

Formalization of Bachmair and Ganzinger's Ordered Resolution Prover

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

March 19, 2025

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	1
2	Map Function on Two Parallel Lists	1
3	Supremum and Liminf of Lazy Lists	3
3.1	Library	3
3.2	Supremum	4
3.3	Supremum up-to	4
3.4	Liminf	5
3.5	Liminf up-to	8
4	Relational Chains over Lazy Lists	9
4.1	Chains	9
4.2	A Coinductive Puzzle	13
4.3	Full Chains	19
5	Clausal Logic	20
5.1	Literals	20
5.2	Clauses	23
6	Herbrand Interpretation	25
7	Abstract Substitutions	27
7.1	Library	27
7.2	Substitution Operators	28
7.3	Substitution Lemmas	30
7.3.1	Identity Substitution	30
7.3.2	Associativity of Composition	31
7.3.3	Compatibility of Substitution and Composition	31
7.3.4	"Commutativity" of Membership and Substitution	32
7.3.5	Signs and Substitutions	32
7.3.6	Substitution on Literal(s)	32
7.3.7	Substitution on Empty	33
7.3.8	Substitution on a Union	34
7.3.9	Substitution on a Singleton	34
7.3.10	Substitution on (#)	35
7.3.11	Substitution on tl	35

7.3.12	Substitution on (!)	35
7.3.13	Substitution on Various Other Functions	36
7.3.14	Renamings	36
7.3.15	Monotonicity	37
7.3.16	Size after Substitution	38
7.3.17	Variable Disjointness	38
7.3.18	Ground Expressions and Substitutions	38
7.3.19	Subsumption	42
7.3.20	Unifiers	42
7.3.21	Most General Unifier	42
7.3.22	Generalization and Subsumption	43
7.3.23	Generalization and Subsumption	45
7.4	Most General Unifiers	47
7.5	Idempotent Most General Unifiers	48
8	Refutational Inference Systems	48
8.1	Preliminaries	48
8.2	Refutational Completeness	50
8.3	Compactness	51
9	Candidate Models for Ground Resolution	52
10	Ground Unordered Resolution Calculus	58
10.1	Inference Rule	58
10.2	Inference System	60
11	Ground Ordered Resolution Calculus with Selection	60
11.1	Inference Rule	61
11.2	Inference System	67
12	Theorem Proving Processes	68
13	The Standard Redundancy Criterion	72
14	First-Order Ordered Resolution Calculus with Selection	77
14.1	Library	78
14.2	Calculus	79
14.3	Soundness	80
14.4	Other Basic Properties	82
14.5	Inference System	83
14.6	Lifting	85
15	An Ordered Resolution Prover for First-Order Clauses	99

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. Section 4.2 generalizes the refutational completeness argument and introduces the standard redundancy criterion, which can be used in conjunction with ordered resolution. Finally, Section 4.3 lifts the result to a first-order prover, specified as a calculus. Figure 1 shows the corresponding Isabelle theory structure.

We refer to the following publications for details:

Anders Schlichtkrull, Jasmin Christian Blanchette, Dmitriy Traytel, Uwe Waldmann:
 Formalizing Bachmair and Ganzinger's Ordered Resolution Prover.
 IJCAR 2018: 89-107
http://matryoshka.gforge.inria.fr/pubs/rp_paper.pdf

Anders Schlichtkrull, Jasmin Blanchette, Dmitriy Traytel, Uwe Waldmann:
 Formalizing Bachmair and Ganzinger's Ordered Resolution Prover.
 Journal of Automated Reasoning
http://matryoshka.gforge.inria.fr/pubs/rp_article.pdf

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

```
abbreviation map2 :: ('a ⇒ 'b ⇒ 'c) ⇒ 'a list ⇒ 'b list ⇒ 'c list where
  map2 f xs ys ≡ map (case_prod f) (zip xs ys)
```

```
lemma map2_empty_iff[simp]: map2 f xs ys = [] ⟷ xs = [] ∨ 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 ⟹ g ` set (map2 f t s) = set (map2 (λa b. g (f a b)) t s)
  by auto
```

```
lemma map2_tl: length t = length s ⟹ 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 map2_tl tl_Nil zip_Cons_Cons)
```

```
lemma map_zip_assoc:
  map f (zip (zip xs ys) zs) = map (λ(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. ∃ i < length t. x = f (s ! i) (t ! i)}
proof (rule; rule)
  fix x
  assume x ∈ set (map2 f s t)
  then obtain i where i_p: i < length (map2 f s t) ∧ 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 ∈ {x. ∃ i < length t. x = f (s ! i) (t ! i)}
    using assms by auto
next
```

```
fix x
assume x ∈ {x. ∃ i < length t. x = f (s ! i) (t ! i)}
then obtain i where i_p: i < length t ∧ 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 ∈ set (map2 f s t)
  by (metis in_set_conv_nth)
qed
```

```
end
```

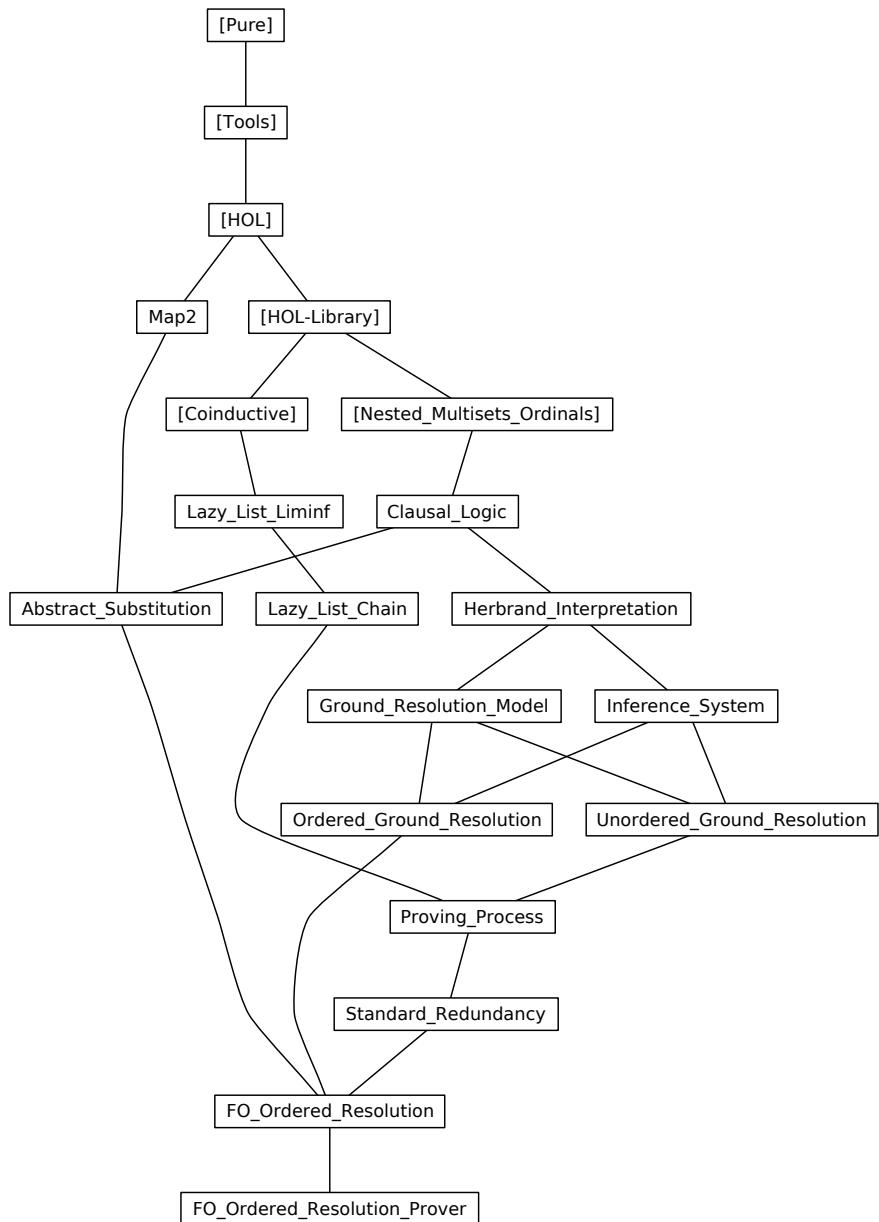


Figure 1: Theory dependency graph

3 Supremum and 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.

3.1 Library

```
lemma less_llength_ltake:  $i < \text{llength} (\text{ltake } k \text{ } Xs) \longleftrightarrow i < k \wedge i < \text{llength } Xs$ 
  by simp
```

3.2 Supremum

```
definition Sup_llist :: 'a set llist ⇒ 'a set where
  Sup_llist Xs = ( $\bigcup_{i \in \{i. \text{enat } i < \text{llength } Xs\}} \text{lnth } Xs \ i$ )
```

```
lemma lnth_subset_Sup_llist:  $\text{enat } i < \text{llength } Xs \implies \text{lnth } Xs \ i \subseteq \text{Sup\_llist } Xs$ 
  unfolding Sup_llist_def by auto
```

```
lemma Sup_llist_imp_exists_index:  $x \in \text{Sup\_llist } Xs \implies \exists i. \text{enat } i < \text{llength } Xs \wedge x \in \text{lnth } Xs \ i$ 
  unfolding Sup_llist_def by auto
```

```
lemma exists_index_imp_Sup_llist:  $\text{enat } i < \text{llength } Xs \implies x \in \text{lnth } Xs \ i \implies x \in \text{Sup\_llist } Xs$ 
  unfolding Sup_llist_def by auto
```

```
lemma Sup_llist_LNil[simp]:  $\text{Sup\_llist } LNil = \{\}$ 
  unfolding Sup_llist_def by auto
```

```
lemma Sup_llist_LCons[simp]:  $\text{Sup\_llist } (\text{LCons } X \text{ } Xs) = X \cup \text{Sup\_llist } Xs$ 
  unfolding Sup_llist_def
proof (intro subset_antisym subsetI)
  fix x
  assume  $x \in (\bigcup_{i \in \{i. \text{enat } i < \text{llength } (\text{LCons } X \text{ } Xs)\}} \text{lnth } (\text{LCons } X \text{ } Xs) \ i)$ 
  then obtain i where len:  $\text{enat } i < \text{llength } (\text{LCons } X \text{ } Xs)$  and nth:  $x \in \text{lnth } (\text{LCons } X \text{ } Xs) \ i$ 
    by blast
  from nth have  $x \in X \vee i > 0 \wedge x \in \text{lnth } Xs \ (i - 1)$ 
    by (metis lnth_LCons' neq0_conv)
  then have  $x \in X \vee (\exists i. \text{enat } i < \text{llength } Xs \wedge x \in \text{lnth } Xs \ i)$ 
    by (metis len Suc_pred' eSuc_enat illess_Suc_eq less_irrefl lnth_LCons not_less_order_trans)
  then show  $x \in X \cup (\bigcup_{i \in \{i. \text{enat } i < \text{llength } Xs\}} \text{lnth } Xs \ i)$ 
    by blast
qed ((auto)[], metis i0_lb lnth_0 zero_enat_def, metis Suc ile_eq lnth_Suc_LCons)
```

```
lemma lhd_subset_Sup_llist:  $\neg \text{lnull } Xs \implies \text{lhd } Xs \subseteq \text{Sup\_llist } Xs$ 
  by (cases Xs) simp_all
```

3.3 Supremum up-to

```
definition Sup_upto_llist :: 'a set llist ⇒ enat ⇒ 'a set where
  Sup_upto_llist Xs j = ( $\bigcup_{i \in \{i. \text{enat } i < \text{llength } Xs \wedge \text{enat } i \leq j\}} \text{lnth } Xs \ i$ )
```

```
lemma Sup_upto_llist_eq_Sup_llist_ltake:  $\text{Sup\_upto\_llist } Xs \ j = \text{Sup\_llist } (\text{ltake } (\text{eSuc } j) \text{ } Xs)$ 
  unfolding Sup_upto_llist_def Sup_llist_def
  by (smt (verit) Collect_cong Sup.SUP_cong illess_Suc_eq lnth_ltake less_llength_ltake mem_Collect_eq)
```

```
lemma Sup_upto_llist_enat_0[simp]:
  Sup_upto_llist Xs (enat 0) = (if  $\text{lnull } Xs$  then {} else lhd Xs)
proof (cases lnull Xs)
```

```

case True
then show ?thesis
  unfolding Sup upto llist def by auto
next
  case False
  show ?thesis
    unfolding Sup upto llist def image def by (simp add: lhd_conv_lnth enat_0 enat_0_iff)
qed

lemma Sup upto llist Suc[simp]:
  Sup upto llist Xs (enat (Suc j)) =
  Sup upto llist Xs (enat j) ∪ (if enat (Suc j) < llength Xs then lngth Xs (Suc j) else {})
  unfolding Sup upto llist def image def by (auto intro: le_SucI elim: le_SucE)

lemma Sup upto llist infinity[simp]: Sup upto llist Xs ∞ = Sup llist Xs
  unfolding Sup upto llist def Sup llist def by simp

lemma Sup upto llist 0[simp]: Sup upto llist Xs 0 = (if lnull Xs then {} else lhd Xs)
  unfolding zero_enat_def by (rule Sup upto llist_enat_0)

lemma Sup upto llist eSuc[simp]:
  Sup upto llist Xs (eSuc j) =
  (case j of
    enat k ⇒ Sup upto llist Xs (enat (Suc k))
    | ∞ ⇒ Sup llist Xs)
  by (auto simp: eSuc_enat split: enat.split)

lemma Sup upto llist mono[simp]: j ≤ k ⇒ Sup upto llist Xs j ⊆ Sup upto llist Xs k
  unfolding Sup upto llist def by auto

lemma Sup upto llist subset_Sup llist: Sup upto llist Xs j ⊆ Sup llist Xs
  unfolding Sup llist def Sup upto llist def by auto

lemma elem_Sup llist imp_Sup upto llist:
  x ∈ Sup llist Xs ⇒ ∃j < llength Xs. x ∈ Sup upto llist Xs (enat j)
  unfolding Sup llist def Sup upto llist def by blast

lemma lngth_subset_Sup upto llist: j < llength Xs ⇒ lngth Xs j ⊆ Sup upto llist Xs j
  unfolding Sup upto llist def by auto

lemma finite_Sup llist imp_Sup upto llist:
  assumes finite X and X ⊆ Sup llist Xs
  shows ∃k. X ⊆ Sup upto llist Xs (enat k)
  using assms
proof induct
  case (insert x X)
  then have x: x ∈ Sup llist Xs and X: X ⊆ Sup llist Xs
    by simp+
  from x obtain k where k: x ∈ Sup upto llist Xs (enat k)
    using elem_Sup llist imp_Sup upto llist by fast
  from X obtain k' where k': X ⊆ Sup upto llist Xs (enat k')
    using insert.hyps(3) by fast
  have insert x X ⊆ Sup upto llist Xs (max k k')
    using k k' by (metis (mono_tags) Sup upto llist_mono enat_ord_simps(1) insert_subset
      max.cobounded1 max.cobounded2 subset_iff)
  then show ?case
    by fast
qed simp

```

3.4 Liminf

```

definition Liminf llist :: 'a set llist ⇒ 'a set where
  Liminf llist Xs =
  (∪ i ∈ {i. enat i < llength Xs}. ∩ j ∈ {j. i ≤ j ∧ enat j < llength Xs}. lngth Xs j)

```

```

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_lll: ¬ lnull (lll Xs) ==> Liminf_llist Xs = Liminf_llist (lll Xs)
by (metis Liminf_llist_LCons lhd_LCons_lll lnull_lll)

lemma Liminf_llist_subset_Sup_llist: Liminf_llist Xs ⊆ Sup_llist Xs
unfolding Liminf_llist_def Sup_llist_def by fast

```

```

lemma image_Liminf_llist_subset:  $f` \text{Liminf\_llist } Ns \subseteq \text{Liminf\_llist} (\text{lmap} ((\lambda) f) Ns)$ 
  unfolding Liminf_llist_def by auto

lemma Liminf_llist_imp_exists_index:
   $x \in \text{Liminf\_llist } Xs \implies \exists i. \text{enat } i < \text{llength } Xs \wedge x \in \text{lnth } Xs i$ 
  unfolding Liminf_llist_def by auto

lemma not_Liminf_llist_imp_exists_index:
   $\neg \text{lnull } Xs \implies x \notin \text{Liminf\_llist } Xs \implies \text{enat } i < \text{llength } Xs \implies$ 
   $(\exists j. i \leq j \wedge \text{enat } j < \text{llength } Xs \wedge x \notin \text{lnth } Xs j)$ 
  unfolding Liminf_llist_def by auto

lemma finite_subset_Liminf_llist_imp_exists_index:
assumes
  nnil:  $\neg \text{lnull } Xs$  and
  fin:  $\text{finite } X$  and
  in_lim:  $X \subseteq \text{Liminf\_llist } Xs$ 
shows  $\exists i. \text{enat } i < \text{llength } Xs \wedge X \subseteq (\bigcap j \in \{j. i \leq j \wedge \text{enat } j < \text{llength } Xs\}. \text{lnth } Xs j)$ 
proof -
  show ?thesis
  proof (cases  $X = \{\}$ )
    case True
    then show ?thesis
    using nnil by (auto intro: exI[of _ 0] simp: zero_enat_def[symmetric])
  next
    case nemp: False
      have in_lim':
         $\forall x \in X. \exists i. \text{enat } i < \text{llength } Xs \wedge x \in (\bigcap j \in \{j. i \leq j \wedge \text{enat } j < \text{llength } Xs\}. \text{lnth } Xs j)$ 
        using in_lim[unfolded Liminf_llist_def] in_mono by fastforce
      obtain i_of where
        i_of_lt:  $\forall x \in X. \text{enat } (i\_of x) < \text{llength } Xs$  and
        in_inter:  $\forall x \in X. x \in (\bigcap j \in \{j. i\_of x \leq j \wedge \text{enat } j < \text{llength } Xs\}. \text{lnth } Xs j)$ 
        using bchoice[OF in_lim'] by blast

      define i_max where
        i_max = Max (i_of ` X)

      have i_max ∈ i_of ` X
        by (simp add: fin i_max_def nemp)
      then obtain x_max where
        x_max_in:  $x\_max \in X$  and
        i_max_is:  $i\_max = i\_of x\_max$ 
        unfolding i_max_def by blast
      have le_i_max:  $\forall x \in X. i\_of x \leq i\_max$ 
        unfolding i_max_def by (simp add: fin)
      have enat i_max < llength Xs
        using i_of_lt x_max_in i_max_is by auto
      moreover have  $X \subseteq (\bigcap j \in \{j. i\_max \leq j \wedge \text{enat } j < \text{llength } Xs\}. \text{lnth } Xs j)$ 
      proof
        fix x
        assume x_in:  $x \in X$ 
        then have x_in_inter:  $x \in (\bigcap j \in \{j. i\_of x \leq j \wedge \text{enat } j < \text{llength } Xs\}. \text{lnth } Xs j)$ 
        using in_inter by auto
        moreover have {j. i_max ≤ j ∧ enat j < llength Xs}
          ⊆ {j. i_of x ≤ j ∧ enat j < llength Xs}
        using x_in le_i_max by auto
        ultimately show x ∈ (bigcap{j. i_max ≤ j ∧ enat j < llength Xs}. lenth Xs j)
        by auto
      qed
      ultimately show ?thesis
      by auto
    qed
  qed

```

qed

```

lemma Liminf_llist_lmap_image:
  assumes f_inj: inj_on f (Sup_llist (lmap g xs))
  shows Liminf_llist (lmap (λx. f ` g x) xs) = f ` Liminf_llist (lmap g xs) (is ?lhs = ?rhs)
proof
  show ?lhs ⊆ ?rhs
  proof
    fix x
    assume x ∈ Liminf_llist (lmap (λx. f ` g x) xs)
    then obtain i where
      i_lt: enat i < llength xs and
      x_in_fgj: ∀ j. i ≤ j → enat j < llength xs → x ∈ f ` g (lnth xs j)
    unfolding Liminf_llist_def by auto

    have ex_in_gi: ∃ y. y ∈ g (lnth xs i) ∧ x = f y
      using f_inj i_lt x_in_fgj unfolding inj_on_def Sup_llist_def by auto
    have ∃ y. ∀ j. i ≤ j → enat j < llength xs → y ∈ g (lnth xs j) ∧ x = f y
      apply (rule exI[of _ SOME y. y ∈ g (lnth xs i) ∧ x = f y])
      using someI_ex[OF ex_in_gi] x_in_fgj f_inj i_lt x_in_fgj unfolding inj_on_def Sup_llist_def
      by simp (metis (no_types, lifting) imageE)
    then show x ∈ f ` Liminf_llist (lmap g xs)
      using i_lt unfolding Liminf_llist_def by auto
  qed
next
  show ?rhs ⊆ ?lhs
  using image_Liminf_llist_subset[of f lmap g xs, unfolded llist.map_comp] by auto
qed

```

```

lemma Liminf_llist_lmap_union:
  assumes ∀ x ∈ lset xs. ∀ Y ∈ lset xs. g x ∩ h Y = {}
  shows Liminf_llist (lmap (λx. g x ∪ h x) xs) =
    Liminf_llist (lmap g xs) ∪ Liminf_llist (lmap h xs) (is ?lhs = ?rhs)
proof (intro equalityI subsetI)
  fix x
  assume x_in: x ∈ ?lhs
  then obtain i where
    i_lt: enat i < llength xs and
    j: ∀ j. i ≤ j ∧ enat j < llength xs → x ∈ g (lnth xs j) ∨ x ∈ h (lnth xs j)
    using x_in[unfolded Liminf_llist_def, simplified] by blast

  then have (∃ i'. enat i' < llength xs ∧ (∀ j. i' ≤ j ∧ enat j < llength xs → x ∈ g (lnth xs j)))
    ∨ (∃ i'. enat i' < llength xs ∧ (∀ j. i' ≤ j ∧ enat j < llength xs → x ∈ h (lnth xs j)))
    using assms[unfolded disjoint_iff_not_equal] by (metis in_lset_conv_lnth)
  then show x ∈ ?rhs
    unfolding Liminf_llist_def by simp
next
  fix x
  show x ∈ ?rhs ⇒ x ∈ ?lhs
    using assms unfolding Liminf_llist_def by auto
qed

```

```

lemma Liminf_set_filter_commute:
  Liminf_llist (lmap (λX. {x ∈ X. p x}) Xs) = {x ∈ Liminf_llist Xs. p x}
  unfolding Liminf_llist_def by force

```

3.5 Liminf up-to

```

definition Liminf_uppto_llist :: 'a set llist ⇒ enat ⇒ 'a set where
  Liminf_uppto_llist Xs k =
    (⋃ i ∈ {i. enat i < llength Xs ∧ enat i ≤ k}.
     ⋂ j ∈ {j. i ≤ j ∧ enat j < llength Xs ∧ enat j ≤ k}. lnth Xs j)

```

```

lemma Liminf_uppto_llist_eq_Liminf_llist_ltake:

```

```

 $\text{Liminf\_upto\_llist } Xs \ j = \text{Liminf\_llist} (\text{ltake} (\text{eSuc} \ j) \ Xs)$ 
unfolding Liminf_upto_llist_def Liminf_llist_def
by (auto simp add: lnth_ltake less_llength_ltake)

lemma Liminf_upto_llist_enat[simp]:
 $\text{Liminf\_upto\_llist } Xs (\text{enat} \ k) =$ 
 $(\text{if enat } k < \text{llength } Xs \text{ then } \text{lnth } Xs \ k \text{ else if lnull } Xs \text{ then } \{\} \text{ else llast } Xs)$ 
proof (cases enat k < llength Xs)
  case True
  then show ?thesis
    unfolding Liminf_upto_llist_def by force
next
  case k_ge: False
  show ?thesis
proof (cases lnull Xs)
  case nil: True
  then show ?thesis
    unfolding Liminf_upto_llist_def by simp
next
  case nnil: False
  then obtain j where
    j: eSuc (enat j) = llength Xs
    using k_ge by (metis eSuc_enat_iff enat_le le_less_linear lhd_LCons_lltl_lllength_LCons)
  have fin: lfinit Xs
  using k_ge not_lfinit_llength by fastforce
  have le_k: enat i < llength Xs ∧ i ≤ k ↔ enat i < llength Xs for i
  using k_ge linear_order_le_less_subst2 by fastforce
  have Liminf_upto_llist Xs (enat k) = llast Xs
  using j nnil lfinit_Liminf_llist[OF fin] nnil
  unfolding Liminf_upto_llist_def Liminf_llist_def using llast_conv_lnth[OF j[symmetric]]
  by (simp add: le_k)
  then show ?thesis
    using k_ge nnil by simp
  qed
qed

lemma Liminf_upto_llist_infinity[simp]: Liminf_upto_llist Xs ∞ = Liminf_llist Xs
  unfolding Liminf_upto_llist_def Liminf_llist_def by simp

lemma Liminf_upto_llist_0[simp]:
 $\text{Liminf\_upto\_llist } Xs \ 0 = (\text{if lnull } Xs \text{ then } \{\} \text{ else lhd } Xs)$ 
unfolding Liminf_upto_llist_def image_def
by (simp add: enat_0[symmetric]) (simp add: enat_0_lnth_0_conv_lhd)

lemma Liminf_upto_llist_eSuc[simp]:
 $\text{Liminf\_upto\_llist } Xs (\text{eSuc} \ j) =$ 
 $(\text{case } j \text{ of}$ 
 $\quad \text{enat } k \Rightarrow \text{Liminf\_upto\_llist } Xs (\text{enat} (\text{Suc} \ k))$ 
 $\quad | \infty \Rightarrow \text{Liminf\_llist } Xs)$ 
by (auto simp: eSuc_enat split: enat.split)

lemma elem_Liminf_llist_imp_Liminf_upto_llist:
 $x \in \text{Liminf\_llist } Xs \implies$ 
 $\exists i < \text{llength } Xs. \forall j. i \leq j \wedge j < \text{llength } Xs \longrightarrow x \in \text{Liminf\_upto\_llist } Xs (\text{enat} \ j)$ 
unfolding Liminf_llist_def Liminf_upto_llist_def using enat_ord_simps(1) by force

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 list 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_nnnil: False

have r_xs: chain R xs

by (metis chain.simps ltl.simps(2) r_xxs xs_nnnil)

have mid': R (llast xs) (lhd ys)

by (metis llast_LCons_lnull_def mid xs_nnnil)

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_nnnil)

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 inf_chain_ldropn_chain: chain R xs ==> ¬ lfinite xs ==> chain R (ldropn n xs)

```

using chain.simps[of R xs] by (simp add: chain_ldropn_not_lfinite_llength)

lemma inf_chain_ltl_chain: chain R xs ==> ~ lfinite xs ==> chain R (ltl xs)
  by (metis inf_chain_ldropn_chain ldropn_0 ldropn_ltl)

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 length 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
qed

lemma chain_lmap:
assumes "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
qed

lemma chain_mono:
assumes "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 chain_ldropnI:
assumes
  rel: "∀ j. j ≥ i → enat (Suc j) < llength xs → R (lnth xs j) (lnth xs (Suc j))" and
  si_lt: "enat (Suc i) < llength xs"
shows "chain R (ldropn i xs)"
proof (rule lnth_rel_chain)
show "¬ lnull (ldropn i xs)"
using si_lt by (simp add: Suc_ile_eq less_le_not_le)
next
show "∀ j. enat (j + 1) < llength (ldropn i xs) →
R (lnth (ldropn i xs) j) (lnth (ldropn i xs) (j + 1))"
using rel
by (smt (verit, best) Suc_ile_eq add.commute ldropn_eq_LNil ldropn_ldropn_leD
le_add1 linorder_le_less_linear lnth_ldropn_order_less_imp_le plus_1_eq_Suc)
qed

lemma chain_ldropn_lmapI:
assumes
  rel: "∀ j. j ≥ i → enat (Suc j) < llength xs → R (f (lnth xs j)) (f (lnth xs (Suc j)))" and
  si_lt: "enat (Suc i) < llength xs"
shows "chain R (ldropn i (lmap f xs))"
proof -
have "chain R (lmap f (ldropn i xs))"
using chain_lmap[of λx y. R (f x) (f y) R f, of ldropn i xs] chain_ldropnI[OF rel si_lt]
by auto
thus ?thesis
by auto
qed

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_nnnil: False
  then have r_xs: chain R xs
  using r_xs_by (blast elim: chain.cases)
  then show ?thesis
  using ih[OF r_xs] xs_nnnil r_xs
  by (metis chain.cases converse_rtranclp_into_rtranclp lhd_LCons llast_LCons chain_not_lnull
       ltl_simp(2))
qed simp
qed simp

lemma tranclp_imp_exists_finite_chain_list:
   $R^{++} x y \implies \exists xs. \text{chain } R (\text{llist\_of } (x \# xs @ [y]))$ 
proof (induct rule: tranclp.induct)
  case (r_into_trancl x y)
  then have chain R (llist_of (x # [] @ [y]))
  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: chain R (llist_of (x # xs @ [y]))
    using ih by blast
  define ys where
    ys = xs @ [y]

  have chain R (llist_of (x # ys @ [z]))
  unfolding ys_def using r_yz chain_lappend[OF xs chain_singleton, of z]
  by (auto simp: lappend_llist_of_LCons llast_LCons)
  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))

```

4.2 A Coinductive Puzzle

```

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

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

lemma lhd_prepend[simp]: lhd (prepend xs ys) = (if xs  $\neq$  [] 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)  $\longleftrightarrow$  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]:  $\neg$  lnull ys  $\implies$  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
  lfinite xs ==> emb LNil xs
  | emb xs ys ==> emb (LCons x xs) (prepend zs (LCons x ys))

inductive-cases emb_LConsE: emb (LCons z zs) ys
inductive-cases emb_LNil1E: emb LNil ys
inductive-cases emb_LNil2E: emb xs LNil

lemma emb_lfinite:
  assumes emb xs ys
  shows lfinite ys ↔ lfinite xs
proof
  assume lfinite xs
  then show lfinite ys using assms
    by (induct xs arbitrary: ys rule: lfinite_induct)
      (auto simp: lnull_def neq_Nil_conv elim!: emb_LNil1E emb_LConsE)
next
  assume lfinite ys
  then show lfinite xs using assms
  proof (induction ys arbitrary: xs rule: lfinite_less_induct)

```

```

case (less ys)
from less.prems <lfinite ys> show ?case
by (cases xs)
  (auto simp: eSuc_enat elim!: emb_LNil1E emb_LConsE less.IH[rotated]
   dest!: lfinite_llength_enat)
qed
qed

inductive prepend_cong1 for X where
  prepend_cong1_base: X xs  $\Rightarrow$  prepend_cong1 X xs
| prepend_cong1-prepend: prepend_cong1 X ys  $\Rightarrow$  prepend_cong1 X (prepend xs ys)

lemma prepend_cong1_alt: prepend_cong1 X xs  $\leftrightarrow$  ( $\exists$  ys zs. xs = prepend ys zs  $\wedge$  X zs)
by (rule iffI, induct xs rule: prepend_cong1.induct)
  (force simp: prepend_prepend intro: prepend_cong1.intros exI[of _ []])+

lemma emb_prepend_coinduct_cong[rotated, case_names emb]:
assumes ( $\bigwedge$ x1 x2. X x1 x2  $\Rightarrow$ 
  ( $\exists$  xs. x1 = LNil  $\wedge$  x2 = xs  $\wedge$  lfinite xs)
   $\vee$  ( $\exists$  xs ys x zs. x1 = LCons x xs  $\wedge$  x2 = prepend zs (LCons x ys)
   $\wedge$  (prepend_cong1 (X xs) ys  $\vee$  emb xs ys))) (is  $\bigwedge$ x1 x2. X x1 x2  $\Rightarrow$  ?bisim x1 x2)
shows X x1 x2  $\Rightarrow$  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

lemma emb_prepend: emb xs ys  $\Rightarrow$  emb xs (prepend zs ys)
by (coinduction arbitrary: xs zs ys rule: emb_prepend_coinduct_cong)
  (force elim: emb.cases simp: prepend_prepend)

lemma prepend_cong1_emb: prepend_cong1 (emb xs) ys = emb xs ys
by (rule iffI, induct ys rule: prepend_cong1.induct)
  (simp_all add: emb_prepend prepend_cong1_base)

lemma prepend_cong_distrib:
  prepend_cong1 (P  $\sqcup$  Q) xs  $\leftrightarrow$  prepend_cong1 P xs  $\vee$  prepend_cong1 Q xs
  unfolding prepend_cong1_alt by auto

lemma emb_prepend_coinduct_aux[case_names emb]:
assumes X x1 x2 ( $\bigwedge$ x1 x2. X x1 x2  $\Rightarrow$ 
  ( $\exists$  xs. x1 = LNil  $\wedge$  x2 = xs  $\wedge$  lfinite xs)
   $\vee$  ( $\exists$  xs ys x zs. x1 = LCons x xs  $\wedge$  x2 = prepend zs (LCons x ys)
   $\wedge$  (prepend_cong1 (X xs  $\sqcup$  emb xs) ys)))
shows emb x1 x2
using assms unfolding prepend_cong_distrib prepend_cong1_emb
by (rule emb_prepend_coinduct_cong)

lemma emb_prepend_coinduct[rotated, case_names emb]:
assumes ( $\bigwedge$ x1 x2. X x1 x2  $\Rightarrow$ 
  ( $\exists$  xs. x1 = LNil  $\wedge$  x2 = xs  $\wedge$  lfinite xs)
   $\vee$  ( $\exists$  xs ys x zs zs'. x1 = LCons x xs  $\wedge$  x2 = prepend zs (LCons x (prepend zs' ys))
   $\wedge$  (X xs ys  $\vee$  emb xs ys)))
shows X x1 x2  $\Rightarrow$  emb x1 x2
by (erule emb_prepend_coinduct_aux[of X]) (force simp: prepend_cong1_alt dest: assms)

```

context
begin

private coinductive chain' **for** R **where**

```

chain' R (LCons x LNil)
| chain R (llist_of (x # zs @ [lhd xs])) ==>
  chain' R xs ==> chain' R (prepend zs xs)

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 exI[of _ z] exI[of _ []] exI[of _ zs])
         (auto intro: chain.intros)
  qed simp
qed

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 y zs ys)
    then show ?thesis
      by (intro disjI2 exI[of _ prepend zs ys] exI[of _ y])
         (force dest!: neq_Nil_conv[THEN iffD1] elim: chain.cases chain_nontrivE
          intro: chain'.intros)
  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 ==>
    (exists z. x = LCons z LNil) ∨
    (exists y xs zs. x = LCons y (prepend zs xs)) ∧
    (X xs ∨ chain R xs) ∧ chain R (llist_of (y # zs @ [lhd xs]))) ==> chain R x
  by (subst chain_chain', erule chain'.coinduct) (force simp: chain_chain')

end

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

private definition pick where
  pick x y = (SOME xs. chain R (llist_of (x # xs @ [y])))

private lemma pick[simp]:
  assumes R++ x y
  shows chain R (llist_of (x # pick x y @ [y]))
  unfolding pick_def using tranclp_imp_exists_finite_chain_list[THEN someI_ex, OF assms] by auto

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) =>
    LCons x (prepend (pick x y) (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)) =
  (LCons x (prepend (pick x y) (wit (LCons y xs))))
by (subst wit.code; auto)+

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

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

private lemma LNil_eq_iff_lnull: LNil = xs  $\longleftrightarrow$  lnull xs
by (cases xs) auto

lemma emb_wit[simp]: chain R++ xs  $\implies$  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 0 3 split: llist.splits intro: exI[of _ []] exI[of _ pick z _]
      intro!: exI[of _ :: _ llist])
  qed (auto intro!: exI[of _ LNil] exI[of _ []] emb.intros)
qed

private lemma lfinite_wit[simp]:
  assumes chain R++ xs
  shows lfinite (wit xs)  $\longleftrightarrow$  lfinite xs
  using emb_wit emb_lfinite assms by blast

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_lfinite assms)

lemma chain_tranclp_imp_exists_chain:
  chain R++ xs  $\implies$ 
   $\exists ys. \text{chain } R ys \wedge \text{emb } xs ys \wedge \text{lhd } ys = \text{lhd } xs \wedge \text{llast } ys = \text{llast } xs$ 
proof (intro exI[of _ wit xs] conjI, coinduction arbitrary: xs rule: chain_prepend_coinduct)
  case chain
  then show ?case
  by (subst (1 2) wit.code) (erule chain.cases; force split: llist.splits dest: pick)
qed auto

lemma emb_lset_mono[rotated]:  $x \in \text{lset } xs \implies \text{emb } xs ys \implies x \in \text{lset } ys$ 
by (induct x xs arbitrary: ys rule: llist.set_induct) (auto elim!: emb_LConsE)

lemma emb_Ball_lset_antimono:
  assumes emb Xs Ys
  shows  $\forall Y \in \text{lset } Ys. x \in Y \implies \forall X \in \text{lset } Xs. x \in X$ 
  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 not_lfinite Xs and not_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 not_lfinite Xs
  shows Liminf_llist Ys ⊆ Liminf_llist Xs
proof -
  from assms have not_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
  qed

```

```

moreover have  $\exists xsa\ ysa. \ ?xsa = lmap f xsa \wedge \ ?ysa = lmap f ysa \wedge emb xsa\ ysa$ 
  using  $emb'$  by blast
ultimately show  $?thesis$ 
  by blast
qed (simp add: emb_lfinite[OF emb])
qed

end

lemma chain_inf_llist_if_infinite_chain_function:
assumes  $\forall i. r(f(Suc i))(f i)$ 
shows  $\neg lfinite(inf_llist f) \wedge chain r^{-1-1}(inf_llist f)$ 
using assms by (simp add: lnth_rel_chain)

lemma infinite_chain_function_iff_infinite_chain_llist:
 $(\exists f. \forall i. r(f(Suc i))(f i)) \longleftrightarrow (\exists c. \neg lfinite c \wedge chain r^{-1-1} c)$ 
using chain_inf_llist_if_infinite_chain_function infinite_chain_lnth_rel by blast

lemma wfP_iff_no_infinite_down_chain_llist: wfP r  $\longleftrightarrow (\nexists c. \neg lfinite c \wedge chain r^{-1-1} c)$ 
proof -
have wfP r  $\longleftrightarrow wf\{(x, y). r x y\}$ 
  unfolding wfP_def by auto
also have ...  $\longleftrightarrow (\nexists f. \forall i. (f(Suc i), f i) \in \{(x, y). r x y\})$ 
  using wf_iff_no_infinite_down_chain by blast
also have ...  $\longleftrightarrow (\nexists f. \forall i. r(f(Suc i))(f i))$ 
  by auto
also have ...  $\longleftrightarrow (\nexists c. \neg lfinite c \wedge chain r^{-1-1} c)$ 
  using infinite_chain_function_iff_infinite_chain_llist by blast
finally show ?thesis
  by auto
qed

```

4.3 Full Chains

```

coinductive full_chain :: "('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a llist  $\Rightarrow$  bool for R :: "'a  $\Rightarrow$  'a  $\Rightarrow$  bool where
full_chain_singleton:  $(\forall y. \neg R x y) \Longrightarrow full\_chain R (LCons x LNil)$ 
| full_chain_cons:  $full\_chain R xs \Longrightarrow R x (lhd xs) \Longrightarrow full\_chain R (LCons x xs)$ 

```

```

lemma
full_chain_LNil[simp]:  $\neg full\_chain R LNil$  and
full_chain_not_lnull:  $full\_chain R xs \Longrightarrow \neg lnull xs$ 
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  $\longleftrightarrow chain R xs \wedge (lfinite xs \longrightarrow (\forall y. \neg R (llast xs) y))$ 
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_simp(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])

lemma full_chain_lnth_not_rel:
assumes
  full: full_chain R xs and
  sj: enat (Suc j) = llength xs
shows ¬ R (lnth xs j) y
proof -
have lfinite xs
  by (metis llength_eq_enat_lfiniteD sj)
hence ¬ R (llast xs) y
  using full_chain_iff_chain full by metis
thus ?thesis
  by (metis eSuc_enat_llast_conv_lnth sj)
qed

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 ∧ lhd ys = lhd xs ∧ llast ys = llast xs
proof -
obtain ys where ys:
  chain R ys emb xs ys 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(1,4) emb_lfinite[OF ys(2)] full unfolding full_chain_iff_chain by auto
then show ?thesis
  using ys(2-4) 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  $\Rightarrow$  bool where  
  is_neg  $L \equiv \neg$  is_pos  $L$ 
```

```
lemma Pos_atm_of_iff[simp]: Pos (atm_of  $L$ )  $= L \leftrightarrow$  is_pos  $L$   
  by (cases  $L$ ) simp+
```

```
lemma Neg_atm_of_iff[simp]: Neg (atm_of  $L$ )  $= L \leftrightarrow$  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:  $(\exists L. P\ L) \leftrightarrow (\exists A. P\ (Pos\ A) \vee P\ (Neg\ A))$   
  by (metis literal.exhaust)
```

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

```
definition uminus_literal :: ' $a$  literal  $\Rightarrow$  ' $a$  literal where  
  uminus  $L = (\text{if } is\_pos\ L \text{ then } Neg \text{ else } Pos)\ (atm\_of\ L)$ 
```

```
instance ..
```

```
end
```

```
lemma  
  uminus_Pos[simp]:  $\neg$  Pos  $A =$  Neg  $A$  and  
  uminus_Neg[simp]:  $\neg$  Neg  $A =$  Pos  $A$   
  unfolding uminus_literal_def by simp_all
```

```
lemma atm_of_uminus[simp]: atm_of  $(\neg L) = atm\_of\ L$   
  by (case_tac  $L$ , auto)
```

```
lemma uminus_of_uminus_id[simp]:  $\neg (\neg (x :: 'v\ literal)) = x$   
  by (simp add: uminus_literal_def)
```

```
lemma uminus_not_id[simp]:  $x \neq \neg (x :: 'v\ literal)$   
  by (case_tac  $x$ ) auto
```

```
lemma uminus_not_id'[simp]:  $\neg x \neq (x :: 'v\ literal)$   
  by (case_tac  $x$ , auto)
```

```
lemma uminus_eq_inj[iff]:  $\neg (a :: 'v\ literal) = \neg b \leftrightarrow a = b$   
  by (case_tac  $a$ ; case_tac  $b$ ) auto+
```

```
lemma uminus_lit_swap:  $(a :: 'a\ literal) = \neg b \leftrightarrow \neg a = b$   
  by auto
```

```
lemma is_pos_neg_not_is_pos: is_pos  $(\neg L) \leftrightarrow \neg$  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 ↔ A < B
  unfolding less_literal_def by simp

lemma pos_less_neg_iff[simp]: Pos A < Neg B ↔ A ≤ B
  unfolding less_literal_def by (auto simp: less_le_not_le)

lemma neg_less_pos_iff[simp]: Neg A < Pos B ↔ A < B
  unfolding less_literal_def by simp

lemma neg_less_neg_iff[simp]: Neg A < Neg B ↔ A < B
  unfolding less_literal_def by simp

lemma pos_le_neg[simp]: Pos A ≤ Neg A
  unfolding less_eq_literal_def by simp

lemma pos_le_pos_iff[simp]: Pos A ≤ Pos B ↔ A ≤ B
  unfolding less_eq_literal_def by (auto simp: less_le_not_le)

lemma pos_le_neg_iff[simp]: Pos A ≤ Neg B ↔ A ≤ B
  unfolding less_eq_literal_def by (auto simp: less_imp_le)

lemma neg_le_pos_iff[simp]: Neg A ≤ Pos B ↔ A < B
  unfolding less_eq_literal_def by simp

lemma neg_le_neg_iff[simp]: Neg A ≤ Neg B ↔ A ≤ B
  unfolding less_eq_literal_def by (auto simp: less_imp_le)

lemma leq_imp_less_eq_atm_of: L ≤ M ⇒ atm_of L ≤ 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:  $\bigwedge L. (\bigwedge M. M < L \Rightarrow P M) \Rightarrow P L$ 
  have  $\bigwedge x. (\bigwedge y. y < x \Rightarrow P (\text{Pos } y) \wedge P (\text{Neg } y)) \Rightarrow P (\text{Pos } x) \wedge P (\text{Neg } x)$ 
    by (rule conjI[OF ih ih])
    (auto simp: less_literal_def atm_of_def split: literal.splits intro: ih>)
  then have  $\bigwedge A. P (\text{Pos } A) \wedge P (\text{Neg } A)$ 
    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  $\Rightarrow$  'b)  $\Rightarrow$  'a clause  $\Rightarrow$  'b clause where
  map_clause f  $\equiv$  image_mset (map_literal f)

abbreviation rel_clause :: ('a  $\Rightarrow$  'b  $\Rightarrow$  bool)  $\Rightarrow$  'a clause  $\Rightarrow$  'b clause  $\Rightarrow$  bool where
  rel_clause R  $\equiv$  rel_mset (rel_literal R)

abbreviation poss :: 'a multiset  $\Rightarrow$  'a clause where poss AA  $\equiv$  {#Pos A. A  $\in\#$  AA#}
abbreviation negs :: 'a multiset  $\Rightarrow$  'a clause where negs AA  $\equiv$  {#Neg A. A  $\in\#$  AA#}

lemma Max_in_lits: C  $\neq \{\#\} \Rightarrow \text{Max\_mset } C \in\# C$ 
  by simp

lemma Max_atm_of_set_mset_commute: C  $\neq \{\#\} \Rightarrow \text{Max} (\text{atm\_of } \text{set\_mset } C) = \text{atm\_of} (\text{Max\_mset } C)$ 
  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  $\in\# D$ 
  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  $\Rightarrow$  Neg A  $\notin\# C$ 
  using Max_pos_neg_less_multiset by blast

lemma less_eq_Max_lit: C  $\neq \{\#\} \Rightarrow C \leq D \Rightarrow \text{Max\_mset } C \leq \text{Max\_mset } D$ 
proof (unfold less_eq_multisetHO)
  assume
    ne: C  $\neq \{\#\}$  and
    ex_gt:  $\forall x. \text{count } D x < \text{count } C x \longrightarrow (\exists y > x. \text{count } C y < \text{count } D y)$ 
  from ne have Max_mset C  $\in\# C$ 
    by (fast intro: Max_in_lits)
  then have  $\exists l. l \in\# D \wedge \neg l < \text{Max\_mset } C$ 
    using ex_gt by (metis count_greater_zero_iff count_inI less_not_sym)

```

```

then have  $\neg \text{Max\_mset } D < \text{Max\_mset } C$ 
  by (metis Max.coboundedI[OF finite_set_mset] le_less_trans)
then show ?thesis
  by simp
qed

definition atms_of :: 'a clause  $\Rightarrow$  '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  $\Rightarrow$  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)  $\Rightarrow$  x  $\in$  atms_of A
  by (auto simp: atms_of_def dest: in_diffD)

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

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

lemma atm_imp_pos_or_neg_lit: A  $\in$  atms_of C  $\Rightarrow$  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  $\Rightarrow$  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_pos[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  $\neq$  {#}  $\Rightarrow$  C  $\leq$  D  $\Rightarrow$  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  $\Rightarrow$  C  $\leq$  D  $\Rightarrow$  A  $\in$  atms_of C  $\Rightarrow$  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_Cmpl_self_eq)

lemma atm_of_Max_lit[simp]: C  $\neq \{\#\}$   $\Rightarrow$  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  $\neq \{\#\}$   $\Rightarrow$  Max_mset C = Pos (Max (atms_of C))  $\vee$  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  $\Rightarrow$  B  $<$  A)  $\Rightarrow$  L  $\in \#$  C  $\Rightarrow$  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  $\Rightarrow$  B  $\leq$  A)  $\Rightarrow$  L  $\in \#$  C  $\Rightarrow$  L  $\leq$  Neg A
  unfolding less_eq_literal_def by (simp add: atm_of_lit_in_atms_of)

end

```

6 Herbrand Interpretation

```

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  $\models_l L \leftrightarrow$  (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  $\leftrightarrow$  A  $\in$  I
  I  $\models_l$  Neg A  $\leftrightarrow$  A  $\notin$  I
  unfolding true_lit_def by auto

```

```

lemma true_lit_iff[iff]: I  $\models_l L \leftrightarrow$  ( $\exists A. L =$  Pos A  $\wedge$  A  $\in$  I  $\vee$  L = Neg A  $\wedge$  A  $\notin$  I)
  by (cases L) simp+

```

```

definition true_cls :: 'a interp  $\Rightarrow$  'a clause  $\Rightarrow$  bool (infix  $\models_l 50$ ) where
  I  $\models_l C \leftrightarrow$  ( $\exists L \in \# C. I \models_l L$ )

```

```

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

```

```

lemma true_cls_singleton[iff]: I  $\models \{\#L\} \leftrightarrow$  I  $\models_l L$ 
  unfolding true_cls_def by simp

```

```

lemma true_cls_add_mset[iff]: I  $\models$  add_mset C D  $\leftrightarrow$  I  $\models_l C \vee I \models D$ 
  unfolding true_cls_def by auto

```

```

lemma true_cls_union[iff]: I  $\models C + D \leftrightarrow$  I  $\models C \vee I \models D$ 
  unfolding true_cls_def by auto

```

```

lemma true_cls_mono: set_mset C  $\subseteq$  set_mset D  $\Rightarrow$  I  $\models C \Rightarrow I \models D$ 

```

```

unfolding true_cls_def subset_eq by metis

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

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

lemma pos_literal_in_imp_true_cls[intro]:  $Pos A \in\# C \implies A \in I \implies I \models C$ 
  using true_cls_def by blast

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

lemma pos_neg_in_imp_true:  $Pos A \in\# C \implies Neg A \in\# C \implies I \models C$ 
  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]:  $I \models_s insert C DD \longleftrightarrow I \models C \wedge I \models_s DD$ 
  unfolding true_clss_def by blast

lemma true_clss_union[iff]:  $I \models_s CC \cup DD \longleftrightarrow I \models_s CC \wedge I \models_s DD$ 
  unfolding true_clss_def by blast

lemma true_clss_Union[iff]:  $I \models_s \bigcup CCC \longleftrightarrow (\forall CC \in CCC. I \models_s CC)$ 
  unfolding true_clss_def by simp

lemma true_clss_mono:  $DD \subseteq CC \implies I \models_s CC \implies I \models_s DD$ 
  by (simp add: subsetD true_clss_def)

lemma true_clss_mono_strong:  $(\forall D \in DD. \exists C \in CC. C \subseteq\# D) \implies I \models_s CC \implies I \models_s DD$ 
  unfolding true_clss_def true_cls_def true_lit_def by (meson mset_subset_eqD)

lemma true_clss_subclause:  $C \subseteq\# D \implies I \models_s \{C\} \implies I \models_s \{D\}$ 
  by (rule true_clss_mono_strong[of _ {C}]) auto

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

lemma satisfiable_antimono:  $CC \subseteq DD \implies satisfiable DD \implies satisfiable CC$ 
  using true_clss_mono by blast

lemma unsatisfiable_mono:  $CC \subseteq DD \implies \neg satisfiable CC \implies \neg satisfiable DD$ 
  using satisfiable_antimono by blast

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]:  $I \models_m \{\#C\#} \leftrightarrow I \models C$ 
  by (simp add: true_cls_mset_def)

lemma true_cls_mset_union[iff]:  $I \models_m CC + DD \leftrightarrow I \models_m CC \wedge I \models_m DD$ 
  unfolding true_cls_mset_def by auto

lemma true_cls_mset_Union[iff]:  $I \models_m \sum_{\#} CCC \leftrightarrow (\forall CC \in \# CCC. I \models_m CC)$ 
  unfolding true_cls_mset_def by simp

lemma true_cls_mset_add_mset[iff]:  $I \models_m add\_mset C CC \leftrightarrow I \models C \wedge I \models_m CC$ 
  unfolding true_cls_mset_def by auto

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

lemma true_cls_mset_mono:  $set\_mset DD \subseteq set\_mset CC \implies I \models_m CC \implies I \models_m DD$ 
  unfolding true_cls_mset_def subset_iff by auto

lemma true_cls_mset_mono_strong:  $(\forall D \in \# DD. \exists C \in \# CC. C \subseteq \# D) \implies I \models_m CC \implies I \models_m DD$ 
  unfolding true_cls_mset_def true_cls_def true_lit_def by (meson mset_subset_eqD)

lemma true_clss_set_mset[iff]:  $I \models_s set\_mset CC \leftrightarrow I \models_m CC$ 
  unfolding true_clss_def true_cls_mset_def by auto

lemma true_clss_mset_set[simp]:  $finite CC \implies I \models_m mset\_set CC \leftrightarrow I \models_s CC$ 
  unfolding true_clss_def true_cls_mset_def by auto

lemma true_cls_mset_true_cls:  $I \models_m CC \implies C \in \# CC \implies I \models C$ 
  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:  $\forall i. f(Suc i) \leq f i$ 
  shows  $\exists l. \forall l' \geq l. f l' = f(Suc l')$ 
proof (rule ccontr)
  assume a:  $\nexists l. \forall l' \geq l. f l' = f(Suc l')$ 
  have  $\forall i. \exists i'. i' > i \wedge f i' < f i$ 
  proof
    fix i
    from a have  $\exists l' \geq i. f l' \neq f(Suc l')$ 
      by auto
    then obtain l' where
      l'_p:  $l' \geq i \wedge f l' \neq f(Suc l')$ 
      by metis
    then have  $f l' > f(Suc l')$ 
      using leq_leq_less_or_eq by auto
    moreover have  $f i \geq f l'$ 
      using leq_l'_p by (induction l' arbitrary: i) (blast intro: lift_Suc_antimono_le)+
```

```

ultimately show  $\exists i' > i. f i' < f i$ 
  using l'_p less_le_trans by blast
qed
then obtain g_sm :: nat  $\Rightarrow$  nat where
  g_sm_p:  $\forall i. g_{sm} i > i \wedge f(g_{sm} i) < f i$ 
  by metis

define c :: nat  $\Rightarrow$  nat where
   $\lambda n. c n = (g_{sm} \wedge n) 0$ 

have  $f(c i) > f(c(Suc i))$  for i
  by (induction i) (auto simp: c_def g_sm_p)
then have  $\forall i. (f \circ c) i > (f \circ c)(Suc i)$ 
  by auto
then have  $\exists fc :: nat \Rightarrow nat. \forall i. fc i > fc(Suc i)$ 
  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  $\Rightarrow$  's  $\Rightarrow$  'a and
  id_subst :: 's and
  comp_subst :: 's  $\Rightarrow$  's  $\Rightarrow$  's
begin

abbreviation subst_atm_abbrev :: 'a  $\Rightarrow$  's  $\Rightarrow$  'a (infixl  $\cdot a$  67) where
  subst_atm_abbrev  $\equiv$  subst_atm

abbreviation comp_subst_abbrev :: 's  $\Rightarrow$  's  $\Rightarrow$  's (infixl  $\odot$  67) where
  comp_subst_abbrev  $\equiv$  comp_subst

definition comp_substs :: 's list  $\Rightarrow$  's list  $\Rightarrow$  's list (infixl  $\odot s$  67) where
   $\sigma s \odot \tau s = map2 comp\_subst \sigma s \tau s$ 

definition subst_atms :: 'a set  $\Rightarrow$  's  $\Rightarrow$  'a set (infixl  $\cdot as$  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
   $AAs \cdot aml \sigma = map2(\cdot am) AAs \sigma$ 

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

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  $\leftrightarrow$  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  $\leftrightarrow$  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  $\leftrightarrow$  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  $\leftrightarrow$  cl 67) where
  Cs  $\cdot$  cl  $\sigma$ s = map2 (.) Cs  $\sigma$ s

definition subst_cls_mset :: 'a clause multiset  $\Rightarrow$  's  $\Rightarrow$  'a clause multiset (infixl  $\leftrightarrow$  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 :: 'a clause  $\Rightarrow$  'a clause  $\Rightarrow$  bool where
  generalizes C D  $\longleftrightarrow$  ( $\exists \sigma$ .  $C \cdot \sigma = D$ )

definition strictly_generalizes :: 'a clause  $\Rightarrow$  'a clause  $\Rightarrow$  bool where
  strictly_generalizes C D  $\longleftrightarrow$  generalizes C D  $\wedge$   $\neg$  generalizes 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 C D  $\wedge$  generalizes 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 = ( $\forall A \in AA. \text{is\_ground\_atm } A$ )
definition is_ground_atm_list :: 'a list  $\Rightarrow$  bool where
  is_ground_atm_list As  $\longleftrightarrow$  ( $\forall A \in \text{set } As. \text{is\_ground\_atm } A$ )
definition is_ground_atm_mset :: 'a multiset  $\Rightarrow$  bool where
  is_ground_atm_mset AA  $\longleftrightarrow$  ( $\forall A. A \in \# AA \longrightarrow \text{is\_ground\_atm } A$ )
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  $\longleftrightarrow$  ( $\forall L. L \in \# C \longrightarrow \text{is\_ground\_lit } L$ )
definition is_ground_clss :: 'a clause set  $\Rightarrow$  bool where
  is_ground_clss CC  $\longleftrightarrow$  ( $\forall C \in \text{set } CC. \text{is\_ground\_cls } C$ )
definition is_ground_cls_list :: 'a clause list  $\Rightarrow$  bool where
  is_ground_cls_list CC  $\longleftrightarrow$  ( $\forall C \in \text{set } CC. \text{is\_ground\_cls } C$ )
definition is_ground_subst :: 's  $\Rightarrow$  bool where
  is_ground_subst  $\sigma$   $\longleftrightarrow$  ( $\forall A. \text{is\_ground\_atm } (A \cdot a \sigma)$ )
definition is_ground_subst_list :: 's list  $\Rightarrow$  bool where
  is_ground_subst_list  $\sigma s$   $\longleftrightarrow$  ( $\forall \sigma \in \text{set } \sigma s. \text{is\_ground\_subst } \sigma$ )
definition grounding_of_cls :: 'a clause  $\Rightarrow$  'a clause set where
  grounding_of_cls C = { $C \cdot \sigma \mid \sigma. \text{is\_ground\_subst } \sigma$ }
definition grounding_of_clss :: 'a clause set  $\Rightarrow$  'a clause set where
  grounding_of_clss CC = ( $\bigcup C \in CC. \text{grounding\_of\_cls } C$ )
definition is_unifier :: 's  $\Rightarrow$  'a set  $\Rightarrow$  bool where
  is_unifier  $\sigma AA$   $\longleftrightarrow$  card (AA  $\cdot$  as  $\sigma$ )  $\leq 1$ 
definition is_unifiers :: 's  $\Rightarrow$  'a set set  $\Rightarrow$  bool where
  is_unifiers  $\sigma AAA$   $\longleftrightarrow$  ( $\forall AA \in AAA. \text{is\_unifier } \sigma AA$ )
definition is_mgu :: 's  $\Rightarrow$  'a set set  $\Rightarrow$  bool where
  is_mgu  $\sigma AAA$   $\longleftrightarrow$  is_unifiers  $\sigma AAA \wedge (\forall \tau. \text{is\_unifiers } \tau AAA \longrightarrow (\exists \gamma. \tau = \sigma \odot \gamma))$ 
definition is_imgu :: 's  $\Rightarrow$  'a set set  $\Rightarrow$  bool where
  is_imgu  $\sigma AAA$   $\longleftrightarrow$  is_unifiers  $\sigma AAA \wedge (\forall \tau. \text{is\_unifiers } \tau AAA \longrightarrow \tau = \sigma \odot \tau)$ 
definition var_disjoint :: 'a clause list  $\Rightarrow$  bool where
  var_disjoint Cs  $\longleftrightarrow$  ( $\forall \sigma s. \text{length } \sigma s = \text{length } Cs \longrightarrow (\exists \tau. \forall i < \text{length } Cs. \forall S. S \subseteq \# Cs ! i \longrightarrow S \cdot \sigma s ! i = S \cdot \tau)$ )
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 +
assumes
  subst_atm_id_subst[simp]:  $A \cdot a \text{id\_subst} = A$  and
  subst_atm_comp_subst[simp]:  $A \cdot a (\sigma \odot \tau) = (A \cdot a \sigma) \cdot a \tau$  and
  subst_ext: ( $\bigwedge A. A \cdot a \sigma = A \cdot a \tau \implies \sigma = \tau$ ) and
  make_ground_subst: is_ground_cls (C  $\cdot \sigma$ )  $\implies \exists \tau. \text{is\_ground\_subst } \tau \wedge C \cdot \tau = C \cdot \sigma$  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  $\sigma s$ ) 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  $\sigma s$ ) 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]:  $AAs \cdot aml replicate (length AAs) id\_subst = AAs$ 
  unfolding subst_atm_mset_lists_def by (induct AAs) auto

lemma subst_lit_id_subst[simp]:  $L \cdot l id\_subst = L$ 
  unfolding subst_lit_def by (simp add: literal.map_ident)

lemma subst_cls_id_subst[simp]:  $C \cdot id\_subst = C$ 
  unfolding subst_cls_def by simp

lemma subst_clss_id_subst[simp]:  $CC \cdot cs id\_subst = CC$ 
  unfolding subst_clss_def by simp

lemma subst_cls_list_id_subst[simp]:  $Cs \cdot cl id\_subst = Cs$ 
  unfolding subst_cls_list_def by simp

lemma subst_cls_lists_id_subst[simp]:  $Cs \cdot cl replicate (length Cs) id\_subst = Cs$ 
  unfolding subst_cls_lists_def by (induct Cs) auto

lemma subst_cls_mset_id_subst[simp]:  $CC \cdot cm id\_subst = CC$ 
  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]:  $AA \cdot as (\tau \odot \sigma) = AA \cdot as \tau \cdot as \sigma$ 
  unfolding subst_atms_def by auto

```

```

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

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

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

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

lemma subst_atm_mset_lists_comp_substs[simp]: AAs ..aml ( $\tau s \odot s \sigma s$ ) = AAs ..aml  $\tau s$  ..aml  $\sigma s$ 
  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  $\tau$  · l  $\sigma$ 
  unfolding subst_lit_def by (auto simp: literal.map_comp o_def)

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

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

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

lemma subst_cls_lists_comp_substs[simp]: Cs ..cl ( $\tau s \odot s \sigma s$ ) = Cs ..cl  $\tau s$  ..cl  $\sigma s$ 
  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 · cm ( $\tau \odot \sigma$ ) = CC · cm  $\tau$  · 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]:  $(\text{Neg } A' \cdot l \eta) = \text{Neg } A \longleftrightarrow A' \cdot a \eta = A$ 
  by (simp add: subst_lit_def)

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

```

```

lemma subst_cls_negs[simp]: (negs AA) · σ = negs (AA · am σ)
  unfolding subst_cls_def subst_lit_def subst_atm_mset_def by auto

lemma subst_cls_pos[simp]: (poss AA) · σ = poss (AA · am σ)
  unfolding subst_cls_def subst_lit_def subst_atm_mset_def by auto

lemma atms_of_subst_atms: atms_of C · as σ = atms_of (C · σ)
proof -
  have atms_of (C · σ) = set_mset (image_mset atm_of (image_mset (map_literal (λA. A · a σ)) C))
    unfolding subst_cls_def subst_atms_def subst_lit_def atms_of_def by auto
  also have ... = set_mset (image_mset (λA. A · a σ) (image_mset atm_of C))
    by simp (meson literal.map_sel)
  finally show atms_of C · as σ = atms_of (C · σ)
    unfolding subst_atms_def atms_of_def by auto
qed

lemma in_image_Neg_is_neg[simp]: L · l σ ∈ Neg ` AA ⇒ 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 · l σ ∈# (negs AA) · τ ⇒ is_neg L
  by simp

lemma subst_lit_in_negs_is_neg: L · l σ ∈# negs AA ⇒ is_neg L
  by simp

```

7.3.7 Substitution on Empty

```

lemma subst_atms_empty[simp]: {} · as σ = {}
  unfolding subst_atms_def by auto

lemma subst_atmss_empty[simp]: {} · ass σ = {}
  unfolding subst_atmss_def by auto

lemma comp_substs_empty_iff[simp]: σs ⊕s ηs = [] ↔ σs = [] ∨ ηs = []
  using comp_substs_def map2_empty_iff by auto

lemma subst_atm_list_empty[simp]: [] · al σ = []
  unfolding subst_atm_list_def by auto

lemma subst_atm_mset_empty[simp]: {#} · am σ = {#}
  unfolding subst_atm_mset_def by auto

lemma subst_atm_mset_list_empty[simp]: [] · aml σ = []
  unfolding subst_atm_mset_list_def by auto

lemma subst_atm_mset_lists_empty[simp]: [] ..aml σs = []
  unfolding subst_atm_mset_lists_def by auto

lemma subst_cls_empty[simp]: {#} · σ = {#}
  unfolding subst_cls_def by auto

lemma subst_clss_empty[simp]: {} · cs σ = {}
  unfolding subst_clss_def by auto

lemma subst_cls_list_empty[simp]: [] · cl σ = []
  unfolding subst_cls_list_def by auto

lemma subst_cls_lists_empty[simp]: [] ..cl σs = []
  unfolding subst_cls_lists_def by auto

lemma subst_scls_mset_empty[simp]: {#} · cm σ = {#}
  unfolding subst_scls_mset_def by auto

```

```

lemma subst_atms_empty_iff[simp]: AA ·as η = {}  $\longleftrightarrow$  AA = {}
  unfolding subst_atms_def by auto

lemma subst_atmss_empty_iff[simp]: AAA ·ass η = {}  $\longleftrightarrow$  AAA = {}
  unfolding subst_atmss_def by auto

lemma subst_atm_list_empty_iff[simp]: As ·al η = []  $\longleftrightarrow$  As = []
  unfolding subst_atm_list_def by auto

lemma subst_atm_mset_empty_iff[simp]: AA ·am η = {#}  $\longleftrightarrow$  AA = {#}
  unfolding subst_atm_mset_def by auto

lemma subst_atm_mset_list_empty_iff[simp]: AAs ·aml η = []  $\longleftrightarrow$  AAs = []
  unfolding subst_atm_mset_list_def by auto

lemma subst_atm_mset_lists_empty_iff[simp]: AAs ..aml ηs = []  $\longleftrightarrow$  (AAs = []  $\vee$  ηs = [])
  using map2_empty_iff subst_atm_mset_lists_def by auto

lemma subst_cls_empty_iff[simp]: C ·η = {#}  $\longleftrightarrow$  C = {#}
  unfolding subst_cls_def by auto

lemma subst_clss_empty_iff[simp]: CC ·cs η = {}  $\longleftrightarrow$  CC = {}
  unfolding subst_clss_def by auto

lemma subst_cls_list_empty_iff[simp]: Cs ·cl η = []  $\longleftrightarrow$  Cs = []
  unfolding subst_cls_list_def by auto

lemma subst_cls_lists_empty_iff[simp]: Cs ..cl ηs = []  $\longleftrightarrow$  Cs = []  $\vee$  ηs = []
  using map2_empty_iff subst_cls_lists_def by auto

lemma subst_cls_mset_empty_iff[simp]: CC ·cm η = {#}  $\longleftrightarrow$  CC = {#}
  unfolding subst_cls_mset_def by auto

```

7.3.8 Substitution on a Union

```

lemma subst_atms_union[simp]: (AA  $\cup$  BB) ·as σ = AA ·as σ  $\cup$  BB ·as σ
  unfolding subst_atms_def by auto

lemma subst_atmss_union[simp]: (AAA  $\cup$  BBB) ·ass σ = AAA ·ass σ  $\cup$  BBB ·ass σ
  unfolding subst_atmss_def by auto

lemma subst_atm_list_append[simp]: (As @ Bs) ·al σ = As ·al σ @ Bs ·al σ
  unfolding subst_atm_list_def by auto

lemma subst_atm_mset_union[simp]: (AA + BB) ·am σ = AA ·am σ + BB ·am σ
  unfolding subst_atm_mset_def by auto

lemma subst_atm_mset_list_append[simp]: (AAs @ BBs) ·aml σ = AAs ·aml σ @ BBs ·aml σ
  unfolding subst_atm_mset_list_def by auto

lemma subst_cls_union[simp]: (C + D) ·σ = C ·σ + D ·σ
  unfolding subst_cls_def by auto

lemma subst_clss_union[simp]: (CC  $\cup$  DD) ·cs σ = CC ·cs σ  $\cup$  DD ·cs σ
  unfolding subst_clss_def by auto

lemma subst_cls_list_append[simp]: (Cs @ Ds) ·cl σ = Cs ·cl σ @ Ds ·cl σ
  unfolding subst_cls_list_def by auto

lemma subst_cls_lists_append[simp]:
  length Cs = length σs  $\Longrightarrow$  length Cs' = length σs'  $\Longrightarrow$ 
  (Cs @ Cs') ..cl (σs @ σs') = Cs ..cl σs @ Cs' ..cl σs'
  unfolding subst_cls_lists_def by auto

```

```
lemma subst_cls_mset_union[simp]: (CC + DD) ·cm σ = CC ·cm σ + DD ·cm σ
  unfolding subst_cls_mset_def by auto
```

7.3.9 Substitution on a Singleton

```
lemma subst_atms_single[simp]: {A} ·as σ = {A ·a σ}
  unfolding subst_atms_def by auto
```

```
lemma subst_atmss_single[simp]: {AA} ·ass σ = {AA ·as σ}
  unfolding subst_atmss_def by auto
```

```
lemma subst_atm_list_single[simp]: [A] ·al σ = [A ·a σ]
  unfolding subst_atm_list_def by auto
```

```
lemma subst_atm_mset_single[simp]: {#A#} ·am σ = {#A ·a σ#}
  unfolding subst_atm_mset_def by auto
```

```
lemma subst_atm_mset_list[simp]: [AA] ·aml σ = [AA ·am σ]
  unfolding subst_atm_mset_list_def by auto
```

```
lemma subst_cls_single[simp]: {#L#} ·σ = {#L ·l σ#}
  by simp
```

```
lemma subst_clss_single[simp]: {C} ·cs σ = {C ·σ}
  unfolding subst_clss_def by auto
```

```
lemma subst_cls_list_single[simp]: [C] ·cl σ = [C ·σ]
  unfolding subst_cls_list_def by auto
```

```
lemma subst_cls_lists_single[simp]: [C] ..cl [σ] = [C ·σ]
  unfolding subst_cls_lists_def by auto
```

```
lemma subst_cls_mset_single[simp]: {#C#} ·cm σ = {#C ·σ#}
  by simp
```

7.3.10 Substitution on (#)

```
lemma subst_atm_list_Cons[simp]: (A # As) ·al σ = A ·a σ # As ·al σ
  unfolding subst_atm_list_def by auto
```

```
lemma subst_atm_mset_list_Cons[simp]: (A # As) ·aml σ = A ·am σ # As ·aml σ
  unfolding subst_atm_mset_list_def by auto
```

```
lemma subst_atm_mset_lists_Cons[simp]: (C # Cs) ..aml (σ # σs) = C ·am σ # Cs ..aml σs
  unfolding subst_atm_mset_lists_def by auto
```

```
lemma subst_cls_list_Cons[simp]: (C # Cs) ·cl σ = C ·σ # Cs ·cl σ
  unfolding subst_cls_list_def by auto
```

```
lemma subst_cls_lists_Cons[simp]: (C # Cs) ..cl (σ # σs) = C ·σ # Cs ..cl σs
  unfolding subst_cls_lists_def by auto
```

7.3.11 Substitution on tl

```
lemma subst_atm_list_tl[simp]: tl (As ·al σ) = tl As ·al σ
  by (cases As) auto
```

```
lemma subst_atm_mset_list_tl[simp]: tl (AAs ·aml σ) = tl AAs ·aml σ
  by (cases AAs) auto
```

```
lemma subst_cls_list_tl[simp]: tl (Cs ·cl σ) = tl Cs ·cl σ
  by (cases Cs) auto
```

```
lemma subst_cls_lists_tl[simp]: length Cs = length σs ==> tl (Cs ..cl σs) = tl Cs ..cl tl σs
```

by (cases Cs ; cases σs) auto

7.3.12 Substitution on (!)

lemma $comp_subssts_nth[simp]$:

$length \tau s = length \sigma s \implies i < length \tau s \implies (\tau s \odot s \sigma s) ! i = (\tau s ! i) \odot (\sigma s ! i)$
by (simp add: $comp_subssts_def$)

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

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

lemma $subst_atm_mset_lists_nth[simp]$:

$length AAs = length \sigma s \implies i < length AAs \implies (AAs \cdot aml \sigma s) ! i = (AAs ! i) \cdot am (\sigma s ! i)$
unfolding $subst_atm_mset_lists_def$ **by** auto

lemma $subst_cls_list_nth[simp]$: $i < length Cs \implies (Cs \cdot cl \tau) ! i = (Cs ! i) \cdot \tau$

unfolding $subst_cls_list_def$ **using** $less_Suc_eq_0_disj nth_map$ **by** (induction Cs) auto

lemma $subst_cls_lists_nth[simp]$:

$length Cs = length \sigma s \implies i < length Cs \implies (Cs \cdot cl \sigma s) ! i = (Cs ! i) \cdot (\sigma s ! i)$
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 $Neg_Melem_subst_atm_subst_cls[simp]$: $Neg A \in \# C \implies Neg (A \cdot a \sigma) \in \# C \cdot \sigma$
by (metis Melem_subst_cls eql_neg_lit_eql_atm)

lemma $Pos_Melem_subst_atm_subst_cls[simp]$: $Pos A \in \# C \implies Pos (A \cdot a \sigma) \in \# C \cdot \sigma$
by (metis Melem_subst_cls eql_pos_lit_eql_atm)

lemma $in_atms_of_subst[simp]$: $B \in atms_of C \implies B \cdot a \sigma \in atms_of (C \cdot \sigma)$
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 $is_renamingD$: $is_renaming \sigma \implies (\forall A1 A2. A1 \cdot a \sigma = A2 \cdot a \sigma \longleftrightarrow A1 = A2)$
by (metis is_renaming_def subst_atm_comp_subst subst_atm_id_subst)

lemma $inv_renaming_cancel_r[simp]$: $is_renaming r \implies r \odot inv_renaming r = id_subst$
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 ρs ==> is_renaming_list ρs ==> AAs · aml ρs · aml map inv_renaming ρs = 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 ρ ==> C · ρ · inv_renaming ρ = 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 ρ ==> CC · cs ρ · cs inv_renaming ρ = 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 ρ ==> Cs · cl ρ · cl inv_renaming ρ = 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 ρs ==> is_renaming_list ρs ==> Cs · cl ρs · cl map inv_renaming ρs = 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  $\varrho \implies CC \cdot cm \varrho \cdot cm inv_renaming \varrho = CC$ 
```

```
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 ⊆ set_mset D  $\implies$  set_mset (C · σ) ⊆ set_mset (D · σ)
  by force
```

```
lemma subst_cls_mono_mset: C ⊆# D  $\implies$  C · σ ⊆# D · σ
```

```
unfolding subst_clss_def by (metis mset_subset_eq_exists_conv subst_cls_union)
```

```
lemma subst_subset_mono: D ⊂# C  $\implies$  D · σ ⊂# C · σ
  unfolding subst_cls_def by (simp add: image_mset_subset_mono)
```

7.3.16 Size after Substitution

```
lemma size_subst[simp]: size (D · σ) = size D
  unfolding subst_cls_def by auto
```

```
lemma subst_atm_list_length[simp]: length (As · al σ) = length As
  unfolding subst_atm_list_def by auto
```

```
lemma length_subst_atm_mset_list[simp]: length (AAs · aml η) = length AAs
  unfolding subst_atm_mset_list_def by auto
```

```
lemma subst_atm_mset_lists_length[simp]: length (AAs ..aml σs) = min (length AAs) (length σs)
  unfolding subst_atm_mset_lists_def by auto
```

```
lemma subst_cls_list_length[simp]: length (Cs · cl σ) = length Cs
  unfolding subst_cls_list_def by auto
```

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

```
lemma subst_cls_lists_length[simp]: length (Cs ..cl σs) = 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  $\forall \sigma s. \text{length } \sigma s = \text{length } Cs \longrightarrow (\exists \tau. Cs ..cl \sigma s = Cs \cdot cl \tau)$ 
proof clarify
  fix  $\sigma s :: 's list$ 
  assume a:  $\text{length } \sigma s = \text{length } Cs$ 
  then obtain  $\tau$  where  $\forall i < \text{length } Cs. \forall S. S \subseteq# Cs ! i \longrightarrow S \cdot \sigma s ! i = S \cdot \tau$ 
    using assms unfolding var_disjoint_def by blast
  then have  $\forall i < \text{length } Cs. (Cs ! i) \cdot \sigma s ! i = (Cs ! i) \cdot \tau$ 
    by auto
  then have  $Cs ..cl \sigma s = Cs \cdot cl \tau$ 
    using a by (auto intro: nth_equalityI)
  then show  $\exists \tau. Cs ..cl \sigma s = Cs \cdot cl \tau$ 
    by auto
qed
```

7.3.18 Ground Expressions and Substitutions

```
lemma ex_ground_subst:  $\exists \sigma. \text{is\_ground\_subst } \sigma$ 
  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)`
`unfoldings 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`
`unfoldings 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`
`unfoldings 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`
`unfoldings 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`
`unfoldings 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)`

Grounding simplifications lemma `grounding_of_clss_empty[simp]: grounding_of_clss {} = {}`
`by (simp add: grounding_of_clss_def)`

lemma `grounding_of_clss_singleton[simp]: grounding_of_clss {C} = grounding_of_cls C`
`by (simp add: grounding_of_clss_def)`

lemma `grounding_of_clss_insert:`
`grounding_of_clss (insert C N) = grounding_of_cls C ∪ grounding_of_clss N`
`by (simp add: grounding_of_clss_def)`

lemma `grounding_of_clss_union:`
`grounding_of_clss (A ∪ B) = grounding_of_clss A ∪ grounding_of_clss B`
`by (simp add: grounding_of_clss_def)`

Grounding monotonicity lemma `is_ground_cls_mono: C ⊆# D ==> is_ground_cls D ==> is_ground_cls C`
`unfoldings 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`
`unfoldings is_ground_clss_def by blast`

lemma `grounding_of_clss_mono: CC ⊆ DD ==> grounding_of_cls CC ⊆ grounding_of_cls DD`
`using grounding_of_cls_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 (τ ⊕ σ)`
`unfoldings is_ground_subst_def is_ground_atm_def by auto`

lemma `ground_subst_ground_atm[simp]: is_ground_subst σ ==> is_ground_atm (A · a σ)`
`by (simp add: is_ground_subst_def)`

lemma `ground_subst_ground_lit[simp]: is_ground_subst σ ==> is_ground_lit (L · l σ)`
`unfoldings 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 · σ)`
`unfoldings is_ground_cls_def by auto`

```

lemma ground_subst_ground_clss[simp]: is_ground_subst σ ==> is_ground_clss (CC · cs σ)
  unfolding is_ground_clss_def subst_clss_def by auto

lemma ground_subst_ground_cls_list[simp]: is_ground_subst σ ==> is_ground_cls_list (Cs · cl σ)
  unfolding is_ground_cls_list_def subst_cls_list_def by auto

lemma ground_subst_ground_cls_lists[simp]:
  ∀σ ∈ set σs. is_ground_subst σ ==> is_ground_cls_list (Cs .. cl σs)
  unfolding is_ground_cls_list_def subst_cls_lists_def by (auto simp: set_zip)

lemma subst_cls_eq_grounding_of_cls_subset_eq:
  assumes D · σ = C
  shows grounding_of_cls C ⊆ grounding_of_cls D
proof
  fix Cσ'
  assume Cσ' ∈ grounding_of_cls C
  then obtain σ' where
    Cσ': C · σ' = Cσ' is_ground_subst σ'
    unfolding grounding_of_cls_def by auto
  then have C · σ' = D · σ · σ' ∧ is_ground_subst (σ ⊕ σ')
    using assms by auto
  then show Cσ' ∈ grounding_of_cls D
    unfolding grounding_of_cls_def using Cσ'(1) by force
qed

```

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

```

lemma is_ground_subst_atms[simp]: is_ground_atms AA ==> AA · as σ = 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 ==> AA · am σ = 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 ==> As · al σ = 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 ==> i < length As ==> As ! i · a σ = As ! i
  unfolding is_ground_atm_list_def by auto

```

```

lemma is_ground_subst_lit[simp]: is_ground_lit L ==> L · l σ = L
  unfolding is_ground_lit_def subst_lit_def by (cases L) simp_all

```

```

lemma is_ground_subst_cls[simp]: is_ground_cls C ==> C · σ = C
  unfolding is_ground_cls_def subst_cls_def by simp

```

```

lemma is_ground_subst_clss[simp]: is_ground_clss CC ==> CC · cs σ = 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 .. 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 ↔ (forall σ. L = L · l σ)
  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 ↔ (forall σ. C = C · σ)
  by (metis ex_ground_subst ground_subst_ground_cls is_ground_subst_cls)

```

Grounding of substitutions

```

lemma grounding_of_subst_cls_subset: grounding_of_cls (C · μ) ⊆ grounding_of_cls C
proof (rule subsetI)
  fix D
  assume D ∈ grounding_of_cls (C · μ)
  then obtain γ where D_def: D = C · μ · γ and gr_γ: is_ground_subst γ
    unfolding grounding_of_cls_def mem_Collect_eq by auto

  show D ∈ grounding_of_cls C
    unfolding grounding_of_cls_def mem_Collect_eq D_def
    using is_ground_comp_subst[OF gr_γ, of μ]
    by force
qed

```

lemma grounding_of_subst_clss_subset: grounding_of_clss (CC · cs μ) ⊆ grounding_of_clss CC

```

using grounding_of_subst_cls_subset
by (auto simp: grounding_of_clss_def subst_clss_def)

```

lemma grounding_of_subst_cls_renaming_ident[simp]:

```

assumes is_renaming ρ
shows grounding_of_cls (C · ρ) = grounding_of_cls C
by (metis (no_types, lifting) assms subset_antisym subst_cls_comp_subst
    subst_cls_eq_grounding_of_cls_subset_eq subst_cls_id_subst is_renaming_def)

```

lemma grounding_of_subst_clss_renaming_ident[simp]:

```

assumes is_renaming ρ
shows grounding_of_clss (CC · cs ρ) = grounding_of_clss CC
by (metis assms dual_order.eq_iff grounding_of_subst_clss_subset
    is_renaming_inv_renaming_cancel_clss)

```

Members of ground expressions are ground

lemma is_ground_cls_as_atms: is_ground_cls C ↔ (A ∈ 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 ∈# C ⇒ is_ground_cls C ⇒ is_ground_lit L

```

by (simp add: is_ground_cls_def)

```

lemma is_ground_cls_imp_is_ground_atm: A ∈ atms_of C ⇒ is_ground_cls C ⇒ 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 ⇒ is_ground_atms (atms_of C)

```

by (simp add: is_ground_cls_imp_is_ground_atm is_ground_atms_def)

```

lemma grounding_ground: C ∈ grounding_of_clss M ⇒ is_ground_cls C

```

unfolding grounding_of_clss_def grounding_of_cls_def by auto

```

lemma is_ground_cls_if_in_grounding_of_cls: C' ∈ grounding_of_cls C ⇒ is_ground_cls C'

```

using grounding_ground grounding_of_clss_singleton by blast

```

lemma in_subset_eq_grounding_of_clss_is_ground_cls[simp]:

```

C ∈ CC ⇒ CC ⊆ grounding_of_clss DD ⇒ 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 is_ground_cls_add_mset[simp]:

```

is_ground_cls (add_mset L C) ↔ is_ground_lit L ∧ is_ground_cls C
by (auto simp: is_ground_cls_def)

```

lemma grounding_of_cls_ground: is_ground_cls C ⇒ 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)

lemma union_grounding_of_cls_ground: is_ground_clss (UN (grounding_of_cls ` N))
  by (simp add: grounding_ground grounding_of_clss_def is_ground_clss_def)

lemma is_ground_clss_grounding_of_clss[simp]: is_ground_clss (grounding_of_clss N)
  using grounding_of_clss_def union_grounding_of_cls_ground by metis

```

Grounding idempotence lemma grounding_of_grounding_of_cls: $E \in \text{grounding_of_cls } D \implies D \in \text{grounding_of_cls } C \implies E = D$
 using grounding_of_cls_def by auto

```

lemma image_grounding_of_cls_grounding_of_cls:
  grounding_of_cls ` grounding_of_cls C = (\lambda x. {x}) ` grounding_of_cls C
proof (rule image_cong)
  show \(\lambda x. x \in \text{grounding\_of\_cls } C \implies \text{grounding\_of\_cls } x = {x}\)
    using grounding_of_cls_ground is_ground_cls_if_in_grounding_of_cls by blast
qed simp

```

```

lemma grounding_of_clss_grounding_of_clss[simp]:
  grounding_of_clss (grounding_of_clss N) = grounding_of_clss N
  unfolding grounding_of_clss_def UN_UN_flatten
  unfolding image_grounding_of_cls_grounding_of_cls
  by simp

```

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 \(\implies\) 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 \(\sigma\) AA \(\implies\) A \(\in\) AA \(\implies\) B \(\in\) AA \(\implies\) A \(\cdot a \sigma\) = B \(\cdot a \sigma\)
  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 \(\sigma\) AA \(\longleftrightarrow\) (\(\forall A \in AA\). \(\forall B \in AA\). A \(\cdot a \sigma\) = B \(\cdot a \sigma\))
  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 \(\sigma\) AAA AA \(\in\) AAA A \(\in\) AA B \(\in\) AA
  shows A \(\cdot a \sigma\) = B \(\cdot a \sigma\)
  by (metis assms is_unifiers_def is_unifier_subst_atm_eqI)

theorem is_unifiers_comp:
  is_unifiers \(\sigma\) (set_mset ` set (map2 add_mset As Bs) \(\cdot\) ass \(\eta\)) \(\longleftrightarrow\)
  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 \(\sigma\) AAA \(\implies\) is_unifiers \(\sigma\) AAA

```

```

using is_mgu_def by blast

lemma is_mgu_is_most_general: is_mgu σ AAA ⇒ is_unifiers τ AAA ⇒ ∃γ. τ = σ ⊕ γ
  using is_mgu_def by blast

lemma is_unifiers_is_unifier: is_unifiers σ AAA ⇒ AA ∈ AAA ⇒ is_unifier σ AA
  using is_unifiers_def by simp

lemma is_imgu_is_mgu[intro]: is_imgu σ AAA ⇒ is_mgu σ AAA
  by (auto simp: is_imgu_def is_mgu_def)

lemma is_imgu_comp_idempotent[simp]: is_imgu σ AAA ⇒ σ ⊕ σ = σ
  by (simp add: is_imgu_def)

lemma is_imgu_subst_atm_idempotent[simp]: is_imgu σ AAA ⇒ A · a σ · a σ = A · a σ
  using is_imgu_comp_idempotent[of σ] subst_atm_comp_subst[of A σ σ] by simp

lemma is_imgu_subst_atms_idempotent[simp]: is_imgu σ AAA ⇒ AA · as σ · as σ = AA · as σ
  using is_imgu_comp_idempotent[of σ] subst_atms_comp_subst[of AA σ σ] by simp

lemma is_imgu_subst_lit_idempotent[simp]: is_imgu σ AAA ⇒ L · l σ · l σ = L · l σ
  using is_imgu_comp_idempotent[of σ] subst_lit_comp_subst[of L σ σ] by simp

lemma is_imgu_subst_cls_idempotent[simp]: is_imgu σ AAA ⇒ C · σ · σ = C · σ
  using is_imgu_comp_idempotent[of σ] subst_cls_comp_subst[of C σ σ] by simp

lemma is_imgu_subst_clss_idempotent[simp]: is_imgu σ AAA ⇒ CC · cs σ · cs σ = CC · cs σ
  using is_imgu_comp_idempotent[of σ] subst_clsscomp_subst[of CC σ σ] by simp

```

7.3.22 Generalization and Subsumption

```

lemma variants_sym: variants D D' ↔ variants D' D
  unfolding variants_def by auto

lemma variants_iff_subsumes: variants C D ↔ subsumes C D ∧ subsumes D C
proof
  assume variants C D
  then show subsumes C D ∧ subsumes D C
    unfolding variants_def generalizes_def subsumes_def
    by (metis subset_mset.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_def
    by (metis leD mset_subset_size_size_mset_mono size_subst
        subset_mset.not_eq_order_implies_strict)
qed

lemma strict_subset_subst_strictly_subsumes: C · η ⊂# D ⇒ strictly_subsumes C D
  by (metis leD mset_subset_size_size_mset_mono size_subst strictly_subsumes_def
      subset_mset.dual_order.strict_implies_order_substitution_ops.subsumes_def)

lemma generalizes_lit_refl[simp]: generalizes_lit L L
  unfolding generalizes_lit_def by (rule exI[of _ id_subst]) simp

lemma generalizes_lit_trans:
  generalizes_lit L1 L2 ⇒ generalizes_lit L2 L3 ⇒ generalizes_lit L1 L3
  unfolding generalizes_lit_def using subst_lit_comp_subst by blast

lemma generalizes_refl[simp]: generalizes C C
  unfolding generalizes_def by (rule exI[of _ id_subst]) simp

```

```

lemma generalizes_trans: generalizes C D ==> generalizes D E ==> generalizes C E
  unfolding generalizes_def using subst_cls_comp_subst by blast

lemma subsumes_refl: subsumes C C
  unfolding subsumes_def by (rule exI[of _ id_subst]) auto

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

lemma strictly_generalizes_irrefl: ¬ strictly_generalizes C C
  unfolding strictly_generalizes_def by blast

lemma strictly_generalizes_antisym: strictly_generalizes C D ==> ¬ strictly_generalizes D C
  unfolding strictly_generalizes_def by blast

lemma strictly_generalizes_trans:
  strictly_generalizes C D ==> strictly_generalizes D E ==> strictly_generalizes C E
  unfolding strictly_generalizes_def using generalizes_trans by blast

lemma strictly_subsumes_irrefl: ¬ strictly_subsumes C C
  unfolding strictly_subsumes_def by blast

lemma strictly_subsumes_antisym: strictly_subsumes C D ==> ¬ strictly_subsumes D C
  unfolding strictly_subsumes_def by blast

lemma strictly_subsumes_trans:
  strictly_subsumes C D ==> strictly_subsumes D E ==> strictly_subsumes C E
  unfolding strictly_subsumes_def using subsumes_trans by blast

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

lemma strictly_generalizes_neq: strictly_generalizes D' D ==> D' ≠ D · σ
  unfolding strictly_generalizes_def generalizes_def by blast

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

lemma variants_imp_exists_substitution: variants D D' ==> ∃σ. D · σ = D'
  unfolding variants_iff_subsumes subsumes_def
  by (meson strictly_subsumes_def subset_mset_def strict_subset_subst_strictly_subsumes subsumes_def)

lemma strictly_subsumes_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 · σ = D' ∧ D' · σ' = D
    using variants_imp_exists_substitution variants_sym by metis

  from assms obtain σ'' where
    E · σ'' ⊆# D
    unfolding strictly_subsumes_def subsumes_def by auto
  then have E · σ'' · σ ⊆# D · σ
    using subst_cls_mono_mset by blast
  then have E · (σ'' ⊕ σ) ⊆# D'
    using σ_σ'_p by auto
  moreover from assms have n: #σ. D · σ ⊆# E
    unfolding strictly_subsumes_def subsumes_def by auto
  have #σ. D' · σ ⊆# E
  proof
    assume ∃σ''. D' · σ'' ⊆# E

```

```

then obtain  $\sigma'''$  where
   $D' \cdot \sigma''' \subseteq E$ 
  by auto
then have  $D \cdot (\sigma \odot \sigma'') \subseteq E$ 
  using  $\sigma \cdot \sigma' \cdot p$  by auto
then show False
  using n by metis
qed
ultimately show ?thesis
  unfolding strictly_subsumes_def subsumes_def by metis
qed

lemma neg_strictly_subsumes_variants:
assumes  $\neg \text{strictly\_subsumes } E D \text{ and } \text{variants } D D'$ 
shows  $\neg \text{strictly\_subsumes } E D'$ 
using assms strictly_subsumes_variants variants_sym by auto

end

locale substitution_renamings = substitution subst_atm id_subst comp_subst
for
  subst_atm :: ' $a \Rightarrow 's \Rightarrow 'a$ ' and
  id_subst :: ' $'s \Rightarrow 'a$ ' and
  comp_subst :: ' $'s \Rightarrow 's \Rightarrow 's +$ ' and
fixes
  renamings_apart :: ' $'a \text{ clause list} \Rightarrow 's \text{ list}$ ' and
  atm_of_atms :: ' $'a \text{ list} \Rightarrow 'a$ ' and
assumes
  renamings_apart_length:  $\text{length } (\text{renamings\_apart } Cs) = \text{length } Cs$  and
  renamings_apart_renaming:  $\varrho \in \text{set } (\text{renamings\_apart } Cs) \implies \text{is\_renaming } \varrho$  and
  renamings_apart_var_disjoint:  $\text{var\_disjoint } (Cs \cdot \text{cl } (\text{renamings\_apart } Cs))$  and
  atm_of_atms_subst:
     $\bigwedge As Bs. \text{atm\_of\_atms } As \cdot a \sigma = \text{atm\_of\_atms } Bs \longleftrightarrow \text{map } (\lambda A. A \cdot a \sigma) As = Bs$ 
begin

```

7.3.23 Generalization and Subsumption

```

lemma wf_strictly_generalizes: wfP strictly_generalizes
proof -
{
  assume  $\exists C_{\text{at}}. \forall i. \text{strictly\_generalizes } (C_{\text{at}} (\text{Suc } i)) (C_{\text{at}} i)$ 
  then obtain C_at :: nat  $\Rightarrow 'a \text{ clause}$  where
    sg_C:  $\bigwedge i. \text{strictly\_generalizes } (C_{\text{at}} (\text{Suc } i)) (C_{\text{at}} i)$ 
    by blast

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

  have sz_C:  $\text{size } (C_{\text{at}} i) = n$  for i
  proof (induct i)
    case (Suc i)
    then show ?case
      using sg_C[of i] unfolding strictly_generalizes_def generalizes_def subst_cls_def
      by (metis size_image_mset)
  qed (simp add: n_def)

  obtain sigma_at :: nat  $\Rightarrow 's$  where
    C_sigma:  $\bigwedge i. \text{image\_mset } (\lambda L. L \cdot \sigma_{\text{at}} i) (C_{\text{at}} (\text{Suc } i)) = C_{\text{at}} i$ 
    using sg_C[unfolded strictly_generalizes_def generalizes_def subst_cls_def] by metis

  define Ls_at :: nat  $\Rightarrow 'a \text{ literal list}$  where
    Ls_at = rec_nat (SOME Ls. mset Ls = C_at 0)
    ( $\lambda i Lsi. \text{SOME } Ls. \text{mset } Ls = C_{\text{at}} (\text{Suc } i) \wedge \text{map } (\lambda L. L \cdot \sigma_{\text{at}} i) Ls = Lsi$ )

```

```

have
  Ls_at_0: Ls_at 0 = (SOME Ls. mset Ls = C_at 0) and
  Ls_at_Suc:  $\bigwedge i. Ls\_at(Suc i) =$ 
     $(SOME Ls. mset Ls = C\_at(Suc i) \wedge map(\lambda L. L \cdot l \sigma\_at i) Ls = Ls\_at i)$ 
  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)  $\wedge$  map(\lambda L. L \cdot l \sigma_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:  $\bigwedge i. length(Ls\_at i) = n$ 
  by (metis mset_Ls mset_Lt_at_0 not0_implies_Suc size_mset sz_C)

have is_pos_Ls:  $\bigwedge i j. j < n \implies is\_pos(Ls\_at(Suc i) ! j) \longleftrightarrow is\_pos(Ls\_at i ! j)$ 
  using Ls_σ len_Ls by (metis literal.map_disc_iff nth_map subst_lit_def)

have Ls_τ_strict_lit:  $\bigwedge i \tau. map(\lambda L. L \cdot l \tau)(Ls\_at i) \neq Ls\_at(Suc i)$ 
  by (metis C_σ mset_Ls Ls_σ mset_map sg_C generalizes_def strictly_generalizes_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  $\Rightarrow$  'a where
   $\bigwedge i. tm\_at i = atm\_of\_atms(map atm\_of(Ls\_at i))$ 

have  $\bigwedge i. generalizes\_atm(tm\_at(Suc i))(tm\_at i)$ 
  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  $\bigwedge i. \neg generalizes\_atm(tm\_at i)(tm\_at(Suc i))$ 
  unfolding tm_at_def generalizes_atm_def atm_of_atms_subst by (simp add: Ls_τ_strict_tm)
ultimately have  $\bigwedge i. strictly\_generalizes\_atm(tm\_at(Suc i))(tm\_at i)$ 
  unfolding strictly_generalizes_atm_def by blast
then have False
  using wf_strictly_generalizes_atm[unfolded wf_def wf_iff_no_infinite_down_chain] by blast
}

```

```

then show wfP (strictly_generalizes :: 'a clause ⇒ _ ⇒ _)
  unfolding wfp_def by (blast intro: wf_iff_no_infinite_down_chain[THEN iffD2])
qed

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
  λn. 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 o c) i ≥ (size o 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 (c (Suc l')) (c l')
  using ps unfolding strictly_generalizes_def generalizes_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 (c (Suc i + l)) (c (i + l))
  unfolding strictly_generalizes_def generalizes_def by auto
then have ∃ f. ∀ i. strictly_generalizes (f (Suc i)) (f i)
  by (rule exI[of _ λx. c (x + l)])
then show False
  using wf_strictly_generalizes
    wf_iff_no_infinite_down_chain[of {(x, y). strictly_generalizes x y}]
  unfolding wfp_def by auto
qed

lemma wf_strictly_subsumes: wfP strictly_subsumes
  using strictly_subsumes_has_minimum by (metis equals0D wfp_eq_minimal)

end

```

7.4 Most General Unifiers

```

locale mgu = substitution_renamings subst_atm id_subst comp_subst renamings_apart atm_of_atms
for
  subst_atm :: 'a ⇒ 's ⇒ 'a and
  id_subst :: 's and
  comp_subst :: 's ⇒ 's ⇒ 's and
  renamings_apart :: 'a literal multiset list ⇒ 's list and
  atm_of_atms :: 'a list ⇒ 'a+
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 ∈# AAs ! i
    shows A · a σ = As ! i · a σ
  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

```

7.5 Idempotent Most General Unifiers

```

locale imgu = mgu subst_atm id_subst comp_subst renamings_apart atm_of_atms mgu
for
  subst_atm :: 'a ⇒ 's ⇒ 'a and
  id_subst :: 's and
  comp_subst :: 's ⇒ 's ⇒ 's and
  renamings_apart :: 'a literal multiset list ⇒ 's list and
  atm_of_atms :: 'a list ⇒ 'a and
  mgu :: 'a set set ⇒ 's option +
assumes
  mgu_is_imgu: finite AAA ==> (∀ AA ∈ AAA. finite AA) ==> mgu AAA = Some σ ==> is_imgu σ AAA
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_prem_of: 'a clause multiset) (main_prem_of: 'a clause) (concl_of: 'a clause)

```

```

abbreviation prems_of :: "'a inference ⇒ 'a clause multiset where
  prems_of γ ≡ side_prem_of γ + {#main_prem_of γ#}

abbreviation concls_of :: "'a inference set ⇒ 'a clause set where
  concls_of Γ ≡ concl_of 'Γ

definition infer_from :: "'a clause set ⇒ 'a inference ⇒ bool where
  infer_from CC γ ↔ set_mset (prems_of γ) ⊆ CC

locale inference_system =
  fixes Γ :: "'a inference set
begin

definition inferences_from :: "'a clause set ⇒ 'a inference set where
  inferences_from CC = {γ. γ ∈ Γ ∧ infer_from CC γ}

definition inferences_between :: "'a clause set ⇒ 'a clause ⇒ 'a inference set where
  inferences_between CC C = {γ. γ ∈ Γ ∧ infer_from (CC ∪ {C}) γ ∧ C ∈# prems_of γ}

lemma inferences_from_mono: CC ⊆ DD ⟹ inferences_from CC ⊆ inferences_from DD
  unfolding inferences_from_def infer_from_def by fast

definition saturated :: "'a clause set ⇒ bool where
  saturated N ↔ concls_of (inferences_from N) ⊆ 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  $I \models s N \cup \text{concls\_of}(\text{inferences\_from } N)$ 
  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  $\Gamma$  for  $\Gamma :: ('a :: \text{wellorder}) \text{ inference set} +$ 
  assumes  $\Gamma_{\text{reductive}}: \gamma \in \Gamma \implies \text{concl\_of } \gamma < \text{main\_prem\_of } \gamma$ 

```

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  $\Gamma$  for  $\Gamma :: ('a :: \text{wellorder}) \text{ inference set} +$ 
  fixes  $I_{\text{of}} :: 'a \text{ clause set} \Rightarrow 'a \text{ interp}$ 
  assumes  $\Gamma_{\text{counterex\_reducing}}:$ 
     $\{\#\} \notin N \implies D \in N \implies \neg I_{\text{of}} N \models D \implies (\bigwedge C. C \in N \implies \neg I_{\text{of}} N \models C \implies D \leq C) \implies$ 
     $\exists CC E. \text{set\_mset } CC \subseteq N \wedge I_{\text{of}} N \models_m CC \wedge \text{Infer } CC D E \in \Gamma \wedge \neg I_{\text{of}} N \models E \wedge E < D$ 
begin

```

```

lemma ex_min_counterex:
  fixes  $N :: ('a :: \text{wellorder}) \text{ clause set}$ 
  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  $c_{\text{in}}: C \in \{C \in N. \neg I \models C\}$ 
    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: {#} ∉ N
shows  $I_{\text{of}} N \models s N$ 
proof –
{
  assume  $\neg I_{\text{of}} N \models s N$ 
  then obtain  $D$  where
    d_in_n: D ∈ N and
    d_cex: ¬ I_of N ⊨ D and
    d_min: ∀C. C ∈ N ⇒ C < D ⇒ I_of N ⊨ C
    by (meson ex_min_counterex)
  then obtain  $CC E$  where
    cc_subs_n: set_mset CC ⊆ N and
    inf_e: Infer CC D E ∈ Γ and
    e_cex: ¬ I_of N ⊨ E and
    e_lt_d: E < D
    using  $\Gamma_{\text{counterex\_reducing}}[\text{OF } ec_{\text{ni}}_n \text{ not\_less by metis}]$ 
    from cc_subs_n inf_e have E ∈ 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 ==> ¬ satisfiable N ==> {#} ∈ 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 ⇒ 'a clause set for CC :: 'a clause set where
  base: C ∈ CC ==> C ∈ saturate CC
  | step: Infer CC' D E ∈ Γ ==> (A C'. C' ∈# CC' ==> C' ∈ saturate CC) ==> D ∈ saturate CC ==>
    E ∈ saturate CC

```

```

lemma saturate_mono: C ∈ saturate CC ==> CC ⊆ DD ==> C ∈ 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 ∈ saturate CC ==> ∃ DD. DD ⊆ CC ∧ finite DD ∧ C ∈ saturate DD
proof (induct rule: saturate.induct)

```

```

  case (base C)
  then have {C} ⊆ CC and finite {C} and C ∈ saturate {C}
    by (auto intro: saturate.intros)
  then show ?case
    by blast

```

next

```

  case (step CC' D E)
  obtain DD_of where
    A C. C ∈# CC' ==> DD_of C ⊆ CC ∧ finite (DD_of C) ∧ C ∈ saturate (DD_of C)
    using step(3) by metis

```

```

  then have
    (U C ∈ set_mset CC'. DD_of C) ⊆ CC
    finite (U C ∈ set_mset CC'. DD_of C) ∧ set_mset CC' ⊆ saturate (U C ∈ set_mset CC'. DD_of C)
    by (auto intro: saturate_mono)

```

then obtain DD where

```

  d_sub: DD ⊆ CC and d_fin: finite DD and in_sat_d: set_mset CC' ⊆ saturate DD
  by blast

```

obtain EE where

```

  e_sub: EE ⊆ CC and e_fin: finite EE and in_sat_e: D ∈ saturate EE
  using step(5) by blast

```

have DD ∪ EE ⊆ CC

```

  using d_sub e_sub step(1) by fast

```

moreover have finite (DD ∪ EE)

```

  using d_fin e_fin by fast

```

moreover have E ∈ saturate (DD ∪ EE)

```

  using in_sat_d in_sat_e 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:  $C \in \text{saturate } CC \implies I \models_s CC \implies I \models C$ 
  by (induct rule: saturate.induct) (auto simp: true_cls_mset_def true_clss_def  $\Gamma$ _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  $CC \implies \text{satisfiable } (\text{saturate } CC)$ 
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  $N :: ('a :: \text{wellorder}) \text{ clause set}$ 
  shows  $\neg \text{satisfiable } N \longleftrightarrow (\exists DD \subseteq N. \text{finite } DD \wedge \neg \text{satisfiable } DD)$ 
proof
  assume  $\neg \text{satisfiable } N$ 
  then have  $\{\#\} \in \text{saturate } N$ 
    using saturated_complete saturated_saturate.saturate.base unfolding true_clss_def by meson
  then have  $\exists DD \subseteq N. \text{finite } DD \wedge \{\#\} \in \text{saturate } DD$ 
    using saturate_finite by fastforce
  then show  $\exists DD \subseteq N. \text{finite } DD \wedge \neg \text{satisfiable } DD$ 
    using saturate_sound by auto
next
  assume  $\exists DD \subseteq N. \text{finite } DD \wedge \neg \text{satisfiable } DD$ 
  then show  $\neg \text{satisfiable } N$ 
    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:  $L \in\# S C \implies \text{is\_neg } L$ 

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  $\varepsilon_C$  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.  $C \in N \wedge C \neq \{\#\} \wedge \text{Max\_mset } C = \text{Pos } A \wedge \neg (\bigcup D \in \{D. D < C\}. \text{production } D) \models C \wedge S C = \{\#\}}$ 
  by auto
termination by (rule termination[OF wf, simplified])

declare production.simps [simp del]

definition interp :: 'a clause  $\Rightarrow$  'a interp where
  interp C = ( $\bigcup D \in \{D. D < C\}. \text{production } D$ )

lemma production_unfold:
  production C = {A.  $C \in N \wedge C \neq \{\#\} \wedge \text{Max\_mset } C = \text{Pos } A \wedge \neg \text{interp } C \models C \wedge S C = \{\#\}}$ 
  unfolding interp_def by (rule production.simps)

abbreviation productive :: 'a clause  $\Rightarrow$  bool where
  productive C  $\equiv$  production C  $\neq \{\}$ 

abbreviation produces :: 'a clause  $\Rightarrow$  'a  $\Rightarrow$  bool where
  produces C A  $\equiv$  production C = {A}

lemma producesD: produces C A  $\implies C \in N \wedge C \neq \{\#\} \wedge \text{Pos } A = \text{Max\_mset } C \wedge \neg \text{interp } C \models C \wedge S C = \{\#\}$ 
  unfolding production_unfold by auto

definition Interp :: 'a clause  $\Rightarrow$  'a interp where
  Interp C = interp C  $\cup$  production C

lemma interp_subseteq_Interp[simp]: interp C  $\subseteq$  Interp C
  by (simp add: Interp_def)

lemma Interp_as_UNION: Interp C = ( $\bigcup D \in \{D. D \leq C\}. \text{production } D$ )
  unfolding Interp_def interp_def less_eq_multiset_def by fast

lemma productive_not_empty: productive C  $\implies C \neq \{\#\}$ 
  unfolding production_unfold by simp

lemma productive_imp_produces_Max_literal: productive C  $\implies$  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  $\implies$  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_subseq_interp: C ≤ D ==> interp C ⊆ interp D
  unfolding interp_def by auto (metis order.strict_trans2)

lemma less_eq_imp_interp_subseq_Interp: C ≤ D ==> interp C ⊆ Interp D
  unfolding Interp_def using less_eq_imp_interp_subseq_interp by blast

lemma less_imp_production_subseq_interp: C < D ==> production C ⊆ interp D
  unfolding interp_def by fast

lemma less_eq_imp_production_subseq_Interp: C ≤ D ==> production C ⊆ Interp D
  unfolding Interp_def using less_imp_production_subseq_interp
  by (metis le_imp_less_or_eq le_supI1 sup_ge2)

lemma less_imp_Interp_subseq_interp: C < D ==> Interp C ⊆ interp D
  by (simp add: Interp_def less_eq_imp_interp_subseq_interp less_imp_production_subseq_interp)

lemma less_eq_imp_Interp_subseq_Interp: C ≤ D ==> Interp C ⊆ Interp D
  using Interp_def less_eq_imp_interp_subseq_Interp less_eq_imp_production_subseq_Interp by auto

lemma not_Interp_to_interp_imp_less: A ∉ Interp C ==> A ∈ interp D ==> C < D
  using less_eq_imp_interp_subseq_Interp not_less by blast

lemma not_interp_to_interp_imp_less: A ∉ interp C ==> A ∈ interp D ==> C < D
  using less_eq_imp_interp_subseq_interp not_less by blast

lemma not_Interp_to_Interp_imp_less: A ∉ Interp C ==> A ∈ Interp D ==> C < D
  using less_eq_imp_Interp_subseq_Interp not_less by blast

lemma not_interp_to_Interp_imp_le: A ∉ interp C ==> A ∈ Interp D ==> C ≤ D
  using less_eq_imp_Interp_subseq_interp not_less by blast

definition INTERP :: 'a interp where
  INTERP = (UNION C ∈ N. production C)

lemma interp_subseq_INTERP: interp C ⊆ INTERP
  unfolding interp_def INTERP_def by (auto simp: production_unfold)

lemma production_subseq_INTERP: production C ⊆ INTERP
  unfolding INTERP_def using production_unfold by blast

lemma Interp_subseq_INTERP: Interp C ⊆ INTERP
  by (simp add: Interp_def interp_subseq_INTERP production_subseq_INTERP)

lemma produces_imp_in_interp:

```

```

assumes a_in_c: Neg A ∈# C and d: produces D A
shows A ∈ 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 ∈# C ==> A ∉ Interp D ==> C ≤ D ==> ¬ produces D'' A
  using less_eq_imp_interp_subseq_Interp_produces_imp_in_interp by blast

lemma in_production_imp_produces: A ∈ production C ==> produces C A
  using productive_imp_produces_Max_atom by fastforce

lemma not_produces_imp_notin_production: ¬ produces C A ==> A ∉ production C
  using in_production_imp_produces by blast

lemma not_produces_imp_notin_interp: (¬ produces D A) ==> A ∉ 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 ≤ D and
  d_lt_d': D < D' and
  c_at_d: Interp D ⊨ C and
  subs: interp D' ⊆ (⋃ C ∈ CC. production C)
shows (⋃ C ∈ CC. production C) ⊨ C
proof (cases ∃ A. Pos A ∈# C ∧ A ∈ Interp D)
  case True
  then obtain A where a_in_c: Pos A ∈# C and a_at_d: A ∈ Interp D
    by blast
  from a_at_d have A ∈ interp D'
    using d_lt_d' less_eq_imp_interp_subseq_interp by blast
  then show ?thesis
    using subs a_in_c by (blast dest: contra_subsetD)
next
  case False
  then obtain A where a_in_c: Neg A ∈# C and A ∉ Interp D
    using c_at_d unfolding true_cls_def by blast
  then have ¬ D''. ¬ produces D'' A
    using c_le_d neg_notin_Interp_not_produce by simp
  then show ?thesis
    using a_in_c subs not_produces_imp_notin_production by auto
qed

lemma Interp_imp_interp: C ≤ D ==> D < D' ==> Interp D ⊨ C ==> interp D' ⊨ C
  using interp_def Interp_imp_general by simp

lemma Interp_imp_Interp: C ≤ D ==> D ≤ D' ==> Interp D ⊨ C ==> Interp D' ⊨ C
  using Interp_as_UNION interp_subseq_Interp_Interp_imp_general by (metis antisym_conv2)

lemma Interp_imp_INTERP: C ≤ D ==> Interp D ⊨ C ==> INTERP ⊨ C
  using INTERP_def interp_subseq_INTERP_Interp_imp_general[OF _ le_multiset_right_total] by simp

lemma interp_imp_general:
assumes
  c_le_d: C ≤ D and
  d_le_d': D ≤ D' and
  c_at_d: interp D ⊨ C and
  subs: interp D' ⊆ (⋃ C ∈ CC. production C)
shows (⋃ C ∈ CC. production C) ⊨ C
proof (cases ∃ A. Pos A ∈# C ∧ A ∈ interp D)
  case True
  then obtain A where a_in_c: Pos A ∈# C and a_at_d: A ∈ interp D
    by blast

```

```

from a_at_d have A ∈ interp D'
  using d_le_d' less_eq_imp_interp_subseq_interp by blast
then show ?thesis
  using subs a_in_c by (blast dest: contra_subsetD)
next
  case False
  then obtain A where a_in_c: Neg A ∈# C and A ∉ interp D
    using c_at_d unfolding true_cls_def by blast
  then have ⋀D''. ¬ produces D'' A
    using c_le_d by (auto dest: produces_imp_in_interp less_eq_imp_interp_subseq_interp)
  then show ?thesis
    using a_in_c subs not_produces_imp_notin_production by auto
qed

```

lemma interp_imp_interp: $C \leq D \implies D \leq D' \implies \text{interp } D \models C \implies \text{interp } D' \models C$
 using interp_def interp_imp_general by simp

lemma interp_imp_Interp: $C \leq D \implies D \leq D' \implies \text{interp } D \models C \implies \text{Interp } D' \models C$
 using Interp_as_UNION interp_subseq_Interp[of D'] interp_imp_general by simp

lemma interp_imp_INTERP: $C \leq D \implies \text{interp } D \models C \implies \text{INTERP } \models C$
 using INTERP_def interp_subseq_INTERP interp_imp_general_linear by metis

lemma productive_imp_not_interp: productive C $\implies \neg \text{interp } C \models C$
 unfolding production_unfold by simp

This corresponds to Lemma 3.3:

```

lemma productive_imp_Interp:
  assumes productive C
  shows Interp C ⊨ 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 ∈# C
    by (rule produces_imp_Pos_in_lits)
  moreover have A ∈ Interp C
    using a less_eq_imp_production_subseq_Interp by blast
  ultimately show ?thesis
    by fast
qed

```

lemma productive_imp_INTERP: productive C $\implies \text{INTERP } \models C$
 by (fast intro: productive_imp_Interp Interp_imp_INTERP)

This corresponds to Lemma 3.5:

```

lemma max_pos_imp_Interp:
  assumes C ∈ N and C ≠ {#} and Max_mset C = Pos A and S C = {#}
  shows Interp C ⊨ C
proof (cases productive C)
  case True
  then show ?thesis
    by (fast intro: productive_imp_Interp)
next
  case False
  then have interp C ⊨ 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:  $\text{Pos } A \in \# C$  and
max_atm:  $A = \text{Max}(\text{atms\_of } C)$  and
s_c_e:  $S C = \{\#\}$ 
shows  $\text{Interp } C \models C$ 
proof ( $\text{cases Neg } A \in \# C$ )
  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:  $D' \in N \wedge S D' = \{\#\} \vee \text{Max}(\text{atms\_of } D') < \text{Max}(\text{atms\_of } D)$  and
d'_at_d:  $\neg \text{Interp } D \models D'$  and
d_lt_c:  $D < C$  and
subs: interp C ⊆ (⋃ C ∈ CC. production C)
shows  $\neg (\bigcup C \in CC. \text{production } C) \models D'$ 
proof –
{
  assume cc_blw_d': ( $\bigcup C \in CC. \text{production } C$ )  $\models D'$ 
  have Interp D ⊆ ( $\bigcup C \in CC. \text{production } C$ )
    using less_imp_Interp_subseq_interp_d_lt_c subs by blast
  then obtain A where a_in_d':  $\text{Pos } A \in \# D'$  and a_blw_cc:  $A \in (\bigcup C \in CC. \text{production } C)$ 
    using cc_blw_d' d'_at_d false_to_true_imp_ex_pos by metis
  from a_in_d' have a_at_d:  $A \notin \text{Interp } D$ 
    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':  $\text{Max}(\text{atms\_of } C') = A$ 
    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' ≤ C'
    using d'_le_d order_trans by blast
  then have max_d':  $\text{Max}(\text{atms\_of } D') = A$ 
    using a_in_d' max_c' by (fast intro: pos_lit_in_atms_of_le_multiset_Max_in_imp_Max)

  {
    assume D' ∈ 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:
   $D' \leq D \implies D' \in N \wedge S D' = \{\#\} \vee \text{Max}(\text{atms\_of } D') < \text{Max}(\text{atms\_of } D) \implies \neg \text{Interp } D \models D' \implies$ 
   $D < C \implies \neg \text{interp } C \models D'$ 
  using interp_def not_Interp_imp_general by simp

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

lemma not_Interp_imp_not_INTERP:
   $D' \leq D \implies D' \in N \wedge S D' = \{\#\} \vee \text{Max}(\text{atms\_of } D') < \text{Max}(\text{atms\_of } D) \implies \neg \text{Interp } D \models D' \implies$ 
   $\neg \text{INTERP} \models D'$ 
  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  $\bigwedge D'. D' < D \implies \text{Interp } D' \models C$ 
  shows  $\text{interp } D \models C$ 
  oops

lemma
  assumes  $d: D = \{\#\}$  and  $n: N = \{D, C\}$  and  $c: C = \{\#Pos A\#}$ 
  shows  $D \in N$  and  $\bigwedge D'. D' < D \implies \text{Interp } D' \models C$  and  $\neg \text{interp } D \models C$ 
  using n unfolding d c 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 clause \Rightarrow 'a clause \Rightarrow 'a clause \Rightarrow bool" where

```

```
unord_resolve (C + replicate_mset (Suc n) (Pos A)) (add_mset (Neg A) D) (C + D)
```

```
lemma unord_resolve_sound: unord_resolve C D E ==> I ⊨ C ==> I ⊨ D ==> I ⊨ 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: {#} ∉ N and
c_in_n: C ∈ N and
c_cex: ¬ INTERP N ⊨ C and
c_min: ⋀ D. D ∈ N ==> ¬ INTERP N ⊨ D ==> C ≤ D
```

obtains D E where

```
D ∈ N
INTERP N ⊨ D
productive N D
unord_resolve D C E
¬ INTERP N ⊨ E
E < C
```

proof –

```
have c_ne: C ≠ {#}
  using c_in_n ec_ni_n by blast
have ∃ A. A ∈ atms_of C ∧ A = Max(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:  $A = \text{Max}(\text{atms\_of } D)$ 
  using d0 productive_imp_produces_Max_atom by auto
then have  $D' \neq \{\#\} \implies \text{Max}(\text{atms\_of } D') \leq A$ 
  using d'_le_d by (blast intro: less_eq_Max_atms_of)
moreover have  $D' \neq \{\#\} \implies \text{Max}(\text{atms\_of } D') \neq A$ 
  using a_ni_d' Max_in by (blast intro: atms_empty_iff_empty[THEN iffD1])
ultimately have max_d'_lt_a:  $D' \neq \{\#\} \implies \text{Max}(\text{atms\_of } D') < A$ 
  using dual_order.strict_iff_order by blast

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

have  $\bigwedge B. B \in \text{atms\_of } D' \implies B \leq A$ 
  using d0 d'_subs contra_subsetD lits_subseq_imp_atms_subseq produces_imp_atms_leq by metis
then have  $\bigwedge L. L \in \# D' \implies L < \text{Neg } A$ 
  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

Theorem 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  $\neg \text{INTERP } N \models D$  and  $\bigwedge C. C \in N \implies \neg \text{INTERP } N \models C \implies D \leq C$ 
  then obtain C E where
    c_in_n:  $C \in N$  and
    c_true:  $\text{INTERP } N \models C$  and
    res_e: unord_resolve C D E and
    e_cex:  $\neg \text{INTERP } N \models E$  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
     $\exists CC E. \text{set\_mset } CC \subseteq N \wedge \text{INTERP } N \models m CC \wedge \text{Infer } CC D E \in \text{unord\_Γ} \wedge \neg \text{INTERP } N \models E \wedge E < D$ 
    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 (clarify 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 ⇒ 'a literal multiset ⇒ bool where
  maximal_wrt A DA ↔ DA = {#} ∨ A = Max (atms_of DA)
```

```
definition strictly_maximal_wrt :: 'a ⇒ 'a literal multiset ⇒ bool where
  strictly_maximal_wrt A CA ↔ (∀ B ∈ atms_of CA. B < A)
```

```
inductive eligible :: 'a list ⇒ 'a clause ⇒ bool where
  eligible: (S DA = negs (mset As)) ∨ (S DA = {#} ∧ length As = 1 ∧ maximal_wrt (As ! 0) DA) ==>
    eligible As DA
```

```
lemma (S DA = negs (mset As) ∨ S DA = {#} ∧ length As = 1 ∧ maximal_wrt (As ! 0) DA) ↔
  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 ⇒ 'a clause ⇒ 'a multiset list ⇒ 'a list ⇒ 'a clause ⇒ bool
```

```
where
```

```
  ord_resolve:
    length CAs = n ==>
    length Cs = n ==>
    length AAs = n ==>
    length As = n ==>
    n ≠ 0 ==>
    (∀ i < n. CAs ! i = Cs ! i + poss (AAs ! i)) ==>
    (∀ i < n. AAs ! i ≠ {#}) ==>
    (∀ i < n. ∀ A ∈# AAs ! i. A = As ! i) ==>
    eligible As (D + negs (mset As)) ==>
    (∀ i < n. strictly_maximal_wrt (As ! i) (Cs ! i)) ==>
    (∀ i < n. S (CAs ! i) = {#}) ==>
    ord_resolve CAs (D + negs (mset As)) AAs As (sum # (mset Cs) + D)
```

```
lemma ord_resolve_sound:
```

```
assumes
```

```

res_e: ord_resolve CAs DA AAs As E and
cc_true: I ⊨ m mset CAs and
d_true: I ⊨ DA
shows I ⊨ 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 ∀ A ∈ set As. A ∈ I)
case True
then have ¬ I ⊨ negs (mset As)
unfolding true_cls_def by fastforce
then have I ⊨ 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 ∉ 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 (sum # (mset Cs)) and
  mc_max: ⋀ B. B ∈ atms_of (sum # (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: sum # (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 (sum # (mset Cs)). B < A_max
  using mc_max mc_lt_ma by fastforce
moreover have ¬ Neg A_max ∈# sum # (mset Cs)
  using ucas_lt_ma neg_lit_in_atms_of[of A_max sum # (mset Cs)] by auto
moreover have Neg A_max ∈# negs (mset As)
  using cm_in_cas cas_len ai_len A_max_def by auto
ultimately have sum # (mset Cs) < negs (mset As)
  unfolding 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: {#} ≠ N and
  d_in_n: DA ∈ N and
  d_cex: ¬ INTERP N ⊨ DA and
  d_min: ⋀ C. C ∈ N ⇒ ¬ INTERP N ⊨ C ⇒ DA ≤ C
obtains CAs AAs As E where
  set CAs ⊆ N
  INTERP N ⊨ m mset CAs
  ⋀ CA. CA ∈ set CAs ⇒ productive N CA
  ord_resolve CAs DA AAs As E
  ¬ INTERP N ⊨ E
  E < DA
proof –
  have d_ne: DA ≠ {#}
  using d_in_n ec_ni_n by blast
  have ∃ As. As ≠ [] ∧ negs (mset As) ≤# DA ∧ 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 ∈# 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) ⊆# DA
  by (simp add: As_def)
moreover have hd: 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 ≠ {#}
define As :: 'a list where
  As = list_of_mset {#atm_of L. L ∈# S DA#}
define D :: 'a clause where
  D = DA - negs {#atm_of L. L ∈# S DA#}

have As ≠ [] 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 ∈# S DA#} ⊆# DA
  using S_selects_subseteq by (auto simp: filter_neg_atm_of_S)
then have negs (mset As) ⊆# 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
   $\lambda A. Aj\_of A = \text{image\_mset atm\_of } \{\#L \in \# CA\_of A. L = Pos A\#}$ 

have pospos:  $\bigwedge LL A. \{\#Pos (\text{atm\_of } x). x \in \# \{ \#L \in \# LL. L = Pos A\#\} \# \} = \{\#L \in \# LL. L = Pos A\#\}$ 
  by (metis (mono_tags, lifting) image_filter_cong literal.sel(1) multiset.map_ident)
have ca_of_c_of_aj_of:  $\bigwedge A. CA\_of A = C\_of A + poss (Aj\_of A)$ 
  using pospos[of _ CA_of_] by (simp add: C_of_def Aj_of_def)

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:  $\bigwedge A. A \in set As \implies Aj\_of A \neq \{\#\}$ 
  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  $\in$  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_sub_n: set CAs  $\subseteq N$ 
  using productive_in_N by auto
have cs_true: INTERP N  $\vdash_m \text{mset } CAs$ 
  unfolding true_cls_mset_def using prod_c productive_imp_INTERP by auto

have  $\bigwedge A. A \in set As \implies \neg Neg A \in \# CA\_of A$ 
  using prod_c0 produces_imp_neg_notin_lits by auto
then have a_ni_c':  $\bigwedge A. A \in set As \implies A \notin \text{atms\_of } (C\_of A)$ 
  unfolding C_of_def using atm_imp_pos_or_neg_lit by force
have c'_le_c:  $\bigwedge A. C\_of A \leq CA\_of A$ 
  unfolding C_of_def by (auto intro: subset_eq_imp_le_multiset)
have a_max_c:  $\bigwedge A. A \in set As \implies A = \text{Max } (\text{atms\_of } (CA\_of A))$ 
  using prod_c0 productive_imp_produces_Max_atom[of N] by auto
then have  $\bigwedge A. A \in set As \implies C\_of A \neq \{\#\} \implies \text{Max } (\text{atms\_of } (C\_of A)) \leq A$ 
  using c'_le_c by (metis less_eq_Max_atms_of)
moreover have  $\bigwedge A. A \in set As \implies C\_of A \neq \{\#\} \implies \text{Max } (\text{atms\_of } (C\_of A)) \neq A$ 
  using a_ni_c' Max_in by (metis (no_types) atms_empty_iff_empty_finite_atms_of)
ultimately have max_c_lt_a:  $\bigwedge A. A \in set As \implies C\_of A \neq \{\#\} \implies \text{Max } (\text{atms\_of } (C\_of A)) < A$ 
  by (metis order.strict_iff_order)

have le_cs_as:  $\text{length } CAs = \text{length } As$ 
  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  $\forall i. i < \text{length } AAs \rightarrow (\forall A \in \# AAs ! i. A = As ! i)$ 
  using  $AAs\_def Aj\_of\_def$  by auto

have  $\bigwedge x B. \text{production } N (CA\_of x) = \{x\} \Rightarrow B \in \# CA\_of x \Rightarrow B \neq Pos x \Rightarrow atm\_of B < x$ 
  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  $\bigwedge A. A \in \text{set } As \Rightarrow B \in \# CA\_of A \Rightarrow B \neq Pos A \Rightarrow atm\_of B < A$ 
  using  $prod\_c0$  by auto
have  $\forall i. i < \text{length } AAs \rightarrow AAs ! i \neq \{\#\}$ 
  unfolding  $AAs\_def$  using  $m\_nz$  by simp
have  $\forall i < n. CAs ! i = Cs ! i + poss (AAs ! i)$ 
  unfolding  $CAs\_def Cs\_def AAs\_def$  using  $ca\_of\_c\_of\_aj\_of$  by (simp add:  $n\_def$ )
moreover have  $\forall i < n. AAs ! i \neq \{\#\}$ 
  using  $\langle \forall i < \text{length } AAs. AAs ! i \neq \{\#\} \rangle$  calculation(3) by blast
moreover have  $\forall i < n. \forall A \in \# AAs ! i. A = As ! i$ 
  by (simp add:  $\forall i < \text{length } AAs. \forall A \in \# AAs ! i. A = As ! i \rangle$  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  $\bigwedge i. i < \text{length } AAs \Rightarrow strictly\_maximal\_wrt (As ! i) ((Cs ! i))$ 
  by (simp add:  $C\_of\_def Cs\_def \langle \bigwedge x B. [\text{production } N (CA\_of x) = \{x\}; B \in \# CA\_of x; B \neq Pos x] \Rightarrow atm\_of B < x \rangle$  atms_of_def calculation(3)  $n\_def prod\_c0 strictly\_maximal\_wrt\_def$ )
have  $\forall i < n. strictly\_maximal\_wrt (As ! i) (Cs ! i)$ 
  by (simp add:  $\langle \bigwedge i. i < \text{length } AAs \Rightarrow strictly\_maximal\_wrt (As ! i) (Cs ! i) \rangle$  calculation(3))
moreover have  $\forall CA \in \text{set } CAs. S CA = \{\#\}$ 
  using  $prod\_c producesD productive\_imp\_produces\_Max\_literal$  by blast
have  $\forall CA \in \text{set } CAs. S CA = \{\#\}$ 
  using  $\langle \forall CA \in \text{set } CAs. S CA = \{\#\} \rangle$  by simp
then have  $\forall i < n. S (CAs ! i) = \{\#\}$ 
  using  $\langle \text{length } CAs = n \rangle$  nth_mem by blast
ultimately have  $res\_e: ord\_resolve CAs (D + negs (mset As)) AAs As (\sum \# (mset Cs) + D)$ 
  using  $ord\_resolve$  by auto

have  $\bigwedge A. A \in \text{set } As \Rightarrow \neg interp N (CA\_of A) \models CA\_of A$ 
  by (simp add:  $prod\_c0 producesD$ )
then have  $\bigwedge A. A \in \text{set } As \Rightarrow \neg Interp N (CA\_of A) \models C\_of A$ 
  unfolding  $prod\_c0 C\_of\_def Interp\_def true\_cls\_def$  using  $true\_lit\_def not\_gr\_zero prod\_c0$ 
  by auto
then have  $c'\_at\_n: \bigwedge A. A \in \text{set } As \Rightarrow \neg INTERP N \models C\_of A$ 
  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  $\neg INTERP N \models \sum \# (mset Cs)$ 
  unfolding  $Cs\_def true\_cls\_def$  using  $c'\_at\_n$  by fastforce
moreover have  $\neg INTERP N \models D$ 
  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  $e\_cex: \neg INTERP N \models \sum \# (mset Cs) + D$ 
  by simp

have  $\text{set } CAs \subseteq N$ 
  by (simp add: cs_subst_n)
moreover have  $INTERP N \models mset CAs$ 
  by (simp add: cs_true)
moreover have  $\bigwedge CA. CA \in \text{set } CAs \Rightarrow productive N CA$ 
  by (simp add:  $prod\_c$ )
moreover have  $ord\_resolve CAs DA AAs As (\sum \# (mset Cs) + D)$ 
  using  $D\_def negs\_as\_le\_d res\_e$  by auto
moreover have  $\neg INTERP N \models \sum \# (mset Cs) + D$ 

```

```

using e_cex by simp
moreover have  $\sum \# (\text{mset } Cs) + D < DA$ 
  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 $\subseteq (\bigcup C \in \text{set } CAs. \text{atms_of } C) \cup \text{atms_of } DA$

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 $\forall i < n. \text{set_mset } (Cs ! i) \subseteq \text{set_mset } (CAs ! i)$

using cas by auto

then have $\forall i < n. Cs ! i \subseteq \# (\text{mset } CAs)$

by (metis cas cas_len mset_subset_eq_add_left nth_mem_mset sum_mset.remove union_assoc)

then have $\forall C \in \text{set } Cs. C \subseteq \# (\text{mset } CAs)$

using cs_len in_set_conv_nth[of _ Cs] by auto

then have set_mset ($\sum \# (\text{mset } Cs)$) $\subseteq \text{set_mset } (\sum \# (\text{mset } CAs))$

by auto (meson in_mset_sum_list2 mset_subset_eqD)

then have atms_of ($\sum \# (\text{mset } Cs)$) $\subseteq \text{atms_of } (\sum \# (\text{mset } CAs))$

by (meson lits_subseteq_imp_atms_subseteq mset_subset_eqD subsetI)

moreover have atms_of ($\sum \# (\text{mset } CAs)$) = ($\bigcup CA \in \text{set } CAs. \text{atms_of } CA$)

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 ($\sum \# (\text{mset } Cs)$) $\subseteq (\bigcup CA \in \text{set } CAs. \text{atms_of } CA)$

by auto

moreover have atms_of D $\subseteq \text{atms_of } DA$

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 N :: 'a clause set and DA :: 'a clause
  assume {#}  $\notin N$  and DA  $\in N$  and  $\neg \text{INTERP } N \models DA$  and  $\bigwedge C. C \in N \implies \neg \text{INTERP } N \models C \implies DA \leq C$ 
  then obtain CAs AAs As E where
    dd_sset_n: set CAs  $\subseteq N$  and
    dd_true: INTERP N  $\models m \text{mset } CAs$  and
    res_e: ord_resolve CAs DA AAs As E and
    e_cex:  $\neg \text{INTERP } N \models E$  and
    e_lt_c: E < DA
    using ord_resolve_counterex_reducing[of N DA thesis] by auto

  have Infer (mset CAs) DA E  $\in \text{ord\_}\Gamma$ 
  using res_e unfolding ord_Γ_def by (metis (mono_tags, lifting) mem_Collect_eq)
  then show  $\exists CC E. \text{set\_mset } CC \subseteq N \wedge \text{INTERP } N \models m CC \wedge \text{Infer } CC DA E \in \text{ord\_}\Gamma$ 
   $\wedge \neg \text{INTERP } N \models E \wedge E < DA$ 

```

```

  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}_F ; Ri denotes \mathcal{R}_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:  $Ri N \subseteq \Gamma$  and
  Rf_mono:  $N \subseteq N' \implies Rf N \subseteq Rf N'$  and
  Ri_mono:  $N \subseteq N' \implies Ri N \subseteq Ri N'$  and
  Rf_indep:  $N' \subseteq Rf N \implies Rf N \subseteq Rf(N - N')$  and
  Ri_indep:  $N' \subseteq Rf N \implies Ri N \subseteq Ri(N - N')$  and
  Rf_sat: satisfiable( $N - Rf N$ ) ⇒ satisfiable  $N$ 
begin

```

```

definition saturated_up to :: "'a clause set ⇒ bool" where
  saturated_up to  $N \longleftrightarrow \text{inferences\_from}(N - Rf N) \subseteq Ri N$ 

```

```

inductive derive :: "'a clause set ⇒ 'a clause set ⇒ bool (infix <⇒> 50)" where
  deduction_deletion:  $N - M \subseteq \text{concls\_of}(\text{inferences\_from } M) \implies M - N \subseteq Rf N \implies M > N$ 

```

```

lemma derive_subset:  $M > N \implies N \subseteq M \cup \text{concls\_of}(\text{inferences\_from } M)$ 
  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(>)  $Ns$  and
  sat_n0: satisfiable(lhd  $Ns$ )
shows satisfiable(Sup_llist  $Ns$ )
proof -
  have len_ns: llength  $Ns > 0$ 
  using deriv by (case_tac  $Ns$ ) simp+
  {
    fix DD
    assume fin: finite  $DD$  and sset_lun:  $DD \subseteq \text{Sup\_llist } Ns$ 
    then obtain k where

```

```

dd_sset:  $DD \subseteq Sup\_upto\_llist\ Ns$  ( $enat\ k$ )
  using finite_Sup_llist_imp_Sup_uppto_llist by blast
  have satisfiable ( $Sup\_upto\_llist\ Ns\ k$ )
    proof (induct k)
      case 0
        then show ?case
        using len_ns sat_n0
        unfolding Sup_uppto_llist_def true_clss_def lhd_conv_lnth[ $OF\ chain\_not\_lnull[OF\ deriv]$ ]
        by auto
    next
      case ( $Suc\ k$ )
        show ?case
        proof (cases enat ( $Suc\ k$ )  $\geq$  llength Ns)
          case True
            then have  $Sup\_upto\_llist\ Ns\ (enat\ k) = Sup\_upto\_llist\ Ns\ (enat\ (Suc\ k))$ 
            unfolding Sup_uppto_llist_def using le_Suc_eq by auto
            then show ?thesis
            using Suc by simp
        next
          case False
          then have lnth Ns k  $\triangleright$  lnth Ns ( $Suc\ k$ )
          using deriv by (auto simp: chain_lnth_rel)
          then have lnth Ns ( $Suc\ k$ )  $\subseteq$  lnth Ns k  $\cup$  concls_of (inferences_from (lnth Ns k))
          by (rule derive_subset)
          moreover have lnth Ns k  $\subseteq$  Sup_uppto_llist Ns k
          unfolding Sup_uppto_llist_def using False Suc_ile_eq linear by blast
          ultimately have lnth Ns ( $Suc\ k$ )
           $\subseteq$  Sup_uppto_llist Ns k  $\cup$  concls_of (inferences_from (Sup_uppto_llist Ns k))
          by clarsimp (metis UnCI UnE image_Un inferences_from_mono le_iff_sup)
          moreover have  $Sup\_upto\_llist\ Ns\ (Suc\ k) = Sup\_upto\_llist\ Ns\ k \cup lnth\ Ns\ (Suc\ k)$ 
          unfolding Sup_uppto_llist_def using False by (force elim: le_SucE)
          moreover have
            satisfiable ( $Sup\_upto\_llist\ Ns\ k \cup concls\_of\ (inferences\_from\ (Sup\_upto\_llist\ Ns\ k))$ )
            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_uppto_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 ( $\triangleright$ ) Ns
  shows
    Rf_Sup_subset_Rf_Liminf:  $Rf\ (Sup\_llist\ Ns) \subseteq Rf\ (Liminf\_llist\ Ns)$  and
    Ri_Sup_subset_Ri_Liminf:  $Ri\ (Sup\_llist\ Ns) \subseteq Ri\ (Liminf\_llist\ Ns)$  and
    sat_limit_iff: satisfiable ( $Liminf\_llist\ Ns$ )  $\longleftrightarrow$  satisfiable ( $lhd\ Ns$ )
  proof -
  {
    fix C i j
    assume
      c_in:  $C \in lnth\ Ns\ i$  and
      c_ni:  $C \notin Rf\ (Sup\_llist\ Ns)$  and
      j:  $j \geq i$  and
      j':  $enat\ j < llength\ Ns$ 
    from c_ni have c_ni':  $\bigwedge i. enat\ i < llength\ Ns \implies C \notin Rf\ (lnth\ Ns\ i)$ 
    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 (▷) Ns
shows
  Rf_limit_Sup: Rf (Liminf_llist Ns) = Rf (Sup_llist Ns) and
  Ri_limit_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:  $\gamma \in \Gamma \implies \text{concl\_of } \gamma \in N \cup Rf N \implies \gamma \in Ri N$ 
begin

definition fair_clss_seq :: 'a clause set llist  $\Rightarrow$  bool where
  fair_clss_seq Ns  $\longleftrightarrow$  (let  $N' = \text{Liminf\_llist } Ns - Rf (\text{Liminf\_llist } Ns)$  in
   $\text{concls\_of} (\text{inferences\_from } N' - Ri N') \subseteq \text{Sup\_llist } Ns \cup Rf (\text{Sup\_llist } Ns)$ )
end

```

```

locale sat_preserving_effective_redundancy_criterion =
sat_preserving_inference_system  $\Gamma :: ('a :: \text{wellorder})$  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 ( $\triangleright$ ) Ns and
  fair: fair_clss_seq Ns
shows saturated_upto (Liminf_llist Ns)
unfolding saturated_upto_def
proof
fix  $\gamma$ 
let  $?N' = \text{Liminf\_llist } Ns - Rf (\text{Liminf\_llist } Ns)$ 
assume  $\gamma: \gamma \in \text{inferences\_from } ?N'$ 
show  $\gamma \in Ri (\text{Liminf\_llist } Ns)$ 
proof (cases  $\gamma \in Ri ?N'$ )
case True
then show ?thesis
using Ri_mono by blast
next
case False
have  $\text{concl\_of} (\text{inferences\_from } ?N' - Ri ?N') \subseteq \text{Sup\_llist } Ns \cup Rf (\text{Sup\_llist } Ns)$ 
using fair unfolding fair_clss_seq_def Let_def .
then have  $\text{concl\_of } \gamma \in \text{Sup\_llist } Ns \cup Rf (\text{Sup\_llist } Ns)$ 
using False  $\gamma$  by auto
moreover
{
assume  $\text{concl\_of } \gamma \in \text{Sup\_llist } Ns$ 
then have  $\gamma \in Ri (\text{Sup\_llist } Ns)$ 
using  $\gamma$  Ri_effective inferences_from_def by blast
then have  $\gamma \in Ri (\text{Liminf\_llist } Ns)$ 
using deriv Ri_Sup_subset_Ri_Liminf by fast
}
moreover
{
assume  $\text{concl\_of } \gamma \in Rf (\text{Sup\_llist } Ns)$ 
then have  $\text{concl\_of } \gamma \in Rf (\text{Liminf\_llist } Ns)$ 
using deriv Rf_Sup_subset_Rf_Liminf by blast
then have  $\gamma \in Ri (\text{Liminf\_llist } Ns)$ 
using  $\gamma$  Ri_effective inferences_from_def by auto
}
ultimately show  $\gamma \in Ri (\text{Liminf\_llist } Ns)$ 
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_up_to_iff: saturated_up_to N ↔ concls_of (inferences_from N) ⊆ N
  unfolding saturated_up_to_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 Γ ⊆ Γ' and redundancy_criterion Γ Rf Ri
  shows redundancy_criterion Γ' Rf (λN. Ri N ∪ (Γ' − Γ))
  using assms unfolding redundancy_criterion_def by (intro conjI) ((auto simp: rev_subsetD)[5], sat)

```

```

lemma redundancy_criterion_standard_extension_saturated_up_to_iff:
  assumes Γ ⊆ Γ' and redundancy_criterion Γ Rf Ri
  shows redundancy_criterion.saturated_up_to Γ Rf Ri M ↔
    redundancy_criterion.saturated_up_to Γ' Rf (λN. Ri N ∪ (Γ' − Γ)) M
  using assms redundancy_criterion.saturated_up_to_def redundancy_criterion.saturated_up_to_def
    redundancy_criterion_standard_extension
  unfolding inference_system.inferences_from_def by blast

```

```

lemma redundancy_criterion_standard_extension_effective:
  assumes Γ ⊆ Γ' and effective_redundancy_criterion Γ Rf Ri
  shows effective_redundancy_criterion Γ' Rf (λN. Ri N ∪ (Γ' − Γ))
  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 Γ ⊆ Γ' and effective_redundancy_criterion Γ Rf Ri
  shows effective_redundancy_criterion.fair_clss_seq Γ' Rf (λN. Ri N ∪ (Γ' − Γ)) Ns ↔
    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 (λN. Ri N ∪ (Γ' − Γ)) Ns]
  unfolding inference_system.inferences_from_def Let_def by auto

```

```

theorem redundancy_criterion_standard_extension_fair_derive_saturated_up_to:
  assumes
    subs: Γ ⊆ Γ' and
    red: redundancy_criterion Γ Rf Ri and
    red': sat_preserving_effective_redundancy_criterion Γ' Rf (λN. Ri N ∪ (Γ' − Γ)) and
    deriv: chain (redundancy_criterion.derive Γ' Rf) Ns and
    fair: effective_redundancy_criterion.fair_clss_seq Γ' Rf (λN. Ri N ∪ (Γ' − Γ)) Ns
  shows redundancy_criterion.saturated_up_to Γ Rf Ri (Liminf_llist Ns)

proof -
  have redundancy_criterion.saturated_up_to Γ' Rf (λN. Ri N ∪ (Γ' − Γ)) (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

definition redundant_infer :: 'a clause set ⇒ 'a inference ⇒ bool where
redundant_infer N γ ↔→
(∃ DD. set_mset DD ⊆ N ∧ (∀ I. I ⊨ m DD + side_prem_of γ → I ⊨ concl_of γ)
 ∧ (∀ D. D ∈# DD → D < main_prem_of γ))

definition Rf :: 'a clause set ⇒ 'a clause set where
Rf N = {C. ∃ DD. set_mset DD ⊆ N ∧ (∀ I. I ⊨ m DD → I ⊨ C) ∧ (∀ D. D ∈# DD → D < C)}

definition Ri :: 'a clause set ⇒ 'a inference set where
Ri N = {γ ∈ Γ. redundant_infer N γ}

```

```

lemma tautology_Rf:
assumes Pos A ∈# C
assumes Neg A ∈# C
shows C ∈ Rf N
proof -
have set_mset {#} ⊆ N ∧ (∀ I. I ⊨ m {#} → I ⊨ C) ∧ (∀ D. D ∈# {#} → D < C)
using assms by auto
then show C ∈ Rf N
unfolding Rf_def by blast
qed

```

```

lemma tautology_redundant_infer:
assumes
pos: Pos A ∈# concl_of i and
neg: Neg A ∈# concl_of i
shows redundant_infer N i
by (metis empty_iff empty_subsetI neg pos pos_neg_in_imp_true redundant_infer_def set_mset_empty)

```

```

lemma contradiction_Rf: {#} ∈ N ⇒ 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 ⊆ N' ⇒ Rf N ⊆ Rf N'
unfolding Rf_def by auto

```

```

lemma wlog_non_Rf:
assumes ex: ∃ DD. set_mset DD ⊆ N ∧ (∀ I. I ⊨ m DD + CC → I ⊨ E) ∧ (∀ D'. D' ∈# DD → D' < D)
shows ∃ DD. set_mset DD ⊆ N - Rf N ∧ (∀ I. I ⊨ m DD + CC → I ⊨ E) ∧ (∀ D'. D' ∈# DD → D' < D)
proof -
from ex obtain DD0 where

```

```

dd0:  $DD0 \in \{DD. \text{set\_mset } DD \subseteq N \wedge (\forall I. I \models_m DD + CC \rightarrow I \models E) \wedge (\forall D'. D' \in \# DD \rightarrow D' < D)\}$ 
by blast
have  $\exists DD. \text{set\_mset } DD \subseteq N \wedge (\forall I. I \models_m DD + CC \rightarrow I \models E) \wedge (\forall D'. D' \in \# DD \rightarrow D' < D) \wedge$ 
 $(\forall DD'. \text{set\_mset } DD' \subseteq N \wedge (\forall I. I \models_m DD' + CC \rightarrow I \models E) \wedge (\forall D'. D' \in \# DD' \rightarrow D' < D) \rightarrow$ 
 $DD \leq DD')$ 
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 ⊆ N and
  ddcc_imp_e: ∀ I. I ⊨_m DD + CC → I ⊨ E and
  dd_lt_d: ∀ D'. D' ∈ # DD → D' < D and
  d_min: ∀ DD'. set_mset DD' ⊆ N ∧ (forall I. I ⊨_m DD' + CC → I ⊨ E) ∧ (forall D'. D' ∈ # DD' → D' < D) →
    DD ≤ DD'
  by blast

have ∀ Da. Da ∈ # DD → Da ∉ Rf N
proof clarify
fix Da
assume
  da_in_dd: Da ∈ # DD and
  da_rf: Da ∈ Rf N

from da_rf obtain DD' where
  dd'_subs_n: set_mset DD' ⊆ N and
  dd'_imp_da: ∀ I. I ⊨_m DD' → I ⊨ Da and
  dd'_lt_da: ∀ D'. D' ∈ # DD' → D' < Da
  unfolding Rf_def by blast

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

have set_mset DDa ⊆ 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  $\forall D. D \in \# CC \rightarrow D \notin Rf N$ 
  using cc_subs by (simp add: subset_iff)
then have cc_nr:
   $\bigwedge C DD. C \in \# CC \Rightarrow set\_mset DD \subseteq N \Rightarrow \forall I. I \models m DD \rightarrow I \models C \Rightarrow \exists D. D \in \# DD \wedge \neg D < C$ 
    unfolding Rf_def by auto metis
have set_mset CC ⊆ N
  using cc_subs by auto
then have set_mset CC ⊆
   $N - \{C. \exists DD. set\_mset DD \subseteq N \wedge (\forall I. I \models m DD \rightarrow I \models C) \wedge (\forall D. D \in \# DD \rightarrow D < C)\}$ 
    using cc_nr by blast
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: $N \subseteq N' \Rightarrow Ri N \subseteq Ri N'$
 unfolding Ri_def redundant_infer_def by auto

lemma Ri_subs_Ri_diff_Rf: $Ri N \subseteq Ri (N - Rf N)$

proof

fix γ

assume γ_ri: $\gamma \in Ri N$

then obtain CC D E where γ: Infer CC D E

by (cases γ)

have cc: CC = side_prem_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 $\forall I. I \models m DD + CC \rightarrow I \models E$ and $\forall C. C \in \# DD \rightarrow C < D$
 using γ_ri unfolding Ri_def redundant_infer_def cc d e by blast

then obtain DD' where

set_mset DD' ⊆ N - Rf N and $\forall I. I \models m DD' + CC \rightarrow I \models E$ and $\forall D'. D' \in \# DD' \rightarrow D' < D$
 using wlog_non_Rf by atomize_elim blast

then show γ ∈ Ri (N - Rf N)

using γ_ri unfolding Ri_def redundant_infer_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: $Ri N \subseteq \Gamma$

unfolding Ri_def by blast

lemma Rf_indep: $N' \subseteq Rf N \Rightarrow Rf N \subseteq Rf (N - N')$
 by (metis Diff_cancel Diff_eq_empty_iff Diff_mono Rf_eq_Rf_diff_Rf Rf_mono)

lemma Ri_indep: $N' \subseteq Rf N \Rightarrow Ri N \subseteq Ri (N - N')$
 by (metis Diff_mono Ri_eq_Ri_diff_Rf Ri_mono order_refl)

lemma Rf_model:

assumes $I \models s N - Rf N$

shows $I \models s N$

proof –

have $I \models s Rf (N - Rf N)$

unfolding true_clss_def

by (subst Rf_def, simp add: true_clss_mset_def, metis assms subset_eq true_clss_def)

then have $I \models s Rf N$

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*) \implies satisfiable *N*
by (*metis Rf_model*)

The following corresponds to Theorem 4.7:

```
sublocale redundancy_criterion  $\Gamma$  Rf Ri
  by unfold_locales (rule Ri_subset_ $\Gamma$ , (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_ $\gamma$ :  $\gamma \in \Gamma$  and
    concl_of_in_n_un_rf_n: concl_of  $\gamma \in N \cup Rf N$ 
  shows  $\gamma \in Ri N$ 
proof -
  obtain CC D E where
     $\gamma: \gamma = Infer CC D E$ 
    by (cases  $\gamma$ )
  then have cc: CC = side_prem_of  $\gamma$  and d: D = main_prem_of  $\gamma$  and e: E = concl_of  $\gamma$ 
    unfolding  $\gamma$  by simp_all
  note e_in_n_un_rf_n = concl_of_in_n_un_rf_n[folded e]

  {
    assume E  $\in N$ 
    moreover have E < D
      using  $\Gamma\_reductive e d in_\gamma$  by auto
    ultimately have
      set_mset {#E#} ⊆ N and  $\forall I. I \models_m \{#E#\} + CC \longrightarrow I \models E$  and  $\forall D'. D' \in \{#E#\} \longrightarrow D' < D$ 
      by simp_all
    then have redundant_infer  $N \gamma$ 
      using redundant_infer_def cc d e by blast
  }
  moreover
  {
    assume E  $\in Rf N$ 
    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  $\forall Da. Da \in \{#DD\} \longrightarrow Da < D$ 
      using d e in_γ Γ_reductive less_trans by blast
    then have redundant_infer  $N \gamma$ 
      using redundant_infer_def dd_sset dd_imp_e cc d e by blast
  }
  ultimately show  $\gamma \in Ri N$ 
    using in_γ e_in_n_un_rf_n unfolding Ri_def by blast
qed
```

```
sublocale effective_redundancy_criterion  $\Gamma$  Rf Ri
  unfolding effective_redundancy_criterion_def
  by (intro conjI redundancy_criterion_axioms, unfold_locales, rule Ri_effective)
```

```
lemma contradiction_Rf:  $\{\#\} \in N \implies Ri N = \Gamma$ 
  unfolding Ri_def redundant_infer_def using  $\Gamma\_reductive le_multiset_empty_right$ 
  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 ⊨ M
      proof (rule ccontr)
        assume ¬ I_of M ⊨ 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 ⊨m CC and
          γ_in: γ ∈ Γ and
          e_cex: ¬ I_of M ⊨ E and
          e_lt_d: E < D
          using Γ_counterex_reducing[OF ec_ni_m] not_less by metis
        have cc: CC = side_prem_of γ and d: D = main_prem_of γ and e: E = concl_of γ
          unfolding γ by simp_all
        have γ ∈ Ri N
          by (rule subsetD[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 γ ∈ Ri M
          unfolding M_def using Ri_indep by fast
        then obtain DD where
          dd_subs_m: set_mset DD ⊆ M and
          dd_cc_imp_d: ∀ I. I ⊨m DD + CC → I ⊨ E and
          dd_lt_d: ∀ C. C ∈# DD → C < D
          unfolding Ri_def redundant_infer_def cc d e by blast
        from dd_subs_m dd_lt_d have I_of M ⊨m DD
          using d_min unfolding true_cls_mset_def by (metis contra_subsetD)
        then have I_of M ⊨ E
          using dd_cc_imp_d cc_true by auto
        then show False
          using e_cex by auto
      qed
    then have I_of M ⊨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 ¬ satisfiable N ↔ {#} ∈ 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 renamings_apart atm_of_atms mgu
for
  subst_atm :: 'a :: wellorder ⇒ 's ⇒ 'a and
  id_subst :: 's and
  comp_subst :: 's ⇒ 's ⇒ 's and
  renamings_apart :: 'a literal multiset list ⇒ 's list and
  atm_of_atms :: 'a list ⇒ 'a and
  mgu :: 'a set set ⇒ 's option +
fixes
  less_atm :: 'a ⇒ 'a ⇒ bool
assumes
  less_atm_stable: less_atm A B ⇒ less_atm (A · a σ) (B · a σ) and
  less_atm_ground: is_ground_atm A ⇒ is_ground_atm B ⇒ less_atm A B ⇒ A < B
begin

```

14.1 Library

```

lemma Bex_cartesian_product: (Ǝ xy ∈ A × B. P xy) ≡ (Ǝ x ∈ A. Ǝ y ∈ B. P (x, y))
  by simp

```

```

lemma eql_map_neg_lit_eql_atm:
  assumes map (λL. L · l η) (map Neg As') = map Neg As
  shows As' · al η = As
  using assms by (induction As' arbitrary: As) auto

```

```

lemma instance_list:
  assumes negs (mset As) = SDA' · η
  shows ∃ As'. negs (mset As') = SDA' ∧ As' · al η = As
proof -
  from assms have negL: ∀ L ∈ # SDA'. is_neg L
    using Melem_subst_cls subst_lit_in_negs_is_neg by metis

```

```

from assms have {#L · l η. L ∈ # SDA' #} = mset (map Neg As)
  using subst_cls_def by auto
then have ∃ NAs'. map (λL. L · l η) NAs' = map Neg As ∧ mset NAs' = SDA'
  using image_mset_of_subset_list[of λL. L · l η SDA' map Neg As] by auto
then obtain As' where As'_p:
  map (λL. L · l η) (map Neg As') = map Neg As ∧ 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' · al  $\eta$  = As
  using eql_map_neg_lit_eql_atm by auto
ultimately show ?thesis
  by blast
qed

lemma map2_add_mset_map:
assumes length AAs' = n and length As' = n
shows map2 add_mset (As' · al  $\eta$ ) (AAs' · aml  $\eta$ ) = map2 add_mset As' AAs' · aml  $\eta$ 
using assms
proof (induction n arbitrary: AAs' As')
case (Suc n)
then have map2 add_mset (tl (As' · al  $\eta$ )) (tl (AAs' · aml  $\eta$ )) = map2 add_mset (tl As') (tl AAs') · aml  $\eta$ 
  by simp
moreover have Succ: length (As' · al  $\eta$ ) = Suc n length (AAs' · aml  $\eta$ ) = Suc n
  using Suc(3) Suc(2) by auto
then have length (tl (As' · al  $\eta$ )) = n length (tl (AAs' · aml  $\eta$ )) = n
  by auto
then have length (map2 add_mset (tl (As' · al  $\eta$ )) (tl (AAs' · aml  $\eta$ ))) = n
  length (map2 add_mset (tl As') (tl AAs') · aml  $\eta$ ) = n
  using Suc(2,3) by auto
ultimately have  $\forall i < n. tl (map2 add_mset ((As' · al  $\eta$ )) ((AAs' · aml  $\eta$ ))) ! i =$ 
  tl (map2 add_mset (As') (AAs') · aml  $\eta$ ) ! 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  $\eta$ )) ((AAs' · aml  $\eta$ ))) = Suc n
  length (map2 add_mset (As') (AAs') · aml  $\eta$ ) = Suc n
  using Succ Suc by auto
ultimately have  $\forall i. i < Suc n \longrightarrow i > 0 \longrightarrow$ 
  map2 add_mset (As' · al  $\eta$ ) (AAs' · aml  $\eta$ ) ! i = (map2 add_mset As' AAs' · aml  $\eta$ ) ! i
  by (auto simp: subst_atm_mset_list_def gr0_conv_Suc subst_atm_mset_def)
moreover have add_mset (hd As' · a  $\eta$ ) (hd AAs' · am  $\eta$ ) = add_mset (hd As') (hd AAs') · am  $\eta$ 
  unfolding subst_atm_mset_def by auto
then have (map2 add_mset (As' · al  $\eta$ ) (AAs' · aml  $\eta$ )) ! 0 = (map2 add_mset (As') (AAs') · aml  $\eta$ ) ! 0
  using Suc by (simp add: Succ(2) subst_atm_mset_def)
ultimately have  $\forall i < Suc n. (map2 add_mset (As' · al  $\eta$ ) (AAs' · aml  $\eta$ )) ! i =$ 
  (map2 add_mset (As') (AAs') · aml  $\eta$ ) ! i
  using Suc by auto
then show ?case
  using nn list_eq_iff_nth_eq by metis
qed auto

```

context

```

fixes S :: 'a clause  $\Rightarrow$  'a clause
begin

```

14.2 Calculus

The following corresponds to Figure 4.

```

definition maximal_wrt :: 'a  $\Rightarrow$  'a literal multiset  $\Rightarrow$  bool where
maximal_wrt A C  $\longleftrightarrow$  ( $\forall B \in \text{atms\_of } C. \neg \text{less\_atm } A B$ )

```

```

definition strictly_maximal_wrt :: 'a  $\Rightarrow$  'a literal multiset  $\Rightarrow$  bool where
strictly_maximal_wrt A C  $\equiv$   $\forall B \in \text{atms\_of } C. A \neq B \wedge \neg \text{less\_atm } A B$ 

```

```

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

```

```

lemma maximal_wrt_subst: maximal_wrt (A · a  $\sigma$ ) (C ·  $\sigma$ )  $\Longrightarrow$  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 ( $A \cdot a \sigma$ ) ( $C \cdot \sigma$ )  $\Rightarrow$  strictly_maximal_wrt  $A \cdot C$ 
unfolding strictly_maximal_wrt_def using in_atms_of_subst less_atm_stable by blast

inductive eligible :: 's  $\Rightarrow$  'a list  $\Rightarrow$  'a clause  $\Rightarrow$  bool where
  eligible:
     $S \cdot DA = \text{negs } (\text{mset } As) \vee S \cdot DA = \{\#\} \wedge \text{length } As = 1 \wedge \text{maximal_wrt } (As ! 0 \cdot a \sigma) (DA \cdot \sigma) \Rightarrow$ 
    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  $\Rightarrow$ 
    length Cs = n  $\Rightarrow$ 
    length AAs = n  $\Rightarrow$ 
    length As = n  $\Rightarrow$ 
     $n \neq 0 \Rightarrow$ 
     $(\forall i < n. CAs ! i = Cs ! i + \text{poss } (AAs ! i)) \Rightarrow$ 
     $(\forall i < n. AAs ! i \neq \{\#\}) \Rightarrow$ 
    Some  $\sigma = \text{mgu } (\text{set\_mset } ' \text{set } (\text{map2 add\_mset } As \cdot AAs)) \Rightarrow$ 
    eligible  $\sigma$  As ( $D + \text{negs } (\text{mset } As)$ )  $\Rightarrow$ 
     $(\forall i < n. \text{strictly\_maximal\_wrt } (As ! i \cdot a \sigma) (Cs ! i \cdot \sigma)) \Rightarrow$ 
     $(\forall i < n. S(CAs ! i) = \{\#\}) \Rightarrow$ 
    ord_resolve CAs ( $D + \text{negs } (\text{mset } As)$ ) AAs As  $\sigma ((\sum \# (\text{mset } Cs) + D) \cdot \sigma)$ 

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  $\Rightarrow$ 
    length AAs = n  $\Rightarrow$ 
    length As = n  $\Rightarrow$ 
     $(\forall i < n. \text{poss } (AAs ! i) \subseteq \# CAs ! i) \Rightarrow$ 
    negs (mset As)  $\subseteq \# DA \Rightarrow$ 
     $\varrho = \text{hd } (\text{renamings\_apart } (DA \# CAs)) \Rightarrow$ 
     $\varphi = \text{tl } (\text{renamings\_apart } (DA \# CAs)) \Rightarrow$ 
    ord_resolve (CAs .. cl  $\varphi s$ ) ( $DA \cdot \varrho$ ) (AAs .. aml  $\varphi s$ ) (As .. al  $\varrho$ )  $\sigma E \Rightarrow$ 
    ord_resolve_rename CAs DA AAs As  $\sigma E$ 

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

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

```

14.3 Soundness

Soundness is not discussed in the chapter, but it is an important property.

```

lemma ord_resolve_ground_inst_sound:
assumes
  res_e: ord_resolve CAs DA AAs As  $\sigma E$  and
  cc_inst_true:  $I \models m \text{mset } CAs \cdot cm \sigma \cdot cm \eta$  and
  d_inst_true:  $I \models DA \cdot \sigma \cdot \eta$  and
  ground_subst_eta: is_ground_subst  $\eta$ 
shows  $I \models E \cdot \eta$ 
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 ( $\sigma \odot \eta$ )
  using ground_subst_<math>\eta</math> by (rule is_ground_comp_subst)
then have cc_true:  $I \models mset CAs \cdot cm \sigma \cdot cm \eta$  and d_true:  $I \models DA \cdot \sigma \cdot \eta$ 
  using cc_inst_true d_inst_true by auto

from mgu have unif:  $\forall i < n. \forall A \in \#AAs ! i. A \cdot a \sigma = As ! i \cdot a \sigma$ 
  using mgu_unifier as_len aas_len by blast

show  $I \models E \cdot \eta$ 
proof (cases  $\forall A \in set As. A \cdot a \sigma \cdot a \eta \in I$ )
  case True
    then have  $\neg I \models negs (mset As) \cdot \sigma \cdot \eta$ 
      unfolding true_cls_def[of I] by auto
    then have  $I \models D \cdot \sigma \cdot \eta$ 
      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:  $(As ! i) \cdot a \sigma \cdot a \eta \notin I$ 
      using da_len by (metis in_set_conv_nth)
    define C where  $C \equiv Cs ! i$ 
    define BB where  $BB \equiv AAs ! i$ 
    have c_cf':  $C \subseteq \# \sum \# (mset CAs)$ 
      unfolding C_def using a_in_aa cas_len
      by (metis less_subset_eq_Union_mset mset_subset_eq_add_left subset_mset_trans)
    have c_in_cc:  $C + poss BB \in \# mset CAs$ 
      using C_def BB_def a_in_aa cas_len in_set_conv_nth cas by fastforce
    {
      fix B
      assume B:  $B \in \# BB$ 
      then have B:  $B \cdot a \sigma = (As ! i) \cdot a \sigma$ 
        using unif a_in_aa cas_len unfolding BB_def by auto
    }
    then have  $\neg I \models poss BB \cdot \sigma \cdot \eta$ 
      using a_false by (auto simp: true_cls_def)
    moreover have I:  $I \models (C + poss BB) \cdot \sigma \cdot \eta$ 
      using c_in_cc cc_true true_cls_mset_true_cls[of I mset CAs · cm σ · cm η] by force
    ultimately have I:  $I \models C \cdot \sigma \cdot \eta$ 
      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

```

The previous lemma is not only used to prove soundness, but also the following lemma which is used to prove Lemma 4.10.

```

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)

```

Here follows the soundness theorem for the resolution rule.

```

theorem ord_resolve_sound:
assumes
  res_e: ord_resolve CAs DA AAs As σ E and
  cc_d_true:  $\bigwedge \sigma. \text{is\_ground\_subst } \sigma \implies I \models_m (\text{mset } CAs + \{\#DA\}) \cdot cm \sigma$  and
  ground_subst_η: is_ground_subst η
shows  $I \models E \cdot \eta$ 
proof (use res_e in ⟨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)
    have ground_subst_σ_η: is_ground_subst ( $\sigma \odot \eta$ )
      using ground_subst_η by (rule is_ground_comp_subst)
    have cas_true:  $I \models_m \text{mset } CAs \cdot cm \sigma \cdot cm \eta$ 
      using cc_d_true ground_subst_σ_η by fastforce
    have da_true:  $I \models DA \cdot \sigma \cdot \eta$ 
      using cc_d_true ground_subst_σ_η by fastforce
    show  $I \models E \cdot \eta$ 
      using ord_resolve_ground_inst_sound[OF res_e cas_true da_true] ground_subst_η by auto
qed

```

```

lemma subst_sound:
assumes
   $\bigwedge \sigma. \text{is\_ground\_subst } \sigma \implies I \models C \cdot \sigma$  and
  is_ground_subst_η
shows  $I \models C \cdot \varrho \cdot \eta$ 
using assms is_ground_comp_subst subst_cls_comp_subst by metis

```

```

lemma subst_sound_scl:
assumes
  len: length P = length CAs and
  true_cas:  $\bigwedge \sigma. \text{is\_ground\_subst } \sigma \implies I \models_m \text{mset } CAs \cdot cm \sigma$  and
  ground_subst_η: is_ground_subst η
shows  $I \models_m \text{mset } (CAs \cdot \cdot cl P) \cdot cm \eta$ 
proof -
  from true_cas have  $\bigwedge CA. CA \in \# \text{mset } CAs \implies (\bigwedge \sigma. \text{is\_ground\_subst } \sigma \implies I \models CA \cdot \sigma)$ 
    unfolding true_cls_mset_def by force
  then have  $\forall i < \text{length } CAs. \forall \sigma. \text{is\_ground\_subst } \sigma \longrightarrow (I \models CAs ! i \cdot \sigma)$ 
    using in_set_conv_nth by auto
  then have true_cp:  $\forall i < \text{length } CAs. \forall \sigma. \text{is\_ground\_subst } \sigma \longrightarrow I \models CAs ! i \cdot P ! i \cdot \sigma$ 
    using subst_sound len by auto

```

```

{
  fix CA
  assume CA ∈ # mset (CAs · · cl P)
  then obtain i where
    i_x:  $i < \text{length } (CAs \cdot \cdot cl P)$  CA = (CAs · · cl P) ! i
    by (metis in_mset_conv_nth)
  then have  $\forall \sigma. \text{is\_ground\_subst } \sigma \longrightarrow I \models CA \cdot \sigma$ 
    using true_cp unfolding subst_cls_lists_def by (simp add: len)
}
then show ?thesis
  using assms unfolding true_cls_mset_def by auto
qed

```

Here follows the soundness theorem for the resolution rule with renaming.

```

lemma ord_resolve_rename_sound:
assumes
  res_e: ord_resolve_rename CAs DA AAs As σ E and
  cc_d_true:  $\bigwedge \sigma. \text{is\_ground\_subst } \sigma \implies I \models_m ((\text{mset } CAs) + \{\#DA\}) \cdot cm \sigma$  and
  ground_subst_η: is_ground_subst η
shows  $I \models E \cdot \eta$ 
using res_e
proof (cases rule: ord_resolve_rename.cases)

```

```

case (ord_resolve_rename n  $\varrho$   $\varrho s$ )
note  $\varrho s = \text{this}(7)$  and  $\text{res} = \text{this}(8)$ 
have  $\text{len}: \text{length } \varrho s = \text{length } CAs$ 
  using  $\varrho s \text{ renamings\_apart\_length}$  by auto
have  $\bigwedge \sigma. \text{is\_ground\_subst } \sigma \implies I \models m (\text{mset } (CAs \cdot \text{cl } \varrho s) + \{\#\text{DA} \cdot \varrho\#}) \cdot cm \sigma$ 
  using subst_sound_scl[ $\text{OF } \text{len}$ , of  $I$ ] subst_sound cc_d_true by auto
then show  $I \models E \cdot \eta$ 
  using ground_subst_η ord_resolve_sound[ $\text{OF } \text{res}$ ] by simp
qed

```

14.4 Other Basic Properties

```

lemma ord_resolve_unique:
assumes
  ord_resolve CAs DA AAs As  $\sigma$  E and
  ord_resolve CAs DA AAs As  $\sigma'$  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  $\sigma''$  DA CAs' n' Cs' AAs' As'  $\sigma'''$  DA')
  note res = this(1–17) and res' = this(18–34)

  show  $\sigma: \sigma = \sigma'$ 
    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)  $\sigma$  by blast
qed

```

```

lemma ord_resolve_rename_unique:
assumes
  ord_resolve_rename CAs DA AAs As  $\sigma$  E and
  ord_resolve_rename CAs DA AAs As  $\sigma'$  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_prem: ord_resolve CAs DA AAs As  $\sigma$  E  $\implies$  length CAs  $\leq$  size DA
  by (auto elim!: ord_resolve.cases)

```

```

lemma ord_resolve_rename_max_side_prem:
  ord_resolve_rename CAs DA AAs As  $\sigma$  E  $\implies$  length CAs  $\leq$  size DA
  by (elim ord_resolve_rename.cases, drule ord_resolve_max_side_prem, simp add: renamings_apart_length)

```

14.5 Inference System

```

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

```

```

interpretation ord_FO_resolution: inference_system ord_FO_Γ .

```

```

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 = ( $\bigcup D \in ?CCC. \text{atms\_of } D$ )
  define max_ary where max_ary = Max (size ` ?CCC)
  define CAS where CAS = {CAs. CAs ∈ lists ?CCC ∧ length CAs  $\leq$  max_ary}
  define AS where AS = {As. As ∈ lists all_AA ∧ length As  $\leq$  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_prem[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_pos 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_premms[OF res_e] by auto
also have size DA ≤ max_ary
  unfolding max_ary_def using fin_cc da_in by auto
finally show length As ≤ max_ary
  .

qed
qed
}
then show ?thesis
  by simp fast

```

```

qed
also have ... ⊑ (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

```

14.6 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 ⇒ 'a literal multiset where
  S_M C =
    (if C ∈ grounding_of_clss M then
      (SOME C'. ∃ D σ. D ∈ M ∧ C = D · σ ∧ C' = S D · σ ∧ is_ground_subst σ)
    else
      S C)

lemma S_M_grounding_of_clss:
  assumes C ∈ grounding_of_clss M
  obtains D σ where
    D ∈ M ∧ C = D · σ ∧ S_M C = S D · σ ∧ is_ground_subst σ
proof (atomize_elim, unfold S_M_def eqTrueI[OF assms] if_True, rule someI_ex)
  from assms show ∃ C' D σ. D ∈ M ∧ C = D · σ ∧ C' = S D · σ ∧ is_ground_subst σ
    by (auto simp: grounding_of_clss_def grounding_of_cls_def)
qed

lemma S_M_not_grounding_of_clss: C ∉ grounding_of_clss M ⇒ S_M C = S C
  unfolding S_M_def by simp

lemma S_M_selects_subseteq: S_M C ⊆# 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 ∈# S_M C ⇒ 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

```

The following corresponds to Lemma 4.12:

```

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:  $\sum_{\#} (\text{mset } Cs) \subseteq \# \sum_{\#} (\text{mset } CAs)$ 
using subseq_list_Union_mset cas_len cs_len by force
then have cs_sub_cas:  $\sum_{\#} (\text{mset } Cs) \subseteq \# \sum_{\#} (\text{mset } CAs)$ 
using subseq_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  $\subseteq \# DA$ 
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 ( $\sum_{\#} (\text{mset } Cs) + D$ )
using gr_cs gr_d by auto
with e have E =  $\sum_{\#} (\text{mset } Cs) + D$ 
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} ∪ set CAs ⊆ grounding_of_clss M and
n: length CAs = n and
d: DA = D + negs (mset As) and
c: (∀ i < n. CAs ! i = Cs ! i + poss (AAs ! i)) length Cs = n length AAs = n
obtains DA0 η0 CAs0 ηs0 As0 AAs0 D0 Cs0 where
length CAs0 = n
length ηs0 = n
DA0 ∈ M
DA0 · η0 = DA
S DA0 · η0 = S_M S M DA
∀ CA0 ∈ set CAs0. CA0 ∈ M
CAs0 ..cl ηs0 = CAs
map S CAs0 ..cl ηs0 = map (S_M S M) CAs
is_ground_subst η0
is_ground_subst_list ηs0
As0 ·al η0 = As
AAs0 ..aml ηs0 = AAs
length As0 = n
D0 · η0 = D
DA0 = D0 + (negs (mset As0))
S_M S M (D + negs (mset As)) ≠ {#}  $\Rightarrow$  negs (mset As0) = S DA0
length Cs0 = n
Cs0 ..cl ηs0 = Cs
∀ i < n. CAs0 ! i = Cs0 ! i + poss (AAs0 ! i)
length AAs0 = 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 ≠ 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:  $\forall i < n. \text{CAs} ! i = \text{Cs} ! i + \text{poss} (\text{AAs} ! i)$ 
using ord_resolve by force+

note  $n = \langle n \neq 0 \rangle \langle \text{length } \text{CAs} = n \rangle \langle \text{length } \text{Cs} = n \rangle \langle \text{length } \text{AAs} = n \rangle \langle \text{length } \text{As} = n \rangle$ 

interpret  $S$ : selection  $S$  by (rule select)

— Obtain FO side premises
have  $\forall CA \in \text{set } \text{CAs}. \exists CA0 \eta c0. CA0 \in M \wedge CA0 \cdot \eta c0 = CA \wedge S CA0 \cdot \eta c0 = S\_M S M CA \wedge \text{is\_ground\_subst}$ 
 $\eta c0$ 
using grounding  $S\_M\_\text{grounding\_of\_clss}$  select by (metis (no_types) le_supE subset_iff)
then have  $\forall i < n. \exists CA0 \eta c0. CA0 \in M \wedge CA0 \cdot \eta c0 = (\text{CAs} ! i) \wedge S CA0 \cdot \eta c0 = S\_M S M (\text{CAs} ! i) \wedge$ 
 $\text{is\_ground\_subst } \eta c0$ 
using  $n$  by force
then obtain  $\eta s0f \text{CAs0f}$  where  $f\_p$ :
 $\forall i < n. \text{CAs0f } i \in M$ 
 $\forall i < n. (\text{CAs0f } i) \cdot (\eta s0f i) = (\text{CAs} ! i)$ 
 $\forall i < n. S (\text{CAs0f } i) \cdot (\eta s0f i) = S\_M S M (\text{CAs} ! i)$ 
 $\forall i < n. \text{is\_ground\_subst } (\eta s0f i)$ 
using  $n$  by (metis (no_types))

define  $\eta s0$  where
 $\eta s0 = \text{map } \eta s0f [0 .. < n]$ 
define  $\text{CAs0}$  where
 $\text{CAs0} = \text{map } \text{CAs0f} [0 .. < n]$ 

have  $\text{length } \eta s0 = n \text{ length } \text{CAs0} = n$ 
unfolding  $\eta s0\_\text{def}$   $\text{CAs0}\_\text{def}$  by auto
note  $n = \langle \text{length } \eta s0 = n \rangle \langle \text{length } \text{CAs0} = n \rangle n$ 

— The properties we need of the FO side premises
have  $\text{CAs0\_in\_M}: \forall CA0 \in \text{set } \text{CAs0}. CA0 \in M$ 
unfolding  $\text{CAs0}\_\text{def}$  using  $f\_p(1)$  by auto
have  $\text{CAs0\_to\_CAs}: \text{CAs0} \cdot \text{cl } \eta s0 = \text{CAs}$ 
unfolding  $\text{CAs0}\_\text{def}$   $\eta s0\_\text{def}$  using  $f\_p(2)$  by (auto simp: n_intro nth_equalityI)
have  $\text{SCAs0\_to\_SMCAs}: (\text{map } S \text{CAs0}) \cdot \text{cl } \eta s0 = \text{map } (S\_M S M) \text{CAs}$ 
unfolding  $\text{CAs0}\_\text{def}$   $\eta s0\_\text{def}$  using  $f\_p(3)$   $n$  by (force intro: nth_equalityI)
have  $\text{sub\_ground}: \forall \eta c0 \in \text{set } \eta s0. \text{is\_ground\_subst } \eta c0$ 
unfolding  $\eta s0\_\text{def}$  using  $f\_p n$  by force
then have  $\text{is\_ground\_subst\_list } \eta s0$ 
using  $n$  unfolding  $\text{is\_ground\_subst\_list}\_\text{def}$  by auto

— Split side premises CAs0 into Cs0 and AAs0
obtain  $\text{AAs0 } \text{Cs0}$  where  $\text{AAs0\_Cs0\_p}$ :
 $\text{AAs0} \cdot \text{aml } \eta s0 = \text{AAs} \text{ length } \text{Cs0} = n \text{ Cs0} \cdot \text{cl } \eta s0 = \text{Cs}$ 
 $\forall i < n. \text{CAs0} ! i = \text{Cs0} ! i + \text{poss} (\text{AAs0} ! i) \text{ length } \text{AAs0} = n$ 
proof —
have  $\forall i < n. \exists AA0. AA0 \cdot \text{am } \eta s0 ! i = \text{AAs} ! i \wedge \text{poss } AA0 \subseteq \# \text{CAs0} ! i$ 
proof (rule, rule)
fix  $i$ 
assume  $i < n$ 
have  $\text{CAs0} ! i \cdot \eta s0 ! i = \text{CAs} ! i$ 
using  $\langle i < n \rangle \langle \text{CAs0} \cdot \text{cl } \eta s0 = \text{CAs} \rangle n$  by force
moreover have  $\text{poss} (\text{AAs} ! i) \subseteq \# \text{CAs} ! i$ 
using  $\langle i < n \rangle \text{cas}$  by auto
ultimately obtain  $\text{poss\_AA0}$  where
 $nn: \text{poss\_AA0} \cdot \eta s0 ! i = \text{poss} (\text{AAs} ! i) \wedge \text{poss\_AA0} \subseteq \# \text{CAs0} ! i$ 
using cas_image_mset_of_subset unfolding subst_cls_def by metis
then have  $l: \forall L \in \# \text{poss\_AA0}. \text{is\_pos } L$ 
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  $AA0$  where

```

```

AA0 = image_mset atm_of poss_AA0

have na: poss AA0 = poss_AA0
  using l unfolding AA0_def by auto
then have AA0 · am ηs0 ! i = AAs ! i
  using nn by (metis (mono_tags) literal.inject(1) multiset.inj_map_strong subst_cls_poss)
moreover have poss AA0 ⊆# CAs0 ! i
  using na nn by auto
ultimately show ∃ AA0. AA0 · am ηs0 ! i = AAs ! i ∧ poss AA0 ⊆# CAs0 ! i
  by blast
qed
then obtain AAs0f where
  AAs0f_p: ∀ i < n. AAs0f i · am ηs0 ! i = AAs ! i ∧ (poss (AAs0f i)) ⊆# CAs0 ! i
  by metis

define AAs0 where AAs0 = map AAs0f [0 ..

```

— Obtain FO main premise
have $\exists DA0 \eta0. DA0 \in M \wedge DA = DA0 \cdot \eta0 \wedge S DA0 \cdot \eta0 = S_M S M DA \wedge is_ground_subst \eta0$
 using grounding S_M_grounding_of_clss select **by** (metis le_supE singletonI subsetCE)
then obtain DA0 η0 **where**

DA0_η0_p: DA0 ∈ M ∧ DA = DA0 · η0 ∧ S DA0 · η0 = S_M S M DA ∧ is_ground_subst η0
 by auto

— The properties we need of the FO main premise

have DA0_in_M: DA0 ∈ M
 using DA0_η0_p by auto
have DA0_to_DA: DA0 · η0 = DA
 using DA0_η0_p by auto
have SDA0_to_SMDA: S DA0 · η0 = S_M S M DA
 using DA0_η0_p by auto
have is_ground_subst η0
 using DA0_η0_p by auto

— Split main premise DA0 into D0 and As0

obtain D0 As0 where D0As0_p:

$$As0 \cdot al \eta 0 = As \text{ length } As0 = n \quad D0 \cdot \eta 0 = D \text{ DA0} = D0 + (\text{negs } (\text{mset } As0))$$

$$S_M S M (D + \text{negs } (\text{mset } As)) \neq \{\#\} \implies \text{negs } (\text{mset } As0) = S \text{ DA0}$$

proof —

```
{
  assume a:  $S\_M S M (D + \text{negs } (\text{mset } As)) = \{\#\} \wedge \text{length } As = (\text{Suc } 0)$ 
     $\wedge \text{maximal\_wrt } (As ! 0 \cdot a \sigma) ((D + \text{negs } (\text{mset } As)) \cdot \sigma)$ 
  then have as:  $\text{mset } As = \{\#As ! 0\#\}$ 
    by (auto intro: nth_equalityI)
  then have negs (mset As) =  $\{\#Neg(As ! 0)\#}$ 
    by (simp add: mset As =  $\{\#As ! 0\#\}$ )
  then have DA =  $D + \{\#Neg(As ! 0)\#}$ 
    using da by auto
  then obtain L where  $L \in \# DA0 \wedge L \cdot l \eta 0 = Neg(As ! 0)$ 
    using DA0_to_DA by (metis Melem_subst_cls mset_subset_eq_add_right single_subset_iff)
  then have Neg (atm_of L)  $\in \# DA0 \wedge Neg(atm\_of L) \cdot l \eta 0 = Neg(As ! 0)$ 
    by (metis Neg_atm_of_iff literal.sel(2) subst_lit_is_pos)
  then have [atm_of L]  $\cdot al \eta 0 = As \wedge \text{negs } (\text{mset } [atm\_of L]) \subseteq \# DA0$ 
    using as_subst_lit_def by auto
  then have  $\exists As0. As0 \cdot al \eta 0 = As \wedge \text{negs } (\text{mset } As0) \subseteq \# DA0$ 
     $\wedge (S\_M S M (D + \text{negs } (\text{mset } As)) \neq \{\#\}) \longrightarrow \text{negs } (\text{mset } As0) = S \text{ DA0}$ 
    using a by blast
}
moreover
{
  assume  $S\_M S M (D + \text{negs } (\text{mset } As)) = \text{negs } (\text{mset } As)$ 
  then have negs (mset As) =  $S \text{ DA0} \cdot \eta 0$ 
    using da  $\langle S \text{ DA0} \cdot \eta 0 = S\_M S M \text{ DA} \rangle$  by auto
  then have  $\exists As0. \text{negs } (\text{mset } As0) = S \text{ DA0} \wedge As0 \cdot al \eta 0 = As$ 
    using instance_list[of As S DA0  $\eta 0$ ] S.S_selects_neg_lits by auto
  then have  $\exists As0. As0 \cdot al \eta 0 = As \wedge \text{negs } (\text{mset } As0) \subseteq \# DA0$ 
     $\wedge (S\_M S M (D + \text{negs } (\text{mset } As)) \neq \{\#\}) \longrightarrow \text{negs } (\text{mset } As0) = S \text{ DA0}$ 
    using S.S_selects_subseteq by auto
}
ultimately have  $\exists As0. As0 \cdot al \eta 0 = As \wedge (\text{negs } (\text{mset } As0)) \subseteq \# DA0$ 
 $\wedge (S\_M S M (D + \text{negs } (\text{mset } As)) \neq \{\#\}) \longrightarrow \text{negs } (\text{mset } As0) = S \text{ DA0}$ 
using eligible_unfolding eligible.simps by auto


then obtain As0 where


$$\begin{aligned} As0\_p: As0 \cdot al \eta 0 &= As \wedge \text{negs } (\text{mset } As0) \subseteq \# DA0 \\ \wedge (S\_M S M (D + \text{negs } (\text{mset } As)) \neq \{\#\}) &\longrightarrow \text{negs } (\text{mset } As0) = S \text{ DA0} \end{aligned}$$

by blast


then have length As0 = n

using as_len by auto


note n = n this



have As0  $\cdot al \eta 0 = As$ 
using As0_p by auto



define D0 where


$$D0 = DA0 - \text{negs } (\text{mset } As0)$$


then have DA0 =  $D0 + \text{negs } (\text{mset } As0)$ 
using As0_p by auto



then have  $D0 \cdot \eta 0 = D$ 
using DA0_to_DA da As0_p by auto



have  $S\_M S M (D + \text{negs } (\text{mset } As)) \neq \{\#\} \implies \text{negs } (\text{mset } As0) = S \text{ DA0}$ 
using As0_p by blast



then show ?thesis
  using that  $\langle As0 \cdot al \eta 0 = As \rangle \langle D0 \cdot \eta 0 = D \rangle \langle DA0 = D0 + (\text{negs } (\text{mset } As0)) \rangle \langle \text{length } As0 = n \rangle$ 
by metis



qed


```

```

show ?thesis
using that[ $OF n(2,1) DA0\_in\_M DA0\_to\_DA SDA0\_to\_SMDA CAs0\_in\_M CAs0\_to\_CAs SCAs0\_to\_SMCAs$ 
   $\langle is\_ground\_subst \eta 0 \rangle \langle is\_ground\_subst\_list \eta s0 \rangle \langle As0 \cdot al \eta 0 = As \rangle$ 
   $\langle AAs0 \cdot aml \eta s0 = AAs \rangle$ 
   $\langle length As0 = n \rangle$ 
   $\langle D0 \cdot \eta 0 = D \rangle$ 
   $\langle DA0 = D0 + (negs (mset As0)) \rangle$ 
   $\langle S\_M S M (D + negs (mset As)) \neq \{\#} \implies negs (mset As0) = S DA0 \rangle$ 
   $\langle length Cs0 = n \rangle$ 
   $\langle Cs0 \cdot cl \eta s0 = Cs \rangle$ 
   $\langle \forall i < n. CAs0 ! i = Cs0 ! i + poss (AAs0 ! i) \rangle$ 
   $\langle length AAs0 = n \rangle]$ 
by auto
qed

lemma ord_resolve_rename_lifting:
assumes
  sel_stable:  $\bigwedge \varrho C. is\_renaming \varrho \implies S (C \cdot \varrho) = S C \cdot \varrho$  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  $\eta s \eta \eta 2 CAs0 DA0 AAs0 As0 E0 \tau$  where
  is_ground_subst  $\eta$ 
  is_ground_subst_list  $\eta s$ 
  is_ground_subst  $\eta 2$ 
  ord_resolve_rename S CAs0 DA0 AAs0 As0  $\tau$  E0
  CAs0  $\cdot cl \eta s = CAs DA0 \cdot \eta = DA E0 \cdot \eta 2 = E$ 
  {DA0}  $\cup$  set CAs0  $\subseteq$  M
  length CAs0 = length CAs
  length  $\eta s$  = length CAs
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_empty = this(9) and mgu = this(10) and eligible = this(11) and str_max = this(12) and
  sel_empty = this(13)

have sel_ren_list_inv:
   $\bigwedge \varrho Cs. length \varrho s = length Cs \implies is\_renaming\_list \varrho s = map S (Cs \cdot cl \varrho s) = map S Cs \cdot cl \varrho s$ 
  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 DA0  $\eta 0 CAs0 \eta s0 As0 AAs0 D0 Cs0$  where as0:
  length CAs0 = n
  length  $\eta s0 = n$ 
  DA0  $\in$  M
  DA0  $\cdot \eta 0 = DA$ 
  S DA0  $\cdot \eta 0 = S\_M S M DA$ 
   $\forall CA0 \in$  set CAs0. CA0  $\in$  M
  CAs0  $\cdot cl \eta s0 = CAs$ 
  map S CAs0  $\cdot cl \eta s0 = map (S\_M S M) CAs$ 
  is_ground_subst  $\eta 0$ 
  is_ground_subst_list  $\eta s0$ 
  As0  $\cdot al \eta 0 = As$ 
  AAs0  $\cdot aml \eta s0 = AAs$ 
  length As0 = n
  D0  $\cdot \eta 0 = D$ 
  DA0 = D0 + (negs (mset As0))

```

```

S_M S M (D + negs (mset As)) ≠ {#} ⇒ negs (mset As0) = S DA0
length Cs0 = n
Cs0 ..cl ηs0 = Cs
∀ i < n. CAs0 ! i = Cs0 ! i + poss (AAs0 ! i)
length AAs0 = n
using ord_resolve_obtain_clauses[of S M CAs DA, OF res_e select grounding n(2) ⟨DA = D + negs (mset As)⟩
⟨∀ i < n. CAs ! i = Cs ! i + poss (AAs ! i)⟩ ⟨length Cs = n⟩ ⟨length AAs = n⟩, of thesis] by blast

note n = ⟨length CAs0 = n⟩ ⟨length ηs0 = n⟩ ⟨length As0 = n⟩ ⟨length AAs0 = n⟩ ⟨length Cs0 = n⟩ n

have length (renamings_apart (DA0 # CAs0)) = Suc n
using n renamings_apart_length by auto

note n = this n

define ρ where
ρ = hd (renamings_apart (DA0 # CAs0))
define ρs where
ρs = tl (renamings_apart (DA0 # CAs0))
define DA0' where
DA0' = DA0 · ρ
define D0' where
D0' = D0 · ρ
define As0' where
As0' = As0 · al ρ
define CAs0' where
CAs0' = CAs0 ..cl ρs
define Cs0' where
Cs0' = Cs0 ..cl ρs
define AAs0' where
AAs0' = AAs0 ..aml ρs
define η0' where
η0' = inv_renaming ρ ⊕ η0
define ηs0' where
ηs0' = map inv_renaming ρs ⊕s ηs0

have renames_DA0: is_renaming ρ
using renamings_apart_length renamings_apart_renaming unfolding ρ_def
by (metis length_greater_0_conv list.exhaust_sel list.set_intro(1) list.simps(3))

have renames_CAs0: is_renaming_list ρs
using renamings_apart_length renamings_apart_renaming unfolding ρs_def
by (metis is_renaming_list_def length_greater_0_conv list.set_sel(2) list.simps(3))

have length ρs = n
unfolding ρs_def using n by auto
note n = n ⟨length ρs = n⟩
have length As0' = n
unfolding As0'_def using n by auto
have length CAs0' = n
using as0(1) n unfolding CAs0'_def by auto
have length Cs0' = n
unfolding Cs0'_def using n by auto
have length AAs0' = n
unfolding AAs0'_def using n by auto
have length ηs0' = n
using as0(2) n unfolding ηs0'_def by auto
note n = ⟨length CAs0' = n⟩ ⟨length ηs0' = n⟩ ⟨length As0' = n⟩ ⟨length AAs0' = n⟩ ⟨length Cs0' = n⟩ n

have DA0'_DA: DA0' · η0' = DA
using as0(4) unfolding η0'_def DA0'_def using renames_DA0 by simp
have D0'_D: D0' · η0' = D
using as0(14) unfolding η0'_def D0'_def using renames_DA0 by simp

```

```

have As0'_As: As0' · al η0' = As
  using as0(11) unfolding η0'_def As0'_def using renames_DA0 by auto
have S DA0' · η0' = S_M S M DA
  using as0(5) unfolding η0'_def DA0'_def using renames_DA0 sel_stable by auto
have CAs0'_CAs: CAs0' .. cl ηs0' = CAs
  using as0(7) unfolding CAs0'_def ηs0'_def using renames_CAs0 n by auto
have Cs0'_Cs: Cs0' .. cl ηs0' = Cs
  using as0(18) unfolding Cs0'_def ηs0'_def using renames_CAs0 n by auto
have AAs0'_AAs: AAs0' .. aml ηs0' = AAs
  using as0(12) unfolding ηs0'_def AAs0'_def using renames_CAs0 using n by auto
have map S CAs0' .. cl ηs0' = map (S_M S M) CAs
  unfolding CAs0'_def ηs0'_def using as0(8) n renames_CAs0 sel_ren_list_inv by auto

have DA0'_split: DA0' = D0' + negs (mset As0')
  using as0(15) DA0'_def D0'_def As0'_def by auto
then have D0'_subset_DA0': D0' ⊆# DA0'
  by auto
from DA0'_split have negs_As0'_subset_DA0': negs (mset As0') ⊆# DA0'
  by auto

have CAs0'_split: ∀ i < n. CAs0' ! i = Cs0' ! i + poss (AAs0' ! i)
  using as0(19) CAs0'_def Cs0'_def AAs0'_def n by auto
then have ∀ i < n. Cs0' ! i ⊆# CAs0' ! i
  by auto
from CAs0'_split have poss_AAs0'_subset_CAs0': ∀ i < n. poss (AAs0' ! i) ⊆# CAs0' ! i
  by auto
then have AAs0'_in_atms_of_CAs0': ∀ i < n. ∀ A ∈ # AAs0' ! i. A ∈ atms_of (CAs0' ! i)
  by (auto simp add: atm_iff_pos_or_neg_lit)

have as0':
  S_M S M (D + negs (mset As)) ≠ {#} ⇒ negs (mset As0') = S DA0'
proof -
  assume a: S_M S M (D + negs (mset As)) ≠ {#}
  then have negs (mset As0) · ρ = S DA0 · ρ
    using as0(16) unfolding ρ_def by metis
  then show negs (mset As0') = S DA0'
    using As0'_def DA0'_def using sel_stable[of ρ DA0] renames_DA0 by auto
qed

have vd: var_disjoint (DA0' # CAs0')
  unfolding DA0'_def CAs0'_def using renamings_apart_var_disjoint
  unfolding ρ_def ρs_def
  by (metis length_greater_0_conv list.exhaust_sel n(6) subst_cls_lists_Cons zero_less_Suc)

— Introduce ground substitution
from vd DA0'_DA CAs0'_CAs have ∃ η. ∀ i < Suc n. ∀ S. S ⊆# (DA0' # CAs0') ! i → S · (η0' # ηs0') ! i = S · η
  unfolding var_disjoint_def using n by auto
then obtain η where η_p: ∀ i < Suc n. ∀ S. S ⊆# (DA0' # CAs0') ! i → S · (η0' # ηs0') ! i = S · η
  by auto
have η_p_lit: ∀ i < Suc n. ∀ L. L ∈# (DA0' # CAs0') ! i → L · l (η0' # ηs0') ! i = L · l η
  proof (rule, rule, rule, rule)
    fix i :: nat and L :: 'a literal
    assume a:
      i < Suc n
      L ∈# (DA0' # CAs0') ! i
    then have ∀ S. S ⊆# (DA0' # CAs0') ! i → S · (η0' # ηs0') ! i = S · η
      using η_p by auto
    then have {# L #} · (η0' # ηs0') ! i = {# L #} · η
      using a by (meson single_subset_iff)
    then show L · l (η0' # ηs0') ! i = L · l η by auto
  qed
have η_p_atm: ∀ i < Suc n. ∀ A. A ∈ atms_of ((DA0' # CAs0') ! i) → A · a (η0' # ηs0') ! i = A · a η

```

```

proof (rule, rule, rule, rule)
  fix i :: nat and A :: 'a
  assume a:
    i < Suc n
    A ∈ atms_of ((DA0' # CAs0') ! i)
    then obtain L where L_p: atm_of L = A ∧ L ∈# (DA0' # CAs0') ! i
      unfolding atms_of_def by auto
    then have L · l (η0' # ηs0') ! i = L · l η
      using η_p_lit a by auto
    then show A · a (η0' # ηs0') ! i = A · a η
      using L_p unfolding subst_lit_def by (cases L) auto
  qed

  have DA0'_DA: DA0' · η = DA
    using DA0'_DA η_p by auto
  have D0' · η = D using η_p D0'_D n D0'_subset_DA0' by auto
  have As0' · al η = As
  proof (rule nth_equalityI)
    show length (As0' · al η) = length As
      using n by auto
  next
    fix i
    show i < length (As0' · al η) ⇒ (As0' · al η) ! i = As ! i
    proof –
      assume a: i < length (As0' · al η)
      have A_eq: ∀ A. A ∈ atms_of DA0' → A · a η0' = A · a η
        using η_p_atm n by force
      have As0' ! i ∈ atms_of DA0'
        using negs_As0'_subset_DA0' unfolding atms_of_def
        using a n by force
      then have As0' ! i · a η0' = As0' ! i · a η
        using A_eq by simp
      then show (As0' · al η) ! i = As ! i
        using As0'_As <length As0' = n> a by auto
    qed
  qed

  interpret selection
    by (rule select)

  have S DA0' · η = S_M S M DA
    using <S DA0' · η0' = S_M S M DA, η_p S.S_selects_subseteq by auto

  from η_p have η_p_CAs0': ∀ i < n. (CAs0' ! i) · (ηs0' ! i) = (CAs0' ! i) · η
    using n by auto
  then have CAs0' .. cl ηs0' = CAs0' .. cl η
    using n by (auto intro: nth_equalityI)
  then have CAs0'_η_fo_CAs: CAs0' · cl η = CAs
    using CAs0'_CAs η_p n by auto

  from η_p have ∀ i < n. S (CAs0' ! i) · ηs0' ! i = S (CAs0' ! i) · η
    using S.S_selects_subseteq n by auto
  then have map S CAs0' .. cl ηs0' = map S CAs0' · cl η
    using n by (auto intro: nth_equalityI)
  then have SCAs0'_η_fo_SMCAs: map S CAs0' · cl η = map (S_M S M) CAs
    using <map S CAs0' .. cl ηs0' = map (S_M S M) CAs> by auto

  have Cs0' · cl η = Cs
  proof (rule nth_equalityI)
    show length (Cs0' · cl η) = length Cs
      using n by auto
  next
    fix i

```

```

show i < length (Cs0' · cl η)  $\implies$  (Cs0' · cl η) ! i = Cs ! i
proof -
  assume i < length (Cs0' · cl η)
  then have a: i < n
    using n by force
  have (Cs0' · cl ηs0') ! i = Cs ! i
    using Cs0'_Cs a n by force
  moreover
    have η_p_CAs0':  $\forall S. S \subseteq \# CAs0' ! i \rightarrow S \cdot \eta s0' ! i = S \cdot \eta$ 
      using η_p a by force
    have Cs0' ! i · ηs0' ! i = (Cs0' · cl η) ! i
      using η_p_CAs0'  $\forall i < n. Cs0' ! i \subseteq \# CAs0' ! i$  a n by force
    then have (Cs0' .. cl ηs0') ! i = (Cs0' · cl η) ! i
      using a n by force
    ultimately show (Cs0' · cl η) ! i = Cs ! i
      by auto
  qed
qed

have AAs0'_AAs: AAs0' · aml η = AAs
proof (rule nth_equalityI)
  show length (AAs0' · aml η) = length AAs
    using n by auto
next
  fix i
  show i < length (AAs0' · aml η)  $\implies$  (AAs0' · aml η) ! i = AAs ! i
  proof -
    assume a: i < length (AAs0' · aml η)
    then have i < n
      using n by force
    then have  $\forall A. A \in \text{atms\_of}((DA0' \# CAs0') ! Suc i) \rightarrow A \cdot a (\eta0' \# \eta s0') ! Suc i = A \cdot a \eta$ 
      using η_p_atm n by force
    then have A_eq:  $\forall A. A \in \text{atms\_of}(CAs0' ! i) \rightarrow A \cdot a \eta s0' ! i = A \cdot a \eta$ 
      by auto
    have AAs_CAs0':  $\forall A \in \# AAs0' ! i. A \in \text{atms\_of}(CAs0' ! i)$ 
      using AAs0'_in_atms_of_CAs0' unfolding atms_of_def
      using a n by force
    then have AAs0' ! i · am ηs0' ! i = AAs0' ! i · am η
      unfolding subst_atm_mset_def using A_eq unfolding subst_atm_mset_def by auto
    then show (AAs0' · aml η) ! i = AAs ! i
      using AAs0'_AAs  $\langle \text{length } AAs0' = n \rangle \langle \text{length } \eta s0' = n \rangle a$  by auto
  qed
qed

— Obtain MGU and substitution
obtain τ φ where τφ:
  Some τ = mgu (set_mset ‘ set (map2 add_mset As0' AAs0'))
  τ ⊕ φ = η ⊕ σ
proof -
  have uu: is_unifiers σ (set_mset ‘ set (map2 add_mset (As0' · al η) (AAs0' · aml η)))
    using mgu mgu_sound is_mgu_def unfolding ⟨AAs0' · aml η = AAs⟩ using ⟨As0' · al η = As⟩ by auto
  have ησuni: is_unifiers (η ⊕ σ) (set_mset ‘ set (map2 add_mset As0' AAs0'))
  proof -
    have set_mset ‘ set (map2 add_mset As0' AAs0' · aml η) =
      set_mset ‘ set (map2 add_mset As0' AAs0') · 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 As0' AAs0') · 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

```

```

 $\tau_p$ : Some  $\tau = \text{mgu}(\text{set\_mset} ` \text{set}(\text{map2} \text{add\_mset} As0' AAs0'))$ 
using mgu_complete
by (metis (mono_tags, opaque_lifting) List.finite_set finite_imageI finite_set_mset image_iff)
moreover then obtain  $\varphi$  where  $\varphi_p: \tau \odot \varphi = \eta \odot \sigma$ 
by (metis (mono_tags, opaque_lifting) finite_set_eta_uni 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 eligible0': eligible  $S \tau As0' (D0' + \text{negs}(\text{mset} As0'))$ 
proof —
  have  $S\_M S M (D + \text{negs}(\text{mset} As)) = \text{negs}(\text{mset} As) \vee S\_M S M (D + \text{negs}(\text{mset} As)) = \{\#\} \wedge$ 
     $\text{length} As = 1 \wedge \text{maximal\_wrt}(As ! 0 \cdot a \sigma) ((D + \text{negs}(\text{mset} As)) \cdot \sigma)$ 
    using eligible unfolding eligible.simps by auto
  then show ?thesis
proof
  assume  $S\_M S M (D + \text{negs}(\text{mset} As)) = \text{negs}(\text{mset} As)$ 
  then have  $S\_M S M (D + \text{negs}(\text{mset} As)) \neq \{\#\}$ 
  using n by force
  then have  $S (D0' + \text{negs}(\text{mset} As0')) = \text{negs}(\text{mset} As0')$ 
  using as0' DA0'_split by auto
  then show ?thesis
  unfolding eligible.simps[simplified] by auto
next
  assume asm:  $S\_M S M (D + \text{negs}(\text{mset} As)) = \{\#\} \wedge \text{length} As = 1 \wedge$ 
     $\text{maximal\_wrt}(As ! 0 \cdot a \sigma) ((D + \text{negs}(\text{mset} As)) \cdot \sigma)$ 
  then have  $S (D0' + \text{negs}(\text{mset} As0')) = \{\#\}$ 
  using <D0' · η = D>[symmetric] <As0' · al η = As>[symmetric] <S (DA0') · η = S\_M S M (DA)>
    da DA0'_split subst_cls_empty_if by metis
  moreover from asm have l:  $\text{length} As0' = 1$ 
  using <As0' · al η = As> by auto
  moreover from asm have maximal_wrt(As0' ! 0 · a (τ ⊕ φ)) ((D0' + negs(mset As0')) · (τ ⊕ φ))
  using <As0' · al η = As> <D0' · η = D> using l τφ by auto
  then have maximal_wrt(As0' ! 0 · a τ · a φ) ((D0' + negs(mset As0')) · τ · φ)
  by auto
  then have maximal_wrt(As0' ! 0 · a τ) ((D0' + negs(mset As0')) · τ)
  using maximal_wrt_subst by blast
  ultimately show ?thesis
  unfolding eligible.simps[simplified] by auto
qed

```

— Lifting maximality

```

have maximality:  $\forall i < n. \text{strictly\_maximal\_wrt}(As0' ! i \cdot a \tau) (Cs0' ! i \cdot \tau)$ 

```

```

proof —
  from str_max have  $\forall i < n. \text{strictly\_maximal\_wrt}((As0' · al η) ! i · a σ) ((Cs0' · cl η) ! i · σ)$ 
  using <As0' · al η = As> <Cs0' · cl η = Cs> by simp
  then have  $\forall i < n. \text{strictly\_maximal\_wrt}(As0' ! i \cdot a (\tau \odot \varphi)) (Cs0' ! i \cdot (\tau \odot \varphi))$ 
  using n τφ by simp
  then have  $\forall i < n. \text{strictly\_maximal\_wrt}(As0' ! i \cdot a \tau \cdot a \varphi) (Cs0' ! i \cdot \tau \cdot \varphi)$ 
  by auto
  then show  $\forall i < n. \text{strictly\_maximal\_wrt}(As0' ! i \cdot a \tau) (Cs0' ! i \cdot \tau)$ 
  using strictly_maximal_wrt_subst τφ by blast
qed

```

— Lifting nothing being selected

```

have nothing_selected:  $\forall i < n. S (CAs0' ! i) = \{\#\}$ 

```

```

proof —
  have  $\forall i < n. (\text{map} S CAs0' · cl η) ! i = \text{map}(S\_M S M) CAs ! i$ 
  by (simp add: <map S CAs0' · cl η = map(S\_M S M) CAs>)

```

```

then have  $\forall i < n. S(CAs0' ! i) \cdot \eta = S\_M S M(CAs ! i)$ 
  using  $n$  by auto
then have  $\forall i < n. S(CAs0' ! i) \cdot \eta = \{\#\}$ 
  using  $sel\_empt \langle \forall i < n. S(CAs0' ! i) \cdot \eta = S\_M S M(CAs ! i) \rangle$  by auto
then show  $\forall i < n. S(CAs0' ! i) = \{\#\}$ 
  using  $subst\_cls\_empty\_iff$  by blast
qed

— Lifting AAs0's non-emptiness
have  $\forall i < n. AAs0' ! i \neq \{\#\}$ 
  using  $n aas\_not\_empt \langle AAs0' \cdot aml \eta = AAs \rangle$  by auto

— Resolve the lifted clauses
define  $E0'$  where

$$E0' = ((\sum_{\#} (mset Cs0')) + D0') \cdot \tau$$


have  $res\_e0': ord\_resolve S CAs0' DA0' AAs0' As0' \tau E0'$ 
  using  $ord\_resolve.intros[of CAs0' n Cs0' AAs0' As0' \tau S D0',$ 
     $OF\_--- \langle \forall i < n. AAs0' ! i \neq \{\#\} \rangle \tau \varphi(1) eligible0'$ 
     $\langle \forall i < n. strictly\_maximal\_wrt (As0' ! i \cdot a \tau) (Cs0' ! i \cdot \tau) \rangle \langle \forall i < n. S(CAs0' ! i) = \{\#\} \rangle]$ 
  unfolding  $E0'\_def$  using  $DA0'\_split n \langle \forall i < n. CAs0' ! i = Cs0' ! i + poss(AAs0' ! i) \rangle$  by blast

— Prove resolvent instantiates to ground resolvent
have  $e0'\varphi e: E0' \cdot \varphi = E$ 
proof —
  have  $E0' \cdot \varphi = ((\sum_{\#} (mset Cs0')) + D0') \cdot (\tau \odot \varphi)$ 
    unfolding  $E0'\_def$  by auto
  also have  $\dots = (\sum_{\#} (mset Cs0')) + D0' \cdot (\eta \odot \sigma)$ 
    using  $\tau \varphi$  by auto
  also have  $\dots = (\sum_{\#} (mset Cs)) + D \cdot \sigma$ 
    using  $\langle Cs0' \cdot cl \eta = Cs \rangle \langle D0' \cdot \eta = D \rangle$  by auto
  also have  $\dots = E$ 
    using  $e$  by auto
  finally show  $e0'\varphi e: E0' \cdot \varphi = E$ 
  .
qed

— Replace  $\varphi$  with a true ground substitution
obtain  $\eta2$  where

$$\text{ground\_}\eta2: is\_ground\_subst \eta2 E0' \cdot \eta2 = E$$

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  $that e0'\varphi e make\_ground\_subst$  by auto
qed

have  $\langle length CAs0 = length CAs \rangle$ 
  using  $n$  by simp

have  $\langle length \eta s0 = length CAs \rangle$ 
  using  $n$  by simp

— Wrap up the proof
have  $ord\_resolve S (CAs0 \cdot cl \varrho_s) (DA0 \cdot \varrho) (AAs0 \cdot aml \varrho_s) (As0 \cdot al \varrho) \tau E0'$ 
  using  $res\_e0' As0'\_def \varrho\_def AAs0'\_def \varrho_s\_def DA0'\_def \varrho\_def CAs0'\_def \varrho_s\_def$  by simp
moreover have  $\forall i < n. poss(AAs0 ! i) \subseteq \# CAs0 ! i$ 
  using  $as0(19)$  by auto
moreover have  $negs(mset As0) \subseteq \# DA0$ 
  using  $local.as0(15)$  by auto
ultimately have  $ord\_resolve\_rename S CAs0 DA0 AAs0 As0 \tau E0'$ 

```

```

using ord_resolve_rename[of CAs0 n AAs0 As0 DA0 ρ ρs S τ E0'] ρ_def ρs_def n by auto
then show thesis
  using that[of η0 ηs0 η2 CAs0 DA0] ⟨is_ground_subst η0⟩ ⟨is_ground_subst_list ηs0⟩
    ⟨is_ground_subst η2⟩ ⟨CAs0 ..cl ηs0 = CAs⟩ ⟨DA0 · η0 = DA⟩ ⟨E0' · η2 = E⟩ ⟨DA0 ∈ M,
    ∀ CA ∈ set CAs0. CA ∈ M⟩ ⟨length CAs0 = length CAs⟩ ⟨length ηs0 = length CAs⟩
  by blast
qed

lemma ground_ord_resolve_ground:
assumes
  select: selection S and
  CAs_p: ground_resolution_with_selection.ord_resolve S CAs DA AAs As E and
  ground_cas: is_ground_cls_list CAs and
  ground_da: is_ground_cls DA
shows is_ground_cls E
proof -
  have a1: atms_of E ⊆ (∪ CA ∈ set CAs. atms_of CA) ∪ atms_of DA
    using ground_resolution_with_selection.ord_resolve_atms_of_concl_subset[OF _ CAs_p]
    ground_resolution_with_selection.intro[OF select] by blast
  {
    fix L :: 'a literal
    assume L ∈# E
    then have atm_of L ∈ atms_of E
      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

lemma ground_ord_resolve_imp_ord_resolve:
assumes
  ground_da: is_ground_cls DA and
  ground_cas: is_ground_cls_list CAs and
  gr: ground_resolution_with_selection S_G and
  gr_res: ⟨ground_resolution_with_selection.ord_resolve S_G CAs DA AAs As E⟩
shows ⟨∃ σ. ord_resolve S_G CAs DA AAs As σ E⟩
proof (cases rule: ground_resolution_with_selection.ord_resolve.cases[OF gr gr_res])
  case (1 CAs n Cs AAs As D)
  note cas = this(1) and da = this(2) and aas = this(3) and as = this(4) and e = this(5) and
  cas_len = this(6) and cs_len = this(7) and aas_len = this(8) and as_len = this(9) and
  nz = this(10) and casi = this(11) and aas_not_empty = this(12) and as_aas = this(13) and
  eligibility = this(14) and str_max = this(15) and sel_empty = this(16)

  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
    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  $\sigma$  where
 $\sigma_p$ : Some  $\sigma = \text{mgu}(\text{set\_mset} ` \text{set}(\text{map2 add\_mset As AAs}))$ 
using  $\text{mgu\_complete}$  by metis

have ground_elig: ground_resolution_with_selection.eligible S_G As (D + negs (mset As))
  using eligibility by simp
have ground_cs:  $\forall i < n. \text{is\_ground\_cls} (Cs ! i)$ 
  using cas cas_len cs_len casi ground_cas nth_mem unfolding is_ground_cls_list_def by force
have ground_set_as: is_ground_atms (set As)
  using da ground_da by (metis atms_of_negs is_ground_cls_is_ground_atms_atms_of
    is_ground_cls_union set_mset_mset)
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 da by simp

from as_len nz have atms:
  atms_of D  $\cup$  set As  $\neq \{\}$ 
  finite (atms_of D  $\cup$  set As)
  by auto
then have Max (atms_of D  $\cup$  set As)  $\in$  atms_of D  $\cup$  set As
  using Max_in by metis
then have is_ground_Max: is_ground_atm (Max (atms_of D  $\cup$  set As))
  using ground_d ground_mset_as is_ground_cls_imp_is_ground_atm
  unfolding is_ground_atm_mset_def by auto

have maximal_wrt (Max (atms_of D  $\cup$  set As)) (D + negs (mset As))
  unfolding maximal_wrt_def
  by clarsimp (metis atms Max_less_iff UnCI ground_d ground_set_as infinite_growing
    is_ground_Max is_ground_atms_def is_ground_cls_imp_is_ground_atm less_atm_ground)
moreover have
  Max (atms_of D  $\cup$  set As)  $\cdot a \sigma = \text{Max} (\text{atms\_of } D \cup \text{set } As)$  and
   $D \cdot \sigma + \text{negs} (\text{mset } As \cdot am \sigma) = D + \text{negs} (\text{mset } As)$ 
  using ground_elig is_ground_Max ground_mset_as ground_d by auto
ultimately have fo_elig: eligible S_G  $\sigma$  As (D + negs (mset As))
  using ground_elig unfolding ground_resolution_with_selection.eligible.simps[OF gr]
  ground_resolution_with_selection.maximal_wrt_def[OF gr] eligible.simps
  by auto

have  $\forall i < n. \text{strictly\_maximal\_wrt} (\text{As} ! i) (Cs ! i)$ 
  using str_max[unfolded ground_resolution_with_selection.strictly_maximal_wrt_def[OF gr]]
  ground_as[unfolded is_ground_atm_list_def] ground_cs as_len less_atm_ground
  unfolding strictly_maximal_wrt_def byclarsimp (fastforce simp: is_ground_cls_as_atms)+
then have ll:  $\forall i < n. \text{strictly\_maximal\_wrt} (\text{As} ! i \cdot a \sigma) (Cs ! i \cdot \sigma)$ 
  by (simp add: ground_as ground_cs as_len)

have ground_e: is_ground_cls E
  using ground_d ground_cs cs_len unfolding e is_ground_cls_def
  by simp (metis in_mset_sum_list2 in_set_conv_nth)

show ?thesis
  using cas da aas as e ground_e ord_resolve.intros[OF cas_len cs_len aas_len as_len nz casi
    aas_not_empty  $\sigma_p$  fo_elig ll sel_empty]
  by auto
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 is_least :: (nat ⇒ bool) ⇒ nat ⇒ bool where
  is_least P n ↔ P n ∧ (∀ n' < n. ¬ P n')
```

```
lemma least_exists: P n ⇒ ∃ n. is_least P n
  using exists_least_iff unfolding 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 state = 'a clause set × 'a clause set × 'a clause set
```

```
locale FO_resolution_prover =
  FO_resolution subst_atm id_subst comp_subst renamings_apart atm_of_atms mgu less_atm +
  selection S
for
  S :: ('a :: wellorder) clause ⇒ 'a clause and
  subst_atm :: 'a ⇒ 's ⇒ 'a and
  id_subst :: 's and
  comp_subst :: 's ⇒ 's ⇒ 's and
  renamings_apart :: 'a clause list ⇒ 's list and
  atm_of_atms :: 'a list ⇒ 'a and
  mgu :: 'a set set ⇒ 's option and
  less_atm :: 'a ⇒ 'a ⇒ bool +
assumes
  sel_stable: ∀ ρ C. is_renaming ρ ⇒ S (C · ρ) = S C · ρ
begin
```

```
fun N_of_state :: 'a state ⇒ 'a clause set where
  N_of_state (N, P, Q) = N
```

```
fun P_of_state :: 'a state ⇒ 'a clause set where
  P_of_state (N, P, Q) = P
```

O denotes relation composition in Isabelle, so the formalization uses Q instead.

```
fun Q_of_state :: 'a state ⇒ 'a clause set where
  Q_of_state (N, P, Q) = Q
```

```
abbreviation clss_of_state :: 'a state ⇒ 'a clause set where
  clss_of_state St ≡ N_of_state St ∪ P_of_state St ∪ Q_of_state St
```

```
abbreviation grounding_of_state :: 'a state ⇒ 'a clause set where
  grounding_of_state St ≡ grounding_of_clss (clss_of_state St)
```

```
interpretation ord_FO_resolution: inference_system ord_FO_Γ S .
```

The following inductive predicate formalizes the resolution prover in Figure 5.

```
inductive RP :: 'a state ⇒ 'a state ⇒ bool (infix ↔ 50) where
  tautology_deletion: Neg A ∈# C ⇒ Pos A ∈# C ⇒ (N ∪ {C}, P, Q) ↔ (N, P, Q)
  | forward_subsumption: D ∈ P ∪ Q ⇒ subsumes D C ⇒ (N ∪ {C}, P, Q) ↔ (N, P, Q)
  | backward_subsumption_P: D ∈ N ⇒ strictly_subsumes D C ⇒ (N, P ∪ {C}, Q) ↔ (N, P, Q)
  | backward_subsumption_Q: D ∈ N ⇒ strictly_subsumes D C ⇒ (N, P, Q ∪ {C}) ↔ (N, P, Q)
  | forward_reduction: D + {#L'#} ∈ P ∪ Q ⇒ - L = L' · l σ ⇒ D · σ ⊆# C ⇒
    (N ∪ {C + {#L#}}, P, Q) ↔ (N ∪ {C}, P, Q)
  | backward_reduction_P: D + {#L'#} ∈ N ⇒ - L = L' · l σ ⇒ D · σ ⊆# C ⇒
    (N, P ∪ {C + {#L#}}, Q) ↔ (N, P ∪ {C}, Q)
```

```

| backward_reduction_Q:  $D + \{\#L'\#\} \in N \implies - L = L' \cdot l \sigma \implies D \cdot \sigma \subseteq \# C \implies$ 
   $(N, P, Q \cup \{C + \{\#L'\#\}\}) \rightsquigarrow (N, P \cup \{C\}, Q)$ 
| clause_processing:  $(N \cup \{C\}, P, Q) \rightsquigarrow (N, P \cup \{C\}, Q)$ 
| inference_computation:  $N = \text{concls\_of } (\text{ord\_FO\_resolution.inferences\_between } Q C) \implies$ 
   $(\{\}, P \cup \{C\}, Q) \rightsquigarrow (N, P, Q \cup \{C\})$ 

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:  $N_{\text{of\_state}}(\text{Liminf\_state Sts}') = N_{\text{of\_state}}(\text{Liminf\_state Sts})$  and
  P_of_Liminf_state_fin:  $P_{\text{of\_state}}(\text{Liminf\_state Sts}') = P_{\text{of\_state}}(\text{Liminf\_state Sts})$  and
  Q_of_Liminf_state_fin:  $Q_{\text{of\_state}}(\text{Liminf\_state Sts}') = Q_{\text{of\_state}}(\text{Liminf\_state Sts})$ 
  using Sts by (simp_all add: Liminf_state_def lfinite_Liminf_llist_llast_lmap)

lemma Liminf_state_fin:  $\text{Liminf\_state Sts}' = \text{Liminf\_state Sts}$ 
  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:  $N_{\text{of\_state}}(\text{Liminf\_state Sts}') \subseteq N_{\text{of\_state}}(\text{Liminf\_state Sts})$  and
  P_of_Liminf_state_inf:  $P_{\text{of\_state}}(\text{Liminf\_state Sts}') \subseteq P_{\text{of\_state}}(\text{Liminf\_state Sts})$  and
  Q_of_Liminf_state_inf:  $Q_{\text{of\_state}}(\text{Liminf\_state Sts}') \subseteq Q_{\text{of\_state}}(\text{Liminf\_state Sts})$ 
  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)
  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  $\longleftrightarrow$   $N_{\text{of\_state}}(\text{Liminf\_state Sts}) = \{\} \wedge P_{\text{of\_state}}(\text{Liminf\_state Sts}) = \{\}$ 

```

The following formalizes Lemma 4.10.

```

context
  fixes Sts :: 'a state llist
begin

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. ( ∀ I. I ⊨ m CC → I ⊨ D → I ⊨ E) }

```

We prove that we indeed defined an extension.

```

lemma gd_ord_Γ_ngd_ord_Γ: gr.ord_Γ ⊆ 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 ⇒ 'a inference set where
  sr_ext_Ri N = sr.Ri N ∪ (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 $C \in sr.Rf(\text{grounding_of_cls } D)$

proof –

from sub have $\forall I. I \models D \cdot \sigma \rightarrow I \models C$

unfolding true_cls_def by blast

moreover have $C > D \cdot \sigma$

using sub by (simp add: subset_imp_less_mset)

moreover have $D \cdot \sigma \in \text{grounding_of_cls } D$

using ground_σ by (metis (mono_tags) mem_Collect_eq substitution_ops.grounding_of_cls_def)

ultimately have set_mset {#D · σ#} ⊆ grounding_of_cls D

($\forall I. I \models m \{#D · \sigma#\} \rightarrow I \models C$)

($\forall D'. D' \in \{#D · \sigma#\} \rightarrow D' < C$)

by auto

then show ?thesis

using sr.Rf_def by blast

qed

```

lemma strict_subset_subsumption_redundant_clss:

```

assumes

$D \cdot \sigma \subset\# C$ and

is_ground_subst σ and

```

 $D \in CC$ 
shows  $C \in sr.Rf(\text{grounding\_of\_cls } CC)$ 
using assms
proof –
  have  $C \in sr.Rf(\text{grounding\_of\_cls } D)$ 
    using strict_subset_subsumption_redundant_clause assms by auto
  then show ?thesis
    using assms unfolding grounding_of_cls_def
    by (metis (no_types) sr.Rf_mono sup_ge1 SUP_absorb contra_subsetD)
qed

lemma strict_subset_subsumption_grounding_redundant_cls:
assumes
   $D\sigma_{\text{subset\_}}C: D \cdot \sigma \subset\# C$  and
   $D_{\text{in\_}}St: D \in CC$ 
shows  $\text{grounding\_of\_cls } C \subseteq sr.Rf(\text{grounding\_of\_cls } CC)$ 
proof
  fix  $C\mu$ 
  assume  $C\mu \in \text{grounding\_of\_cls } C$ 
  then obtain  $\mu$  where
     $\mu_p: C\mu = C \cdot \mu \wedge \text{is\_ground\_subst } \mu$ 
    unfolding grounding_of_cls_def by auto
  have  $D\sigma\mu C\mu: D \cdot \sigma \cdot \mu \subset\# C \cdot \mu$ 
    using  $D\sigma_{\text{subset\_}}C \text{ subst\_subset\_mono}$  by auto
  then show  $C\mu \in sr.Rf(\text{grounding\_of\_cls } CC)$ 
    using  $\mu_p$  strict_subset_subsumption_redundant_cls[ $\text{of } D \sigma \odot \mu C \cdot \mu$ ]  $D_{\text{in\_}}St$  by auto
qed

lemma derive_if_remove_subsumed:
assumes
   $D \in \text{cls\_of\_state } St$  and
   $\text{subsumes } D C$ 
shows  $sr\_ext.\text{derive}(\text{grounding\_of\_state } St \cup \text{grounding\_of\_cls } C) = \text{grounding\_of\_state } St$ 
proof –
  from assms obtain  $\sigma$  where
     $D \cdot \sigma = C \vee D \cdot \sigma \subset\# C$ 
    by (auto simp: subsumes_def subset_mset_def)
  then have  $D \cdot \sigma = C \vee D \cdot \sigma \subset\# C$ 
    by (simp add: subset_mset_def)
  then show ?thesis
proof
  assume  $D \cdot \sigma = C$ 
  then have  $\text{grounding\_of\_cls } C \subseteq \text{grounding\_of\_cls } D$ 
    using subst_cls_eq_grounding_of_cls_subset_eq
    by (auto dest: sym)
  then have  $(\text{grounding\_of\_state } St \cup \text{grounding\_of\_cls } C) = \text{grounding\_of\_state } St$ 
    using assms unfolding grounding_of_cls_def by auto
  then show ?thesis
    by (auto intro: sr_ext.derive.intros)
next
  assume  $a: D \cdot \sigma \subset\# C$ 
  then have  $\text{grounding\_of\_cls } C \subseteq sr.Rf(\text{grounding\_of\_state } St)$ 
    using strict_subset_subsumption_grounding_redundant_cls assms by auto
  then show ?thesis
    unfolding grounding_of_cls_def by (force intro: sr_ext.derive.intros)
qed
qed

lemma reduction_in_concls_of:
assumes
   $C\mu \in \text{grounding\_of\_cls } C$  and
   $D + \{\#L'\#\} \in CC$  and
   $- L = L' \cdot l \sigma$  and

```

```

 $D \cdot \sigma \subseteq \# C$ 
shows  $C\mu \in \text{concls\_of}(\text{sr\_ext.inferences\_from}(\text{grounding\_of\_clss}(CC \cup \{C + \{\#L\#\}\})))$ 
proof -
  from assms
  obtain  $\mu$  where
     $\mu_p: C\mu = C \cdot \mu \wedge \text{is\_ground\_subst } \mu$ 
    unfolding  $\text{grounding\_of\_cls\_def}$  by auto

  define  $\gamma$  where
     $\gamma = \text{Infer} \{ \#(C + \{\#L\#\}) \cdot \mu \# \} ((D + \{\#L'\#\}) \cdot \sigma \cdot \mu) (C \cdot \mu)$ 

  have  $(D + \{\#L'\#\}) \cdot \sigma \cdot \mu \in \text{grounding\_of\_clss}(CC \cup \{C + \{\#L\#\}\})$ 
    unfolding  $\text{grounding\_of\_clss\_def}$   $\text{grounding\_of\_cls\_def}$ 
    by (rule UN_I[of  $D + \{\#L'\#\}$ ], use assms(2) in simp,
      metis (mono_tags, lifting)  $\mu_p$  is_ground_comp_subst mem_Collect_eq subst_cls_comp_subst)
  moreover have  $(C + \{\#L\#\}) \cdot \mu \in \text{grounding\_of\_clss}(CC \cup \{C + \{\#L\#\}\})$ 
    using  $\mu_p$  unfolding  $\text{grounding\_of\_clss\_def}$   $\text{grounding\_of\_cls\_def}$  by auto
  moreover have
     $\forall I. I \models D \cdot \sigma \cdot \mu + \{\#-(L \cdot l \mu)\# \} \rightarrow I \models C \cdot \mu + \{\#L \cdot l \mu\# \} \rightarrow I \models D \cdot \sigma \cdot \mu + C \cdot \mu$ 
    by auto
  then have  $\forall I. I \models (D + \{\#L'\#\}) \cdot \sigma \cdot \mu \rightarrow I \models (C + \{\#L\#\}) \cdot \mu \rightarrow I \models D \cdot \sigma \cdot \mu + C \cdot \mu$ 
    using assms
    by (metis add_mset_add_single subst_cls_add_mset subst_cls_union subst_minus)
  then have  $\forall I. I \models (D + \{\#L'\#\}) \cdot \sigma \cdot \mu \rightarrow I \models (C + \{\#L\#\}) \cdot \mu \rightarrow I \models C \cdot \mu$ 
    using assms by (metis (no_types, lifting) subset_mset_le_iff_add subst_cls_union true_cls_union)
  then have  $\forall I. I \models m \{ \#(D + \{\#L'\#\}) \cdot \sigma \cdot \mu \# \} \rightarrow I \models (C + \{\#L\#\}) \cdot \mu \rightarrow I \models C \cdot \mu$ 
    by (meson true_cls_mset_singleton)
  ultimately have  $\gamma \in \text{sr\_ext.inferences\_from}(\text{grounding\_of\_clss}(CC \cup \{C + \{\#L\#\}\}))$ 
    unfolding  $\text{sr\_ext.inferences\_from\_def}$  unfolding  $\text{ground\_sound\_}\Gamma\text{\_def}$   $\text{infer\_from\_def}$   $\gamma\text{\_def}$  by auto
  then have  $C \cdot \mu \in \text{concls\_of}(\text{sr\_ext.inferences\_from}(\text{grounding\_of\_clss}(CC \cup \{C + \{\#L\#\}\})))$ 
    using image_iff unfolding  $\gamma\text{\_def}$  by fastforce
  then show  $C\mu \in \text{concls\_of}(\text{sr\_ext.inferences\_from}(\text{grounding\_of\_clss}(CC \cup \{C + \{\#L\#\}\})))$ 
    using  $\mu_p$  by auto
qed

```

```

lemma reduction_derivable:
assumes
   $D + \{\#L'\#\} \in CC$  and
   $-L = L' \cdot l \sigma$  and
   $D \cdot \sigma \subseteq \# C$ 
shows  $\text{sr\_ext.derive}(\text{grounding\_of\_clss}(CC \cup \{C + \{\#L\#\}\})) (\text{grounding\_of\_clss}(CC \cup \{C\}))$ 
proof -
  from assms have  $\text{grounding\_of\_clss}(CC \cup \{C\}) - \text{grounding\_of\_clss}(CC \cup \{C + \{\#L\#\}\}) \subseteq \text{concls\_of}(\text{sr\_ext.inferences\_from}(\text{grounding\_of\_clss}(CC \cup \{C + \{\#L\#\}\})))$ 
    using reduction_in_concls_of unfolding  $\text{grounding\_of\_clss\_def}$  by auto
  moreover
  have  $\text{grounding\_of\_cls}(C + \{\#L\#\}) \subseteq sr.Rf(\text{grounding\_of\_clss}(CC \cup \{C\}))$ 
    using strict_subset_subsumption_grounding_redundant_clss[of C id_subst]
    by auto
  then have  $\text{grounding\_of\_clss}(CC \cup \{C + \{\#L\#\}\}) - \text{grounding\_of\_clss}(CC \cup \{C\}) \subseteq sr.Rf(\text{grounding\_of\_clss}(CC \cup \{C\}))$ 
    unfolding  $\text{grounding\_of\_clss\_def}$  by auto
  ultimately show
     $\text{sr\_ext.derive}(\text{grounding\_of\_clss}(CC \cup \{C + \{\#L\#\}\})) (\text{grounding\_of\_clss}(CC \cup \{C\}))$ 
    using sr_ext.derive.intros[of grounding_of_clss(CC \cup \{C\})]
      grounding_of_clss(CC \cup \{C + \{\#L\#\}\})
    by auto
qed

```

The following corresponds the part of Lemma 4.10 that states we have a theorem proving process:

```

lemma RP_ground_derive:
   $St \rightsquigarrow St' \implies \text{sr\_ext.derive}(\text{grounding\_of\_state } St) (\text{grounding\_of\_state } St')$ 
proof (induction rule: RP.induct)

```

```

case (tautology_deletion A C N P Q)
{
  fix  $C\sigma$ 
  assume  $C\sigma \in \text{grounding\_of\_cls } C$ 
  then obtain  $\sigma$  where
     $C\sigma = C \cdot \sigma$ 
    unfolding grounding_of_cls_def by auto
    then have  $\text{Neg}(A \cdot a \sigma) \in \# C\sigma \wedge \text{Pos}(A \cdot a \sigma) \in \# C\sigma$ 
    using tautology_deletion Neg_Melem_subst_atm_subst_cls Pos_Melem_subst_atm_subst_cls by auto
    then have  $C\sigma \in sr.Rf(\text{grounding\_of\_state}(N, P, Q))$ 
    using sr.tautology_Rf by auto
}
then have  $\text{grounding\_of\_state}(N \cup \{C\}, P, Q) = \text{grounding\_of\_state}(N, P, Q)$ 
 $\subseteq sr.Rf(\text{grounding\_of\_state}(N, P, Q))$ 
unfolding grounding_of_clss_def by auto
moreover have  $\text{grounding\_of\_state}(N, P, Q) = \text{grounding\_of\_state}(N \cup \{C\}, P, Q) = \{\}$ 
unfolding grounding_of_clss_def by auto
ultimately show ?case
  using sr_ext.derive.intros[of grounding_of_state ( $N, P, Q$ )
   $\text{grounding\_of\_state}(N \cup \{C\}, P, Q)]$ 
  by auto
next
case (forward_subsumption D P Q C N)
then show ?case
  using derive_if_remove_subsumed[of D (N, P, Q) C] unfolding grounding_of_clss_def
  by (simp add: sup_commute sup_left_commute)
next
case (backward_subsumption_P D N C P Q)
then show ?case
  using derive_if_remove_subsumed[of D (N, P, Q) C] strictly_subsumes_def
  unfolding grounding_of_clss_def by (simp add: sup_commute sup_left_commute)
next
case (backward_subsumption_Q D N C P Q)
then show ?case
  using derive_if_remove_subsumed[of D (N, P, Q) C] strictly_subsumes_def
  unfolding grounding_of_clss_def by (simp add: sup_commute sup_left_commute)
next
case (forward_reduction D L' P Q L σ C N)
then show ?case
  using reduction_derivable[of __ N ∪ P ∪ Q] by force
next
case (backward_reduction_P D L' N L σ C P Q)
then show ?case
  using reduction_derivable[of __ N ∪ P ∪ Q] by force
next
case (backward_reduction_Q D L' N L σ C P Q)
then show ?case
  using reduction_derivable[of __ N ∪ P ∪ Q] by force
next
case (clause_processing N C P Q)
then show ?case
  using sr_ext.derive.intros by auto
next
case (inference_computation N Q C P)
{
  fix  $E\mu$ 
  assume  $E\mu \in \text{grounding\_of\_clss } N$ 
  then obtain  $\mu E$  where
     $E_{\mu_p}: E\mu = E \cdot \mu \wedge E \in N \wedge \text{is\_ground\_subst } \mu$ 
    unfolding grounding_of_clss_def grounding_of_cls_def by auto
    then have  $E_{\text{concl}}: E \in \text{concls\_of}(\text{ord\_FO\_resolution.inferences\_between } Q C)$ 
    using inference_computation by auto
    then obtain  $\gamma$  where

```

$\gamma_p: \gamma \in \text{ord_FO_}\Gamma S \wedge \text{infer_from}(Q \cup \{C\}) \gamma \wedge C \in \# \text{prems_of } \gamma \wedge \text{concl_of } \gamma = E$
unfolding $\text{ord_FO_resolution.inferences_between_def by auto}$
then obtain $CC \text{ CAs } D \text{ AAs } As \sigma \text{ where}$
 $\gamma_p2: \gamma = \text{Infer } CC D E \wedge \text{ord_resolve_rename } S \text{ CAs } D \text{ AAs } As \sigma E \wedge \text{mset CAs} = CC$
unfolding $\text{ord_FO_}\Gamma \text{ def by auto}$
define ϱ **where**
 $\varrho = \text{hd}(\text{renamings_apart}(D \# \text{CAs}))$
define ϱs **where**
 $\varrho s = \text{tl}(\text{renamings_apart}(D \# \text{CAs}))$
define γ_ground **where**
 $\gamma_ground = \text{Infer}(\text{mset}(\text{CAs} \cdot \cdot cl \varrho s) \cdot cm \sigma \cdot cm \mu) (D \cdot \varrho \cdot \sigma \cdot \mu) (E \cdot \mu)$
have $\forall I. I \models m \text{mset}(\text{CAs} \cdot \cdot cl \varrho s) \cdot cm \sigma \cdot cm \mu \rightarrow I \models D \cdot \varrho \cdot \sigma \cdot \mu \rightarrow I \models E \cdot \mu$
using $\text{ord_resolve_rename_ground_inst_sound}[\text{of } \dots \mu] \varrho \text{def } \varrho s \text{def } E \cdot \mu \cdot p \gamma_p2$
by auto
then have $\gamma_ground \in \{\text{Infer } cc d e \mid cc d e. \forall I. I \models m cc \rightarrow I \models d \rightarrow I \models e\}$
unfolding γ_ground_def **by** auto
moreover have $\text{set_mset}(\text{prems_of } \gamma_ground) \subseteq \text{grounding_of_state}(\{\}, P \cup \{C\}, Q)$
proof –
have $D = C \vee D \in Q$
unfolding γ_ground_def **using** $E \cdot \mu \cdot p \gamma_p2 \gamma_p$ **unfolding** infer_from_def
unfolding $\text{grounding_of_cls_def}$ $\text{grounding_of_cls_def}$ **by** simp
then have $D \cdot \varrho \cdot \sigma \cdot \mu \in \text{grounding_of_cls } C \vee (\exists x \in Q. D \cdot \varrho \cdot \sigma \cdot \mu \in \text{grounding_of_cls } x)$
using $E \cdot \mu \cdot p$
unfolding $\text{grounding_of_cls_def}$
by $(\text{metis}(\text{mono_tags}, \text{lifting}) \text{ is_ground_comp_subst } \text{mem_Collect_eq_subst_cls_comp_subst})$
then have $(D \cdot \varrho \cdot \sigma \cdot \mu \in \text{grounding_of_cls } C \vee$
 $(\exists x \in P. D \cdot \varrho \cdot \sigma \cdot \mu \in \text{grounding_of_cls } x) \vee$
 $(\exists x \in Q. D \cdot \varrho \cdot \sigma \cdot \mu \in \text{grounding_of_cls } x))$
by metis
moreover have $\forall i < \text{length}(\text{CAs} \cdot \cdot cl \varrho s \cdot cl \sigma \cdot cl \mu). (\text{CAs} \cdot \cdot cl \varrho s \cdot cl \sigma \cdot cl \mu) ! i \in$
 $\{C \cdot \sigma \mid \sigma. \text{is_ground_subst } \sigma\} \cup$
 $((\bigcup C \in P. \{C \cdot \sigma \mid \sigma. \text{is_ground_subst } \sigma\}) \cup (\bigcup C \in Q. \{C \cdot \sigma \mid \sigma. \text{is_ground_subst } \sigma\}))$
proof (*rule, rule*)
fix i
assume $i < \text{length}(\text{CAs} \cdot \cdot cl \varrho s \cdot cl \sigma \cdot cl \mu)$
then have $a: i < \text{length } \text{CAs} \wedge i < \text{length } \varrho s$
by simp
moreover from a **have** $\text{CAs} ! i \in \{C\} \cup Q$
using $\gamma_p2 \gamma_p$ **unfolding** infer_from_def
by $(\text{metis}(\text{no_types}, \text{lifting}) \text{ Un_subset_iff } \text{inference.sel}(1) \text{ set_mset_union}$
 $\text{sup_commute } \text{nth_mem_mset } \text{subsetCE})$
ultimately have $(\text{CAs} \cdot \cdot cl \varrho s \cdot cl \sigma \cdot cl \mu) ! i \in$
 $\{C \cdot \sigma \mid \sigma. \text{is_ground_subst } \sigma\} \vee$
 $((\text{CAs} \cdot \cdot cl \varrho s \cdot cl \sigma \cdot cl \mu) ! i \in (\bigcup C \in P. \{C \cdot \sigma \mid \sigma. \text{is_ground_subst } \sigma\}) \vee$
 $(\text{CAs} \cdot \cdot cl \varrho s \cdot cl \sigma \cdot cl \mu) ! i \in (\bigcup C \in Q. \{C \cdot \sigma \mid \sigma. \text{is_ground_subst } \sigma\}))$
using $E \cdot \mu \cdot p \gamma_p2 \gamma_p$
unfolding γ_ground_def **infer_from_def** $\text{grounding_of_cls_def}$ $\text{grounding_of_cls_def}$
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 $\varrho s \text{def}$ **using** $\text{renamings_apart_length}$ **by** (*auto; fail*)
subgoal
apply (*rule disjI2*)
apply (*rule disjI2*)
apply (*rule tac a = CAs ! i in UN_I*)
subgoal by *blast*
subgoal
apply (*rule Set.CollectI*)
apply (*rule tac x = (ρs ! i) ⊕ σ ⊕ μ in exI*)
using $\varrho s \text{def}$ **using** $\text{renamings_apart_length}$ **by** (*auto; fail*)

```

done
done
then show (CAs ..cl gs ·cl σ ·cl μ) ! i ∈ {C · σ |σ. is_ground_subst σ} ∪
  ((U C ∈ P. {C · σ |σ. is_ground_subst σ}) ∪ (U C ∈ Q. {C · σ |σ. is_ground_subst σ}))
  by blast
qed
then have ∀ x ∈# mset (CAs ..cl gs ·cl σ ·cl μ). x ∈ {C · σ |σ. is_ground_subst σ} ∪
  ((U C ∈ P. {C · σ |σ. is_ground_subst σ}) ∪ (U 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 gs) ·cm σ ·cm μ) ⊆
  grounding_of_cls C ∪ grounding_of_cls P ∪ grounding_of_cls Q
  unfolding grounding_of_cls_def grounding_of_cls_def
  using mset_subst_cls_list_subst_cls_mset by auto
ultimately show ?thesis
  unfolding γ_ground_def grounding_of_cls_def by auto
qed
ultimately have
  E · μ ∈ concls_of (sr_ext.inferences_from (grounding_of_state ({}, P ∪ {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 (mono_tags, lifting) image_eqI inference.sel(3) mem_Collect_eq)
then have Eμ ∈ concls_of (sr_ext.inferences_from (grounding_of_state ({}, P ∪ {C}, Q)))
  using E_μ_p by auto
}
then have grounding_of_state (N, P, Q ∪ {C}) = grounding_of_state ({}, P ∪ {C}, Q)
  ⊆ concls_of (sr_ext.inferences_from (grounding_of_state ({}, P ∪ {C}, Q)))
  unfolding grounding_of_cls_def by auto
moreover have grounding_of_state ({}, P ∪ {C}, Q) = grounding_of_state (N, P, Q ∪ {C}) = {}
  unfolding grounding_of_cls_def by auto
ultimately show ?case
  using sr_ext.derive.intros[of (grounding_of_state (N, P, Q ∪ {C}))]
  (grounding_of_state ({}, P ∪ {C}, Q)) by auto
qed

```

A useful consequence:

theorem RP_model: $St \rightsquigarrow St' \implies I \models s \text{ grounding_of_state } St' \longleftrightarrow I \models s \text{ grounding_of_state } St$

proof (drule RP_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 $I \models s \text{ ?gSt}' \longleftrightarrow I \models s \text{ ?gSt}$

proof

assume bef: $I \models s \text{ ?gSt}$

then have $I \models s \text{ ?concls}$

unfolding ground_sound_Γ_def inference_system.inferences_from_def true_cls_def
true_cls_mset_def

by (auto simp add: image_def infer_from_def dest!: spec[of _ I])

then have diff: $I \models s \text{ ?gSt}' - ?gSt$

using deduct by (blast intro: true_cls_mono)

then show $I \models s \text{ ?gSt}'$

using bef unfolding true_cls_def by blast

next

assume aft: $I \models s \text{ ?gSt}'$

have $I \models s \text{ ?gSt}' \cup sr.Rf ?gSt'$

by (rule sr.Rf_model) (smt (verit) Diff_eq_empty_iff Diff_subset Un_Diff aft
standard_redundancy_criterion.Rf_mono sup_bot.right_neutral sup_ge1 true_cls_mono)

then have $I \models s \text{ sr.Rf ?gSt}'$

using true_cls_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 ground_derive_chain: chain ( $\rightsquigarrow$ ) Sts  $\implies$  chain sr_ext.derive (lmap grounding_of_state Sts)
  using RP_ground_derive by (simp add: chain_lmap[of ( $\rightsquigarrow$ )])

```

The following is used prove to Lemma 4.11:

```

lemma Sup_llist_grounding_of_state_ground:
  assumes C ∈ Sup_llist (lmap grounding_of_state Sts)
  shows is_ground_cls C
proof -
  have  $\exists j. \text{enat } j < \text{llength} (\text{lmap grounding_of_state Sts})$ 
     $\wedge C \in \text{lnth} (\text{lmap grounding_of_state Sts})^j$ 
    using assms Sup_llist_imp_exists_index by fast
  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)  $\implies$  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  $\exists D \sigma. D \in \text{clss\_of\_state} (\text{Sup\_state Sts}) \wedge D \cdot \sigma = C \wedge \text{is\_ground\_subst} \sigma$ 
proof -
  from assms obtain i where
    i_p:  $\text{enat } i < \text{llength Sts} \wedge C \in \text{lnth} (\text{lmap grounding_of_state Sts})^i$ 
    using Sup_llist_imp_exists_index by fastforce
  then obtain D σ where
    D ∈ clss_of_state (lnth Sts i)  $\wedge D \cdot \sigma = C \wedge \text{is\_ground\_subst} \sigma$ 
    using assms unfolding grounding_of_clss_def grounding_of_cls_def by fastforce
  then have D ∈ clss_of_state (Sup_state Sts)  $\wedge D \cdot \sigma = C \wedge \text{is\_ground\_subst} \sigma$ 
    using i_p unfolding Sup_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:  $N_{\text{of\_state}} (\text{Liminf\_state Sts}) = \text{Liminf\_llist} (\text{lmap } N_{\text{of\_state}} \text{ Sts})$  and
  P_of_state_Liminf:  $P_{\text{of\_state}} (\text{Liminf\_state Sts}) = \text{Liminf\_llist} (\text{lmap } P_{\text{of\_state}} \text{ Sts})$ 
  unfolding Liminf_state_def by auto

```

```

lemma eventually_removed_from_N:
  assumes
    d_in:  $D \in N_{\text{of\_state}} (\text{lnth Sts } i)$  and
    fair: fair_state_seq Sts and
    i_Sts:  $\text{enat } i < \text{llength Sts}$ 
  shows  $\exists l. D \in N_{\text{of\_state}} (\text{lnth Sts } l) \wedge D \notin N_{\text{of\_state}} (\text{lnth Sts } (\text{Suc } l)) \wedge i \leq l$ 
     $\wedge \text{enat } (\text{Suc } l) < \text{llength Sts}$ 
proof (rule ccontr)
  assume a:  $\neg ?\text{thesis}$ 
  have i ≤ l  $\implies$   $\text{enat } l < \text{llength Sts} \implies D \in N_{\text{of\_state}} (\text{lnth Sts } l)$  for l
    using d_in by (induction l, blast, metis a Suc ile_eq le_SucE less_imp_le)

```

```

then have  $D \in \text{Liminf\_llist} (\text{lmap } N\_\text{of}\_\text{state } Sts)$ 
  unfolding  $\text{Liminf\_llist\_def}$  using  $i\_\text{Sts}$  by auto
then show False
  using fair unfolding  $\text{fair\_state\_seq\_def}$  by (simp add:  $N\_\text{of}\_\text{state}\_\text{Liminf}$ )
qed

lemma eventually_removed_from_P:
assumes
   $d\_\text{in}: D \in P\_\text{of}\_\text{state} (\text{lnth } Sts i) \text{ and}$ 
   $\text{fair}: \text{fair\_state\_seq } Sts \text{ and}$ 
   $i\_\text{Sts}: \text{enat } i < \text{llength } Sts$ 
shows  $\exists l. D \in P\_\text{of}\_\text{state} (\text{lnth } Sts l) \wedge D \notin P\_\text{of}\_\text{state} (\text{lnth } Sts (\text{Suc } l)) \wedge i \leq l$ 
   $\wedge \text{enat } (\text{Suc } l) < \text{llength } Sts$ 
proof (rule ccontr)
assume  $a: \neg ?\text{thesis}$ 
have  $i \leq l \implies \text{enat } l < \text{llength } Sts \implies D \in P\_\text{of}\_\text{state} (\text{lnth } Sts l) \text{ for } l$ 
  using  $d\_\text{in}$  by (induction l, blast, metis a Suc_ilc_eq le_SucE less_imp_le)
then have  $D \in \text{Liminf\_llist} (\text{lmap } P\_\text{of}\_\text{state } Sts)$ 
  unfolding  $\text{Liminf\_llist\_def}$  using  $i\_\text{Sts}$  by auto
then show False
  using fair unfolding  $\text{fair\_state\_seq\_def}$  by (simp add:  $P\_\text{of}\_\text{state}\_\text{Liminf}$ )
qed

lemma instance_if_subsumed_and_in_limit:
assumes
  deriv:  $\text{chain } (\rightsquigarrow) Sts \text{ and}$ 
  ns:  $Gs = \text{lmap } \text{grounding\_of\_state } Sts \text{ and}$ 
  c:  $C \in \text{Liminf\_llist } Gs - sr.Rf (\text{Liminf\_llist } Gs) \text{ and}$ 
  d:  $D \in \text{clss\_of\_state} (\text{lnth } Sts i) \text{ enat } i < \text{llength } Sts \text{ subsumes } D C$ 
shows  $\exists \sigma. D \cdot \sigma = C \wedge \text{is\_ground\_subst } \sigma$ 
proof -
let ?Ps =  $\lambda i. P\_\text{of}\_\text{state} (\text{lnth } Sts i)$ 
let ?Qs =  $\lambda i. Q\_\text{of}\_\text{state} (\text{lnth } Sts i)$ 

have ground_C:  $\text{is\_ground\_cls } C$ 
  using c using  $\text{Liminf\_grounding\_of\_state\_ground } ns$  by auto

have derivns:  $\text{chain } sr.\text{ext}.derive Gs$ 
  using ground_derive_chain 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  $\tau\_\text{proto}$  where
   $D \cdot \tau\_\text{proto} \subseteq\# C$  unfolding subsumes_def
  by blast
then obtain  $\tau$  where
   $\tau\_p: D \cdot \tau \subseteq\# C \wedge \text{is\_ground\_subst } \tau$ 
  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  $\tau$ 
  using  $\tau\_p$  by auto
moreover have  $D \in \text{clss\_of\_state} (\text{lnth } Sts i)$ 
  using d by auto
ultimately have  $C \in sr.Rf (\text{grounding\_of\_state} (\text{lnth } Sts i))$ 
  using strict_subset_subsumption_redundant_clss by auto
then have  $C \in sr.Rf (\text{Sup\_llist } Gs)$ 
  using d ns by (smt (verit) contra_subsetD llength_lmap lnth_lmap lnth_subset_Sup_llist sr.Rf_mono)
then have  $C \in sr.Rf (\text{Liminf\_llist } Gs)$ 
  unfolding ns using local.sr_ext.Rf_limit_Sup derivns ns by auto
then show False
  using c by auto

```

```

qed
then obtain  $\sigma$  where
   $D \cdot \sigma = C \wedge \text{is\_ground\_subst } \sigma$ 
  using ground_C by (metis make_ground_subst)
then show ?thesis
  by auto
qed

lemma from_Q_to_Q_inf:
assumes
  deriv: chain ( $\rightsquigarrow$ ) Sts and
  fair: fair_state_seq Sts and
  ns:  $Gs = lmap \text{grounding\_of\_state} Sts$  and
  c:  $C \in \text{Liminf\_llist} Gs - sr.Rf (\text{Liminf\_llist} Gs)$  and
  d:  $D \in Q_{\text{of\_state}} (\text{lnth} Sts i) \text{ enat } i < \text{llength} Sts \text{ subsumes } D C$  and
  d_least:  $\forall E \in \{E. E \in (\text{clss\_of\_state} (\text{Sup\_state} Sts)) \wedge \text{subsumes } E C\}.$ 
     $\neg \text{strictly\_subsumes } E D$ 
shows  $D \in Q_{\text{of\_state}} (\text{Liminf\_state} Sts)$ 
proof -
  let ?Ps =  $\lambda i. P_{\text{of\_state}} (\text{lnth} Sts i)$ 
  let ?Qs =  $\lambda i. Q_{\text{of\_state}} (\text{lnth} Sts i)$ 

  have ground_C: is_ground_cls C
    using c using Liminf_grounding_of_state_ground ns by auto

  have derivns: chain sr_ext.derive Gs
    using ground_derive_chain deriv ns by auto

  have  $\exists \sigma. D \cdot \sigma = C \wedge \text{is\_ground\_subst } \sigma$ 
    using instance_if_subsumed_and_in_limit[OF deriv] c d unfolding ns by blast
  then obtain  $\sigma$  where
     $\sigma: D \cdot \sigma = C \text{ is\_ground\_subst } \sigma$ 
    by auto

  have in_Sts_in_Sts_Suc:
     $\forall l \geq i. \text{enat} (\text{Suc } l) < \text{llength} Sts \longrightarrow D \in Q_{\text{of\_state}} (\text{lnth} Sts l) \longrightarrow$ 
     $D \in Q_{\text{of\_state}} (\text{lnth} Sts (\text{Suc } l))$ 
  proof (rule, rule, rule, rule)
    fix l
    assume
      len:  $i \leq l$  and
      llen:  $\text{enat} (\text{Suc } l) < \text{llength} Sts$  and
      d_in_q:  $D \in Q_{\text{of\_state}} (\text{lnth} Sts l)$ 

    have lnth_Sts_l ~> lnth_Sts_(Suc l)
      using llen deriv chain_lnth_rel by blast
    then show  $D \in Q_{\text{of\_state}} (\text{lnth} Sts (\text{Suc } l))$ 
    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:  $D_{\text{subsumes}} \in N \wedge \text{strictly\_subsumes } D_{\text{subsumes}} D$ 
          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  $\in \text{clss\_of\_state} (\text{Sup\_state} Sts)$ 
          using D_subsumes_p llen
          by (metis (no_types) UnI1 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
      }
    qed
  qed
qed

```

```

}
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 P_of_state.simps llen_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) < llen 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 < llen 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
  deriv: chain (↝) Sts and
  fair: fair_state_seq Sts and
  ns: Gs = lmap grounding_of_state Sts and
  c: C ∈ Liminf_llist Gs – sr.Rf (Liminf_llist Gs) and
  d: D ∈ P_of_state (lnth Sts i) enat i < llen 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 < llen 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 Gs
  using ground_derive_chain deriv ns by auto

have  $\exists \sigma. D \cdot \sigma = C \wedge \text{is\_ground\_subst } \sigma$ 
  using instance_if_subsumed_and_in_limit[OF deriv] ns c d by blast
then obtain  $\sigma$  where
   $\sigma: D \cdot \sigma = C \text{ is\_ground\_subst } \sigma$ 
  by auto

obtain l where
   $l\_p: D \in P\_of\_state (\lnth Sts l) \wedge D \notin P\_of\_state (\lnth Sts (\Suc l)) \wedge i \leq l$ 
   $\wedge \text{enat} (\Suc l) < \text{llength } Sts$ 
  using fair using eventually_removed_from_P d unfolding ns by auto
then have l_Gs: enat (Suc l) < llength Gs
  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 (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  $\cup \{D\}$  ?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  $\sigma$ 
    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 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  $\sigma$  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 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  $\cup \{D\}$ 
    ?Qs (Suc l) = Q  $\cup \{D\}$  ?Qs l = Q
    using l_p by auto
  then show ?thesis
    using d  $\sigma$  l_p by auto
qed (use l_p in auto)
qed

```

lemma from_N_to_P_or_Q:

assumes

deriv: chain (\rightsquigarrow) Sts **and**
fair: fair_state_seq Sts **and**
ns: $Gs = lmap grounding_of_state Sts$ **and**
c: $C \in Liminf_llist Gs - sr.Rf(Liminf_llist Gs)$ **and**
d: $D \in N_of_state (\lnth Sts i) \wedge enat i < llength Sts \wedge subsumes D C$ **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 deriv_ns: chain sr_ext.derive Gs
using ground_derive_chain deriv ns by auto

have $\exists \sigma. D \cdot \sigma = C \wedge is_ground_subst \sigma$
using instance_if_subsumed_and_in_limit[OF deriv] ns c d by blast
then obtain σ where
 $\sigma: D \cdot \sigma = C$ is_ground_subst σ
by auto

from c have no_taut: $\neg (\exists A. Pos A \in\# C \wedge Neg A \in\# C)$
using sr.tautology_Rf by auto

have $\exists l. D \in N_of_state (\lnth Sts l) \wedge D \notin N_of_state (\lnth Sts (\Suc l)) \wedge i \leq l \wedge enat (\Suc l) < llength Sts$
using fair using eventually_removed_from_N d unfolding ns by auto
then obtain l where
l_p: $D \in N_of_state (\lnth Sts l) \wedge D \notin N_of_state (\lnth Sts (\Suc l)) \wedge i \leq l$
 $\wedge enat (\Suc l) < llength Sts$
by auto

then have l_Gs: $enat (\Suc l) < llength Gs$
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 (tautology_deletion A D_twin N P Q)
then have D_twin = D
using l_p by auto

then have Pos (A · a σ) $\in\# C \wedge Neg (A \cdot a \sigma) \in\# C$
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 lhs = this(1,2) **and** D'_p = this(3,4)
then have twins: $D_twin = D \cdot ?Ns (\Suc l) = N \cdot ?Ns l = N \cup \{D_twin\} \quad ?Ps (\Suc l) = P \cdot ?Ps l = P \cdot ?Qs (\Suc l) = Q \cdot ?Qs l = Q$
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 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_strictly_subsumes_variants[of _ D D'] by auto

from v have ∃ σ'. D' · σ' = C
  using σ variants_imp_exists_substitution 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 (forward_reduction E L' P Q L σ D' N)
  then have twins: D' + {#L#} = D ?Ns (Suc l) = N ∪ {D'} ?Ns l = N ∪ {D' + {#L#}}
    ?Ps (Suc l) = P ?Ps l = P ?Qs (Suc l) = Q ?Qs l = Q
    using l_p by auto
  then have D'_p: strictly_subsumes D' D ∧ D' ∈ ?Ns (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 blast
  from D'_p have D' ∈ clss_of_state (Sup_state Sts)
    using l_p by (metis (no_types) UniI 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 (clause_processing N D_twin P Q)
  then have twins: D_twin = D ?Ns (Suc l) = N ?Ns l = N ∪ {D} ?Ps (Suc l) = P ∪ {D}
    ?Ps l = P ?Qs (Suc l) = Q ?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

deriv: chain (\rightsquigarrow) Sts **and**
 $D_p: D \in \text{clss_of_state}(\text{Sup_state Sts})$
 $\text{subsumes } D \ C \ \forall E \in \{E. E \in (\text{clss_of_state}(\text{Sup_state Sts})) \wedge \text{subsumes } E \ C\}$.
 $\neg \text{strictly_subsumes } E \ D$ **and**

fair: fair_state_seq Sts **and**
 $ns: Gs = lmap \text{grounding_of_state} Sts$ **and**
 $c: C \in \text{Liminf_llist} Gs - sr.Rf(\text{Liminf_llist} Gs)$ **and**
 $\text{ground_C: is_ground_cls } C$

shows $\exists D' \sigma'. D' \in Q_{\text{of_state}}(\text{Liminf_state Sts}) \wedge D' \cdot \sigma' = C \wedge \text{is_ground_subst } \sigma'$

proof –

```

let ?Ns =  $\lambda i. N\_of\_state (lnth Sts i)$ 
let ?Ps =  $\lambda i. P\_of\_state (lnth Sts i)$ 
let ?Qs =  $\lambda i. Q\_of\_state (lnth Sts i)$ 

from D_p obtain i where
  i_p:  $i < llength Sts$   $D \in ?Ns i \vee D \in ?Ps i \vee D \in ?Qs i$ 
  unfolding Sup_state_def
  by simp_all (metis (no_types) Sup_llist_imp_exists_index llength_lmap lnth_lmap)

have derivns: chain sr_ext.derive Gs
  using ground_derive_chain deriv ns by auto

have  $\exists \sigma. D \cdot \sigma = C \wedge is\_ground\_subst \sigma$ 
  using instance_if_subsumed_and_in_limit[OF deriv ns c] D_p i_p by blast
then obtain  $\sigma$  where
   $\sigma: D \cdot \sigma = C$  is_ground_subst  $\sigma$ 
  by blast

{
  assume a:  $D \in ?Ns i$ 
  then obtain D'  $\sigma'$  l where D'_p:
     $D' \in ?Ps l \cup ?Qs l$ 
     $D' \cdot \sigma' = C$ 
    enat  $l < llength Sts$ 
    is_ground_subst  $\sigma'$ 
     $\forall E \in \{E. E \in (clss\_of\_state (Sup\_state Sts)) \wedge subsumes E C\}. \neg strictly\_subsumes E D'$ 
    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:  $D' \in ?Qs l' l' < llength Sts$ 
    using from_P_to_Q[OF deriv fair ns c D'_p(3) D'_p(6) D'_p(5)] by blast
    then have D'  $\in Q\_of\_state (Liminf\_state Sts)$ 
      using from_Q_to_Q_inf[OF deriv fair ns c l'_p(2)] D'_p by auto
    then have ?thesis
      using D'_p by auto
}
moreover
{
  assume a:  $D \in ?Ps i$ 
  then obtain l' where
    l'_p:  $D \in ?Qs l' l' < llength Sts$ 
    using from_P_to_Q[OF deriv fair ns c a i_p(1) D_p(2) D_p(3)] by auto
  then have D  $\in Q\_of\_state (Liminf\_state Sts)$ 
    using from_Q_to_Q_inf[OF deriv fair ns c l'_p(1) l'_p(2)] D_p(3)  $\sigma(1) \sigma(2)$  D_p(2) by auto
  then have ?thesis
    using D_p  $\sigma$  by auto
}
moreover
{
  assume a:  $D \in ?Qs i$ 
  then have D  $\in Q\_of\_state (Liminf\_state Sts)$ 
    using from_Q_to_Q_inf[OF deriv fair ns c a i_p(1)]  $\sigma$  D_p(2,3) by auto
  then have ?thesis
    using D_p  $\sigma$  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 ( $\rightsquigarrow$ ) Sts and

```

```

fair: fair_state_seq Sts and
  ns: Gs = lmap grounding_of_state Sts
shows Liminf_llist Gs = sr.Rf (Liminf_llist Gs)
  ⊆ 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 by auto

  fix C
  assume C_p: C ∈ Liminf_llist Gs = sr.Rf (Liminf_llist Gs)
  then have C ∈ Sup_llist Gs
    using Liminf_llist_subset_Sup_llist[of Gs] 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 ∃ D' σ'. D' ∈ Q_of_state (Liminf_state Sts) ∧ D' · σ' = C ∧ is_ground_subst σ'
    using eventually_in_Qinf[of D C Gs] using D_p(1–3) deriv fair ns C_p ground_C by auto
  then obtain D' σ' where
    D'_p: D' ∈ Q_of_state (Liminf_state Sts) ∧ D' · σ' = C ∧ is_ground_subst σ'
    by blast
  then have D' ∈ clss_of_state (Liminf_state Sts)
    by simp
  then have C ∈ grounding_of_state (Liminf_state Sts)
    unfolding grounding_of_clss_def grounding_of_cls_def using D'_p by auto
  then show C ∈ grounding_of_clss (Q_of_state (Liminf_state Sts))
    using SQinf fair fair_state_seq_def by auto
qed

```

The following corresponds to (one direction of) Theorem 4.13:

```

lemma subseteq_Liminf_state_eventually_always:
  fixes CC
  assumes
    finite CC and
    CC ≠ {} and
    CC ⊆ Q_of_state (Liminf_state Sts)
  shows ∃ j. enat j < llength Sts ∧ (∀ j' ≥ enat j. j' < llength Sts → CC ⊆ Q_of_state (lnth Sts j'))
proof –
  from assms(3) have ∀ C ∈ CC. ∃ j. enat j < llength Sts ∧
    (∀ j' ≥ enat j. j' < llength Sts → C ∈ Q_of_state (lnth Sts j'))
    unfolding Liminf_state_def Liminf_llist_def by force
  then obtain f where
    f_p: ∀ C ∈ CC. f C < llength Sts ∧ (∀ j' ≥ enat (f C). j' < llength Sts → C ∈ 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  $\forall C j'. C \in CC \rightarrow \text{enat } j \leq j' \rightarrow j' < \text{llength } Sts \rightarrow C \in Q_{\text{of\_state}} (\text{lnth } Sts j')$ 
proof (intro allI impI)
fix  $C :: 'a \text{ clause}$  and  $j' :: \text{nat}$ 
assume  $a: C \in CC \text{ enat } j \leq \text{enat } j' \text{ enat } j' < \text{llength } Sts$ 
then have  $f C \leq j'$ 
unfolding j_def using assms(1) Max.bounded_iff by auto
then show  $C \in Q_{\text{of\_state}} (\text{lnth } Sts j')$ 
using f_p a by auto
qed
ultimately show ?thesis
by auto
qed

lemma empty_clause_in_Q_of_Liminf_state:
assumes
deriv: chain ( $\rightsquigarrow$ ) Sts and
fair: fair_state_seq Sts and
empty_in:  $\{\#\} \in \text{Liminf\_llist} (\text{lmap grounding\_of\_state } Sts)$ 
shows  $\{\#\} \in Q_{\text{of\_state}} (\text{Liminf\_state } Sts)$ 
proof -
define Gs :: 'a clause set llist where
ns:  $Gs = \text{lmap grounding\_of\_state } Sts$ 
from empty_in have in_Liminf_not_Rf:  $\{\#\} \in \text{Liminf\_llist } Gs - sr.Rf (\text{Liminf\_llist } Gs)$ 
unfolding ns sr.Rf_def by auto
then have  $\{\#\} \in \text{grounding\_of\_clss } (Q_{\text{of\_state}} (\text{Liminf\_state } Sts))$ 
using fair_imp_Liminf_minus_Rf_subset_ground_Liminf_state[OF deriv fair ns] by auto
then show ?thesis
unfolding grounding_of_clss_def grounding_of_cls_def by auto
qed

lemma grounding_of_state_Liminf_state_subseteq:
grounding_of_state (Liminf_state Sts)  $\subseteq \text{Liminf\_llist} (\text{lmap grounding\_of\_state } Sts)$ 
proof
fix C :: 'a clause
assume C ∈ grounding_of_state (Liminf_state Sts)
then obtain D σ where
D_σ_p:  $D \in \text{clss\_of\_state } (\text{Liminf\_state } Sts) D \cdot \sigma = C \text{ is\_ground\_subst } \sigma$ 
unfolding grounding_of_clss_def grounding_of_cls_def by auto
then have ii:  $D \in \text{Liminf\_llist} (\text{lmap } N_{\text{of\_state}} Sts)$ 
 $\vee D \in \text{Liminf\_llist} (\text{lmap } P_{\text{of\_state}} Sts) \vee D \in \text{Liminf\_llist} (\text{lmap } Q_{\text{of\_state}} Sts)$ 
unfolding Liminf_state_def by simp
then have C ∈ Liminf_llist (lmap grounding_of_clss (lmap N_of_state Sts))
 $\vee C \in \text{Liminf\_llist} (\text{lmap grounding\_of\_clss } (\text{lmap } P_{\text{of\_state}} Sts))$ 
 $\vee C \in \text{Liminf\_llist} (\text{lmap grounding\_of\_clss } (\text{lmap } Q_{\text{of\_state}} Sts))$ 
unfolding Liminf_llist_def grounding_of_clss_def grounding_of_cls_def
using D_σ_p
apply -
apply (erule disjE)
subgoal
apply (rule disjI1)
using D_σ_p by auto
subgoal
apply (erule 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  $C \in \text{Liminf\_llist}(\text{lmap grounding\_of\_state } Sts)$ 
  unfolding  $\text{Liminf\_llist\_def}$   $\text{grounding\_of\_clss\_def}$  by auto
qed

theorem  $RP\_sound$ :
assumes
   $\text{deriv: chain } (\rightsquigarrow) Sts \text{ and}$ 
   $\{\#\} \in \text{clss\_of\_state}(\text{Liminf\_state } Sts)$ 
shows  $\neg \text{satisfiable}(\text{grounding\_of\_state}(lhd Sts))$ 
proof -
  from assms have  $\{\#\} \in \text{grounding\_of\_state}(\text{Liminf\_state } Sts)$ 
    unfolding  $\text{grounding\_of\_clss\_def}$  by (force intro:  $\text{ex\_ground\_subst}$ )
  then have  $\{\#\} \in \text{Liminf\_llist}(\text{lmap grounding\_of\_state } Sts)$ 
    using  $\text{grounding\_of\_state\_Liminf\_state\_subseteq}$  by auto
  then have  $\neg \text{satisfiable}(\text{Liminf\_llist}(\text{lmap grounding\_of\_state } Sts))$ 
    using  $\text{true\_clss\_def}$  by auto
  then have  $\neg \text{satisfiable}(lhd(\text{lmap grounding\_of\_state } Sts))$ 
    using  $\text{sr\_ext.sat\_limit\_iff ground\_derive\_chain deriv}$  by blast
  then show ?thesis
    using  $\text{chain\_not\_lnull deriv}$  by fastforce
qed

```

```

theorem  $RP\_saturated\_if\_fair$ :
assumes
   $\text{deriv: chain } (\rightsquigarrow) Sts \text{ and}$ 
   $\text{fair: fair\_state\_seq } Sts \text{ and}$ 
   $\text{empty\_Q0: } Q\text{\_of\_state}(lhd Sts) = \{\}$ 
shows  $\text{sr.saturated\_upto}(\text{Liminf\_llist}(\text{lmap grounding\_of\_state } Sts))$ 
proof -
  define  $Gs :: \text{'a clause set llist where}$ 
   $ns: Gs = \text{lmap grounding\_of\_state } Sts$ 

  let ?N =  $\lambda i. \text{grounding\_of\_state}(\text{lnth } Sts i)$ 

  let ?Ns =  $\lambda i. N\text{\_of\_state}(\text{lnth } Sts i)$ 
  let ?Ps =  $\lambda i. P\text{\_of\_state}(\text{lnth } Sts i)$ 
  let ?Qs =  $\lambda i. Q\text{\_of\_state}(\text{lnth } Sts i)$ 

  have  $\text{ground\_ns\_in\_ground\_limit\_st:}$ 
     $\text{Liminf\_llist } Gs - sr.Rf(\text{Liminf\_llist } Gs) \subseteq \text{grounding\_of\_clss}(Q\text{\_of\_state}(\text{Liminf\_state } Sts))$ 
    using  $\text{fair deriv fair\_imp\_Liminf\_minus\_Rf\_subset\_ground\_Liminf\_state ns}$  by blast

  have  $\text{derivns: chain sr\_ext.derive } Gs$ 
    using  $\text{ground\_derive\_chain deriv ns}$  by auto

  {
    fix  $\gamma :: \text{'a inference}$ 
    assume  $\gamma\_p: \gamma \in gr.\text{ord\_}\Gamma$ 
    let ?CC =  $\text{side\_prems\_of } \gamma$ 
    let ?DA =  $\text{main\_prem\_of } \gamma$ 
    let ?E =  $\text{concl\_of } \gamma$ 
    assume  $a: \text{set\_mset } ?CC \cup \{?DA\}$ 
       $\subseteq \text{Liminf\_llist}(\text{lmap grounding\_of\_state } Sts)$ 
       $- sr.Rf(\text{Liminf\_llist}(\text{lmap grounding\_of\_state } Sts))$ 

    have  $\text{ground\_ground\_Liminf: is\_ground\_clss}(\text{Liminf\_llist}(\text{lmap grounding\_of\_state } Sts))$ 
      using  $\text{Liminf\_grounding\_of\_state\_ground}$  unfolding  $\text{is\_ground\_clss\_def}$  by auto

    have  $\text{ground\_cc: is\_ground\_clss}(\text{set\_mset } ?CC)$ 
      using  $\text{a ground\_ground\_Liminf is\_ground\_clss\_def}$  by auto

    have  $\text{ground\_da: is\_ground\_cls } ?DA$ 
  }

```

```

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 fair unfolding fair_state_seq_def
  by (metis (no_types, lifting) Un_empty_left ground_ns_in_ground_limit_st a 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 ∃σ. ord_resolve S_Q CAs ?DA AAs As σ ?E
  by (rule ground_ord_resolve_imp_ord_resolve[OF ground_da_ground_cas
    gr.ground_resolution_with_selection_axioms CAs_p[THEN conjunct1]])
then obtain σ where
  σ_p: ord_resolve S_Q 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[unfolded S_Q_def] 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 subsequeq_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
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 inference_system.inferences_between_def

```

```

apply (rule_tac x = Infer (mset CAs') DA' E' in image_eqI)
subgoal by auto
subgoal
  unfolding infer_from_def
  by (rule ord_resolve_rename.cases[OF s_p(4)]) (use s_p(4) C'_p(3,5) j_p'(2) in force)
done
then show ?thesis
  using C'_p(4) by auto
qed
then have E' ∈ class_of_state (lnth Sts j)
  using j_p' by auto
then have ?E ∈ grounding_of_state (lnth Sts j)
  using s_p(7) s_p(3) unfolding grounding_of_class_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 (smt (verit))
then have γ ∈ sr_ext.Ri (Liminf_llist (lmap grounding_of_state Sts))
  using sr_ext.Ri_limit_Sup[of Gs] 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_up_to_iff[of gr.ord_Γ]
  unfolding sr_ext.Ri_def by auto
qed

```

corollary RP_complete_if_fair:

assumes

```

deriv: chain (↝) Sts and
fair: fair_state_seq Sts and
empty_Q0: Q_of_state (lhd 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))
  using unsat sr_ext.sat_limit_iff[OF ground_derive_chain] chain_not_lnull_deriv by fastforce
moreover have sr.saturated upto (Liminf_llist (lmap grounding_of_state Sts))
  by (rule RP_saturated_if_fair[OF deriv fair empty_Q0, 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[OF deriv fair] by auto
qed

```

end

end

end