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

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

lemma *map2_empty_iff*[*simp*]: $map2\ f\ xs\ ys = [] \longleftrightarrow xs = [] \vee ys = []$
⟨*proof*⟩

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

lemma *map2_tl*: $length\ t = length\ s \Longrightarrow map2\ f\ (tl\ t)\ (tl\ s) = tl\ (map2\ f\ t\ s)$
⟨*proof*⟩

lemma *map_zip_assoc*:
$map\ f\ (zip\ (zip\ xs\ ys)\ zs) = map\ (\lambda(x,\ y,\ z).\ f\ ((x,\ y),\ z))\ (zip\ xs\ (zip\ ys\ zs))$
⟨*proof*⟩

lemma *set_map2_ex*:
assumes $length\ t = length\ s$
shows $set\ (map2\ f\ s\ t) = \{x.\ \exists i < length\ t.\ x = f\ (s\ !\ i)\ (t\ !\ i)\}$
⟨*proof*⟩

end

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*: $i < llength\ (ltake\ k\ Xs) \longleftrightarrow i < k \wedge i < llength\ Xs$
⟨*proof*⟩

3.2 Supremum

definition *Sup_llist* :: $'a\ set\ llist \Rightarrow 'a\ set$ **where**

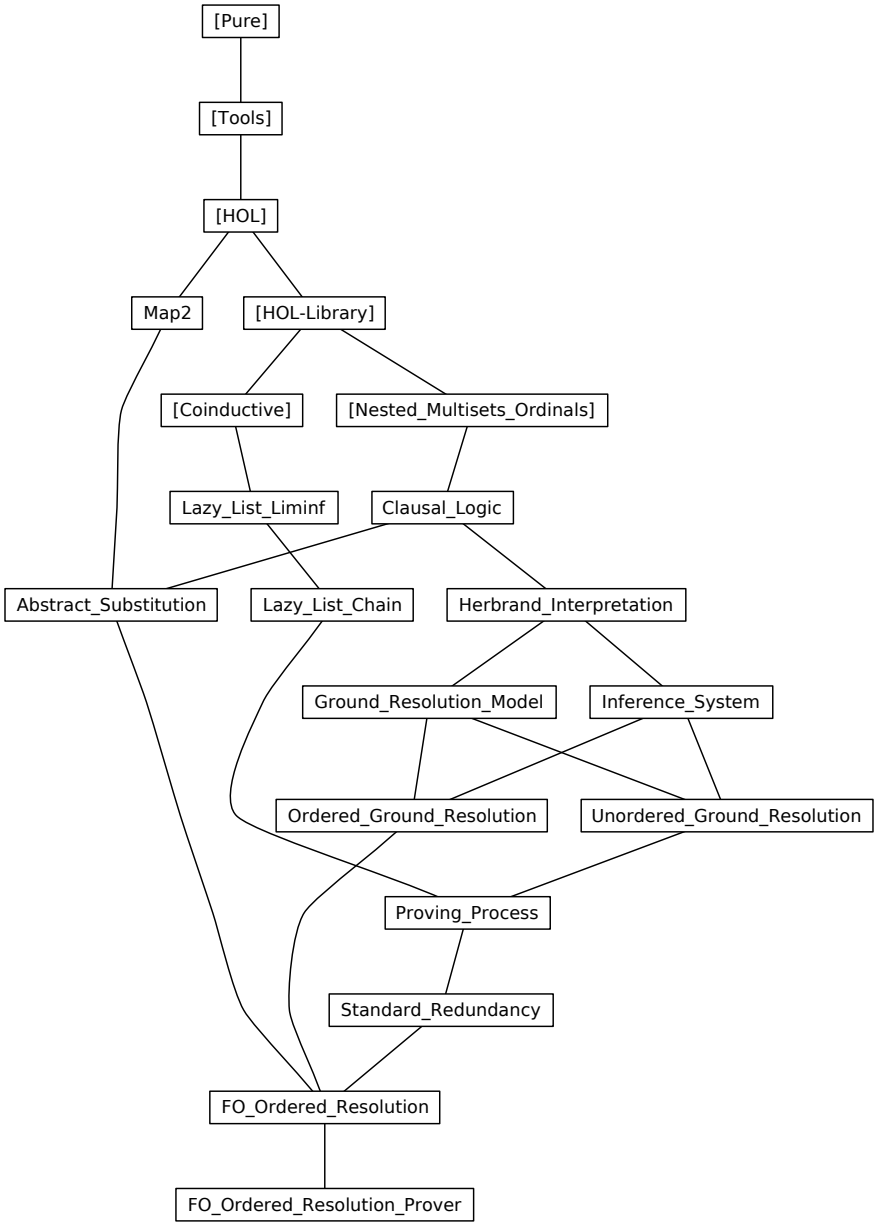

Figure 1: Theory dependency graph

Sup_llist Xs = (⋃ *i* ∈ {*i*. *enat i* < *llength Xs*}. *lnth Xs i*)

lemma *lnth_subset_Sup_llist*: *enat i* < *llength Xs* ⟹ *lnth Xs i* ⊆ *Sup_llist Xs*
⟨*proof*⟩

lemma *Sup_llist_imp_exists_index*: *x* ∈ *Sup_llist Xs* ⟹ ∃ *i*. *enat i* < *llength Xs* ∧ *x* ∈ *lnth Xs i*
⟨*proof*⟩

lemma *exists_index_imp_Sup_llist*: *enat i* < *llength Xs* ⟹ *x* ∈ *lnth Xs i* ⟹ *x* ∈ *Sup_llist Xs*
⟨*proof*⟩

lemma *Sup_llist_LNil*[*simp*]: *Sup_llist LNil* = {}
⟨*proof*⟩

lemma *Sup_llist_LCons*[*simp*]: *Sup_llist* (*LCons X Xs*) = *X* ∪ *Sup_llist Xs*
⟨*proof*⟩

lemma *lhd_subset_Sup_llist*: ¬ *lnull Xs* ⟹ *lhd Xs* ⊆ *Sup_llist Xs*
⟨*proof*⟩

3.3 Supremum up-to

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

lemma *Sup_upto_llist_eq_Sup_llist_ltake*: *Sup_upto_llist Xs j* = *Sup_llist* (*ltake* (*eSuc j*) *Xs*)
⟨*proof*⟩

lemma *Sup_upto_llist_enat_0*[*simp*]:
Sup_upto_llist Xs (*enat 0*) = (*if lnull Xs then* {} *else lhd Xs*)
⟨*proof*⟩

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 lnth Xs* (*Suc j*) *else* {})
⟨*proof*⟩

lemma *Sup_upto_llist_infinity*[*simp*]: *Sup_upto_llist Xs* ∞ = *Sup_llist Xs*
⟨*proof*⟩

lemma *Sup_upto_llist_0*[*simp*]: *Sup_upto_llist Xs 0* = (*if lnull Xs then* {} *else lhd Xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *Sup_upto_llist_mono*[*simp*]: *j* ≤ *k* ⟹ *Sup_upto_llist Xs j* ⊆ *Sup_upto_llist Xs k*
⟨*proof*⟩

lemma *Sup_upto_llist_subset_Sup_llist*: *Sup_upto_llist Xs j* ⊆ *Sup_llist Xs*
⟨*proof*⟩

lemma *elem_Sup_llist_imp_Sup_upto_llist*:
x ∈ *Sup_llist Xs* ⟹ ∃ *j* < *llength Xs*. *x* ∈ *Sup_upto_llist Xs* (*enat j*)
⟨*proof*⟩

lemma *lnth_subset_Sup_upto_llist*: *j* < *llength Xs* ⟹ *lnth Xs j* ⊆ *Sup_upto_llist Xs j*
⟨*proof*⟩

lemma *finite_Sup_llist_imp_Sup_upto_llist*:

assumes *finite X* **and** $X \subseteq$ *Sup_llist Xs*
shows $\exists k.$ $X \subseteq$ *Sup_upto_llist Xs* (*enat k*)
⟨*proof*⟩

3.4 Liminf

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

lemma *Liminf_llist_LNil*[*simp*]: *Liminf_llist LNil* = {}
⟨*proof*⟩

lemma *Liminf_llist_LCons*:
Liminf_llist (*LCons X Xs*) = (*if lnull Xs then X else Liminf_llist Xs*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *lfinite_Liminf_llist*: *lfinite Xs* ⟹ *Liminf_llist Xs* = (*if lnull Xs then* {} *else llast Xs*)
⟨*proof*⟩

lemma *Liminf_llist_ltl*: ¬ *lnull* (*ltl Xs*) ⟹ *Liminf_llist Xs* = *Liminf_llist* (*ltl Xs*)
⟨*proof*⟩

lemma *Liminf_llist_subset_Sup_llist*: *Liminf_llist Xs* ⊆ *Sup_llist Xs*
⟨*proof*⟩

lemma *image_Liminf_llist_subset*: *f* ‘ *Liminf_llist Ns* ⊆ *Liminf_llist* (*lmap* ((‘) *f*) *Ns*)
⟨*proof*⟩

lemma *Liminf_llist_imp_exists_index*:
x ∈ *Liminf_llist Xs* ⟹ $\exists i.$ *enat i* < *llength Xs* ∧ *x* ∈ *lnth Xs i*
⟨*proof*⟩

lemma *not_Liminf_llist_imp_exists_index*:
¬ *lnull Xs* ⟹ *x* ∉ *Liminf_llist Xs* ⟹ *enat i* < *llength Xs* ⟹
($\exists j.$ $i \leq j \wedge$ *enat j* < *llength Xs* ∧ *x* ∉ *lnth Xs j*)
⟨*proof*⟩

lemma *finite_subset_Liminf_llist_imp_exists_index*:
assumes
nnil: ¬ *lnull Xs* **and**
fin: *finite X* **and**
in_lim: *X* ⊆ *Liminf_llist Xs*
shows $\exists i.$ *enat i* < *llength Xs* ∧ *X* ⊆ ($\bigcap j \in \{j.$ $i \leq j \wedge$ *enat j* < *llength Xs*$\}.$ *lnth Xs j*)
⟨*proof*⟩

lemma *Liminf_llist_lmap_image*:
assumes *f_inj*: *inj_on f* (*Sup_llist* (*lmap g xs*))
shows *Liminf_llist* (*lmap* (λ*x*. *f* ‘ *g x*) *xs*) = *f* ‘ *Liminf_llist* (*lmap g xs*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *Liminf_llist_lmap_union*:
assumes ∀ *x* ∈ *lset xs*. ∀ *Y* ∈ *lset xs*. *g x* ∩ *h Y* = {}
shows *Liminf_llist* (*lmap* (λ*x*. *g x* ∪ *h x*) *xs*) =
Liminf_llist (*lmap g xs*) ∪ *Liminf_llist* (*lmap h xs*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *Liminf_set_filter_commute*:
Liminf_llist (*lmap* (λ*X*. {*x* ∈ *X*. *p x*}) *Xs*) = {*x* ∈ *Liminf_llist Xs*. *p x*}
⟨*proof*⟩

3.5 Liminf up-to

definition *Liminf_upto_llist* :: *'a set llist* ⇒ *enat* ⇒ *'a set* **where**

Liminf_upto_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_upto_llist_eq_Liminf_llist_ltake*:
Liminf_upto_llist Xs j = *Liminf_llist* (*ltake* (*eSuc j*) *Xs*)
⟨*proof*⟩

lemma *Liminf_upto_llist_enat*[*simp*]:
Liminf_upto_llist Xs (*enat k*) =
(*if enat k* < *llength Xs then lnth Xs k else if lnull Xs then* {} *else llast Xs*)
⟨*proof*⟩

lemma *Liminf_upto_llist_infinity*[*simp*]: *Liminf_upto_llist Xs* ∞ = *Liminf_llist Xs*
⟨*proof*⟩

lemma *Liminf_upto_llist_0*[*simp*]:
Liminf_upto_llist Xs 0 = (*if lnull Xs then* {} *else lhd Xs*)
⟨*proof*⟩

lemma *Liminf_upto_llist_eSuc*[*simp*]:
Liminf_upto_llist Xs (*eSuc j*) =
(*case j of*
enat k ⇒ *Liminf_upto_llist Xs* (*enat* (*Suc k*))
| ∞ ⇒ *Liminf_llist Xs*)
⟨*proof*⟩

lemma *elem_Liminf_llist_imp_Liminf_upto_llist*:
x ∈ *Liminf_llist Xs* ⟹
∃ *i* < *llength Xs*. ∀ *j*. *i* ≤ *j* ∧ *j* < *llength Xs* ⟶ *x* ∈ *Liminf_upto_llist Xs* (*enat j*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *chain_length_pos*: *chain R xs* ⟹ *llength xs* > *0*
⟨*proof*⟩

lemma *chain_ldropn*:
assumes *chain R xs* **and** *enat n* < *llength xs*
shows *chain R* (*ldropn n xs*)
⟨*proof*⟩

lemma *inf_chain_ldropn_chain*: *chain R xs* ⟹ ¬ *lfinite xs* ⟹ *chain R* (*ldropn n xs*)
⟨*proof*⟩

lemma *inf_chain_ltl_chain*: *chain R xs* ⟹ ¬ *lfinite xs* ⟹ *chain R* (*ltl xs*)
⟨*proof*⟩

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*⟩

lemma *infinite_chain_lnth_rel*:
assumes ¬ *lfinite c* **and** *chain r c*
shows *r* (*lnth c i*) (*lnth c* (*Suc i*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *chain_lmap*:
assumes ∀ *x y*. *R x y* ⟶ *R*′ (*f x*) (*f y*) **and** *chain R xs*
shows *chain R*′ (*lmap f xs*)
⟨*proof*⟩

lemma *chain_mono*:
assumes ∀ *x y*. *R x y* ⟶ *R*′ *x y* **and** *chain R xs*
shows *chain R*′ *xs*
⟨*proof*⟩

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*⟩

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*⟩

lemma *lfinite_chain_imp_rtranclp_lhd_llast*: *lfinite xs* ⟹ *chain R xs* ⟹ *R*** (*lhd xs*) (*llast xs*)
⟨*proof*⟩

lemma *tranclp_imp_exists_finite_chain_list*:

$R^{++}\ x\ y \Longrightarrow \exists xs.\ chain\ R\ (llist\_of\ (x \#\ xs\ @\ [y]))$
⟨*proof*⟩

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* = [] ∧ *lnull ys*)
⟨*proof*⟩

lemma *lhd_prepend*[*simp*]: *lhd* (*prepend xs ys*) = (*if xs* ≠ [] *then hd xs else lhd ys*)
⟨*proof*⟩

lemma *prepend_LNil*[*simp*]: *prepend xs LNil* = *llist_of xs*
⟨*proof*⟩

lemma *lfinite_prepend*[*simp*]: *lfinite* (*prepend xs ys*) ⟷ *lfinite ys*
⟨*proof*⟩

lemma *llength_prepend*[*simp*]: *llength* (*prepend xs ys*) = *length xs* + *llength ys*
⟨*proof*⟩

lemma *llast_prepend*[*simp*]: ¬ *lnull ys* ⟹ *llast* (*prepend xs ys*) = *llast ys*
⟨*proof*⟩

lemma *prepend_prepend*: *prepend xs* (*prepend ys zs*) = *prepend* (*xs* @ *ys*) *zs*
⟨*proof*⟩

lemma *chain_prepend*:
chain R (*llist_of zs*) ⟹ *last zs* = *lhd xs* ⟹ *chain R xs* ⟹ *chain R* (*prepend zs* (*ltl xs*))
⟨*proof*⟩

lemma *lmap_prepend*[*simp*]: *lmap f* (*prepend xs ys*) = *prepend* (*map f xs*) (*lmap f ys*)
⟨*proof*⟩

lemma *lset_prepend*[*simp*]: *lset* (*prepend xs ys*) = *set xs* ∪ *lset ys*
⟨*proof*⟩

lemma *prepend_LCons*: *prepend xs* (*LCons y ys*) = *prepend* (*xs* @ [*y*]) *ys*
⟨*proof*⟩

lemma *lnth_prepend*:
lnth (*prepend xs ys*) *i* = (*if i* < *length xs then nth xs i else lnth ys* (*i* − *length xs*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

coinductive *emb* :: *'a llist* ⇒ *'a llist* ⇒ *bool* **where**

lfinite xs $\Longrightarrow$ *emb LNil xs*
| *emb xs ys* $\Longrightarrow$ *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* $\longleftrightarrow$ *lfinite xs*
⟨*proof*⟩

inductive *prepend_cong1* **for** *X* **where**
prepend_cong1_base: *X xs* $\Longrightarrow$ *prepend_cong1 X xs*
| *prepend_cong1_prepend*: *prepend_cong1 X ys* $\Longrightarrow$ *prepend_cong1 X* (*prepend xs ys*)

lemma *prepend_cong1_alt*: *prepend_cong1 X xs* $\longleftrightarrow$ ($\exists$ *ys zs*. *xs* = *prepend ys zs* $\wedge$ *X zs*)
⟨*proof*⟩

lemma *emb_prepend_coinduct_cong*[*rotated*, *case_names emb*]:
assumes ($\bigwedge$*x1 x2*. *X x1 x2* $\Longrightarrow$
($\exists$ *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* $\Longrightarrow$ *?bisim x1 x2*)
shows *X x1 x2* $\Longrightarrow$ *emb x1 x2*
⟨*proof*⟩

lemma *emb_prepend*: *emb xs ys* $\Longrightarrow$ *emb xs* (*prepend zs ys*)
⟨*proof*⟩

lemma *prepend_cong1_emb*: *prepend_cong1* (*emb xs*) *ys* = *emb xs ys*
⟨*proof*⟩

lemma *prepend_cong_distrib*:
prepend_cong1 (*P* $\sqcup$ *Q*) *xs* $\longleftrightarrow$ *prepend_cong1 P xs* $\vee$ *prepend_cong1 Q xs*
⟨*proof*⟩

lemma *emb_prepend_coinduct_aux*[*case_names emb*]:
assumes *X x1 x2* ($\bigwedge$*x1 x2*. *X x1 x2* $\Longrightarrow$
($\exists$ *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*
⟨*proof*⟩

lemma *emb_prepend_coinduct*[*rotated*, *case_names emb*]:
assumes ($\bigwedge$*x1 x2*. *X x1 x2* $\Longrightarrow$
($\exists$ *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* $\Longrightarrow$ *emb x1 x2*
⟨*proof*⟩

context
begin

private coinductive *chain'* **for** *R* **where**
chain' R (*LCons x LNil*)
| *chain R* (*llist_of* (*x # zs @* [*lhd xs*])) $\Longrightarrow$
chain' R xs $\Longrightarrow$ *chain' R* (*LCons x* (*prepend zs xs*))

private lemma *chain_imp_chain'*: *chain R xs* $\Longrightarrow$ *chain' R xs*

⟨*proof*⟩ **lemma** *chain′_imp_chain*: *chain′ R xs* ⟹ *chain R xs*
⟨*proof*⟩ **lemma** *chain_chain′*: *chain* = *chain′*
⟨*proof*⟩

lemma *chain_prepend_coinduct*[*case_names chain*]:
X x ⟹ (⋀*x*. *X x* ⟹
(∃ *z*. *x* = *LCons z LNil*) ∨
(∃ *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*
⟨*proof*⟩

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*]))
⟨*proof*⟩ **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*))
⟨*proof*⟩ **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*))))
⟨*proof*⟩ **lemma** *lnull_wit*[*simp*]: *lnull* (*wit xs*) ⟷ *lnull xs*
⟨*proof*⟩ **lemma** *lhd_wit*[*simp*]: *chain* R^{++} *xs* ⟹ *lhd* (*wit xs*) = *lhd xs*
⟨*proof*⟩ **lemma** *LNil_eq_iff_lnull*: *LNil* = *xs* ⟷ *lnull xs*
⟨*proof*⟩

lemma *emb_wit*[*simp*]: *chain* R^{++} *xs* ⟹ *emb xs* (*wit xs*)
⟨*proof*⟩ **lemma** *lfinite_wit*[*simp*]:
assumes *chain* R^{++} *xs*
shows *lfinite* (*wit xs*) ⟷ *lfinite xs*
⟨*proof*⟩ **lemma** *llast_wit*[*simp*]:
assumes *chain* R^{++} *xs*
shows *llast* (*wit xs*) = *llast xs*
⟨*proof*⟩

lemma *chain_tranclp_imp_exists_chain*:
chain R^{++} *xs* ⟹
∃ *ys*. *chain R ys* ∧ *emb xs ys* ∧ *lhd ys* = *lhd xs* ∧ *llast ys* = *llast xs*
⟨*proof*⟩

lemma *emb_lset_mono*[*rotated*]: *x* ∈ *lset xs* ⟹ *emb xs ys* ⟹ *x* ∈ *lset ys*
⟨*proof*⟩

lemma *emb_Ball_lset_antimono*:
assumes *emb Xs Ys*
shows ∀ *Y* ∈ *lset Ys*. *x* ∈ *Y* ⟹ ∀ *X* ∈ *lset Xs*. *x* ∈ *X*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *emb_Liminf_llist_mono_aux*:
assumes *emb Xs Ys* **and** ¬ *lfinite Xs* **and** ¬ *lfinite Ys* **and** ∀ *j*≥*i*. *x* ∈ *lnth Ys j*
shows ∀ *j*≥*i*. *x* ∈ *lnth Xs j*
⟨*proof*⟩

lemma *emb_Liminf_llist_infinite*:
assumes *emb Xs Ys* **and** ¬ *lfinite Xs*
shows *Liminf_llist Ys* ⊆ *Liminf_llist Xs*
⟨*proof*⟩

lemma *emb_lmap*: *emb xs ys* ⟹ *emb* (*lmap f xs*) (*lmap f ys*)
⟨*proof*⟩

end

lemma *chain_inf_llist_if_infinite_chain_function*:
assumes ∀ *i*. *r* (*f* (*Suc i*)) (*f i*)
shows ¬ *lfinite* (*inf_llist f*) ∧ *chain* r^{-1-1} (*inf_llist f*)
⟨*proof*⟩

lemma *infinite_chain_function_iff_infinite_chain_llist*:
(∃ *f*. ∀ *i*. *r* (*f* (*Suc i*)) (*f i*)) ⟷ (∃ *c*. ¬ *lfinite c* ∧ *chain* r^{-1-1} *c*)
⟨*proof*⟩

lemma *wfP_iff_no_infinite_down_chain_llist*: *wfP r* ⟷ (∄ *c*. ¬ *lfinite c* ∧ *chain* r^{-1-1} *c*)
⟨*proof*⟩

4.3 Full Chains

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

lemma
full_chain_LNil[*simp*]: ¬ *full_chain R LNil* **and**
full_chain_not_lnull: *full_chain R xs* ⟹ ¬ *lnull xs*
⟨*proof*⟩

lemma *full_chain_ldropn*:
assumes *full*: *full_chain R xs* **and** *enat n* < *llength xs*
shows *full_chain R* (*ldropn n xs*)
⟨*proof*⟩

lemma *full_chain_iff_chain*:
full_chain R xs ⟷ *chain R xs* ∧ (*lfinite xs* ⟶ (∀ *y*. ¬ *R* (*llast xs*) *y*))
⟨*proof*⟩

lemma *full_chain_imp_chain*: *full_chain R xs* ⟹ *chain R xs*
⟨*proof*⟩

lemma *full_chain_length_pos*: *full_chain R xs* ⟹ *llength xs* > *0*
⟨*proof*⟩

lemma *full_chain_lnth_rel*:
full_chain R xs ⟹ *enat* (*Suc j*) < *llength xs* ⟹ *R* (*lnth xs j*) (*lnth xs* (*Suc j*))
⟨*proof*⟩

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*⟩

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*⟩

end

5 Clausal Logic

theory *Clausal_Logic*
  imports *Nested_Multisets_Ordinals.Multiset_More*
begin

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

5.1 Literals

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

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

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

lemma *Pos_atm_of_iff*[*simp*]: *Pos* (*atm_of L*) = *L* ⟷ *is_pos L*
  ⟨*proof*⟩

lemma *Neg_atm_of_iff*[*simp*]: *Neg* (*atm_of L*) = *L* ⟷ *is_neg L*
  ⟨*proof*⟩

lemma *set_literal_atm_of*: *set_literal L* = {*atm_of L*}
  ⟨*proof*⟩

lemma *ex_lit_cases*: (∃ *L*. *P L*) ⟷ (∃ *A*. *P* (*Pos A*) ∨ *P* (*Neg A*))
  ⟨*proof*⟩

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

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

instance ⟨*proof*⟩

end

lemma
  uminus_Pos[*simp*]: − *Pos A* = *Neg A* **and**
  uminus_Neg[*simp*]: − *Neg A* = *Pos A*
  ⟨*proof*⟩

lemma *atm_of_uminus*[*simp*]: *atm_of* (−*L*) = *atm_of L*
  ⟨*proof*⟩

lemma *uminus_of_uminus_id*[*simp*]: $-\ (-\ (x :: 'v\ literal)) = x$
⟨*proof*⟩

lemma *uminus_not_id*[*simp*]: $x \neq -\ (x :: 'v\ literal)$
⟨*proof*⟩

lemma *uminus_not_id'*[*simp*]: $-\ x \neq (x :: 'v\ literal)$
⟨*proof*⟩

lemma *uminus_eq_inj*[*iff*]: $-\ (a :: 'v\ literal) = -\ b \longleftrightarrow a = b$
⟨*proof*⟩

lemma *uminus_lit_swap*: $(a :: 'a\ literal) = -\ b \longleftrightarrow -\ a = b$
⟨*proof*⟩

lemma *is_pos_neg_not_is_pos*: $is\_pos\ (-\ L) \longleftrightarrow \neg\ is\_pos\ L$
⟨*proof*⟩

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

definition *less_literal* :: $'a\ literal \Rightarrow 'a\ literal \Rightarrow bool$ **where**
$less\_literal\ L\ M \longleftrightarrow atm\_of\ L < atm\_of\ M \vee atm\_of\ L \leq atm\_of\ M \wedge is\_neg\ L < is\_neg\ M$

definition *less_eq_literal* :: $'a\ literal \Rightarrow 'a\ literal \Rightarrow bool$ **where**
$less\_eq\_literal\ L\ M \longleftrightarrow atm\_of\ L < atm\_of\ M \vee atm\_of\ L \leq atm\_of\ M \wedge is\_neg\ L \leq is\_neg\ M$

instance
⟨*proof*⟩

end

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

instance
⟨*proof*⟩

end

lemma *pos_less_neg*[*simp*]: $Pos\ A < Neg\ A$
⟨*proof*⟩

lemma *pos_less_pos_iff*[*simp*]: $Pos\ A < Pos\ B \longleftrightarrow A < B$
⟨*proof*⟩

lemma *pos_less_neg_iff*[*simp*]: $Pos\ A < Neg\ B \longleftrightarrow A \leq B$
⟨*proof*⟩

lemma *neg_less_pos_iff*[*simp*]: $Neg\ A < Pos\ B \longleftrightarrow A < B$
⟨*proof*⟩

lemma *neg_less_neg_iff*[*simp*]: $Neg\ A < Neg\ B \longleftrightarrow A < B$
⟨*proof*⟩

lemma *pos_le_neg*[*simp*]: $Pos\ A \leq Neg\ A$
⟨*proof*⟩

lemma *pos_le_pos_iff*[*simp*]: $Pos\ A \leq Pos\ B \longleftrightarrow A \leq B$
⟨*proof*⟩

lemma *pos_le_neg_iff*[*simp*]: $Pos\ A \leq Neg\ B \longleftrightarrow A \leq B$
⟨*proof*⟩

lemma *neg_le_pos_iff*[*simp*]: *Neg A* ≤ *Pos B* ⟷ *A* < *B*
⟨*proof*⟩

lemma *neg_le_neg_iff*[*simp*]: *Neg A* ≤ *Neg B* ⟷ *A* ≤ *B*
⟨*proof*⟩

lemma *leq_imp_less_eq_atm_of*: *L* ≤ *M* ⟹ *atm_of L* ≤ *atm_of M*
⟨*proof*⟩

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

instance
⟨*proof*⟩

end

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

instance
⟨*proof*⟩

end

5.2 Clauses

Clauses are (finite) multisets of literals.

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

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

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

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

lemma *Max_in_lits*: *C* ≠ {#} ⟹ *Max_mset C* ∈# *C*
⟨*proof*⟩

lemma *Max_atm_of_set_mset_commute*: *C* ≠ {#} ⟹ *Max* (*atm_of* ‘ *set_mset C*) = *atm_of* (*Max_mset C*)
⟨*proof*⟩

lemma *Max_pos_neg_less_multiset*:
assumes *max*: *Max_mset C* = *Pos A* **and** *neg*: *Neg A* ∈# *D*
shows *C* < *D*
⟨*proof*⟩

lemma *pos_Max_imp_neg_notin*: *Max_mset C* = *Pos A* ⟹ *Neg A* ∉# *C*
⟨*proof*⟩

lemma *less_eq_Max_lit*: *C* ≠ {#} ⟹ *C* ≤ *D* ⟹ *Max_mset C* ≤ *Max_mset D*
⟨*proof*⟩

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

lemma *atms_of_empty*[*simp*]: *atms_of* {#} = {}
⟨*proof*⟩

lemma *atms_of_singleton*[*simp*]: *atms_of* {#*L*#} = {*atm_of L*}
⟨*proof*⟩

lemma *atms_of_add_mset*[*simp*]: *atms_of* (*add_mset a A*) = *insert* (*atm_of a*) (*atms_of A*)
⟨*proof*⟩

lemma *atms_of_union_mset*[*simp*]: *atms_of* (*A* ∪# *B*) = *atms_of A* ∪ *atms_of B*
⟨*proof*⟩

lemma *finite_atms_of*[*iff*]: *finite* (*atms_of C*)
⟨*proof*⟩

lemma *atm_of_lit_in_atms_of*: *L* ∈# *C* ⟹ *atm_of L* ∈ *atms_of C*
⟨*proof*⟩

lemma *atms_of_plus*[*simp*]: *atms_of* (*C* + *D*) = *atms_of C* ∪ *atms_of D*
⟨*proof*⟩

lemma *in_atms_of_minusD*: *x* ∈ *atms_of* (*A* − *B*) ⟹ *x* ∈ *atms_of A*
⟨*proof*⟩

lemma *pos_lit_in_atms_of*: *Pos A* ∈# *C* ⟹ *A* ∈ *atms_of C*
⟨*proof*⟩

lemma *neg_lit_in_atms_of*: *Neg A* ∈# *C* ⟹ *A* ∈ *atms_of C*
⟨*proof*⟩

lemma *atm_imp_pos_or_neg_lit*: *A* ∈ *atms_of C* ⟹ *Pos A* ∈# *C* ∨ *Neg A* ∈# *C*
⟨*proof*⟩

lemma *atm_iff_pos_or_neg_lit*: *A* ∈ *atms_of L* ⟷ *Pos A* ∈# *L* ∨ *Neg A* ∈# *L*
⟨*proof*⟩

lemma *atm_of_eq_atm_of*: *atm_of L* = *atm_of L′* ⟷ (*L* = *L′* ∨ *L* = −*L′*)
⟨*proof*⟩

lemma *atm_of_in_atm_of_set_iff_in_set_or_uminus_in_set*: *atm_of L* ∈ *atm_of* ‘ *I* ⟷ (*L* ∈ *I* ∨ −*L* ∈ *I*)
⟨*proof*⟩

lemma *lits_subseteq_imp_atms_subseteq*: *set_mset C* ⊆ *set_mset D* ⟹ *atms_of C* ⊆ *atms_of D*
⟨*proof*⟩

lemma *atms_empty_iff_empty*[*iff*]: *atms_of C* = {} ⟷ *C* = {#}
⟨*proof*⟩

lemma
atms_of_poss[*simp*]: *atms_of* (*poss AA*) = *set_mset AA* **and**
atms_of_negs[*simp*]: *atms_of* (*negs AA*) = *set_mset AA*
⟨*proof*⟩

lemma *less_eq_Max_atms_of*: *C* ≠ {#} ⟹ *C* ≤ *D* ⟹ *Max* (*atms_of C*) ≤ *Max* (*atms_of D*)
⟨*proof*⟩

lemma *le_multiset_Max_in_imp_Max*:
Max (*atms_of D*) = *A* ⟹ *C* ≤ *D* ⟹ *A* ∈ *atms_of C* ⟹ *Max* (*atms_of C*) = *A*
⟨*proof*⟩

lemma *atm_of_Max_lit*[*simp*]: *C* ≠ {#} ⟹ *atm_of* (*Max_mset C*) = *Max* (*atms_of C*)
⟨*proof*⟩

lemma *Max_lit_eq_pos_or_neg_Max_atm*:
C ≠ {#} ⟹ *Max_mset C* = *Pos* (*Max* (*atms_of C*)) ∨ *Max_mset C* = *Neg* (*Max* (*atms_of C*))
⟨*proof*⟩

lemma *atms_less_imp_lit_less_pos*: $(\bigwedge B.\ B \in atms\_of\ C \Longrightarrow B < A) \Longrightarrow L \in\# C \Longrightarrow L < Pos\ A$
⟨*proof*⟩

lemma *atms_less_eq_imp_lit_less_eq_neg*: $(\bigwedge B.\ B \in atms\_of\ C \Longrightarrow B \leq A) \Longrightarrow L \in\# C \Longrightarrow L \leq Neg\ A$
⟨*proof*⟩

end

6 Herbrand Intepretation

theory *Herbrand_Interpretation*
imports *Clausal_Logic*
begin

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

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

type-synonym *'a interp* = *'a set*

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

lemma *true_lit_simps*[*simp*]:
$I \models l\ Pos\ A \longleftrightarrow A \in I$
$I \models l\ Neg\ A \longleftrightarrow A \notin I$
⟨*proof*⟩

lemma *true_lit_iff*[*iff*]: $I \models l\ L \longleftrightarrow (\exists A.\ L = Pos\ A \wedge A \in I \vee L = Neg\ A \wedge A \notin I)$
⟨*proof*⟩

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

lemma *true_cls_empty*[*iff*]: $\neg\ I \models \{\#\}$
⟨*proof*⟩

lemma *true_cls_singleton*[*iff*]: $I \models \{\#L\#\} \longleftrightarrow I \models l\ L$
⟨*proof*⟩

lemma *true_cls_add_mset*[*iff*]: $I \models add\_mset\ C\ D \longleftrightarrow I \models l\ C \vee I \models D$
⟨*proof*⟩

lemma *true_cls_union*[*iff*]: $I \models C + D \longleftrightarrow I \models C \vee I \models D$
⟨*proof*⟩

lemma *true_cls_mono*: $set\_mset\ C \subseteq set\_mset\ D \Longrightarrow I \models C \Longrightarrow I \models D$
⟨*proof*⟩

lemma
assumes $I \subseteq J$
shows
false_to_true_imp_ex_pos: $\neg\ I \models C \Longrightarrow J \models C \Longrightarrow \exists A \in J.\ Pos\ A \in\# C$ **and**
true_to_false_imp_ex_neg: $I \models C \Longrightarrow \neg\ J \models C \Longrightarrow \exists A \in J.\ Neg\ A \in\# C$
⟨*proof*⟩

lemma *true_cls_replicate_mset*[*iff*]: $I \models replicate\_mset\ n\ L \longleftrightarrow n \neq 0 \wedge I \models l\ L$
⟨*proof*⟩

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

lemma *neg_literal_notin_imp_true_cls*[*intro*]: *Neg A* $\in\#$ *C* $\Longrightarrow$ *A* $\notin$ *I* $\Longrightarrow$ *I* $\models$ *C*
⟨*proof*⟩

lemma *pos_neg_in_imp_true*: *Pos A* $\in\#$ *C* $\Longrightarrow$ *Neg A* $\in\#$ *C* $\Longrightarrow$ *I* $\models$ *C*
⟨*proof*⟩

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* $\models s$ {}
⟨*proof*⟩

lemma *true_clss_singleton*[*iff*]: *I* $\models s$ {*C*} $\longleftrightarrow$ *I* $\models$ *C*
⟨*proof*⟩

lemma *true_clss_insert*[*iff*]: *I* $\models s$ *insert C DD* $\longleftrightarrow$ *I* $\models$ *C* $\wedge$ *I* $\models s$ *DD*
⟨*proof*⟩

lemma *true_clss_union*[*iff*]: *I* $\models s$ *CC* $\cup$ *DD* $\longleftrightarrow$ *I* $\models s$ *CC* $\wedge$ *I* $\models s$ *DD*
⟨*proof*⟩

lemma *true_clss_Union*[*iff*]: *I* $\models s$ $\bigcup$ *CCC* $\longleftrightarrow$ ($\forall$ *CC* $\in$ *CCC*. *I* $\models s$ *CC*)
⟨*proof*⟩

lemma *true_clss_mono*: *DD* $\subseteq$ *CC* $\Longrightarrow$ *I* $\models s$ *CC* $\Longrightarrow$ *I* $\models s$ *DD*
⟨*proof*⟩

lemma *true_clss_mono_strong*: ($\forall$ *D* $\in$ *DD*. $\exists$ *C* $\in$ *CC*. *C* $\subseteq\#$ *D*) $\Longrightarrow$ *I* $\models s$ *CC* $\Longrightarrow$ *I* $\models s$ *DD*
⟨*proof*⟩

lemma *true_clss_subclause*: *C* $\subseteq\#$ *D* $\Longrightarrow$ *I* $\models s$ {*C*} $\Longrightarrow$ *I* $\models s$ {*D*}
⟨*proof*⟩

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

lemma *satisfiable_antimono*: *CC* $\subseteq$ *DD* $\Longrightarrow$ *satisfiable DD* $\Longrightarrow$ *satisfiable CC*
⟨*proof*⟩

lemma *unsatisfiable_mono*: *CC* $\subseteq$ *DD* $\Longrightarrow$ $\neg$ *satisfiable CC* $\Longrightarrow$ $\neg$ *satisfiable DD*
⟨*proof*⟩

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* $\models m$ {#}
⟨*proof*⟩

lemma *true_cls_mset_singleton*[*iff*]: *I* $\models m$ {#*C*#} $\longleftrightarrow$ *I* $\models$ *C*
⟨*proof*⟩

lemma *true_cls_mset_union*[*iff*]: *I* $\models m$ *CC* + *DD* $\longleftrightarrow$ *I* $\models m$ *CC* $\wedge$ *I* $\models m$ *DD*
⟨*proof*⟩

lemma *true_cls_mset_Union*[*iff*]: *I* $\models m$ $\sum_{\#}$ *CCC* $\longleftrightarrow$ ($\forall$ *CC* $\in\#$ *CCC*. *I* $\models m$ *CC*)
⟨*proof*⟩

lemma *true_cls_mset_add_mset*[*iff*]: *I* $\models m$ *add_mset C CC* $\longleftrightarrow$ *I* $\models$ *C* $\wedge$ *I* $\models m$ *CC*
⟨*proof*⟩

lemma *true_cls_mset_image_mset*[*iff*]: *I* $\models m$ *image_mset f A* $\longleftrightarrow$ ($\forall$ *x* $\in\#$ *A*. *I* $\models$ *f x*)
⟨*proof*⟩

lemma *true_cls_mset_mono*: $set\_mset\ DD \subseteq set\_mset\ CC \Longrightarrow I \models m\ CC \Longrightarrow I \models m\ DD$
⟨*proof*⟩

lemma *true_cls_mset_mono_strong*: $(\forall D \in\# DD.\ \exists C \in\# CC.\ C \subseteq\# D) \Longrightarrow I \models m\ CC \Longrightarrow I \models m\ DD$
⟨*proof*⟩

lemma *true_clss_set_mset*[*iff*]: $I \models s\ set\_mset\ CC \longleftrightarrow I \models m\ CC$
⟨*proof*⟩

lemma *true_clss_mset_set*[*simp*]: *finite* $CC \Longrightarrow I \models m\ mset\_set\ CC \longleftrightarrow I \models s\ CC$
⟨*proof*⟩

lemma *true_cls_mset_true_cls*: $I \models m\ CC \Longrightarrow C \in\# CC \Longrightarrow I \models C$
⟨*proof*⟩

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 $f :: nat \Rightarrow nat$
assumes *leq*: $\forall i.\ f\ (Suc\ i) \leq f\ i$
shows $\exists l.\ \forall l' \geq l.\ f\ l' = f\ (Suc\ l')$
⟨*proof*⟩

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** ‹·a› 67) **where**
subst_atm_abbrev ≡ *subst_atm*

abbreviation *comp_subst_abbrev* :: $'s \Rightarrow 's \Rightarrow 's$ (**infixl** ‹⊙› 67) **where**
comp_subst_abbrev ≡ *comp_subst*

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

definition *subst_atms* :: $'a\ set \Rightarrow 's \Rightarrow 'a\ set$ (**infixl** ‹·as› 67) **where**
$AA \cdot as\ \sigma = (\lambda A.\ A \cdot a\ \sigma)\ `\ AA$

definition *subst_atmss* :: $'a\ set\ set \Rightarrow 's \Rightarrow 'a\ set\ set$ (**infixl** ‹·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** ‹·al› 67) **where**
$As \cdot al\ \sigma = map\ (\lambda A.\ A \cdot a\ \sigma)\ As$

definition *subst_atm_mset* :: *'a multiset ⇒ 's ⇒ 'a multiset* (**infixl** ‹·*am*› *67*) **where**
AA ·am σ = image_mset (λ*A*. *A ·a σ*) *AA*

definition
subst_atm_mset_list :: *'a multiset list ⇒ 's ⇒ 'a multiset list* (**infixl** ‹·*aml*› *67*)
where
AAA ·aml σ = map (λ*AA*. *AA ·am σ*) *AAA*

definition
subst_atm_mset_lists :: *'a multiset list ⇒ 's list ⇒ 'a multiset list* (**infixl** ‹··*aml*› *67*)
where
AAs ··aml σs = map2 (·*am*) *AAs σs*

definition *subst_lit* :: *'a literal ⇒ 's ⇒ 'a literal* (**infixl** ‹·*l*› *67*) **where**
L ·l σ = map_literal (λ*A*. *A ·a σ*) *L*

lemma *atm_of_subst_lit*[*simp*]: *atm_of* (*L ·l σ*) = *atm_of L ·a σ*
⟨*proof*⟩

definition *subst_cls* :: *'a clause ⇒ 's ⇒ 'a clause* (**infixl** ‹·› *67*) **where**
AA · σ = image_mset (λ*A*. *A ·l σ*) *AA*

definition *subst_clss* :: *'a clause set ⇒ 's ⇒ 'a clause set* (**infixl** ‹·*cs*› *67*) **where**
AA ·cs σ = (λ*A*. *A · σ*) ‘ *AA*

definition *subst_cls_list* :: *'a clause list ⇒ 's ⇒ 'a clause list* (**infixl** ‹·*cl*› *67*) **where**
Cs ·cl σ = map (λ*A*. *A · σ*) *Cs*

definition *subst_cls_lists* :: *'a clause list ⇒ 's list ⇒ 'a clause list* (**infixl** ‹··*cl*› *67*) **where**
Cs ··cl σs = map2 (·) *Cs σs*

definition *subst_cls_mset* :: *'a clause multiset ⇒ 's ⇒ 'a clause multiset* (**infixl** ‹·*cm*› *67*) **where**
CC ·cm σ = image_mset (λ*A*. *A · σ*) *CC*

lemma *subst_cls_add_mset*[*simp*]: *add_mset L C · σ = add_mset* (*L ·l σ*) (*C · σ*)
⟨*proof*⟩

lemma *subst_cls_mset_add_mset*[*simp*]: *add_mset C CC ·cm σ = add_mset* (*C · σ*) (*CC ·cm σ*)
⟨*proof*⟩

definition *generalizes_atm* :: *'a ⇒ 'a ⇒ bool* **where**
generalizes_atm A B ⟷ (∃*σ*. *A ·a σ = B*)

definition *strictly_generalizes_atm* :: *'a ⇒ 'a ⇒ bool* **where**
strictly_generalizes_atm A B ⟷ *generalizes_atm A B* ∧ ¬ *generalizes_atm B A*

definition *generalizes_lit* :: *'a literal ⇒ 'a literal ⇒ bool* **where**
generalizes_lit L M ⟷ (∃*σ*. *L ·l σ = M*)

definition *strictly_generalizes_lit* :: *'a literal ⇒ 'a literal ⇒ bool* **where**
strictly_generalizes_lit L M ⟷ *generalizes_lit L M* ∧ ¬ *generalizes_lit M L*

definition *generalizes* :: *'a clause ⇒ 'a clause ⇒ bool* **where**
generalizes C D ⟷ (∃*σ*. *C · σ = D*)

definition *strictly_generalizes* :: *'a clause ⇒ 'a clause ⇒ bool* **where**
strictly_generalizes C D ⟷ *generalizes C D* ∧ ¬ *generalizes D C*

definition *subsumes* :: *'a clause ⇒ 'a clause ⇒ bool* **where**
subsumes C D ⟷ (∃*σ*. *C · σ* ⊆# *D*)

definition *strictly_subsumes* :: *'a clause ⇒ 'a clause ⇒ bool* **where**
strictly_subsumes C D ⟷ *subsumes C D* ∧ ¬ *subsumes D C*

definition *variants* :: *'a clause* ⇒ *'a clause* ⇒ *bool* **where**
variants C D ⟷ *generalizes C D* ∧ *generalizes D C*

definition *is_renaming* :: *'s* ⇒ *bool* **where**
is_renaming σ ⟷ (∃ *τ*. *σ* ⊙ *τ* = *id_subst*)

definition *is_renaming_list* :: *'s list* ⇒ *bool* **where**
is_renaming_list σs ⟷ (∀ *σ* ∈ *set σs*. *is_renaming σ*)

definition *inv_renaming* :: *'s* ⇒ *'s* **where**
inv_renaming σ = (*SOME τ*. *σ* ⊙ *τ* = *id_subst*)

definition *is_ground_atm* :: *'a* ⇒ *bool* **where**
is_ground_atm A ⟷ (∀ *σ*. *A* = *A* ·*a σ*)

definition *is_ground_atms* :: *'a set* ⇒ *bool* **where**
is_ground_atms AA = (∀ *A* ∈ *AA*. *is_ground_atm A*)

definition *is_ground_atm_list* :: *'a list* ⇒ *bool* **where**
is_ground_atm_list As ⟷ (∀ *A* ∈ *set As*. *is_ground_atm A*)

definition *is_ground_atm_mset* :: *'a multiset* ⇒ *bool* **where**
is_ground_atm_mset AA ⟷ (∀ *A*. *A* ∈# *AA* ⟶ *is_ground_atm A*)

definition *is_ground_lit* :: *'a literal* ⇒ *bool* **where**
is_ground_lit L ⟷ *is_ground_atm* (*atm_of L*)

definition *is_ground_cls* :: *'a clause* ⇒ *bool* **where**
is_ground_cls C ⟷ (∀ *L*. *L* ∈# *C* ⟶ *is_ground_lit L*)

definition *is_ground_clss* :: *'a clause set* ⇒ *bool* **where**
is_ground_clss CC ⟷ (∀ *C* ∈ *CC*. *is_ground_cls C*)

definition *is_ground_cls_list* :: *'a clause list* ⇒ *bool* **where**
is_ground_cls_list CC ⟷ (∀ *C* ∈ *set CC*. *is_ground_cls C*)

definition *is_ground_subst* :: *'s* ⇒ *bool* **where**
is_ground_subst σ ⟷ (∀ *A*. *is_ground_atm* (*A* ·*a σ*))

definition *is_ground_subst_list* :: *'s list* ⇒ *bool* **where**
is_ground_subst_list σs ⟷ (∀ *σ* ∈ *set σs*. *is_ground_subst σ*)

definition *grounding_of_cls* :: *'a clause* ⇒ *'a clause set* **where**
grounding_of_cls C = {*C* · *σ* |*σ*. *is_ground_subst σ*}

definition *grounding_of_clss* :: *'a clause set* ⇒ *'a clause set* **where**
grounding_of_clss CC = (⋃ *C* ∈ *CC*. *grounding_of_cls C*)

definition *is_unifier* :: *'s* ⇒ *'a set* ⇒ *bool* **where**
is_unifier σ AA ⟷ *card* (*AA* ·*as σ*) ≤ *1*

definition *is_unifiers* :: *'s* ⇒ *'a set set* ⇒ *bool* **where**
is_unifiers σ AAA ⟷ (∀ *AA* ∈ *AAA*. *is_unifier σ AA*)

definition *is_mgu* :: *'s* ⇒ *'a set set* ⇒ *bool* **where**
is_mgu σ AAA ⟷ *is_unifiers σ AAA* ∧ (∀ *τ*. *is_unifiers τ AAA* ⟶ (∃ *γ*. *τ* = *σ* ⊙ *γ*))

definition *is_imgu* :: *'s* ⇒ *'a set set* ⇒ *bool* **where**
is_imgu σ AAA ⟷ *is_unifiers σ AAA* ∧ (∀ *τ*. *is_unifiers τ AAA* ⟶ *τ* = *σ* ⊙ *τ*)

definition *var_disjoint* :: *'a clause list* ⇒ *bool* **where**
var_disjoint Cs ⟷

$(\forall \sigma s.\ length\ \sigma s = length\ Cs \longrightarrow (\exists \tau.\ \forall i < length\ Cs.\ \forall S.\ S \subseteq\#\ Cs\ !\ i \longrightarrow S \cdot \sigma s\ !\ i = S \cdot \tau))$

end

7.3 Substitution Lemmas

locale *substitution* = *substitution_ops subst_atm id_subst comp_subst*
  for
    subst_atm :: $'a \Rightarrow 's \Rightarrow 'a$ **and**
    id_subst :: $'s$ **and**
    comp_subst :: $'s \Rightarrow 's \Rightarrow 's$ +
  assumes
    subst_atm_id_subst[*simp*]: $A \cdot a\ id\_subst = A$ **and**
    subst_atm_comp_subst[*simp*]: $A \cdot a\ (\sigma \odot \tau) = (A \cdot a\ \sigma) \cdot a\ \tau$ **and**
    subst_ext: $(\bigwedge A.\ A \cdot a\ \sigma = A \cdot a\ \tau) \Longrightarrow \sigma = \tau$ **and**
    make_ground_subst: *is_ground_cls* $(C \cdot \sigma) \Longrightarrow \exists \tau.\ is\_ground\_subst\ \tau \wedge C \cdot \tau = C \cdot \sigma$ **and**
    wf_strictly_generalizes_atm: *wfP strictly_generalizes_atm*
begin

lemma *subst_ext_iff*: $\sigma = \tau \longleftrightarrow (\forall A.\ A \cdot a\ \sigma = A \cdot a\ \tau)$
  ⟨*proof*⟩

7.3.1 Identity Substitution

lemma *id_subst_comp_subst*[*simp*]: $id\_subst \odot \sigma = \sigma$
  ⟨*proof*⟩

lemma *comp_subst_id_subst*[*simp*]: $\sigma \odot id\_subst = \sigma$
  ⟨*proof*⟩

lemma *id_subst_comp_substs*[*simp*]: $replicate\ (length\ \sigma s)\ id\_subst \odot s\ \sigma s = \sigma s$
  ⟨*proof*⟩

lemma *comp_substs_id_subst*[*simp*]: $\sigma s \odot s\ replicate\ (length\ \sigma s)\ id\_subst = \sigma s$
  ⟨*proof*⟩

lemma *subst_atms_id_subst*[*simp*]: $AA \cdot as\ id\_subst = AA$
  ⟨*proof*⟩

lemma *subst_atmss_id_subst*[*simp*]: $AAA \cdot ass\ id\_subst = AAA$
  ⟨*proof*⟩

lemma *subst_atm_list_id_subst*[*simp*]: $As \cdot al\ id\_subst = As$
  ⟨*proof*⟩

lemma *subst_atm_mset_id_subst*[*simp*]: $AA \cdot am\ id\_subst = AA$
  ⟨*proof*⟩

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

lemma *subst_atm_mset_lists_id_subst*[*simp*]: $AAs \cdot\cdot aml\ replicate\ (length\ AAs)\ id\_subst = AAs$
  ⟨*proof*⟩

lemma *subst_lit_id_subst*[*simp*]: $L \cdot l\ id\_subst = L$
  ⟨*proof*⟩

lemma *subst_cls_id_subst*[*simp*]: $C \cdot id\_subst = C$
  ⟨*proof*⟩

lemma *subst_clss_id_subst*[*simp*]: $CC \cdot cs\ id\_subst = CC$
  ⟨*proof*⟩

lemma *subst_cls_list_id_subst*[*simp*]: $Cs \cdot cl\ id\_subst = Cs$

⟨*proof*⟩

lemma *subst_cls_lists_id_subst*[*simp*]: $Cs \cdot\cdot cl\ replicate\ (length\ Cs)\ id\_subst = Cs$
⟨*proof*⟩

lemma *subst_cls_mset_id_subst*[*simp*]: $CC \cdot cm\ id\_subst = CC$
⟨*proof*⟩

7.3.2 Associativity of Composition

lemma *comp_subst_assoc*[*simp*]: $\sigma \odot (\tau \odot \gamma) = \sigma \odot \tau \odot \gamma$
⟨*proof*⟩

7.3.3 Compatibility of Substitution and Composition

lemma *subst_atms_comp_subst*[*simp*]: $AA \cdot as\ (\tau \odot \sigma) = AA \cdot as\ \tau \cdot as\ \sigma$
⟨*proof*⟩

lemma *subst_atmss_comp_subst*[*simp*]: $AAA \cdot ass\ (\tau \odot \sigma) = AAA \cdot ass\ \tau \cdot ass\ \sigma$
⟨*proof*⟩

lemma *subst_atm_list_comp_subst*[*simp*]: $As \cdot al\ (\tau \odot \sigma) = As \cdot al\ \tau \cdot al\ \sigma$
⟨*proof*⟩

lemma *subst_atm_mset_comp_subst*[*simp*]: $AA \cdot am\ (\tau \odot \sigma) = AA \cdot am\ \tau \cdot am\ \sigma$
⟨*proof*⟩

lemma *subst_atm_mset_list_comp_subst*[*simp*]: $AAs \cdot aml\ (\tau \odot \sigma) = (AAs \cdot aml\ \tau) \cdot aml\ \sigma$
⟨*proof*⟩

lemma *subst_atm_mset_lists_comp_substs*[*simp*]: $AAs \cdot\cdot aml\ (\tau s \odot s\ \sigma s) = AAs \cdot\cdot aml\ \tau s \cdot\cdot aml\ \sigma s$
⟨*proof*⟩

lemma *subst_lit_comp_subst*[*simp*]: $L \cdot l\ (\tau \odot \sigma) = L \cdot l\ \tau \cdot l\ \sigma$
⟨*proof*⟩

lemma *subst_cls_comp_subst*[*simp*]: $C \cdot (\tau \odot \sigma) = C \cdot \tau \cdot \sigma$
⟨*proof*⟩

lemma *subst_clsscomp_subst*[*simp*]: $CC \cdot cs\ (\tau \odot \sigma) = CC \cdot cs\ \tau \cdot cs\ \sigma$
⟨*proof*⟩

lemma *subst_cls_list_comp_subst*[*simp*]: $Cs \cdot cl\ (\tau \odot \sigma) = Cs \cdot cl\ \tau \cdot cl\ \sigma$
⟨*proof*⟩

lemma *subst_cls_lists_comp_substs*[*simp*]: $Cs \cdot\cdot cl\ (\tau s \odot s\ \sigma s) = Cs \cdot\cdot cl\ \tau s \cdot\cdot cl\ \sigma s$
⟨*proof*⟩

lemma *subst_cls_mset_comp_subst*[*simp*]: $CC \cdot cm\ (\tau \odot \sigma) = CC \cdot cm\ \tau \cdot cm\ \sigma$
⟨*proof*⟩

7.3.4 "Commutativity" of Membership and Substitution

lemma *Melem_subst_atm_mset*[*simp*]: $A \in\# AA \cdot am\ \sigma \longleftrightarrow (\exists B.\ B \in\# AA \wedge A = B \cdot a\ \sigma)$
⟨*proof*⟩

lemma *Melem_subst_cls*[*simp*]: $L \in\# C \cdot \sigma \longleftrightarrow (\exists M.\ M \in\# C \wedge L = M \cdot l\ \sigma)$
⟨*proof*⟩

lemma *Melem_subst_cls_mset*[*simp*]: $AA \in\# CC \cdot cm\ \sigma \longleftrightarrow (\exists BB.\ BB \in\# CC \wedge AA = BB \cdot \sigma)$
⟨*proof*⟩

7.3.5 Signs and Substitutions

lemma *subst_lit_is_neg*[*simp*]: $is\_neg\ (L \cdot l\ \sigma) = is\_neg\ L$

⟨*proof*⟩

lemma *subst_lit_is_pos*[*simp*]: *is_pos* (*L* ·*l* *σ*) = *is_pos L*
⟨*proof*⟩

lemma *subst_minus*[*simp*]: (− *L*) ·*l* *μ* = − (*L* ·*l* *μ*)
⟨*proof*⟩

7.3.6 Substitution on Literal(s)

lemma *eql_neg_lit_eql_atm*[*simp*]: (*Neg A′* ·*l* *η*) = *Neg A* ⟷ *A′* ·*a* *η* = *A*
⟨*proof*⟩

lemma *eql_pos_lit_eql_atm*[*simp*]: (*Pos A′* ·*l* *η*) = *Pos A* ⟷ *A′* ·*a* *η* = *A*
⟨*proof*⟩

lemma *subst_cls_negs*[*simp*]: (*negs AA*) · *σ* = *negs* (*AA* ·*am* *σ*)
⟨*proof*⟩

lemma *subst_cls_poss*[*simp*]: (*poss AA*) · *σ* = *poss* (*AA* ·*am* *σ*)
⟨*proof*⟩

lemma *atms_of_subst_atms*: *atms_of C* ·*as* *σ* = *atms_of* (*C* · *σ*)
⟨*proof*⟩

lemma *in_image_Neg_is_neg*[*simp*]: *L* ·*l* *σ* ∈ *Neg* ‘ *AA* ⟹ *is_neg L*
⟨*proof*⟩

lemma *subst_lit_in_negs_subst_is_neg*: *L* ·*l* *σ* ∈# (*negs AA*) · *τ* ⟹ *is_neg L*
⟨*proof*⟩

lemma *subst_lit_in_negs_is_neg*: *L* ·*l* *σ* ∈# *negs AA* ⟹ *is_neg L*
⟨*proof*⟩

7.3.7 Substitution on Empty

lemma *subst_atms_empty*[*simp*]: {} ·*as* *σ* = {}
⟨*proof*⟩

lemma *subst_atmss_empty*[*simp*]: {} ·*ass* *σ* = {}
⟨*proof*⟩

lemma *comp_substs_empty_iff*[*simp*]: *σs* ⊙*s* *ηs* = [] ⟷ *σs* = [] ∨ *ηs* = []
⟨*proof*⟩

lemma *subst_atm_list_empty*[*simp*]: [] ·*al* *σ* = []
⟨*proof*⟩

lemma *subst_atm_mset_empty*[*simp*]: {#} ·*am* *σ* = {#}
⟨*proof*⟩

lemma *subst_atm_mset_list_empty*[*simp*]: [] ·*aml* *σ* = []
⟨*proof*⟩

lemma *subst_atm_mset_lists_empty*[*simp*]: [] ··*aml* *σs* = []
⟨*proof*⟩

lemma *subst_cls_empty*[*simp*]: {#} · *σ* = {#}
⟨*proof*⟩

lemma *subst_clss_empty*[*simp*]: {} ·*cs* *σ* = {}
⟨*proof*⟩

lemma *subst_cls_list_empty*[*simp*]: [] ·*cl* *σ* = []

⟨*proof*⟩

lemma *subst_cls_lists_empty*[*simp*]: $[] \cdot\cdot cl\ \sigma s = []$
⟨*proof*⟩

lemma *subst_scls_mset_empty*[*simp*]: $\{\#\} \cdot cm\ \sigma = \{\#\}$
⟨*proof*⟩

lemma *subst_atms_empty_iff*[*simp*]: $AA \cdot as\ \eta = \{\} \longleftrightarrow AA = \{\}$
⟨*proof*⟩

lemma *subst_atmss_empty_iff*[*simp*]: $AAA \cdot ass\ \eta = \{\} \longleftrightarrow AAA = \{\}$
⟨*proof*⟩

lemma *subst_atm_list_empty_iff*[*simp*]: $As \cdot al\ \eta = [] \longleftrightarrow As = []$
⟨*proof*⟩

lemma *subst_atm_mset_empty_iff*[*simp*]: $AA \cdot am\ \eta = \{\#\} \longleftrightarrow AA = \{\#\}$
⟨*proof*⟩

lemma *subst_atm_mset_list_empty_iff*[*simp*]: $AAs \cdot aml\ \eta = [] \longleftrightarrow AAs = []$
⟨*proof*⟩

lemma *subst_atm_mset_lists_empty_iff*[*simp*]: $AAs \cdot\cdot aml\ \eta s = [] \longleftrightarrow (AAs = [] \vee \eta s = [])$
⟨*proof*⟩

lemma *subst_cls_empty_iff*[*simp*]: $C \cdot \eta = \{\#\} \longleftrightarrow C = \{\#\}$
⟨*proof*⟩

lemma *subst_clss_empty_iff*[*simp*]: $CC \cdot cs\ \eta = \{\} \longleftrightarrow CC = \{\}$
⟨*proof*⟩

lemma *subst_cls_list_empty_iff*[*simp*]: $Cs \cdot cl\ \eta = [] \longleftrightarrow Cs = []$
⟨*proof*⟩

lemma *subst_cls_lists_empty_iff*[*simp*]: $Cs \cdot\cdot cl\ \eta s = [] \longleftrightarrow Cs = [] \vee \eta s = []$
⟨*proof*⟩

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

7.3.8 Substitution on a Union

lemma *subst_atms_union*[*simp*]: $(AA \cup BB) \cdot as\ \sigma = AA \cdot as\ \sigma \cup BB \cdot as\ \sigma$
⟨*proof*⟩

lemma *subst_atmss_union*[*simp*]: $(AAA \cup BBB) \cdot ass\ \sigma = AAA \cdot ass\ \sigma \cup BBB \cdot ass\ \sigma$
⟨*proof*⟩

lemma *subst_atm_list_append*[*simp*]: $(As\ @\ Bs) \cdot al\ \sigma = As \cdot al\ \sigma\ @\ Bs \cdot al\ \sigma$
⟨*proof*⟩

lemma *subst_atm_mset_union*[*simp*]: $(AA + BB) \cdot am\ \sigma = AA \cdot am\ \sigma + BB \cdot am\ \sigma$
⟨*proof*⟩

lemma *subst_atm_mset_list_append*[*simp*]: $(AAs\ @\ BBs) \cdot aml\ \sigma = AAs \cdot aml\ \sigma\ @\ BBs \cdot aml\ \sigma$
⟨*proof*⟩

lemma *subst_cls_union*[*simp*]: $(C + D) \cdot \sigma = C \cdot \sigma + D \cdot \sigma$
⟨*proof*⟩

lemma *subst_clss_union*[*simp*]: $(CC \cup DD) \cdot cs\ \sigma = CC \cdot cs\ \sigma \cup DD \cdot cs\ \sigma$
⟨*proof*⟩

lemma *subst_cls_list_append*[*simp*]: $(Cs \mathbin{@} Ds) \cdot cl\ \sigma = Cs \cdot cl\ \sigma \mathbin{@} Ds \cdot cl\ \sigma$
⟨*proof*⟩

lemma *subst_cls_lists_append*[*simp*]:
$length\ Cs = length\ \sigma s \Longrightarrow length\ Cs' = length\ \sigma s' \Longrightarrow$
$(Cs \mathbin{@} Cs') \cdot\cdot cl\ (\sigma s \mathbin{@} \sigma s') = Cs \cdot\cdot cl\ \sigma s \mathbin{@} Cs' \cdot\cdot cl\ \sigma s'$
⟨*proof*⟩

lemma *subst_cls_mset_union*[*simp*]: $(CC + DD) \cdot cm\ \sigma = CC \cdot cm\ \sigma + DD \cdot cm\ \sigma$
⟨*proof*⟩

7.3.9 Substitution on a Singleton

lemma *subst_atms_single*[*simp*]: $\{A\} \cdot as\ \sigma = \{A \cdot a\ \sigma\}$
⟨*proof*⟩

lemma *subst_atmss_single*[*simp*]: $\{AA\} \cdot ass\ \sigma = \{AA \cdot as\ \sigma\}$
⟨*proof*⟩

lemma *subst_atm_list_single*[*simp*]: $[A] \cdot al\ \sigma = [A \cdot a\ \sigma]$
⟨*proof*⟩

lemma *subst_atm_mset_single*[*simp*]: $\{\#A\#\} \cdot am\ \sigma = \{\#A \cdot a\ \sigma\#\}$
⟨*proof*⟩

lemma *subst_atm_mset_list*[*simp*]: $[AA] \cdot aml\ \sigma = [AA \cdot am\ \sigma]$
⟨*proof*⟩

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

lemma *subst_clss_single*[*simp*]: $\{C\} \cdot cs\ \sigma = \{C \cdot \sigma\}$
⟨*proof*⟩

lemma *subst_cls_list_single*[*simp*]: $[C] \cdot cl\ \sigma = [C \cdot \sigma]$
⟨*proof*⟩

lemma *subst_cls_lists_single*[*simp*]: $[C] \cdot\cdot cl\ [\sigma] = [C \cdot \sigma]$
⟨*proof*⟩

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

7.3.10 Substitution on $(\#)$

lemma *subst_atm_list_Cons*[*simp*]: $(A \mathbin{\#} As) \cdot al\ \sigma = A \cdot a\ \sigma \mathbin{\#} As \cdot al\ \sigma$
⟨*proof*⟩

lemma *subst_atm_mset_list_Cons*[*simp*]: $(A \mathbin{\#} As) \cdot aml\ \sigma = A \cdot am\ \sigma \mathbin{\#} As \cdot aml\ \sigma$
⟨*proof*⟩

lemma *subst_atm_mset_lists_Cons*[*simp*]: $(C \mathbin{\#} Cs) \cdot\cdot aml\ (\sigma \mathbin{\#} \sigma s) = C \cdot am\ \sigma \mathbin{\#} Cs \cdot\cdot aml\ \sigma s$
⟨*proof*⟩

lemma *subst_cls_list_Cons*[*simp*]: $(C \mathbin{\#} Cs) \cdot cl\ \sigma = C \cdot \sigma \mathbin{\#} Cs \cdot cl\ \sigma$
⟨*proof*⟩

lemma *subst_cls_lists_Cons*[*simp*]: $(C \mathbin{\#} Cs) \cdot\cdot cl\ (\sigma \mathbin{\#} \sigma s) = C \cdot \sigma \mathbin{\#} Cs \cdot\cdot cl\ \sigma s$
⟨*proof*⟩

7.3.11 Substitution on *tl*

lemma *subst_atm_list_tl*[*simp*]: $tl\ (As \cdot al\ \sigma) = tl\ As \cdot al\ \sigma$
⟨*proof*⟩

lemma *subst_atm_mset_list_tl*[*simp*]: *tl* (*AAs* ·*aml* σ) = *tl AAs* ·*aml* σ
⟨*proof*⟩

lemma *subst_cls_list_tl*[*simp*]: *tl* (*Cs* ·*cl* σ) = *tl Cs* ·*cl* σ
⟨*proof*⟩

lemma *subst_cls_lists_tl*[*simp*]: *length Cs* = *length* σs ⟹ *tl* (*Cs* ··*cl* σs) = *tl Cs* ··*cl tl* σs
⟨*proof*⟩

7.3.12 Substitution on (!)

lemma *comp_substs_nth*[*simp*]:
length τs = *length* σs ⟹ *i* < *length* τs ⟹ (τs ⊙*s* σs) ! *i* = (τs ! *i*) ⊙ (σs ! *i*)
⟨*proof*⟩

lemma *subst_atm_list_nth*[*simp*]: *i* < *length As* ⟹ (*As* ·*al* τ) ! *i* = *As* ! *i* ·*a* τ
⟨*proof*⟩

lemma *subst_atm_mset_list_nth*[*simp*]: *i* < *length AAs* ⟹ (*AAs* ·*aml* η) ! *i* = (*AAs* ! *i*) ·*am* η
⟨*proof*⟩

lemma *subst_atm_mset_lists_nth*[*simp*]:
length AAs = *length* σs ⟹ *i* < *length AAs* ⟹ (*AAs* ··*aml* σs) ! *i* = (*AAs* ! *i*) ·*am* (σs ! *i*)
⟨*proof*⟩

lemma *subst_cls_list_nth*[*simp*]: *i* < *length Cs* ⟹ (*Cs* ·*cl* τ) ! *i* = (*Cs* ! *i*) · τ
⟨*proof*⟩

lemma *subst_cls_lists_nth*[*simp*]:
length Cs = *length* σs ⟹ *i* < *length Cs* ⟹ (*Cs* ··*cl* σs) ! *i* = (*Cs* ! *i*) · (σs ! *i*)
⟨*proof*⟩

7.3.13 Substitution on Various Other Functions

lemma *subst_clss_image*[*simp*]: *image f X* ·*cs* σ = {*f x* · σ | *x*. *x* ∈ *X*}
⟨*proof*⟩

lemma *subst_cls_mset_image_mset*[*simp*]: *image_mset f X* ·*cm* σ = {# *f x* · σ. *x* ∈# *X* #}
⟨*proof*⟩

lemma *mset_subst_atm_list_subst_atm_mset*[*simp*]: *mset* (*As* ·*al* σ) = *mset* (*As*) ·*am* σ
⟨*proof*⟩

lemma *mset_subst_cls_list_subst_cls_mset*: *mset* (*Cs* ·*cl* σ) = (*mset Cs*) ·*cm* σ
⟨*proof*⟩

lemma *sum_list_subst_cls_list_subst_cls*[*simp*]: *sum_list* (*Cs* ·*cl* η) = *sum_list Cs* · η
⟨*proof*⟩

lemma *set_mset_subst_cls_mset_subst_clss*: *set_mset* (*CC* ·*cm* μ) = (*set_mset CC*) ·*cs* μ
⟨*proof*⟩

lemma *Neg_Melem_subst_atm_subst_cls*[*simp*]: *Neg A* ∈# *C* ⟹ *Neg* (*A* ·*a* σ) ∈# *C* · σ
⟨*proof*⟩

lemma *Pos_Melem_subst_atm_subst_cls*[*simp*]: *Pos A* ∈# *C* ⟹ *Pos* (*A* ·*a* σ) ∈# *C* · σ
⟨*proof*⟩

lemma *in_atms_of_subst*[*simp*]: *B* ∈ *atms_of C* ⟹ *B* ·*a* σ ∈ *atms_of* (*C* · σ)
⟨*proof*⟩

7.3.14 Renamings

lemma *is_renaming_id_subst*[*simp*]: *is_renaming id_subst*
⟨*proof*⟩

lemma *is_renamingD*: *is_renaming* $\sigma \Longrightarrow (\forall A1\ A2.\ A1 \cdot a\ \sigma = A2 \cdot a\ \sigma \longleftrightarrow A1 = A2)$
⟨*proof*⟩

lemma *inv_renaming_cancel_r*[*simp*]: *is_renaming* $r \Longrightarrow r \odot$ *inv_renaming* $r =$ *id_subst*
⟨*proof*⟩

lemma *inv_renaming_cancel_r_list*[*simp*]:
is_renaming_list $rs \Longrightarrow rs \odot s$ *map inv_renaming* $rs =$ *replicate* (*length* rs) *id_subst*
⟨*proof*⟩

lemma *Nil_comp_substs*[*simp*]: $[] \odot s\ s = []$
⟨*proof*⟩

lemma *comp_substs_Nil*[*simp*]: $s \odot s\ [] = []$
⟨*proof*⟩

lemma *is_renaming_idempotent_id_subst*: *is_renaming* $r \Longrightarrow r \odot r = r \Longrightarrow r =$ *id_subst*
⟨*proof*⟩

lemma *is_renaming_left_id_subst_right_id_subst*:
is_renaming $r \Longrightarrow s \odot r =$ *id_subst* $\Longrightarrow r \odot s =$ *id_subst*
⟨*proof*⟩

lemma *is_renaming_closure*: *is_renaming* $r1 \Longrightarrow$ *is_renaming* $r2 \Longrightarrow$ *is_renaming* $(r1 \odot r2)$
⟨*proof*⟩

lemma *is_renaming_inv_renaming_cancel_atm*[*simp*]: *is_renaming* $\varrho \Longrightarrow A \cdot a\ \varrho \cdot a$ *inv_renaming* $\varrho = A$
⟨*proof*⟩

lemma *is_renaming_inv_renaming_cancel_atms*[*simp*]: *is_renaming* $\varrho \Longrightarrow AA \cdot as\ \varrho \cdot as$ *inv_renaming* $\varrho = AA$
⟨*proof*⟩

lemma *is_renaming_inv_renaming_cancel_atmss*[*simp*]: *is_renaming* $\varrho \Longrightarrow AAA \cdot ass\ \varrho \cdot ass$ *inv_renaming* $\varrho = AAA$
⟨*proof*⟩

lemma *is_renaming_inv_renaming_cancel_atm_list*[*simp*]: *is_renaming* $\varrho \Longrightarrow As \cdot al\ \varrho \cdot al$ *inv_renaming* $\varrho = As$
⟨*proof*⟩

lemma *is_renaming_inv_renaming_cancel_atm_mset*[*simp*]: *is_renaming* $\varrho \Longrightarrow AA \cdot am\ \varrho \cdot am$ *inv_renaming* $\varrho = AA$
⟨*proof*⟩

lemma *is_renaming_inv_renaming_cancel_atm_mset_list*[*simp*]: *is_renaming* $\varrho \Longrightarrow (AAs \cdot aml\ \varrho) \cdot aml$ *inv_renaming* $\varrho = AAs$
⟨*proof*⟩

lemma *is_renaming_list_inv_renaming_cancel_atm_mset_lists*[*simp*]:
length $AAs =$ *length* $\varrho s \Longrightarrow$ *is_renaming_list* $\varrho s \Longrightarrow AAs \cdot\cdot aml\ \varrho s \cdot\cdot aml$ *map inv_renaming* $\varrho s = AAs$
⟨*proof*⟩

lemma *is_renaming_inv_renaming_cancel_lit*[*simp*]: *is_renaming* $\varrho \Longrightarrow (L \cdot l\ \varrho) \cdot l$ *inv_renaming* $\varrho = L$
⟨*proof*⟩

lemma *is_renaming_inv_renaming_cancel_cls*[*simp*]: *is_renaming* $\varrho \Longrightarrow C\ \cdot\ \varrho\ \cdot$ *inv_renaming* $\varrho = C$
⟨*proof*⟩

lemma *is_renaming_inv_renaming_cancel_clss*[*simp*]:
is_renaming $\varrho \Longrightarrow CC \cdot cs\ \varrho \cdot cs$ *inv_renaming* $\varrho = CC$

⟨*proof*⟩

lemma *is_renaming_inv_renaming_cancel_cls_list*[*simp*]:
is_renaming $\varrho \Longrightarrow Cs \cdot cl\ \varrho \cdot cl\ inv\_renaming\ \varrho = Cs$
⟨*proof*⟩

lemma *is_renaming_list_inv_renaming_cancel_cls_list*[*simp*]:
length $Cs = length\ \varrho s \Longrightarrow is\_renaming\_list\ \varrho s \Longrightarrow Cs \cdot\cdot cl\ \varrho s \cdot\cdot cl\ map\ inv\_renaming\ \varrho s = Cs$
⟨*proof*⟩

lemma *is_renaming_inv_renaming_cancel_cls_mset*[*simp*]:
is_renaming $\varrho \Longrightarrow CC \cdot cm\ \varrho \cdot cm\ inv\_renaming\ \varrho = CC$
⟨*proof*⟩

7.3.15 Monotonicity

lemma *subst_cls_mono*: $set\_mset\ C \subseteq set\_mset\ D \Longrightarrow set\_mset\ (C \cdot \sigma) \subseteq set\_mset\ (D \cdot \sigma)$
⟨*proof*⟩

lemma *subst_cls_mono_mset*: $C \subseteq\# D \Longrightarrow C \cdot \sigma \subseteq\# D \cdot \sigma$
⟨*proof*⟩

lemma *subst_subset_mono*: $D \subset\# C \Longrightarrow D \cdot \sigma \subset\# C \cdot \sigma$
⟨*proof*⟩

7.3.16 Size after Substitution

lemma *size_subst*[*simp*]: $size\ (D \cdot \sigma) = size\ D$
⟨*proof*⟩

lemma *subst_atm_list_length*[*simp*]: $length\ (As \cdot al\ \sigma) = length\ As$
⟨*proof*⟩

lemma *length_subst_atm_mset_list*[*simp*]: $length\ (AAs \cdot aml\ \eta) = length\ AAs$
⟨*proof*⟩

lemma *subst_atm_mset_lists_length*[*simp*]: $length\ (AAs \cdot\cdot aml\ \sigma s) = min\ (length\ AAs)\ (length\ \sigma s)$
⟨*proof*⟩

lemma *subst_cls_list_length*[*simp*]: $length\ (Cs \cdot cl\ \sigma) = length\ Cs$
⟨*proof*⟩

lemma *comp_substs_length*[*simp*]: $length\ (\tau s \odot s\ \sigma s) = min\ (length\ \tau s)\ (length\ \sigma s)$
⟨*proof*⟩

lemma *subst_cls_lists_length*[*simp*]: $length\ (Cs \cdot\cdot cl\ \sigma s) = min\ (length\ Cs)\ (length\ \sigma s)$
⟨*proof*⟩

7.3.17 Variable Disjointness

lemma *var_disjoint_clauses*:
assumes *var_disjoint Cs*
shows $\forall \sigma s.\ length\ \sigma s = length\ Cs \longrightarrow (\exists \tau.\ Cs \cdot\cdot cl\ \sigma s = Cs \cdot cl\ \tau)$
⟨*proof*⟩

7.3.18 Ground Expressions and Substitutions

lemma *ex_ground_subst*: $\exists \sigma.\ is\_ground\_subst\ \sigma$
⟨*proof*⟩

lemma *is_ground_cls_list_Cons*[*simp*]:
$is\_ground\_cls\_list\ (C\ \#\ Cs) = (is\_ground\_cls\ C \wedge is\_ground\_cls\_list\ Cs)$
⟨*proof*⟩

Ground union **lemma** *is_ground_atms_union*[*simp*]: *is_ground_atms* (*AA* ∪ *BB*) ⟷ *is_ground_atms AA* ∧ *is_ground_atms BB*
⟨*proof*⟩

lemma *is_ground_atm_mset_union*[*simp*]:
is_ground_atm_mset (*AA* + *BB*) ⟷ *is_ground_atm_mset AA* ∧ *is_ground_atm_mset BB*
⟨*proof*⟩

lemma *is_ground_cls_union*[*simp*]: *is_ground_cls* (*C* + *D*) ⟷ *is_ground_cls C* ∧ *is_ground_cls D*
⟨*proof*⟩

lemma *is_ground_clss_union*[*simp*]:
is_ground_clss (*CC* ∪ *DD*) ⟷ *is_ground_clss CC* ∧ *is_ground_clss DD*
⟨*proof*⟩

lemma *is_ground_cls_list_is_ground_cls_sum_list*[*simp*]:
is_ground_cls_list Cs ⟹ *is_ground_cls* (*sum_list Cs*)
⟨*proof*⟩

Grounding simplifications **lemma** *grounding_of_clss_empty*[*simp*]: *grounding_of_clss* {} = {}
⟨*proof*⟩

lemma *grounding_of_clss_singleton*[*simp*]: *grounding_of_clss* {*C*} = *grounding_of_cls C*
⟨*proof*⟩

lemma *grounding_of_clss_insert*:
grounding_of_clss (*insert C N*) = *grounding_of_cls C* ∪ *grounding_of_clss N*
⟨*proof*⟩

lemma *grounding_of_clss_union*:
grounding_of_clss (*A* ∪ *B*) = *grounding_of_clss A* ∪ *grounding_of_clss B*
⟨*proof*⟩

Grounding monotonicity **lemma** *is_ground_cls_mono*: *C* ⊆# *D* ⟹ *is_ground_cls D* ⟹ *is_ground_cls C*
⟨*proof*⟩

lemma *is_ground_clss_mono*: *CC* ⊆ *DD* ⟹ *is_ground_clss DD* ⟹ *is_ground_clss CC*
⟨*proof*⟩

lemma *grounding_of_clss_mono*: *CC* ⊆ *DD* ⟹ *grounding_of_clss CC* ⊆ *grounding_of_clss DD*
⟨*proof*⟩

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*
⟨*proof*⟩

Substituting on ground expression preserves ground **lemma** *is_ground_comp_subst*[*simp*]: *is_ground_subst* *σ* ⟹ *is_ground_subst* (*τ* ⊙ *σ*)
⟨*proof*⟩

lemma *ground_subst_ground_atm*[*simp*]: *is_ground_subst* *σ* ⟹ *is_ground_atm* (*A* ·*a* *σ*)
⟨*proof*⟩

lemma *ground_subst_ground_lit*[*simp*]: *is_ground_subst* *σ* ⟹ *is_ground_lit* (*L* ·*l* *σ*)
⟨*proof*⟩

lemma *ground_subst_ground_cls*[*simp*]: *is_ground_subst* *σ* ⟹ *is_ground_cls* (*C* · *σ*)
⟨*proof*⟩

lemma *ground_subst_ground_clss*[*simp*]: *is_ground_subst* *σ* ⟹ *is_ground_clss* (*CC* ·*cs* *σ*)
⟨*proof*⟩

lemma *ground_subst_ground_cls_list*[*simp*]: *is_ground_subst* $\sigma \Longrightarrow$ *is_ground_cls_list* (*Cs* $\cdot cl$ σ)
⟨*proof*⟩

lemma *ground_subst_ground_cls_lists*[*simp*]:
$\forall \sigma \in$ *set* σs. *is_ground_subst* $\sigma \Longrightarrow$ *is_ground_cls_list* (*Cs* $\cdot\cdot cl$ σs)
⟨*proof*⟩

lemma *subst_cls_eq_grounding_of_cls_subset_eq*:
assumes $D \cdot \sigma = C$
shows *grounding_of_cls* $C \subseteq$ *grounding_of_cls* D
⟨*proof*⟩

Substituting on ground expression has no effect **lemma** *is_ground_subst_atm*[*simp*]: *is_ground_atm* $A \Longrightarrow A \cdot a \sigma = A$
⟨*proof*⟩

lemma *is_ground_subst_atms*[*simp*]: *is_ground_atms* $AA \Longrightarrow AA \cdot as \sigma = AA$
⟨*proof*⟩

lemma *is_ground_subst_atm_mset*[*simp*]: *is_ground_atm_mset* $AA \Longrightarrow AA \cdot am \sigma = AA$
⟨*proof*⟩

lemma *is_ground_subst_atm_list*[*simp*]: *is_ground_atm_list* $As \Longrightarrow As \cdot al \sigma = As$
⟨*proof*⟩

lemma *is_ground_subst_atm_list_member*[*simp*]:
is_ground_atm_list $As \Longrightarrow i <$ *length* $As \Longrightarrow As ! i \cdot a \sigma = As ! i$
⟨*proof*⟩

lemma *is_ground_subst_lit*[*simp*]: *is_ground_lit* $L \Longrightarrow L \cdot l \sigma = L$
⟨*proof*⟩

lemma *is_ground_subst_cls*[*simp*]: *is_ground_cls* $C \Longrightarrow C \cdot \sigma = C$
⟨*proof*⟩

lemma *is_ground_subst_clss*[*simp*]: *is_ground_clss* $CC \Longrightarrow CC \cdot cs \sigma = CC$
⟨*proof*⟩

lemma *is_ground_subst_cls_lists*[*simp*]:
assumes *length* $P =$ *length* Cs **and** *is_ground_cls_list* Cs
shows $Cs \cdot\cdot cl P = Cs$
⟨*proof*⟩

lemma *is_ground_subst_lit_iff*: *is_ground_lit* $L \longleftrightarrow (\forall \sigma.\ L = L \cdot l \sigma)$
⟨*proof*⟩

lemma *is_ground_subst_cls_iff*: *is_ground_cls* $C \longleftrightarrow (\forall \sigma.\ C = C \cdot \sigma)$
⟨*proof*⟩

Grounding of substitutions **lemma** *grounding_of_subst_cls_subset*: *grounding_of_cls* $(C \cdot \mu) \subseteq$ *grounding_of_cls* C
⟨*proof*⟩

lemma *grounding_of_subst_clss_subset*: *grounding_of_clss* $(CC \cdot cs \mu) \subseteq$ *grounding_of_clss* CC
⟨*proof*⟩

lemma *grounding_of_subst_cls_renaming_ident*[*simp*]:
assumes *is_renaming* ϱ
shows *grounding_of_cls* $(C \cdot \varrho) =$ *grounding_of_cls* C
⟨*proof*⟩

lemma *grounding_of_subst_clss_renaming_ident*[*simp*]:
assumes *is_renaming* ϱ

shows *grounding_of_clss* (*CC* ·*cs* ϱ) = *grounding_of_clss CC*
⟨*proof*⟩

Members of ground expressions are ground **lemma** *is_ground_cls_as_atms*: *is_ground_cls C* ⟷ (∀ *A* ∈ *atms_of C*. *is_ground_atm A*)
⟨*proof*⟩

lemma *is_ground_cls_imp_is_ground_lit*: *L* ∈# *C* ⟹ *is_ground_cls C* ⟹ *is_ground_lit L*
⟨*proof*⟩

lemma *is_ground_cls_imp_is_ground_atm*: *A* ∈ *atms_of C* ⟹ *is_ground_cls C* ⟹ *is_ground_atm A*
⟨*proof*⟩

lemma *is_ground_cls_is_ground_atms_atms_of*[*simp*]: *is_ground_cls C* ⟹ *is_ground_atms* (*atms_of C*)
⟨*proof*⟩

lemma *grounding_ground*: *C* ∈ *grounding_of_clss M* ⟹ *is_ground_cls C*
⟨*proof*⟩

lemma *is_ground_cls_if_in_grounding_of_cls*: *C′* ∈ *grounding_of_cls C* ⟹ *is_ground_cls C′*
⟨*proof*⟩

lemma *in_subset_eq_grounding_of_clss_is_ground_cls*[*simp*]:
C ∈ *CC* ⟹ *CC* ⊆ *grounding_of_clss DD* ⟹ *is_ground_cls C*
⟨*proof*⟩

lemma *is_ground_cls_empty*[*simp*]: *is_ground_cls* {#}
⟨*proof*⟩

lemma *is_ground_cls_add_mset*[*simp*]:
is_ground_cls (*add_mset L C*) ⟷ *is_ground_lit L* ∧ *is_ground_cls C*
⟨*proof*⟩

lemma *grounding_of_cls_ground*: *is_ground_cls C* ⟹ *grounding_of_cls C* = {*C*}
⟨*proof*⟩

lemma *grounding_of_cls_empty*[*simp*]: *grounding_of_cls* {#} = {{#}}
⟨*proof*⟩

lemma *union_grounding_of_cls_ground*: *is_ground_clss* (⋃ (*grounding_of_cls* ‘ *N*))
⟨*proof*⟩

lemma *is_ground_clss_grounding_of_clss*[*simp*]: *is_ground_clss* (*grounding_of_clss N*)
⟨*proof*⟩

Grounding idempotence **lemma** *grounding_of_grounding_of_cls*: *E* ∈ *grounding_of_cls D* ⟹ *D* ∈ *grounding_of_cls C* ⟹ *E* = *D*
⟨*proof*⟩

lemma *image_grounding_of_cls_grounding_of_cls*:
grounding_of_cls ‘ *grounding_of_cls C* = (λ*x*. {*x*}) ‘ *grounding_of_cls C*
⟨*proof*⟩

lemma *grounding_of_clss_grounding_of_clss*[*simp*]:
grounding_of_clss (*grounding_of_clss N*) = *grounding_of_clss N*
⟨*proof*⟩

7.3.19 Subsumption

lemma *subsumes_empty_left*[*simp*]: *subsumes* {#} *C*
⟨*proof*⟩

lemma *strictly_subsumes_empty_left*[*simp*]: *strictly_subsumes* {#} *C* ⟷ *C* ≠ {#}
⟨*proof*⟩

7.3.20 Unifiers

lemma *card_le_one_alt*: *finite* $X \Longrightarrow$ *card* $X \leq 1 \longleftrightarrow X = \{\} \vee (\exists x.\ X = \{x\})$
⟨*proof*⟩

lemma *is_unifier_subst_atm_eqI*:
assumes *finite* AA
shows *is_unifier* σ $AA \Longrightarrow A \in AA \Longrightarrow B \in AA \Longrightarrow A \cdot a\ \sigma = B \cdot a\ \sigma$
⟨*proof*⟩

lemma *is_unifier_alt*:
assumes *finite* AA
shows *is_unifier* σ $AA \longleftrightarrow (\forall A \in AA.\ \forall B \in AA.\ A \cdot a\ \sigma = B \cdot a\ \sigma)$
⟨*proof*⟩

lemma *is_unifiers_subst_atm_eqI*:
assumes *finite* AA *is_unifiers* σ AAA $AA \in AAA$ $A \in AA$ $B \in AA$
shows $A \cdot a\ \sigma = B \cdot a\ \sigma$
⟨*proof*⟩

theorem *is_unifiers_comp*:
is_unifiers σ (*set_mset* ‘ *set* (*map2 add_mset As Bs*) ·*ass* η) $\longleftrightarrow$
is_unifiers ($\eta \odot \sigma$) (*set_mset* ‘ *set* (*map2 add_mset As Bs*))
⟨*proof*⟩

7.3.21 Most General Unifier

lemma *is_mgu_is_unifiers*: *is_mgu* σ $AAA \Longrightarrow$ *is_unifiers* σ AAA
⟨*proof*⟩

lemma *is_mgu_is_most_general*: *is_mgu* σ $AAA \Longrightarrow$ *is_unifiers* τ $AAA \Longrightarrow \exists \gamma.\ \tau = \sigma \odot \gamma$
⟨*proof*⟩

lemma *is_unifiers_is_unifier*: *is_unifiers* σ $AAA \Longrightarrow AA \in AAA \Longrightarrow$ *is_unifier* σ AA
⟨*proof*⟩

lemma *is_imgu_is_mgu*[*intro*]: *is_imgu* σ $AAA \Longrightarrow$ *is_mgu* σ AAA
⟨*proof*⟩

lemma *is_imgu_comp_idempotent*[*simp*]: *is_imgu* σ $AAA \Longrightarrow \sigma \odot \sigma = \sigma$
⟨*proof*⟩

lemma *is_imgu_subst_atm_idempotent*[*simp*]: *is_imgu* σ $AAA \Longrightarrow A \cdot a\ \sigma \cdot a\ \sigma = A \cdot a\ \sigma$
⟨*proof*⟩

lemma *is_imgu_subst_atms_idempotent*[*simp*]: *is_imgu* σ $AAA \Longrightarrow AA \cdot as\ \sigma \cdot as\ \sigma = AA \cdot as\ \sigma$
⟨*proof*⟩

lemma *is_imgu_subst_lit_idemptotent*[*simp*]: *is_imgu* σ $AAA \Longrightarrow L \cdot l\ \sigma \cdot l\ \sigma = L \cdot l\ \sigma$
⟨*proof*⟩

lemma *is_imgu_subst_cls_idemptotent*[*simp*]: *is_imgu* σ $AAA \Longrightarrow C \cdot \sigma \cdot \sigma = C \cdot \sigma$
⟨*proof*⟩

lemma *is_imgu_subst_clss_idemptotent*[*simp*]: *is_imgu* σ $AAA \Longrightarrow CC \cdot cs\ \sigma \cdot cs\ \sigma = CC \cdot cs\ \sigma$
⟨*proof*⟩

7.3.22 Generalization and Subsumption

lemma *variants_sym*: *variants* $D\ D' \longleftrightarrow$ *variants* $D'\ D$
⟨*proof*⟩

lemma *variants_iff_subsumes*: *variants* $C\ D \longleftrightarrow$ *subsumes* $C\ D \wedge$ *subsumes* $D\ C$
⟨*proof*⟩

lemma *strict_subset_subst_strictly_subsumes*: $C \cdot \eta \subset\# D \Longrightarrow$ *strictly_subsumes C D*
⟨*proof*⟩

lemma *generalizes_lit_refl*[*simp*]: *generalizes_lit L L*
⟨*proof*⟩

lemma *generalizes_lit_trans*:
generalizes_lit L1 L2 $\Longrightarrow$ *generalizes_lit L2 L3* $\Longrightarrow$ *generalizes_lit L1 L3*
⟨*proof*⟩

lemma *generalizes_refl*[*simp*]: *generalizes C C*
⟨*proof*⟩

lemma *generalizes_trans*: *generalizes C D* $\Longrightarrow$ *generalizes D E* $\Longrightarrow$ *generalizes C E*
⟨*proof*⟩

lemma *subsumes_refl*: *subsumes C C*
⟨*proof*⟩

lemma *subsumes_trans*: *subsumes C D* $\Longrightarrow$ *subsumes D E* $\Longrightarrow$ *subsumes C E*
⟨*proof*⟩

lemma *strictly_generalizes_irrefl*: $\neg$ *strictly_generalizes C C*
⟨*proof*⟩

lemma *strictly_generalizes_antisym*: *strictly_generalizes C D* $\Longrightarrow \neg$ *strictly_generalizes D C*
⟨*proof*⟩

lemma *strictly_generalizes_trans*:
strictly_generalizes C D $\Longrightarrow$ *strictly_generalizes D E* $\Longrightarrow$ *strictly_generalizes C E*
⟨*proof*⟩

lemma *strictly_subsumes_irrefl*: $\neg$ *strictly_subsumes C C*
⟨*proof*⟩

lemma *strictly_subsumes_antisym*: *strictly_subsumes C D* $\Longrightarrow \neg$ *strictly_subsumes D C*
⟨*proof*⟩

lemma *strictly_subsumes_trans*:
strictly_subsumes C D $\Longrightarrow$ *strictly_subsumes D E* $\Longrightarrow$ *strictly_subsumes C E*
⟨*proof*⟩

lemma *subset_strictly_subsumes*: $C \subset\# D \Longrightarrow$ *strictly_subsumes C D*
⟨*proof*⟩

lemma *strictly_generalizes_neq*: *strictly_generalizes D′ D* $\Longrightarrow D' \neq D \cdot \sigma$
⟨*proof*⟩

lemma *strictly_subsumes_neq*: *strictly_subsumes D′ D* $\Longrightarrow D' \neq D \cdot \sigma$
⟨*proof*⟩

lemma *variants_imp_exists_substitution*: *variants D D′* $\Longrightarrow \exists \sigma.\ D \cdot \sigma = D'$
⟨*proof*⟩

lemma *strictly_subsumes_variants*:
assumes *strictly_subsumes E D* **and** *variants D D′*
shows *strictly_subsumes E D′*
⟨*proof*⟩

lemma *neg_strictly_subsumes_variants*:
assumes $\neg$ *strictly_subsumes E D* **and** *variants D D′*
shows $\neg$ *strictly_subsumes E D′*
⟨*proof*⟩

end

locale *substitution_renamings* = *substitution subst_atm id_subst comp_subst*
  for
    subst_atm :: *'a* ⇒ *'s* ⇒ *'a* **and**
    id_subst :: *'s* **and**
    comp_subst :: *'s* ⇒ *'s* ⇒ *'s* +
  fixes
    renamings_apart :: *'a clause list* ⇒ *'s list* **and**
    atm_of_atms :: *'a list* ⇒ *'a*
  assumes
    renamings_apart_length: *length* (*renamings_apart Cs*) = *length Cs* **and**
    renamings_apart_renaming: *ϱ* ∈ *set* (*renamings_apart Cs*) ⟹ *is_renaming ϱ* **and**
    renamings_apart_var_disjoint: *var_disjoint* (*Cs* ··*cl* (*renamings_apart Cs*)) **and**
    atm_of_atms_subst:
      ⋀*As Bs*. *atm_of_atms As* ·*a σ* = *atm_of_atms Bs* ⟷ *map* (λ*A*. *A* ·*a σ*) *As* = *Bs*
begin

7.3.23 Generalization and Subsumption

lemma *wf_strictly_generalizes*: *wfP strictly_generalizes*
⟨*proof*⟩

lemma *strictly_subsumes_has_minimum*:
  assumes *CC* ≠ {}
  shows ∃ *C* ∈ *CC*. ∀ *D* ∈ *CC*. ¬ *strictly_subsumes D C*
⟨*proof*⟩

lemma *wf_strictly_subsumes*: *wfP strictly_subsumes*
  ⟨*proof*⟩

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*⟩

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_prems_of*: *'a clause multiset*) (*main_prem_of*: *'a clause*) (*concl_of*: *'a clause*)

abbreviation *prems_of* :: *'a inference* ⇒ *'a clause multiset* **where**
  prems_of γ ≡ *side_prems_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*
  ⟨*proof*⟩

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* $\subseteq$ *N* **and**
d_in_n: *D* $\in$ *N*
shows *E* $\in$ *N*
⟨*proof*⟩

end

Satisfiability preservation is a weaker requirement than soundness.

locale *sat_preserving_inference_system* = *inference_system* +
assumes Γ*_sat_preserving*: *satisfiable N* $\Longrightarrow$ *satisfiable* (*N* $\cup$ *concls_of* (*inferences_from N*))

locale *sound_inference_system* = *inference_system* +
assumes Γ*_sound*: *Infer CC D E* $\in$ Γ $\Longrightarrow$ *I* $\models m$ *CC* $\Longrightarrow$ *I* $\models$ *D* $\Longrightarrow$ *I* $\models$ *E*
begin

lemma Γ*_sat_preserving*:
assumes *sat_n*: *satisfiable N*
shows *satisfiable* (*N* $\cup$ *concls_of* (*inferences_from N*))
⟨*proof*⟩

sublocale *sat_preserving_inference_system*
⟨*proof*⟩

end

locale *reductive_inference_system* = *inference_system* Γ **for** Γ :: (′*a* :: *wellorder*) *inference set* +
assumes Γ*_reductive*: $\gamma \in \Gamma \Longrightarrow$ *concl_of* $\gamma <$ *main_prem_of* γ

8.2 Refutational Completeness

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

locale *counterex_reducing_inference_system* =
inference_system Γ **for** Γ :: (′*a* :: *wellorder*) *inference set* +
fixes *I_of* :: ′*a clause set* $\Rightarrow$ ′*a interp*
assumes Γ*_counterex_reducing*:
$\{\#\} \notin N \Longrightarrow D \in N \Longrightarrow \neg$ *I_of N* $\models D \Longrightarrow (\bigwedge C.\ C \in N \Longrightarrow \neg$ *I_of N* $\models C \Longrightarrow D \leq C) \Longrightarrow$
$\exists$ *CC E. set_mset CC* $\subseteq N \wedge$ *I_of N* $\models m$ *CC* $\wedge$ *Infer CC D E* $\in \Gamma \wedge \neg$ *I_of N* $\models E \wedge E < D$
begin

lemma *ex_min_counterex*:
fixes *N* :: (′*a* :: *wellorder*) *clause set*
assumes $\neg$ *I* $\models s$ *N*
shows $\exists\, C \in N.\ \neg$ *I* $\models$ *C* $\wedge$ ($\forall\, D \in N.\ D < C \longrightarrow$ *I* $\models$ *D*)
⟨*proof*⟩

theorem *saturated_model*:
assumes
satur: *saturated N* **and**
ec_ni_n: $\{\#\} \notin N$
shows *I_of N* $\models s$ *N*
⟨*proof*⟩

Cf. Corollary 3.10:

corollary *saturated_complete*: *saturated N* $\Longrightarrow \neg$ *satisfiable N* $\Longrightarrow \{\#\} \in N$
⟨*proof*⟩

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

lemma *saturate_mono*: $C \in saturate\ CC \Longrightarrow CC \subseteq DD \Longrightarrow C \in saturate\ DD$
  ⟨*proof*⟩

lemma *saturated_saturate*[*simp*, *intro*]: *saturated* (*saturate N*)
  ⟨*proof*⟩

lemma *saturate_finite*: $C \in saturate\ CC \Longrightarrow \exists DD.\ DD \subseteq CC \wedge finite\ DD \wedge C \in saturate\ DD$
⟨*proof*⟩

end

context *sound_inference_system*
begin

theorem *saturate_sound*: $C \in saturate\ CC \Longrightarrow I \models s\ CC \Longrightarrow I \models C$
  ⟨*proof*⟩

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* ⟹ *satisfiable* (*saturate CC*)
  ⟨*proof*⟩

end

locale *sound_counterex_reducing_inference_system* =
  counterex_reducing_inference_system + *sound_inference_system*
begin

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

theorem *clausal_logic_compact*:
  fixes *N* :: (*'a* :: *wellorder*) *clause set*
  shows ¬ *satisfiable N* ⟷ (∃ *DD* ⊆ *N*. *finite DD* ∧ ¬ *satisfiable DD*)
⟨*proof*⟩

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* ⇒ *'a clause*
assumes
S_selects_subseteq: $S\ C \subseteq\#\ C$ **and**
S_selects_neg_lits: $L \in\#\ S\ C \Longrightarrow$ *is_neg L*

locale *ground_resolution_with_selection* = *selection S*
for S :: (*'a* :: *wellorder*) *clause* ⇒ *'a clause*
begin

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

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

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

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

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

lemma *production_unfold*:
production C = $\{A.\ C \in N \wedge C \neq \{\#\} \wedge$ *Max_mset C* = *Pos A* $\wedge \neg$ *interp C* $\models C \wedge S\ C = \{\#\}\}$
⟨*proof*⟩

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

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

lemma *producesD*: *produces C A* $\Longrightarrow C \in N \wedge C \neq \{\#\} \wedge$ *Pos A* = *Max_mset C* $\wedge \neg$ *interp C* $\models C \wedge S\ C = \{\#\}$
⟨*proof*⟩

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

lemma *interp_subseteq_Interp*[*simp*]: *interp C* ⊆ *Interp C*
⟨*proof*⟩

lemma *Interp_as_UNION*: *Interp C* = $(\bigcup D \in \{D.\ D \leq C\}.$ *production D*$)$
⟨*proof*⟩

lemma *productive_not_empty*: *productive C* $\Longrightarrow C \neq \{\#\}$

⟨*proof*⟩

lemma *productive_imp_produces_Max_literal*: *productive C* ⟹ *produces C* (*atm_of* (*Max_mset C*))
⟨*proof*⟩

lemma *productive_imp_produces_Max_atom*: *productive C* ⟹ *produces C* (*Max* (*atms_of C*))
⟨*proof*⟩

lemma *produces_imp_Max_literal*: *produces C A* ⟹ *A* = *atm_of* (*Max_mset C*)
⟨*proof*⟩

lemma *produces_imp_Max_atom*: *produces C A* ⟹ *A* = *Max* (*atms_of C*)
⟨*proof*⟩

lemma *produces_imp_Pos_in_lits*: *produces C A* ⟹ *Pos A* ∈# *C*
⟨*proof*⟩

lemma *productive_in_N*: *productive C* ⟹ *C* ∈ *N*
⟨*proof*⟩

lemma *produces_imp_atms_leq*: *produces C A* ⟹ *B* ∈ *atms_of C* ⟹ *B* ≤ *A*
⟨*proof*⟩

lemma *produces_imp_neg_notin_lits*: *produces C A* ⟹ ¬ *Neg A* ∈# *C*
⟨*proof*⟩

lemma *less_eq_imp_interp_subseteq_interp*: *C* ≤ *D* ⟹ *interp C* ⊆ *interp D*
⟨*proof*⟩

lemma *less_eq_imp_interp_subseteq_Interp*: *C* ≤ *D* ⟹ *interp C* ⊆ *Interp D*
⟨*proof*⟩

lemma *less_imp_production_subseteq_interp*: *C* < *D* ⟹ *production C* ⊆ *interp D*
⟨*proof*⟩

lemma *less_eq_imp_production_subseteq_Interp*: *C* ≤ *D* ⟹ *production C* ⊆ *Interp D*
⟨*proof*⟩

lemma *less_imp_Interp_subseteq_interp*: *C* < *D* ⟹ *Interp C* ⊆ *interp D*
⟨*proof*⟩

lemma *less_eq_imp_Interp_subseteq_Interp*: *C* ≤ *D* ⟹ *Interp C* ⊆ *Interp D*
⟨*proof*⟩

lemma *not_Interp_to_interp_imp_less*: *A* ∉ *Interp C* ⟹ *A* ∈ *interp D* ⟹ *C* < *D*
⟨*proof*⟩

lemma *not_interp_to_interp_imp_less*: *A* ∉ *interp C* ⟹ *A* ∈ *interp D* ⟹ *C* < *D*
⟨*proof*⟩

lemma *not_Interp_to_Interp_imp_less*: *A* ∉ *Interp C* ⟹ *A* ∈ *Interp D* ⟹ *C* < *D*
⟨*proof*⟩

lemma *not_interp_to_Interp_imp_le*: *A* ∉ *interp C* ⟹ *A* ∈ *Interp D* ⟹ *C* ≤ *D*
⟨*proof*⟩

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

lemma *interp_subseteq_INTERP*: *interp C* ⊆ *INTERP*
⟨*proof*⟩

lemma *production_subseteq_INTERP*: *production C* ⊆ *INTERP*

⟨*proof*⟩

lemma *Interp_subseteq_INTERP*: *Interp C* ⊆ *INTERP*
⟨*proof*⟩

lemma *produces_imp_in_interp*:
assumes *a_in_c*: *Neg A* ∈# *C* **and** *d*: *produces D A*
shows *A* ∈ *interp C*
⟨*proof*⟩

lemma *neg_notin_Interp_not_produce*: *Neg A* ∈# *C* ⟹ *A* ∉ *Interp D* ⟹ *C* ≤ *D* ⟹ ¬ *produces D*′′ *A*
⟨*proof*⟩

lemma *in_production_imp_produces*: *A* ∈ *production C* ⟹ *produces C A*
⟨*proof*⟩

lemma *not_produces_imp_notin_production*: ¬ *produces C A* ⟹ *A* ∉ *production C*
⟨*proof*⟩

lemma *not_produces_imp_notin_interp*: (⋀*D*. ¬ *produces D A*) ⟹ *A* ∉ *interp C*
⟨*proof*⟩

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*⟩

lemma *Interp_imp_interp*: *C* ≤ *D* ⟹ *D* < *D*′ ⟹ *Interp D* ⊨ *C* ⟹ *interp D*′ ⊨ *C*
⟨*proof*⟩

lemma *Interp_imp_Interp*: *C* ≤ *D* ⟹ *D* ≤ *D*′ ⟹ *Interp D* ⊨ *C* ⟹ *Interp D*′ ⊨ *C*
⟨*proof*⟩

lemma *Interp_imp_INTERP*: *C* ≤ *D* ⟹ *Interp D* ⊨ *C* ⟹ *INTERP* ⊨ *C*
⟨*proof*⟩

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*⟩

lemma *interp_imp_interp*: *C* ≤ *D* ⟹ *D* ≤ *D*′ ⟹ *interp D* ⊨ *C* ⟹ *interp D*′ ⊨ *C*
⟨*proof*⟩

lemma *interp_imp_Interp*: *C* ≤ *D* ⟹ *D* ≤ *D*′ ⟹ *interp D* ⊨ *C* ⟹ *Interp D*′ ⊨ *C*
⟨*proof*⟩

lemma *interp_imp_INTERP*: *C* ≤ *D* ⟹ *interp D* ⊨ *C* ⟹ *INTERP* ⊨ *C*
⟨*proof*⟩

lemma *productive_imp_not_interp*: *productive C* ⟹ ¬ *interp C* ⊨ *C*
⟨*proof*⟩

This corresponds to Lemma 3.3:

lemma *productive_imp_Interp*:
assumes *productive C*
shows *Interp C* $\models$ *C*
⟨*proof*⟩

lemma *productive_imp_INTERP*: *productive C* $\Longrightarrow$ *INTERP* $\models$ *C*
⟨*proof*⟩

This corresponds to Lemma 3.5:

lemma *max_pos_imp_Interp*:
assumes *C* $\in$ *N* **and** *C* $\neq$ {#} **and** *Max_mset C* = *Pos A* **and** *S C* = {#}
shows *Interp C* $\models$ *C*
⟨*proof*⟩

The following results correspond to Lemma 3.6:

lemma *max_atm_imp_Interp*:
assumes
c_in_n: *C* $\in$ *N* **and**
pos_in: *Pos A* $\in\#$ *C* **and**
max_atm: *A* = *Max* (*atms_of C*) **and**
s_c_e: *S C* = {#}
shows *Interp C* $\models$ *C*
⟨*proof*⟩

lemma *not_Interp_imp_general*:
assumes
d'_le_d: *D'* $\leq$ *D* **and**
in_n_or_max_gt: *D'* $\in$ *N* $\wedge$ *S D'* = {#} $\vee$ *Max* (*atms_of D'*) < *Max* (*atms_of D*) **and**
d'_at_d: $\neg$ *Interp D* $\models$ *D'* **and**
d_lt_c: *D* < *C* **and**
subs: *interp C* $\subseteq$ ($\bigcup$ *C* $\in$ *CC*. *production C*)
shows $\neg$ ($\bigcup$ *C* $\in$ *CC*. *production C*) $\models$ *D'*
⟨*proof*⟩

lemma *not_Interp_imp_not_interp*:
D' $\leq$ *D* $\Longrightarrow$ *D'* $\in$ *N* $\wedge$ *S D'* = {#} $\vee$ *Max* (*atms_of D'*) < *Max* (*atms_of D*) $\Longrightarrow$ $\neg$ *Interp D* $\models$ *D'* $\Longrightarrow$
D < *C* $\Longrightarrow$ $\neg$ *interp C* $\models$ *D'*
⟨*proof*⟩

lemma *not_Interp_imp_not_Interp*:
D' $\leq$ *D* $\Longrightarrow$ *D'* $\in$ *N* $\wedge$ *S D'* = {#} $\vee$ *Max* (*atms_of D'*) < *Max* (*atms_of D*) $\Longrightarrow$ $\neg$ *Interp D* $\models$ *D'* $\Longrightarrow$
D < *C* $\Longrightarrow$ $\neg$ *Interp C* $\models$ *D'*
⟨*proof*⟩

lemma *not_Interp_imp_not_INTERP*:
D' $\leq$ *D* $\Longrightarrow$ *D'* $\in$ *N* $\wedge$ *S D'* = {#} $\vee$ *Max* (*atms_of D'*) < *Max* (*atms_of D*) $\Longrightarrow$ $\neg$ *Interp D* $\models$ *D'* $\Longrightarrow$
$\neg$ *INTERP* $\models$ *D'*
⟨*proof*⟩

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

lemma
assumes *D* $\in$ *N* **and** $\bigwedge$*D'*. *D'* < *D* $\Longrightarrow$ *Interp D'* $\models$ *C*
shows *interp D* $\models$ *C*
⟨*proof*⟩

lemma
assumes *d*: *D* = {#} **and** *n*: *N* = {*D*, *C*} **and** *c*: *C* = {#*Pos A*#}
shows *D* $\in$ *N* **and** $\bigwedge$*D'*. *D'* < *D* $\Longrightarrow$ *Interp D'* $\models$ *C* **and** $\neg$ *interp D* $\models$ *C*
⟨*proof*⟩

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** $S = \lambda\_.\ \{\#\}$
  ⟨*proof*⟩

inductive *unord_resolve* :: *'a clause* ⇒ *'a clause* ⇒ *'a clause* ⇒ *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*
  ⟨*proof*⟩

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*⟩

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*⟩

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
⟨*proof*⟩

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* (∑ # (*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*
⟨*proof*⟩

lemma *filter_neg_atm_of_S*: $\{\#Neg\ (atm\_of\ L).\ L \in\#\ S\ C\#\} = S\ C$
⟨*proof*⟩

This corresponds to Lemma 3.13:

lemma *ord_resolve_reductive*:
assumes *ord_resolve CAs DA AAs As E*
shows $E < DA$
⟨*proof*⟩

This corresponds to Theorem 3.15:

theorem *ord_resolve_counterex_reducing*:
assumes
ec_ni_n: $\{\#\} \notin N$ **and**
d_in_n: $DA \in N$ **and**
d_cex: $\neg\ INTERP\ N \models DA$ **and**
d_min: $\bigwedge C.\ C \in N \Longrightarrow \neg\ INTERP\ N \models C \Longrightarrow DA \leq C$
obtains *CAs AAs As E* **where**
set CAs $\subseteq N$
INTERP N $\models m$ *mset CAs*
$\bigwedge CA.\ CA \in set\ CAs \Longrightarrow productive\ N\ CA$
ord_resolve CAs DA AAs As E
$\neg\ INTERP\ N \models E$
$E < DA$
⟨*proof*⟩

lemma *ord_resolve_atms_of_concl_subset*:
assumes *ord_resolve CAs DA AAs As E*
shows $atms\_of\ E \subseteq (\bigcup C \in set\ CAs.\ atms\_of\ C) \cup atms\_of\ DA$
⟨*proof*⟩

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**
$ord\_\Gamma = \{Infer\ (mset\ CAs)\ DA\ E \mid CAs\ DA\ AAs\ As\ E.\ ord\_resolve\ CAs\ DA\ AAs\ As\ E\}$

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*⟩

lemmas *clausal_logic_compact* = *ord_Γ_sound_counterex_reducing.clausal_logic_compact*

end

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

lemmas *clausal_logic_compact* = *ground_resolution_with_selection.clausal_logic_compact*

end

12 Theorem Proving Processes

theory *Proving_Process*
imports *Unordered_Ground_Resolution Lazy_List_Chain*
begin

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

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

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

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

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

lemma *derive_subset*: *M* ▷ *N* ⟹ *N* ⊆ *M* ∪ *concls_of* (*inferences_from M*)
  ⟨*proof*⟩

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*⟩

This corresponds to Lemma 4.2:

lemma
  assumes *deriv*: *chain* (▷) *Ns*
  shows
    Rf_Sup_subset_Rf_Liminf: *Rf* (*Sup_llist Ns*) ⊆ *Rf* (*Liminf_llist Ns*) **and**
    Ri_Sup_subset_Ri_Liminf: *Ri* (*Sup_llist Ns*) ⊆ *Ri* (*Liminf_llist Ns*) **and**
    sat_limit_iff: *satisfiable* (*Liminf_llist Ns*) ⟷ *satisfiable* (*lhd Ns*)
⟨*proof*⟩

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*)
  ⟨*proof*⟩

end

The assumption below corresponds to condition R4 of Definition 4.1.

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

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

end

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

sublocale *sat_preserving_redundancy_criterion*
  ⟨*proof*⟩

The result below corresponds to Theorem 4.3.

theorem *fair_derive_saturated_upto*:
  assumes
    deriv: *chain* (▷) *Ns* **and**
    fair: *fair_clss_seq Ns*
  shows *saturated_upto* (*Liminf_llist Ns*)
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *saturated_upto_iff*: *saturated_upto N* ⟷ *concls_of* (*inferences_from N*) ⊆ *N*
  ⟨*proof*⟩

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* (λ*N*. *Ri N* ∪ (Γ′ − Γ))
  ⟨*proof*⟩

lemma *redundancy_criterion_standard_extension_saturated_upto_iff*:
  assumes Γ ⊆ Γ′ **and** *redundancy_criterion* Γ *Rf Ri*
  shows *redundancy_criterion.saturated_upto* Γ *Rf Ri M* ⟷
    redundancy_criterion.saturated_upto Γ′ *Rf* (λ*N*. *Ri N* ∪ (Γ′ − Γ)) *M*
  ⟨*proof*⟩

lemma *redundancy_criterion_standard_extension_effective*:
  assumes Γ ⊆ Γ′ **and** *effective_redundancy_criterion* Γ *Rf Ri*
  shows *effective_redundancy_criterion* Γ′ *Rf* (λ*N*. *Ri N* ∪ (Γ′ − Γ))
  ⟨*proof*⟩

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* (λ*N*. *Ri N* ∪ (Γ′ − Γ)) *Ns* ⟷
    effective_redundancy_criterion.fair_clss_seq Γ *Rf Ri Ns*
  ⟨*proof*⟩

theorem *redundancy_criterion_standard_extension_fair_derive_saturated_upto*:
assumes
subs: $\Gamma \subseteq \Gamma'$ **and**
red: *redundancy_criterion* Γ *Rf Ri* **and**
red': *sat_preserving_effective_redundancy_criterion* Γ' *Rf* $(\lambda N.\ Ri\ N \cup (\Gamma' - \Gamma))$ **and**
deriv: *chain* (*redundancy_criterion.derive* Γ' *Rf*) *Ns* **and**
fair: *effective_redundancy_criterion.fair_clss_seq* Γ' *Rf* $(\lambda N.\ Ri\ N \cup (\Gamma' - \Gamma))$ *Ns*
shows *redundancy_criterion.saturated_upto* Γ *Rf Ri* (*Liminf_llist Ns*)
⟨*proof*⟩

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 $N\ \gamma \longleftrightarrow$
$(\exists\, DD.\ set\_mset\ DD \subseteq N \wedge (\forall\, I.\ I \models m\ DD + side\_prems\_of\ \gamma \longrightarrow I \models concl\_of\ \gamma)$
$\wedge\ (\forall\, D.\ D \in\#\ DD \longrightarrow D < main\_prem\_of\ \gamma))$

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

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

lemma *tautology_Rf*:
assumes *Pos A* ∈# *C*
assumes *Neg A* ∈# *C*
shows $C \in Rf\ N$
⟨*proof*⟩

lemma *tautology_redundant_infer*:
assumes
pos: *Pos A* ∈# *concl_of* ι **and**
neg: *Neg A* ∈# *concl_of* ι
shows *redundant_infer N* ι
⟨*proof*⟩

lemma *contradiction_Rf*: $\{\#\} \in N \Longrightarrow Rf\ N = UNIV - \{\{\#\}\}$
⟨*proof*⟩

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

lemma *Rf_mono*: $N \subseteq N' \Longrightarrow Rf\ N \subseteq Rf\ N'$
⟨*proof*⟩

lemma *wlog_non_Rf*:
assumes *ex*: $\exists\, DD.\ set\_mset\ DD \subseteq N \wedge (\forall\, I.\ I \models m\ DD + CC \longrightarrow I \models E) \wedge (\forall\, D'.\ D' \in\#\ DD \longrightarrow D' < D)$
shows $\exists\, DD.\ set\_mset\ DD \subseteq N - Rf\ N \wedge (\forall\, I.\ I \models m\ DD + CC \longrightarrow I \models E) \wedge (\forall\, D'.\ D' \in\#\ DD \longrightarrow D' < D)$
⟨*proof*⟩

lemma *Rf_imp_ex_non_Rf*:

assumes $C \in Rf\ N$
shows $\exists\, CC.\ set\_mset\ CC \subseteq N - Rf\ N \wedge (\forall\, I.\ I \models m\ CC \longrightarrow I \models C) \wedge (\forall\, C'.\ C' \in\# CC \longrightarrow C' < C)$
⟨*proof*⟩

lemma *Rf_subs_Rf_diff_Rf*: $Rf\ N \subseteq Rf\ (N - Rf\ N)$
⟨*proof*⟩

lemma *Rf_eq_Rf_diff_Rf*: $Rf\ N = Rf\ (N - Rf\ N)$
⟨*proof*⟩

The following results correspond to Lemma 4.6.

lemma *Ri_mono*: $N \subseteq N' \Longrightarrow Ri\ N \subseteq Ri\ N'$
⟨*proof*⟩

lemma *Ri_subs_Ri_diff_Rf*: $Ri\ N \subseteq Ri\ (N - Rf\ N)$
⟨*proof*⟩

lemma *Ri_eq_Ri_diff_Rf*: $Ri\ N = Ri\ (N - Rf\ N)$
⟨*proof*⟩

lemma *Ri_subset_Γ*: $Ri\ N \subseteq \Gamma$
⟨*proof*⟩

lemma *Rf_indep*: $N' \subseteq Rf\ N \Longrightarrow Rf\ N \subseteq Rf\ (N - N')$
⟨*proof*⟩

lemma *Ri_indep*: $N' \subseteq Rf\ N \Longrightarrow Ri\ N \subseteq Ri\ (N - N')$
⟨*proof*⟩

lemma *Rf_model*:
assumes $I \models s\ N - Rf\ N$
shows $I \models s\ N$
⟨*proof*⟩

lemma *Rf_sat*: $satisfiable\ (N - Rf\ N) \Longrightarrow satisfiable\ N$
⟨*proof*⟩

The following corresponds to Theorem 4.7:

sublocale *redundancy_criterion* Γ *Rf Ri*
⟨*proof*⟩

end

locale *standard_redundancy_criterion_reductive* =
standard_redundancy_criterion + *reductive_inference_system*
begin

The following corresponds to Theorem 4.8:

lemma *Ri_effective*:
assumes
in_γ: $\gamma \in \Gamma$ **and**
concl_of_in_n_un_rf_n: $concl\_of\ \gamma \in N \cup Rf\ N$
shows $\gamma \in Ri\ N$
⟨*proof*⟩

sublocale *effective_redundancy_criterion* Γ *Rf Ri*
⟨*proof*⟩

lemma *contradiction_Rf*: $\{\#\} \in N \Longrightarrow Ri\ N = \Gamma$
⟨*proof*⟩

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*⟩

theorem *saturated_upto_complete*:
  assumes *saturated_upto N*
  shows ¬ *satisfiable N* ⟷ {#} ∈ *N*
  ⟨*proof*⟩

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*: (∃ *xy* ∈ *A* × *B*. *P xy*) ≡ (∃ *x* ∈ *A*. ∃ *y* ∈ *B*. *P* (*x*, *y*))
  ⟨*proof*⟩

lemma *eql_map_neg_lit_eql_atm*:
  assumes *map* (λ*L*. *L* ·*l* η) (*map Neg As'*) = *map Neg As*
  shows *As'* ·*al* η = *As*
  ⟨*proof*⟩

lemma *instance_list*:
  assumes *negs* (*mset As*) = *SDA'* · η
  shows ∃ *As'*. *negs* (*mset As'*) = *SDA'* ∧ *As'* ·*al* η = *As*
⟨*proof*⟩

lemma *map2_add_mset_map*:
assumes *length AAs′ = n* **and** *length As′ = n*
shows *map2 add_mset (As′ ·al η) (AAs′ ·aml η) = map2 add_mset As′ AAs′ ·aml η*
⟨*proof*⟩

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

14.2 Calculus

The following corresponds to Figure 4.

definition *maximal_wrt* :: *'a ⇒ 'a literal multiset ⇒ bool* **where**
maximal_wrt A C ⟷ (∀ B ∈ atms_of C. ¬ less_atm A B)

definition *strictly_maximal_wrt* :: *'a ⇒ 'a literal multiset ⇒ bool* **where**
strictly_maximal_wrt A C ≡ ∀ B ∈ atms_of C. A ≠ B ∧ ¬ less_atm A B

lemma *strictly_maximal_wrt_maximal_wrt*: *strictly_maximal_wrt A C ⟹ maximal_wrt A C*
⟨*proof*⟩

lemma *maximal_wrt_subst*: *maximal_wrt (A ·a σ) (C · σ) ⟹ maximal_wrt A C*
⟨*proof*⟩

lemma *strictly_maximal_wrt_subst*:
strictly_maximal_wrt (A ·a σ) (C · σ) ⟹ strictly_maximal_wrt A C
⟨*proof*⟩

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

inductive
ord_resolve
:: *'a clause list ⇒ 'a clause ⇒ 'a multiset list ⇒ 'a list ⇒ 's ⇒ '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 ≠ {#}) ⟹
Some σ = mgu (set_mset ‘ set (map2 add_mset As AAs)) ⟹
eligible σ As (D + negs (mset As)) ⟹
(∀ i < n. strictly_maximal_wrt (As ! i ·a σ) (Cs ! i · σ)) ⟹
(∀ i < n. S (CAs ! i) = {#}) ⟹
ord_resolve CAs (D + negs (mset As)) AAs As σ ((∑# (mset Cs) + D) · σ)

inductive
ord_resolve_rename
:: *'a clause list ⇒ 'a clause ⇒ 'a multiset list ⇒ 'a list ⇒ 's ⇒ 'a clause ⇒ bool*
where
ord_resolve_rename:
length CAs = n ⟹
length AAs = n ⟹
length As = n ⟹
(∀ i < n. poss (AAs ! i) ⊆# CAs ! i) ⟹
negs (mset As) ⊆# DA ⟹
ϱ = hd (renamings_apart (DA # CAs)) ⟹
ϱs = tl (renamings_apart (DA # CAs)) ⟹

ord_resolve (CAs ··cl ϱs) (DA · ϱ) (AAs ··aml ϱs) (As ·al ϱ) σ E ⟹
ord_resolve_rename CAs DA AAs As σ E

lemma *ord_resolve_empty_main_prem*: ¬ *ord_resolve Cs {#} AAs As σ E*
⟨*proof*⟩

lemma *ord_resolve_rename_empty_main_prem*: ¬ *ord_resolve_rename Cs {#} AAs As σ E*
⟨*proof*⟩

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 σ E* **and**
cc_inst_true: *I* ⊨*m mset CAs ·cm σ ·cm η* **and**
d_inst_true: *I* ⊨ *DA · σ · η* **and**
ground_subst_η: *is_ground_subst η*
shows *I* ⊨ *E · η*
⟨*proof*⟩

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*
⟨*proof*⟩

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: ⋀*σ*. *is_ground_subst σ* ⟹ *I* ⊨*m* (*mset CAs* + {#*DA*#}) *·cm σ* **and**
ground_subst_η: *is_ground_subst η*
shows *I* ⊨ *E · η*
⟨*proof*⟩

lemma *subst_sound*:
assumes
⋀*σ*. *is_ground_subst σ* ⟹ *I* ⊨ *C · σ* **and**
is_ground_subst η
shows *I* ⊨ *C · ϱ · η*
⟨*proof*⟩

lemma *subst_sound_scl*:
assumes
len: *length P* = *length CAs* **and**
true_cas: ⋀*σ*. *is_ground_subst σ* ⟹ *I* ⊨*m mset CAs ·cm σ* **and**
ground_subst_η: *is_ground_subst η*
shows *I* ⊨*m mset* (*CAs ··cl P*) *·cm η*
⟨*proof*⟩

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: $\bigwedge\sigma$. *is_ground_subst* $\sigma \Longrightarrow I \models m ((mset\ CAs) + \{\#DA\#\}) \cdot cm\ \sigma$ **and**
ground_subst_η: *is_ground_subst* η
shows $I \models E \cdot \eta$
⟨*proof*⟩

14.4 Other Basic Properties

lemma *ord_resolve_unique*:
assumes
ord_resolve CAs DA AAs As σ E **and**
ord_resolve CAs DA AAs As σ' E'
shows $\sigma = \sigma' \wedge E = E'$
⟨*proof*⟩

lemma *ord_resolve_rename_unique*:
assumes
ord_resolve_rename CAs DA AAs As σ E **and**
ord_resolve_rename CAs DA AAs As σ' E'
shows $\sigma = \sigma' \wedge E = E'$
⟨*proof*⟩

lemma *ord_resolve_max_side_prems*: *ord_resolve CAs DA AAs As* σ $E \Longrightarrow$ *length CAs* $\leq$ *size DA*
⟨*proof*⟩

lemma *ord_resolve_rename_max_side_prems*:
ord_resolve_rename CAs DA AAs As σ $E \Longrightarrow$ *length CAs* $\leq$ *size DA*
⟨*proof*⟩

14.5 Inference System

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

interpretation *ord_FO_resolution*: *inference_system ord_FO_*Γ ⟨*proof*⟩

lemma *finite_ord_FO_resolution_inferences_between*:
assumes *fin_cc*: *finite CC*
shows *finite* (*ord_FO_resolution.inferences_between CC C*)
⟨*proof*⟩

lemma *ord_FO_resolution_inferences_between_empty_empty*:
ord_FO_resolution.inferences_between {} {#} = {}
⟨*proof*⟩

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*
⟨*proof*⟩

definition *S_M* :: *'a literal multiset* ⇒ *'a literal multiset* **where**
S_M C =
(*if* $C \in$ *grounding_of_clss M then*
(*SOME* C'. $\exists D\ \sigma$. $D \in M \wedge C = D \cdot \sigma \wedge C' = S\ D \cdot \sigma \wedge$ *is_ground_subst* σ)
else
S C)

lemma *S_M_grounding_of_clss*:

assumes $C \in grounding\_of\_clss\ M$
obtains $D\ \sigma$ **where**
$D \in M \wedge C = D \cdot \sigma \wedge S\_M\ C = S\ D \cdot \sigma \wedge is\_ground\_subst\ \sigma$
⟨*proof*⟩

lemma *S_M_not_grounding_of_clss*: $C \notin grounding\_of\_clss\ M \Longrightarrow S\_M\ C = S\ C$
⟨*proof*⟩

lemma *S_M_selects_subseteq*: $S\_M\ C \subseteq\#\ C$
⟨*proof*⟩

lemma *S_M_selects_neg_lits*: $L \in\#\ S\_M\ C \Longrightarrow is\_neg\ L$
⟨*proof*⟩

end

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: $ord\_resolve\ S\ CAs\ DA\ AAs\ As\ \sigma\ E$
shows $E \subseteq\#\ \sum_{\#}\ (mset\ CAs) + DA$
⟨*proof*⟩

lemma *ord_resolve_obtain_clauses*:
assumes
res_e: $ord\_resolve\ (S\_M\ S\ M)\ CAs\ DA\ AAs\ As\ \sigma\ E$ **and**
select: $selection\ S$ **and**
grounding: $\{DA\} \cup set\ CAs \subseteq grounding\_of\_clss\ M$ **and**
n: $length\ CAs = n$ **and**
d: $DA = D + negs\ (mset\ As)$ **and**
c: $(\forall\, i < n.\ CAs\ !\ i = Cs\ !\ i + poss\ (AAs\ !\ i))\ length\ Cs = n\ length\ AAs = n$
obtains $DA0\ \eta 0\ CAs0\ \eta s0\ As0\ AAs0\ D0\ Cs0$ **where**
$length\ CAs0 = n$
$length\ \eta s0 = n$
$DA0 \in M$
$DA0 \cdot \eta 0 = DA$
$S\ DA0 \cdot \eta 0 = S\_M\ S\ M\ DA$
$\forall\, CA0 \in set\ CAs0.\ CA0 \in M$
$CAs0 \cdot\cdot cl\ \eta s0 = CAs$
$map\ S\ CAs0 \cdot\cdot cl\ \eta s0 = map\ (S\_M\ S\ M)\ CAs$
$is\_ground\_subst\ \eta 0$
$is\_ground\_subst\_list\ \eta s0$
$As0\ \cdot al\ \eta 0 = As$
$AAs0 \cdot\cdot aml\ \eta s0 = AAs$
$length\ As0 = n$
$D0 \cdot \eta 0 = D$
$DA0 = D0 + (negs\ (mset\ As0))$
$S\_M\ S\ M\ (D + negs\ (mset\ As)) \neq \{\#\} \Longrightarrow negs\ (mset\ As0) = S\ DA0$
$length\ Cs0 = n$
$Cs0 \cdot\cdot cl\ \eta s0 = Cs$
$\forall\, i < n.\ CAs0\ !\ i = Cs0\ !\ i + poss\ (AAs0\ !\ i)$
$length\ AAs0 = n$
⟨*proof*⟩

lemma *ord_resolve_rename_lifting*:
assumes
sel_stable: $\bigwedge \varrho\ C.\ is\_renaming\ \varrho \Longrightarrow S\ (C \cdot \varrho) = S\ C \cdot \varrho$ **and**
res_e: $ord\_resolve\ (S\_M\ S\ M)\ CAs\ DA\ AAs\ As\ \sigma\ E$ **and**
select: $selection\ S$ **and**

grounding: $\{DA\} \cup set\ CAs \subseteq grounding\_of\_clss\ M$
obtains $\eta s\ \eta\ \eta 2\ CAs0\ DA0\ AAs0\ As0\ E0\ \tau$ **where**
is_ground_subst η
is_ground_subst_list ηs
is_ground_subst $\eta 2$
ord_resolve_rename $S\ CAs0\ DA0\ AAs0\ As0\ \tau\ E0$
$CAs0 \cdot\cdot cl\ \eta s = CAs\ DA0 \cdot \eta = DA\ E0 \cdot \eta 2 = E$
$\{DA0\} \cup set\ CAs0 \subseteq M$
$length\ CAs0 = length\ CAs$
$length\ \eta s = length\ CAs$
⟨*proof*⟩

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*⟩

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 ‹$\exists \sigma.$ *ord_resolve* $S\_G\ CAs\ DA\ AAs\ As\ \sigma\ E$›
⟨*proof*⟩

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* :: $(nat \Rightarrow bool) \Rightarrow nat \Rightarrow bool$ **where**
$is\_least\ P\ n \longleftrightarrow P\ n \wedge (\forall n' < n.\ \neg\ P\ n')$

lemma *least_exists*: $P\ n \Longrightarrow \exists n.\ is\_least\ P\ n$
⟨*proof*⟩

The following corresponds to page 42 and 43 of Section 4.3, from the explanation of RP to Lemma 4.10.

type-synonym $'a\ state = 'a\ clause\ set \times 'a\ clause\ set \times 'a\ clause\ set$

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* $\Rightarrow$ $'a$ *clause* **and**
subst_atm :: $'a \Rightarrow 's \Rightarrow 'a$ **and**
id_subst :: $'s$ **and**
comp_subst :: $'s \Rightarrow 's \Rightarrow 's$ **and**
renamings_apart :: $'a$ *clause list* $\Rightarrow$ $'s$ *list* **and**
atm_of_atms :: $'a$ *list* $\Rightarrow$ $'a$ **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 ⟨proof⟩
```

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'#} ∈ N ⟹ − L = L' ·l σ ⟹ D · σ ⊆# C ⟹
    (N, P, Q ∪ {C + {#L#}}) ⇝ (N, P ∪ {C}, Q)
| clause_processing: (N ∪ {C}, P, Q) ⇝ (N, P ∪ {C}, Q)
| inference_computation: N = concls_of (ord_FO_resolution.inferences_between Q C) ⟹
    ({}, P ∪ {C}, Q) ⇝ (N, P, Q ∪ {C})

lemma final_RP: ¬ ({}, {}, Q) ⇝ St
  ⟨proof⟩

definition Sup_state :: 'a state llist ⇒ '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 ⇒ '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' ¬ lnull Sts ¬ lnull Sts' llast Sts' = llast Sts
begin

lemma
  N_of_Liminf_state_fin: N_of_state (Liminf_state Sts') = N_of_state (Liminf_state Sts) and
  P_of_Liminf_state_fin: P_of_state (Liminf_state Sts') = P_of_state (Liminf_state Sts) and
  Q_of_Liminf_state_fin: Q_of_state (Liminf_state Sts') = Q_of_state (Liminf_state Sts)
  ⟨proof⟩
```

lemma *Liminf_state_fin*: *Liminf_state Sts'* = *Liminf_state Sts*
⟨*proof*⟩

end

context
fixes *Sts Sts'* :: *'a state llist*
assumes *Sts*: ¬ *lfinite Sts emb Sts Sts'*
begin

lemma
N_of_Liminf_state_inf: *N_of_state* (*Liminf_state Sts'*) ⊆ *N_of_state* (*Liminf_state Sts*) **and**
P_of_Liminf_state_inf: *P_of_state* (*Liminf_state Sts'*) ⊆ *P_of_state* (*Liminf_state Sts*) **and**
Q_of_Liminf_state_inf: *Q_of_state* (*Liminf_state Sts'*) ⊆ *Q_of_state* (*Liminf_state Sts*)
⟨*proof*⟩

lemma *clss_of_Liminf_state_inf*:
clss_of_state (*Liminf_state Sts'*) ⊆ *clss_of_state* (*Liminf_state Sts*)
⟨*proof*⟩

end

definition *fair_state_seq* :: *'a state llist* ⇒ *bool* **where**
fair_state_seq Sts ⟷ *N_of_state* (*Liminf_state Sts*) = {} ∧ *P_of_state* (*Liminf_state Sts*) = {}

The following formalizes Lemma 4.10.

context
fixes *Sts* :: *'a state llist*
begin

definition *S_Q* :: *'a clause* ⇒ *'a clause* **where**
S_Q = *S_M S* (*Q_of_state* (*Liminf_state Sts*))

interpretation *sq*: *selection S_Q*
⟨*proof*⟩

interpretation *gr*: *ground_resolution_with_selection S_Q*
⟨*proof*⟩

interpretation *sr*: *standard_redundancy_criterion_reductive gr.ord_*Γ
⟨*proof*⟩

interpretation *sr*: *standard_redundancy_criterion_counterex_reducing gr.ord_*Γ
ground_resolution_with_selection.INTERP S_Q
⟨*proof*⟩

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
⟨*proof*⟩

lemma *sound_ground_sound_*Γ: *sound_inference_system ground_sound_*Γ
⟨*proof*⟩

lemma *sat_preserving_ground_sound_*Γ: *sat_preserving_inference_system ground_sound_*Γ
⟨*proof*⟩

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
⟨*proof*⟩

lemma *strict_subset_subsumption_redundant_clause*:
assumes
sub: $D \cdot \sigma \subset\# C$ **and**
ground_σ: *is_ground_subst σ*
shows $C \in$ *sr.Rf* (*grounding_of_cls D*)
⟨*proof*⟩

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* (*grounding_of_clss CC*)
⟨*proof*⟩

lemma *strict_subset_subsumption_grounding_redundant_clss*:
assumes
Dσ_subset_C: $D \cdot \sigma \subset\# C$ **and**
D_in_St: $D \in CC$
shows *grounding_of_cls C* $\subseteq$ *sr.Rf* (*grounding_of_clss CC*)
⟨*proof*⟩

lemma *derive_if_remove_subsumed*:
assumes
$D \in$ *clss_of_state St* **and**
subsumes D C
shows *sr_ext.derive* (*grounding_of_state St* ∪ *grounding_of_cls C*) (*grounding_of_state St*)
⟨*proof*⟩

lemma *reduction_in_concls_of*:
assumes
$C\mu \in$ *grounding_of_cls C* **and**
$D + \{\#L'\#\} \in CC$ **and**
$-L = L' \cdot l\ \sigma$ **and**
$D \cdot \sigma \subseteq\# C$
shows $C\mu \in$ *concls_of* (*sr_ext.inferences_from* (*grounding_of_clss* ($CC \cup \{C + \{\#L\#\}\}$)))
⟨*proof*⟩

lemma *reduction_derivable*:
assumes
$D + \{\#L'\#\} \in CC$ **and**
$-L = L' \cdot l\ \sigma$ **and**
$D \cdot \sigma \subseteq\# C$
shows *sr_ext.derive* (*grounding_of_clss* ($CC \cup \{C + \{\#L\#\}\}$)) (*grounding_of_clss* ($CC \cup \{C\}$))
⟨*proof*⟩

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

lemma *RP_ground_derive*:
$St \rightsquigarrow St' \Longrightarrow$ *sr_ext.derive* (*grounding_of_state St*) (*grounding_of_state St'*)
⟨*proof*⟩

A useful consequence:

theorem *RP_model*: $St \rightsquigarrow St' \Longrightarrow I \models s$ *grounding_of_state* $St' \longleftrightarrow I \models s$ *grounding_of_state St*
⟨*proof*⟩

Another formulation of the part of Lemma 4.10 that states we have a theorem proving process:

lemma *ground_derive_chain*: *chain* (⇝) *Sts* ⟹ *chain sr_ext.derive* (*lmap grounding_of_state Sts*)
⟨*proof*⟩

The following is used prove to Lemma 4.11:

lemma *Sup_llist_grounding_of_state_ground*:
  assumes $C \in$ *Sup_llist* (*lmap grounding_of_state Sts*)
  shows *is_ground_cls C*
⟨*proof*⟩

lemma *Liminf_grounding_of_state_ground*:
  $C \in$ *Liminf_llist* (*lmap grounding_of_state Sts*) $\Longrightarrow$ *is_ground_cls C*
  ⟨*proof*⟩

lemma *in_Sup_llist_in_Sup_state*:
  assumes $C \in$ *Sup_llist* (*lmap grounding_of_state Sts*)
  shows $\exists\, D\ \sigma.\ D \in$ *clss_of_state* (*Sup_state Sts*) $\wedge\ D \cdot \sigma = C \wedge$ *is_ground_subst* σ
⟨*proof*⟩

lemma
  N_of_state_Liminf: *N_of_state* (*Liminf_state Sts*) = *Liminf_llist* (*lmap N_of_state Sts*) **and**
  P_of_state_Liminf: *P_of_state* (*Liminf_state Sts*) = *Liminf_llist* (*lmap P_of_state Sts*)
  ⟨*proof*⟩

lemma *eventually_removed_from_N*:
  assumes
    d_in: $D \in$ *N_of_state* (*lnth Sts i*) **and**
    fair: *fair_state_seq Sts* **and**
    i_Sts: *enat i* < *llength Sts*
  shows $\exists\, l.\ D \in$ *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*
⟨*proof*⟩

lemma *eventually_removed_from_P*:
  assumes
    d_in: $D \in$ *P_of_state* (*lnth Sts i*) **and**
    fair: *fair_state_seq Sts* **and**
    i_Sts: *enat i* < *llength Sts*
  shows $\exists\, l.\ D \in$ *P_of_state* (*lnth Sts l*) $\wedge\ D \notin$ *P_of_state* (*lnth Sts* (*Suc l*)) $\wedge\ i \leq l$
    $\wedge$ *enat* (*Suc l*) < *llength Sts*
⟨*proof*⟩

lemma *instance_if_subsumed_and_in_limit*:
  assumes
    deriv: *chain* ($\rightsquigarrow$) *Sts* **and**
    ns: *Gs* = *lmap grounding_of_state Sts* **and**
    c: $C \in$ *Liminf_llist Gs* − *sr.Rf* (*Liminf_llist Gs*) **and**
    d: $D \in$ *clss_of_state* (*lnth Sts i*) *enat i* < *llength Sts subsumes D C*
  shows $\exists \sigma.\ D \cdot \sigma = C \wedge$ *is_ground_subst* σ
⟨*proof*⟩

lemma *from_Q_to_Q_inf*:
  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: $D \in$ *Q_of_state* (*lnth Sts i*) *enat i* < *llength Sts subsumes D C* **and**
    d_least: $\forall\, E \in \{E.\ E \in$ (*clss_of_state* (*Sup_state Sts*)) $\wedge$ *subsumes E C*$\}$.
      ¬ *strictly_subsumes E D*
  shows $D \in$ *Q_of_state* (*Liminf_state Sts*)
⟨*proof*⟩

lemma *from_P_to_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: $D \in P\_of\_state\ (lnth\ Sts\ i)\ enat\ i < llength\ Sts\ subsumes\ D\ C$ **and**
d_least: $\forall E \in \{E.\ E \in (clss\_of\_state\ (Sup\_state\ Sts)) \wedge subsumes\ E\ C\}.$
$\neg\ strictly\_subsumes\ E\ D$
shows $\exists l.\ D \in Q\_of\_state\ (lnth\ Sts\ l) \wedge enat\ l < llength\ Sts$
⟨*proof*⟩

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: $D \in N\_of\_state\ (lnth\ Sts\ i)\ enat\ i < llength\ Sts\ subsumes\ D\ C$ **and**
d_least: $\forall E \in \{E.\ E \in (clss\_of\_state\ (Sup\_state\ Sts)) \wedge subsumes\ E\ C\}.\ \neg\ strictly\_subsumes\ E\ D$
shows $\exists l\ D'\ \sigma'.\ D' \in P\_of\_state\ (lnth\ Sts\ l) \cup Q\_of\_state\ (lnth\ Sts\ l) \wedge$
$enat\ l < llength\ Sts \wedge$
$(\forall E \in \{E.\ E \in (clss\_of\_state\ (Sup\_state\ Sts)) \wedge subsumes\ E\ C\}.\ \neg\ strictly\_subsumes\ E\ D') \wedge$
$D' \cdot \sigma' = C \wedge is\_ground\_subst\ \sigma' \wedge subsumes\ D'\ C$
⟨*proof*⟩

lemma *eventually_in_Qinf*:
assumes
deriv: $chain\ (\rightsquigarrow)\ Sts$ **and**
D_p: $D \in clss\_of\_state\ (Sup\_state\ Sts)$
$subsumes\ D\ C\ \forall E \in \{E.\ E \in (clss\_of\_state\ (Sup\_state\ Sts)) \wedge subsumes\ E\ C\}.$
$\neg\ strictly\_subsumes\ E\ D$ **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**
ground_C: $is\_ground\_cls\ C$
shows $\exists D'\ \sigma'.\ D' \in Q\_of\_state\ (Liminf\_state\ Sts) \wedge D' \cdot \sigma' = C \wedge is\_ground\_subst\ \sigma'$
⟨*proof*⟩

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 $Liminf\_llist\ Gs - sr.Rf\ (Liminf\_llist\ Gs)$
$\subseteq grounding\_of\_clss\ (Q\_of\_state\ (Liminf\_state\ Sts))$
⟨*proof*⟩

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

lemma *subseteq_Liminf_state_eventually_always*:
fixes *CC*
assumes
finite CC **and**
$CC \neq \{\}$ **and**
$CC \subseteq Q\_of\_state\ (Liminf\_state\ Sts)$
shows $\exists j.\ enat\ j < llength\ Sts \wedge (\forall j' \geq enat\ j.\ j' < llength\ Sts \longrightarrow CC \subseteq Q\_of\_state\ (lnth\ Sts\ j'))$
⟨*proof*⟩

lemma *empty_clause_in_Q_of_Liminf_state*:
assumes
deriv: $chain\ (\rightsquigarrow)\ Sts$ **and**
fair: $fair\_state\_seq\ Sts$ **and**
empty_in: $\{\#\} \in Liminf\_llist\ (lmap\ grounding\_of\_state\ Sts)$
shows $\{\#\} \in Q\_of\_state\ (Liminf\_state\ Sts)$
⟨*proof*⟩

lemma *grounding_of_state_Liminf_state_subseteq*:
grounding_of_state (*Liminf_state Sts*) ⊆ *Liminf_llist* (*lmap grounding_of_state Sts*)
⟨*proof*⟩

theorem *RP_sound*:
assumes
deriv: *chain* (⇝) *Sts* **and**
{#} ∈ *clss_of_state* (*Liminf_state Sts*)
shows ¬ *satisfiable* (*grounding_of_state* (*lhd Sts*))
⟨*proof*⟩

theorem *RP_saturated_if_fair*:
assumes
deriv: *chain* (⇝) *Sts* **and**
fair: *fair_state_seq Sts* **and**
empty_Q0: *Q_of_state* (*lhd Sts*) = {}
shows *sr.saturated_upto* (*Liminf_llist* (*lmap grounding_of_state Sts*))
⟨*proof*⟩

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*⟩

end

end

end