' ! i)
      using AAs'_in_atms_of_CAs' unfolding atms_of_def
      using a n by force
    then have AAs' ! i ·am  ηs' ! i = AAs' ! i ·am η
      unfolding subst_atm_mset_def using A_eq unfolding subst_atm_mset_def by auto
    then show (AAs' ·aml η) ! i = AAs ! i
       using AAs'_AAs ‹length AAs' = n› ‹length ηs' = n› a by auto
  qed
qed

— Obtain MGU and substitution
obtain τ φ where τφ:
  Some τ = mgu (set_mset ‘ set (map2 add_mset As' AAs'))
  τ ⊙ φ = η ⊙ σ
proof –
  have uu: is_unifiers σ (set_mset ‘ set (map2 add_mset (As' ·al η) (AAs' ·aml η)))
    using mgu mgu_sound is_mgu_def unfolding ‹AAs' ·aml η = AAs› using ‹As' ·al η = As› by auto
  have ησuni: is_unifiers (η ⊙ σ) (set_mset ‘ set (map2 add_mset As' AAs'))
  proof –
    have set_mset ‘ set (map2 add_mset As' AAs' ·aml η) =
      set_mset ‘ set (map2 add_mset As' AAs') ·ass η
      unfolding subst_atmss_def subst_atm_mset_list_def using subst_atm_mset_def subst_atms_def
      by (simp add: image_image subst_atm_mset_def subst_atms_def)
    then have is_unifiers σ (set_mset ‘ set (map2 add_mset As' AAs') ·ass η)
      using uu by (auto simp: n map2_add_mset_map)
    then show ?thesis
      using is_unifiers_comp by auto
  qed
  then obtain τ where
    τ_p: Some τ = mgu (set_mset ‘ set (map2 add_mset As' AAs'))
    using mgu_complete
    by (metis (mono_tags, hide_lams) List.finite_set finite_imageI finite_set_mset image_iff)
  moreover then obtain φ where φ_p: τ ⊙ φ = η ⊙ σ
    by (metis (mono_tags, hide_lams) finite_set ησuni finite_imageI finite_set_mset image_iff
        mgu_sound set_mset_mset substitution_ops.is_mgu_def)
  ultimately show thesis
    using that by auto
qed
```

— Lifting eligibility
have *eligible'*: $eligible\ S\ \tau\ As'\ (D' + negs\ (mset\ As'))$
proof −
have $S\_M\ S\ M\ (D + negs\ (mset\ As)) = negs\ (mset\ As) \lor S\_M\ S\ M\ (D + negs\ (mset\ As)) = \{\#\} \land$
$length\ As = 1 \land maximal\_wrt\ (As\ !\ 0 \cdot a\ \sigma)\ ((D + negs\ (mset\ As)) \cdot \sigma)$
using *eligible* **unfolding** *eligible.simps* **by** *auto*
then show *?thesis*
proof
assume $S\_M\ S\ M\ (D + negs\ (mset\ As)) = negs\ (mset\ As)$
then have $S\_M\ S\ M\ (D + negs\ (mset\ As)) \neq \{\#\}$
using *n* **by** *force*
then have $S\ (D' + negs\ (mset\ As')) = negs\ (mset\ As')$
using *as' DA'_split* **by** *auto*
then show *?thesis*
unfolding *eligible.simps*[*simplified*] **by** *auto*
next
assume *asm*: $S\_M\ S\ M\ (D + negs\ (mset\ As)) = \{\#\} \land length\ As = 1 \land$
$maximal\_wrt\ (As\ !\ 0 \cdot a\ \sigma)\ ((D + negs\ (mset\ As)) \cdot \sigma)$
then have $S\ (D' + negs\ (mset\ As')) = \{\#\}$
using $\langle D' \cdot \eta = D\rangle$[*symmetric*] $\langle As' \cdot al\ \eta = As\rangle$[*symmetric*] $\langle S\ (DA') \cdot \eta = S\_M\ S\ M\ (DA)\rangle$
da DA'_split subst_cls_empty_iff **by** *metis*
moreover from *asm* **have** *l*: $length\ As' = 1$
using $\langle As' \cdot al\ \eta = As\rangle$ **by** *auto*
moreover from *asm* **have** $maximal\_wrt\ (As'\ !\ 0 \cdot a\ (\tau \odot \varphi))\ ((D' + negs\ (mset\ As')) \cdot (\tau \odot \varphi))$
using $\langle As' \cdot al\ \eta = As\rangle$ $\langle D' \cdot \eta = D\rangle$ **using** $l\ \tau\varphi$ **by** *auto*
then have $maximal\_wrt\ (As'\ !\ 0 \cdot a\ \tau \cdot a\ \varphi)\ ((D' + negs\ (mset\ As')) \cdot \tau \cdot \varphi)$
by *auto*
then have $maximal\_wrt\ (As'\ !\ 0 \cdot a\ \tau)\ ((D' + negs\ (mset\ As')) \cdot \tau)$
using *maximal_wrt_subst* **by** *blast*
ultimately show *?thesis*
unfolding *eligible.simps*[*simplified*] **by** *auto*
qed
qed

— Lifting maximality
have *maximality*: $\forall i < n.\ strictly\_maximal\_wrt\ (As'\ !\ i \cdot a\ \tau)\ (Cs'\ !\ i \cdot \tau)$

proof −
from *str_max* **have** $\forall i < n.\ strictly\_maximal\_wrt\ ((As' \cdot al\ \eta)\ !\ i \cdot a\ \sigma)\ ((Cs' \cdot cl\ \eta)\ !\ i \cdot \sigma)$
using $\langle As' \cdot al\ \eta = As\rangle$ $\langle Cs' \cdot cl\ \eta = Cs\rangle$ **by** *simp*
then have $\forall i < n.\ strictly\_maximal\_wrt\ (As'\ !\ i \cdot a\ (\tau \odot \varphi))\ (Cs'\ !\ i \cdot (\tau \odot \varphi))$
using $n\ \tau\varphi$ **by** *simp*
then have $\forall i < n.\ strictly\_maximal\_wrt\ (As'\ !\ i \cdot a\ \tau \cdot a\ \varphi)\ (Cs'\ !\ i \cdot \tau \cdot \varphi)$
by *auto*
then show $\forall i < n.\ strictly\_maximal\_wrt\ (As'\ !\ i \cdot a\ \tau)\ (Cs'\ !\ i \cdot \tau)$
using *strictly_maximal_wrt_subst* $\tau\varphi$ **by** *blast*
qed

— Lifting nothing being selected
have *nothing_selected*: $\forall i < n.\ S\ (CAs'\ !\ i) = \{\#\}$
proof −
have $\forall i < n.\ (map\ S\ CAs' \cdot cl\ \eta)\ !\ i = map\ (S\_M\ S\ M)\ CAs\ !\ i$
by (*simp add*: $\langle map\ S\ CAs' \cdot cl\ \eta = map\ (S\_M\ S\ M)\ CAs\rangle$)
then have $\forall i < n.\ S\ (CAs'\ !\ i) \cdot \eta = S\_M\ S\ M\ (CAs\ !\ i)$
using *n* **by** *auto*
then have $\forall i < n.\ S\ (CAs'\ !\ i) \cdot \eta = \{\#\}$
using *sel_empt* $\langle \forall i < n.\ S\ (CAs'\ !\ i) \cdot \eta = S\_M\ S\ M\ (CAs\ !\ i)\rangle$ **by** *auto*
then show $\forall i < n.\ S\ (CAs'\ !\ i) = \{\#\}$
using *subst_cls_empty_iff* **by** *blast*
qed

— Lifting AAs's non-emptiness
have $\forall i < n.\ AAs'\ !\ i \neq \{\#\}$

```
    using n aas_not_empt ‹AAs' ·aml η = AAs› by auto

  — Resolve the lifted clauses
  define E' where
    E' = ((⋃# mset Cs') + D') · τ

  have res_e': ord_resolve S CAs' DA' AAs' As' τ E'
    using ord_resolve.intros[of CAs' n Cs' AAs' As' τ S D',
      OF _ _ _ _ _ _ ‹∀ i < n. AAs' ! i ≠ {#}› τφ(1) eligible'
        ‹∀ i < n. strictly_maximal_wrt (As' ! i ·a τ) (Cs' ! i · τ)› ‹∀ i < n. S (CAs' ! i) = {#}›]
    unfolding E'_def using DA'_split n ‹∀ i<n. CAs' ! i = Cs' ! i + poss (AAs' ! i)› by blast

  — Prove resolvent instantiates to ground resolvent
  have e'φe: E' · φ = E
  proof –
    have E' · φ = ((⋃# mset Cs') + D') · (τ ⊙ φ)
      unfolding E'_def by auto
    also have ... = (⋃# mset Cs' + D') · (η ⊙ σ)
      using τφ by auto
    also have ... = (⋃# mset Cs + D) · σ
      using ‹Cs' ·cl η = Cs› ‹D' · η = D› by auto
    also have ... = E
      using e by auto
    finally show e'φe: E' · φ = E
      .
  qed

  — Replace φ with a true ground substitution
  obtain η2 where
    ground_η2: is_ground_subst η2 E' · η2 = E
  proof –
    have is_ground_cls_list CAs is_ground_cls DA
      using grounding grounding_ground unfolding is_ground_cls_list_def by auto
    then have is_ground_cls E
      using res_e ground_resolvent_subset by (force intro: is_ground_cls_mono)
    then show thesis
      using that e'φe make_ground_subst by auto
  qed


  — Wrap up the proof
  have ord_resolve S (CAs'' ·cl ϱs) (DA'' · ϱ) (AAs'' ·aml ϱs) (As'' ·al ϱ) τ E'
    using res_e' As'_def ϱ_def AAs'_def ϱs_def DA'_def ϱ_def CAs'_def ϱs_def by simp
  moreover have ∀ i<n. poss (AAs'' ! i) ⊆# CAs'' ! i
    using as''(19) by auto
  moreover have negs (mset As'') ⊆# DA''
    using local.as''(15) by auto
  ultimately have ord_resolve_rename S CAs'' DA'' AAs'' As'' τ E'
    using ord_resolve_rename[of CAs'' n AAs'' As'' DA'' ϱ ϱs S τ E'] ϱ_def ϱs_def n by auto
  then show thesis
    using that[of η'' ηs'' η2 CAs'' DA''] ‹is_ground_subst η''› ‹is_ground_subst_list ηs''›
      ‹is_ground_subst η2› ‹CAs'' ·cl ηs'' = CAs› ‹DA'' · η'' = DA› ‹E' · η2 = E› ‹DA'' ∈ M›
      ‹∀ CA ∈ set CAs''. CA ∈ M› by blast
qed

end

end
```

15 An Ordered Resolution Prover for First-Order Clauses

```
theory FO_Ordered_Resolution_Prover
  imports FO_Ordered_Resolution
```

begin

This material is based on Section 4.3 ("A Simple Resolution Prover for First-Order Clauses") of Bachmair and Ganzinger's chapter. Specifically, it formalizes the RP prover defined in Figure 5 and its related lemmas and theorems, including Lemmas 4.10 and 4.11 and Theorem 4.13 (completeness).

definition $\mathit{is\_least} :: (\mathit{nat} \Rightarrow \mathit{bool}) \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$ **where**
$\mathit{is\_least}\ P\ n \longleftrightarrow P\ n \wedge (\forall n' < n.\ \neg\ P\ n')$

lemma $\mathit{least\_exists}$: $P\ n \Longrightarrow \exists n.\ \mathit{is\_least}\ P\ n$
using $\mathit{exists\_least\_iff}$ **unfolding** $\mathit{is\_least\_def}$ **by** auto

The following corresponds to page 42 and 43 of Section 4.3, from the explanation of RP to Lemma 4.10.

type-synonym $'a\ \mathit{state} = {'a}\ \mathit{clause\ set} \times {'a}\ \mathit{clause\ set} \times {'a}\ \mathit{clause\ set}$

locale $\mathit{FO\_resolution\_prover}$ =
$\mathit{FO\_resolution}\ \mathit{subst\_atm}\ \mathit{id\_subst}\ \mathit{comp\_subst}\ \mathit{renamings\_apart}\ \mathit{atm\_of\_atms}\ \mathit{mgu}\ \mathit{less\_atm}$ +
$\mathit{selection}\ S$
for
$S :: ('a :: \mathit{wellorder})\ \mathit{clause} \Rightarrow {'a}\ \mathit{clause}$ **and**
$\mathit{subst\_atm} :: {'a} \Rightarrow {'s} \Rightarrow {'a}$ **and**
$\mathit{id\_subst} :: {'s}$ **and**
$\mathit{comp\_subst} :: {'s} \Rightarrow {'s} \Rightarrow {'s}$ **and**
$\mathit{renamings\_apart} :: {'a}\ \mathit{clause\ list} \Rightarrow {'s}\ \mathit{list}$ **and**
$\mathit{atm\_of\_atms} :: {'a}\ \mathit{list} \Rightarrow {'a}$ **and**
$\mathit{mgu} :: {'a}\ \mathit{set\ set} \Rightarrow {'s}\ \mathit{option}$ **and**
$\mathit{less\_atm} :: {'a} \Rightarrow {'a} \Rightarrow \mathit{bool}$ +
assumes
$\mathit{sel\_stable}$: $\bigwedge \varrho\ C.\ \mathit{is\_renaming}\ \varrho \Longrightarrow S\ (C \cdot \varrho) = S\ C \cdot \varrho$ **and**
$\mathit{less\_atm\_ground}$: $\mathit{is\_ground\_atm}\ A \Longrightarrow \mathit{is\_ground\_atm}\ B \Longrightarrow \mathit{less\_atm}\ A\ B \Longrightarrow A < B$
begin

fun $\mathit{N\_of\_state} :: {'a}\ \mathit{state} \Rightarrow {'a}\ \mathit{clause\ set}$ **where**
$\mathit{N\_of\_state}\ (N,\ P,\ Q) = N$

fun $\mathit{P\_of\_state} :: {'a}\ \mathit{state} \Rightarrow {'a}\ \mathit{clause\ set}$ **where**
$\mathit{P\_of\_state}\ (N,\ P,\ Q) = P$

O denotes relation composition in Isabelle, so the formalization uses Q instead.

fun $\mathit{Q\_of\_state} :: {'a}\ \mathit{state} \Rightarrow {'a}\ \mathit{clause\ set}$ **where**
$\mathit{Q\_of\_state}\ (N,\ P,\ Q) = Q$

definition $\mathit{clss\_of\_state} :: {'a}\ \mathit{state} \Rightarrow {'a}\ \mathit{clause\ set}$ **where**
$\mathit{clss\_of\_state}\ \mathit{St} = \mathit{N\_of\_state}\ \mathit{St} \cup \mathit{P\_of\_state}\ \mathit{St} \cup \mathit{Q\_of\_state}\ \mathit{St}$

abbreviation $\mathit{grounding\_of\_state} :: {'a}\ \mathit{state} \Rightarrow {'a}\ \mathit{clause\ set}$ **where**
$\mathit{grounding\_of\_state}\ \mathit{St} \equiv \mathit{grounding\_of\_clss}\ (\mathit{clss\_of\_state}\ \mathit{St})$

interpretation $\mathit{ord\_FO\_resolution}$: $\mathit{inference\_system}\ \mathit{ord\_FO\_\Gamma}\ S$ **.**

The following inductive predicate formalizes the resolution prover in Figure 5.

inductive $\mathit{RP} :: {'a}\ \mathit{state} \Rightarrow {'a}\ \mathit{state} \Rightarrow \mathit{bool}$ (**infix** $\rightsquigarrow$ *50*) **where**
$\mathit{tautology\_deletion}$: $\mathit{Neg}\ A \in\#\ C \Longrightarrow \mathit{Pos}\ A \in\#\ C \Longrightarrow (N \cup \{C\},\ P,\ Q) \rightsquigarrow (N,\ P,\ Q)$
| $\mathit{forward\_subsumption}$: $D \in P \cup Q \Longrightarrow \mathit{subsumes}\ D\ C \Longrightarrow (N \cup \{C\},\ P,\ Q) \rightsquigarrow (N,\ P,\ Q)$
| $\mathit{backward\_subsumption\_P}$: $D \in N \Longrightarrow \mathit{strictly\_subsumes}\ D\ C \Longrightarrow (N,\ P \cup \{C\},\ Q) \rightsquigarrow (N,\ P,\ Q)$
| $\mathit{backward\_subsumption\_Q}$: $D \in N \Longrightarrow \mathit{strictly\_subsumes}\ D\ C \Longrightarrow (N,\ P,\ Q \cup \{C\}) \rightsquigarrow (N,\ P,\ Q)$
| $\mathit{forward\_reduction}$: $D + \{\#L'\#\} \in P \cup Q \Longrightarrow -\ L = L' \cdot l\ \sigma \Longrightarrow D \cdot \sigma \subseteq\#\ C \Longrightarrow$
$(N \cup \{C + \{\#L\#\}\},\ P,\ Q) \rightsquigarrow (N \cup \{C\},\ P,\ Q)$
| $\mathit{backward\_reduction\_P}$: $D + \{\#L'\#\} \in N \Longrightarrow -\ L = L' \cdot l\ \sigma \Longrightarrow D \cdot \sigma \subseteq\#\ C \Longrightarrow$
$(N,\ P \cup \{C + \{\#L\#\}\},\ Q) \rightsquigarrow (N,\ P \cup \{C\},\ Q)$
| $\mathit{backward\_reduction\_Q}$: $D + \{\#L'\#\} \in N \Longrightarrow -\ L = L' \cdot l\ \sigma \Longrightarrow D \cdot \sigma \subseteq\#\ C \Longrightarrow$
$(N,\ P,\ Q \cup \{C + \{\#L\#\}\}) \rightsquigarrow (N,\ P \cup \{C\},\ Q)$
| $\mathit{clause\_processing}$: $(N \cup \{C\},\ P,\ Q) \rightsquigarrow (N,\ P \cup \{C\},\ Q)$

| *inference_computation*: N = *concls_of* (*ord_FO_resolution.inferences_between* Q C) $\Longrightarrow$
  ({}, $P \cup \{C\}$, Q) $\rightsquigarrow$ (N, P, $Q \cup \{C\}$)

lemma *final_RP*: $\neg$ ({}, {}, Q) $\rightsquigarrow$ *St*
  by (*auto elim*: *RP.cases*)

definition *Sup_state* :: *'a state llist* $\Rightarrow$ *'a state* **where**
  Sup_state Sts =
   (*Sup_llist* (*lmap N_of_state Sts*), *Sup_llist* (*lmap P_of_state Sts*),
    Sup_llist (*lmap Q_of_state Sts*))

definition *Liminf_state* :: *'a state llist* $\Rightarrow$ *'a state* **where**
  Liminf_state Sts =
   (*Liminf_llist* (*lmap N_of_state Sts*), *Liminf_llist* (*lmap P_of_state Sts*),
    Liminf_llist (*lmap Q_of_state Sts*))

context
  fixes *Sts Sts'* :: *'a state llist*
  assumes *Sts*: *lfinite Sts lfinite Sts'* $\neg$ *lnull Sts* $\neg$ *lnull Sts' llast Sts'* = *llast Sts*
begin

lemma
  N_of_Liminf_state_fin: *N_of_state* (*Liminf_state Sts'*) = *N_of_state* (*Liminf_state Sts*) **and**
  P_of_Liminf_state_fin: *P_of_state* (*Liminf_state Sts'*) = *P_of_state* (*Liminf_state Sts*) **and**
  Q_of_Liminf_state_fin: *Q_of_state* (*Liminf_state Sts'*) = *Q_of_state* (*Liminf_state Sts*)
  using *Sts* **by** (*simp_all add*: *Liminf_state_def lfinite_Liminf_llist llast_lmap*)

lemma *Liminf_state_fin*: *Liminf_state Sts'* = *Liminf_state Sts*
  using *N_of_Liminf_state_fin P_of_Liminf_state_fin Q_of_Liminf_state_fin*
  by (*simp add*: *Liminf_state_def*)

end

context
  fixes *Sts Sts'* :: *'a state llist*
  assumes *Sts*: $\neg$ *lfinite Sts emb Sts Sts'*
begin

lemma
  N_of_Liminf_state_inf: *N_of_state* (*Liminf_state Sts'*) $\subseteq$ *N_of_state* (*Liminf_state Sts*) **and**
  P_of_Liminf_state_inf: *P_of_state* (*Liminf_state Sts'*) $\subseteq$ *P_of_state* (*Liminf_state Sts*) **and**
  Q_of_Liminf_state_inf: *Q_of_state* (*Liminf_state Sts'*) $\subseteq$ *Q_of_state* (*Liminf_state Sts*)
  using *Sts* **by** (*simp_all add*: *Liminf_state_def emb_Liminf_llist_infinite emb_lmap*)

lemma *clss_of_Liminf_state_inf*:
  clss_of_state (*Liminf_state Sts'*) $\subseteq$ *clss_of_state* (*Liminf_state Sts*)
  unfolding *clss_of_state_def*
  using *N_of_Liminf_state_inf P_of_Liminf_state_inf Q_of_Liminf_state_inf* **by** *blast*

end

definition *fair_state_seq* :: *'a state llist* $\Rightarrow$ *bool* **where**
  fair_state_seq Sts $\longleftrightarrow$ *N_of_state* (*Liminf_state Sts*) = {} $\wedge$ *P_of_state* (*Liminf_state Sts*) = {}

The following formalizes Lemma 4.10.

context
  fixes
    Sts :: *'a state llist*
  assumes
    deriv: *chain* (*op* $\rightsquigarrow$) *Sts* **and**
    empty_Q0: *Q_of_state* (*lhd Sts*) = {}

begin

lemmas *lhd_lmap_Sts* = *llist.map_sel*(*1*)[*OF chain_not_lnull*[*OF deriv*]]

definition *S_Q* :: *'a clause* $\Rightarrow$ *'a clause* **where**
S_Q = *S_M S* (*Q_of_state* (*Liminf_state Sts*))

interpretation *sq*: *selection S_Q*
unfolding *S_Q_def* **using** *S_M_selects_subseteq S_M_selects_neg_lits selection_axioms*
by *unfold_locales auto*

interpretation *gr*: *ground_resolution_with_selection S_Q*
by *unfold_locales*

interpretation *sr*: *standard_redundancy_criterion_reductive gr.ord_*Γ
by *unfold_locales*

interpretation *sr*: *standard_redundancy_criterion_counterex_reducing gr.ord_*Γ
ground_resolution_with_selection.INTERP S_Q
by *unfold_locales*

The extension of ordered resolution mentioned in 4.10. We let it consist of all sound rules.

definition *ground_sound_*Γ:: *'a inference set* **where**
*ground_sound_*Γ = {*Infer CC D E* | *CC D E*. ($\forall$ *I*. *I* $\models$*m CC* $\longrightarrow$ *I* $\models$ *D* $\longrightarrow$ *I* $\models$ *E*)}

We prove that we indeed defined an extension.

lemma *gd_ord_*Γ*_ngd_ord_*Γ: *gr.ord_*Γ $\subseteq$ *ground_sound_*Γ
unfolding *ground_sound_*Γ*_def* **using** *gr.ord_*Γ*_def gr.ord_resolve_sound* **by** *fastforce*

lemma *sound_ground_sound_*Γ: *sound_inference_system ground_sound_*Γ
unfolding *sound_inference_system_def ground_sound_*Γ*_def* **by** *auto*

lemma *sat_preserving_ground_sound_*Γ: *sat_preserving_inference_system ground_sound_*Γ
using *sound_ground_sound_*Γ *sat_preserving_inference_system.intro*
sound_inference_system.Γ*_sat_preserving* **by** *blast*

definition *sr_ext_Ri* :: *'a clause set* $\Rightarrow$ *'a inference set* **where**
sr_ext_Ri N = *sr.Ri N* $\cup$ (*ground_sound_*Γ $-$ *gr.ord_*Γ)

interpretation *sr_ext*:
*sat_preserving_redundancy_criterion ground_sound_*Γ *sr.Rf sr_ext_Ri*
unfolding *sat_preserving_redundancy_criterion_def sr_ext_Ri_def*
using *sat_preserving_ground_sound_*Γ *redundancy_criterion_standard_extension gd_ord_*Γ*_ngd_ord_*Γ
sr.redundancy_criterion_axioms **by** *auto*

lemma *strict_subset_subsumption_redundant_clause*:
assumes
sub: $D \cdot \sigma \subset\# C$ **and**
*ground_*σ: *is_ground_subst* σ
shows $C \in$ *sr.Rf* (*grounding_of_cls D*)
proof $-$
from *sub* **have** $\forall I.\ I \models D \cdot \sigma \longrightarrow I \models C$
unfolding *true_cls_def* **by** *blast*
moreover have $C > D \cdot \sigma$
using *sub* **by** (*simp add*: *subset_imp_less_mset*)
moreover have $D \cdot \sigma \in$ *grounding_of_cls D*
using *ground_*σ **by** (*metis* (*mono_tags*, *lifting*) *mem_Collect_eq substitution_ops.grounding_of_cls_def*)
ultimately have *set_mset* $\{\#D \cdot \sigma\#\} \subseteq$ *grounding_of_cls D*
($\forall I.\ I \models m\ \{\#D \cdot \sigma\#\} \longrightarrow I \models C$)
($\forall D'.\ D' \in\# \{\#D \cdot \sigma\#\} \longrightarrow D' < C$)
by *auto*
then show *?thesis*
using *sr.Rf_def* **by** *blast*
qed

lemma *strict_subset_subsumption_redundant_state*:
assumes
$D \cdot \sigma \subset\# C$ **and**
is_ground_subst σ **and**
$D \in$ *clss_of_state St*
shows $C \in$ *sr.Rf* (*grounding_of_state St*)
using *assms*
proof (*induction St*)
case (*fields N P Q*)
note *sub* = *this*(*1*) **and** *gr* = *this*(*2*) **and** *d_in* = *this*(*3*)

have $C \in$ *sr.Rf* (*grounding_of_cls D*)
by (*rule strict_subset_subsumption_redundant_clause*[*OF sub gr*])
then show *?case*
using *d_in* **unfolding** *clss_of_state_def grounding_of_clss_def*
by (*metis* (*no_types*) *sr.Rf_mono sup_ge1 SUP_absorb contra_subsetD*)
qed

lemma *subst_cls_eq_grounding_of_cls_subset_eq*:
assumes $D \cdot \sigma = C$
shows *grounding_of_cls C* $\subseteq$ *grounding_of_cls D*
proof
fix $C\sigma'$
assume $C\sigma' \in$ *grounding_of_cls C*
then obtain σ' **where**
$C\sigma'$: $C \cdot \sigma' = C\sigma'$ *is_ground_subst* σ'
unfolding *grounding_of_cls_def* **by** *auto*
then have $C \cdot \sigma' = D \cdot \sigma \cdot \sigma' \wedge$ *is_ground_subst* $(\sigma \odot \sigma')$
using *assms* **by** *auto*
then show $C\sigma' \in$ *grounding_of_cls D*
unfolding *grounding_of_cls_def* **using** $C\sigma'(1)$ **by** *force*
qed

The following corresponds the part of Lemma 4.10 that states we have a theorem proving process:

lemma *resolution_prover_ground_derive*:
$St \leadsto St' \Longrightarrow$ *sr_ext.derive* (*grounding_of_state St*) (*grounding_of_state St'*)
proof (*induction rule*: *RP.induct*)
case (*tautology_deletion A C N P Q*)
{
fix $C\sigma$
assume $C\sigma \in$ *grounding_of_cls C*
then obtain σ **where**
$C\sigma = C \cdot \sigma$
unfolding *grounding_of_cls_def* **by** *auto*
then have *Neg* $(A \cdot a\ \sigma) \in\# C\sigma \wedge$ *Pos* $(A \cdot a\ \sigma) \in\# C\sigma$
using *tautology_deletion Neg_Melem_subst_atm_subst_cls Pos_Melem_subst_atm_subst_cls* **by** *auto*
then have $C\sigma \in$ *sr.Rf* (*grounding_of_state* (*N*, *P*, *Q*))
using *sr.tautology_redundant* **by** *auto*
}
then have *grounding_of_state* $(N \cup \{C\}, P, Q)$ − *grounding_of_state* (N, P, Q)
$\subseteq$ *sr.Rf* (*grounding_of_state* (*N*, *P*, *Q*))
unfolding *clss_of_state_def grounding_of_clss_def* **by** *auto*
moreover have *grounding_of_state* (N, P, Q) − *grounding_of_state* $(N \cup \{C\}, P, Q) = \{\}$
unfolding *clss_of_state_def grounding_of_clss_def* **by** *auto*
ultimately show *?case*
using *sr_ext.derive.intros*[*of grounding_of_state* (N, P, Q) *grounding_of_state* $(N \cup \{C\}, P, Q)$]
by *auto*
next
case (*forward_subsumption D P Q C N*)
note *D_p* = *this*
then obtain σ **where**
$D \cdot \sigma = C \vee D \cdot \sigma \subset\# C$

by (*auto simp*: *subsumes_def subset_mset_def*)
then have $D \cdot \sigma = C \lor D \cdot \sigma \subset\# C$
by (*simp add*: *subset_mset_def*)
then show *?case*
proof
assume $D \cdot \sigma = C$
then have *grounding_of_cls* $C \subseteq$ *grounding_of_cls* D
using *subst_cls_eq_grounding_of_cls_subset_eq* **by** *simp*
then have *grounding_of_state* $(N \cup \{C\}, P, Q) =$ *grounding_of_state* (N, P, Q)
using *D_p* **unfolding** *clss_of_state_def grounding_of_clss_def* **by** *auto*
then show *?case*
by (*auto intro*: *sr_ext.derive.intros*)
next
assume *a*: $D \cdot \sigma \subset\# C$
have *grounding_of_cls* $C \subseteq$ *sr.Rf* (*grounding_of_state* (N, P, Q))
proof
fix $C\mu$
assume $C\mu \in$ *grounding_of_cls* C
then obtain μ **where**
μ_p: $C\mu = C \cdot \mu \land$ *is_ground_subst* μ
unfolding *grounding_of_cls_def* **by** *auto*
have $D\sigma\mu C\mu$: $D \cdot \sigma \cdot \mu \subset\# C \cdot \mu$
using *a subst_subset_mono* **by** *auto*
then show $C\mu \in$ *sr.Rf* (*grounding_of_state* (N, P, Q))
using *μ_p strict_subset_subsumption_redundant_state*[*of* $D\ \sigma \odot \mu\ C \cdot \mu\ (N, P, Q)$] *D_p*
unfolding *clss_of_state_def* **by** *auto*
qed
then show *?case*
unfolding *clss_of_state_def grounding_of_clss_def* **by** (*force intro*: *sr_ext.derive.intros*)
qed
next
case (*backward_subsumption_P D N C P Q*)

note *D_p* = *this*
then obtain σ **where**
$D \cdot \sigma = C \lor D \cdot \sigma \subset\# C$
by (*auto simp*: *strictly_subsumes_def subsumes_def subset_mset_def*)
then show *?case*
proof
assume $D \cdot \sigma = C$
then have *grounding_of_cls* $C \subseteq$ *grounding_of_cls* D
using *subst_cls_eq_grounding_of_cls_subset_eq* **by** *simp*
then have *grounding_of_state* $(N, P \cup \{C\}, Q) =$ *grounding_of_state* (N, P, Q)
using *D_p* **unfolding** *clss_of_state_def grounding_of_clss_def* **by** *auto*
then show *?case*
by (*auto intro*: *sr_ext.derive.intros*)
next
assume *a*: $D \cdot \sigma \subset\# C$
have *grounding_of_cls* $C \subseteq$ *sr.Rf* (*grounding_of_state* (N, P, Q))
proof
fix $C\mu$
assume $C\mu \in$ *grounding_of_cls* C
then obtain μ **where**
μ_p: $C\mu = C \cdot \mu \land$ *is_ground_subst* μ
unfolding *grounding_of_cls_def* **by** *auto*
have $D\sigma\mu C\mu$: $D \cdot \sigma \cdot \mu \subset\# C \cdot \mu$
using *a subst_subset_mono* **by** *auto*
then show $C\mu \in$ *sr.Rf* (*grounding_of_state* (N, P, Q))
using *μ_p strict_subset_subsumption_redundant_state*[*of* $D\ \sigma \odot \mu\ C \cdot \mu\ (N, P, Q)$] *D_p*
unfolding *clss_of_state_def* **by** *auto*
qed
then show *?case*
unfolding *clss_of_state_def grounding_of_clss_def* **by** (*force intro*: *sr_ext.derive.intros*)

qed
next
case (*backward_subsumption_Q D N C P Q*)
note *D_p* = *this*
then obtain σ **where**
$D \cdot \sigma = C \lor D \cdot \sigma \subset\# C$
by (*auto simp*: *strictly_subsumes_def subsumes_def subset_mset_def*)
then show *?case*
proof
assume $D \cdot \sigma = C$
then have *grounding_of_cls* $C \subseteq$ *grounding_of_cls* D
using *subst_cls_eq_grounding_of_cls_subset_eq* **by** *simp*
then have *grounding_of_state* $(N, P, Q \cup \{C\})$ = *grounding_of_state* (N, P, Q)
using *D_p* **unfolding** *clss_of_state_def grounding_of_clss_def* **by** *auto*
then show *?case*
by (*auto intro*: *sr_ext.derive.intros*)
next
assume *a*: $D \cdot \sigma \subset\# C$
have *grounding_of_cls* $C \subseteq$ *sr.Rf* (*grounding_of_state* (N, P, Q))
proof
fix $C\mu$
assume $C\mu \in$ *grounding_of_cls* C
then obtain μ **where**
μ*_p*: $C\mu = C \cdot \mu \land$ *is_ground_subst* μ
unfolding *grounding_of_cls_def* **by** *auto*
have $D\sigma\mu C\mu$: $D \cdot \sigma \cdot \mu \subset\# C \cdot \mu$
using *a subst_subset_mono* **by** *auto*
then show $C\mu \in$ *sr.Rf* (*grounding_of_state* (N, P, Q))
using μ*_p strict_subset_subsumption_redundant_state*[*of* D $\sigma \odot \mu$ $C \cdot \mu$ (N, P, Q)] *D_p*
unfolding *clss_of_state_def* **by** *auto*
qed
then show *?case*
unfolding *clss_of_state_def grounding_of_clss_def* **by** (*force intro*: *sr_ext.derive.intros*)
qed
next
case (*forward_reduction* D L' P Q L σ C N)
note DL'*_p* = *this*
{
fix $C\mu$
assume $C\mu \in$ *grounding_of_cls* C
then obtain μ **where**
μ*_p*: $C\mu = C \cdot \mu \land$ *is_ground_subst* μ
unfolding *grounding_of_cls_def* **by** *auto*

define γ **where**
γ = *Infer* $\{\#(C + \{\#L\#\}) \cdot \mu\#\}$ $((D + \{\#L'\#\}) \cdot \sigma \cdot \mu)$ $(C \cdot \mu)$

have $(D + \{\#L'\#\}) \cdot \sigma \cdot \mu \in$ *grounding_of_state* $(N \cup \{C + \{\#L\#\}\}, P, Q)$
unfolding *clss_of_state_def grounding_of_clss_def grounding_of_cls_def*
by (*rule UN_I*[*of* $D + \{\#L'\#\}$], *use* DL'*_p*(*1*) **in** *simp*,
metis (*mono_tags*, *lifting*) μ*_p is_ground_comp_subst mem_Collect_eq subst_cls_comp_subst*)
moreover have $(C + \{\#L\#\}) \cdot \mu \in$ *grounding_of_state* $(N \cup \{C + \{\#L\#\}\}, P, Q)$
using μ*_p* **unfolding** *clss_of_state_def grounding_of_clss_def grounding_of_cls_def* **by** *auto*
moreover have $\forall I.\ I \models D \cdot \sigma \cdot \mu + \{\#- (L \ \cdot l\ \mu)\#\} \longrightarrow I \models C\ \cdot \mu + \{\#L\ \cdot l\ \mu\#\} \longrightarrow I \models D \cdot \sigma \cdot \mu + C$
$\cdot\ \mu$
by *auto*
then have $\forall I.\ I \models (D + \{\#L'\#\}) \cdot \sigma \cdot \mu \longrightarrow I \models (C + \{\#L\#\}) \cdot \mu \longrightarrow I \models D \cdot \sigma \cdot \mu + C \cdot \mu$
using DL'*_p*
by (*metis add_mset_add_single subst_cls_add_mset subst_cls_union subst_minus*)
then have $\forall I.\ I \models (D + \{\#L'\#\}) \cdot \sigma \cdot \mu \longrightarrow I \models (C + \{\#L\#\}) \cdot \mu \longrightarrow I \models C \cdot \mu$
using DL'*_p* **by** (*metis* (*no_types*, *lifting*) *subset_mset.le_iff_add subst_cls_union true_cls_union*)
then have $\forall I.\ I \models m\ \{\#(D + \{\#L'\#\}) \cdot \sigma \cdot \mu\#\} \longrightarrow I \models (C + \{\#L\#\}) \cdot \mu \longrightarrow I \models C \cdot \mu$
by (*meson true_cls_mset_singleton*)

ultimately have $\gamma \in$ *sr_ext.inferences_from* (*grounding_of_state* ($N \cup \{C + \{\#L\#\}\}, P, Q$))
unfolding *sr_ext.inferences_from_def* **unfolding** *ground_sound_Γ_def infer_from_def γ_def* **by** *auto*
then have $C \cdot \mu \in$ *concls_of* (*sr_ext.inferences_from* (*grounding_of_state* ($N \cup \{C + \{\#L\#\}\}, P, Q$)))
using *image_iff* **unfolding** *γ_def* **by** *fastforce*
then have $C\mu \in$ *concls_of* (*sr_ext.inferences_from* (*grounding_of_state* ($N \cup \{C + \{\#L\#\}\}, P, Q$)))
using *μ_p* **by** *auto*
}
then have *grounding_of_state* ($N \cup \{C\}, P, Q$) $-$ *grounding_of_state* ($N \cup \{C + \{\#L\#\}\}, P, Q$)
$\subseteq$ *concls_of* (*sr_ext.inferences_from* (*grounding_of_state* ($N \cup \{C + \{\#L\#\}\}, P, Q$)))
unfolding *grounding_of_clss_def clss_of_state_def* **by** *auto*
moreover
{
fix $CL\mu$
assume $CL\mu \in$ *grounding_of_cls* ($C + \{\#L\#\}$)
then obtain μ **where**
μ_def: $CL\mu = (C + \{\#L\#\}) \cdot \mu \wedge$ *is_ground_subst* μ
unfolding *grounding_of_cls_def* **by** *auto*
have *Cμ_CLμ*: $C \cdot \mu \subset\# (C + \{\#L\#\}) \cdot \mu$
by *auto*
then have $(C + \{\#L\#\}) \cdot \mu \in$ *sr.Rf* (*grounding_of_state* ($N \cup \{C\}, P, Q$))
using *sr.Rf_def*[*of grounding_of_cls C*]
using *strict_subset_subsumption_redundant_state*[*of* $C\ \mu\ (C + \{\#L\#\}) \cdot \mu\ (N \cup \{C\}, P, Q)$] *μ_def*
unfolding *clss_of_state_def* **by** *force*
then have $CL\mu \in$ *sr.Rf* (*grounding_of_state* ($N \cup \{C\}, P, Q$))
using *μ_def* **by** *auto*
}
then have *grounding_of_state* ($N \cup \{C + \{\#L\#\}\}, P, Q$) $-$ *grounding_of_state* ($N \cup \{C\}, P, Q$)
$\subseteq$ *sr.Rf* (*grounding_of_state* ($N \cup \{C\}, P, Q$))
unfolding *clss_of_state_def grounding_of_clss_def* **by** *auto*
ultimately show *?case*
using *sr_ext.derive.intros*[*of grounding_of_state* ($N \cup \{C\}, P, Q$)
grounding_of_state ($N \cup \{C + \{\#L\#\}\}, P, Q$)]
by *auto*
next
case (*backward_reduction_P* $D\ L'\ N\ L\ \sigma\ C\ P\ Q$)
note *DL′_p* = *this*
{
fix $C\mu$
assume $C\mu \in$ *grounding_of_cls* C
then obtain μ **where**
μ_p: $C\mu = C \cdot \mu \wedge$ *is_ground_subst* μ
unfolding *grounding_of_cls_def* **by** *auto*

define γ **where**
$\gamma =$ *Infer* $\{\#(C + \{\#L\#\})\cdot \mu\#\}\ ((D + \{\#L'\#\}) \cdot \sigma \cdot \mu)\ (C \cdot \mu)$

have $(D + \{\#L'\#\}) \cdot \sigma \cdot \mu \in$ *grounding_of_state* ($N, P \cup \{C + \{\#L\#\}\}, Q$)
unfolding *clss_of_state_def grounding_of_clss_def grounding_of_cls_def*
by (*rule UN_I*[*of* $D + \{\#L'\#\}$], *use DL′_p*(*1*) **in** *simp*,
metis (*mono_tags*, *lifting*) *μ_p is_ground_comp_subst mem_Collect_eq subst_cls_comp_subst*)
moreover have $(C + \{\#L\#\}) \cdot \mu \in$ *grounding_of_state* ($N, P \cup \{C + \{\#L\#\}\}, Q$)
using *μ_p* **unfolding** *clss_of_state_def grounding_of_clss_def grounding_of_cls_def* **by** *auto*
moreover have $\forall I.\ I \models D \cdot \sigma \cdot \mu + \{\#-\ (L\ \cdot l\ \mu)\#\} \longrightarrow I \models C\ \cdot \mu + \{\#L\ \cdot l\ \mu\#\} \longrightarrow I \models D \cdot \sigma \cdot \mu + C$
$\cdot\ \mu$
by *auto*
then have $\forall I.\ I \models (D + \{\#L'\#\}) \cdot \sigma \cdot \mu \longrightarrow I \models (C + \{\#L\#\}) \cdot \mu \longrightarrow I \models D \cdot \sigma \cdot \mu + C \cdot \mu$
using *DL′_p*
by (*metis add_mset_add_single subst_cls_add_mset subst_cls_union subst_minus*)
then have $\forall I.\ I \models (D + \{\#L'\#\}) \cdot \sigma \cdot \mu \longrightarrow I \models (C + \{\#L\#\}) \cdot \mu \longrightarrow I \models C \cdot \mu$
using *DL′_p* **by** (*metis* (*no_types*, *lifting*) *subset_mset.le_iff_add subst_cls_union true_cls_union*)
then have $\forall I.\ I \models m\ \{\#(D + \{\#L'\#\}) \cdot \sigma \cdot \mu\#\} \longrightarrow I \models (C + \{\#L\#\}) \cdot \mu \longrightarrow\ I \models C \cdot \mu$
by (*meson true_cls_mset_singleton*)
ultimately have $\gamma \in$ *sr_ext.inferences_from* (*grounding_of_state* ($N, P \cup \{C + \{\#L\#\}\}, Q$))

unfolding *sr_ext.inferences_from_def* **unfolding** *ground_sound_Γ_def infer_from_def γ_def* **by** *simp*
then have $C \cdot \mu \in$ *concls_of* (*sr_ext.inferences_from* (*grounding_of_state* (N, $P \cup \{C + \{\#L\#\}\}$, Q)))
using *image_iff* **unfolding** *γ_def* **by** *fastforce*
then have $C\mu \in$ *concls_of* (*sr_ext.inferences_from* (*grounding_of_state* (N, $P \cup \{C + \{\#L\#\}\}$, Q)))
using *μ_p* **by** *auto*
}
then have *grounding_of_state* (N, $P \cup \{C\}$, Q) − *grounding_of_state* (N, $P \cup \{C + \{\#L\#\}\}$, Q)
⊆ *concls_of* (*sr_ext.inferences_from* (*grounding_of_state* (N, $P \cup \{C + \{\#L\#\}\}$, Q)))
unfolding *grounding_of_clss_def clss_of_state_def* **by** *auto*
moreover
{
fix *CLμ*
assume $CL\mu \in$ *grounding_of_cls* ($C + \{\#L\#\}$)
then obtain μ **where**
μ_def: $CL\mu = (C + \{\#L\#\}) \cdot \mu \wedge$ *is_ground_subst* μ
unfolding *grounding_of_cls_def* **by** *auto*
have *Cμ_CLμ*: $C \cdot \mu \subset\# (C + \{\#L\#\}) \cdot \mu$
by *auto*
then have $(C + \{\#L\#\}) \cdot \mu \in$ *sr.Rf* (*grounding_of_state* (N, $P \cup \{C\}$, Q))
using *sr.Rf_def*[*of grounding_of_cls C*]
using *strict_subset_subsumption_redundant_state*[*of C μ* ($C + \{\#L\#\}) \cdot \mu$ (N, $P \cup \{C\}$, Q)] *μ_def*
unfolding *clss_of_state_def* **by** *force*
then have $CL\mu \in$ *sr.Rf* (*grounding_of_state* (N, $P \cup \{C\}$, Q))
using *μ_def* **by** *auto*
}
then have *grounding_of_state* (N, $P \cup \{C + \{\#L\#\}\}$, Q) − *grounding_of_state* (N, $P \cup \{C\}$, Q)
⊆ *sr.Rf* (*grounding_of_state* (N, $P \cup \{C\}$, Q))
unfolding *clss_of_state_def grounding_of_clss_def* **by** *auto*
ultimately show *?case*
using *sr_ext.derive.intros*[*of grounding_of_state* (N, $P \cup \{C\}$, Q)
grounding_of_state (N, $P \cup \{C + \{\#L\#\}\}$, Q)]
by *auto*
next
case (*backward_reduction_Q D L' N L σ C P Q*)
note *DL'_p* = *this*
{
fix *Cμ*
assume $C\mu \in$ *grounding_of_cls C*
then obtain μ **where**
μ_p: $C\mu = C \cdot \mu \wedge$ *is_ground_subst* μ
unfolding *grounding_of_cls_def* **by** *auto*

define γ **where**
$\gamma =$ *Infer* $\{\#(C + \{\#L\#\})\cdot \mu\#\}$ $((D + \{\#L'\#\}) \cdot \sigma \cdot \mu)$ $(C \cdot \mu)$

have $(D + \{\#L'\#\}) \cdot \sigma \cdot \mu \in$ *grounding_of_state* (N, P, $Q \cup \{C + \{\#L\#\}\}$)
unfolding *clss_of_state_def grounding_of_clss_def grounding_of_cls_def*
by (*rule UN_I*[*of* $D + \{\#L'\#\}$], *use DL'_p*(*1*) **in** *simp*,
metis (*mono_tags*, *lifting*) *μ_p is_ground_comp_subst mem_Collect_eq subst_cls_comp_subst*)
moreover have $(C + \{\#L\#\}) \cdot \mu \in$ *grounding_of_state* (N, P, $Q \cup \{C + \{\#L\#\}\}$)
using *μ_p* **unfolding** *clss_of_state_def grounding_of_clss_def grounding_of_cls_def* **by** *auto*
moreover have $\forall I.\ I \models D \cdot \sigma \cdot \mu + \{\#-\ (L\ \cdot l\ \mu)\#\} \longrightarrow I \models C\ \cdot \mu + \{\#L\ \cdot l\ \mu\#\} \longrightarrow I \models D \cdot \sigma \cdot \mu + C \cdot \mu$
by *auto*
then have $\forall I.\ I \models (D + \{\#L'\#\}) \cdot \sigma \cdot \mu \longrightarrow I \models (C + \{\#L\#\}) \cdot \mu \longrightarrow I \models D \cdot \sigma \cdot \mu + C \cdot \mu$
using *DL'_p*
by (*metis add_mset_add_single subst_cls_add_mset subst_cls_union subst_minus*)
then have $\forall I.\ I \models (D + \{\#L'\#\}) \cdot \sigma \cdot \mu \longrightarrow I \models (C + \{\#L\#\}) \cdot \mu \longrightarrow I \models C \cdot \mu$
using *DL'_p* **by** (*metis* (*no_types*, *lifting*) *subset_mset.le_iff_add subst_cls_union true_cls_union*)
then have $\forall I.\ I \models m\ \{\#(D + \{\#L'\#\}) \cdot \sigma \cdot \mu\#\} \longrightarrow I \models (C + \{\#L\#\}) \cdot \mu \longrightarrow I \models C \cdot \mu$
by (*meson true_cls_mset_singleton*)
ultimately have $\gamma \in$ *sr_ext.inferences_from* (*grounding_of_state* (N, P, $Q \cup \{C + \{\#L\#\}\}$))
unfolding *sr_ext.inferences_from_def* **unfolding** *ground_sound_Γ_def infer_from_def γ_def* **by** *simp*

```
    then have C · μ ∈ concls_of (sr_ext.inferences_from (grounding_of_state (N, P, Q ∪ {C + {#L#}})))
      using image_iff unfolding γ_def by fastforce
    then have Cμ ∈ concls_of (sr_ext.inferences_from (grounding_of_state (N, P, Q ∪ {C + {#L#}})))
      using μ_p by auto
  }
  then have grounding_of_state (N, P ∪ {C}, Q) − grounding_of_state (N, P, Q ∪ {C + {#L#}})
    ⊆ concls_of (sr_ext.inferences_from (grounding_of_state (N, P, Q ∪ {C + {#L#}})))
    unfolding grounding_of_clss_def clss_of_state_def by auto
  moreover
  {
    fix CLμ
    assume CLμ ∈ grounding_of_cls (C + {#L#})
    then obtain μ where
      μ_def: CLμ = (C + {#L#}) · μ ∧ is_ground_subst μ
      unfolding grounding_of_cls_def by auto
    have Cμ_CLμ: C · μ ⊂# (C + {#L#}) · μ
      by auto
    then have (C + {#L#}) · μ ∈ sr.Rf (grounding_of_state (N, P∪ {C}, Q))
      using sr.Rf_def[of grounding_of_cls C]
      using strict_subset_subsumption_redundant_state[of C μ (C + {#L#}) · μ (N, P ∪ {C}, Q)] μ_def
      unfolding clss_of_state_def by force
    then have CLμ ∈ sr.Rf (grounding_of_state (N, P ∪ {C}, Q))
      using μ_def by auto
  }
  then have grounding_of_state (N, P , Q ∪ {C + {#L#}}) − grounding_of_state (N, P  ∪ {C}, Q)
    ⊆ sr.Rf (grounding_of_state (N, P ∪ {C}, Q))
    unfolding clss_of_state_def grounding_of_clss_def by auto
  ultimately show ?case
    using sr_ext.derive.intros[of grounding_of_state (N, P ∪ {C}, Q)
        grounding_of_state (N, P, Q ∪ {C + {#L#}})]
    by auto
next
  case (clause_processing N C P Q)
  then show ?case
    unfolding clss_of_state_def using sr_ext.derive.intros by auto
next
  case (inference_computation N Q C P)
  {
    fix Eμ
    assume Eμ ∈ grounding_of_clss N
    then obtain μ E where
      E_μ_p: Eμ = E · μ ∧ E ∈ N ∧ is_ground_subst μ
      unfolding grounding_of_clss_def grounding_of_cls_def by auto
    then have E_concl: E ∈ concls_of (ord_FO_resolution.inferences_between Q C)
      using inference_computation by auto
    then obtain γ where
      γ_p: γ ∈ ord_FO_Γ S ∧ infer_from (Q ∪ {C}) γ ∧ C ∈# prems_of γ ∧ concl_of γ = E
      unfolding ord_FO_resolution.inferences_between_def by auto
    then obtain CC CAs D AAs As σ where
      γ_p2: γ = Infer CC D E ∧ ord_resolve_rename S CAs D AAs As σ E ∧ mset CAs = CC
      unfolding ord_FO_Γ_def by auto
    define ϱ where
      ϱ = hd (renamings_apart (D # CAs))
    define ϱs where
      ϱs = tl (renamings_apart (D # CAs))
    define γ_ground where
      γ_ground = Infer (mset (CAs ··cl ϱs) ·cm σ ·cm μ) (D · ϱ · σ · μ) (E · μ)
    have ∀I. I ⊨m mset (CAs ··cl ϱs) ·cm σ ·cm μ ⟶ I ⊨ D · ϱ · σ · μ ⟶ I ⊨ E · μ
      using ord_resolve_rename_ground_inst_sound[of _ _ _ _ _ _ _ _ _ _ μ] ϱ_def ϱs_def E_μ_p γ_p2
      by auto
    then have γ_ground ∈ {Infer cc d e | cc d e. ∀I. I ⊨m cc ⟶ I ⊨ d ⟶ I ⊨ e}
      unfolding γ_ground_def by auto
    moreover have set_mset (prems_of γ_ground) ⊆ grounding_of_state ({}, P ∪ {C}, Q)
```

```
proof −
  have D = C ∨ D ∈ Q
    unfolding γ_ground_def using E_μ_p γ_p2 γ_p unfolding infer_from_def
    unfolding clss_of_state_def grounding_of_clss_def
    unfolding grounding_of_cls_def
    by simp
  then have D · ϱ · σ · μ ∈ grounding_of_cls C ∨ (∃ x ∈ Q. D · ϱ · σ · μ ∈ grounding_of_cls x)
    using E_μ_p
    unfolding grounding_of_cls_def
    by (metis (mono_tags, lifting) is_ground_comp_subst mem_Collect_eq subst_cls_comp_subst)
  then have (D · ϱ · σ · μ ∈ grounding_of_cls C ∨
    (∃ x ∈ P. D · ϱ · σ · μ ∈ grounding_of_cls x) ∨
    (∃ x ∈ Q. D · ϱ · σ · μ ∈ grounding_of_cls x))
    by metis
  moreover have ∀ i < length (CAs ·· cl ϱs ·cl σ ·cl μ). ((CAs ··cl ϱs ·cl σ ·cl μ) ! i) ∈
    {C · σ |σ. is_ground_subst σ} ∪
    ((⋃ C ∈ P. {C · σ | σ. is_ground_subst σ}) ∪ (⋃ C∈Q. {C · σ | σ. is_ground_subst σ}))
  proof (rule, rule)
    fix i
    assume i < length (CAs ··cl ϱs ·cl σ ·cl μ)
    then have a: i < length CAs ∧ i < length ϱs
      by simp
    moreover from a have CAs ! i ∈ {C} ∪ Q
      using γ_p2 γ_p unfolding infer_from_def
      by (metis (no_types, lifting) Un_subset_iff inference.sel(1) set_mset_union
          sup_commute nth_mem_mset subsetCE)
    ultimately have (CAs ··cl ϱs ·cl σ ·cl μ) ! i ∈
      {C · σ |σ. is_ground_subst σ} ∨
      ((CAs ··cl ϱs ·cl σ ·cl μ) ! i ∈ (⋃ C∈P. {C · σ |σ. is_ground_subst σ}) ∨
      (CAs ··cl ϱs ·cl σ ·cl μ) ! i ∈ (⋃ C ∈ Q. {C · σ | σ. is_ground_subst σ}))
      unfolding γ_ground_def using E_μ_p γ_p2 γ_p unfolding infer_from_def
      unfolding clss_of_state_def grounding_of_clss_def
      unfolding grounding_of_cls_def
      apply −
      apply (cases CAs ! i = C)
      subgoal
        apply (rule disjI1)
        apply (rule Set.CollectI)
        apply (rule_tac x=(ϱs ! i) ⊙ σ ⊙ μ in exI)
        using ϱs_def using renames_apart apply (auto;fail)
        done
      subgoal
        apply (rule disjI2)
        apply (rule disjI2)
        apply (rule_tac a=CAs ! i in UN_I)
        subgoal
          apply blast
          done
        subgoal
          apply (rule Set.CollectI)
          apply (rule_tac x=(ϱs ! i) ⊙ σ ⊙ μ in exI)
          using ϱs_def using renames_apart apply (auto;fail)
          done
        done
      done
    then show (CAs ··cl ϱs ·cl σ ·cl μ) ! i ∈ {C · σ |σ. is_ground_subst σ} ∪
      ((⋃ C ∈ P. {C · σ |σ. is_ground_subst σ}) ∪ (⋃ C ∈ Q. {C · σ |σ. is_ground_subst σ}))
      by blast
  qed
  then have ∀ x ∈# mset (CAs ··cl ϱs ·cl σ ·cl μ). x ∈ {C · σ |σ. is_ground_subst σ} ∪
    ((⋃ C ∈ P. {C · σ |σ. is_ground_subst σ}) ∪ (⋃ C ∈ Q. {C · σ |σ. is_ground_subst σ}))
    by (metis (lifting) in_set_conv_nth set_mset_mset)
  then have set_mset (mset (CAs ··cl ϱs) ·cm σ ·cm μ) ⊆
```

        grounding_of_cls C ∪ *grounding_of_clss* P ∪ *grounding_of_clss* Q
        unfolding *grounding_of_cls_def grounding_of_clss_def*
        using *mset_subst_cls_list_subst_cls_mset* **by** *auto*
      ultimately show *?thesis*
        unfolding *γ_ground_def clss_of_state_def grounding_of_clss_def* **by** *auto*
    qed
    ultimately have $E \cdot \mu \in$ *concls_of* (*sr_ext.inferences_from* (*grounding_of_state* ({}, $P \cup \{C\}$, Q)))
      unfolding *sr_ext.inferences_from_def inference_system.inferences_from_def ground_sound_Γ_def infer_from_def*
      using *γ_ground_def* **by** (*metis* (*no_types*, *lifting*) *imageI inference.sel*(*3*) *mem_Collect_eq*)
    then have $E\mu \in$ *concls_of* (*sr_ext.inferences_from* (*grounding_of_state* ({}, $P \cup \{C\}$, Q)))
      using *E_μ_p* **by** *auto*
  }
  then have *grounding_of_state* (N, P, $Q \cup \{C\}$) − *grounding_of_state* ({}, $P \cup \{C\}$, Q)
    ⊆ *concls_of* (*sr_ext.inferences_from* (*grounding_of_state* ({}, $P \cup \{C\}$, Q)))
    unfolding *clss_of_state_def grounding_of_clss_def* **by** *auto*
  moreover have *grounding_of_state* ({}, $P \cup \{C\}$, Q) − *grounding_of_state* (N, P, $Q \cup \{C\}$) = {}
    unfolding *clss_of_state_def grounding_of_clss_def* **by** *auto*
  ultimately show *?case*
    using *sr_ext.derive.intros*[*of* (*grounding_of_state* (N, P, $Q \cup \{C\}$))
        (*grounding_of_state* ({}, $P \cup \{C\}$, Q))] **by** *auto*
qed

A useful consequence:

lemma *RP_model*: $St \leadsto St' \Longrightarrow I \models s$ *grounding_of_state* $St' \longleftrightarrow I \models s$ *grounding_of_state* St
proof (*drule resolution_prover_ground_derive*, *erule sr_ext.derive.cases*, *hypsubst*)
  let
    ?gSt = *grounding_of_state* *St* **and**
    ?gSt′ = *grounding_of_state* *St′*

  assume
    deduct: *?gSt′* − *?gSt* ⊆ *concls_of* (*sr_ext.inferences_from ?gSt*) (**is** _ ⊆ *?concls*) **and**
    delete: *?gSt* − *?gSt′* ⊆ *sr.Rf ?gSt′*

  show $I \models s$ *?gSt′* ⟷ $I \models s$ *?gSt*
  proof
    assume *bef*: $I \models s$ *?gSt*
    then have $I \models s$ *?concls*
      unfolding *ground_sound_Γ_def inference_system.inferences_from_def true_clss_def true_cls_mset_def*
      by (*auto simp add*: *image_def infer_from_def dest*!: *spec*[*of* _ *I*])
    then have *diff*: $I \models s$ *?gSt′* − *?gSt*
      using *deduct* **by** (*blast intro*: *true_clss_mono*)
    then show $I \models s$ *?gSt′*
      using *bef* **unfolding** *true_clss_def* **by** *blast*
  next
    assume *aft*: $I \models s$ *?gSt′*
    have $I \models s$ *?gSt′* ∪ *sr.Rf ?gSt′*
      by (*rule sr.Rf_model*) (*metis aft sr.Rf_mono*[*OF Un_upper1*] *Diff_eq_empty_iff Diff_subset*
          Un_Diff true_clss_mono true_clss_union)
    then have $I \models s$ *sr.Rf ?gSt′*
      using *true_clss_union* **by** *blast*
    then have *diff*: $I \models s$ *?gSt* − *?gSt′*
      using *delete* **by** (*blast intro*: *true_clss_mono*)
    then show $I \models s$ *?gSt*
      using *aft* **unfolding** *true_clss_def* **by** *blast*
  qed
qed

Another formulation of the part of Lemma 4.10 that states we have a theorem proving process:

lemma *resolution_prover_ground_derivation*:
  chain (*op* ⇝) *Sts* ⟹ *chain sr_ext.derive* (*lmap grounding_of_state Sts*)
  using *resolution_prover_ground_derive* **by** (*simp add*: *chain_lmap*[*of op* ⇝])

The following is used prove to Lemma 4.11:

```
lemma in_Sup_llist_in_nth: C ∈ Sup_llist Ns ⟹ ∃j. enat j < llength Ns ∧ C ∈ lnth Ns j
  unfolding Sup_llist_def by auto

lemma Sup_llist_grounding_of_state_ground:
  assumes C ∈ Sup_llist (lmap grounding_of_state Sts)
  shows is_ground_cls C
proof −
  have ∃j. enat j < llength (lmap grounding_of_state Sts) ∧ C ∈ lnth (lmap grounding_of_state Sts) j
    using assms in_Sup_llist_in_nth by metis
  then obtain j where
    enat j < llength (lmap grounding_of_state Sts)
    C ∈ lnth (lmap grounding_of_state Sts) j
    by blast
  then show ?thesis
    unfolding grounding_of_clss_def grounding_of_cls_def by auto
qed

lemma Liminf_grounding_of_state_ground:
  C ∈ Liminf_llist (lmap grounding_of_state Sts) ⟹ is_ground_cls C
  using Liminf_llist_subset_Sup_llist[of lmap grounding_of_state Sts]
    Sup_llist_grounding_of_state_ground
  by blast

lemma in_Sup_llist_in_Sup_state:
  assumes C ∈ Sup_llist (lmap grounding_of_state Sts)
  shows ∃D σ. D ∈ clss_of_state (Sup_state Sts) ∧ D · σ = C ∧ is_ground_subst σ
proof −
  from assms obtain i where
    i_p: enat i < llength Sts ∧ C ∈ lnth (lmap grounding_of_state Sts) i
    using in_Sup_llist_in_nth by fastforce
  then obtain D σ where
    D ∈ clss_of_state (lnth Sts i) ∧ D · σ = C ∧ is_ground_subst σ
    using assms unfolding grounding_of_clss_def grounding_of_cls_def by fastforce
  then have D ∈ clss_of_state (Sup_state Sts) ∧ D · σ = C ∧ is_ground_subst σ
    using i_p unfolding Sup_state_def clss_of_state_def
    by (metis (no_types, lifting) UnCI UnE contra_subsetD N_of_state.simps P_of_state.simps
        Q_of_state.simps llength_lmap lnth_lmap lnth_subset_Sup_llist)
  then show ?thesis
    by auto
qed

lemma
  N_of_state_Liminf: N_of_state (Liminf_state Sts) = Liminf_llist (lmap N_of_state Sts) and
  P_of_state_Liminf: P_of_state (Liminf_state Sts) = Liminf_llist (lmap P_of_state Sts)
  unfolding Liminf_state_def by auto

lemma eventually_removed_from_N:
  assumes
    d_in: D ∈ N_of_state (lnth Sts i) and
    fair: fair_state_seq Sts and
    i_Sts: enat i < llength Sts
  shows ∃l. D ∈ N_of_state (lnth Sts l) ∧ D ∉ N_of_state (lnth Sts (Suc l)) ∧ i ≤ l ∧ enat (Suc l) < llength Sts
proof (rule ccontr)
  assume a: ¬ ?thesis
  have i ≤ l ⟹ enat l < llength Sts ⟹ D ∈ N_of_state (lnth Sts l) for l
    using d_in by (induction l, blast, metis a Suc_ile_eq le_SucE less_imp_le)
  then have D ∈ Liminf_llist (lmap N_of_state Sts)
    unfolding Liminf_llist_def using i_Sts by auto
  then show False
    using fair unfolding fair_state_seq_def by (simp add: N_of_state_Liminf)
qed

lemma eventually_removed_from_P:
```

assumes
d_in: $D \in$ *P_of_state* (*lnth Sts i*) **and**
fair: *fair_state_seq Sts* **and**
i_Sts: *enat i* $<$ *llength Sts*
shows $\exists\, l.\ D \in$ *P_of_state* (*lnth Sts l*) $\wedge$ $D \notin$ *P_of_state* (*lnth Sts* (*Suc l*)) $\wedge$ $i \le l$ $\wedge$ *enat* (*Suc l*) $<$ *llength Sts*
proof (*rule ccontr*)
assume *a*: $\neg$ *?thesis*
have $i \le l \Longrightarrow$ *enat l* $<$ *llength Sts* $\Longrightarrow D \in$ *P_of_state* (*lnth Sts l*) **for** *l*
using *d_in* **by** (*induction l*, *blast*, *metis a Suc_ile_eq le_SucE less_imp_le*)
then have $D \in$ *Liminf_llist* (*lmap P_of_state Sts*)
unfolding *Liminf_llist_def* **using** *i_Sts* **by** *auto*
then show *False*
using *fair* **unfolding** *fair_state_seq_def* **by** (*simp add*: *P_of_state_Liminf*)
qed

lemma *instance_if_subsumed_and_in_limit*:
assumes
ns: *Ns* = *lmap grounding_of_state Sts* **and**
c: $C \in$ *Liminf_llist Ns* $-$ *sr.Rf* (*Liminf_llist Ns*) **and**

d: $D \in$ *N_of_state* (*lnth Sts i*) $\cup$ *P_of_state* (*lnth Sts i*) $\cup$ *Q_of_state* (*lnth Sts i*)
enat i $<$ *llength Sts subsumes D C*
shows $\exists\, \sigma.\ D \cdot \sigma = C \wedge$ *is_ground_subst* σ
proof $-$
let *?Ps* = $\lambda i.$ *P_of_state* (*lnth Sts i*)
let *?Qs* = $\lambda i.$ *Q_of_state* (*lnth Sts i*)

have *ground_C*: *is_ground_cls C*
using *c* **using** *Liminf_grounding_of_state_ground ns* **by** *auto*

have *derivns*: *chain sr_ext.derive Ns*
using *resolution_prover_ground_derivation deriv ns* **by** *auto*

have $\exists\, \sigma.\ D \cdot \sigma = C$
proof (*rule ccontr*)
assume $\nexists\, \sigma.\ D \cdot \sigma = C$
moreover from *d*(*3*) **obtain** τ*_proto* **where**
$D \cdot \tau$*_proto* $\subseteq\#$ C **unfolding** *subsumes_def*
by *blast*
then obtain τ **where**
τ*_p*: $D \cdot \tau \subseteq\# C \wedge$ *is_ground_subst* τ
using *ground_C* **by** (*metis is_ground_cls_mono make_ground_subst subset_mset.order_refl*)
ultimately have *subsub*: $D \cdot \tau \subset\# C$
using *subset_mset.le_imp_less_or_eq* **by** *auto*
moreover have *is_ground_subst* τ
using τ*_p* **by** *auto*
moreover have $D \in$ *clss_of_state* (*lnth Sts i*)
using *d* **unfolding** *clss_of_state_def* **by** *auto*
ultimately have $C \in$ *sr.Rf* (*grounding_of_state* (*lnth Sts i*))
using *strict_subset_subsumption_redundant_state*[*of D* τ *C lnth Sts i*] **by** *auto*
then have $C \in$ *sr.Rf* (*Sup_llist Ns*)
using *d ns* **by** (*metis contra_subsetD llength_lmap lnth_lmap lnth_subset_Sup_llist sr.Rf_mono*)
then have $C \in$ *sr.Rf* (*Liminf_llist Ns*)
unfolding *ns* **using** *local.sr_ext.Rf_Sup_subset_Rf_Liminf derivns ns* **by** *auto*
then show *False*
using *c* **by** *auto*
qed
then obtain σ **where**
$D \cdot \sigma = C \wedge$ *is_ground_subst* σ
using *ground_C* **by** (*metis make_ground_subst*)
then show *?thesis*
by *auto*

```
qed

lemma from_Q_to_Q_inf:
  assumes
    fair: fair_state_seq Sts and
    ns: Ns = lmap grounding_of_state Sts and
    c: C ∈ Liminf_llist Ns − sr.Rf (Liminf_llist Ns) and
    d: D ∈ Q_of_state (lnth Sts i) enat i < llength Sts subsumes D C and
    d_least: ∀ E ∈ {E. E ∈ (clss_of_state (Sup_state Sts)) ∧ subsumes E C}. ¬ strictly_subsumes E D
  shows D ∈ Q_of_state (Liminf_state Sts)
proof −
  let ?Ps = λi. P_of_state (lnth Sts i)
  let ?Qs = λi. Q_of_state (lnth Sts i)

  have ground_C: is_ground_cls C
    using c using Liminf_grounding_of_state_ground ns by auto

  have derivns: chain sr_ext.derive Ns
    using resolution_prover_ground_derivation deriv ns by auto

  have ∃σ. D · σ = C ∧ is_ground_subst σ
    using instance_if_subsumed_and_in_limit ns c d by blast
  then obtain σ where
    σ: D · σ = C is_ground_subst σ
    by auto

  from deriv have four_ten: chain sr_ext.derive Ns
    using resolution_prover_ground_derivation ns by auto

  have in_Sts_in_Sts_Suc:
    ∀ l ≥ i. enat (Suc l) < llength Sts ⟶ D ∈ Q_of_state (lnth Sts l) ⟶ D ∈ Q_of_state (lnth Sts (Suc l))
  proof (rule, rule, rule, rule)
    fix l
    assume
      len: i ≤ l and
      llen: enat (Suc l) < llength Sts and
      d_in_q: D ∈ Q_of_state (lnth Sts l)

    have lnth Sts l ↝ lnth Sts (Suc l)
      using llen deriv chain_lnth_rel by blast
    then show D ∈ Q_of_state (lnth Sts (Suc l))
    proof (cases rule: RP.cases)
      case (backward_subsumption_Q D' N D_removed P Q)
      moreover
      {
        assume D_removed = D
        then obtain D_subsumes where
          D_subsumes_p: D_subsumes ∈ N ∧ strictly_subsumes D_subsumes D
          using backward_subsumption_Q by auto
        moreover from D_subsumes_p have subsumes D_subsumes C
          using d subsumes_trans unfolding strictly_subsumes_def by blast
        moreover from backward_subsumption_Q have D_subsumes ∈ clss_of_state (Sup_state Sts)
          using D_subsumes_p llen
          by (metis (no_types) UnI1 clss_of_state_def N_of_state.simps llength_lmap lnth_lmap
              lnth_subset_Sup_llist rev_subsetD Sup_state_def)
        ultimately have False
          using d_least unfolding subsumes_def by auto
      }
      ultimately show ?thesis
        using d_in_q by auto
    next
      case (backward_reduction_Q E L' N L σ D' P Q)
      {
```

```
      assume D' + {#L#} = D
      then have D'_p: strictly_subsumes D' D ∧ D' ∈ ?Ps (Suc l)
        using subset_strictly_subsumes[of D' D] backward_reduction_Q by auto
      then have subc: subsumes D' C
        using d(3) subsumes_trans unfolding strictly_subsumes_def by auto
      from D'_p have D' ∈ clss_of_state (Sup_state Sts)
        using llen by (metis (no_types) UnI1 clss_of_state_def P_of_state.simps llength_lmap
            lnth_lmap lnth_subset_Sup_llist subsetCE sup_ge2 Sup_state_def)
      then have False
        using d_least D'_p subc by auto
    }
    then show ?thesis
      using backward_reduction_Q d_in_q by auto
  qed (use d_in_q in auto)
 qed
 have D_in_Sts: D ∈ Q_of_state (lnth Sts l) and D_in_Sts_Suc: D ∈ Q_of_state (lnth Sts (Suc l))
   if l_i: l ≥ i and enat: enat (Suc l) < llength Sts for l
 proof −
   show D ∈ Q_of_state (lnth Sts l)
     using l_i enat
     apply (induction l − i arbitrary: l)
     subgoal using d by auto
     subgoal using d(1) in_Sts_in_Sts_Suc
       by (metis (no_types, lifting) Suc_ile_eq add_Suc_right add_diff_cancel_left' le_SucE
           le_Suc_ex less_imp_le)
     done
   then show D ∈ Q_of_state (lnth Sts (Suc l))
     using l_i enat in_Sts_in_Sts_Suc by blast
 qed
 have i ≤ x ⟹ enat x < llength Sts ⟹ D ∈ Q_of_state (lnth Sts x) for x
   apply (cases x)
   subgoal using d(1) by (auto intro!: exI[of _ i] simp: less_Suc_eq)
   subgoal for x'
     using d(1) D_in_Sts_Suc[of x'] by (cases ‹i ≤ x'›) (auto simp: not_less_eq_eq)
   done
 then have D ∈ Liminf_llist (lmap Q_of_state Sts)
   unfolding Liminf_llist_def by (auto intro!: exI[of _ i] simp: d)
 then show ?thesis
   unfolding Liminf_state_def by auto
qed

lemma from_P_to_Q:
 assumes
   fair: fair_state_seq Sts and
   ns: Ns = lmap grounding_of_state Sts and
   c: C ∈ Liminf_llist Ns − sr.Rf (Liminf_llist Ns) and
   d: D ∈ P_of_state (lnth Sts i) enat i < llength Sts subsumes D C and
   d_least: ∀ E ∈ {E. E ∈ (clss_of_state (Sup_state Sts)) ∧ subsumes E C}. ¬ strictly_subsumes E D
 shows ∃ l. D ∈ Q_of_state (lnth Sts l) ∧ enat l < llength Sts
proof −
 let ?Ns = λi. N_of_state (lnth Sts i)
 let ?Ps = λi. P_of_state (lnth Sts i)
 let ?Qs = λi. Q_of_state (lnth Sts i)

 have ground_C: is_ground_cls C
   using c using Liminf_grounding_of_state_ground ns by auto

 have derivns: chain sr_ext.derive Ns
   using resolution_prover_ground_derivation deriv ns by auto

 have ∃ σ. D · σ = C ∧ is_ground_subst σ
   using instance_if_subsumed_and_in_limit ns c d by blast
 then obtain σ where
```

σ: $D \cdot \sigma = C$ *is_ground_subst* σ
by *auto*

from *deriv* **have** *four_ten*: *chain sr_ext.derive Ns*
using *resolution_prover_ground_derivation ns* **by** *auto*

obtain l **where**
l_p: $D \in$ *P_of_state* (*lnth Sts l*) $\wedge$ $D \notin$ *P_of_state* (*lnth Sts* (*Suc l*)) $\wedge$ $i \leq l$ $\wedge$ *enat* (*Suc l*) $<$ *llength Sts*
using *fair* **using** *eventually_removed_from_P d* **unfolding** *ns* **by** *auto*
then have *l_Ns*: *enat* (*Suc l*) $<$ *llength Ns*
using *ns* **by** *auto*
from *l_p* **have** *lnth Sts l* $\rightsquigarrow$ *lnth Sts* (*Suc l*)
using *deriv* **using** *chain_lnth_rel* **by** *auto*
then show *?thesis*
proof (*cases rule*: *RP.cases*)
case (*backward_subsumption_P* D' *N D_twin P Q*)
note *lrhs* = *this*(*1*,*2*) **and** *D'_p* = *this*(*3*,*4*)
then have *twins*: *D_twin* = *D ?Ns* (*Suc l*) = *N ?Ns l* = *N* *?Ps* (*Suc l*) = *P*
?Ps l = *P* $\cup$ {*D_twin*} *?Qs* (*Suc l*) = *Q ?Qs l* = *Q*
using *l_p* **by** *auto*
note *D'_p* = *D'_p*[*unfolded twins*(*1*)]
then have *subc*: *subsumes* D' *C*
unfolding *strictly_subsumes_def subsumes_def* **using** σ
by (*metis subst_cls_comp_subst subst_cls_mono_mset*)
from *D'_p* **have** $D' \in$ *clss_of_state* (*Sup_state Sts*)
unfolding *twins*(*2*)[*symmetric*] **using** *l_p*
by (*metis* (*no_types*) *UnI1 clss_of_state_def N_of_state.simps llength_lmap lnth_lmap lnth_subset_Sup_llist subsetCE Sup_state_def*)
then have *False*
using *d_least D'_p subc* **by** *auto*
then show *?thesis*
by *auto*
next
case (*backward_reduction_P E* L' *N L* σ D' *P Q*)
then have *twins*: D' + {#*L*#} = *D ?Ns* (*Suc l*) = *N ?Ns l* = *N* *?Ps* (*Suc l*) = *P* $\cup$ {D'}
?Ps l = *P* $\cup$ {D' + {#*L*#}} *?Qs* (*Suc l*) = *Q ?Qs l* = *Q*
using *l_p* **by** *auto*
then have *D'_p*: *strictly_subsumes* D' *D* $\wedge$ $D' \in$ *?Ps* (*Suc l*)
using *subset_strictly_subsumes*[*of* D' *D*] **by** *auto*
then have *subc*: *subsumes* D' *C*
using *d*(*3*) *subsumes_trans* **unfolding** *strictly_subsumes_def* **by** *auto*
from *D'_p* **have** $D' \in$ *clss_of_state* (*Sup_state Sts*)
using *l_p* **by** (*metis* (*no_types*) *UnI1 clss_of_state_def P_of_state.simps llength_lmap lnth_lmap lnth_subset_Sup_llist subsetCE sup_ge2 Sup_state_def*)
then have *False*
using *d_least D'_p subc* **by** *auto*
then show *?thesis*
by *auto*
next
case (*inference_computation N Q D_twin P*)
then have *twins*: *D_twin* = *D ?Ps* (*Suc l*) = *P ?Ps l* = *P* $\cup$ {*D_twin*}
?Qs (*Suc l*) = *Q* $\cup$ {*D_twin*} *?Qs l* = *Q*
using *l_p* **by** *auto*
then show *?thesis*
using *d* σ *l_p* **by** *auto*
qed (*use l_p* **in** *auto*)
qed

lemma *variants_sym*: *variants D* D' $\longleftrightarrow$ *variants* D' *D*
unfolding *variants_def* **by** *auto*

lemma *variants_imp_exists_subtitution*: *variants D* D' $\Longrightarrow$ $\exists \sigma.\ D \cdot \sigma = D'$
unfolding *variants_iff_subsumes subsumes_def*

by (*meson strictly_subsumes_def subset_mset_def strict_subset_subst_strictly_subsumes subsumes_def*)

lemma *properly_subsume_variants*:
assumes *strictly_subsumes* E D **and** *variants* D D'
shows *strictly_subsumes* E D'
proof –
from *assms* **obtain** σ σ' **where**
σ_σ'_p: $D \cdot \sigma = D' \wedge D' \cdot \sigma' = D$
using *variants_imp_exists_subtitution variants_sym* **by** *metis*

from *assms* **obtain** σ'' **where**
$E \cdot \sigma'' \subseteq\# D$
unfolding *strictly_subsumes_def subsumes_def* **by** *auto*
then have $E \cdot \sigma'' \cdot \sigma \subseteq\# D \cdot \sigma$
using *subst_cls_mono_mset* **by** *blast*
then have $E \cdot (\sigma'' \odot \sigma) \subseteq\# D'$
using *σ_σ'_p* **by** *auto*
moreover from *assms* **have** *n*: $(\nexists \sigma.\ D \cdot \sigma \subseteq\# E)$
unfolding *strictly_subsumes_def subsumes_def* **by** *auto*
have $\nexists \sigma.\ D' \cdot \sigma \subseteq\# E$
proof
assume $\exists \sigma'''.\ D' \cdot \sigma''' \subseteq\# E$
then obtain σ''' **where**
$D' \cdot \sigma''' \subseteq\# E$
by *auto*
then have $D \cdot (\sigma \odot \sigma''') \subseteq\# E$
using *σ_σ'_p* **by** *auto*
then show *False*
using *n* **by** *metis*
qed
ultimately show *?thesis*
unfolding *strictly_subsumes_def subsumes_def* **by** *metis*
qed

lemma *neg_properly_subsume_variants*:
assumes $\neg$ *strictly_subsumes* E D **and** *variants* D D'
shows $\neg$ *strictly_subsumes* E D'
using *assms properly_subsume_variants variants_sym* **by** *auto*

lemma *from_N_to_P_or_Q*:
assumes
fair: *fair_state_seq Sts* **and**
ns: *Ns* = *lmap grounding_of_state Sts* **and**
c: $C \in$ *Liminf_llist Ns* $-$ *sr.Rf* (*Liminf_llist Ns*) **and**
d: $D \in$ *N_of_state* (*lnth Sts i*) *enat i* < *llength Sts subsumes* D C **and**
d_least: $\forall E \in \{E.\ E \in$ (*clss_of_state* (*Sup_state Sts*)) $\wedge$ *subsumes* E $C\}$. $\neg$ *strictly_subsumes* E D
shows $\exists l\ D'\ \sigma'.\ D' \in$ *P_of_state* (*lnth Sts l*) $\cup$ *Q_of_state* (*lnth Sts l*) $\wedge$
enat l < *llength Sts* $\wedge$
($\forall E \in \{E.\ E \in$ (*clss_of_state* (*Sup_state Sts*)) $\wedge$ *subsumes* E $C\}$. $\neg$ *strictly_subsumes* E D') $\wedge$
$D' \cdot \sigma' = C \wedge$ *is_ground_subst* $\sigma' \wedge$ *subsumes* D' C
proof –
let *?Ns* = $\lambda i.$ *N_of_state* (*lnth Sts i*)
let *?Ps* = $\lambda i.$ *P_of_state* (*lnth Sts i*)
let *?Qs* = $\lambda i.$ *Q_of_state* (*lnth Sts i*)

have *ground_C*: *is_ground_cls* C
using *c* **using** *Liminf_grounding_of_state_ground ns* **by** *auto*

have *derivns*: *chain sr_ext.derive Ns*
using *resolution_prover_ground_derivation deriv ns* **by** *auto*

have $\exists \sigma.\ D \cdot \sigma = C \wedge$ *is_ground_subst* σ
using *instance_if_subsumed_and_in_limit ns c d* **by** *blast*

```
then obtain σ where
  σ: D · σ = C is_ground_subst σ
  by auto

from c have no_taut: ¬ (∃ A. Pos A ∈# C ∧ Neg A ∈# C)
  using sr.tautology_redundant by auto

from deriv have four_ten: chain sr_ext.derive Ns
  using resolution_prover_ground_derivation ns by auto

have ∃ l. D ∈ N_of_state (lnth Sts l) ∧ D ∉ N_of_state (lnth Sts (Suc l)) ∧ i ≤ l ∧ enat (Suc l) < llength Sts
  using fair using eventually_removed_from_N d unfolding ns by auto
then obtain l where
  l_p: D ∈ N_of_state (lnth Sts l) ∧ D ∉ N_of_state (lnth Sts (Suc l)) ∧ i ≤ l ∧ enat (Suc l) < llength Sts
  by auto
then have l_Ns: enat (Suc l) < llength Ns
  using ns by auto
from l_p have lnth Sts l ⇝ lnth Sts (Suc l)
  using deriv using chain_lnth_rel by auto
then show ?thesis
proof (cases rule: RP.cases)
  case (tautology_deletion A D_twin N P Q)
  then have D_twin = D
    using l_p by auto
  then have Pos (A ·a σ) ∈# C ∧ Neg (A ·a σ) ∈# C
    using tautology_deletion(3,4) σ
    by (metis Melem_subst_cls eql_neg_lit_eql_atm eql_pos_lit_eql_atm)
  then have False
    using no_taut by metis
  then show ?thesis
    by blast
next
  case (forward_subsumption D' P Q D_twin N)
  note lrhs = this(1,2) and D'_p = this(3,4)
  then have twins: D_twin = D ?Ns (Suc l) = N ?Ns l = N ∪ {D_twin}  ?Ps (Suc l) = P
    ?Ps l = P ?Qs (Suc l) = Q ?Qs l = Q
    using l_p by auto
  note D'_p = D'_p[unfolded twins(1)]
  from D'_p(2) have subs: subsumes D' C
    using d(3) by (blast intro: subsumes_trans)
  moreover have D' ∈ clss_of_state (Sup_state Sts)
    using twins D'_p l_p unfolding clss_of_state_def Sup_state_def
    by simp (metis (no_types) contra_subsetD llength_lmap lnth_lmap lnth_subset_Sup_llist)
  ultimately have ¬ strictly_subsumes D' D
    using d_least by auto
  then have subsumes D D'
    unfolding strictly_subsumes_def using D'_p by auto
  then have v: variants D D'
    using D'_p unfolding variants_iff_subsumes by auto
  then have mini: ∀ E ∈ {E ∈ clss_of_state (Sup_state Sts). subsumes E C}. ¬ strictly_subsumes E D'
    using d_least D'_p neg_properly_subsume_variants[of _ D D'] by auto

  from v have ∃ σ'. D' · σ' = C
    using σ variants_imp_exists_subtitution variants_sym by (metis subst_cls_comp_subst)
  then have ∃ σ'. D' · σ' = C ∧ is_ground_subst σ'
    using ground_C by (meson make_ground_subst refl)
  then obtain σ' where
    σ'_p: D' · σ' = C ∧ is_ground_subst σ'
    by metis

  show ?thesis
    using D'_p twins l_p subs mini σ'_p by auto
next
```

case $(\mathit{forward\_reduction}\ E\ L'\ P\ Q\ L\ \sigma\ D'\ N)$
then have *twins*: $D' + \{\#L\#\} = D$ $\mathit{?Ns}\ (\mathit{Suc}\ l) = N \cup \{D'\}$ $\mathit{?Ns}\ l = N \cup \{D' + \{\#L\#\}\}$
$\mathit{?Ps}\ (\mathit{Suc}\ l) = P$ $\mathit{?Ps}\ l = P$ $\mathit{?Qs}\ (\mathit{Suc}\ l) = Q$ $\mathit{?Qs}\ l = Q$
using *l_p* **by** *auto*
then have *D′_p*: $\mathit{strictly\_subsumes}\ D'\ D \wedge D' \in \mathit{?Ns}\ (\mathit{Suc}\ l)$
using *subset_strictly_subsumes*[*of D′ D*] **by** *auto*
then have *subc*: $\mathit{subsumes}\ D'\ C$
using $d(3)$ *subsumes_trans* **unfolding** *strictly_subsumes_def* **by** *blast*
from *D′_p* **have** $D' \in \mathit{clss\_of\_state}\ (\mathit{Sup\_state}\ \mathit{Sts})$
using *l_p* **by** (*metis* (*no_types*) *UnI1 clss_of_state_def N_of_state.simps llength_lmap lnth_lmap lnth_subset_Sup_llist subsetCE Sup_state_def*)
then have *False*
using *d_least D′_p subc* **by** *auto*
then show *?thesis*
by *auto*
next
case $(\mathit{clause\_processing}\ N\ \mathit{D\_twin}\ P\ Q)$
then have *twins*: $\mathit{D\_twin} = D$ $\mathit{?Ns}\ (\mathit{Suc}\ l) = N$ $\mathit{?Ns}\ l = N \cup \{D\}$ $\mathit{?Ps}\ (\mathit{Suc}\ l) = P \cup \{D\}$
$\mathit{?Ps}\ l = P$ $\mathit{?Qs}\ (\mathit{Suc}\ l) = Q$ $\mathit{?Qs}\ l = Q$
using *l_p* **by** *auto*
then show *?thesis*
using *d σ l_p d_least* **by** *blast*
qed (*use l_p* **in** *auto*)
qed

lemma *eventually_in_Qinf*:
assumes
D_p: $D \in \mathit{clss\_of\_state}\ (\mathit{Sup\_state}\ \mathit{Sts})$
$\mathit{subsumes}\ D\ C$ $\forall E \in \{E.\ E \in (\mathit{clss\_of\_state}\ (\mathit{Sup\_state}\ \mathit{Sts})) \wedge \mathit{subsumes}\ E\ C\}.\ \neg\ \mathit{strictly\_subsumes}\ E\ D$ **and**
fair: *fair_state_seq Sts* **and**

ns: $\mathit{Ns} = \mathit{lmap}\ \mathit{grounding\_of\_state}\ \mathit{Sts}$ **and**
c: $C \in \mathit{Liminf\_llist}\ \mathit{Ns} - \mathit{sr.Rf}\ (\mathit{Liminf\_llist}\ \mathit{Ns})$ **and**
ground_C: *is_ground_cls C*
shows $\exists D'\ \sigma'.\ D' \in \mathit{Q\_of\_state}\ (\mathit{Liminf\_state}\ \mathit{Sts}) \wedge D' \cdot \sigma' = C \wedge \mathit{is\_ground\_subst}\ \sigma'$
proof −
let $\mathit{?Ns} = \lambda i.\ \mathit{N\_of\_state}\ (\mathit{lnth}\ \mathit{Sts}\ i)$
let $\mathit{?Ps} = \lambda i.\ \mathit{P\_of\_state}\ (\mathit{lnth}\ \mathit{Sts}\ i)$
let $\mathit{?Qs} = \lambda i.\ \mathit{Q\_of\_state}\ (\mathit{lnth}\ \mathit{Sts}\ i)$

from *D_p* **obtain** *i* **where**
i_p: $i < \mathit{llength}\ \mathit{Sts}$ $D \in \mathit{?Ns}\ i \vee D \in \mathit{?Ps}\ i \vee D \in \mathit{?Qs}\ i$
unfolding *clss_of_state_def Sup_state_def*
by *simp_all* (*metis* (*no_types*) *in_Sup_llist_in_nth llength_lmap lnth_lmap*)

have *derivns*: *chain sr_ext.derive Ns* **using** *resolution_prover_ground_derivation deriv ns* **by** *auto*

have $\exists \sigma.\ D \cdot \sigma = C \wedge \mathit{is\_ground\_subst}\ \sigma$
using *instance_if_subsumed_and_in_limit*[*OF ns c*] *D_p i_p* **by** *blast*
then obtain σ **where**
σ: $D \cdot \sigma = C$ $\mathit{is\_ground\_subst}\ \sigma$
by *blast*

{
assume *a*: $D \in \mathit{?Ns}\ i$
then obtain $D'\ \sigma'\ l$ **where** *D′_p*:
$D' \in \mathit{?Ps}\ l \cup \mathit{?Qs}\ l$
$D' \cdot \sigma' = C$
$\mathit{enat}\ l < \mathit{llength}\ \mathit{Sts}$
$\mathit{is\_ground\_subst}\ \sigma'$
$\forall E \in \{E.\ E \in (\mathit{clss\_of\_state}\ (\mathit{Sup\_state}\ \mathit{Sts})) \wedge \mathit{subsumes}\ E\ C\}.\ \neg\ \mathit{strictly\_subsumes}\ E\ D'$
$\mathit{subsumes}\ D'\ C$
using *from_N_to_P_or_Q deriv fair ns c i_p*(*1*) *D_p*(*2*) *D_p*(*3*) **by** *blast*

```
    then obtain l' where
      l'_p: D' ∈ ?Qs l' l' < llength Sts
      using from_P_to_Q[OF fair ns c _ D'_p(3) D'_p(6) D'_p(5)] by blast
    then have D' ∈ Q_of_state (Liminf_state Sts)
      using from_Q_to_Q_inf[OF fair ns c _ l'_p(2)] D'_p by auto
    then have ?thesis
      using D'_p by auto
  }
  moreover
  {
    assume a: D ∈ ?Ps i
    then obtain l' where
      l'_p: D ∈ ?Qs l' l' < llength Sts
      using from_P_to_Q[OF fair ns c a i_p(1) D_p(2) D_p(3)] by auto
    then have D ∈ Q_of_state (Liminf_state Sts)
      using from_Q_to_Q_inf[OF fair ns c l'_p(1) l'_p(2)] D_p(3) σ(1) σ(2) D_p(2) by auto
    then have ?thesis
      using D_p σ by auto
  }
  moreover
  {
    assume a: D ∈ ?Qs i
    then have D ∈ Q_of_state (Liminf_state Sts)
      using from_Q_to_Q_inf[OF fair ns c a i_p(1)] σ D_p(2,3) by auto
    then have ?thesis
      using D_p σ by auto
  }
  ultimately show ?thesis
    using i_p by auto
qed
```

The following corresponds to Lemma 4.11:

```
lemma fair_imp_Liminf_minus_Rf_subset_ground_Liminf_state:
  assumes
    deriv: chain (op ⇝) Sts and
    fair: fair_state_seq Sts and
    ns: Ns = lmap grounding_of_state Sts
  shows Liminf_llist Ns − sr.Rf (Liminf_llist Ns) ⊆ grounding_of_clss (Q_of_state (Liminf_state Sts))
proof
  let ?Ns = λi. N_of_state (lnth Sts i)
  let ?Ps = λi. P_of_state (lnth Sts i)
  let ?Qs = λi. Q_of_state (lnth Sts i)

  have SQinf: clss_of_state (Liminf_state Sts) = Liminf_llist (lmap Q_of_state Sts)
    using fair unfolding fair_state_seq_def Liminf_state_def clss_of_state_def by auto

  fix C
  assume C_p: C ∈ Liminf_llist Ns − sr.Rf (Liminf_llist Ns)
  then have C ∈ Sup_llist Ns
    using Liminf_llist_subset_Sup_llist[of Ns] by blast
  then obtain D_proto where
    D_proto ∈ clss_of_state (Sup_state Sts) ∧ subsumes D_proto C
    using in_Sup_llist_in_Sup_state unfolding ns subsumes_def by blast
  then obtain D where
    D_p: D ∈ clss_of_state (Sup_state Sts)
    subsumes D C
    ∀ E ∈ {E. E ∈ clss_of_state (Sup_state Sts) ∧ subsumes E C}. ¬ strictly_subsumes E D
    using strictly_subsumes_has_minimum[of {E. E ∈ clss_of_state (Sup_state Sts) ∧ subsumes E C}]
    by auto

  have ground_C: is_ground_cls C
    using C_p using Liminf_grounding_of_state_ground ns by auto
```

have $\exists D' \sigma'$. $D' \in$ *Q_of_state* (*Liminf_state Sts*) $\wedge$ $D' \cdot \sigma' = C$ $\wedge$ *is_ground_subst* σ'
  using *eventually_in_Qinf*[*of D C Ns*] **using** *D_p*(*1*) *D_p*(*2*) *D_p*(*3*) *fair ns C_p ground_C* **by** *auto*
then obtain D' σ' **where**
  D'_p: $D' \in$ *Q_of_state* (*Liminf_state Sts*) $\wedge$ $D' \cdot \sigma' = C$ $\wedge$ *is_ground_subst* σ'
  by *blast*
then have $D' \in$ *clss_of_state* (*Liminf_state Sts*)
  by (*simp add*: *clss_of_state_def*)
then have $C \in$ *grounding_of_state* (*Liminf_state Sts*)
  unfolding *grounding_of_clss_def grounding_of_cls_def* **using** *D'_p* **by** *auto*
then show $C \in$ *grounding_of_clss* (*Q_of_state* (*Liminf_state Sts*))
  using *SQinf clss_of_state_def fair fair_state_seq_def* **by** *auto*
qed

The following corresponds to (one direction of) Theorem 4.13:

lemma *ground_subclauses*:
  assumes
    $\forall i <$ *length CAs*. *CAs* ! i = *Cs* ! i + *poss* (*AAs* ! i) **and**
    length Cs = *length CAs* **and**
    is_ground_cls_list CAs
  shows *is_ground_cls_list Cs*
  unfolding *is_ground_cls_list_def*
  by (*metis assms in_set_conv_nth is_ground_cls_list_def is_ground_cls_union*)

lemma *subseteq_Liminf_state_eventually_always*:
  fixes *CC*
  assumes
    finite CC **and**
    $CC \neq \{\}$ **and**
    $CC \subseteq$ *Q_of_state* (*Liminf_state Sts*)
  shows $\exists j$. *enat* $j <$ *llength Sts* $\wedge$ ($\forall j' \geq$ *enat* j. $j' <$ *llength Sts* $\longrightarrow$ $CC \subseteq$ *Q_of_state* (*lnth Sts* j'))
proof −
  from *assms*(*3*) **have** $\forall C \in CC$. $\exists j$. *enat* $j <$ *llength Sts* $\wedge$
    ($\forall j' \geq$ *enat* j. $j' <$ *llength Sts* $\longrightarrow$ $C \in$ *Q_of_state* (*lnth Sts* j'))
    unfolding *Liminf_state_def Liminf_llist_def* **by** *force*
  then obtain f **where**
    f_p: $\forall C \in CC$. $f\ C <$ *llength Sts* $\wedge$ ($\forall j' \geq$ *enat* ($f\ C$). $j' <$ *llength Sts* $\longrightarrow$ $C \in$ *Q_of_state* (*lnth Sts* j'))
    by *moura*

  define j :: *nat* **where**
    j = *Max* (f ‘ *CC*)

  have *enat* $j <$ *llength Sts*
    unfolding *j_def* **using** *f_p assms*(*1*)
    by (*metis* (*mono_tags*) *Max_in assms*(*2*) *finite_imageI imageE image_is_empty*)
  moreover have $\forall C\ j'$. $C \in CC$ $\longrightarrow$ *enat* $j \leq j'$ $\longrightarrow$ $j' <$ *llength Sts* $\longrightarrow$ $C \in$ *Q_of_state* (*lnth Sts* j')
  proof (*intro allI impI*)
    fix C :: *'a clause* **and** j' :: *nat*
    assume *a*: $C \in CC$ *enat* $j \leq$ *enat* j' *enat* $j' <$ *llength Sts*
    then have $f\ C \leq j'$
      unfolding *j_def* **using** *assms*(*1*) *Max.bounded_iff* **by** *auto*
    then show $C \in$ *Q_of_state* (*lnth Sts* j')
      using *f_p a* **by** *auto*
  qed
  ultimately show *?thesis*
    by *auto*
qed

lemma *empty_clause_in_Q_of_Liminf_state*:
  assumes
    empty_in: {#} $\in$ *Liminf_llist* (*lmap grounding_of_state Sts*) **and**
    fair: *fair_state_seq Sts*
  shows {#} $\in$ *Q_of_state* (*Liminf_state Sts*)
proof −

```
  define Ns :: 'a clause set llist where
    ns: Ns = lmap grounding_of_state Sts

  from empty_in have in_Liminf_not_Rf: {#} ∈ Liminf_llist Ns − sr.Rf (Liminf_llist Ns)
    unfolding ns sr.Rf_def by auto

  from assms obtain i where
    i_p: enat i < llength (lmap grounding_of_state Sts)
    {#} ∈ lnth (lmap grounding_of_state Sts) i
    unfolding Liminf_llist_def by force
  then have {#} ∈ grounding_of_state (lnth Sts i)
    by auto
  then have {#} ∈ clss_of_state (lnth Sts i)
    unfolding grounding_of_clss_def grounding_of_cls_def by auto
  then have in_Sup_state: {#} ∈ clss_of_state (Sup_state Sts)
    using i_p(1) unfolding Sup_state_def clss_of_state_def
    by simp (metis llength_lmap lnth_lmap lnth_subset_Sup_llist set_mp)
  then have ∃ D' σ'. D' ∈ Q_of_state (Liminf_state Sts) ∧ D' · σ' = {#} ∧ is_ground_subst σ'
    using eventually_in_Qinf[of {#} {#} Ns, OF in_Sup_state _ _ fair ns in_Liminf_not_Rf]
    unfolding is_ground_cls_def strictly_subsumes_def subsumes_def by simp
  then show ?thesis
    by simp
qed

lemma grounding_of_state_Liminf_state_subseteq:
  grounding_of_state (Liminf_state Sts) ⊆ Liminf_llist (lmap grounding_of_state Sts)
proof
  fix C :: 'a clause
  assume C ∈ grounding_of_state (Liminf_state Sts)
  then obtain D σ where
    D_σ_p: D ∈ clss_of_state (Liminf_state Sts) D · σ = C is_ground_subst σ
    unfolding clss_of_state_def grounding_of_clss_def grounding_of_cls_def by auto
  then have ii: D ∈ Liminf_llist (lmap N_of_state Sts) ∨
    D ∈ Liminf_llist (lmap P_of_state Sts) ∨
    D ∈ Liminf_llist (lmap Q_of_state Sts)
    unfolding clss_of_state_def  Liminf_state_def by simp
  then have C ∈ Liminf_llist (lmap grounding_of_clss (lmap N_of_state Sts)) ∨
    C ∈ Liminf_llist (lmap grounding_of_clss (lmap P_of_state Sts)) ∨
    C ∈ Liminf_llist (lmap grounding_of_clss (lmap Q_of_state Sts))
    unfolding Liminf_llist_def grounding_of_clss_def grounding_of_cls_def
    apply −
    apply (erule disjE)
    subgoal
      apply (rule disjI1)
      using D_σ_p by auto
    subgoal
      apply (erule HOL.disjE)
      subgoal
        apply (rule disjI2)
        apply (rule disjI1)
        using D_σ_p by auto
      subgoal
        apply (rule disjI2)
        apply (rule disjI2)
        using D_σ_p by auto
      done
    done
  then show C ∈ Liminf_llist (lmap grounding_of_state Sts)
    unfolding Liminf_llist_def clss_of_state_def grounding_of_clss_def by auto
qed

theorem RP_sound:
  assumes {#} ∈ clss_of_state (Liminf_state Sts)
```

shows ¬ *satisfiable* (*grounding_of_state* (*lhd Sts*))
proof −
from *assms* **have** {#} ∈ *grounding_of_state* (*Liminf_state Sts*)
unfolding *grounding_of_clss_def* **by** (*force intro*: *ex_ground_subst*)
then have ¬ *satisfiable* (*grounding_of_state* (*Liminf_state Sts*))
unfolding *true_clss_def* **by** *auto*
then have ¬ *satisfiable* (*Liminf_llist* (*lmap grounding_of_state Sts*))
using *grounding_of_state_Liminf_state_subseteq true_clss_mono* **by** *blast*
then have ¬ *satisfiable* (*lhd* (*lmap grounding_of_state Sts*))
using *sr_ext.sat_deriv_Liminf_iff* [*of lmap grounding_of_state Sts*]
by (*metis deriv resolution_prover_ground_derivation*)
then show *?thesis*
unfolding *lhd_lmap_Sts* **.**
qed

theorem *RP_saturated_if_fair*:
assumes *fair*: *fair_state_seq Sts*
shows *sr.saturated_upto* (*Liminf_llist* (*lmap grounding_of_state Sts*))
proof −
define *Ns* :: *'a clause set llist* **where**
ns: *Ns* = *lmap grounding_of_state Sts*

let *?N* = *λi*. *grounding_of_state* (*lnth Sts i*)

let *?Ns* = *λi*. *N_of_state* (*lnth Sts i*)
let *?Ps* = *λi*. *P_of_state* (*lnth Sts i*)
let *?Qs* = *λi*. *Q_of_state* (*lnth Sts i*)

have *ground_ns_in_ground_limit_st*:
Liminf_llist Ns − *sr.Rf* (*Liminf_llist Ns*) ⊆ *grounding_of_clss* (*Q_of_state* (*Liminf_state Sts*))
using *fair deriv fair_imp_Liminf_minus_Rf_subset_ground_Liminf_state ns* **by** *blast*

have *derivns*: *chain sr_ext.derive Ns*
using *resolution_prover_ground_derivation deriv ns* **by** *auto*

{
fix *γ* :: *'a inference*
assume *γ_p*: *γ* ∈ *gr.ord_Γ*
let *?CC* = *side_prems_of γ*
let *?DA* = *main_prem_of γ*
let *?E* = *concl_of γ*
assume *a*: *set_mset ?CC* ∪ {*?DA*}
⊆ *Liminf_llist* (*lmap grounding_of_state Sts*) − *sr.Rf* (*Liminf_llist* (*lmap grounding_of_state Sts*))

have *ground_ground_Liminf*: *is_ground_clss* (*Liminf_llist* (*lmap grounding_of_state Sts*))
using *Liminf_grounding_of_state_ground* **unfolding** *is_ground_clss_def* **by** *auto*

have *ground_cc*: *is_ground_clss* (*set_mset ?CC*)
using *a ground_ground_Liminf is_ground_clss_def* **by** *auto*

have *ground_da*: *is_ground_cls ?DA*
using *a grounding_ground singletonI ground_ground_Liminf*
by (*simp add*: *Liminf_grounding_of_state_ground*)

from *γ_p* **obtain** *CAs AAs As* **where**
CAs_p: *gr.ord_resolve CAs ?DA AAs As ?E* ∧ *mset CAs* = *?CC*
unfolding *gr.ord_Γ_def* **by** *auto*

have *DA_CAs_in_ground_Liminf*:
{*?DA*} ∪ *set CAs* ⊆ *grounding_of_clss* (*Q_of_state* (*Liminf_state Sts*))
using *a CAs_p* **unfolding** *clss_of_state_def* **using** *fair* **unfolding** *fair_state_seq_def*
by (*metis* (*no_types*, *lifting*) *Un_empty_left ground_ns_in_ground_limit_st a clss_of_state_def*
ns set_mset_mset subset_trans sup_commute)

```
then have ground_cas: is_ground_cls_list CAs
  using CAs_p unfolding is_ground_cls_list_def by auto

have ground_e: is_ground_cls ?E
proof -
  have a1: atms_of ?E ⊆ (⋃ CA ∈ set CAs. atms_of CA) ∪ atms_of ?DA
    using γ_p ground_cc ground_da gr.ord_resolve_atms_of_concl_subset[of CAs ?DA _ _ ?E] CAs_p
    by auto
  {
    fix L :: 'a literal
    assume L ∈# concl_of γ
    then have atm_of L ∈ atms_of (concl_of γ)
      by (meson atm_of_lit_in_atms_of)
    then have is_ground_atm (atm_of L)
      using a1 ground_cas ground_da is_ground_cls_imp_is_ground_atm is_ground_cls_list_def
      by auto
  }
  then show ?thesis
    unfolding is_ground_cls_def is_ground_lit_def by simp
qed

have ∃ AAs As σ. ord_resolve (S_M S (Q_of_state (Liminf_state Sts))) CAs ?DA AAs As σ ?E
  using CAs_p[THEN conjunct1]
proof (cases rule: gr.ord_resolve.cases)
  case (ord_resolve n Cs D)
  note DA = this(1) and e = this(2) and cas_len = this(3) and cs_len = this(4) and
    aas_len = this(5) and as_len = this(6) and nz = this(7) and cas = this(8) and
    aas_not_empt = this(9) and as_aas = this(10) and eligibility = this(11) and
    str_max = this(12) and sel_empt = this(13)

  have len_aas_len_as: length AAs = length As
    using aas_len as_len by auto

  from as_aas have ∀ i<n. ∀ A ∈# add_mset (As ! i) (AAs ! i). A = As ! i
    using ord_resolve by simp
  then have ∀ i < n. card (set_mset (add_mset (As ! i) (AAs ! i))) ≤ Suc 0
    using all_the_same by metis
  then have ∀ i < length AAs. card (set_mset (add_mset (As ! i) (AAs ! i))) ≤ Suc 0
    using aas_len by auto
  then have ∀ AA ∈ set (map2 add_mset As AAs). card (set_mset AA) ≤ Suc 0
    using set_map2_ex[of AAs As add_mset, OF len_aas_len_as] by auto
  then have is_unifiers id_subst (set_mset ` set (map2 add_mset As AAs))
    unfolding is_unifiers_def is_unifier_def by auto
  moreover have finite (set_mset ` set (map2 add_mset As AAs))
    by auto
  moreover have ∀ AA ∈ set_mset ` set (map2 add_mset As AAs). finite AA
    by auto
  ultimately obtain σ where
    σ_p: Some σ = mgu (set_mset ` set (map2 add_mset As AAs))
    using mgu_complete by metis

  have ground_elig: gr.eligible As (D + negs (mset As))
    using ord_resolve by simp
  have ground_cs: ∀ i < n. is_ground_cls (Cs ! i)
    using ord_resolve(8) ord_resolve(3,4) ground_cas
    using ground_subclauses[of CAs Cs AAs] unfolding is_ground_cls_list_def by auto
  have ground_set_as: is_ground_atms (set As)
    using ord_resolve(1) ground_da
    by (metis atms_of_negs is_ground_cls_union set_mset_mset is_ground_cls_is_ground_atms_atms_of)
  then have ground_mset_as: is_ground_atm_mset (mset As)
    unfolding is_ground_atm_mset_def is_ground_atms_def by auto
  have ground_as: is_ground_atm_list As
```

```
      using ground_set_as is_ground_atm_list_def is_ground_atms_def by auto
    have ground_d: is_ground_cls D
      using ground_da ord_resolve by simp

    from as_len nz have atms_of D ∪ set As ≠ {} finite (atms_of D ∪ set As)
      by auto
    then have Max (atms_of D ∪ set As) ∈ atms_of D ∪ set As
      using Max_in by metis
    then have is_ground_Max: is_ground_atm (Max (atms_of D ∪ set As))
      using ground_d ground_mset_as is_ground_cls_imp_is_ground_atm
      unfolding is_ground_atm_mset_def by auto
    then have Maxσ_is_Max: ∀σ. Max (atms_of D ∪ set As) ·a σ = Max (atms_of D ∪ set As)
      by auto

    have ann1: maximal_wrt (Max (atms_of D ∪ set As)) (D + negs (mset As))
      unfolding maximal_wrt_def
      by clarsimp (metis Max_less_iff UnCI ‹atms_of D ∪ set As ≠ {}›
          ‹finite (atms_of D ∪ set As)› ground_d ground_set_as infinite_growing is_ground_Max
          is_ground_atms_def is_ground_cls_imp_is_ground_atm less_atm_ground)

    from ground_elig have ann2:
      Max (atms_of D ∪ set As) ·a σ = Max (atms_of D ∪ set As)
      D · σ + negs (mset As ·am σ) = D + negs (mset As)
      using is_ground_Max ground_mset_as ground_d by auto

    from ground_elig have fo_elig:
      eligible (S_M S (Q_of_state (Liminf_state Sts))) σ As (D + negs (mset As))
      unfolding gr.eligible.simps eligible.simps gr.maximal_wrt_def using ann1 ann2
      by (auto simp: S_Q_def)

    have l: ∀i < n. gr.strictly_maximal_wrt (As ! i) (Cs ! i)
      using ord_resolve by simp
    then have ∀i < n. strictly_maximal_wrt (As ! i) (Cs ! i)
      unfolding gr.strictly_maximal_wrt_def strictly_maximal_wrt_def
      using ground_as[unfolded is_ground_atm_list_def] ground_cs as_len less_atm_ground
      by clarsimp (fastforce simp: is_ground_cls_as_atms)+

    then have ll: ∀i < n. strictly_maximal_wrt (As ! i ·a σ) (Cs ! i · σ)
      by (simp add: ground_as ground_cs as_len)

    have m: ∀i < n. S_Q (CAs ! i) = {#}
      using ord_resolve by simp

    have ground_e: is_ground_cls (⋃#mset Cs + D)
      using ground_d ground_cs ground_e e by simp
    show ?thesis
      using ord_resolve.intros[OF cas_len cs_len aas_len as_len nz cas aas_not_empt σ_p fo_elig ll] m DA e ground_e
      unfolding S_Q_def by auto
  qed
  then obtain AAs As σ where
    σ_p: ord_resolve (S_M S (Q_of_state (Liminf_state Sts))) CAs ?DA AAs As σ ?E
    by auto
  then obtain ηs' η' η2' CAs' DA' AAs' As' τ' E' where s_p:
    is_ground_subst η'
    is_ground_subst_list ηs'
    is_ground_subst η2'
    ord_resolve_rename S CAs' DA' AAs' As' τ' E'
    CAs' ·cl ηs' = CAs
    DA' · η' = ?DA
    E' · η2' = ?E
    {DA'} ∪ set CAs' ⊆ Q_of_state (Liminf_state Sts)
    using ord_resolve_rename_lifting[OF sel_stable, of Q_of_state (Liminf_state Sts) CAs ?DA]
      σ_p selection_axioms DA_CAs_in_ground_Liminf by metis
```

```
    from this(8) have ∃j. enat j < llength Sts ∧ (set CAs' ∪ {DA'} ⊆ ?Qs j)
      unfolding Liminf_llist_def
      using subseteq_Liminf_state_eventually_always[of {DA'} ∪ set CAs'] by auto
    then obtain j where
      j_p: is_least (λj. enat j < llength Sts ∧ set CAs' ∪ {DA'} ⊆ ?Qs j) j
      using least_exists[of λj. enat j < llength Sts ∧ set CAs' ∪ {DA'} ⊆ ?Qs j] by force
    then have j_p': enat j < llength Sts set CAs' ∪ {DA'} ⊆ ?Qs j
      unfolding is_least_def by auto
    then have jn0: j ≠ 0
      using empty_Q0 by (metis bot_eq_sup_iff gr_implies_not_zero insert_not_empty llength_lnull
          lnth_0_conv_lhd sup.orderE)
    then have j_adds_CAs': ¬ set CAs' ∪ {DA'} ⊆ ?Qs (j − 1) set CAs' ∪ {DA'} ⊆ ?Qs j
      using j_p unfolding is_least_def
       apply (metis (no_types) One_nat_def Suc_diff_Suc Suc_ile_eq diff_diff_cancel diff_zero
          less_imp_le less_one neq0_conv zero_less_diff)
      using j_p'(2) by blast
    have lnth Sts (j − 1) ⇝ lnth Sts j
      using j_p'(1) jn0  deriv chain_lnth_rel[of _ _ j − 1] by force
    then obtain C' where C'_p:
      ?Ns (j − 1) = {}
      ?Ps (j − 1) = ?Ps j ∪ {C'}
      ?Qs j = ?Qs (j − 1) ∪ {C'}
      ?Ns j = concls_of (ord_FO_resolution.inferences_between (?Qs (j − 1)) C')
      C' ∈ set CAs' ∪ {DA'}
      C' ∉ ?Qs (j − 1)
      using j_adds_CAs' by (induction rule: RP.cases) auto

    then have ihih: set CAs' ∪ {DA'} − {C'} ⊆ ?Qs (j − 1)
      using j_adds_CAs' by auto
    have E' ∈ ?Ns j
    proof −
      have E' ∈ concls_of (ord_FO_resolution.inferences_between (Q_of_state (lnth Sts (j − 1))) C')
        unfolding infer_from_def ord_FO_Γ_def unfolding inference_system.inferences_between_def
        apply (rule_tac x = Infer (mset CAs') DA' E' in image_eqI)
        subgoal by auto
        subgoal
          using s_p(4)
          unfolding infer_from_def
          apply (rule ord_resolve_rename.cases)
          using s_p(4)
          using C'_p(3) C'_p(5) j_p'(2) apply force
          done
        done
      then show ?thesis
        using C'_p(4) by auto
    qed
    then have E' ∈ clss_of_state (lnth Sts j)
      using j_p' unfolding clss_of_state_def by auto
    then have ?E ∈ grounding_of_state (lnth Sts j)
      using s_p(7) s_p(3) unfolding grounding_of_clss_def grounding_of_cls_def by force
    then have γ ∈ sr.Ri (grounding_of_state (lnth Sts j))
      using sr.Ri_effective γ_p by auto
    then have γ ∈ sr_ext_Ri (?N j)
      unfolding sr_ext_Ri_def by auto
    then have γ ∈ sr_ext_Ri (Sup_llist (lmap grounding_of_state Sts))
      using j_p' contra_subsetD llength_lmap lnth_lmap lnth_subset_Sup_llist sr_ext.Ri_mono by metis
    then have γ ∈ sr_ext_Ri (Liminf_llist (lmap grounding_of_state Sts))
      using sr_ext.Ri_Sup_subset_Ri_Liminf[of Ns] derivns ns by blast
  }
  then have sr_ext.saturated_upto (Liminf_llist (lmap grounding_of_state Sts))
    unfolding sr_ext.saturated_upto_def sr_ext.inferences_from_def infer_from_def sr_ext_Ri_def
    by auto
  then show ?thesis
```

using *gd_ord_Γ_ngd_ord_Γ sr.redundancy_criterion_axioms*
redundancy_criterion_standard_extension_saturated_upto_iff [*of gr.ord_Γ*]
unfolding *sr_ext_Ri_def* **by** *auto*
qed

corollary *RP_complete_if_fair*:
assumes
fair: *fair_state_seq Sts* **and**
unsat: ¬ *satisfiable* (*grounding_of_state* (*lhd Sts*))
shows {#} ∈ *Q_of_state* (*Liminf_state Sts*)
proof −
have ¬ *satisfiable* (*Liminf_llist* (*lmap grounding_of_state Sts*))
unfolding *sr_ext.sat_deriv_Liminf_iff* [*OF resolution_prover_ground_derivation*[*OF deriv*]]
by (*rule unsat*[*folded lhd_lmap_Sts*[*of grounding_of_state*]])
moreover have *sr.saturated_upto* (*Liminf_llist* (*lmap grounding_of_state Sts*))
by (*rule RP_saturated_if_fair*[*OF fair*, *simplified*])
ultimately have {#} ∈ *Liminf_llist* (*lmap grounding_of_state Sts*)
using *sr.saturated_upto_complete_if* **by** *auto*
then show *?thesis*
using *empty_clause_in_Q_of_Liminf_state fair* **by** *auto*
qed

end

end

end