

A Verified Functional Implementation of Bachmair and Ganzinger’s Ordered Resolution Prover

Anders Schlichtkrull, Jasmin Christian Blanchette, and Dmitriy Traytel

February 6, 2026

Abstract

This Isabelle/HOL formalization refines the abstract ordered resolution prover presented in Section 4.3 of Bachmair and Ganzinger’s “Resolution Theorem Proving” chapter in the *Handbook of Automated Reasoning*. The result is a functional implementation of a first-order prover.

Contents

1	Introduction	1
2	A Fair Ordered Resolution Prover for First-Order Clauses with Weights	1
3	A Deterministic Ordered Resolution Prover for First-Order Clauses	8
3.1	Library	8
3.2	Prover	8
4	Integration of IsaFoR Terms and the Knuth–Bendix Order	17
5	An Executable Algorithm for Clause Subsumption	22
5.1	Naive Implementation of Clause Subsumption	22
5.2	Optimized Implementation of Clause Subsumption	24
5.3	Definition of Deterministic QuickSort	24
6	An Executable Simple Ordered Resolution Prover for First-Order Clauses	25

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 starts from an existing formalization of Bachmair and Ganzinger’s chapter, up to and including Section 4.3. It refines the abstract ordered resolution prover presented in Section 4.3 to obtain an executable, functional implementation of a first-order prover. Figure 1 shows the corresponding Isabelle theory structure.

We refer to the following conference paper for details:

Anders Schlichtkrull, Jasmin Christian Blanchette, Dmitriy Traytel:
A verified prover based on ordered resolution.
CPP 2019: 152-165
http://matryoshka.gforge.inria.fr/pubs/fun_rp_paper.pdf

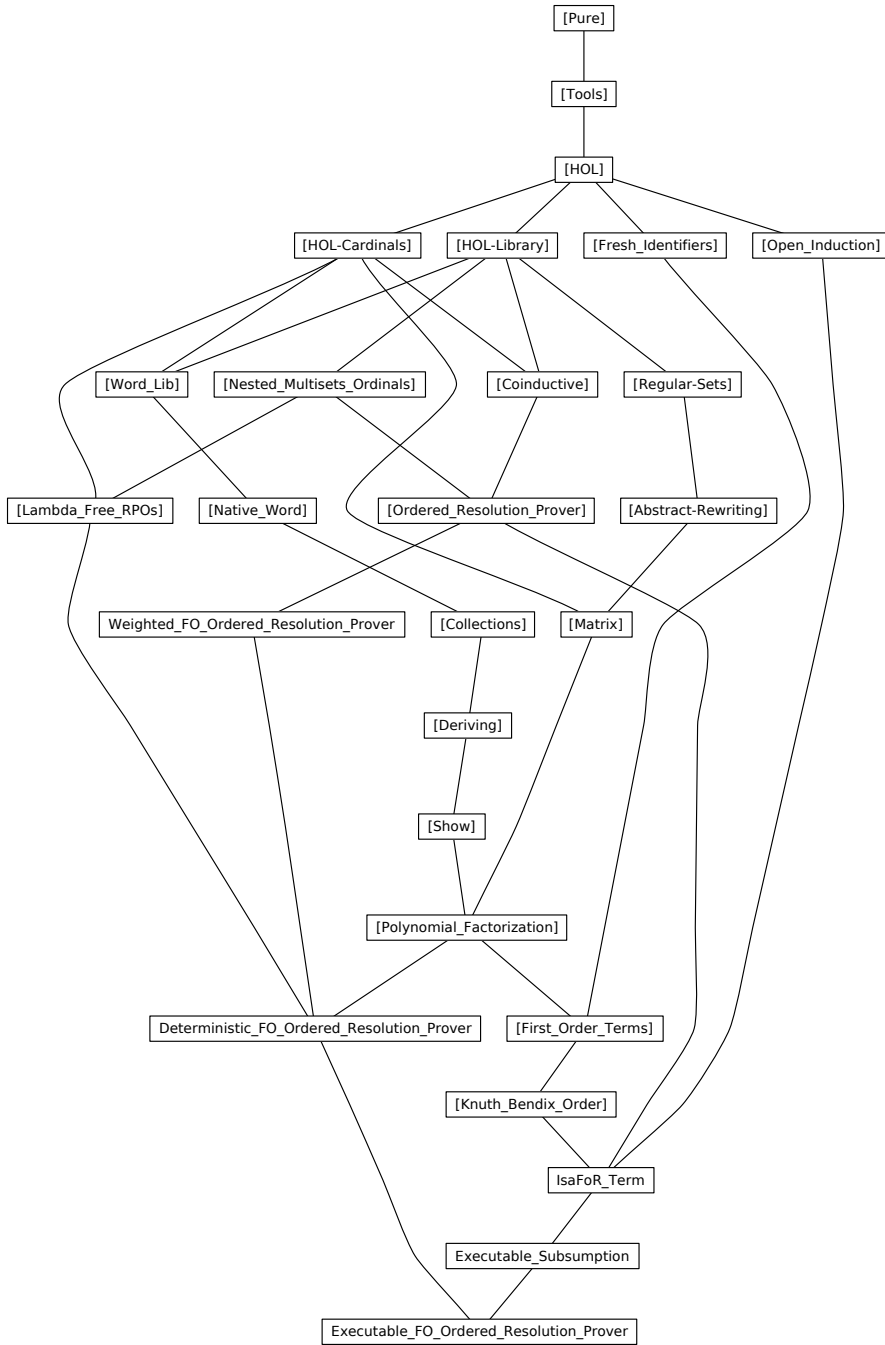


Figure 1: Theory dependency graph

2 A Fair Ordered Resolution Prover for First-Order Clauses with Weights

The *weighted_RP* prover introduced below operates on finite multisets of clauses and organizes the multiset of processed clauses as a priority queue to ensure that inferences are performed in a fair manner, to guarantee completeness.

```

theory Weighted_FO_Ordered_Resolution_Prover
  imports Ordered_Resolution_Prover.FO_Ordered_Resolution_Prover
begin

type-synonym 'a wclause = 'a clause × nat
type-synonym 'a wstate = 'a wclause multiset × 'a wclause multiset × 'a wclause multiset × nat

fun state_of_wstate :: 'a wstate ⇒ 'a state where
  state_of_wstate (N, P, Q, n) =
    (set_mset (image_mset fst N), set_mset (image_mset fst P), set_mset (image_mset fst Q))

locale weighted_FO_resolution_prover =
  FO_resolution_prover S subst_atm id_subst comp_subst renamings_apart atm_of_atms mgu less_atm
for
  S :: ('a :: wellorder) clause ⇒ 'a clause and
  subst_atm :: 'a ⇒ 's ⇒ 'a and
  id_subst :: 's and
  comp_subst :: 's ⇒ 's ⇒ 's and
  renamings_apart :: 'a clause list ⇒ 's list and
  atm_of_atms :: 'a list ⇒ 'a and
  mgu :: 'a set set ⇒ 's option and
  less_atm :: 'a ⇒ 'a ⇒ bool +
fixes
  weight :: 'a clause × nat ⇒ nat
assumes
  weight_mono: i < j ⇒ weight (C, i) < weight (C, j)
begin

abbreviation cls_of_wstate :: 'a wstate ⇒ 'a clause set where
  cls_of_wstate St ≡ cls_of_state (state_of_wstate St)

abbreviation N_of_wstate :: 'a wstate ⇒ 'a clause set where
  N_of_wstate St ≡ N_of_state (state_of_wstate St)

abbreviation P_of_wstate :: 'a wstate ⇒ 'a clause set where
  P_of_wstate St ≡ P_of_state (state_of_wstate St)

abbreviation Q_of_wstate :: 'a wstate ⇒ 'a clause set where
  Q_of_wstate St ≡ Q_of_state (state_of_wstate St)

fun wN_of_wstate :: 'a wstate ⇒ 'a wclause multiset where
  wN_of_wstate (N, P, Q, n) = N

fun wP_of_wstate :: 'a wstate ⇒ 'a wclause multiset where
  wP_of_wstate (N, P, Q, n) = P

fun wQ_of_wstate :: 'a wstate ⇒ 'a wclause multiset where
  wQ_of_wstate (N, P, Q, n) = Q

fun n_of_wstate :: 'a wstate ⇒ nat where
  n_of_wstate (N, P, Q, n) = n

lemma of_wstate_split[simp]:
  (wN_of_wstate St, wP_of_wstate St, wQ_of_wstate St, n_of_wstate St) = St
  (proof)

```

abbreviation *grounding_of_wstate* :: 'a wstate \Rightarrow 'a clause set **where**
grounding_of_wstate St \equiv *grounding_of_state* (state_of_wstate St)

abbreviation *Liminf_wstate* :: 'a wstate llist \Rightarrow 'a state **where**
Liminf_wstate Sts \equiv *Liminf_state* (lmap state_of_wstate Sts)

lemma *timestamp_le_weight*: $n \leq \text{weight } (C, n)$
 ⟨proof⟩

inductive *weighted_RP* :: 'a wstate \Rightarrow 'a wstate \Rightarrow bool (**infix** $\langle \rightsquigarrow_w \rangle$ 50) **where**
tautology_deletion: $\text{Neg } A \in \# C \Longrightarrow \text{Pos } A \in \# C \Longrightarrow (N + \{\#(C, i)\}, P, Q, n) \rightsquigarrow_w (N, P, Q, n)$
 | *forward_subsumption*: $D \in \# \text{image_mset fst } (P + Q) \Longrightarrow \text{subsumes } D C \Longrightarrow$
 $(N + \{\#(C, i)\}, P, Q, n) \rightsquigarrow_w (N, P, Q, n)$
 | *backward_subsumption_P*: $D \in \# \text{image_mset fst } N \Longrightarrow C \in \# \text{image_mset fst } P \Longrightarrow$
 $\text{strictly_subsumes } D C \Longrightarrow (N, P, Q, n) \rightsquigarrow_w (N, \{\#(E, k) \in \# P. E \neq C\}, Q, n)$
 | *backward_subsumption_Q*: $D \in \# \text{image_mset fst } N \Longrightarrow \text{strictly_subsumes } D C \Longrightarrow$
 $(N, P, Q + \{\#(C, i)\}, n) \rightsquigarrow_w (N, P, Q, n)$
 | *forward_reduction*: $D + \{\#L'\} \in \# \text{image_mset fst } (P + Q) \Longrightarrow -L = L' \cdot l \sigma \Longrightarrow D \cdot \sigma \subseteq \# C \Longrightarrow$
 $(N + \{\#(C + \{\#L'\}, i)\}, P, Q, n) \rightsquigarrow_w (N + \{\#(C, i)\}, P, Q, n)$
 | *backward_reduction_P*: $D + \{\#L'\} \in \# \text{image_mset fst } N \Longrightarrow -L = L' \cdot l \sigma \Longrightarrow D \cdot \sigma \subseteq \# C \Longrightarrow$
 $(\forall j. (C + \{\#L'\}, j) \in \# P \longrightarrow j \leq i) \Longrightarrow$
 $(N, P + \{\#(C + \{\#L'\}, i)\}, Q, n) \rightsquigarrow_w (N, P + \{\#(C, i)\}, Q, n)$
 | *backward_reduction_Q*: $D + \{\#L'\} \in \# \text{image_mset fst } N \Longrightarrow -L = L' \cdot l \sigma \Longrightarrow D \cdot \sigma \subseteq \# C \Longrightarrow$
 $(N, P, Q + \{\#(C + \{\#L'\}, i)\}, n) \rightsquigarrow_w (N, P + \{\#(C, i)\}, Q, n)$
 | *clause_processing*: $(N + \{\#(C, i)\}, P, Q, n) \rightsquigarrow_w (N, P + \{\#(C, i)\}, Q, n)$
 | *inference_computation*: $(\forall (D, j) \in \# P. \text{weight } (C, i) \leq \text{weight } (D, j)) \Longrightarrow$
 $N = \text{mset_set } ((\lambda D. (D, n)) \text{ `concls_of$
 $(\text{inference_system.inferences_between } (\text{ord_FO_}\Gamma S) (\text{set_mset } (\text{image_mset fst } Q)) C)) \Longrightarrow$
 $(\{\#\}, P + \{\#(C, i)\}, Q, n) \rightsquigarrow_w (N, \{\#(D, j) \in \# P. D \neq C\}, Q + \{\#(C, i)\}, \text{Suc } n)$

lemma *weighted_RP_imp_RP*: $St \rightsquigarrow_w St' \Longrightarrow \text{state_of_wstate } St \rightsquigarrow \text{state_of_wstate } St'$
 ⟨proof⟩

lemma *final_weighted_RP*: $\neg (\{\#\}, \{\#\}, Q, n) \rightsquigarrow_w St$
 ⟨proof⟩

context
fixes

Sts :: 'a wstate llist

assumes

full_deriv: *full_chain* (\rightsquigarrow_w) *Sts* **and**

empty_P0: *P_of_wstate* (lhd *Sts*) = {} **and**

empty_Q0: *Q_of_wstate* (lhd *Sts*) = {}

begin

lemma *finite_Sts0*: *finite* (*class_of_wstate* (lhd *Sts*))
 ⟨proof⟩

lemmas *deriv = full_chain_imp_chain*[OF *full_deriv*]

lemmas *lhd_lmap_Sts = llist.map_sel(1)*[OF *chain_not_lnull*[OF *deriv*]]

lemma *deriv_RP*: *chain* (\rightsquigarrow) (lmap state_of_wstate *Sts*)
 ⟨proof⟩

lemma *finite_Sts0_RP*: *finite* (*class_of_state* (lhd (lmap state_of_wstate *Sts*)))
 ⟨proof⟩

lemma *empty_P0_RP*: *P_of_state* (lhd (lmap state_of_wstate *Sts*)) = {}
 ⟨proof⟩

lemma *empty_Q0_RP*: *Q_of_state* (lhd (lmap state_of_wstate *Sts*)) = {}
 ⟨proof⟩

lemmas $Sts_thms = deriv_RP\ finite_Sts0_RP\ empty_P0_RP\ empty_Q0_RP$

theorem $weighted_RP_model$:

$St \rightsquigarrow_w St' \implies I \models_s grounding_of_wstate\ St' \longleftrightarrow I \models_s grounding_of_wstate\ St$
{proof}

abbreviation $S_gQ :: 'a\ clause \Rightarrow 'a\ clause\ \mathbf{where}$

$S_gQ \equiv S_Q\ (lmap\ state_of_wstate\ Sts)$

interpretation sq : selection S_gQ

{proof}

interpretation gd : ground_resolution_with_selection S_gQ

{proof}

interpretation src : standard_redundancy_criterion_reductive $gd.ord_F$

{proof}

interpretation src : standard_redundancy_criterion_counterex_reducing $gd.ord_F$

ground_resolution_with_selection.INTERP S_gQ

{proof}

lemmas $ord_F_saturated_upto_def = src.saturated_upto_def$

lemmas $ord_F_saturated_upto_complete = src.saturated_upto_complete$

lemmas $ord_F_contradiction_Rf = src.contradiction_Rf$

theorem $weighted_RP_sound$:

assumes $\{\#\} \in cls_of_state\ (Liminf_wstate\ Sts)$

shows $\neg\ satisfiable\ (grounding_of_wstate\ (lhd\ Sts))$

{proof}

abbreviation $RP_filtered_measure :: ('a\ wclause \Rightarrow bool) \Rightarrow 'a\ wstate \Rightarrow nat \times nat \times nat\ \mathbf{where}$

$RP_filtered_measure \equiv \lambda p\ (N, P, Q, n).$

$(sum_mset\ (image_mset\ (\lambda(C, i). Suc\ (size\ C))\ \{\#Di \in\# N + P + Q.\ p\ Di\#\}),$

$size\ \{\#Di \in\# N.\ p\ Di\#\}, size\ \{\#Di \in\# P.\ p\ Di\#\})$

abbreviation $RP_combined_measure :: nat \Rightarrow 'a\ wstate \Rightarrow nat \times (nat \times nat \times nat) \times (nat \times nat \times nat)\ \mathbf{where}$

$RP_combined_measure \equiv \lambda w\ St.$

$(w + 1 - n_of_wstate\ St, RP_filtered_measure\ (\lambda(C, i). i \leq w)\ St,$

$RP_filtered_measure\ (\lambda Ci.\ True)\ St)$

abbreviation (input) $RP_filtered_relation :: ((nat \times nat \times nat) \times (nat \times nat \times nat))\ set\ \mathbf{where}$

$RP_filtered_relation \equiv natLess\ <*\lex*\>\ natLess\ <*\lex*\>\ natLess$

abbreviation (input) $RP_combined_relation :: ((nat \times ((nat \times nat \times nat) \times (nat \times nat \times nat))) \times$

$(nat \times ((nat \times nat \times nat) \times (nat \times nat \times nat))))\ set\ \mathbf{where}$

$RP_combined_relation \equiv natLess\ <*\lex*\>\ RP_filtered_relation\ <*\lex*\>\ RP_filtered_relation$

abbreviation $(fst3 :: 'b * 'c * 'd \Rightarrow 'b) \equiv fst$

abbreviation $(snd3 :: 'b * 'c * 'd \Rightarrow 'c) \equiv \lambda x.\ fst\ (snd\ x)$

abbreviation $(trd3 :: 'b * 'c * 'd \Rightarrow 'd) \equiv \lambda x.\ snd\ (snd\ x)$

lemma

$wf_RP_filtered_relation$: wf $RP_filtered_relation$ **and**

$wf_RP_combined_relation$: wf $RP_combined_relation$

{proof}

lemma $multiset_sum_of_Suc_f_monotone$: $N \subset\# M \implies (\sum x \in\# N.\ Suc\ (f\ x)) < (\sum x \in\# M.\ Suc\ (f\ x))$

{proof}

lemma $multiset_sum_monotone_f'$:

assumes $CC \subset\# DD$

shows $(\sum (C, i) \in\# CC.\ Suc\ (f\ C)) < (\sum (C, i) \in\# DD.\ Suc\ (f\ C))$

<proof>

lemma *filter_mset_strict_subset*:

assumes $x \in\# M$ **and** $\neg p x$

shows $\{\#y \in\# M. p y\# \} \subset\# M$

<proof>

lemma *weighted_RP_measure_decreasing_N*:

assumes $St \rightsquigarrow_w St'$ **and** $(C, l) \in\# wN_of_wstate St$

shows $(RP_filtered_measure (\lambda Ci. True) St', RP_filtered_measure (\lambda Ci. True) St)$
 $\in RP_filtered_relation$

<proof>

lemma *weighted_RP_measure_decreasing_P*:

assumes $St \rightsquigarrow_w St'$ **and** $(C, i) \in\# wP_of_wstate St$

shows $(RP_combined_measure (weight (C, i)) St', RP_combined_measure (weight (C, i)) St)$
 $\in RP_combined_relation$

<proof>

lemma *preserve_min_or_delete_completely*:

assumes $St \rightsquigarrow_w St'$ $(C, i) \in\# wP_of_wstate St$

$\forall k. (C, k) \in\# wP_of_wstate St \longrightarrow i \leq k$

shows $(C, i) \in\# wP_of_wstate St' \vee (\forall j. (C, j) \notin\# wP_of_wstate St')$

<proof>

lemma *preserve_min_P*:

assumes

$St \rightsquigarrow_w St'$ $(C, j) \in\# wP_of_wstate St'$ **and**

$(C, i) \in\# wP_of_wstate St$ **and**

$\forall k. (C, k) \in\# wP_of_wstate St \longrightarrow i \leq k$

shows $(C, i) \in\# wP_of_wstate St'$

<proof>

lemma *preserve_min_P_Sts*:

assumes

$enat (Suc k) < llength Sts$ **and**

$(C, i) \in\# wP_of_wstate (lnth Sts k)$ **and**

$(C, j) \in\# wP_of_wstate (lnth Sts (Suc k))$ **and**

$\forall j. (C, j) \in\# wP_of_wstate (lnth Sts k) \longrightarrow i \leq j$

shows $(C, i) \in\# wP_of_wstate (lnth Sts (Suc k))$

<proof>

lemma *in_lnth_in_Supremum_ldrop*:

assumes $i < llength xs$ **and** $x \in\# (lnth xs i)$

shows $x \in Sup_llist (lmap set_mset (ldrop (enat i) xs))$

<proof>

lemma *persistent_wclause_in_P_if_persistent_clause_in_P*:

assumes $C \in Liminf_llist (lmap P_of_state (lmap state_of_wstate Sts))$

shows $\exists i. (C, i) \in Liminf_llist (lmap (set_mset \circ wP_of_wstate) Sts)$

<proof>

lemma *lfinite_not_LNil_nth_llast*:

assumes $lfinite Sts$ **and** $Sts \neq LNil$

shows $\exists i < llength Sts. lnth Sts i = llast Sts \wedge (\forall j < llength Sts. j \leq i)$

<proof>

lemma *fair_if_finite*:

assumes $fin: lfinite Sts$

shows $fair_state_seq (lmap state_of_wstate Sts)$

<proof>

lemma *N_of_state_state_of_wstate_wN_of_wstate*:

assumes $C \in N_of_state (state_of_wstate\ St)$
shows $\exists i. (C, i) \in\# wN_of_wstate\ St$
 ⟨proof⟩

lemma $in_wN_of_wstate_in_N_of_wstate: (C, i) \in\# wN_of_wstate\ St \implies C \in N_of_wstate\ St$
 ⟨proof⟩

lemma $in_wP_of_wstate_in_P_of_wstate: (C, i) \in\# wP_of_wstate\ St \implies C \in P_of_wstate\ St$
 ⟨proof⟩

lemma $in_wQ_of_wstate_in_Q_of_wstate: (C, i) \in\# wQ_of_wstate\ St \implies C \in Q_of_wstate\ St$
 ⟨proof⟩

lemma $n_of_wstate_weighted_RP_increasing: St \rightsquigarrow_w St' \implies n_of_wstate\ St \leq n_of_wstate\ St'$
 ⟨proof⟩

lemma $nth_of_wstate_monotonic:$
assumes $j < llength\ Sts$ **and** $i \leq j$
shows $n_of_wstate\ (lnth\ Sts\ i) \leq n_of_wstate\ (lnth\ Sts\ j)$
 ⟨proof⟩

lemma $infinite_chain_relation_measure:$
assumes
 $measure_decreasing: \bigwedge St\ St'. P\ St \implies R\ St\ St' \implies (m\ St', m\ St) \in mR$ **and**
 $non_infer_chain: chain\ R\ (ldrop\ (enat\ k)\ Sts)$ **and**
 $inf: llength\ Sts = \infty$ **and**
 $P: \bigwedge i. P\ (lnth\ (ldrop\ (enat\ k)\ Sts)\ i)$
shows $chain\ (\lambda x\ y. (x, y) \in mR)^{-1-1}\ (lmap\ m\ (ldrop\ (enat\ k)\ Sts))$
 ⟨proof⟩

theorem $weighted_RP_fair: fair_state_seq\ (lmap\ state_of_wstate\ Sts)$
 ⟨proof⟩

corollary $weighted_RP_saturated: src.saturated_upto\ (Liminf_llist\ (lmap\ grounding_of_wstate\ Sts))$
 ⟨proof⟩

corollary $weighted_RP_complete:$
 $\neg\ satisfiable\ (grounding_of_wstate\ (lhd\ Sts)) \implies \{\#\} \in Q_of_state\ (Liminf_wstate\ Sts)$
 ⟨proof⟩

end

end

locale $weighted_FO_resolution_prover_with_size_timestamp_factors =$
 $FO_resolution_prover\ S\ subst_atm\ id_subst\ comp_subst\ renamings_apart\ atm_of_atms\ mgu\ less_atm$
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**
 $renamings_apart :: 'a\ literal\ multiset\ list \Rightarrow 's\ list$ **and**
 $atm_of_atms :: 'a\ list \Rightarrow 'a$ **and**
 $mgu :: 'a\ set\ set \Rightarrow 's\ option$ **and**
 $less_atm :: 'a \Rightarrow 'a \Rightarrow bool$ +
fixes
 $size_atm :: 'a \Rightarrow nat$ **and**
 $size_factor :: nat$ **and**
 $timestamp_factor :: nat$
assumes
 $timestamp_factor_pos: timestamp_factor > 0$
begin

```

fun weight :: 'a wclause  $\Rightarrow$  nat where
  weight (C, i) = size_factor * size_multiset (size_literal size_atm) C + timestamp_factor * i

lemma weight_mono: i < j  $\implies$  weight (C, i) < weight (C, j)
  <proof>

declare weight.simps [simp del]

sublocale wrp: weighted_FO_resolution_prover _____ weight
  <proof>

notation wrp.weighted_RP (infix <math>\langle \rightsquigarrow_w \rangle</math> 50)

end

end

```

3 A Deterministic Ordered Resolution Prover for First-Order Clauses

The *deterministic_RP* prover introduced below is a deterministic program that works on finite lists, committing to a strategy for assigning priorities to clauses. However, it is not fully executable: It abstracts over operations on atoms and employs logical specifications instead of executable functions for auxiliary notions.

```

theory Deterministic_FO_Ordered_Resolution_Prover
imports
  Polynomial_Factorization.Missing_List
  Weighted_FO_Ordered_Resolution_Prover
  Lambda_Free_RPOs.Lambda_Free_Util
begin

```

3.1 Library

```

lemma apfst_fst_snd: apfst f x = (f (fst x), snd x)
  <proof>

```

```

lemma apfst_comp_rpair_const: apfst f  $\circ$  ( $\lambda x. (x, y)$ ) = ( $\lambda x. (x, y)$ )  $\circ$  f
  <proof>

```

```

lemma length_remove1_less[termination_simp]: x  $\in$  set xs  $\implies$  length (remove1 x xs) < length xs
  <proof>

```

```

lemma map_filter_neq_eq_filter_map:
  map f (filter ( $\lambda y. f x \neq f y$ ) xs) = filter ( $\lambda z. f x \neq z$ ) (map f xs)
  <proof>

```

```

lemma mset_map_remdups_gen:
  mset (map f (remdups_gen f xs)) = mset (remdups_gen ( $\lambda x. x$ ) (map f xs))
  <proof>

```

```

lemma mset_remdups_gen_ident: mset (remdups_gen ( $\lambda x. x$ ) xs) = mset_set (set xs)
  <proof>

```

```

lemma funpow_fixpoint: f x = x  $\implies$  (f  $\hat{\sim}$  n) x = x
  <proof>

```

```

lemma rtranclp_imp_eq_image: ( $\forall x y. R x y \longrightarrow f x = f y$ )  $\implies$  R** x y  $\implies$  f x = f y
  <proof>

```

```

lemma tranclp_imp_eq_image: ( $\forall x y. R x y \longrightarrow f x = f y$ )  $\implies$  R++ x y  $\implies$  f x = f y
  <proof>

```

3.2 Prover

```

type-synonym 'a lclause = 'a literal list
type-synonym 'a dclause = 'a lclause × nat
type-synonym 'a dstate = 'a dclause list × 'a dclause list × 'a dclause list × nat

locale deterministic_FO_resolution_prover =
  weighted_FO_resolution_prover_with_size_timestamp_factors S subst_atm id_subst comp_subst
  renamings_apart atm_of_atms mgu less_atm size_atm timestamp_factor size_factor
for
  S :: ('a :: wellorder) clause ⇒ 'a clause and
  subst_atm :: 'a ⇒ 's ⇒ 'a and
  id_subst :: 's and
  comp_subst :: 's ⇒ 's ⇒ 's and
  renamings_apart :: 'a literal multiset list ⇒ 's list and
  atm_of_atms :: 'a list ⇒ 'a and
  mgu :: 'a set set ⇒ 's option and
  less_atm :: 'a ⇒ 'a ⇒ bool and
  size_atm :: 'a ⇒ nat and
  timestamp_factor :: nat and
  size_factor :: nat +
assumes
  S_empty: S C = {#}
begin

lemma less_atm_irrefl: ¬ less_atm A A
  ⟨proof⟩

fun wstate_of_dstate :: 'a dstate ⇒ 'a wstate where
  wstate_of_dstate (N, P, Q, n) =
    (mset (map (apfst mset) N), mset (map (apfst mset) P), mset (map (apfst mset) Q), n)

fun state_of_dstate :: 'a dstate ⇒ 'a state where
  state_of_dstate (N, P, Q, _) =
    (set (map (mset ∘ fst) N), set (map (mset ∘ fst) P), set (map (mset ∘ fst) Q))

abbreviation cls_of_dstate :: 'a dstate ⇒ 'a clause set where
  cls_of_dstate St ≡ cls_of_state (state_of_dstate St)

fun is_final_dstate :: 'a dstate ⇒ bool where
  is_final_dstate (N, P, Q, n) ⟷ N = [] ∧ P = []

declare is_final_dstate.simps [simp del]

abbreviation rtrancl_weighted_RP (infix ⟨↗w*⟩ 50) where
  (↗w*) ≡ (↗w)**

abbreviation trancl_weighted_RP (infix ⟨↗w+⟩ 50) where
  (↗w+) ≡ (↗w)++

definition is_tautology :: 'a lclause ⇒ bool where
  is_tautology C ⟷ (∃ A ∈ set (map atm_of C). Pos A ∈ set C ∧ Neg A ∈ set C)

definition subsume :: 'a lclause list ⇒ 'a lclause ⇒ bool where
  subsume Ds C ⟷ (∃ D ∈ set Ds. subsumes (mset D) (mset C))

definition strictly_subsume :: 'a lclause list ⇒ 'a lclause ⇒ bool where
  strictly_subsume Ds C ⟷ (∃ D ∈ set Ds. strictly_subsumes (mset D) (mset C))

definition is_reducible_on :: 'a literal ⇒ 'a lclause ⇒ 'a literal ⇒ 'a lclause ⇒ bool where
  is_reducible_on M D L C ⟷ subsumes (mset D + {#- M#}) (mset C + {#L#})

definition is_reducible_lit :: 'a lclause list ⇒ 'a lclause ⇒ 'a literal ⇒ bool where
  is_reducible_lit Ds C L ⟷

```

$(\exists D \in \text{set } Ds. \exists L' \in \text{set } D. \exists \sigma. - L = L' \cdot l \sigma \wedge \text{mset } (\text{remove1 } L' D) \cdot \sigma \subseteq\# \text{mset } C)$

primrec *reduce* :: 'a lclause list \Rightarrow 'a lclause \Rightarrow 'a lclause \Rightarrow 'a lclause **where**
reduce _ _ [] = []
| *reduce* Ds C (L # C') =
 (if *is_irreducible_lit* Ds (C @ C') L then *reduce* Ds C C' else L # *reduce* Ds (L # C) C')

abbreviation *is_irreducible* :: 'a lclause list \Rightarrow 'a lclause \Rightarrow bool **where**
is_irreducible Ds C \equiv *reduce* Ds [] C = C

abbreviation *is_reducible* :: 'a lclause list \Rightarrow 'a lclause \Rightarrow bool **where**
is_reducible Ds C \equiv *reduce* Ds [] C \neq C

definition *reduce_all* :: 'a lclause \Rightarrow 'a dclause list \Rightarrow 'a dclause list **where**
reduce_all D = *map* (*apfst* (*reduce* [D] []))

fun *reduce_all2* :: 'a lclause \Rightarrow 'a dclause list \Rightarrow 'a dclause list \times 'a dclause list **where**
reduce_all2 _ [] = ([], [])
| *reduce_all2* D (Ci # Cs) =
 (let
 (C, i) = Ci;
 C' = *reduce* [D] [] C
 in
 (if C' = C then *apsnd* else *apfst*) (Cons (C', i)) (*reduce_all2* D Cs))

fun *remove_all* :: 'b list \Rightarrow 'b list \Rightarrow 'b list **where**
remove_all xs [] = xs
| *remove_all* xs (y # ys) = (if $y \in \text{set } xs$ then *remove_all* (*remove1* y xs) ys else *remove_all* xs ys)

lemma *remove_all_mset_minus*: $\text{mset } ys \subseteq\# \text{mset } xs \implies \text{mset } (\text{remove_all } xs \ ys) = \text{mset } xs - \text{mset } ys$
 <proof>

definition *resolvent* :: 'a lclause \Rightarrow 'a \Rightarrow 'a lclause \Rightarrow 'a lclause \Rightarrow 'a lclause **where**
resolvent D A CA Ls =
map ($\lambda M. M \cdot l$ (*the* (*mgu* {*insert* A (*atms_of* (*mset* Ls))})) (*remove_all* CA Ls @ D))

definition *resolvable* :: 'a \Rightarrow 'a lclause \Rightarrow 'a lclause \Rightarrow 'a lclause \Rightarrow bool **where**
resolvable A D CA Ls \longleftrightarrow
 (let $\sigma =$ (*mgu* {*insert* A (*atms_of* (*mset* Ls))}) in
 $\sigma \neq \text{None}$
 \wedge Ls \neq []
 \wedge *maximal_wrt* (A \cdot a *the* σ) ((*add_mset* (*Neg* A) (*mset* D)) \cdot *the* σ)
 \wedge *strictly_maximal_wrt* (A \cdot a *the* σ) ((*mset* CA - *mset* Ls) \cdot *the* σ)
 \wedge ($\forall L \in \text{set } Ls. \text{is_pos } L$)

definition *resolve_on* :: 'a \Rightarrow 'a lclause \Rightarrow 'a lclause \Rightarrow 'a lclause list **where**
resolve_on A D CA = *map* (*resolvent* D A CA) (*filter* (*resolvable* A D CA) (*subseqs* CA))

definition *resolve* :: 'a lclause \Rightarrow 'a lclause \Rightarrow 'a lclause list **where**
resolve C D =
concat (*map* ($\lambda L.$
 (*case* L of
 Pos A \Rightarrow []
 | Neg A \Rightarrow
 if *maximal_wrt* A (*mset* D) then
resolve_on A (*remove1* L D) C
 else
 [])) D)

definition *resolve_rename* :: 'a lclause \Rightarrow 'a lclause \Rightarrow 'a lclause list **where**
resolve_rename C D =
 (let $\sigma s =$ *renamings_apart* [*mset* D, *mset* C] in
resolve (*map* ($\lambda L. L \cdot l$ last σs) C) (*map* ($\lambda L. L \cdot l$ hd σs) D))

definition *resolve_rename_either_way* :: 'a lclause \Rightarrow 'a lclause \Rightarrow 'a lclause list **where**
resolve_rename_either_way C D = *resolve_rename* C D @ *resolve_rename* D C

fun *select_min_weight_clause* :: 'a dclause \Rightarrow 'a dclause list \Rightarrow 'a dclause **where**
select_min_weight_clause Ci [] = Ci
| *select_min_weight_clause* Ci (Dj # Djs) =
select_min_weight_clause
(if *weight* (*apfst* *mset* Dj) < *weight* (*apfst* *mset* Ci) then Dj else Ci) Djs

lemma *select_min_weight_clause_in*: *select_min_weight_clause* P0 P \in set (P0 # P)
⟨*proof*⟩

function *remdups_clss* :: 'a dclause list \Rightarrow 'a dclause list **where**
remdups_clss [] = []
| *remdups_clss* (Ci # Cis) =
(let
Ci' = *select_min_weight_clause* Ci Cis
in
Ci' # *remdups_clss* (filter ($\lambda(D, _).$ *mset* D \neq *mset* (*fst* Ci')) (Ci # Cis)))
⟨*proof*⟩

termination
⟨*proof*⟩

declare *remdups_clss.simps(2)* [*simp del*]

fun *deterministic_RP_step* :: 'a dstate \Rightarrow 'a dstate **where**
deterministic_RP_step (N, P, Q, n) =
(if \exists Ci \in set (P @ Q). *fst* Ci = [] then
([], [], *remdups_clss* P @ Q, n + length (*remdups_clss* P))
else
(case N of
[] \Rightarrow
(case P of
[] \Rightarrow (N, P, Q, n)
| P0 # P' \Rightarrow
let
(C, i) = *select_min_weight_clause* P0 P';
N = map ($\lambda D. (D, n)$) (*remdups_gen* *mset* (*resolve_rename* C C
@ *concat* (map (*resolve_rename_either_way* C \circ *fst*) Q)));
P = filter ($\lambda(D, j).$ *mset* D \neq *mset* C) P;
Q = (C, i) # Q;
n = *Suc* n
in
(N, P, Q, n))
| (C, i) # N \Rightarrow
let
C = *reduce* (map *fst* (P @ Q)) [] C
in
if C = [] then
([], [], [([]), i], *Suc* n)
else if *is_tautology* C \vee *subsume* (map *fst* (P @ Q)) C then
(N, P, Q, n)
else
let
P = *reduce_all* C P;
(*back_to_P*, Q) = *reduce_all2* C Q;
P = *back_to_P* @ P;
Q = filter (*Not* \circ *strictly_subsume* [C] \circ *fst*) Q;
P = filter (*Not* \circ *strictly_subsume* [C] \circ *fst*) P;
P = (C, i) # P
in
(N, P, Q, n)))

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

partial-function (*option*) *deterministic_RP* :: 'a *dstate* \Rightarrow 'a *lclause list option* **where**

deterministic_RP St =
 (if *is_final_dstate St* then
 let (*_*, *_*, *Q*, *_*) = *St* in *Some (map fst Q)*
 else
 deterministic_RP (deterministic_RP_step St))

lemma *is_final_dstate_imp_not_weighted_RP*: *is_final_dstate St* \Longrightarrow \neg *wstate_of_dstate St* $\rightsquigarrow_w St'$
 <proof>

lemma *is_final_dstate_funpow_imp_deterministic_RP_neq_None*:
is_final_dstate ((deterministic_RP_step \sim^k) St) \Longrightarrow *deterministic_RP St* \neq *None*
 <proof>

lemma *is_reducible_lit_mono_cls*:
mset C $\subseteq\#$ *mset C'* \Longrightarrow *is_reducible_lit Ds C L* \Longrightarrow *is_reducible_lit Ds C' L*
 <proof>

lemma *is_reducible_lit_mset_iff*:
mset C = *mset C'* \Longrightarrow *is_reducible_lit Ds C' L* \longleftrightarrow *is_reducible_lit Ds C L*
 <proof>

lemma *is_reducible_lit_remove1_Cons_iff*:
assumes *L* \in *set C'*
shows *is_reducible_lit Ds (C @ remove1 L (M # C')) L* \longleftrightarrow
is_reducible_lit Ds (M # C @ remove1 L C') L
 <proof>

lemma *reduce_mset_eq*: *mset C* = *mset C'* \Longrightarrow *reduce Ds C E* = *reduce Ds C' E*
 <proof>

lemma *reduce_rotate[simp]*: *reduce Ds (C @ [L]) E* = *reduce Ds (L # C) E*
 <proof>

lemma *mset_reduce_subset*: *mset (reduce Ds C E)* $\subseteq\#$ *mset E*
 <proof>

lemma *reduce_idem*: *reduce Ds C (reduce Ds C E)* = *reduce Ds C E*
 <proof>

lemma *is_reducible_lit_imp_is_reducible*:
L \in *set C'* \Longrightarrow *is_reducible_lit Ds (C @ remove1 L C') L* \Longrightarrow *reduce Ds C C' \neq C'*
 <proof>

lemma *is_reducible_imp_is_reducible_lit*:
reduce Ds C C' \neq C' \Longrightarrow $\exists L \in$ *set C'. is_reducible_lit Ds (C @ remove1 L C') L*
 <proof>

lemma *is_irreducible_iff_nexists_is_reducible_lit*:
reduce Ds C C' = C' \longleftrightarrow $\neg (\exists L \in$ *set C'. is_reducible_lit Ds (C @ remove1 L C') L)
 <proof>*

lemma *is_irreducible_mset_iff*: *mset E* = *mset E'* \Longrightarrow *reduce Ds C E* = *E* \longleftrightarrow *reduce Ds C E' = E'*
 <proof>

lemma *select_min_weight_clause_min_weight*:
assumes *Ci* = *select_min_weight_clause P0 P*
shows *weight (apfst mset Ci)* = *Min ((weight \circ apfst mset) ' set (P0 # P))*
 <proof>

lemma *remdups_class_Nil_iff*: $\text{remdups_class } Cs = [] \longleftrightarrow Cs = []$
 ⟨proof⟩

lemma *empty_N_if_Nil_in_P_or_Q*:
assumes *nil_in*: $[] \in \text{fst } \text{'set } (P @ Q)$
shows $\text{wstate_of_dstate } (N, P, Q, n) \rightsquigarrow_w^* \text{wstate_of_dstate } ([], P, Q, n)$
 ⟨proof⟩

lemma *remove_strictly_subsumed_clauses_in_P*:
assumes
c_in: $C \in \text{fst } \text{'set } N$ **and**
p_nsubs: $\forall D \in \text{fst } \text{'set } P. \neg \text{strictly_subsume } [C] D$
shows $\text{wstate_of_dstate } (N, P @ P', Q, n)$
 $\rightsquigarrow_w^* \text{wstate_of_dstate } (N, P @ \text{filter } (\text{Not } \circ \text{strictly_subsume } [C] \circ \text{fst}) P', Q, n)$
 ⟨proof⟩

lemma *remove_strictly_subsumed_clauses_in_Q*:
assumes *c_in*: $C \in \text{fst } \text{'set } N$
shows $\text{wstate_of_dstate } (N, P, Q @ Q', n)$
 $\rightsquigarrow_w^* \text{wstate_of_dstate } (N, P, Q @ \text{filter } (\text{Not } \circ \text{strictly_subsume } [C] \circ \text{fst}) Q', n)$
 ⟨proof⟩

lemma *reduce_clause_in_P*:
assumes
c_in: $C \in \text{fst } \text{'set } N$ **and**
p_irred: $\forall (E, k) \in \text{set } (P @ P'). k > j \longrightarrow \text{is_irreducible } [C] E$
shows $\text{wstate_of_dstate } (N, P @ (D @ D', j) \# P', Q, n)$
 $\rightsquigarrow_w^* \text{wstate_of_dstate } (N, P @ (D @ \text{reduce } [C] D D', j) \# P', Q, n)$
 ⟨proof⟩

lemma *reduce_clause_in_Q*:
assumes
c_in: $C \in \text{fst } \text{'set } N$ **and**
p_irred: $\forall (E, k) \in \text{set } P. k > j \longrightarrow \text{is_irreducible } [C] E$ **and**
d'_red: $\text{reduce } [C] D D' \neq D'$
shows $\text{wstate_of_dstate } (N, P, Q @ (D @ D', j) \# Q', n)$
 $\rightsquigarrow_w^* \text{wstate_of_dstate } (N, (D @ \text{reduce } [C] D D', j) \# P, Q @ Q', n)$
 ⟨proof⟩

lemma *reduce_clauses_in_P*:
assumes
c_in: $C \in \text{fst } \text{'set } N$ **and**
p_irred: $\forall (E, k) \in \text{set } P. \text{is_irreducible } [C] E$
shows $\text{wstate_of_dstate } (N, P @ P', Q, n) \rightsquigarrow_w^* \text{wstate_of_dstate } (N, P @ \text{reduce_all } C P', Q, n)$
 ⟨proof⟩

lemma *reduce_clauses_in_Q*:
assumes
c_in: $C \in \text{fst } \text{'set } N$ **and**
p_irred: $\forall (E, k) \in \text{set } P. \text{is_irreducible } [C] E$
shows $\text{wstate_of_dstate } (N, P, Q @ Q', n)$
 $\rightsquigarrow_w^* \text{wstate_of_dstate } (N, \text{fst } (\text{reduce_all2 } C Q') @ P, Q @ \text{snd } (\text{reduce_all2 } C Q'), n)$
 ⟨proof⟩

lemma *eligible_iff*:
 $\text{eligible } S \sigma \text{ As } DA \longleftrightarrow \text{As} = [] \vee \text{length } \text{As} = 1 \wedge \text{maximal_wrt } (\text{hd } \text{As} \cdot a \sigma) (DA \cdot \sigma)$
 ⟨proof⟩

lemma *ord_resolve_one_side_prem*:
 $\text{ord_resolve } S \text{ CAs } DA \text{ AAs } \text{As } \sigma E \implies \text{length } \text{CAs} = 1 \wedge \text{length } \text{AAs} = 1 \wedge \text{length } \text{As} = 1$
 ⟨proof⟩

lemma *ord_resolve_rename_one_side_prem*:

$ord_resolve_rename\ S\ CAs\ DA\ AAs\ As\ \sigma\ E \implies length\ CAs = 1 \wedge length\ AAs = 1 \wedge length\ As = 1$
 ⟨proof⟩

abbreviation $Bin_ord_resolve :: 'a\ clause \Rightarrow 'a\ clause \Rightarrow 'a\ clause\ set\ \mathbf{where}$

$Bin_ord_resolve\ C\ D \equiv \{E. \exists AA\ A\ \sigma. ord_resolve\ S\ [C]\ D\ [AA]\ [A]\ \sigma\ E\}$

abbreviation $Bin_ord_resolve_rename :: 'a\ clause \Rightarrow 'a\ clause \Rightarrow 'a\ clause\ set\ \mathbf{where}$

$Bin_ord_resolve_rename\ C\ D \equiv \{E. \exists AA\ A\ \sigma. ord_resolve_rename\ S\ [C]\ D\ [AA]\ [A]\ \sigma\ E\}$

lemma $resolve_on_eq_UNION_Bin_ord_resolve:$

$mset\ 'set\ (resolve_on\ A\ D\ CA) =$
 $\{E. \exists AA\ \sigma. ord_resolve\ S\ [mset\ CA]\ (\{\#Neg\ A\#\} + mset\ D)\ [AA]\ [A]\ \sigma\ E\}$

⟨proof⟩

lemma $set_resolve_eq_UNION_set_resolve_on:$

$set\ (resolve\ C\ D) =$
 $(\bigcup L \in set\ D.$
 (case L of
 $Pos\ _ \Rightarrow \{\}$
 $| Neg\ A \Rightarrow if\ maximal_wrt\ A\ (mset\ D)\ then\ set\ (resolve_on\ A\ (remove1\ L\ D)\ C)\ else\ \{\})$)

⟨proof⟩

lemma $resolve_eq_Bin_ord_resolve: mset\ 'set\ (resolve\ C\ D) = Bin_ord_resolve\ (mset\ C)\ (mset\ D)$

⟨proof⟩

lemma $poss_in_map_clauseD:$

$poss\ AA\ \subseteq\# map_clause\ f\ C \implies \exists AA0. poss\ AA0\ \subseteq\# C \wedge AA = \{\#f\ A. A \in\# AA0\#\}$

⟨proof⟩

lemma $poss_subset_filterD:$

$poss\ AA\ \subseteq\# \{\#L \cdot l\ \varrho. L \in\# mset\ C\#\} \implies \exists AA0. poss\ AA0\ \subseteq\# mset\ C \wedge AA = AA0 \cdot am\ \varrho$

⟨proof⟩

lemma $neg_in_map_literalD: Neg\ A \in map_literal\ f\ 'D \implies \exists A0. Neg\ A0 \in D \wedge A = f\ A0$

⟨proof⟩

lemma $neg_in_filterD: Neg\ A \in\# \{\#L \cdot l\ \varrho'. L \in\# mset\ D\#\} \implies \exists A0. Neg\ A0 \in\# mset\ D \wedge A = A0 \cdot a\ \varrho'$

⟨proof⟩

lemma $resolve_rename_eq_Bin_ord_resolve_rename:$

$mset\ 'set\ (resolve_rename\ C\ D) = Bin_ord_resolve_rename\ (mset\ C)\ (mset\ D)$

⟨proof⟩

lemma $bin_ord_FO_Gamma_def:$

$ord_FO_Gamma\ S = \{Infer\ \{\#CA\#\}\ DA\ E \mid CA\ DA\ AA\ A\ \sigma\ E. ord_resolve_rename\ S\ [CA]\ DA\ [AA]\ [A]\ \sigma\ E\}$

⟨proof⟩

lemma $ord_FO_Gamma_side_prem: \gamma \in ord_FO_Gamma\ S \implies side_prems_of\ \gamma = \{\#THE\ D. D \in\# side_prems_of\ \gamma\#\}$

⟨proof⟩

lemma $ord_FO_Gamma_infer_from_Collect_eq:$

$\{\gamma \in ord_FO_Gamma\ S. infer_from\ (DD \cup \{C\})\ \gamma \wedge C \in\# prems_of\ \gamma\} =$
 $\{\gamma \in ord_FO_Gamma\ S. \exists D \in DD \cup \{C\}. prems_of\ \gamma = \{\#C, D\#\}\}$

⟨proof⟩

lemma $inferences_between_eq_UNION: inference_system.inferences_between\ (ord_FO_Gamma\ S)\ Q\ C =$

$inference_system.inferences_between\ (ord_FO_Gamma\ S)\ \{C\}\ C$
 $\cup (\bigcup D \in Q. inference_system.inferences_between\ (ord_FO_Gamma\ S)\ \{D\}\ C)$

⟨proof⟩

lemma $concls_of_inferences_between_singleton_eq_Bin_ord_resolve_rename:$

$concls_of\ (inference_system.inferences_between\ (ord_FO_Gamma\ S)\ \{D\}\ C) =$
 $Bin_ord_resolve_rename\ C\ C \cup Bin_ord_resolve_rename\ C\ D \cup Bin_ord_resolve_rename\ D\ C$

<proof>

lemma *concls_of_inferences_between_eq_Bin_ord_resolve_rename:*

concls_of (inference_system.inferences_between (ord_FO_Γ S) Q C) =

Bin_ord_resolve_rename C C ∪ (∪ D ∈ Q. Bin_ord_resolve_rename C D ∪ Bin_ord_resolve_rename D C)

<proof>

lemma *resolve_rename_either_way_eq_concls_of_inferences_between:*

mset ' set (resolve_rename C C) ∪ (∪ D ∈ Q. mset ' set (resolve_rename_either_way C D)) =

concls_of (inference_system.inferences_between (ord_FO_Γ S) (mset ' Q) (mset C))

<proof>

lemma *compute_inferences:*

assumes

ci_in: (C, i) ∈ set P and

ci_min: ∀ (D, j) ∈ # mset (map (apfst mset) P). weight (mset C, i) ≤ weight (D, j)

shows

wstate_of_dstate ([], P, Q, n) ↘_w

wstate_of_dstate (map (λD. (D, n)) (remdups_gen mset (resolve_rename C C @

concat (map (resolve_rename_either_way C ∘ fst) Q))),

filter (λ(D, j). mset D ≠ mset C) P, (C, i) # Q, Suc n)

(is _ ↘_w wstate_of_dstate (?N, _))

<proof>

lemma *nonfinal_deterministic_RP_step:*

assumes

nonfinal: ¬ is_final_dstate St and

step: St' = deterministic_RP_step St

shows *wstate_of_dstate St ↘_w⁺ wstate_of_dstate St'*

<proof>

lemma *final_deterministic_RP_step: is_final_dstate St ⇒ deterministic_RP_step St = St*

<proof>

lemma *deterministic_RP_SomeD:*

assumes *deterministic_RP (N, P, Q, n) = Some R*

shows *∃ N' P' Q' n'. (∃ k. (deterministic_RP_step $\overset{\sim}{\sim}$ k) (N, P, Q, n) = (N', P', Q', n'))*

∧ is_final_dstate (N', P', Q', n') ∧ R = map fst Q'

<proof>

context

fixes

N0 :: 'a dclause list and

n0 :: nat and

R :: 'a lclause list

begin

abbreviation *St0 :: 'a dstate where*

St0 ≡ (N0, [], [], n0)

abbreviation *grounded_N0 where*

grounded_N0 ≡ grounding_of_cls (set (map (mset ∘ fst) N0))

abbreviation *grounded_R :: 'a clause set where*

grounded_R ≡ grounding_of_cls (set (map mset R))

primcorec *derivation_from :: 'a dstate ⇒ 'a dstate llist where*

derivation_from St =

LCons St (if is_final_dstate St then LNil else derivation_from (deterministic_RP_step St))

abbreviation *Sts :: 'a dstate llist where*

Sts ≡ derivation_from St0

abbreviation $wSts :: 'a\ wstate\ llist\ \mathbf{where}$

$wSts \equiv \mathit{lmap}\ wstate_of_dstate\ Sts$

lemma $full_deriv_wSts_trancl_weighted_RP: full_chain\ (\rightsquigarrow_w^+)\ wSts$

$\langle proof \rangle$

lemmas $deriv_wSts_trancl_weighted_RP = full_chain_imp_chain[OF\ full_deriv_wSts_trancl_weighted_RP]$

definition $sswSts :: 'a\ wstate\ llist\ \mathbf{where}$

$sswSts = (SOME\ wSts')$

$full_chain\ (\rightsquigarrow_w)\ wSts' \wedge emb\ wSts\ wSts' \wedge lhd\ wSts' = lhd\ wSts \wedge llast\ wSts' = llast\ wSts$

lemma $sswSts:$

$full_chain\ (\rightsquigarrow_w)\ sswSts \wedge emb\ wSts\ sswSts \wedge lhd\ sswSts = lhd\ wSts \wedge llast\ sswSts = llast\ wSts$

$\langle proof \rangle$

lemmas $full_deriv_sswSts_weighted_RP = sswSts[THEN\ conjunct1]$

lemmas $emb_sswSts = sswSts[THEN\ conjunct2,\ THEN\ conjunct1]$

lemmas $lfinite_sswSts_iff = emb_lfinite[OF\ emb_sswSts]$

lemmas $lhd_sswSts = sswSts[THEN\ conjunct2,\ THEN\ conjunct2,\ THEN\ conjunct1]$

lemmas $llast_sswSts = sswSts[THEN\ conjunct2,\ THEN\ conjunct2,\ THEN\ conjunct2]$

lemmas $deriv_sswSts_weighted_RP = full_chain_imp_chain[OF\ full_deriv_sswSts_weighted_RP]$

lemma $not_lnull_sswSts: \neg\ lnull\ sswSts$

$\langle proof \rangle$

lemma $empty_ssgP0: wrp.P_of_wstate\ (lhd\ sswSts) = \{\}$

$\langle proof \rangle$

lemma $empty_ssgQ0: wrp.Q_of_wstate\ (lhd\ sswSts) = \{\}$

$\langle proof \rangle$

lemmas $sswSts_thms = full_deriv_sswSts_weighted_RP\ empty_ssgP0\ empty_ssgQ0$

abbreviation $S_ssgQ :: 'a\ clause \Rightarrow 'a\ clause\ \mathbf{where}$

$S_ssgQ \equiv wrp.S_gQ\ sswSts$

abbreviation $ord_Gamma :: 'a\ inference\ set\ \mathbf{where}$

$ord_Gamma \equiv ground_resolution_with_selection.ord_Gamma\ S_ssgQ$

abbreviation $Rf :: 'a\ clause\ set \Rightarrow 'a\ clause\ set\ \mathbf{where}$

$Rf \equiv standard_redundancy_criterion.Rf$

abbreviation $Ri :: 'a\ clause\ set \Rightarrow 'a\ inference\ set\ \mathbf{where}$

$Ri \equiv standard_redundancy_criterion.Ri\ ord_Gamma$

abbreviation $saturated_upto :: 'a\ clause\ set \Rightarrow bool\ \mathbf{where}$

$saturated_upto \equiv redundancy_criterion.saturated_upto\ ord_Gamma\ Rf\ Ri$

context

assumes $drp_some: deterministic_RP\ St0 = Some\ R$

begin

lemma $lfinite_Sts: lfinite\ Sts$

$\langle proof \rangle$

lemma $lfinite_wSts: lfinite\ wSts$

$\langle proof \rangle$

lemmas $lfinite_sswSts = lfinite_sswSts_iff[THEN\ iffD2,\ OF\ lfinite_wSts]$

theorem

deterministic_RP_saturated: saturated_upto grounded_R (is ?saturated) and
deterministic_RP_model: I ⊨s grounded_N0 ↔ I ⊨s grounded_R (is ?model)
 ⟨proof⟩

corollary *deterministic_RP_refutation:*

\neg *satisfiable grounded_N0 ↔ {#} ∈ grounded_R (is ?lhs ↔ ?rhs)*
 ⟨proof⟩

end

context

assumes *drp_none: deterministic_RP St0 = None*

begin

theorem *deterministic_RP_complete: satisfiable grounded_N0*

⟨proof⟩

end

end

end

end

4 Integration of IsaFoR Terms and the Knuth–Bendix Order

This theory implements the abstract interface for atoms and substitutions using the IsaFoR library.

theory *IsaFoR_Term*

imports

Deriving.Derive
Ordered_Resolution_Prover.Abstract_Substitution
First_Order_Terms.Unification
First_Order_Terms.Subsumption
HOL-Cardinals.Wellorder_Extension
Open_Induction.Restricted_Predicates
Knuth_Bendix_Order.KBO

begin

hide-const (open) *mgv*

abbreviation *subst_apply_literal ::*

$(f, 'v)$ *term literal* $\Rightarrow (f, 'v, 'w)$ *gsubst* $\Rightarrow (f, 'w)$ *term literal* (**infixl** $\langle \cdot \text{lit} \rangle$ 60) **where**
 $L \cdot \text{lit } \sigma \equiv \text{map_literal } (\lambda A. A \cdot \sigma) L$

definition *subst_apply_clause ::*

$(f, 'v)$ *term clause* $\Rightarrow (f, 'v, 'w)$ *gsubst* $\Rightarrow (f, 'w)$ *term clause* (**infixl** $\langle \cdot \text{cls} \rangle$ 60) **where**
 $C \cdot \text{cls } \sigma = \text{image_mset } (\lambda L. L \cdot \text{lit } \sigma) C$

abbreviation *vars_lit :: (f, 'v) term literal* \Rightarrow *'v set* **where**

$\text{vars_lit } L \equiv \text{vars_term } (\text{atm_of } L)$

definition *vars_clause :: (f, 'v) term clause* \Rightarrow *'v set* **where**

$\text{vars_clause } C = \text{Union } (\text{set_mset } (\text{image_mset } \text{vars_lit } C))$

definition *vars_clause_list :: (f, 'v) term clause list* \Rightarrow *'v set* **where**

$\text{vars_clause_list } Cs = \text{Union } (\text{vars_clause } ' \text{set } Cs)$

definition *vars_partitioned :: (f, 'v) term clause list* \Rightarrow *bool* **where**

$\text{vars_partitioned } Cs \longleftrightarrow$

$(\forall i < \text{length } Cs. \forall j < \text{length } Cs. i \neq j \longrightarrow (\text{vars_clause } (Cs ! i) \cap \text{vars_clause } (Cs ! j)) = \{\})$

lemma *vars_clause_mono*: $S \subseteq\# C \implies \text{vars_clause } S \subseteq \text{vars_clause } C$
(*proof*)

interpretation *substitution_ops* (\cdot) *Var* (\circ_s) (*proof*)

lemma *is_ground_atm_is_ground_on_var*:
 assumes *is_ground_atm* ($A \cdot \sigma$) **and** $v \in \text{vars_term } A$
 shows *is_ground_atm* (σv)
(*proof*)

lemma *is_ground_lit_is_ground_on_var*:
 assumes *ground_lit*: *is_ground_lit* (*subst_lit* $L \sigma$) **and** v_in_L : $v \in \text{vars_lit } L$
 shows *is_ground_atm* (σv)
(*proof*)

lemma *is_ground_cls_is_ground_on_var*:
 assumes
 ground_clause: *is_ground_cls* (*subst_cls* $C \sigma$) **and**
 v_in_C : $v \in \text{vars_clause } C$
 shows *is_ground_atm* (σv)
(*proof*)

lemma *is_ground_cls_list_is_ground_on_var*:
 assumes *ground_list*: *is_ground_cls_list* (*subst_cls_list* $Cs \sigma$)
 and v_in_Cs : $v \in \text{vars_clause_list } Cs$
 shows *is_ground_atm* (σv)
(*proof*)

lemma *same_on_vars_lit*:
 assumes $\forall v \in \text{vars_lit } L. \sigma v = \tau v$
 shows *subst_lit* $L \sigma = \text{subst_lit } L \tau$
(*proof*)

lemma *in_list_of_mset_in_S*:
 assumes $i < \text{length } (\text{list_of_mset } S)$
 shows *list_of_mset* $S ! i \in\# S$
(*proof*)

lemma *same_on_vars_clause*:
 assumes $\forall v \in \text{vars_clause } S. \sigma v = \tau v$
 shows *subst_cls* $S \sigma = \text{subst_cls } S \tau$
(*proof*)

interpretation *substitution* (\cdot) *Var* :: $_ \Rightarrow ('f, \text{nat}) \text{ term } (\circ_s)$
(*proof*)

lemma *vars_partitioned_var_disjoint*:
 assumes *vars_partitioned* Cs
 shows *var_disjoint* Cs
(*proof*)

lemma *vars_in_instance_in_range_term*:
 $\text{vars_term } (\text{subst_atm_abbrev } A \sigma) \subseteq \text{Union } (\text{image } \text{vars_term } (\text{range } \sigma))$
(*proof*)

lemma *vars_in_instance_in_range_lit*: $\text{vars_lit } (\text{subst_lit } L \sigma) \subseteq \text{Union } (\text{image } \text{vars_term } (\text{range } \sigma))$
(*proof*)

lemma *vars_in_instance_in_range_cls*:
 $\text{vars_clause } (\text{subst_cls } C \sigma) \subseteq \text{Union } (\text{image } \text{vars_term } (\text{range } \sigma))$
(*proof*)

primrec *renamings_apart* :: $('f, \text{nat}) \text{ term clause list} \Rightarrow (('f, \text{nat}) \text{ subst}) \text{ list where}$

```

renamings_apart [] = []
| renamings_apart (C # Cs) =
  (let  $\sigma s = \text{renamings\_apart } Cs \text{ in}$ 
   ( $\lambda v. \text{Var } (v + \text{Max } (\text{vars\_clause\_list } (\text{subst\_cls\_lists } Cs \sigma s) \cup \{0\}) + 1)) \# \sigma s)$ 
```

definition $\text{var_map_of_subst} :: ('f, \text{nat}) \text{subst} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**
 $\text{var_map_of_subst } \sigma v = \text{the_Var } (\sigma v)$

lemma $\text{len_renamings_apart}$: $\text{length } (\text{renamings_apart } Cs) = \text{length } Cs$
 $\langle \text{proof} \rangle$

lemma $\text{renamings_apart_is_Var}$: $\forall \sigma \in \text{set } (\text{renamings_apart } Cs). \forall x. \text{is_Var } (\sigma x)$
 $\langle \text{proof} \rangle$

lemma $\text{renamings_apart_inj}$: $\forall \sigma \in \text{set } (\text{renamings_apart } Cs). \text{inj } \sigma$
 $\langle \text{proof} \rangle$

lemma $\text{finite_vars_clause}[simp]$: $\text{finite } (\text{vars_clause } x)$
 $\langle \text{proof} \rangle$

lemma $\text{finite_vars_clause_list}[simp]$: $\text{finite } (\text{vars_clause_list } Cs)$
 $\langle \text{proof} \rangle$

lemma Suc_Max_notin_set : $\text{finite } X \Longrightarrow \text{Suc } (v + \text{Max } (\text{insert } 0 X)) \notin X$
 $\langle \text{proof} \rangle$

lemma $\text{vars_partitioned_Nil}[simp]$: $\text{vars_partitioned } []$
 $\langle \text{proof} \rangle$

lemma $\text{subst_cls_lists_Nil}[simp]$: $\text{subst_cls_lists } Cs [] = []$
 $\langle \text{proof} \rangle$

lemma $\text{vars_clause_hd_partitioned_from_tl}$:
assumes $Cs \neq []$
shows $\text{vars_clause } (\text{hd } (\text{subst_cls_lists } Cs (\text{renamings_apart } Cs)))$
 $\cap \text{vars_clause_list } (\text{tl } (\text{subst_cls_lists } Cs (\text{renamings_apart } Cs))) = \{\}$
 $\langle \text{proof} \rangle$

lemma $\text{vars_partitioned_renamings_apart}$: $\text{vars_partitioned } (\text{subst_cls_lists } Cs (\text{renamings_apart } Cs))$
 $\langle \text{proof} \rangle$

interpretation $\text{substitution_renamings } (\cdot) \text{Var} :: _ \Rightarrow ('f, \text{nat}) \text{term } (\circ_s) \text{renamings_apart}$ *Fun undefined*
 $\langle \text{proof} \rangle$

fun $\text{pairs} :: 'a \text{ list} \Rightarrow ('a \times 'a) \text{ list}$ **where**
 $\text{pairs } (x \# y \# xs) = (x, y) \# \text{pairs } (y \# xs) \mid$
 $\text{pairs } _ = []$

derive compare term
derive compare literal

lemma $\text{class_linorder_compare}$: $\text{class.linorder } (\text{le_of_comp } \text{compare}) (\text{lt_of_comp } \text{compare})$
 $\langle \text{proof} \rangle$

context begin

interpretation compare_linorder : linorder
 $\text{le_of_comp } \text{compare}$
 $\text{lt_of_comp } \text{compare}$
 $\langle \text{proof} \rangle$

definition Pairs **where**

$\text{Pairs } AAA = \text{concat } (\text{compare_linorder.sorted_list_of_set}$
 $((\text{pairs} \circ \text{compare_linorder.sorted_list_of_set}) \text{ ` } AAA))$

lemma *unifies_all_pairs_iff*:

$(\forall p \in \text{set } (\text{pairs } xs). \text{fst } p \cdot \sigma = \text{snd } p \cdot \sigma) \longleftrightarrow (\forall a \in \text{set } xs. \forall b \in \text{set } xs. a \cdot \sigma = b \cdot \sigma)$
<proof>

lemma *in_pair_in_set*:

assumes $(A,B) \in \text{set } ((\text{pairs } As))$
shows $A \in \text{set } As \wedge B \in \text{set } As$
<proof>

lemma *in_pairs_sorted_list_of_set_in_set*:

assumes
 finite AAA
 $\forall AA \in AAA. \text{finite } AA$
 $AB_pairs \in (\text{pairs} \circ \text{compare_linorder.sorted_list_of_set}) \text{ ` AAA and}$
 $(A :: _ :: \text{compare}, B) \in \text{set } AB_pairs$
shows $\exists AA. AA \in AAA \wedge A \in AA \wedge B \in AA$
<proof>

lemma *unifiers_Pairs*:

assumes
 finite AAA **and**
 $\forall AA \in AAA. \text{finite } AA$
shows $\text{unifiers } (\text{set } (\text{Pairs } AAA)) = \{\sigma. \text{is_unifiers } \sigma \text{ AAA}\}$
<proof>

end

definition *mgu_sets* AAA = *map_option subst_of* (*unify* (*Pairs* AAA) [])

lemma *mgu_sets_is_imgu*:

fixes AAA :: $('a :: \text{compare}, \text{nat}) \text{ term set set}$ **and** $\sigma :: ('a, \text{nat}) \text{ subst}$
assumes *fin*: *finite* AAA $\forall AA \in AAA. \text{finite } AA$ **and** *mgu_sets* AAA = *Some* σ
shows *is_imgu* σ AAA
<proof>

interpretation *mgu* (\cdot) *Var* :: $_ \Rightarrow ('f :: \text{compare}, \text{nat}) \text{ term } (\circ_s) \text{ renamings_apart}$

Fun undefined mgu_sets
<proof>

interpretation *imgu* (\cdot) *Var* :: $_ \Rightarrow ('f :: \text{compare}, \text{nat}) \text{ term } (\circ_s) \text{ renamings_apart}$

Fun undefined mgu_sets
<proof>

derive *linorder prod*

derive *linorder list*

This part extends and integrates and the Knuth–Bendix order defined in *IsaFoR*.

record *'f weights* =

w :: $'f \times \text{nat} \Rightarrow \text{nat}$
w0 :: *nat*
pr_strict :: $'f \times \text{nat} \Rightarrow 'f \times \text{nat} \Rightarrow \text{bool}$
least :: $'f \Rightarrow \text{bool}$
scf :: $'f \times \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$

class *weighted* =

fixes *weights* :: $'a \text{ weights}$
assumes *weights_adm*:
 admissible_kbo
 $(w \text{ weights}) (w0 \text{ weights}) (\text{pr_strict } \text{weights}) ((\text{pr_strict } \text{weights})^{==}) (\text{least } \text{weights}) (\text{scf } \text{weights})$
and *pr_strict_total*: $f_i = g_j \vee \text{pr_strict } \text{weights } f_i \ g_j \vee \text{pr_strict } \text{weights } g_j \ f_i$
and *pr_strict_asymp*: *asyp* (*pr_strict weights*)
and *scf_ok*: $i < n \implies \text{scf } \text{weights } (f, n) \ i \leq 1$

instantiation *unit* :: *weighted begin*

definition *weights_unit* :: *unit weights where weights_unit* =
 $(w = \text{Suc} \circ \text{snd}, w0 = 1, \text{pr_strict} = \lambda(_, n) (_, m). n > m, \text{least} = \lambda_. \text{True}, \text{scf} = \lambda_. 1)$

instance
 ⟨*proof*⟩
end

global-interpretation *KBO*:

admissible_kbo
 w (*weights* :: '*f* :: *weighted weights*) $w0$ (*weights* :: '*f* :: *weighted weights*)
 $\text{pr_strict weights} ((\text{pr_strict weights})^{==}) \text{least weights scf weights}$
defines $\text{weight} = \text{KBO.weight}$
and $\text{kbo} = \text{KBO.kbo}$
 ⟨*proof*⟩

lemma *kbo_code*[*code*]: $\text{kbo } s \ t =$
 (let $\text{wt} = \text{weight } t$; $\text{ws} = \text{weight } s$ in
 if $\text{vars_term_ms} (\text{KBO.SCF } t) \subseteq \# \text{vars_term_ms} (\text{KBO.SCF } s) \wedge \text{wt} \leq \text{ws}$
 then
 (if $\text{wt} < \text{ws}$ then (True, True)
 else
 (case s of
 Var $y \Rightarrow (\text{False}, \text{case } t \text{ of Var } x \Rightarrow \text{True} \mid \text{Fun } g \ ts \Rightarrow \text{ts} = [] \wedge \text{least weights } g)$
 | Fun $f \ ss \Rightarrow$
 (case t of
 Var $x \Rightarrow (\text{True}, \text{True})$
 | Fun $g \ ts \Rightarrow$
 if $\text{pr_strict weights} (f, \text{length } ss) (g, \text{length } ts)$ then (True, True)
 else if $(f, \text{length } ss) = (g, \text{length } ts)$ then $\text{lex_ext_unbounded kbo } ss \ ts$
 else (False, False))))
 else (False, False))
 ⟨*proof*⟩

definition $\text{less_kbo } s \ t = \text{fst} (\text{kbo } t \ s)$

lemma *less_kbo_gtotal*: $\text{ground } s \Longrightarrow \text{ground } t \Longrightarrow s = t \vee \text{less_kbo } s \ t \vee \text{less_kbo } t \ s$
 ⟨*proof*⟩

lemma *less_kbo_subst*:
fixes $\sigma :: ('f :: \text{weighted}, 'v) \text{subst}$
shows $\text{less_kbo } s \ t \Longrightarrow \text{less_kbo} (s \cdot \sigma) (t \cdot \sigma)$
 ⟨*proof*⟩

lemma *wfP_less_kbo*: wfP less_kbo
 ⟨*proof*⟩

instantiation *term* :: (*weighted, type*) *linorder begin*

definition $\text{leq_term} = (\text{SOME } \text{leq}. \{(s,t). \text{less_kbo } s \ t\} \subseteq \text{leq} \wedge \text{Well_order } \text{leq} \wedge \text{Field } \text{leq} = \text{UNIV})$

lemma *less_trm_extension*: $\{(s,t). \text{less_kbo } s \ t\} \subseteq \text{leq_term}$
 ⟨*proof*⟩

lemma *less_trm_well_order*: $\text{well_order } \text{leq_term}$
 ⟨*proof*⟩

definition $\text{less_eq_term} :: ('a :: \text{weighted}, 'b) \text{term} \Rightarrow _ \Rightarrow \text{bool}$ **where**
 $\text{less_eq_term} = \text{in_rel } \text{leq_term}$

definition $\text{less_term} :: ('a :: \text{weighted}, 'b) \text{term} \Rightarrow _ \Rightarrow \text{bool}$ **where**
 $\text{less_term } s \ t = \text{strict} (\leq) \ s \ t$

```

lemma leq_term_minus_Id:  $leq\_term - Id = \{(x,y). x < y\}$ 
  <proof>

lemma less_term_alt:  $(<) = in\_rel (leq\_term - Id)$ 
  <proof>

instance
  <proof>

end

instantiation term :: (weighted, type) wellorder begin
instance
  <proof>
end

lemma ground_less_less_kbo:  $ground\ s \implies ground\ t \implies s < t \implies less\_kbo\ s\ t$ 
  <proof>

lemma less_kbo_less:  $less\_kbo\ s\ t \implies s < t$ 
  <proof>

lemma is_ground_atm_ground:  $is\_ground\_atm\ t \iff ground\ t$ 
  <proof>

end

```

5 An Executable Algorithm for Clause Subsumption

This theory provides an executable functional implementation of clause subsumption, building on the IsaFoR library.

```

theory Executable_Subsumption
  imports
    IsaFoR_Term
    First_Order_Terms.Matching
begin

```

5.1 Naive Implementation of Clause Subsumption

```

fun subsumes_list where
  subsumes_list [] Ks  $\sigma = True$ 
| subsumes_list (L # Ls) Ks  $\sigma =$ 
  ( $\exists K \in set\ Ks. is\_pos\ K = is\_pos\ L \wedge$ 
   (case match_term_list [(atm_of L, atm_of K)]  $\sigma$  of
    None  $\implies False$ 
  | Some  $\rho \implies subsumes\_list\ Ls\ (remove1\ K\ Ks)\ \rho$ ))

lemma atm_of_map_literal[simp]:  $atm\_of\ (map\_literal\ f\ l) = f\ (atm\_of\ l)$ 
  <proof>

definition extends_subst  $\sigma\ \tau = (\forall x \in dom\ \sigma. \sigma\ x = \tau\ x)$ 

lemma extends_subst_refl[simp]:  $extends\_subst\ \sigma\ \sigma$ 
  <proof>

lemma extends_subst_trans:  $extends\_subst\ \sigma\ \tau \implies extends\_subst\ \tau\ \rho \implies extends\_subst\ \sigma\ \rho$ 
  <proof>

lemma extends_subst_dom:  $extends\_subst\ \sigma\ \tau \implies dom\ \sigma \subseteq dom\ \tau$ 
  <proof>

```

lemma *extends_subst_extends*: $extends_subst\ \sigma\ \tau \implies x \in dom\ \sigma \implies \tau\ x = \sigma\ x$
 ⟨proof⟩

lemma *extends_subst_fun_upd_new*:
 $\sigma\ x = None \implies extends_subst\ (\sigma(x \mapsto t))\ \tau \longleftrightarrow extends_subst\ \sigma\ \tau \wedge \tau\ x = Some\ t$
 ⟨proof⟩

lemma *extends_subst_fun_upd_matching*:
 $\sigma\ x = Some\ t \implies extends_subst\ (\sigma(x \mapsto t))\ \tau \longleftrightarrow extends_subst\ \sigma\ \tau$
 ⟨proof⟩

lemma *extends_subst_empty[simp]*: $extends_subst\ Map.empty\ \tau$
 ⟨proof⟩

lemma *extends_subst_cong_term*:
 $extends_subst\ \sigma\ \tau \implies vars_term\ t \subseteq dom\ \sigma \implies t \cdot subst_of_map\ Var\ \sigma = t \cdot subst_of_map\ Var\ \tau$
 ⟨proof⟩

lemma *extends_subst_cong_lit*:
 $extends_subst\ \sigma\ \tau \implies vars_lit\ L \subseteq dom\ \sigma \implies L \cdot lit\ subst_of_map\ Var\ \sigma = L \cdot lit\ subst_of_map\ Var\ \tau$
 ⟨proof⟩

definition *subsumes_modulo* $C\ D\ \sigma =$
 $(\exists \tau. dom\ \tau = vars_clause\ C \cup dom\ \sigma \wedge extends_subst\ \sigma\ \tau \wedge subst_cls\ C\ (subst_of_map\ Var\ \tau) \subseteq\# D)$

abbreviation *subsumes_list_modulo* **where**
 $subsumes_list_modulo\ Ls\ Ks\ \sigma \equiv subsumes_modulo\ (mset\ Ls)\ (mset\ Ks)\ \sigma$

lemma *vars_clause_add_mset[simp]*: $vars_clause\ (add_mset\ L\ C) = vars_lit\ L \cup vars_clause\ C$
 ⟨proof⟩

lemma *subsumes_list_modulo_Cons*: $subsumes_list_modulo\ (L\ \# Ls)\ Ks\ \sigma \longleftrightarrow$
 $(\exists K \in set\ Ks. \exists \tau. extends_subst\ \sigma\ \tau \wedge dom\ \tau = vars_lit\ L \cup dom\ \sigma \wedge L \cdot lit\ (subst_of_map\ Var\ \tau) = K$
 $\wedge subsumes_list_modulo\ Ls\ (remove1\ K\ Ks)\ \tau)$
 ⟨proof⟩

lemma *decompose_Some_var_terms*: $decompose\ (Fun\ f\ ss)\ (Fun\ g\ ts) = Some\ eqs \implies$
 $f = g \wedge length\ ss = length\ ts \wedge eqs = zip\ ss\ ts \wedge$
 $(\bigcup_{(t,u) \in set\ ((Fun\ f\ ss,\ Fun\ g\ ts)\ \# P). vars_term\ t} =$
 $(\bigcup_{(t,u) \in set\ (eqs\ @\ P). vars_term\ t}$
 ⟨proof⟩

lemma *match_term_list_sound*: $match_term_list\ tus\ \sigma = Some\ \tau \implies$
 $extends_subst\ \sigma\ \tau \wedge dom\ \tau = (\bigcup_{(t,u) \in set\ tus. vars_term\ t} \cup dom\ \sigma \wedge$
 $(\forall (t,u) \in set\ tus. t \cdot subst_of_map\ Var\ \tau = u)$
 ⟨proof⟩

lemma *match_term_list_complete*: $match_term_list\ tus\ \sigma = None \implies$
 $extends_subst\ \sigma\ \tau \implies dom\ \tau = (\bigcup_{(t,u) \in set\ tus. vars_term\ t} \cup dom\ \sigma \implies$
 $(\exists (t,u) \in set\ tus. t \cdot subst_of_map\ Var\ \tau \neq u)$
 ⟨proof⟩

lemma *unique_extends_subst*:
assumes *extends*: $extends_subst\ \sigma\ \tau\ extends_subst\ \sigma\ \rho$ **and**
 dom : $dom\ \tau = vars_term\ t \cup dom\ \sigma\ dom\ \rho = vars_term\ t \cup dom\ \sigma$ **and**
 eq : $t \cdot subst_of_map\ Var\ \rho = t \cdot subst_of_map\ Var\ \tau$
shows $\rho = \tau$
 ⟨proof⟩

lemma *subsumes_list_alt*:
 $subsumes_list\ Ls\ Ks\ \sigma \longleftrightarrow subsumes_list_modulo\ Ls\ Ks\ \sigma$
 ⟨proof⟩

lemma *subsumes_subsumes_list*[code_unfold]:
subsumes (mset Ls) (mset Ks) = *subsumes_list* Ls Ks Map.empty
 ⟨proof⟩

lemma *strictly_subsumes_subsumes_list*[code_unfold]:
strictly_subsumes (mset Ls) (mset Ks) =
 (*subsumes_list* Ls Ks Map.empty ∧ ¬ *subsumes_list* Ks Ls Map.empty)
 ⟨proof⟩

lemma *subsumes_list_filterD*: *subsumes_list* Ls (filter P Ks) σ ⇒ *subsumes_list* Ls Ks σ
 ⟨proof⟩

lemma *subsumes_list_filterI*:
assumes *match*: (∧ L K σ τ. L ∈ set Ls ⇒
match_term_list [(atm_of L, atm_of K)] σ = Some τ ⇒ is_pos L = is_pos K ⇒ P K)
shows *subsumes_list* Ls Ks σ ⇒ *subsumes_list* Ls (filter P Ks) σ
 ⟨proof⟩

lemma *subsumes_list_Cons_filter_iff*:
assumes *sorted_wrt*: sorted_wrt leq (L # Ls) **and** *trans*: transp leq
and *match*: (∧ L K σ τ.
match_term_list [(atm_of L, atm_of K)] σ = Some τ ⇒ is_pos L = is_pos K ⇒ leq L K)
shows *subsumes_list* (L # Ls) (filter (leq L) Ks) σ ↔ *subsumes_list* (L # Ls) Ks σ
 ⟨proof⟩

definition *leq_head* :: ('f::linorder, 'v) term ⇒ ('f, 'v) term ⇒ bool **where**
leq_head t u = (case (root t, root u) of
 (None, _) ⇒ True
 | (_, None) ⇒ False
 | (Some f, Some g) ⇒ f ≤ g)

definition *leq_lit* L K = (case (K, L) of
 (Neg _, Pos _) ⇒ True
 | (Pos _, Neg _) ⇒ False
 | _ ⇒ *leq_head* (atm_of L) (atm_of K))

lemma *transp_leq_lit*[simp]: transp leq_lit
 ⟨proof⟩

lemma *reflp_leq_lit*[simp]: reflp_on A leq_lit
 ⟨proof⟩

lemma *total_leq_lit*[simp]: totalp_on A leq_lit
 ⟨proof⟩

lemma *leq_head_subst*[simp]: *leq_head* t (t · σ)
 ⟨proof⟩

lemma *leq_lit_match*:
fixes L K :: ('f :: linorder, 'v) term literal
shows *match_term_list* [(atm_of L, atm_of K)] σ = Some τ ⇒ is_pos L = is_pos K ⇒ *leq_lit* L K
 ⟨proof⟩

5.2 Optimized Implementation of Clause Subsumption

fun *subsumes_list_filter* **where**
subsumes_list_filter [] Ks σ = True
 | *subsumes_list_filter* (L # Ls) Ks σ =
 (let Ks = filter (leq_lit L) Ks in
 (∃ K ∈ set Ks. is_pos K = is_pos L ∧
 (case *match_term_list* [(atm_of L, atm_of K)] σ of
 None ⇒ False
 | Some ρ ⇒ *subsumes_list_filter* Ls (remove1 K Ks) ρ)))

lemma *sorted_wrt_subsumes_list_subsumes_list_filter*:

sorted_wrt leq_lit Ls \implies *subsumes_list Ls Ks* $\sigma =$ *subsumes_list_filter Ls Ks* σ
 <proof>

5.3 Definition of Deterministic QuickSort

This is the functional description of the standard variant of deterministic QuickSort that always chooses the first list element as the pivot as given by Hoare in 1962. For a list that is already sorted, this leads to $n(n-1)$ comparisons, but as is well known, the average case is much better.

The code below is adapted from Manuel Eberl's *Quick_Sort_Cost* AFP entry, but without invoking probability theory and using a predicate instead of a set.

```
fun quicksort :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  'a list where
  quicksort _ [] = []
| quicksort R (x # xs) =
  quicksort R (filter ( $\lambda y. R y x$ ) xs) @ [x] @ quicksort R (filter ( $\lambda y. \neg R y x$ ) xs)
```

We can easily show that this QuickSort is correct:

theorem *mset_quicksort* [*simp*]: *mset* (quicksort R xs) = *mset* xs
 <proof>

corollary *set_quicksort* [*simp*]: *set* (quicksort R xs) = *set* xs
 <proof>

theorem *sorted_wrt_quicksort*:
assumes *transp* R **and** *totalp_on* (set xs) R **and** *reflp_on* (set xs) R
shows *sorted_wrt* R (quicksort R xs)
 <proof>

End of the material adapted from Eberl's *Quick_Sort_Cost*.

lemma *subsumes_list_subsumes_list_filter*[*abs_def*, *code_unfold*]:
subsumes_list Ls Ks $\sigma =$ *subsumes_list_filter* (quicksort leq_lit Ls) Ks σ
 <proof>

end

6 An Executable Simple Ordered Resolution Prover for First-Order Clauses

This theory provides an executable functional implementation of the *deterministic_RP* prover, building on the *IsaFoR* library for the notion of terms and on the Knuth–Bendix order.

```
theory Executable_FO_Ordered_Resolution_Prover
imports
  Deterministic_FO_Ordered_Resolution_Prover
  Executable_Subsumption
  HOL-Library.Code_Target_Nat
  Show.Show_Instances
  IsaFoR_Term
```

begin

```
global-interpretation RP: deterministic_FO_resolution_prover where
  S =  $\lambda \_.$  {#} and
  subst_atm = ( $\cdot$ ) and
  id_subst = Var ::  $\_ \Rightarrow$  ('f :: {weighted, compare_order}, nat) term and
  comp_subst = ( $\circ_s$ ) and
  renamings_apart = renamings_apart and
  atm_of_atms = Fun undefined and
  mgu = mgu_sets and
  less_atm = less_kbo and
  size_atm = size and
  timestamp_factor = 1 and
  size_factor = 1
```

```

defines deterministic_RP = RP.deterministic_RP
and deterministic_RP_step = RP.deterministic_RP_step
and is_final_dstate = RP.is_final_dstate
and is_reducible_lit = RP.is_reducible_lit
and is_tautology = RP.is_tautology
and maximal_wrt = RP.maximal_wrt
and reduce = RP.reduce
and reduce_all = RP.reduce_all
and reduce_all2 = RP.reduce_all2
and remdups_cls = RP.remdups_cls
and resolve = RP.resolve
and resolve_on = RP.resolve_on
and resolvable = RP.resolvable
and resolvent = RP.resolvent
and resolve_rename = RP.resolve_rename
and resolve_rename_either_way = RP.resolve_rename_either_way
and select_min_weight_clause = RP.select_min_weight_clause
and strictly_maximal_wrt = RP.strictly_maximal_wrt
and strictly_subsume = RP.strictly_subsume
and subsume = RP.subsume
and weight = RP.weight
and St0 = RP.St0
and sorted_list_of_set = linorder.sorted_list_of_set (le_of_comp compare)
and sort_key = linorder.sort_key (le_of_comp compare)
and insort_key = linorder.insort_key (le_of_comp compare)
⟨proof⟩

```

declare

```

RP.deterministic_RP.simps [code]
RP.deterministic_RP_step.simps [code]
RP.is_final_dstate.simps [code]
RP.is_tautology_def [code]
RP.reduce.simps [code]
RP.reduce_all_def [code]
RP.reduce_all2.simps [code]
RP.resolve_rename_def [code]
RP.resolve_rename_either_way_def [code]
RP.select_min_weight_clause.simps [code]
RP.weight.simps [code]
St0_def [code]
substitution_ops.strictly_subsumes_def [code]
substitution_ops.subst_cls_lists_def [code]
substitution_ops.subst_lit_def [code]
substitution_ops.subst_cls_def [code]

```

lemma *remove1_mset_subset_eq*: $remove1_mset\ a\ A \subseteq\# B \longleftrightarrow A \subseteq\# add_mset\ a\ B$
⟨proof⟩

lemma *Bex_cong*: $(\bigwedge b. b \in B \implies P\ b = Q\ b) \implies Bex\ B\ P = Bex\ B\ Q$
⟨proof⟩

lemma *is_reducible_lit_code*[code]: $RP.is_reducible_lit\ Ds\ C\ L =$
 $(\exists D \in set\ Ds. (\exists L' \in set\ D.$
 if *is_pos* $L' = is_neg\ L$ then
 (case *match_term_list* $[(atm_of\ L',\ atm_of\ L)]\ Map.empty\ of$
 None $\Rightarrow False$
 Some $\sigma \Rightarrow subsumes_list\ (remove1\ L'\ D)\ C\ \sigma$)
 else *False*)
⟨proof⟩

lemma [code]:
 ⟨Pairs *AAA* = *concat*
 (*Executable_FO_Ordered_Resolution_Prover.sorted_list_of_set*

$\langle ((\text{pairs} \circ \text{Executable_FO_Ordered_Resolution_Prover.sorted_list_of_set}) \text{ ' AAA})) \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:

$\langle \text{Executable_FO_Ordered_Resolution_Prover.sorted_list_of_set} (\text{set } xs) =$
 $\text{Executable_FO_Ordered_Resolution_Prover.sort_key} (\lambda x. x) (\text{remdups } xs) \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:

$\langle \text{Executable_FO_Ordered_Resolution_Prover.sort_key} f xs =$
 $\text{foldr} (\text{Executable_FO_Ordered_Resolution_Prover.insort_key} f) xs [] \rangle$
 $\langle \text{proof} \rangle$

lemma [code]:

$\langle \text{Executable_FO_Ordered_Resolution_Prover.insort_key} f x [] = [x] \rangle$
 $\langle \text{Executable_FO_Ordered_Resolution_Prover.insort_key} f x (y \# ys) =$
 $(\text{if } \text{le_of_comp } \text{compare} (f x) (f y)$
 $\text{then } x \# y \# ys$
 $\text{else } y \# \text{Executable_FO_Ordered_Resolution_Prover.insort_key} f x ys) \rangle$
 $\langle \text{proof} \rangle$

export-code *St0* in *SML*

export-code *deterministic_RP* in *SML module-name RP*

instantiation *nat* :: *weighted begin*

definition *weights_nat* :: *nat weights where weights_nat* =

$(\lambda w = \text{Suc} \circ \text{prod_encode}, w0 = 1, \text{pr_strict} = \lambda(f, n). f > g \vee f = g \wedge n > m, \text{least} = \lambda n. n = 0, \text{scf} =$
 $\lambda_ _ . 1)$

instance

$\langle \text{proof} \rangle$

end

definition *prover* :: $((\text{nat}, \text{nat}) \text{ Term.term literal list} \times \text{nat}) \text{ list} \Rightarrow \text{bool}$ **where**

prover *N* = $(\text{case } \text{deterministic_RP} (\text{St0 } N 0) \text{ of}$

None $\Rightarrow \text{True}$

| *Some* *R* $\Rightarrow [] \notin \text{set } R)$

theorem *prover_complete_refutation*: *prover* *N* $\longleftrightarrow \text{satisfiable} (\text{RP.grounded_N0 } N)$

$\langle \text{proof} \rangle$

definition *string_literal_of_nat* :: *nat* $\Rightarrow \text{String.literal}$ **where**

string_literal_of_nat *n* = *String.implode* (*show* *n*)

export-code *prover* *Fun* *Var* *Pos* *Neg* *string_literal_of_nat* *0::nat* *Suc* in *SML module-name RPx*

abbreviation *p* $\equiv \text{Fun } 42$

abbreviation *a* $\equiv \text{Fun } 0 []$

abbreviation *b* $\equiv \text{Fun } 1 []$

abbreviation *c* $\equiv \text{Fun } 2 []$

abbreviation *X* $\equiv \text{Var } 0$

abbreviation *Y* $\equiv \text{Var } 1$

abbreviation *Z* $\equiv \text{Var } 2$

value *prover*

$(([\text{Neg} (\text{p}[X, Y, Z]), \text{Pos} (\text{p}[Y, Z, X])], 1),$

$([\text{Pos} (\text{p}[c, a, b])], 1),$

$([\text{Neg} (\text{p}[b, c, a])], 1)]$

$:: ((\text{nat}, \text{nat}) \text{ Term.term literal list} \times \text{nat}) \text{ list}$

value *prover*

$(([\text{Pos} (\text{p}[X, Y])], 1), ([\text{Neg} (\text{p}[X, X])], 1)]$

:: ((nat, nat) Term.term literal list × nat) list)

value prover ([([Neg (p[X,Y,Z]), Pos (p[Y,Z,X])], 1)])
:: ((nat, nat) Term.term literal list × nat) list)

definition mk_MSC015_1 :: nat ⇒ ((nat, nat) Term.term literal list × nat) list **where**
mk_MSC015_1 n =

(let
 init = ([Pos (p (replicate n a))], 1);
 rules = map (λi. ([Neg (p (map Var [0 ..< n - i - 1] @ a # replicate i b)),
 Pos (p (map Var [0 ..< n - i - 1] @ b # replicate i a))], 1)) [0 ..< n];
 goal = ([Neg (p (replicate n b))], 1)
 in init # rules @ [goal])

value prover (mk_MSC015_1 1)
value prover (mk_MSC015_1 2)
value prover (mk_MSC015_1 3)
value prover (mk_MSC015_1 4)
value prover (mk_MSC015_1 5)
value prover (mk_MSC015_1 10)

lemma

assumes

p a a a a a a a a a a a
(∀ x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13.
 ¬ p x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 a ∨
 p x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 b)
(∀ x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12.
 ¬ p x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 a b ∨
 p x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 b a)
(∀ x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11.
 ¬ p x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 a b b ∨
 p x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 b a a)
(∀ x1 x2 x3 x4 x5 x6 x7 x8 x9 x10.
 ¬ p x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 a b b b ∨
 p x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 b a a a)
(∀ x1 x2 x3 x4 x5 x6 x7 x8 x9.
 ¬ p x1 x2 x3 x4 x5 x6 x7 x8 x9 a b b b b ∨
 p x1 x2 x3 x4 x5 x6 x7 x8 x9 b a a a a)
(∀ x1 x2 x3 x4 x5 x6 x7 x8.
 ¬ p x1 x2 x3 x4 x5 x6 x7 x8 a b b b b b ∨
 p x1 x2 x3 x4 x5 x6 x7 x8 b a a a a a)
(∀ x1 x2 x3 x4 x5 x6 x7.
 ¬ p x1 x2 x3 x4 x5 x6 x7 a b b b b b b ∨
 p x1 x2 x3 x4 x5 x6 x7 b a a a a a a)
(∀ x1 x2 x3 x4 x5 x6.
 ¬ p x1 x2 x3 x4 x5 x6 a b b b b b b b ∨
 p x1 x2 x3 x4 x5 x6 b a a a a a a a)
(∀ x1 x2 x3 x4 x5.
 ¬ p x1 x2 x3 x4 x5 a b b b b b b b b ∨
 p x1 x2 x3 x4 x5 b a a a a a a a a)
(∀ x1 x2 x3 x4.
 ¬ p x1 x2 x3 x4 a b b b b b b b b b ∨
 p x1 x2 x3 x4 b a a a a a a a a a)
(∀ x1 x2 x3.
 ¬ p x1 x2 x3 a b b b b b b b b b b ∨
 p x1 x2 x3 b a a a a a a a a a a)
(∀ x1 x2.
 ¬ p x1 x2 a b b b b b b b b b b b ∨
 p x1 x2 b a a a a a a a a a a a)
(∀ x1.
 ¬ p x1 a b b b b b b b b b b b b ∨
 p x1 b a a a a a a a a a a a a a)

```
(¬ p a b b b b b b b b b b b b ∨  
  p b a a a a a a a a a a a a a)  
¬ p b b b b b b b b b b b b b b  
shows False  
⟨proof⟩
```

end