

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

Anders Schlichtkrull, Jasmin Christian Blanchette, and Dmitriy Traytel

December 14, 2021

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	15
3.1	Library	16
3.2	Prover	16
4	Integration of IsaFoR Terms and the Knuth–Bendix Order	43
5	An Executable Algorithm for Clause Subsumption	55
5.1	Naive Implementation of Clause Subsumption	55
5.2	Optimized Implementation of Clause Subsumption	60
5.3	Definition of Deterministic QuickSort	60
6	An Executable Simple Ordered Resolution Prover for First-Order Clauses	61

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

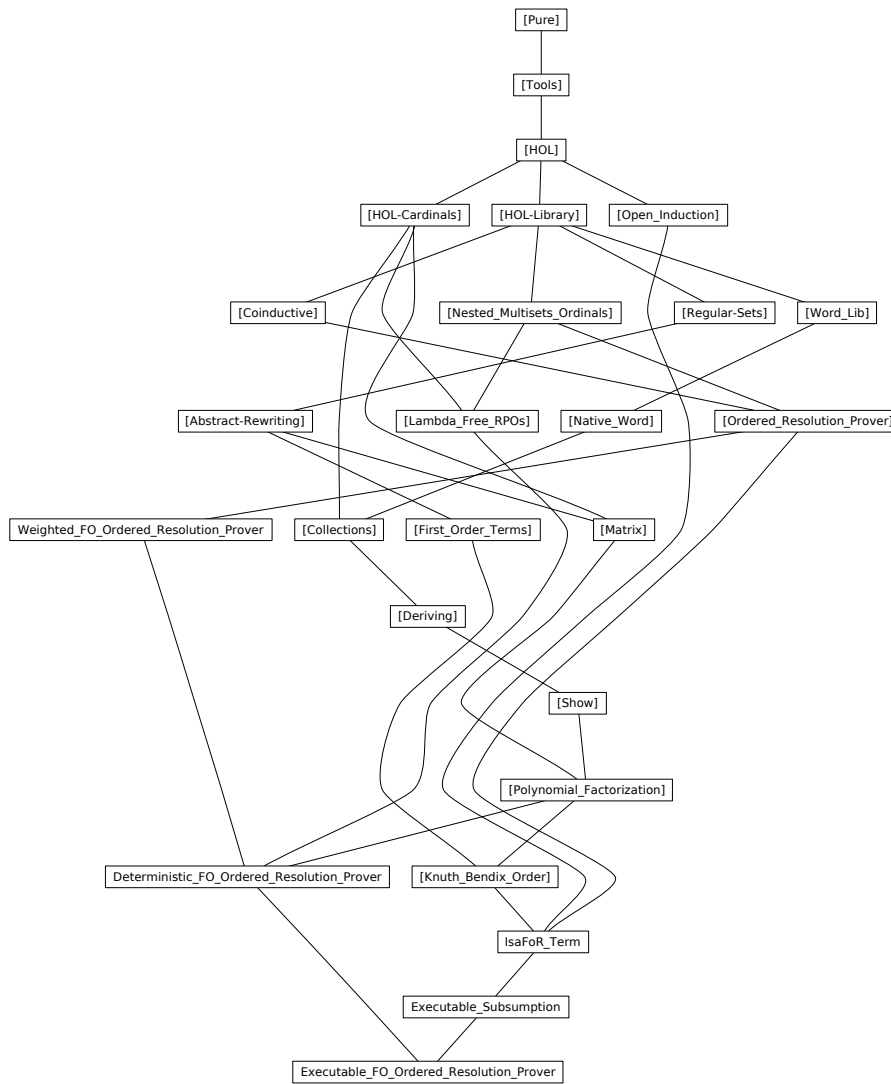


Figure 1: Theory dependency graph

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:  $i < j \implies \text{weight}(C, i) < \text{weight}(C, j)$ 
begin

abbreviation class_of_wstate :: 'a wstate ⇒ 'a clause set where
  class_of_wstate St ≡ class_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
by (cases St) auto

```

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*: $n \leq \text{weight } (C, n)$
by (induct n, simp, metis *weight_mono*[of k Suc k for k] *Suc_le_eq le_less le_trans*)

inductive *weighted_RP* :: 'a wstate \Rightarrow 'a wstate \Rightarrow bool (**infix** \rightsquigarrow_w 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*: $D \in\# \text{image_mset fst } N \Longrightarrow C \in\# \text{image_mset fst } P \Longrightarrow$
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*: $D + \{\#L'\} \in\# \text{image_mset fst } N \Longrightarrow -L = L' \cdot l \sigma \Longrightarrow D \cdot \sigma \subseteq\# C \Longrightarrow$
 $(\forall j. (C + \{\#L\}, j) \in\# P \rightarrow j \leq i) \Longrightarrow$
 $(N, P + \{\#(C + \{\#L\}, i)\}, Q, n) \rightsquigarrow_w (N, P + \{\#(C, i)\}, Q, n)$
| *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*: $(\forall (D, j) \in\# P. \text{weight } (C, i) \leq \text{weight } (D, j)) \Longrightarrow$
 $N = \text{mset_set } ((\lambda D. (D, n)) \text{ `concls_of$
 $(\text{inference_system.inferencess_between } (\text{ord_FO_}\Gamma S) (\text{set_mset } (\text{image_mset fst } Q)) C)) \Longrightarrow$
 $(\{\#\}, P + \{\#(C, i)\}, Q, n) \rightsquigarrow_w (N, \{\#(D, j) \in\# P. D \neq C\}, Q + \{\#(C, i)\}, \text{Suc } n)$

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

proof (induction rule: *weighted_RP.induct*)

case (*backward_subsumption_P* D N C P Q n)

show ?case

by (rule *arg_cong2*[THEN *iffD1*, of $_ _ _ _ (\rightsquigarrow)$, OF $_ _ _ _$
RP.backward_subsumption_P[of D fst `set_mset N C fst `set_mset P - {C}
fst `set_mset Q]])
(use *backward_subsumption_P* in auto)

next

case (*inference_computation* P C i N n Q)

show ?case

by (rule *arg_cong2*[THEN *iffD1*, of $_ _ _ _ (\rightsquigarrow)$, OF $_ _ _ _$
RP.inference_computation[of fst `set_mset N C fst `set_mset Q C
fst `set_mset P - {C}]],
use *inference_computation*(2) *finite_ord_FO_resolution_inferencess_between* in
(auto simp: *comp_def image_comp inference_system.inferencess_between_def*))

qed (use *RP.intros* in *simp_all*)

lemma *final_weighted_RP*: $\neg (\{\#\}, \{\#\}, Q, n) \rightsquigarrow_w St$
by (auto elim: *weighted_RP.cases*)

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* (*clss_of_wstate* (lhd *Sts*))

by (*cases* lhd *Sts*) auto

lemmas *deriv* = *full_chain_imp_chain*[*OF full_deriv*]
lemmas *lhd_lmap_Sts* = *lmap.map_sel(1)*[*OF chain_not_null*[*OF deriv*]]

lemma *deriv_RP*: *chain* (\rightsquigarrow) (*lmap state_of_wstate Sts*)
using *deriv_weighted_RP_imp_RP* **by** (*metis chain_lmap*)

lemma *finite_Sts0_RP*: *finite* (*class_of_state* (*lhd* (*lmap state_of_wstate Sts*)))
using *finite_Sts0_chain_length_pos*[*OF deriv*] **by** *auto*

lemma *empty_P0_RP*: *P_of_state* (*lhd* (*lmap state_of_wstate Sts*)) = {}
using *empty_P0_chain_length_pos*[*OF deriv*] **by** *auto*

lemma *empty_Q0_RP*: *Q_of_state* (*lhd* (*lmap state_of_wstate Sts*)) = {}
using *empty_Q0_chain_length_pos*[*OF deriv*] **by** *auto*

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'* $\iff I \models_s$ *grounding_of_wstate St*
using *RP_model Sts_thms weighted_RP_imp_RP* **by** (*simp only: comp_def*)

abbreviation *S_gQ* :: '*a clause* \implies '*a clause* **where**
S_gQ \equiv *S_Q* (*lmap state_of_wstate Sts*)

interpretation *sq*: *selection S_gQ*

unfolding *S_Q_def* **using** *S_M_selects_subseteq S_M_selects_neg_lits selection_axioms*
by *unfold_locales auto*

interpretation *gd*: *ground_resolution_with_selection S_gQ*
by *unfold_locales*

interpretation *src*: *standard_redundancy_criterion_reductive gd.ord_Γ*
by *unfold_locales*

interpretation *src*: *standard_redundancy_criterion_counterex_reducing gd.ord_Γ*
ground_resolution_with_selection.INTERP S_gQ
by *unfold_locales*

lemmas *ord_Γ_saturated_upto_def* = *src.saturated_upto_def*

lemmas *ord_Γ_saturated_upto_complete* = *src.saturated_upto_complete*

lemmas *ord_Γ_contradiction_Rf* = *src.contradiction_Rf*

theorem *weighted_RP_sound*:

assumes {#} \in *class_of_state* (*Liminf_wstate Sts*)
shows \neg *satisfiable* (*grounding_of_wstate* (*lhd Sts*))
by (*rule RP_sound*[*OF deriv_RP assms, unfolded lhd_lmap_Sts*])

abbreviation *RP_filtered_measure* :: ('*a wclause* \implies *bool*) \implies '*a wstate* \implies *nat* \times *nat* \times *nat* **where**

RP_filtered_measure \equiv λp (*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* \implies '*a wstate* \implies *nat* \times (*nat* \times *nat* \times *nat*) \times (*nat* \times *nat* \times *nat*) **where**

RP_combined_measure \equiv λw *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* **where**

RP_filtered_relation \equiv *natLess* $\langle *lex* \rangle$ *natLess* $\langle *lex* \rangle$ *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* **where**

RP_combined_relation \equiv *natLess* $\langle *lex* \rangle$ *RP_filtered_relation* $\langle *lex* \rangle$ *RP_filtered_relation*

abbreviation ($\text{fst3} :: 'b * 'c * 'd \Rightarrow 'b$) $\equiv \text{fst}$
abbreviation ($\text{snd3} :: 'b * 'c * 'd \Rightarrow 'c$) $\equiv \lambda x. \text{fst} (\text{snd } x)$
abbreviation ($\text{trd3} :: 'b * 'c * 'd \Rightarrow 'd$) $\equiv \lambda x. \text{snd} (\text{snd } x)$

lemma

$\text{wf_RP_filtered_relation}$: $\text{wf } \text{RP_filtered_relation}$ **and**
 $\text{wf_RP_combined_relation}$: $\text{wf } \text{RP_combined_relation}$
unfolding natLess_def **using** wf_less wf_mult **by** auto

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

proof ($\text{induction } N$ arbitrary: M)

case empty
then obtain y **where** $y \in\# M$
by force
then have $(\sum x \in\# M. 1) = (\sum x \in\# M - \{y\}. 1) + \{y\}. 1$
by auto
also have $\dots = (\sum x \in\# M - \{y\}. 1) + (\sum x \in\# \{y\}. 1)$
by ($\text{metis image_mset_union sum_mset.union}$)
also have $\dots > (0 :: \text{nat})$
by auto
finally have $0 < (\sum x \in\# M. \text{Suc } (f x))$
by ($\text{fastforce intro: gr_zeroI}$)
then show $?case$
using empty **by** auto

next

case ($\text{add } x N$)
from $\text{this}(2)$ **have** $(\sum y \in\# N. \text{Suc } (f y)) < (\sum y \in\# M - \{x\}. \text{Suc } (f y))$
using $\text{add}(1)[\text{of } M - \{x\}]$ **by** ($\text{simp add: insert_union_subset_iff}$)
moreover have $\text{add_mset } x (\text{remove1_mset } x M) = M$
by ($\text{meson add.premis add_mset_remove_trivial_If mset_subset_insertD}$)
ultimately show $?case$
by ($\text{metis (no_types) add.commute add_less_cancel_right sum_mset.insert}$)

qed

lemma $\text{multiset_sum_monotone_f'}$:

assumes $CC \subset\# DD$
shows $(\sum (C, i) \in\# CC. \text{Suc } (f C)) < (\sum (C, i) \in\# DD. \text{Suc } (f C))$
using $\text{multiset_sum_of_Suc_f_monotone}[OF \text{ assms, of } f \circ \text{fst}]$
by ($\text{metis (mono_tags) comp_apply image_mset_cong2 split_beta}$)

lemma $\text{filter_mset_strict_subset}$:

assumes $x \in\# M$ **and** $\neg p x$
shows $\{y \in\# M. p y\} \subset\# M$

proof –

have $\text{subteq}: \{E \in\# M. p E\} \subseteq\# M$
by auto
have $\text{count } \{E \in\# M. p E\} x = 0$
using assms **by** auto
moreover have $0 < \text{count } M x$
using assms **by** auto
ultimately have $\text{lt_count}: \text{count } \{y \in\# M. p y\} x < \text{count } M x$
by auto
then show $?thesis$
using subteq **by** ($\text{metis less_not_refl2 subset_mset.le_neq_trans}$)

qed

lemma $\text{weighted_RP_measure_decreasing_N}$:

assumes $St \rightsquigarrow_w St'$ **and** $(C, l) \in\# \text{wN_of_wstate } St$
shows $(\text{RP_filtered_measure } (\lambda Ci. \text{True}) St', \text{RP_filtered_measure } (\lambda Ci. \text{True}) St)$
 $\in \text{RP_filtered_relation}$
using assms **proof** ($\text{induction rule: weighted_RP.induct}$)
case ($\text{backward_subsumption_P } D N C' P Q n$)

then obtain i' **where** $(C', i') \in\# P$
by *auto*
then have $\{\#(E, k) \in\# P. E \neq C'\#\} \subset\# P$
using *filter_mset_strict_subset*[of $(C', i') P \lambda X. \neg \text{fst } X = C'$]
by (*metis* (*mono_tags*, *lifting*) *filter_mset_cong fst_conv prod.case_eq_if*)
then have $(\sum (C, i) \in\# \{\#(E, k) \in\# P. E \neq C'\#\}. \text{Suc } (\text{size } C)) < (\sum (C, i) \in\# P. \text{Suc } (\text{size } C))$
using *multiset_sum_monotone_f'*[of $\{\#(E, k) \in\# P. E \neq C'\#\} P \text{ size}$] **by** *metis*
then show *?case*
unfolding *natLess_def* **by** *auto*
qed (*auto simp: natLess_def*)

lemma *weighted_RP_measure_decreasing_P*:

assumes $St \rightsquigarrow_w St'$ **and** $(C, i) \in\# wP_of_wstate St$
shows $(RP_combined_measure (weight (C, i)) St', RP_combined_measure (weight (C, i)) St)$
 $\in RP_combined_relation$
using *assms proof* (*induction rule: weighted_RP.induct*)
case (*backward_subsumption_P D N C' P Q n*)

define St **where** $St = (N, P, Q, n)$
define P' **where** $P' = \{\#(E, k) \in\# P. E \neq C'\#\}$
define St' **where** $St' = (N, P', Q, n)$

from *backward_subsumption_P* **obtain** i' **where** $(C', i') \in\# P$
by *auto*

then have $P'_sub_P: P' \subset\# P$
unfolding P'_def **using** *filter_mset_strict_subset*[of $(C', i') P \lambda Dj. \text{fst } Dj \neq C'$]
by (*metis* (*no_types*, *lifting*) *filter_mset_cong fst_conv prod.case_eq_if*)

have $P'_subeq_P_filter$:
 $\{\#(Ca, ia) \in\# P'. ia \leq weight (C, i)\#\} \subseteq\# \{\#(Ca, ia) \in\# P. ia \leq weight (C, i)\#\}$
using P'_sub_P **by** (*auto intro: multiset_filter_mono*)

have $\text{fst3 } (RP_combined_measure (weight (C, i)) St')$
 $\leq \text{fst3 } (RP_combined_measure (weight (C, i)) St)$
unfolding $St'_def St_def$ **by** *auto*
moreover have $(\sum (C, i) \in\# \{\#(Ca, ia) \in\# P'. ia \leq weight (C, i)\#\}. \text{Suc } (\text{size } C))$
 $\leq (\sum x \in\# \{\#(Ca, ia) \in\# P. ia \leq weight (C, i)\#\}. \text{case } x \text{ of } (C, i) \Rightarrow \text{Suc } (\text{size } C))$
using $P'_subeq_P_filter$ **by** (*rule sum_image_mset_mono*)
then have $\text{fst3 } (\text{snd3 } (RP_combined_measure (weight (C, i)) St'))$
 $\leq \text{fst3 } (\text{snd3 } (RP_combined_measure (weight (C, i)) St))$
unfolding $St'_def St_def$ **by** *auto*
moreover have $\text{snd3 } (\text{snd3 } (RP_combined_measure (weight (C, i)) St'))$
 $\leq \text{snd3 } (\text{snd3 } (RP_combined_measure (weight (C, i)) St))$
unfolding $St'_def St_def$ **by** *auto*
moreover from $P'_subeq_P_filter$ **have** $\text{size } \{\#(Ca, ia) \in\# P'. ia \leq weight (C, i)\#\}$
 $\leq \text{size } \{\#(Ca, ia) \in\# P. ia \leq weight (C, i)\#\}$
by (*simp add: size_mset_mono*)
then have $\text{trd3 } (\text{snd3 } (RP_combined_measure (weight (C, i)) St'))$
 $< \text{trd3 } (\text{snd3 } (RP_combined_measure (weight (C, i)) St))$
unfolding $St'_def St_def$ **unfolding** fst_def snd_def **by** *auto*
moreover from P'_sub_P **have** $(\sum (C, i) \in\# P'. \text{Suc } (\text{size } C)) < (\sum (C, i) \in\# P. \text{Suc } (\text{size } C))$
using *multiset_sum_monotone_f'*[of $\{\#(E, k) \in\# P. E \neq C'\#\} P \text{ size}$] **unfolding** P'_def **by** *metis*
then have $\text{fst3 } (\text{trd3 } (RP_combined_measure (weight (C, i)) St'))$
 $< \text{fst3 } (\text{trd3 } (RP_combined_measure (weight (C, i)) St))$
unfolding $P'_def St'_def St_def$ **by** *auto*
ultimately show *?case*
unfolding *natLess_def P'_def St'_def St_def* **by** *auto*

next

case (*inference_computation P C' i' N n Q*)
then show *?case*
proof (*cases n ≤ weight (C, i)*)
case *True*
then have $weight (C, i) + 1 - n > weight (C, i) + 1 - \text{Suc } n$

```

  by auto
  then show ?thesis
    unfolding natLess_def by auto
next
case n_nle_w: False

define St :: 'a wstate where St = ({#}, P + {#(C', i')#}, Q, n)
define St' :: 'a wstate where St' = (N, {#(D, j) ∈# P. D ≠ C'#}, Q + {#(C', i')#}, Suc n)
define concls :: 'a wclause set where
  concls = (λD. (D, n)) ' concls_of (inference_system.inferences_between (ord_FO_Γ S)
    (fst ' set_mset Q) C')

have fin: finite concls
  unfolding concls_def using finite_ord_FO_resolution_inferences_between by auto

have {(D, ia) ∈ concls. ia ≤ weight (C, i)} = {}
  unfolding concls_def using n_nle_w by auto
then have {#(D, ia) ∈# mset_set concls. ia ≤ weight (C, i)#} = {#}
  using fin filter_mset_empty_if_finite_and_filter_set_empty[of concls] by auto
then have n_low_weight_empty: {#(D, ia) ∈# N. ia ≤ weight (C, i)#} = {#}
  unfolding inference_computation unfolding concls_def by auto

have weight (C', i') ≤ weight (C, i)
  using inference_computation by auto
then have i'_le_w_Ci: i' ≤ weight (C, i)
  using timestamp_le_weight[of i' C'] by auto

have subs: {#(D, ia) ∈# N + {#(D, j) ∈# P. D ≠ C'#} + (Q + {#(C', i')#}). ia ≤ weight (C, i)#}
  ⊆# {#(D, ia) ∈# {#} + (P + {#(C', i')#}) + Q. ia ≤ weight (C, i)#}
  using n_low_weight_empty by (auto simp: multiset_filter_mono)

have fst3 (RP_combined_measure (weight (C, i)) St')
  ≤ fst3 (RP_combined_measure (weight (C, i)) St)
  unfolding St'_def St_def by auto
moreover have fst (RP_filtered_measure ((λ(D, ia). ia ≤ weight (C, i))) St') =
  (∑ (C, i) ∈# {#(D, ia) ∈# N + {#(D, j) ∈# P. D ≠ C'#} + (Q + {#(C', i')#}).
    ia ≤ weight (C, i)#}. Suc (size C))
  unfolding St'_def by auto
also have ... ≤ (∑ (C, i) ∈# {#(D, ia) ∈# {#} + (P + {#(C', i')#}) + Q. ia ≤ weight (C, i)#}.
  Suc (size C))
  using subs sum_image_mset_mono by blast
also have ... = fst (RP_filtered_measure (λ(D, ia). ia ≤ weight (C, i)) St)
  unfolding St_def by auto
finally have fst3 (snd3 (RP_combined_measure (weight (C, i)) St'))
  ≤ fst3 (snd3 (RP_combined_measure (weight (C, i)) St))
  by auto
moreover have snd3 (snd3 (RP_combined_measure (weight (C, i)) St')) =
  snd3 (snd3 (RP_combined_measure (weight (C, i)) St))
  unfolding St_def St'_def using n_low_weight_empty by auto
moreover have trd3 (snd3 (RP_combined_measure (weight (C, i)) St')) <
  trd3 (snd3 (RP_combined_measure (weight (C, i)) St))
  unfolding St_def St'_def using i'_le_w_Ci
  by (simp add: le_imp_less_Suc multiset_filter_mono size_mset_mono)
ultimately show ?thesis
  unfolding natLess_def St'_def St_def lex_prod_def by force
qed
qed (auto simp: natLess_def)

lemma preserve_min_or_delete_completely:
  assumes St ~w St' (C, i) ∈# wP_of_wstate St
    ∀ k. (C, k) ∈# wP_of_wstate St → i ≤ k
  shows (C, i) ∈# wP_of_wstate St' ∨ (∀ j. (C, j) ∉# wP_of_wstate St')
  using assms proof (induction rule: weighted_RP.induct)

```



```

case (backward_reduction_P D L' N L  $\sigma$  C' P i' Q n)
show ?case
proof (cases C = C' + {#L#})
  case True_outer: True
  then have C_i_in: (C, i)  $\in$  # P + {#(C, i')#}
    using backward_reduction_P by auto
  then have max:  $\bigwedge k. (C, k) \in \# P + \{ \#(C, i') \} \implies k \leq i'$ 
    using backward_reduction_P unfolding True_outer[symmetric] by auto
  then have count (P + {#(C, i')#}) (C, i')  $\geq 1$ 
    by auto
  moreover
  {
    assume asm: count (P + {#(C, i')#}) (C, i') = 1
    then have nin_P: (C, i')  $\notin$  # P
      using not_in_iff by force
    have ?thesis
    proof (cases (C, i) = (C, i'))
      case True
      then have i = i'
        by auto
      then have  $\forall j. (C, j) \in \# P + \{ \#(C, i') \} \longrightarrow j = i'$ 
        using max backward_reduction_P(6) unfolding True_outer[symmetric] by force
      then show ?thesis
        using True_outer[symmetric] nin_P by auto
    next
      case False
      then show ?thesis
        using C_i_in by auto
    qed
  }
  moreover
  {
    assume count (P + {#(C, i')#}) (C, i') > 1
    then have ?thesis
      using C_i_in by auto
  }
  ultimately show ?thesis
    by (cases count (P + {#(C, i')#}) (C, i') = 1) auto
next
  case False
  then show ?thesis
    using backward_reduction_P by auto
qed
qed auto

```

```

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'
using assms preserve_min_or_delete_completely by blast

```

```

lemma preserve_min_P_Sts:
assumes
  enat (Suc k) < llength Sts and
  (C, i)  $\in$  # wP_of_wstate (lth Sts k) and
  (C, j)  $\in$  # wP_of_wstate (lth Sts (Suc k)) and
   $\forall j. (C, j) \in \# wP_of_wstate (lth Sts k) \longrightarrow i \leq j$ 
shows (C, i)  $\in$  # wP_of_wstate (lth Sts (Suc k))
using deriv assms chain_lnth_rel preserve_min_P by metis

```

```

lemma in_lnth_in_Supremum_ldrop:

```

assumes $i < \text{llength } xs$ **and** $x \in \# (\text{lnth } xs \ i)$
shows $x \in \text{Sup_llist } (\text{lmap } \text{set_mset } (\text{ldrop } (\text{enat } i) \ xs))$
using *assms* **by** (*metis* (*no_types*) *ldrop_eq_LConsD* *ldropn_0* *llist.simps(13)* *contra_subsetD*
ldrop_enat *ldropn_Suc_conv* *ldropn_lnth_0* *lnth_lmap* *lnth_subset_Sup_llist*)

lemma *persistent_wclause_in_P_if_persistent_clause_in_P*:

assumes $C \in \text{Liminf_llist } (\text{lmap } P_of_state (\text{lmap } \text{state_of_wstate } Sts))$
shows $\exists i. (C, i) \in \text{Liminf_llist } (\text{lmap } (\text{set_mset} \circ wP_of_wstate) Sts)$

proof –

obtain t_C **where** t_C_p :
 $\text{enat } t_C < \text{llength } Sts$
 $\wedge t. t_C \leq t \implies t < \text{llength } Sts \implies C \in P_of_state (\text{state_of_wstate } (\text{lnth } Sts \ t))$
using *assms* **unfolding** *Liminf_llist_def* **by** *auto*
then obtain i **where** i_p :
 $(C, i) \in \# wP_of_wstate (\text{lnth } Sts \ t_C)$
using t_C_p **by** (*cases* *lnth* *Sts* t_C) *force*

have $Ci_in_nth_wP: \exists i. (C, i) \in \# wP_of_wstate (\text{lnth } Sts \ (t_C + t))$ **if** $t_C + t < \text{llength } Sts$
for t
using *that* $t_C_p(2)$ [*of* $t_C + _$] **by** (*cases* *lnth* *Sts* $(t_C + t)$) *force*

define $\text{in_Sup_wP} :: \text{nat} \implies \text{bool}$ **where**

$\text{in_Sup_wP} = (\lambda i. (C, i) \in \text{Sup_llist } (\text{lmap } (\text{set_mset} \circ wP_of_wstate) (\text{ldrop } t_C \ Sts)))$

have $\text{in_Sup_wP } i$

using i_p *assms(1)* *in_lnth_in_Supremum* *ldrop*[*of* t_C *lmap* $wP_of_wstate \ Sts \ (C, i)$] t_C_p
by (*simp* *add: in_Sup_wP_def* *llist.map_comp*)

then obtain j **where** $j_p: \text{is_least } \text{in_Sup_wP } j$

unfolding *in_Sup_wP_def*[*symmetric*] **using** *least_exists* **by** *metis*

then have $\forall i. (C, i) \in \text{Sup_llist } (\text{lmap } (\text{set_mset} \circ wP_of_wstate) (\text{ldrop } t_C \ Sts)) \longrightarrow j \leq i$

unfolding *is_least_def* *in_Sup_wP_def* **using** *not_less* **by** *blast*

then have $j_smallest$:

$\wedge i t. \text{enat } (t_C + t) < \text{llength } Sts \implies (C, i) \in \# wP_of_wstate (\text{lnth } Sts \ (t_C + t)) \implies j \leq i$
unfolding *comp_def*

by (*smt* *add.commute* *ldrop_enat* *ldrop_eq_LConsD* *ldrop_ldrop* *ldropn_Suc_conv* *ldropn_plus_enat_simps(1)* *lnth_ldropn* *Sup_llist_def* *UN_I* *ldrop_lmap* *llength_lmap* *lnth_lmap* *mem_Collect_eq*)

from j_p **have** $\exists t_Cj. t_Cj < \text{llength } (\text{ldrop } (\text{enat } t_C) \ Sts)$

$\wedge (C, j) \in \# wP_of_wstate (\text{lnth } (\text{ldrop } t_C \ Sts) \ t_Cj)$

unfolding *in_Sup_wP_def* *Sup_llist_def* *is_least_def* **by** *simp*

then obtain t_Cj **where** j_p :

$(C, j) \in \# wP_of_wstate (\text{lnth } Sts \ (t_C + t_Cj))$

$\text{enat } (t_C + t_Cj) < \text{llength } Sts$

by (*smt* *add.commute* *ldrop_enat* *ldrop_eq_LConsD* *ldrop_ldrop* *ldropn_Suc_conv* *ldropn_plus_enat_simps(1)* *lhd_ldropn*)

have Ci_stays :

$t_C + t_Cj + t < \text{llength } Sts \implies (C, j) \in \# wP_of_wstate (\text{lnth } Sts \ (t_C + t_Cj + t))$ **for** t

proof (*induction* t)

case 0

then show *?case*

using j_p **by** (*simp* *add: add.commute*)

next

case (*Suc* t)

have $\text{any_Ck_in_wP}: j \leq k$ **if** $(C, k) \in \# wP_of_wstate (\text{lnth } Sts \ (t_C + t_Cj + t))$ **for** k

using *that* j_p $j_smallest$ *Suc*

by (*smt* *Suc_ile_eq* *add.commute* *add.left_commute* *add_Suc* *less_imp_le* *plus_enat_simps(1)* *the_enat.simps*)

from *Suc* **have** $Cj_in_wP: (C, j) \in \# wP_of_wstate (\text{lnth } Sts \ (t_C + t_Cj + t))$

by (*metis* (*no_types*, *opaque_lifting*) *Suc_ile_eq* *add.commute* *add_Suc_right* *less_imp_le*)

moreover have $C \in P_of_state (\text{state_of_wstate } (\text{lnth } Sts \ (\text{Suc } (t_C + t_Cj + t))))$

using $t_C_p(2)$ *Suc.prem*s **by** *auto*

then have $\exists k. (C, k) \in \# wP_of_wstate (\text{lnth } Sts \ (\text{Suc } (t_C + t_Cj + t)))$

by (*smt* *Suc.prem*s $Ci_in_nth_wP$ *add.commute* *add.left_commute* *add_Suc_right* *enat_ord_code(4)*)

ultimately have $(C, j) \in \# wP_of_wstate (lnth\ Sts\ (Suc\ (t_C + t_Cj + t)))$
using *preserve_min_P_Sts Cj_in_wP any_Ck_in_wP Suc.prem* **by force**
then have $(C, j) \in \# lnth (lmap\ wP_of_wstate\ Sts)\ (Suc\ (t_C + t_Cj + t))$
using *Suc.prem* **by auto**
then show *?case*
by *(smt\ Suc.prem\ add.commute\ add_Suc_right\ lnth_lmap)*
qed
then have $(\bigwedge t. t_C + t_Cj \leq t \implies t < llength\ (lmap\ (set_mset \circ wP_of_wstate)\ Sts) \implies (C, j) \in \# wP_of_wstate\ (lnth\ Sts\ t))$
using *Ci_stays[of_ - (t_C + t_Cj)]* **by** *(metis\ le_add_diff_inverse\ llength_lmap)*
then have $(C, j) \in Liminf_llist\ (lmap\ (set_mset \circ wP_of_wstate)\ Sts)$
unfolding *Liminf_llist_def* **using** *j_p* **by auto**
then show $\exists i. (C, i) \in Liminf_llist\ (lmap\ (set_mset \circ wP_of_wstate)\ Sts)$
by auto
qed

lemma *lfinite_not_LNil_nth_llast*:
assumes *lfinite Sts and Sts \neq LNil*
shows $\exists i < llength\ Sts. lnth\ Sts\ i = llast\ Sts \wedge (\forall j < llength\ Sts. j \leq i)$
using *assms* **proof** *(induction\ rule:\ lfinite.induct)*
case *(lfinite_LConsI xs x)*
then show *?case*
proof *(cases\ xs = LNil)*
case *True*
show *?thesis*
using *True\ zero_enat_def* **by auto**
next
case *False*
then obtain *i* **where**
 $i_p: enat\ i < llength\ xs \wedge lnth\ xs\ i = llast\ xs \wedge (\forall j < llength\ xs. j \leq enat\ i)$
using *lfinite_LConsI* **by auto**
then have $enat\ (Suc\ i) < llength\ (LCons\ x\ xs)$
by *(simp\ add:\ Suc_ile_eq)*
moreover from *i_p* **have** $lnth\ (LCons\ x\ xs)\ (Suc\ i) = llast\ (LCons\ x\ xs)$
by *(metis\ gr_implies_not_zero\ llast_LCons\ llength_inull\ lnth_Suc_LCons)*
moreover from *i_p* **have** $\forall j < llength\ (LCons\ x\ xs). j \leq enat\ (Suc\ i)$
by *(metis\ antisym_conv2\ eSuc_enat\ eSuc_ile_mono\ ileI1\ iless_Suc_eq\ llength_LCons)*
ultimately show *?thesis*
by auto
qed
qed auto

lemma *fair_if_lfinite*:
assumes *fin: lfinite Sts*
shows *fair_state_seq (lmap\ state_of_wstate\ Sts)*
proof *(rule\ ccontr)*
assume *unfair: \neg fair_state_seq (lmap\ state_of_wstate\ Sts)*

have *no_inf_from_last: $\forall y. \neg llast\ Sts \rightsquigarrow_w y$*
using *fin\ full_chain_iff_chain[of\ (\rightsquigarrow_w)\ Sts]* **full_deriv** **by auto**

from *unfair* **obtain** *C* **where**
 $C \in Liminf_llist\ (lmap\ N_of_state\ (lmap\ state_of_wstate\ Sts))$
 $\cup Liminf_llist\ (lmap\ P_of_state\ (lmap\ state_of_wstate\ Sts))$
unfolding *fair_state_seq_def\ Liminf_state_def* **by auto**
then obtain *i* **where** *i_p*:
 $enat\ i < llength\ Sts$
 $\bigwedge j. i \leq j \implies enat\ j < llength\ Sts \implies$
 $C \in N_of_state\ (state_of_wstate\ (lnth\ Sts\ j)) \cup P_of_state\ (state_of_wstate\ (lnth\ Sts\ j))$
unfolding *Liminf_llist_def* **by auto**

have *C_in_llast*:
 $C \in N_of_state\ (state_of_wstate\ (llast\ Sts)) \cup P_of_state\ (state_of_wstate\ (llast\ Sts))$

```

proof –
  obtain  $l$  where
     $l\_p$ :  $enat\ l < llength\ Sts \wedge lnth\ Sts\ l = llast\ Sts \wedge (\forall j < llength\ Sts.\ j \leq enat\ l)$ 
    using  $fin\ lfinite\_not\_LNil\_nth\_llast\ i\_p(1)$  by  $fastforce$ 
  then have
     $C \in N\_of\_state\ (state\_of\_wstate\ (lnth\ Sts\ l)) \cup P\_of\_state\ (state\_of\_wstate\ (lnth\ Sts\ l))$ 
    using  $i\_p(1)\ i\_p(2)[of\ l]$  by  $auto$ 
  then show  $?thesis$ 
    using  $l\_p$  by  $auto$ 
qed

define  $N :: 'a\ wclause\ multiset$  where  $N = wN\_of\_wstate\ (llast\ Sts)$ 
define  $P :: 'a\ wclause\ multiset$  where  $P = wP\_of\_wstate\ (llast\ Sts)$ 
define  $Q :: 'a\ wclause\ multiset$  where  $Q = wQ\_of\_wstate\ (llast\ Sts)$ 
define  $n :: nat$  where  $n = n\_of\_wstate\ (llast\ Sts)$ 

{
  assume  $N\_of\_state\ (state\_of\_wstate\ (llast\ Sts)) \neq \{\}$ 
  then obtain  $D\ j$  where  $(D, j) \in \# N$ 
    unfolding  $N\_def$  by  $(cases\ llast\ Sts)\ auto$ 
  then have  $llast\ Sts \rightsquigarrow_w (N - \{\#(D, j)\#}, P + \{\#(D, j)\#}, Q, n)$ 
    using  $weighted\_RP.clause\_processing[of\ N - \{\#(D, j)\#}\ D\ j\ P\ Q\ n]$ 
    unfolding  $N\_def\ P\_def\ Q\_def\ n\_def$  by  $auto$ 
  then have  $\exists St'. llast\ Sts \rightsquigarrow_w St'$ 
    by  $auto$ 
}
moreover
{
  assume  $a$ :  $N\_of\_state\ (state\_of\_wstate\ (llast\ Sts)) = \{\}$ 
  then have  $b$ :  $N = \{\#\}$ 
    unfolding  $N\_def$  by  $(cases\ llast\ Sts)\ auto$ 
  from  $a$  have  $C \in P\_of\_state\ (state\_of\_wstate\ (llast\ Sts))$ 
    using  $C\_in\_llast$  by  $auto$ 
  then obtain  $D\ j$  where  $(D, j) \in \# P$ 
    unfolding  $P\_def$  by  $(cases\ llast\ Sts)\ auto$ 
  then have  $weight\ (D, j) \in weight\ 'set\_mset\ P$ 
    by  $auto$ 
  then have  $\exists w.$   $is\_least\ (\lambda w.\ w \in (weight\ 'set\_mset\ P))\ w$ 
    using  $least\_exists$  by  $auto$ 
  then have  $\exists D\ j.$   $(\forall (D', j') \in \# P.\ weight\ (D, j) \leq weight\ (D', j') \wedge (D, j) \in \# P)$ 
    using  $assms\ linorder\_not\_less$  unfolding  $is\_least\_def$  by  $(auto\ 6\ 0)$ 
  then obtain  $D\ j$  where
     $min$ :  $(\forall (D', j') \in \# P.\ weight\ (D, j) \leq weight\ (D', j'))$  and
     $Dj\_in\_p$ :  $(D, j) \in \# P$ 
    by  $auto$ 
  from  $min$  have  $min$ :  $(\forall (D', j') \in \# P - \{\#(D, j)\#}.\ weight\ (D, j) \leq weight\ (D', j'))$ 
    using  $mset\_subset\_diff\_self[OF\ Dj\_in\_p]$  by  $auto$ 

  define  $N'$  where
     $N' = mset\_set\ ((\lambda D'. (D', n))\ 'concls\_of\ (inference\_system.inferences\_between\ (ord\_FO\_Gamma\ S)\ (set\_mset\ (image\_mset\ fst\ Q))\ D))$ 

  have  $llast\ Sts \rightsquigarrow_w (N', \{\#(D', j') \in \# P - \{\#(D, j)\#}, D' \neq D\#\}, Q + \{\#(D, j)\#}, Suc\ n)$ 
    using  $weighted\_RP.inference\_computation[of\ P - \{\#(D, j)\#}\ D\ j\ N'\ n\ Q,\ OF\ min\ N'\_def]$ 
     $of\_wstate\_split[symmetric,\ of\ llast\ Sts]\ Dj\_in\_p$ 
    unfolding  $N\_def[symmetric]\ P\_def[symmetric]\ Q\_def[symmetric]\ n\_def[symmetric]\ b$  by  $auto$ 
  then have  $\exists St'. llast\ Sts \rightsquigarrow_w St'$ 
    by  $auto$ 
}
ultimately have  $\exists St'. llast\ Sts \rightsquigarrow_w St'$ 
  by  $auto$ 
then show  $False$ 
  using  $no\_inf\_from\_last$  by  $metis$ 

```

qed

lemma *N_of_state_state_of_wstate_wN_of_wstate:*

assumes $C \in N_of_state (state_of_wstate St)$

shows $\exists i. (C, i) \in \# wN_of_wstate St$

by (*smt* *N_of_state.elims* *assms* *eq_fst_iff_fstI* *fst_conv* *image_iff_of_wstate_split* *set_image_mset* *state_of_wstate.simps*)

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

by (*metis* (*mono_guards_query_query*) *N_of_state.simps* *fst_conv* *image_eqI* *of_wstate_split* *set_image_mset* *state_of_wstate.simps*)

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

by (*metis* (*mono_guards_query_query*) *P_of_state.simps* *fst_conv* *image_eqI* *of_wstate_split* *set_image_mset* *state_of_wstate.simps*)

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

by (*metis* (*mono_guards_query_query*) *Q_of_state.simps* *fst_conv* *image_eqI* *of_wstate_split* *set_image_mset* *state_of_wstate.simps*)

lemma *n_of_wstate_weighted_RP_increasing:* $St \rightsquigarrow_w St' \implies n_of_wstate St \leq n_of_wstate St'$

by (*induction* *rule: weighted_RP.induct*) *auto*

lemma *nth_of_wstate_monotonic:*

assumes $j < llength Sts$ **and** $i \leq j$

shows $n_of_wstate (lnth Sts i) \leq n_of_wstate (lnth Sts j)$

using *assms* **proof** (*induction* $j - i$ *arbitrary: i*)

case (*Suc* x)

then have $x = j - (i + 1)$

by *auto*

then have $n_of_wstate (lnth Sts (i + 1)) \leq n_of_wstate (lnth Sts j)$

using *Suc* **by** *auto*

moreover have $i < j$

using *Suc* **by** *auto*

then have *Suc* $i < llength Sts$

using *Suc* **by** (*metis* *enat_ord_simps(2)* *le_less_Suc_eq* *less_le_trans* *not_le*)

then have $lnth Sts i \rightsquigarrow_w lnth Sts (Suc i)$

using *deriv* *chain* *lnth_rel*[*of* (\rightsquigarrow_w) *Sts* i] **by** *auto*

then have $n_of_wstate (lnth Sts i) \leq n_of_wstate (lnth Sts (i + 1))$

using *n_of_wstate_weighted_RP_increasing*[*of* $lnth Sts i$ $lnth Sts (i + 1)$] **by** *auto*

ultimately show *?case*

by *auto*

qed *auto*

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 (*rule* *lnth_rel_chain*)

show $\neg lnull (lmap m (ldrop (enat k) Sts))$

using *assms* **by** *auto*

next

from *inf* **have** $ldrop_inf: llength (ldrop (enat k) Sts) = \infty \wedge \neg lfinite (ldrop (enat k) Sts)$

using *inf* **by** (*auto* *simp: llength_eq_infty_conv_lfinite*)

{

fix $j :: nat$

define *St* **where** $St = lnth (ldrop (enat k) Sts) j$

define *St'* **where** $St' = lnth (ldrop (enat k) Sts) (j + 1)$

have $P': P St \wedge P St'$

unfolding *St_def* *St'_def* **using** *P* **by** *auto*

```

from ldrop_inf have  $R \text{ St } \text{St}'$ 
  unfolding St_def St'_def
  using non_infer_chain infinite_chain_lnth_rel[of ldrop (enat k) Sts R j] by auto
then have  $(m \text{ St}', m \text{ St}) \in mR$ 
  using measure_decreasing P' by auto
then have  $(\text{lnth } (\text{lmap } m \text{ (ldrop (enat } k) \text{ Sts)) } (j + 1), \text{lnth } (\text{lmap } m \text{ (ldrop (enat } k) \text{ Sts)) } j) \in mR$ 
  unfolding St_def St'_def using lnth_lmap
  by  $(\text{smt enat.distinct}(1) \text{ enat\_add\_left\_cancel enat\_ord\_simps}(4) \text{ inf ldrop\_lmap llength\_lmap lnth\_ldrop\_plus\_enat\_simps}(3))$ 
}
then show  $\forall j. \text{enat } (j + 1) < \text{length } (\text{lmap } m \text{ (ldrop (enat } k) \text{ Sts)}) \longrightarrow$ 
 $(\lambda x y. (x, y) \in mR)^{-1-1} (\text{lnth } (\text{lmap } m \text{ (ldrop (enat } k) \text{ Sts)) } j)$ 
 $(\text{lnth } (\text{lmap } m \text{ (ldrop (enat } k) \text{ Sts)) } (j + 1))$ 
by blast
qed

```

theorem *weighted_RP_fair*: *fair_state_seq (lmap state_of_wstate Sts)*

proof (*rule ccontr*)

assume *asm*: $\neg \text{fair_state_seq } (\text{lmap state_of_wstate } \text{Sts})$

then **have** *inff*: $\neg \text{lfinite } \text{Sts}$ **using** *fair_if_finite*

by *auto*

then **have** *inf*: $\text{length } \text{Sts} = \infty$

using *llength_eq_infy_conv_lfinite* **by** *auto*

from *asm* **obtain** *C* **where**

$C \in \text{Liminf_llist } (\text{lmap } N_of_state \text{ (lmap state_of_wstate } \text{Sts)})$
 $\cup \text{Liminf_llist } (\text{lmap } P_of_state \text{ (lmap state_of_wstate } \text{Sts)})$

unfolding *fair_state_seq_def Liminf_state_def* **by** *auto*

then **show** *False*

proof

assume $C \in \text{Liminf_llist } (\text{lmap } N_of_state \text{ (lmap state_of_wstate } \text{Sts)})$

then **obtain** *x* **where** $\text{enat } x < \text{length } \text{Sts}$

$\forall xa. x \leq xa \wedge \text{enat } xa < \text{length } \text{Sts} \longrightarrow C \in N_of_state \text{ (state_of_wstate } (\text{lnth } \text{Sts } xa))$

unfolding *Liminf_llist_def* **by** *auto*

then **have** $\exists k. \forall j. k \leq j \longrightarrow (\exists i. (C, i) \in \# wN_of_wstate \text{ (lnth } \text{Sts } j))$

unfolding *Liminf_llist_def* **by** $(\text{force simp add: inf } N_of_state_state_of_wstate_wN_of_wstate)$

then **obtain** *k* **where** *k_p*:

$\bigwedge j. k \leq j \implies \exists i. (C, i) \in \# wN_of_wstate \text{ (lnth } \text{Sts } j)$

unfolding *Liminf_llist_def*

by *auto*

have *chain_drop_Sts*: $\text{chain } (\sim_w) \text{ (ldrop } k \text{ Sts)}$

using *deriv_inf_inff* **by** $(\text{simp add: inf_chain_ldropn_chain ldrop_enat})$

have *in_N_j*: $\bigwedge j. \exists i. (C, i) \in \# wN_of_wstate \text{ (lnth } (\text{ldrop } k \text{ Sts}) j)$

using *k_p* **by** $(\text{simp add: add.commute inf})$

then **have** $\text{chain } (\lambda x y. (x, y) \in \text{RP_filtered_relation})^{-1-1} (\text{lmap } (\text{RP_filtered_measure } (\lambda Ci. \text{True})) (\text{ldrop } k \text{ Sts}))$

using *inff_inf_weighted_RP_measure_decreasing_N chain_drop_Sts*

$\text{infinite_chain_relation_measure}$ [of $\lambda \text{St}. \exists i. (C, i) \in \# wN_of_wstate \text{ St } (\sim_w)$] **by** *blast*

then **show** *False*

using *wfP_iff_no_infinite_down_chain_llist*[of $\lambda x y. (x, y) \in \text{RP_filtered_relation}$]
wf_RP_filtered_relation inff

by $(\text{metis } (\text{no_types, lifting}) \text{ inf_llist_lnth ldrop_enat_inf_llist lfinite_inf_llist lfinite_lmap wfPUNIVI wf_induct_rule})$

next

assume *asm*: $C \in \text{Liminf_llist } (\text{lmap } P_of_state \text{ (lmap state_of_wstate } \text{Sts)})$

from *asm* **obtain** *i* **where** *i_p*:

$\text{enat } i < \text{length } \text{Sts}$

$\bigwedge j. i \leq j \wedge \text{enat } j < \text{length } \text{Sts} \implies C \in P_of_state \text{ (state_of_wstate } (\text{lnth } \text{Sts } j))$

unfolding *Liminf_llist_def* **by** *auto*

then **obtain** *i* **where** $(C, i) \in \text{Liminf_llist } (\text{lmap } (\text{set_mset } \circ wP_of_wstate) \text{ Sts})$

using *persistent_wclause_in_P_if_persistent_clause_in_P*[of *C*] **using** *asm inf* **by** *auto*

then **have** $\exists l. \forall k \geq l. (C, i) \in (\text{set_mset } \circ wP_of_wstate) \text{ (lnth } \text{Sts } k)$

unfolding *Liminf_llist_def* **using** *inff inf* **by** *auto*

```

then obtain  $k$  where  $k_p$ :
  ( $\forall k' \geq k. (C, i) \in (\text{set\_mset} \circ \text{wP\_of\_wstate}) (\text{lth } \text{Sts } k')$ )
  by blast
have  $C_i$  in:  $\forall k'. (C, i) \in (\text{set\_mset} \circ \text{wP\_of\_wstate}) (\text{lth } (\text{ldrop } k \text{ Sts}) k')$ 
  using  $k_p$  lth_ldrop[of  $k$   $\text{Sts}$ ] inf inf by force
then have  $C_i$  inn:  $\forall k'. (C, i) \in \# (\text{wP\_of\_wstate}) (\text{lth } (\text{ldrop } k \text{ Sts}) k')$ 
  by auto
have chain ( $\sim_w$ ) ( $\text{ldrop } k \text{ Sts}$ )
  using deriv inf_chain_ldropn_chain inf inf by (simp add: inf_chain_ldropn_chain ldrop_enat)
then have chain ( $\lambda x y. (x, y) \in \text{RP\_combined\_relation}$ )-1-1
  (lmap ( $\text{RP\_combined\_measure } (\text{weight } (C, i))$ ) ( $\text{ldrop } k \text{ Sts}$ ))
  using inf inf  $C_i$  in weighted_RP_measure_decreasing_P
  infinite_chain_relation_measure[of  $\lambda \text{St}. (C, i) \in \# \text{wP\_of\_wstate } \text{St } (\sim_w)$ 
     $\text{RP\_combined\_measure } (\text{weight } (C, i))$  ]
  by auto
then show False
  using wfP_iff_no_infinite_down_chain_llist[of  $\lambda x y. (x, y) \in \text{RP\_combined\_relation}$ ]
  wf_RP_combined_relation inf
  by (smt inf_llist_lth ldrop_enat inf_llist lfinite_inf_llist lfinite_lmap wfPUNIVI
    wf_induct_rule)
qed
qed

corollary weighted_RP_saturated: src.saturated_upto (Liminf_llist (lmap grounding_of_wstate  $\text{Sts}$ ))
  using RP_saturated_if_fair[OF deriv_RP weighted_RP_fair empty_Q0_RP, unfolded llist.map_comp]
  by simp

corollary weighted_RP_complete:
   $\neg \text{satisfiable } (\text{grounding\_of\_wstate } (\text{lhd } \text{Sts})) \implies \{\#\} \in Q\_of\_state (\text{Liminf\_wstate } \text{Sts})$ 
  using RP_complete_if_fair[OF deriv_RP weighted_RP_fair empty_Q0_RP, simplified lhd_lmap_Sts] .

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
   $S :: ('a :: \text{wellorder}) \text{ clause} \Rightarrow 'a \text{ clause}$  and
   $\text{subst\_atm} :: 'a \Rightarrow 's \Rightarrow 'a$  and
   $\text{id\_subst} :: 's$  and
   $\text{comp\_subst} :: 's \Rightarrow 's \Rightarrow 's$  and
   $\text{renamings\_apart} :: 'a \text{ literal multiset list} \Rightarrow 's \text{ list}$  and
   $\text{atm\_of\_atms} :: 'a \text{ list} \Rightarrow 'a$  and
   $\text{mgu} :: 'a \text{ set set} \Rightarrow 's \text{ option}$  and
   $\text{less\_atm} :: 'a \Rightarrow 'a \Rightarrow \text{bool} +$ 
fixes
   $\text{size\_atm} :: 'a \Rightarrow \text{nat}$  and
   $\text{size\_factor} :: \text{nat}$  and
   $\text{timestamp\_factor} :: \text{nat}$ 
assumes
   $\text{timestamp\_factor\_pos: timestamp\_factor} > 0$ 
begin

fun weight ::  $'a \text{ wclause} \Rightarrow \text{nat}$  where
   $\text{weight } (C, i) = \text{size\_factor} * \text{size\_multiset } (\text{size\_literal } \text{size\_atm}) C + \text{timestamp\_factor} * i$ 

lemma weight_mono:  $i < j \implies \text{weight } (C, i) < \text{weight } (C, j)$ 
  using timestamp_factor_pos by simp

declare weight.simps [simp del]

sublocale wrp: weighted_FO_resolution_prover _____ weight

```

```

by unfold_locales (rule weight_mono)

notation wrp.weighted_RP (infix  $\rightsquigarrow_w$  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:  $apfst\ f\ x = (f\ (fst\ x),\ snd\ x)$ 
by (rule apfst_conv[of _ fst x snd x for x, unfolded prod.collapse])

```

```

lemma apfst_comp_rpair_const:  $apfst\ f \circ (\lambda x. (x, y)) = (\lambda x. (x, y)) \circ f$ 
by (simp add: comp_def)

```

```

lemma length_remove1_less[termination_simp]:  $x \in set\ xs \implies length\ (remove1\ x\ xs) < length\ xs$ 
by (induct xs) auto

```

```

lemma map_filter_neq_eq_filter_map:
 $map\ f\ (filter\ (\lambda y. f\ x \neq f\ y)\ xs) = filter\ (\lambda z. f\ x \neq z)\ (map\ f\ xs)$ 
by (induct xs) auto

```

```

lemma mset_map_remdups_gen:
 $mset\ (map\ f\ (remdups\_gen\ f\ xs)) = mset\ (remdups\_gen\ (\lambda x. x)\ (map\ f\ xs))$ 
by (induct f xs rule: remdups_gen.induct) (auto simp: map_filter_neq_eq_filter_map)

```

```

lemma mset_remdups_gen_ident:  $mset\ (remdups\_gen\ (\lambda x. x)\ xs) = mset\_set\ (set\ xs)$ 

```

```

proof -

```

```

  have  $f = (\lambda x. x) \implies mset\ (remdups\_gen\ f\ xs) = mset\_set\ (set\ xs)$  for  $f$ 

```

```

  proof (induct f xs rule: remdups_gen.induct)

```

```

    case ( $2\ f\ xs$ )

```

```

    note  $ih = this(1)$  and  $f = this(2)$ 

```

```

    show ?case

```

```

      unfolding  $f\ remdups\_gen.simps\ ih[OF\ f,\ unfolded\ f]\ mset.simps$ 

```

```

      by (metis finite_set list.simps(15) mset_set.insert_remove removeAll_filter_not_eq remove_code(1) remove_def)

```

```

    qed simp

```

```

  then show ?thesis

```

```

    by simp

```

```

qed

```

```

lemma funpow_fixpoint:  $f\ x = x \implies (f \hat{\ } n)\ x = x$ 
by (induct n) auto

```

```

lemma rtranclp_imp_eq_image:  $(\forall x\ y. R\ x\ y \implies f\ x = f\ y) \implies R^{**}\ x\ y \implies f\ x = f\ y$ 
by (erule rtranclp.induct) auto

```


lemma *tranclp_imp_eq_image*: $(\forall x y. R x y \longrightarrow f x = f y) \Longrightarrow R^{++} x y \Longrightarrow f x = f y$
by (*erule tranclp.induct*) *auto*

3.2 Prover

type-synonym *'a lclause* = *'a literal list*
type-synonym *'a dclause* = *'a lclause* \times *nat*
type-synonym *'a dstate* = *'a dclause list* \times *'a dclause list* \times *'a dclause list* \times *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* \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* **and**
size_atm :: *'a* \Rightarrow *nat* **and**
timestamp_factor :: *nat* **and**
size_factor :: *nat* +
assumes
S_empty: *S C* = $\{\#\}$
begin

lemma *less_atm_irrefl*: \neg *less_atm A A*
using *ex_ground_subst less_atm_ground less_atm_stable unfolding is_ground_subst_def* **by** *blast*

fun *wstate_of_dstate* :: *'a dstate* \Rightarrow *'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* \Rightarrow *'a state* **where**
state_of_dstate (*N*, *P*, *Q*, $_$) =
(*set* (*map* (*mset* \circ *fst*) *N*), *set* (*map* (*mset* \circ *fst*) *P*), *set* (*map* (*mset* \circ *fst*) *Q*))

abbreviation *clss_of_dstate* :: *'a dstate* \Rightarrow *'a clause set* **where**
clss_of_dstate St \equiv *clss_of_state* (*state_of_dstate St*)

fun *is_final_dstate* :: *'a dstate* \Rightarrow *bool* **where**
is_final_dstate (*N*, *P*, *Q*, *n*) \longleftrightarrow *N* = \square \wedge *P* = \square

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

abbreviation *rtrancl_weighted_RP* (**infix** \rightsquigarrow_w^* 50) **where**
 $(\rightsquigarrow_w^*) \equiv (\rightsquigarrow_w)^{**}$

abbreviation *trancl_weighted_RP* (**infix** \rightsquigarrow_w^+ 50) **where**
 $(\rightsquigarrow_w^+) \equiv (\rightsquigarrow_w)^{++}$

definition *is_tautology* :: *'a lclause* \Rightarrow *bool* **where**
is_tautology C \longleftrightarrow $(\exists A \in \text{set } C. \text{map atm_of } C). \text{Pos } A \in \text{set } C \wedge \text{Neg } A \in \text{set } C$

definition *subsume* :: *'a lclause list* \Rightarrow *'a lclause* \Rightarrow *bool* **where**
subsume Ds C \longleftrightarrow $(\exists D \in \text{set } Ds. \text{subsumes } (\text{mset } D) (\text{mset } C))$

definition *strictly_subsume* :: *'a lclause list* \Rightarrow *'a lclause* \Rightarrow *bool* **where**
strictly_subsume Ds C \longleftrightarrow $(\exists D \in \text{set } Ds. \text{strictly_subsumes } (\text{mset } D) (\text{mset } C))$

definition *is_reducible_on* :: *'a literal* \Rightarrow *'a lclause* \Rightarrow *'a literal* \Rightarrow *'a lclause* \Rightarrow *bool* **where**
is_reducible_on M D L C \longleftrightarrow *subsumes* (*mset D* + $\{\#\ - M\#\}$) (*mset C* + $\{\#L\#\}$)

definition *is_reducible_lit* :: 'a lclause list \Rightarrow 'a lclause \Rightarrow 'a literal \Rightarrow bool **where**
is_reducible_lit Ds C L \longleftrightarrow
 $(\exists D \in \text{set } Ds. \exists L' \in \text{set } D. \exists \sigma. -L = L' \cdot l \sigma \wedge \text{mset } (\text{remove1 } L' D) \cdot \sigma \subseteq \# \text{mset } C)$

primrec *reduce* :: 'a lclause list \Rightarrow 'a lclause \Rightarrow 'a lclause \Rightarrow 'a lclause **where**
reduce _ _ [] = []
| *reduce* Ds C (L # C') =
 (if *is_reducible_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 (*induction* ys *arbitrary*: xs)

case (Cons y ys)

show ?case

proof (*cases* y \in set xs)

case y_in: True

then have subs: $\text{mset } ys \subseteq \# \text{mset } (\text{remove1 } y \ xs)$

using Cons(2) **by** (*simp* add: *insert_subset_eq_iff*)

show ?thesis

using y_in Cons subs **by** auto

next

case False

then show ?thesis

using Cons **by** auto

qed

qed auto

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 $resolve :: 'a\ lclause \Rightarrow 'a\ lclause \Rightarrow 'a\ lclause\ list\ \mathbf{where}$

```
resolve C D =
  concat (map (λL.
    (case L of
      Pos A ⇒ []
    | Neg A ⇒
      if maximal_wrt A (mset D) then
        resolve_on A (remove1 L D) C
      else
        [])) D)
```

definition $resolve_rename :: 'a\ lclause \Rightarrow 'a\ lclause \Rightarrow 'a\ lclause\ list\ \mathbf{where}$

```
resolve_rename C D =
  (let σs = renamings_apart [mset D, mset C] in
    resolve (map (λL. L ·l last σs) C) (map (λL. L ·l hd σs) D))
```

definition $resolve_rename_either_way :: 'a\ lclause \Rightarrow 'a\ lclause \Rightarrow 'a\ lclause\ list\ \mathbf{where}$

```
resolve_rename_either_way C D = resolve_rename C D @ resolve_rename D C
```

fun $select_min_weight_clause :: 'a\ dclause \Rightarrow 'a\ dclause\ list \Rightarrow 'a\ dclause\ \mathbf{where}$

```
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 $select_min_weight_clause_in: select_min_weight_clause\ P0\ P \in set\ (P0\ \# P)$

by (induct P arbitrary: P0) auto

function $remdups_clss :: 'a\ dclause\ list \Rightarrow 'a\ dclause\ list\ \mathbf{where}$

```
remdups_clss [] = []
| remdups_clss (Ci # Cis) =
  (let
    Ci' = select_min_weight_clause Ci Cis
  in
    Ci' # remdups_clss (filter (λ(D, _). mset D ≠ mset (fst Ci')) (Ci # Cis)))
by pat_completeness auto
termination
  apply (relation measure length)
  apply (rule wf_measure)
  by (metis (mono_tags) in_measure length_filter_less prod.case_eq_if select_min_weight_clause_in)
```

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

fun $deterministic_RP_step :: 'a\ dstate \Rightarrow 'a\ dstate\ \mathbf{where}$

```
deterministic_RP_step (N, P, Q, n) =
  (if ∃ Ci ∈ set (P @ Q). fst Ci = [] then
    ([], [], remdups_clss P @ Q, n + length (remdups_clss P))
  else
    (case N of
      [] ⇒
      (case P of
        [] ⇒ (N, P, Q, n)
      | P0 # P' ⇒
        let
          (C, i) = select_min_weight_clause P0 P';
          N = map (λD. (D, n)) (remdups_gen mset (resolve_rename C C
            @ concat (map (resolve_rename_either_way C ∘ fst) Q)));
          P = filter (λ(D, j). mset D ≠ mset C) P;
          Q = (C, i) # Q;
          n = Suc n
        in
```

```

      (N, P, Q, n)
| (C, i) # N =>
  let
    C = reduce (map fst (P @ Q)) [] C
  in
    if C = [] then
      ([], [], [([]), i], Suc n)
    else if is_tautology C ∨ 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 ∘ strictly_subsume [C] ∘ fst) Q;
        P = filter (Not ∘ strictly_subsume [C] ∘ 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 => '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 ==> ¬ *wstate_of_dstate* St ~_w St'
using *wrp.final_weighted_RP*
by (cases St) (auto intro: *wrp.final_weighted_RP* simp: *is_final_dstate.simps*)

lemma *is_final_dstate_funpow_imp_deterministic_RP_neq_None*:
is_final_dstate ((*deterministic_RP_step* ^ k) St) ==> *deterministic_RP* St ≠ None
proof (induct k arbitrary: St)
case (Suc k)
note *ih* = *this*(1) **and** *final_Sk* = *this*(2)[*simplified*, *unfolded_funpow_swap1*]
show ?case
using *ih*[OF *final_Sk*] **by** (subst *deterministic_RP.simps*) (simp add: *prod.case_eq_if*)
qed (subst *deterministic_RP.simps*, simp add: *prod.case_eq_if*)

lemma *is_reducible_lit_mono_cls*:
mset C ⊆# *mset* C' ==> *is_reducible_lit* Ds C L ==> *is_reducible_lit* Ds C' L
unfolding *is_reducible_lit_def* **by** (blast intro: *subset_mset.trans*)

lemma *is_reducible_lit_mset_iff*:
mset C = *mset* C' ==> *is_reducible_lit* Ds C' L ↔ *is_reducible_lit* Ds C L
by (metis *is_reducible_lit_mono_cls* *subset_mset.order_refl*)

lemma *is_reducible_lit_remove1_Cons_iff*:
assumes L ∈ set C'
shows *is_reducible_lit* Ds (C @ remove1 L (M # C')) L ↔
is_reducible_lit Ds (M # C @ remove1 L C') L
using *assms* **by** (subst *is_reducible_lit_mset_iff*, auto)

lemma *reduce_mset_eq*: *mset* C = *mset* C' ==> *reduce* Ds C E = *reduce* Ds C' E
proof (induct E arbitrary: C C')
case (Cons L E)
note *ih* = *this*(1) **and** *mset_eq* = *this*(2)
have
mset_lc_eq: *mset* (L # C) = *mset* (L # C') **and**
mset_ce_eq: *mset* (C @ E) = *mset* (C' @ E)

using *mset_eq* **by** *simp+*
show *?case*
using *ih[OF mset_eq] ih[OF mset_lc_eq]* **by** (*simp add: is_reducible_lit_mset_iff[OF mset_ce_eq]*)
qed *simp*

lemma *reduce_rotate[simp]*: $\text{reduce } Ds (C @ [L]) E = \text{reduce } Ds (L \# C) E$
by (*rule reduce_mset_eq simp*)

lemma *mset_reduce_subset*: $\text{mset } (\text{reduce } Ds C E) \subseteq \# \text{mset } E$
by (*induct E arbitrary: C*) (*auto intro: subset_mset_imp_subset_add_mset*)

lemma *reduce_idem*: $\text{reduce } Ds C (\text{reduce } Ds C E) = \text{reduce } Ds C E$
by (*induct E arbitrary: C*)
(auto intro!: mset_reduce_subset
dest!: is_reducible_lit_mono_cls[of C @ reduce Ds (L \# C) E C @ E Ds L for L E C,
rotated])

lemma *is_reducible_lit_imp_is_reducible*:
 $L \in \text{set } C' \implies \text{is_reducible_lit } Ds (C @ \text{remove1 } L C') L \implies \text{reduce } Ds C C' \neq C'$

proof (*induct C' arbitrary: C*)
case (*Cons M C'*)
note *ih = this(1) and l_in = this(2) and l_red = this(3)*

show *?case*

proof (*cases is_reducible_lit Ds (C @ C') M*)

case *True*

then show *?thesis*

by *simp (metis mset.simps(2) mset_reduce_subset multi_self_add_other_not_self subset_mset.eq_iff subset_mset_imp_subset_add_mset)*

next

case *m_irred: False*

have

$L \in \text{set } C'$ **and**

$\text{is_reducible_lit } Ds (M \# C @ \text{remove1 } L C') L$

using *l_in l_red m_irred is_reducible_lit_remove1_Cons_iff* **by** *auto*

then show *?thesis*

by (*simp add: ih[of M \# C] m_irred*)

qed

qed *simp*

lemma *is_reducible_imp_is_reducible_lit*:
 $\text{reduce } Ds C C' \neq C' \implies \exists L \in \text{set } C'. \text{is_reducible_lit } Ds (C @ \text{remove1 } L C') L$

proof (*induct C' arbitrary: C*)
case (*Cons M C'*)
note *ih = this(1) and mc'_red = this(2)*

show *?case*

proof (*cases is_reducible_lit Ds (C @ C') M*)

case *m_irred: False*

show *?thesis*

using *ih[of M \# C] mc'_red[simplified, simplified m_irred, simplified] m_irred is_reducible_lit_remove1_Cons_iff*

by *auto*

qed *simp*

qed *simp*

lemma *is_irreducible_iff_nexists_is_reducible_lit*:
 $\text{reduce } Ds C C' = C' \iff \neg (\exists L \in \text{set } C'. \text{is_reducible_lit } Ds (C @ \text{remove1 } L C') L)$
using *is_reducible_imp_is_reducible_lit is_reducible_lit_imp_is_reducible* **by** *blast*

lemma *is_irreducible_mset_iff*: $\text{mset } E = \text{mset } E' \implies \text{reduce } Ds C E = E \iff \text{reduce } Ds C E' = E'$
unfolding *is_irreducible_iff_nexists_is_reducible_lit*
by (*metis (full_types) is_reducible_lit_mset_iff mset_remove1 set_mset_mset union_code*)

```

lemma select_min_weight_clause_min_weight:
  assumes  $Ci = \text{select\_min\_weight\_clause } P0\ P$ 
  shows  $\text{weight } (\text{apfst } \text{mset } Ci) = \text{Min } ((\text{weight } \circ \text{apfst } \text{mset}) \text{ ` set } (P0 \# P))$ 
  using assms
proof (induct P arbitrary: P0 Ci)
  case (Cons P1 P)
  note  $ih = \text{this}(1)$  and  $ci = \text{this}(2)$ 

  show ?case
  proof (cases weight (apfst mset P1) < weight (apfst mset P0))
  case True
  then have  $\text{min: } \text{Min } ((\text{weight } \circ \text{apfst } \text{mset}) \text{ ` set } (P0 \# P1 \# P)) =$ 
     $\text{Min } ((\text{weight } \circ \text{apfst } \text{mset}) \text{ ` set } (P1 \# P))$ 
  by (simp add: min_def)
  show ?thesis
  unfolding min by (rule ih[of Ci P1]) (simp add: ih[of Ci P1] ci True)
next
  case False
  have  $\text{Min } ((\text{weight } \circ \text{apfst } \text{mset}) \text{ ` set } (P0 \# P1 \# P)) =$ 
     $\text{Min } ((\text{weight } \circ \text{apfst } \text{mset}) \text{ ` set } (P1 \# P0 \# P))$ 
  by (rule arg_cong[of _ _ Min]) auto
  then have  $\text{min: } \text{Min } ((\text{weight } \circ \text{apfst } \text{mset}) \text{ ` set } (P0 \# P1 \# P)) =$ 
     $\text{Min } ((\text{weight } \circ \text{apfst } \text{mset}) \text{ ` set } (P0 \# P))$ 
  by (simp add: min_def) (use False eq_iff in fastforce)
  show ?thesis
  unfolding min by (rule ih[of Ci P0]) (simp add: ih[of Ci P1] ci False)
qed
qed simp

lemma remdups_cls_Nil_iff:  $\text{remdups\_cls } Cs = [] \longleftrightarrow Cs = []$ 
  by (cases Cs, simp, hypsubst, subst remdups_cls_simps(2), simp add: Let_def)

lemma empty_N_if_Nil_in_P_or_Q:
  assumes  $\text{nil\_in: } [] \in \text{fst ` set } (P @ Q)$ 
  shows  $\text{wstate\_of\_dstate } (N, P, Q, n) \rightsquigarrow_w^* \text{wstate\_of\_dstate } ([], P, Q, n)$ 
proof (induct N)
  case ih: (Cons N0 N)
  have  $\text{wstate\_of\_dstate } (N0 \# N, P, Q, n) \rightsquigarrow_w \text{wstate\_of\_dstate } (N, P, Q, n)$ 
  by (rule arg_cong2[THEN iffD1, of _ _ _ _ (rightsquigarrow_w), OF _ _
    urp.forward_subsumption[of {#} mset (map (apfst mset) P) mset (map (apfst mset) Q)
    mset (fst N0) mset (map (apfst mset) N) snd N0 n]])
    (use nil_in in force simp: image_def apfst_fst_snd)+
  then show ?case
  using ih by (rule converse_rtranclp_into_rtranclp)
qed simp

lemma remove_strictly_subsumed_clauses_in_P:
  assumes
   $c\_in: C \in \text{fst ` set } N$  and
   $p\_nsubs: \forall D \in \text{fst ` set } P. \neg \text{strictly\_subsume } [C] D$ 
  shows  $\text{wstate\_of\_dstate } (N, P @ P', Q, n)$ 
     $\rightsquigarrow_w^* \text{wstate\_of\_dstate } (N, P @ \text{filter } (\text{Not } \circ \text{strictly\_subsume } [C] \circ \text{fst}) P', Q, n)$ 
  using p_nsubs
proof (induct length P' arbitrary: P P' rule: less_induct)
  case less
  note  $ih = \text{this}(1)$  and  $p\_nsubs = \text{this}(2)$ 

  show ?case
  proof (cases length P')
  case Suc

  let  $?Dj = \text{hd } P'$ 

```

```

let ?P'' = tl P'
have p': P' = hd P' # tl P'
  using Suc by (metis length_Suc_conv list.distinct(1) list.exhaust_sel)

show ?thesis
proof (cases strictly_subsume [C] (fst ?Dj))
  case subs: True

  have p_filtered: {#(E, k) ∈# image_mset (apfst mset) (mset P). E ≠ mset (fst ?Dj)#} =
    image_mset (apfst mset) (mset P)
  by (rule filter_mset_cong[OF refl, of _ _ λ_. True, simplified],
      use subs p_nsubs in ‹auto simp: strictly_subsume_def›)
  have {#(E, k) ∈# image_mset (apfst mset) (mset P'). E ≠ mset (fst ?Dj)#} =
    {#(E, k) ∈# image_mset (apfst mset) (mset ?P''). E ≠ mset (fst ?Dj)#}
  by (subst (2) p') (simp add: case_prod_beta)
  also have ... =
    image_mset (apfst mset) (mset (filter (λ(E, l). mset E ≠ mset (fst ?Dj)) ?P''))
  by (auto simp: image_mset_filter_swap[symmetric] case_prod_beta)
  finally have p'_filtered:
    {#(E, k) ∈# image_mset (apfst mset) (mset P'). E ≠ mset (fst ?Dj)#} =
    image_mset (apfst mset) (mset (filter (λ(E, l). mset E ≠ mset (fst ?Dj)) ?P''))
  .

  have wstate_of_dstate (N, P @ P', Q, n)
    ~w wstate_of_dstate (N, P @ filter (λ(E, l). mset E ≠ mset (fst ?Dj)) ?P'', Q, n)
  by (rule arg_cong2[THEN iffD1, of _ _ _ (λx. ~w), OF _ _
      wrp.backward_subsumption_P[of mset C mset (map (apfst mset) N) mset (fst ?Dj)
      mset (map (apfst mset) (P @ P')) mset (map (apfst mset) Q) n]],
      use c_in_subs in ‹auto simp add: p_filtered p'_filtered arg_cong[OF p', of set]
      strictly_subsume_def›)
  also have ...
    ~w* wstate_of_dstate (N, P @ filter (Not ∘ strictly_subsume [C] ∘ fst) P', Q, n)
  apply (rule arg_cong2[THEN iffD1, of _ _ _ (λx. ~w*), OF _ _
      ih[of filter (λ(E, l). mset E ≠ mset (fst ?Dj)) ?P'' P]])
  apply simp_all
  apply (subst (3) p')
  using subs
  apply (simp add: case_prod_beta)
  apply (rule arg_cong[of _ _ λx. image_mset (apfst mset) x])
  apply (metis (no_types, opaque_lifting) strictly_subsume_def)
  apply (subst (3) p')
  apply (subst list.size)
  apply (metis (no_types, lifting) less_Suc0 less_add_same_cancel1 linorder_neqE_nat
      not_add_less1 sum_length_filter_compl trans_less_add1)
  using p_nsubs by fast
  ultimately show ?thesis
  by (rule converse_rtranclp_into_rtranclp)
next
case nsubs: False
show ?thesis
  apply (rule arg_cong2[THEN iffD1, of _ _ _ (λx. ~w*), OF _ _
      ih[of ?P'' P @ [?Dj]]])
  using nsubs p_nsubs
  apply (simp_all add: arg_cong[OF p', of mset] arg_cong[OF p', of filter f for f])
  apply (subst (1 2) p')
  by simp
qed
qed simp
qed

```

```

lemma remove_strictly_subsumed_clauses_in_Q:
  assumes c_in: C ∈ fst ' set N
  shows wstate_of_dstate (N, P, Q @ Q', n)

```



```

shows wstate_of_dstate (N, P, Q @ (D @ D', j) # Q', n)
  ~>w* wstate_of_dstate (N, (D @ reduce [C] D D', j) # P, Q @ Q', n)
using d'_red
proof (induct D' arbitrary: D)
  case (Cons L D')
  note ih = this(1) and ld'_red = this(2)
  then show ?case
  proof (cases is_reducible_lit [C] (D @ D') L)
    case l_red: True
    then obtain L' :: 'a literal and  $\sigma$  :: 's where
      l'_in: L' ∈ set C and
      not_l: - L = L' · l  $\sigma$  and
      subs: mset (remove1 L' C) ·  $\sigma$  ⊆# mset (D @ D')
    unfolding is_reducible_lit_def by force

    have wstate_of_dstate (N, P, Q @ (D @ L # D', j) # Q', n)
      ~>w wstate_of_dstate (N, (D @ D', j) # P, Q @ Q', n)
    by (rule arg_cong2[THEN iffD1, of _ _ _ _ (~~w), OF _ _
      wrp.backward_reduction_Q[of mset C - {#L'#} L' mset (map (apfst mset) N) L  $\sigma$ 
      mset (D @ D') mset (map (apfst mset) P) mset (map (apfst mset) (Q @ Q')) j n]],
      use l'_in not_l subs c_in in auto)
    then show ?thesis
      using l_red p_irred reduce_clause_in_P[OF c_in, of [] P j D D' Q @ Q' n] by simp
  next
  case l_nred: False
  then have d'_red: reduce [C] (D @ [L]) D' ≠ D'
    using ld'_red by simp
  show ?thesis
    using ih[OF d'_red] l_nred by simp
  qed
qed simp

lemma reduce_clauses_in_P:
  assumes
    c_in: C ∈ fst ' set N and
    p_irred: ∀(E, k) ∈ set P. is_irreducible [C] E
  shows wstate_of_dstate (N, P @ P', Q, n) ~>w* wstate_of_dstate (N, P @ reduce_all C P', Q, n)
  unfolding reduce_all_def
  using p_irred
proof (induct length P' arbitrary: P P')
  case (Suc l)
  note ih = this(1) and suc_l = this(2) and p_irred = this(3)

  have p'_nnil: P' ≠ []
    using suc_l by auto

  define j :: nat where
    j = Max (snd ' set P')

  obtain Dj :: 'a dclause where
    dj_in: Dj ∈ set P' and
    snd_dj: snd Dj = j
    using Max_in[of snd ' set P', unfolded image_def, simplified]
    by (metis image_def j_def length_Suc_conv list.set_intros(1) suc_l)

  have ∀k ∈ snd ' set P'. k ≤ j
    unfolding j_def using p'_nnil by simp
  then have j_max: ∀(E, k) ∈ set P'. j ≥ k
    unfolding image_def by fastforce

  obtain P1' P2' :: 'a dclause list where
    p': P' = P1' @ Dj # P2'
    using split_list[OF dj_in] by blast

```

```

have wstate_of_dstate (N, P @ P1' @ Dj # P2', Q, n)
  ~w* wstate_of_dstate (N, P @ P1' @ apfst (reduce [C] []) Dj # P2', Q, n)
unfolding append_assoc[symmetric]
apply (subst (1 2) surjective_pairing[of Dj, unfolded snd_dj])
apply (simp only: apfst_conv)
apply (rule reduce_clause_in_P[of _ _ _ _ [], unfolded append_Nil, OF c_in])
using p_irred j_max[unfolded p'] by (force simp: case_prod_beta)
moreover have wstate_of_dstate (N, P @ P1' @ apfst (reduce [C] []) Dj # P2', Q, n)
  ~w* wstate_of_dstate (N, P @ map (apfst (reduce [C] [])) (P1' @ Dj # P2'), Q, n)
apply (rule arg_cong2[THEN iffD1, of _ _ _ _ (~w*), OF _ _
  ih[of P1' @ P2' apfst (reduce [C] []) Dj # P]])
using suc_l reduce_idem p_irred unfolding p' by (auto simp: case_prod_beta)
ultimately show ?case
unfolding p' by simp
qed simp

```

lemma reduce_clauses_in_Q:

```

assumes
  c_in: C ∈ fst ' set N and
  p_irred: ∀ (E, k) ∈ set P. is_irreducible [C] E
shows wstate_of_dstate (N, P, Q @ Q', n)
  ~w* wstate_of_dstate (N, fst (reduce_all2 C Q') @ P, Q @ snd (reduce_all2 C Q'), n)
using p_irred
proof (induct Q' arbitrary: P Q)
case (Cons Dj Q')
note ih = this(1) and p_irred = this(2)
show ?case
proof (cases is_irreducible [C] (fst Dj))
case True
then show ?thesis
using ih[of _ Q @ [Dj]] p_irred by (simp add: case_prod_beta)
next
case d_red: False
have wstate_of_dstate (N, P, Q @ Dj # Q', n)
  ~w* wstate_of_dstate (N, (reduce [C] [] (fst Dj), snd Dj) # P, Q @ Q', n)
using p_irred reduce_clause_in_Q[of _ _ P snd Dj [] _ Q Q' n, OF c_in _ d_red]
by (cases Dj) force
then show ?thesis
using ih[of (reduce [C] [] (fst Dj), snd Dj) # P Q] d_red p_irred reduce_idem
by (force simp: case_prod_beta)
qed
qed simp

```

lemma eligible_iff:

```

eligible S σ As DA ↔ As = [] ∨ length As = 1 ∧ maximal_wrt (hd As · a σ) (DA · σ)
unfolding eligible.simps S_empty by (fastforce dest: hd_conv_nth)

```

lemma ord_resolve_one_side_prem:

```

ord_resolve S CAs DA AAs As σ E ⇒ length CAs = 1 ∧ length AAs = 1 ∧ length As = 1
by (force elim!: ord_resolve.cases simp: eligible_iff)

```

lemma ord_resolve_rename_one_side_prem:

```

ord_resolve_rename S CAs DA AAs As σ E ⇒ length CAs = 1 ∧ length AAs = 1 ∧ length As = 1
by (force elim!: ord_resolve_rename.cases dest: ord_resolve_one_side_prem)

```

abbreviation Bin_ord_resolve :: 'a clause ⇒ 'a clause ⇒ 'a clause set **where**

```

Bin_ord_resolve C D ≡ {E. ∃ AA A σ. ord_resolve S [C] D [AA] [A] σ E}

```

abbreviation Bin_ord_resolve_rename :: 'a clause ⇒ 'a clause ⇒ 'a clause set **where**

```

Bin_ord_resolve_rename C D ≡ {E. ∃ AA A σ. ord_resolve_rename S [C] D [AA] [A] σ E}

```

lemma resolve_on_eq_UNION_Bin_ord_resolve:

```

mset ' set (resolve_on A D CA) =
{E. ∃ AA σ. ord_resolve S [mset CA] ({#Neg A#} + mset D) [AA] [A] σ E}
proof
{
  fix E :: 'a literal list
  assume E ∈ set (resolve_on A D CA)
  then have E ∈ resolvent D A CA ' {Ls. subseq Ls CA ∧ resolvable A D CA Ls}
    unfolding resolve_on_def by simp
  then obtain Ls where Ls_p: resolvent D A CA Ls = E subseq Ls CA ∧ resolvable A D CA Ls
    by auto
  define σ where σ = the (mgu {insert A (atms_of (mset Ls))})
  then have σ_p:
    mgu {insert A (atms_of (mset Ls))} = Some σ
    Ls ≠ []
    eligible S σ [A] (add_mset (Neg A) (mset D))
    strictly_maximal_wrt (A · a σ) ((mset CA - mset Ls) · σ)
    ∀ L ∈ set Ls. is_pos L
    using Ls_p unfolding resolvable_def unfolding Let_def eligible.simps using S_empty by auto
  from σ_p have σ_p2: the (mgu {insert A (atms_of (mset Ls))}) = σ
    by auto
  have Ls_sub_CA: mset Ls ⊆# mset CA
    using subseq_mset_subseteq_mset Ls_p by auto
  then have mset (resolvent D A CA Ls) = sum_list [mset CA - mset Ls] · σ + mset D · σ
    unfolding resolvent_def σ_p2 subst_cls_def using remove_all_mset_minus[of Ls CA] by auto
  moreover
  have length [mset CA - mset Ls] = Suc 0
    by auto
  moreover
  have ∀ L ∈ set Ls. is_pos L
    using σ_p(5) list_all_iff[of is_pos] by auto
  then have {#Pos (atm_of x). x ∈# mset Ls#} = mset Ls
    by (induction Ls) auto
  then have mset CA = [mset CA - mset Ls] ! 0 + {#Pos (atm_of x). x ∈# mset Ls#}
    using Ls_sub_CA by auto
  moreover
  have Ls ≠ []
    using σ_p by -
  moreover
  have Some σ = mgu {insert A (atm_of ' set Ls)}
    using σ_p unfolding atms_of_def by auto
  moreover
  have eligible S σ [A] (add_mset (Neg A) (mset D))
    using σ_p by -
  moreover
  have strictly_maximal_wrt (A · a σ) ([mset CA - mset Ls] ! 0 · σ)
    using σ_p(4) by auto
  moreover have S (mset CA) = {#}
    by (simp add: S_empty)
  ultimately have ∃ Cs. mset (resolvent D A CA Ls) = sum_list Cs · σ + mset D · σ
    ∧ length Cs = Suc 0 ∧ mset CA = Cs ! 0 + {#Pos (atm_of x). x ∈# mset Ls#}
    ∧ Ls ≠ [] ∧ Some σ = mgu {insert A (atm_of ' set Ls)}
    ∧ eligible S σ [A] (add_mset (Neg A) (mset D)) ∧ strictly_maximal_wrt (A · a σ) (Cs ! 0 · σ)
    ∧ S (mset CA) = {#}
    by blast
  then have ord_resolve S [mset CA] (add_mset (Neg A) (mset D)) [image_mset atm_of (mset Ls)] [A]
    σ (mset (resolvent D A CA Ls))
    unfolding ord_resolve.simps by auto
  then have ∃ AA σ. ord_resolve S [mset CA] (add_mset (Neg A) (mset D)) [AA] [A] σ (mset E)
    using Ls_p by auto
}
then show mset ' set (resolve_on A D CA)
  ⊆ {E. ∃ AA σ. ord_resolve S [mset CA] ({#Neg A#} + mset D) [AA] [A] σ E}
  by auto

```

```

next
{
  fix E AA  $\sigma$ 
  assume ord_resolve S [mset CA] (add_mset (Neg A) (mset D)) [AA] [A]  $\sigma$  E
  then obtain Cs where res': E = sum_list Cs ·  $\sigma$  + mset D ·  $\sigma$ 
    length Cs = Suc 0
    mset CA = Cs ! 0 + poss AA
    AA  $\neq$  {#}
    Some  $\sigma$  = mgu {insert A (set_mset AA)}
    eligible S  $\sigma$  [A] (add_mset (Neg A) (mset D))
    strictly_maximal_wrt (A · a  $\sigma$ ) (Cs ! 0 ·  $\sigma$ )
    S (Cs ! 0 + poss AA) = {#}
    unfolding ord_resolve.simps by auto
  moreover define C where C = Cs ! 0
  ultimately have res:
    E = sum_list Cs ·  $\sigma$  + mset D ·  $\sigma$ 
    mset CA = C + poss AA
    AA  $\neq$  {#}
    Some  $\sigma$  = mgu {insert A (set_mset AA)}
    eligible S  $\sigma$  [A] (add_mset (Neg A) (mset D))
    strictly_maximal_wrt (A · a  $\sigma$ ) (C ·  $\sigma$ )
    S (C + poss AA) = {#}
    unfolding ord_resolve.simps by auto
  from this(1) have
    E = C ·  $\sigma$  + mset D ·  $\sigma$ 
    unfolding C_def using res'(2) by (cases Cs) auto
  note res' = this res(2-7)
  have  $\exists$  Al. mset Al = AA  $\wedge$  subseq (map Pos Al) CA
    using res(2)
  proof (induction CA arbitrary: AA C)
    case Nil
    then show ?case by auto
  next
  case (Cons L CA)
  then show ?case
  proof (cases L  $\in$  # poss AA )
    case True
    then have pos_L: is_pos L
      by auto
    have rem:  $\bigwedge$  A'. Pos A'  $\in$  # poss AA  $\implies$ 
      remove1_mset (Pos A') (C + poss AA) = C + poss (remove1_mset A' AA)
      by (induct AA) auto
    have mset CA = C + (poss (AA - {#atm_of L#}))
      using True Cons(2)
    by (metis add_mset_remove_trivial rem literal.collapse(1) mset.simps(2) pos_L)
    then have  $\exists$  Al. mset Al = remove1_mset (atm_of L) AA  $\wedge$  subseq (map Pos Al) CA
      using Cons(1)[of _ ((AA - {#atm_of L#}))] by metis
    then obtain Al where
      mset Al = remove1_mset (atm_of L) AA  $\wedge$  subseq (map Pos Al) CA
      by auto
    then have
      mset (atm_of L # Al) = AA and
      subseq (map Pos (atm_of L # Al)) (L # CA)
      using True by (auto simp add: pos_L)
    then show ?thesis
      by blast
  next
  case False
  then have mset CA = remove1_mset L C + poss AA
    using Cons(2)
  by (metis Un_iff add_mset_remove_trivial mset.simps(2) set_mset_union single_subset_iff
    subset_mset.add_diff_assoc2 union_single_eq_member)
  then have  $\exists$  Al. mset Al = AA  $\wedge$  subseq (map Pos Al) CA

```

```

    using Cons(1)[of C - {#L#} AA] Cons(2) by auto
  then show ?thesis
    by auto
  qed
qed
then obtain Al where Al_p: mset Al = AA subseq (map Pos Al) CA
  by auto

define Ls :: 'a lclause where Ls = map Pos Al
have diff: mset CA - mset Ls = C
  unfolding Ls_def using res(2) Al_p(1) by auto
have ls_subq_ca: subseq Ls CA
  unfolding Ls_def using Al_p by -
moreover
{
  have  $\exists y. \text{mgu } \{ \text{insert } A \text{ (atms\_of (mset Ls))} \} = \text{Some } y$ 
    unfolding Ls_def using res(4) Al_p by (metis atms_of_poss mset_map)
  moreover have  $Ls \neq []$ 
    using Al_p(1) Ls_def res'(3) by auto
  moreover have  $\sigma_p: \text{the (mgu } \{ \text{insert } A \text{ (set Al)} \}) = \sigma$ 
    using res'(4) Al_p(1) by (metis option.sel set_mset_mset)
  then have eligible S (the (mgu {insert A (atms_of (mset Ls))})) [A]
    (add_mset (Neg A) (mset D))
    unfolding Ls_def using res by auto
  moreover have strictly_maximal_wrt (A · a the (mgu {insert A (atms_of (mset Ls))}))
    ((mset CA - mset Ls) · the (mgu {insert A (atms_of (mset Ls))}))
    unfolding Ls_def using res  $\sigma_p$  Al_p by auto
  moreover have  $\forall L \in \text{set } Ls. \text{is\_pos } L$ 
    by (simp add: Ls_def)
  ultimately have resolvable A D CA Ls
    unfolding resolvable_def unfolding eligible.simps using S_empty by simp
}
moreover have ls_sub_ca: mset Ls  $\subseteq$ # mset CA
  using ls_subq_ca subseq_mset_subseteq_mset[of Ls CA] by simp
have {#x · l  $\sigma$ . x  $\in$ # mset CA - mset Ls#} + {#M · l  $\sigma$ . M  $\in$ # mset D#} = C ·  $\sigma$  + mset D ·  $\sigma$ 
  using diff unfolding subst_cls_def by simp
then have {#x · l  $\sigma$ . x  $\in$ # mset CA - mset Ls#} + {#M · l  $\sigma$ . M  $\in$ # mset D#} = E
  using res'(1) by auto
then have {#M · l  $\sigma$ . M  $\in$ # mset (remove_all CA Ls)#} + {#M · l  $\sigma$ . M  $\in$ # mset D#} = E
  using remove_all_mset_minus[of Ls CA] ls_sub_ca by auto
then have mset (resolvent D A CA Ls) = E
  unfolding resolvable_def Let_def resolvent_def using Al_p(1) Ls_def atms_of_poss res'(4)
  by (metis image_mset_union mset_append mset_map option.sel)
ultimately have  $E \in \text{mset ' set (resolve\_on A D CA)}$ 
  unfolding resolve_on_def by auto
}
then show {E.  $\exists AA \sigma. \text{ord\_resolve } S \text{ [mset CA] } (\{ \# \text{Neg } A \# \} + \text{mset D}) \text{ [AA] [A] } \sigma \text{ E}$ 
 $\subseteq \text{mset ' set (resolve\_on A D CA)}$ 
  by auto
qed

lemma set_resolve_eq_UNION_set_resolve_on:
  set (resolve C D) =
  ( $\bigcup L \in \text{set } D.$ 
  (case L of
    Pos _  $\Rightarrow$  {}
    | Neg A  $\Rightarrow$  if maximal_wrt A (mset D) then set (resolve_on A (remove1 L D) C) else {}))
  unfolding resolve_def by (fastforce split: literal.splits if_splits)

lemma resolve_eq_Bin_ord_resolve: mset ' set (resolve C D) = Bin_ord_resolve (mset C) (mset D)
  unfolding set_resolve_eq_UNION_set_resolve_on
  apply (unfold image_UN literal.case_distrib if_distrib)
  apply (subst resolve_on_eq_UNION_Bin_ord_resolve)

```

```

apply (rule order_antisym)
apply (force split: literal.splits if_splits)
apply (clarsimp split: literal.splits if_splits)
apply (rule_tac x = Neg A in beI)
apply (rule conjI)
apply blast
apply clarify
apply (rule conjI)
apply clarify
apply (rule_tac x = AA in exI)
apply (rule_tac x =  $\sigma$  in exI)
apply (erule ord_resolve.simps[THEN iffD1])
apply force
apply (erule ord_resolve.simps[THEN iffD1])
apply (clarsimp simp: eligible_iff simp del: subst_cls_add_mset subst_cls_union)
apply (erule maximal_wrt_subst)
apply sat
apply (erule ord_resolve.simps[THEN iffD1])
using set_mset_mset by fastforce

lemma poss_in_map_clauseD:
  poss AA  $\subseteq\#$  map_clause f C  $\implies \exists AA0. \text{poss } AA0 \subseteq\# C \wedge AA = \{\#f A. A \in\# AA0\#$ 
proof (induct AA arbitrary: C)
case (add A AA)
note ih = this(1) and aaa_sub = this(2)

have Pos A  $\in\#$  map_clause f C
using aaa_sub by auto
then obtain A0 where
  pa0_in: Pos A0  $\in\#$  C and
  a: A = f A0
by clarify (metis literal.distinct(1) literal.exhaust literal.inject(1) literal.simps(9,10))

have poss AA  $\subseteq\#$  map_clause f (C -  $\{\#Pos A0\#$ )
using pa0_in aaa_sub[unfolded a] by (simp add: image_mset_remove1_mset_if insert_subset_eq_iff)
then obtain AA0 where
  paa0_sub: poss AA0  $\subseteq\#$  C -  $\{\#Pos A0\#$  and
  aa: AA = image_mset f AA0
using ih by meson

have poss (add_mset A0 AA0)  $\subseteq\#$  C
using pa0_in paa0_sub by (simp add: insert_subset_eq_iff)
moreover have add_mset A AA = image_mset f (add_mset A0 AA0)
unfolding a aa by simp
ultimately show ?case
by blast
qed simp

lemma poss_subset_filterD:
  poss AA  $\subseteq\#$   $\{\#L \cdot l \varrho. L \in\# \text{mset } C\#\} \implies \exists AA0. \text{poss } AA0 \subseteq\# \text{mset } C \wedge AA = AA0 \cdot \text{am } \varrho$ 
unfolding subst_atm_mset_def subst_lit_def by (rule poss_in_map_clauseD)

lemma neg_in_map_literalD: Neg A  $\in$  map_literal f ' D  $\implies \exists A0. \text{Neg } A0 \in D \wedge A = f A0$ 
unfolding image_def by (clarify, case_tac x, auto)

lemma neg_in_filterD: Neg A  $\in\#$   $\{\#L \cdot l \varrho'. L \in\# \text{mset } D\#\} \implies \exists A0. \text{Neg } A0 \in\# \text{mset } D \wedge A = A0 \cdot a \varrho'$ 
unfolding subst_lit_def image_def by (rule neg_in_map_literalD) simp

lemma resolve_rename_eq_Bin_ord_resolve_rename:
  mset ' set (resolve_rename C D) = Bin_ord_resolve_rename (mset C) (mset D)
proof (intro order_antisym subsetI)
let ?qs = renamings_apart [mset D, mset C]
define  $\varrho'$  :: 's where

```

```

     $\rho' = \text{hd } ?\rho s$ 
define  $\rho :: 's$  where
     $\rho = \text{last } ?\rho s$ 

have  $tl\_?\rho s: tl\ ?\rho s = [\rho]$ 
    unfolding  $\rho\_def$ 
    using  $\text{renamings\_apart\_length Nitpick.size\_list\_simp}(2)$   $\text{Suc\_length\_conv last.simps}$ 
    by  $(\text{smt length\_greater\_0\_conv list.sel}(3))$ 

{
fix  $E$ 
assume  $e\_in: E \in \text{mset ' set (resolve\_rename } C\ D)$ 

from  $e\_in$  obtain  $AA :: 'a$  multiset and  $A :: 'a$  and  $\sigma :: 's$  where
     $aa\_sub: \text{poss } AA \subseteq\# \text{mset } C \cdot \rho$  and
     $a\_in: \text{Neg } A \in\# \text{mset } D \cdot \rho'$  and
     $\text{res\_e: ord\_resolve } S [\text{mset } C \cdot \rho] \{\#L \cdot l\ \rho'. L \in\# \text{mset } D\# \} [AA] [A] \sigma\ E$ 
    unfolding  $\rho'\_def\ \rho\_def$ 
    apply  $\text{atomize\_elim}$ 
    using  $e\_in$  unfolding  $\text{resolve\_rename\_def Let\_def resolve\_eq Bin\_ord\_resolve}$ 
    apply  $\text{clarsimp}$ 
    apply  $(\text{frule ord\_resolve\_one\_side\_prem})$ 
    apply  $(\text{frule ord\_resolve.simps}[THEN\ \text{iff}D1])$ 
    apply  $(\text{rule\_tac } x = AA\ \text{in } exI)$ 
    apply  $(\text{clarsimp simp: subst\_cls\_def})$ 
    apply  $(\text{rule\_tac } x = A\ \text{in } exI)$ 
    by  $(\text{metis (full\_types) Melem\_subst\_cls set\_mset\_mset subst\_cls\_def union\_single\_eq\_member})$ 

obtain  $AA0 :: 'a$  multiset where
     $aa0\_sub: \text{poss } AA0 \subseteq\# \text{mset } C$  and
     $aa: AA = AA0 \cdot am\ \rho$ 
    using  $aa\_sub$ 
    apply  $\text{atomize\_elim}$ 
    apply  $(\text{rule ord\_resolve.cases}[OF\ \text{res\_e}])$ 
    by  $(\text{rule poss\_subset\_filterD}[OF\ aa\_sub[\text{unfolded } \text{subst\_cls\_def}]])$ 

obtain  $A0 :: 'a$  where
     $a0\_in: \text{Neg } A0 \in \text{set } D$  and
     $a: A = A0 \cdot a\ \rho'$ 
    apply  $\text{atomize\_elim}$ 
    apply  $(\text{rule ord\_resolve.cases}[OF\ \text{res\_e}])$ 
    using  $\text{neg\_in\_filterD}[OF\ a\_in[\text{unfolded } \text{subst\_cls\_def}]]$  by  $\text{simp}$ 

show  $E \in \text{Bin\_ord\_resolve\_rename (mset } C) (mset\ D)$ 
    unfolding  $\text{ord\_resolve\_rename.simps}$ 
    using  $\text{res\_e}$ 
    apply  $\text{clarsimp}$ 
    apply  $(\text{rule\_tac } x = AA0\ \text{in } exI)$ 
    apply  $(\text{intro conj}I)$ 
    apply  $(\text{rule } aa0\_sub)$ 
    apply  $(\text{rule\_tac } x = A0\ \text{in } exI)$ 
    apply  $(\text{intro conj}I)$ 
    apply  $(\text{rule } a0\_in)$ 
    apply  $(\text{rule\_tac } x = \sigma\ \text{in } exI)$ 
    unfolding  $aa\ a\ \rho'\_def[\text{symmetric}]\ \rho\_def[\text{symmetric}]\ tl\_?\rho s$  by  $(\text{simp add: subst\_cls\_def})$ 
}
{
fix  $E$ 
assume  $e\_in: E \in \text{Bin\_ord\_resolve\_rename (mset } C) (mset\ D)$ 
show  $E \in \text{mset ' set (resolve\_rename } C\ D)$ 
    using  $e\_in$ 
    unfolding  $\text{resolve\_rename\_def Let\_def resolve\_eq Bin\_ord\_resolve ord\_resolve\_rename.simps}$ 
    apply  $\text{clarsimp}$ 

```

```

apply (rule_tac x = AA · am ρ in exI)
apply (rule_tac x = A · a ρ' in exI)
apply (rule_tac x = σ in exI)
unfolding tl_ρs ρ'_def ρ_def by (simp add: subst_cls_def subst_cls_lists_def)
}
qed

```

```

lemma bin_ord_FO_Γ_def:
  ord_FO_Γ S = {Infer {#CA#} DA E | CA DA AA A σ E. ord_resolve_rename S [CA] DA [AA] [A] σ E}
unfolding ord_FO_Γ_def
apply (rule order.antisym)
apply clarify
apply (frule ord_resolve_rename_one_side_prem)
apply simp
apply (metis Suc_length_conv length_0_conv)
by blast

```

```

lemma ord_FO_Γ_side_prem: γ ∈ ord_FO_Γ S ⇒ side_prem_of γ = {#THE D. D ∈# side_prem_of γ#}
unfolding bin_ord_FO_Γ_def by clarsimp

```

```

lemma ord_FO_Γ_infer_from_Collect_eq:
  {γ ∈ ord_FO_Γ S. infer_from (DD ∪ {C}) γ ∧ C ∈# prem_of γ} =
  {γ ∈ ord_FO_Γ S. ∃ D ∈ DD ∪ {C}. prem_of γ = {#C, D#}}
unfolding infer_from_def
apply (rule set_eq_subset[THEN iffD2])
apply (rule conjI)
apply clarify
apply (subst (asm) (1 2) ord_FO_Γ_side_prem, assumption, assumption)
apply (subst (1) ord_FO_Γ_side_prem, assumption)
apply force
apply clarify
apply (subst (asm) (1) ord_FO_Γ_side_prem, assumption)
apply (subst (1 2) ord_FO_Γ_side_prem, assumption)
by force

```

```

lemma inferences_between_eq_UNION: inference_system.inferences_between (ord_FO_Γ S) Q C =
  inference_system.inferences_between (ord_FO_Γ S) {C} C
  ∪ (∪ D ∈ Q. inference_system.inferences_between (ord_FO_Γ S) {D} C)
unfolding ord_FO_Γ_infer_from_Collect_eq inference_system.inferences_between_def by auto

```

```

lemma concls_of_inferences_between_singleton_eq_Bin_ord_resolve_rename:
  concls_of (inference_system.inferences_between (ord_FO_Γ S) {D} C) =
  Bin_ord_resolve_rename C C ∪ Bin_ord_resolve_rename C D ∪ Bin_ord_resolve_rename D C

```

```

proof (intro order_antisym subsetI)
fix E
assume e_in: E ∈ concls_of (inference_system.inferences_between (ord_FO_Γ S) {D} C)
then show E ∈ Bin_ord_resolve_rename C C ∪ Bin_ord_resolve_rename C D
  ∪ Bin_ord_resolve_rename D C
unfolding inference_system.inferences_between_def ord_FO_Γ_infer_from_Collect_eq
  bin_ord_FO_Γ_def infer_from_def by (fastforce simp: add_mset_eq_add_mset)
qed (force simp: inference_system.inferences_between_def infer_from_def ord_FO_Γ_def)

```

```

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)
by (subst inferences_between_eq_UNION)
  (auto simp: image_Un image_UN concls_of_inferences_between_singleton_eq_Bin_ord_resolve_rename)

```

```

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))
by (simp add: resolve_rename_either_way_def image_Un resolve_rename_eq_Bin_ord_resolve_rename
  concls_of_inferences_between_eq_Bin_ord_resolve_rename UN_Un_distrib)

```


lemma *compute_inferences*:

assumes

ci_in: $(C, i) \in \text{set } P$ **and**

ci_min: $\forall (D, j) \in \# \text{ mset } (\text{map } (\text{apfst mset}) P). \text{weight } (\text{mset } C, i) \leq \text{weight } (D, j)$

shows

wstate_of_dstate ($[], P, Q, n$) \rightsquigarrow_w

wstate_of_dstate ($\text{map } (\lambda D. (D, n)) (\text{remdups_gen mset } (\text{resolve_rename } C \ C \ @$

$\text{concat } (\text{map } (\text{resolve_rename_either_way } C \ \circ \ \text{fst}) Q))$,

$\text{filter } (\lambda(D, j). \text{mset } D \neq \text{mset } C) P, (C, i) \# Q, \text{Suc } n$)

(*is* \rightsquigarrow_w *wstate_of_dstate* ($?N, _$))

proof –

have *ms_ci_in*: $(\text{mset } C, i) \in \# \text{ image_mset } (\text{apfst mset}) (\text{mset } P)$

using *ci_in* **by** *force*

show *?thesis*

apply (*rule arg_cong2*[*THEN iffD1*, *of* $_ _ _ _ (\rightsquigarrow_w)$, *OF* $_ _$

urp.inference_computation[*of* $\text{mset } (\text{map } (\text{apfst mset}) P) - \{\#(\text{mset } C, i)\# \}$ *mset } C i*

mset } (map } (apfst mset) ?N) n mset } (map } (apfst mset) Q)]])

apply (*simp add*: *add_mset_remove_trivial_eq*[*THEN iffD2*, *OF ms_ci_in*, *symmetric*])

using *ms_ci_in*

apply (*simp add*: *ci_in image_mset_remove1_mset_if*)

apply (*smt apfst_conv case_prodE case_prodI2 case_prod_conv filter_mset_cong*

image_mset_filter_swap mset_filter)

apply (*metis ci_min in_diffD*)

apply (*simp only*: *list.map_comp apfst_comp_rpair_const*)

apply (*simp only*: *list.map_comp*[*symmetric*])

apply (*subst mset_map*)

apply (*unfold mset_map_remdups_gen mset_remdups_gen_ident*)

apply (*subst image_mset_mset_set*)

apply (*simp add*: *inj_on_def*)

apply (*subst mset_set_eq_iff*)

apply *simp*

apply (*simp add*: *finite_ord_FO_resolution_inferences_between*)

apply (*rule arg_cong*[*of* $_ _ \lambda N. (\lambda D. (D, n)) \text{' } N$])

apply (*simp only*: *map_concat list.map_comp image_comp*)

using *resolve_rename_either_way_eq_congl_of_inferences_between*[*of* $C \ \text{fst } \text{' } \text{set } Q$, *symmetric*]

by (*simp add*: *image_comp_comp_def image_UN*)

qed

lemma *nonfinal_deterministic_RP_step*:

assumes

nonfinal: $\neg \text{is_final_dstate } St$ **and**

step: $St' = \text{deterministic_RP_step } St$

shows *wstate_of_dstate* $St \rightsquigarrow_w^+ \text{wstate_of_dstate } St'$

proof –

obtain $N \ P \ Q :: \text{' } a \ \text{dclause list}$ **and** $n :: \text{nat}$ **where**

st: $St = (N, P, Q, n)$

by (*cases St*) *blast*

note *step* = *step*[*unfolded st deterministic_RP_step.simps*, *simplified*]

show *?thesis*

proof (*cases* $\exists Ci \in \text{set } P \cup \text{set } Q. \text{fst } Ci = []$)

case *nil_in*: *True*

note *step* = *step*[*simplified nil_in*, *simplified*]

have *nil_in'*: $[] \in \text{fst } \text{' } \text{set } (P \ @ \ Q)$

using *nil_in* **by** (*force simp*: *image_def*)

have *star*: $[] \in \text{fst } \text{' } \text{set } (P \ @ \ Q) \implies$

wstate_of_dstate (N, P, Q, n)

$\rightsquigarrow_w^* \text{wstate_of_dstate } ([], [], \text{remdups_class } P \ @ \ Q, n + \text{length } (\text{remdups_class } P))$

proof (*induct length (remdups_class P) arbitrary*: $N \ P \ Q \ n$)

```

case 0
note len_p = this(1) and nil_in' = this(2)

have p_nil: P = []
  using len_p remdups_clss_Nil_iff by simp
have wstate_of_dstate (N, [], Q, n)  $\rightsquigarrow_w^*$  wstate_of_dstate ([], [], Q, n)
  by (rule empty_N_if_Nil_in_P_or_Q[OF nil_in'[unfolded p_nil]])
then show ?case
  unfolding p_nil by simp
next
case (Suc k)
note ih = this(1) and suc_k = this(2) and nil_in' = this(3)

have P  $\neq$  []
  using suc_k remdups_clss_Nil_iff by force
hence p_cons: P = hd P # tl P
  by simp

obtain C :: 'a lclause and i :: nat where
  ci: (C, i) = select_min_weight_clause (hd P) (tl P)
  by (metis prod.exhaust)

have ci_in: (C, i)  $\in$  set P
  unfolding ci using p_cons select_min_weight_clause_in[of hd P tl P] by simp
have ci_min:  $\forall (D, j) \in \# \text{mset} (\text{map} (\text{apfst } \text{mset}) P). \text{weight} (\text{mset } C, i) \leq \text{weight} (D, j)$ 
  by (subst p_cons) (simp add: select_min_weight_clause_min_weight[OF ci, simplified])

let ?P' = filter ( $\lambda(D, j). \text{mset } D \neq \text{mset } C$ ) P

have ms_p'_ci_q_eq:  $\text{mset} (\text{remdups\_clss } ?P' @ (C, i) \# Q) = \text{mset} (\text{remdups\_clss } P @ Q)$ 
  apply (subst (2) p_cons)
  apply (subst remdups_clss.simps(2))
  by (auto simp: Let_def case_prod_beta p_cons[symmetric] ci[symmetric])
then have len_p:  $\text{length} (\text{remdups\_clss } P) = \text{length} (\text{remdups\_clss } ?P') + 1$ 
  by (smt Suc_eq_plus1_left add.assoc add_right_cancel length_Cons length_append
    mset_eq_length)

have wstate_of_dstate (N, P, Q, n)  $\rightsquigarrow_w^*$  wstate_of_dstate ([], P, Q, n)
  by (rule empty_N_if_Nil_in_P_or_Q[OF nil_in'])
also obtain N' :: 'a dclause list where
  ...  $\rightsquigarrow_w$  wstate_of_dstate (N', ?P', (C, i) # Q, Suc n)
  by (atomize_elim, rule exI, rule compute_inferences[OF ci_in], use ci_min in fastforce)
also have ...  $\rightsquigarrow_w^*$  wstate_of_dstate ([], [], remdups_clss P @ Q, n + length (remdups_clss P))
  apply (rule arg_cong2[THEN iffD1, of _ _ _ _ ( $\rightsquigarrow_w^*$ ), OF _ _
    ih[of ?P' (C, i) # Q N' Suc n], OF refl])
  using ms_p'_ci_q_eq suc_k nil_in' ci_in
  apply (simp_all add: len_p)
  apply (metis (no_types) apfst_conv image_mset_add_mset)
  by force
finally show ?case
  .
qed
show ?thesis
  unfolding st_step using star[OF nil_in'] nonfinal[unfolded st_is_final_dstate.simps]
  by cases simp_all
next
case nil_ni: False
note step = step[simplified nil_ni, simplified]
show ?thesis
proof (cases N)
  case n_nil: Nil
  note step = step[unfolded n_nil, simplified]
  show ?thesis

```

```

proof (cases P)
  case Nil
  then have False
    using n_nil nonfinal[unfolded st] by (simp add: is_final_dstate.simps)
  then show ?thesis
    using step by simp
next
  case p_cons: (Cons P0 P')
  note step = step[unfolded p_cons list.case, folded p_cons]

  obtain C :: 'a lclause and i :: nat where
    ci: (C, i) = select_min_weight_clause P0 P'
  by (metis prod.exhaust)
  note step = step[unfolded select, simplified]

  have ci_in: (C, i) ∈ set P
  by (rule select_min_weight_clause_in[of P0 P', folded ci p_cons])

  show ?thesis
    unfolding st n_nil step p_cons[symmetric] ci[symmetric] prod.case
    by (rule tranclp.r_into_trancl, rule compute_inferences[OF ci_in])
      (simp add: select_min_weight_clause_min_weight[OF ci, simplified] p_cons)
  qed
next
  case n_cons: (Cons Ci N')
  note step = step[unfolded n_cons, simplified]

  obtain C :: 'a lclause and i :: nat where
    ci: Ci = (C, i)
  by (cases Ci) simp
  note step = step[unfolded ci, simplified]

  define C' :: 'a lclause where
    C' = reduce (map fst P @ map fst Q) [] C
  note step = step[unfolded ci C'_def[symmetric], simplified]

  have wstate_of_dstate ((E @ C, i) # N', P, Q, n)
     $\rightsquigarrow_w^*$  wstate_of_dstate ((E @ reduce (map fst P @ map fst Q) E C, i) # N', P, Q, n) for E
  unfolding C'_def
  proof (induct C arbitrary: E)
  case (Cons L C)
  note ih = this(1)
  show ?case
  proof (cases is_reducible_lit (map fst P @ map fst Q) (E @ C) L)
  case l_red: True
  then have red_lc:
    reduce (map fst P @ map fst Q) E (L # C) = reduce (map fst P @ map fst Q) E C
  by simp
  obtain D D' :: 'a literal list and L' :: 'a literal and  $\sigma$  :: 's where
    D ∈ set (map fst P @ map fst Q) and
    D' = remove1 L' D and
    L' ∈ set D and
    - L = L' · l  $\sigma$  and
    mset D' ·  $\sigma$   $\subseteq$  # mset (E @ C)
  using l_red unfolding is_reducible_lit_def comp_def by blast
  then have  $\sigma$ :
    mset D' + {#L'#} ∈ set (map (mset ∘ fst) (P @ Q))
    - L = L' · l  $\sigma$   $\wedge$  mset D' ·  $\sigma$   $\subseteq$  # mset (E @ C)
  unfolding is_reducible_lit_def by (auto simp: comp_def)
  have wstate_of_dstate ((E @ L # C, i) # N', P, Q, n)
     $\rightsquigarrow_w$  wstate_of_dstate ((E @ C, i) # N', P, Q, n)
  by (rule arg_cong2[THEN iffD1, of _ _ _ _ ( $\rightsquigarrow_w$ ), OF _ _
    wrp.forward_reduction[of mset D' L' mset (map (apfst mset) P)

```

```

      mset (map (apfst mset) Q) L σ mset (E @ C) mset (map (apfst mset) N')
      i n]])
    (use σ in ⟨auto simp: comp_def⟩)
  then show ?thesis
    unfolding red_lc using ih[of E] by (rule converse_rtranclp_into_rtranclp)
next
  case False
  then show ?thesis
    using ih[of L # E] by simp
qed
qed simp
then have red_C:
  wstate_of_dstate ((C, i) # N', P, Q, n) ~w* wstate_of_dstate ((C', i) # N', P, Q, n)
  unfolding C'_def by (metis self_append_conv2)

have proc_C: wstate_of_dstate ((C', i) # N', P', Q', n')
  ~w wstate_of_dstate (N', (C', i) # P', Q', n') for P' Q' n'
  by (rule arg_cong2[THEN iffD1, of _ _ _ _ (~w), OF _ _
    wrp.clause_processing[of mset (map (apfst mset) N') mset C' i
      mset (map (apfst mset) P') mset (map (apfst mset) Q') n']],
    simp+)

show ?thesis
proof (cases C' = [])
  case True
  note c'_nil = this
  note step = step[simplified c'_nil, simplified]

  have
    filter_p: filter (Not ∘ strictly_subsume [] ∘ fst) P = [] and
    filter_q: filter (Not ∘ strictly_subsume [] ∘ fst) Q = []
    using nil_ni unfolding strictly_subsume_def filter_empty_conv find_None_iff by force+

  note red_C[unfolded c'_nil]
  also have wstate_of_dstate (([], i) # N', P, Q, n)
    ~w* wstate_of_dstate (([], i) # N', [], Q, n)
    by (rule arg_cong2[THEN iffD1, of _ _ _ _ (~w*), OF _ _
      remove_strictly_subsumed_clauses_in_P[of [] _ [], unfolded append_Nil],
      OF refl])
    (auto simp: filter_p)
  also have ... ~w* wstate_of_dstate (([], i) # N', [], [], n)
    by (rule arg_cong2[THEN iffD1, of _ _ _ _ (~w*), OF _ _
      remove_strictly_subsumed_clauses_in_Q[of [] _ [], unfolded append_Nil],
      OF refl])
    (auto simp: filter_q)
  also note proc_C[unfolded c'_nil, THEN tranclp.r_into_trancl[of (~w)]]
  also have wstate_of_dstate (N', [([], i)], [], n)
    ~w* wstate_of_dstate ([[], [([], i)], [], n)
    by (rule empty_N_if_Nil_in_P_or_Q) simp
  also have ... ~w wstate_of_dstate ([[], [], [([], i)], Suc n)
    by (rule arg_cong2[THEN iffD1, of _ _ _ _ (~w), OF _ _
      wrp.inference_computation[of {#} {#} i {#} n {#}]]])
    (auto simp: ord_FO_resolution_inferences_between_empty_empty)
  finally show ?thesis
    unfolding step st_n_cons ci .
next
  case c'_nnil: False
  note step = step[simplified c'_nnil, simplified]
  show ?thesis
  proof (cases is_tautology C' ∨ subsume (map fst P @ map fst Q) C')
    case taut_or_subs: True
    note step = step[simplified taut_or_subs, simplified]
  end

```

```

have wstate_of_dstate ((C', i) # N', P, Q, n)  $\rightsquigarrow_w$  wstate_of_dstate (N', P, Q, n)
proof (cases is_tautology C')
  case True
  then obtain A :: 'a where
    neg_a: Neg A  $\in$  set C' and pos_a: Pos A  $\in$  set C'
  unfolding is_tautology_def by blast
  show ?thesis
    by (rule arg_cong2[THEN iffD1, of _ _ _ _ ( $\rightsquigarrow_w$ ), OF _ _
      wrp.tautology_deletion[of A mset C' mset (map (apfst mset) N') i
        mset (map (apfst mset) P) mset (map (apfst mset) Q) n]]
      (use neg_a pos_a in simp_all))
  next
  case False
  then have subsume (map fst P @ map fst Q) C'
    using taut_or_subs by blast
  then obtain D :: 'a lclause where
    d_in: D  $\in$  set (map fst P @ map fst Q) and
    subs: subsumes (mset D) (mset C')
  unfolding subsume_def by blast
  show ?thesis
    by (rule arg_cong2[THEN iffD1, of _ _ _ _ ( $\rightsquigarrow_w$ ), OF _ _
      wrp.forward_subsumption[of mset D mset (map (apfst mset) P)
        mset (map (apfst mset) Q) mset C' mset (map (apfst mset) N') i n]],
      use d_in subs in (auto simp: subsume_def))
  qed
  then show ?thesis
    unfolding step st_n_cons ci using red_C by (rule rtranclp_into_tranclp1[rotated])
  next
  case not_taut_or_subs: False
  note step = step[simplified not_taut_or_subs, simplified]

define P' :: ('a literal list  $\times$  nat) list where
  P' = reduce_all C' P

obtain back_to_P Q' :: 'a dclause list where
  red_Q: (back_to_P, Q') = reduce_all2 C' Q
  by (metis prod.exhaust)
note step = step[unfolded red_Q[symmetric], simplified]

define Q'' :: ('a literal list  $\times$  nat) list where
  Q'' = filter (Not  $\circ$  strictly_subsume [C']  $\circ$  fst) Q'
define P'' :: ('a literal list  $\times$  nat) list where
  P'' = filter (Not  $\circ$  strictly_subsume [C']  $\circ$  fst) (back_to_P @ P')
note step = step[unfolded P'_def[symmetric] Q''_def[symmetric] P''_def[symmetric],
  simplified]

note red_C
also have wstate_of_dstate ((C', i) # N', P, Q, n)
   $\rightsquigarrow_w^*$  wstate_of_dstate ((C', i) # N', P', Q, n)
  unfolding P'_def by (rule reduce_clauses_in_P[of _ _ [], unfolded append_Nil]) simp+
also have ...  $\rightsquigarrow_w^*$  wstate_of_dstate ((C', i) # N', back_to_P @ P', Q', n)
  unfolding P'_def
  by (rule reduce_clauses_in_Q[of C' _ _ [] Q, folded red_Q,
    unfolded append_Nil prod.sel])
    (auto intro: reduce_idem simp: reduce_all_def)
also have ...  $\rightsquigarrow_w^*$  wstate_of_dstate ((C', i) # N', back_to_P @ P', Q'', n)
  unfolding Q''_def
  by (rule remove_strictly_subsumed_clauses_in_Q[of _ _ _ [], unfolded append_Nil])
    simp
also have ...  $\rightsquigarrow_w^*$  wstate_of_dstate ((C', i) # N', P'', Q'', n)
  unfolding P''_def
  by (rule remove_strictly_subsumed_clauses_in_P[of _ _ [], unfolded append_Nil]) auto
also note proc_C[THEN tranclp.r_into_trancl[of ( $\rightsquigarrow_w$ )]

```

```

    finally show ?thesis
      unfolding step st n_cons ci P''_def by simp
    qed
  qed
  qed
  qed
  qed

```

lemma *final_deterministic_RP_step*: $is_final_dstate\ St \implies deterministic_RP_step\ St = St$
by (*cases St*) (*auto simp: deterministic_RP_step.simps is_final_dstate.simps*)

lemma *deterministic_RP_SomeD*:
assumes *deterministic_RP* $(N, P, Q, n) = Some\ R$
shows $\exists N' P' Q' n'. (\exists k. (deterministic_RP_step \rightsquigarrow k)\ (N, P, Q, n) = (N', P', Q', n'))$
 $\wedge is_final_dstate\ (N', P', Q', n') \wedge R = map\ fst\ Q'$

proof (*induct rule: deterministic_RP.raw_induct[OF _ assms]*)
case (*1 self_call St R*)
note *ih = this(1) and step = this(2)*

obtain $N\ P\ Q :: 'a\ dclause\ list$ **and** $n :: nat$ **where**
 $st: St = (N, P, Q, n)$
by (*cases St*) *blast*
note $step = step[unfolded\ st,\ simplified]$

show ?*case*
proof (*cases is_final_dstate (N, P, Q, n)*)
case *True*
then have $(deterministic_RP_step \rightsquigarrow 0)\ (N, P, Q, n) = (N, P, Q, n)$
 $\wedge is_final_dstate\ (N, P, Q, n) \wedge R = map\ fst\ Q$
using *step* **by** *simp*
then show ?*thesis*
unfolding *st* **by** *blast*

next
case *nonfinal: False*
note $step = step[simplified\ nonfinal,\ simplified]$

obtain $N' P' Q' :: 'a\ dclause\ list$ **and** $n' k :: nat$ **where**
 $(deterministic_RP_step \rightsquigarrow k)\ (deterministic_RP_step\ (N, P, Q, n)) = (N', P', Q', n')$ **and**
 $is_final_dstate\ (N', P', Q', n')$
 $R = map\ fst\ Q'$
using *ih[OF step]* **by** *blast*
then show ?*thesis*
unfolding *st* *funpow_Suc_right[symmetric, THEN fun_cong, unfolded comp_apply]* **by** *blast*

qed
qed

context
fixes
 $N0 :: 'a\ dclause\ list$ **and**
 $n0 :: nat$ **and**
 $R :: 'a\ lclause\ list$

begin

abbreviation *St0* $:: 'a\ dstate$ **where**
 $St0 \equiv (N0, [], [], n0)$

abbreviation *grounded_N0* **where**
 $grounded_N0 \equiv grounding_of_cls\ (set\ (map\ (mset \circ fst)\ N0))$

abbreviation *grounded_R* $:: 'a\ clause\ set$ **where**
 $grounded_R \equiv grounding_of_cls\ (set\ (map\ mset\ R))$

primcorec *derivation_from* $:: 'a\ dstate \Rightarrow 'a\ dstate\ llist$ **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 \equiv *derivation_from St0*

abbreviation *wSts* :: 'a *wstate* *llist* **where**
wSts \equiv *lmap wstate_of_dstate Sts*

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

proof –

have *Sts'* = *derivation_from St0'* \implies *full_chain* (\rightsquigarrow_w^+) (*lmap wstate_of_dstate Sts'*)
for *St0' Sts'*

proof (*coinduction arbitrary: St0' Sts' rule: full_chain.coinduct*)

case *sts'*: *full_chain*

show ?*case*

proof (*cases is_final_dstate St0'*)

case *True*

then have *ltl (lmap wstate_of_dstate Sts')* = *LNil*

unfolding *sts'* **by** *simp*

then have *lmap wstate_of_dstate Sts'* = *LCons (wstate_of_dstate St0') LNil*

unfolding *sts'* **by** (*subst derivation_from.code, subst (asm) derivation_from.code, auto*)

moreover have $\bigwedge St''. \neg wstate_of_dstate St0' \rightsquigarrow_w St''$

using *True* **by** (*rule is_final_dstate_imp_not_weighted_RP*)

ultimately show ?*thesis*

by (*meson tranclpD*)

next

case *nfinal: False*

have *lmap wstate_of_dstate Sts'* =

LCons (wstate_of_dstate St0') (lmap wstate_of_dstate (ltl Sts'))

unfolding *sts'* **by** (*subst derivation_from.code*) *simp*

moreover have *ltl Sts'* = *derivation_from (deterministic_RP_step St0')*

unfolding *sts'* **using** *nfinal* **by** (*subst derivation_from.code*) *simp*

moreover have *wstate_of_dstate St0' \rightsquigarrow_w^+ wstate_of_dstate (lhd (ltl Sts'))*

unfolding *sts'* **using** *nonfinal_deterministic_RP_step[OF nfinal refl]* *nfinal*

by (*subst derivation_from.code*) *simp*

ultimately show ?*thesis*

by *fastforce*

qed

qed

then show ?*thesis*

by *blast*

qed

lemmas *deriv_wSts_trancl_weighted_RP* = *full_chain_imp_chain[OF full_deriv_wSts_trancl_weighted_RP]*

definition *sswSts* :: 'a *wstate* *llist* **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*

unfolding *sswSts_def*

by (*rule someI_ex[OF full_chain_tranclp_imp_exists_full_chain[OF full_deriv_wSts_trancl_weighted_RP]]*)

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_innull_sswSts*: $\neg \text{lnull } \text{sswSts}$
using *deriv_sswSts_weighted_RP* **by** (*cases rule: chain.cases*) *auto*

lemma *empty_ssgP0*: *wrp.P_of_wstate* (*lhd sswSts*) = {}
unfolding *lhd_sswSts* **by** (*subst derivation_from.code*) *simp*

lemma *empty_ssgQ0*: *wrp.Q_of_wstate* (*lhd sswSts*) = {}
unfolding *lhd_sswSts* **by** (*subst derivation_from.code*) *simp*

lemmas *sswSts_thms* = *full_deriv_sswSts_weighted_RP empty_ssgP0 empty_ssgQ0*

abbreviation *S_ssgQ* :: 'a *clause* \Rightarrow 'a *clause* **where**
S_ssgQ \equiv *wrp.S_gQ sswSts*

abbreviation *ord_Γ* :: 'a *inference set* **where**
ord_Γ \equiv *ground_resolution_with_selection.ord_Γ S_ssgQ*

abbreviation *Rf* :: 'a *clause set* \Rightarrow 'a *clause set* **where**
Rf \equiv *standard_redundancy_criterion.Rf*

abbreviation *Ri* :: 'a *clause set* \Rightarrow 'a *inference set* **where**
Ri \equiv *standard_redundancy_criterion.Ri ord_Γ*

abbreviation *saturated_upto* :: 'a *clause set* \Rightarrow *bool* **where**
saturated_upto \equiv *redundancy_criterion.saturated_upto ord_Γ Rf Ri*

context
assumes *drp_some*: *deterministic_RP St0* = *Some R*
begin

lemma *lfinite_Sts*: *lfinite Sts*
proof (*induct rule: deterministic_RP.raw_induct[OF drp_some]*)
case (*1 self_call St St'*)
note *ih* = *this(1)* **and** *step* = *this(2)*
show *?case*
using *step* **by** (*subst derivation_from.code, auto intro: ih*)
qed

lemma *lfinite_wSts*: *lfinite wSts*
by (*rule lfinite_lmap[THEN iffD2, OF lfinite_Sts]*)

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: $I \models_s \text{grounded_N0} \longleftrightarrow I \models_s \text{grounded_R}$ (**is** *?model*)
proof –
obtain *N' P' Q'* :: 'a *dclause list* **and** *n' k* :: *nat* **where**
k_steps: (*deterministic_RP_step* $\hat{\sim} k$) *St0* = (*N', P', Q', n'*) (**is** $_ = ?Stk$) **and**
final: *is_final_dstate* (*N', P', Q', n'*) **and**
r: *R* = *map fst Q'*
using *deterministic_RP_SomeD[OF drp_some]* **by** *blast*

have *wrp*: *wstate_of_dstate St0* \rightsquigarrow_w^* *wstate_of_dstate (llast Sts)*
using *lfinite_chain_imp_rtranclp_lhd_llast*
by (*metis (no_types) deriv_sswSts_weighted_RP derivation_from.disc_iff derivation_from.simps(2)*
lfinite_Sts lfinite_sswSts llast_lmap llist.map_sel(1) sswSts)

have *last_sts*: *llast Sts* = *?Stk*
proof –
have (*deterministic_RP_step* $\hat{\sim} k'$) *St0'* = *?Stk* \implies *llast (derivation_from St0')* = *?Stk*
for *St0' k'*


```

proof (induct k' arbitrary: St0')
  case 0
  then show ?case
    using final by (subst derivation_from.code) simp
next
  case (Suc k')
  note ih = this(1) and suc_k'_steps = this(2)
  show ?case
  proof (cases is_final_dstate St0')
    case True
    then show ?thesis
      using ih[of deterministic_RP_step St0'] suc_k'_steps final_deterministic_RP_step
        funpow_fixpoint[of deterministic_RP_step]
      by auto
    next
    case False
    then show ?thesis
      using ih[of deterministic_RP_step St0'] suc_k'_steps
      by (subst derivation_from.code) (simp add: llst_LCons funpow_swap1[symmetric])
  qed
qed
then show ?thesis
  using k_steps by blast
qed

have fin_gr_fgsts: lfinite (lmap wrp.grounding_of_wstate sswSts)
  by (rule lfinite_lmap[THEN iffD2, OF lfinite_sswSts])

have lim_last: Liminf_llist (lmap wrp.grounding_of_wstate sswSts) =
  wrp.grounding_of_wstate (llast sswSts)
  unfolding lfinite_Liminf_llist[OF fin_gr_fgsts] llst_lmap[OF lfinite_sswSts not_lnull_sswSts]
  using not_lnull_sswSts by simp

have gr_st0: wrp.grounding_of_wstate (wstate_of_dstate St0) = grounded_NO
  unfolding comp_def by simp

have ?saturated  $\wedge$  ?model
proof (cases []  $\in$  set R)
  case True
  then have emp_in: {#}  $\in$  grounded_R
    unfolding grounding_of_cls_def grounding_of_cls_def by (auto intro: ex_ground_subst)

  have grounded_R  $\subseteq$  wrp.grounding_of_wstate (llast sswSts)
    unfolding r llst_sswSts
    by (simp add: last_sts llst_lmap[OF lfinite_Sts] grounding_of_cls_def)
  then have gr_last_st: grounded_R  $\subseteq$  wrp.grounding_of_wstate (wstate_of_dstate (llast Sts))
    by (simp add: lfinite_Sts llst_lmap llst_sswSts)

  have gr_r_fls:  $\neg I \models_s$  grounded_R
    using emp_in unfolding true_cls_def by force
  then have gr_last_fls:  $\neg I \models_s$  wrp.grounding_of_wstate (wstate_of_dstate (llast Sts))
    using gr_last_st unfolding true_cls_def by auto

  have ?saturated
    unfolding wrp.ord_ $\Gamma$ _saturated_upto_def[OF sswSts_thms]
      wrp.ord_ $\Gamma$ _contradiction_Rf[OF sswSts_thms emp_in] inference_system.inferences_from_def
    by auto
  moreover have ?model
    unfolding gr_r_fls[THEN eq_False[THEN iffD2]]
    by (rule rtranclp_imp_eq_image[of ( $\rightsquigarrow_w$ )  $\lambda St. I \models_s$  wrp.grounding_of_wstate St, OF _ wrp,
      unfolded gr_st0 gr_last_fls[THEN eq_False[THEN iffD2]]])
    (use wrp.weighted_RP_model[OF sswSts_thms] in blast)
  ultimately show ?thesis

```

```

    by blast
next
case False
then have gr_last: wrp.grounding_of_wstate (llast sswSts) = grounded_R
  using final_unfolding r llast_sswSts
  by (simp add: last_sts llast_lmap[OF lfinite_Sts] comp_def is_final_dstate.simps)
then have gr_last_st: wrp.grounding_of_wstate (wstate_of_dstate (llast Sts)) = grounded_R
  by (simp add: lfinite_Sts llast_lmap llast_sswSts)

have ?saturated
  using wrp.weighted_RP_saturated[OF sswSts_thms, unfolded gr_last lim_last] by auto
moreover have ?model
  by (rule rtranclp_imp_eq_image[of ( $\rightsquigarrow_w$ )  $\lambda St. I \models wrp.grounding\_of\_wstate\ St$ , OF_ wrp,
    unfolded gr_st0 gr_last_st])
  (use wrp.weighted_RP_model[OF sswSts_thms] in blast)
ultimately show ?thesis
  by blast
qed
then show ?saturated and ?model
  by blast+
qed

corollary deterministic_RP_refutation:
 $\neg$  satisfiable grounded_NO  $\longleftrightarrow$  {#}  $\in$  grounded_R (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
assume ?rhs
then have  $\neg$  satisfiable grounded_R
  unfolding true_cls_def true_cls_def by force
then show ?lhs
  using deterministic_RP_model[THEN iffD1] by blast
next
assume ?lhs
then have  $\neg$  satisfiable grounded_R
  using deterministic_RP_model[THEN iffD2] by blast
then show ?rhs
  unfolding wrp.ord_ $\Gamma$ _saturated_upto_complete[OF sswSts_thms deterministic_RP_saturated] .
qed

end

context
  assumes drp_none: deterministic_RP St0 = None
begin

theorem deterministic_RP_complete: satisfiable grounded_NO
proof (rule ccontr)
  assume unsat:  $\neg$  satisfiable grounded_NO

  have unsat_wSts0:  $\neg$  satisfiable (wrp.grounding_of_wstate (lhd wSts))
    using unsat by (subst derivation_from.code) (simp add: comp_def)

  have bot_in_ss: {#}  $\in$  Q_of_state (wrp.Liminf_wstate sswSts)
    by (rule wrp.weighted_RP_complete[OF sswSts_thms unsat_wSts0[folded lhd_sswSts]])
  have bot_in_lim: {#}  $\in$  Q_of_state (wrp.Liminf_wstate wSts)
proof (cases lfinite Sts)
  case fin: True
  have wrp.Liminf_wstate sswSts = wrp.Liminf_wstate wSts
    by (rule Liminf_state_fin, simp_all add: fin lfinite_sswSts_iff not_lnull_sswSts,
      subst (1 2) llast_lmap,
      simp_all add: lfinite_sswSts_iff fin not_lnull_sswSts llast_sswSts)
  then show ?thesis
    using bot_in_ss by simp
next

```

```

    case False
  then show ?thesis
    using bot_in_ss Q_of_Liminf_state_inf[OF_emb_lmap[OF_emb_sswSts]] by auto
qed
then obtain k :: nat where
  k_lt: enat k < llength Sts and
  emp_in: {#} ∈ wrp.Q_of_wstate (lnth wSts k)
  unfolding Liminf_state_def Liminf_llist_def by auto
have emp_in: {#} ∈ Q_of_state (state_of_dstate ((deterministic_RP_step  $\sim$  k) St0))
proof -
  have enat k < llength Sts'  $\implies$  Sts' = derivation_from St0'  $\implies$ 
    {#} ∈ wrp.Q_of_wstate (lnth (lmap wstate_of_dstate Sts') k)  $\implies$ 
    {#} ∈ Q_of_state (state_of_dstate ((deterministic_RP_step  $\sim$  k) St0')) for St0' Sts' k
  proof (induction k arbitrary: St0' Sts')
    case 0
    then show ?case
      by (subst (asm) derivation_from.code, cases St0', auto simp: comp_def)
  next
    case (Suc k)
    note ih = this(1) and sk_lt = this(2) and sts' = this(3) and emp_in_sk = this(4)

    have k_lt: enat k < llength (ltl Sts')
      using sk_lt by (cases Sts') (auto simp: Suc_ile_eq)
    moreover have ltl Sts' = derivation_from (deterministic_RP_step St0')
      using sts' k_lt by (cases Sts') auto
    moreover have {#} ∈ wrp.Q_of_wstate (lnth (lmap wstate_of_dstate (ltl Sts')) k)
      using emp_in_sk k_lt by (cases Sts') auto
    ultimately show ?case
      using ih[of ltl Sts' deterministic_RP_step St0'] by (simp add: funpow_swap1)
  qed
  then show ?thesis
    using k_lt emp_in by blast
qed
have deterministic_RP St0  $\neq$  None
  by (rule is_final_dstate_funpow_imp_deterministic_RP_neq_None[of Suc k],
    cases (deterministic_RP_step  $\sim$  k) St0,
    use emp_in in {force simp: deterministic_RP_step.simps is_final_dstate.simps})
then show False
  using drp_none ..
qed
end
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) *mg*

abbreviation *subst_apply_literal* ::

$(f, 'v)$ term literal $\Rightarrow (f, 'v, 'w)$ gsubst $\Rightarrow (f, 'w)$ term literal (**infixl** ·lit 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** ·cls 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*: $S \subseteq \# C \Longrightarrow \text{vars_clause } S \subseteq \text{vars_clause } C$

unfolding *vars_clause_def* **by** *auto*

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

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)

using *assms* **proof** (*induction* *A*)

case (*Var* *x*)

then show ?*case* **by** *auto*

next

case (*Fun* *f* *ts*)

then show ?*case* **unfolding** *is_ground_atm_def*

by *auto*

qed

lemma *is_ground_lit_is_ground_on_var*:

assumes *ground_lit*: *is_ground_lit* $(\text{subst_lit } L \sigma)$ **and** $v \in \text{vars_lit } L$

shows *is_ground_atm* (σv)

proof –

let ?*A* = *atm_of* *L*

from $v \in \text{vars_lit } L$ **have** $A_p: v \in \text{vars_term } ?A$

by *auto*

then have *is_ground_atm* $(?A \cdot \sigma)$

using *ground_lit* **unfolding** *is_ground_lit_def* **by** *auto*

then show ?*thesis*

using A_p *is_ground_atm_is_ground_on_var* **by** *metis*

qed

lemma *is_ground_cls_is_ground_on_var*:

assumes

ground_clause: *is_ground_cls* $(\text{subst_cls } C \sigma)$ **and**

$v \in \text{vars_clause } C$

shows *is_ground_atm* (σv)

proof –

from $v \in \text{vars_clause } C$ **obtain** *L* **where** $L_p: L \in \# C$ $v \in \text{vars_lit } L$

unfolding *vars_clause_def* **by** *auto*

then have *is_ground_lit* $(\text{subst_lit } L \sigma)$

```

    using ground_clause unfolding is_ground_cls_def subst_cls_def by auto
  then show ?thesis
    using L_p is_ground_lit is_ground_on_var by metis
qed

```

```

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 ( $\sigma v$ )
proof -
  from v_in_Cs obtain C where C_p:  $C \in \text{set } Cs$   $v \in \text{vars\_clause } C$ 
    unfolding vars_clause_list_def by auto
  then have is_ground_cls (subst_cls C  $\sigma$ )
    using ground_list unfolding is_ground_cls_list_def subst_cls_list_def by auto
  then show ?thesis
    using C_p is_ground_cls_is_ground_on_var by metis
qed

```

```

lemma same_on_vars_lit:
  assumes  $\forall v \in \text{vars\_lit } L. \sigma v = \tau v$ 
  shows subst_lit L  $\sigma = \text{subst\_lit } L \tau$ 
  using assms
proof (induction L)
  case (Pos x)
  then have  $\forall v \in \text{vars\_term } x. \sigma v = \tau v \implies \text{subst\_atm\_abbrev } x \sigma = \text{subst\_atm\_abbrev } x \tau$ 
    using term_subst_eq by metis+
  then show ?case
    unfolding subst_lit_def using Pos by auto
next
  case (Neg x)
  then have  $\forall v \in \text{vars\_term } x. \sigma v = \tau v \implies \text{subst\_atm\_abbrev } x \sigma = \text{subst\_atm\_abbrev } x \tau$ 
    using term_subst_eq by metis+
  then show ?case
    unfolding subst_lit_def using Neg by auto
qed

```

```

lemma in_list_of_mset_in_S:
  assumes  $i < \text{length } (\text{list\_of\_mset } S)$ 
  shows  $\text{list\_of\_mset } S ! i \in \# S$ 
proof -
  from assms have  $\text{list\_of\_mset } S ! i \in \text{set } (\text{list\_of\_mset } S)$ 
    by auto
  then have  $\text{list\_of\_mset } S ! i \in \# \text{mset } (\text{list\_of\_mset } S)$ 
    by (meson in_multiset_in_set)
  then show ?thesis
    by auto
qed

```

```

lemma same_on_vars_clause:
  assumes  $\forall v \in \text{vars\_clause } S. \sigma v = \tau v$ 
  shows  $\text{subst\_cls } S \sigma = \text{subst\_cls } S \tau$ 
  by (smt assms image_eqI image_mset_cong2 mem_simps(9) same_on_vars_lit set_image_mset
    subst_cls_def vars_clause_def)

```

```

lemma vars_partitioned_var_disjoint:
  assumes vars_partitioned Cs
  shows var_disjoint Cs
  unfolding var_disjoint_def
proof (intro allI impI)
  fix  $\sigma s :: \langle ('b \Rightarrow ('a, 'b) \text{ term}) \text{ list} \rangle$ 
  assume  $\text{length } \sigma s = \text{length } Cs$ 
  with assms[unfolded vars_partitioned_def] Fun_More.fun_merge[of map vars_clause Cs nth  $\sigma s$ ]
  obtain  $\sigma$  where

```

$\sigma_p: \forall i < \text{length } (\text{map vars_clause } Cs). \forall x \in \text{map vars_clause } Cs ! i. \sigma x = (\sigma s ! i) x$
by auto
have $\forall i < \text{length } Cs. \forall S. S \subseteq \# Cs ! i \longrightarrow \text{subst_cls } S (\sigma s ! i) = \text{subst_cls } S \sigma$
proof (rule, rule, rule, rule)
fix $i :: \text{nat}$ **and** $S :: ('a, 'b) \text{ term literal multiset}$
assume
 $i < \text{length } Cs$ **and**
 $S \subseteq \# Cs ! i$
then have $\forall v \in \text{vars_clause } S. (\sigma s ! i) v = \sigma v$
using *vars_clause_mono*[of $S Cs ! i$] σ_p **by auto**
then show $\text{subst_cls } S (\sigma s ! i) = \text{subst_cls } S \sigma$
using *same_on_vars_clause* **by auto**
qed
then show $\exists \tau. \forall i < \text{length } Cs. \forall S. S \subseteq \# Cs ! i \longrightarrow \text{subst_cls } S (\sigma s ! i) = \text{subst_cls } S \tau$
by auto
qed

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

lemma *vars_in_instance_in_range_lit*: $\text{vars_lit } (\text{subst_lit } L \sigma) \subseteq \text{Union } (\text{image vars_term } (\text{range } \sigma))$
proof (*induction L*)
case (*Pos A*)
have $\text{vars_term } (A \cdot \sigma) \subseteq \text{Union } (\text{image vars_term } (\text{range } \sigma))$
using *vars_in_instance_in_range_term*[of $A \sigma$] **by blast**
then show *?case* **by auto**
next
case (*Neg A*)
have $\text{vars_term } (A \cdot \sigma) \subseteq \text{Union } (\text{image vars_term } (\text{range } \sigma))$
using *vars_in_instance_in_range_term*[of $A \sigma$] **by blast**
then show *?case* **by auto**
qed

lemma *vars_in_instance_in_range_cls*:
 $\text{vars_clause } (\text{subst_cls } C \sigma) \subseteq \text{Union } (\text{image vars_term } (\text{range } \sigma))$
unfolding *vars_clause_def subst_cls_def* **using** *vars_in_instance_in_range_lit*[of σ] **by auto**

primrec *renamings_apart* :: $('f, \text{nat}) \text{ term clause list} \Rightarrow (('f, \text{nat}) \text{ subst}) \text{ list}$ **where**
 $\text{renamings_apart } [] = []$
 $|\ \text{renamings_apart } (C \# Cs) =$
 $(\text{let } \sigma s = \text{renamings_apart } Cs \text{ in}$
 $(\lambda v. \text{Var } (v + \text{Max } (\text{vars_clause_list } (\text{subst_cls_lists } Cs \sigma s) \cup \{0\}) + 1)) \# \sigma s)$

definition *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 *len_renamings_apart*: $\text{length } (\text{renamings_apart } Cs) = \text{length } Cs$
by (*induction Cs*) (*auto simp: Let_def*)

lemma *renamings_apart_is_Var*: $\forall \sigma \in \text{set } (\text{renamings_apart } Cs). \forall x. \text{is_Var } (\sigma x)$
by (*induction Cs*) (*auto simp: Let_def*)

lemma *renamings_apart_inj*: $\forall \sigma \in \text{set } (\text{renamings_apart } Cs). \text{inj } \sigma$
proof (*induction Cs*)
case (*Cons a Cs*)
then have *inj* $(\lambda v. \text{Var } (\text{Suc } (v + \text{Max } (\text{vars_clause_list } (\text{subst_cls_lists } Cs (\text{renamings_apart } Cs)) \cup \{0\}))))$
by (*meson add_right_imp_eq injI nat.inject term.inject*(1))
then show *?case*
using *Cons* **by** (*auto simp: Let_def*)
qed auto

```

lemma finite_vars_clause[simp]: finite (vars_clause x)
  unfolding vars_clause_def by auto

lemma finite_vars_clause_list[simp]: finite (vars_clause_list Cs)
  unfolding vars_clause_list_def by (induction Cs) auto

lemma Suc_Max_notin_set: finite X  $\implies$  Suc (v + Max (insert 0 X))  $\notin$  X
  by (metis Max.boundedE Suc_n_not_le_n empty_iff finite.insertI le_add2 vimageE vimageI
    vimage_Suc_insert_0)

lemma vars_partitioned_Nil[simp]: vars_partitioned []
  unfolding vars_partitioned_def by auto

lemma subst_cls_lists_Nil[simp]: subst_cls_lists Cs [] = []
  unfolding subst_cls_lists_def by auto

lemma vars_clause_hd_partitioned_from_tl:
  assumes Cs  $\neq$  []
  shows vars_clause (hd (subst_cls_lists Cs (renamings_apart Cs)))
     $\cap$  vars_clause_list (tl (subst_cls_lists Cs (renamings_apart Cs))) = {}
  using assms
proof (induction Cs)
  case (Cons C Cs)
  define  $\sigma' :: \text{nat} \Rightarrow \text{nat}$ 
    where  $\sigma' = (\lambda v. (\text{Suc } (v + \text{Max } ((\text{vars\_clause\_list } (\text{subst\_cls\_lists } Cs$ 
      (renamings_apart Cs))) \cup \{0\}))))))
  define  $\sigma :: \text{nat} \Rightarrow ('a, \text{nat}) \text{ term}$ 
    where  $\sigma = (\lambda v. \text{Var } (\sigma' v))$ 

  have vars_clause (subst_cls C  $\sigma$ )  $\subseteq \bigcup$  (vars_term 'range  $\sigma$ )
    using vars_in_instance_in_range_cls[of C hd (renamings_apart (C # Cs))]  $\sigma$ _def  $\sigma'$ _def
    by (auto simp: Let_def)
  moreover have  $\bigcup$  (vars_term 'range  $\sigma$ )
     $\cap$  vars_clause_list (subst_cls_lists Cs (renamings_apart Cs)) = {}
  proof -
    have range  $\sigma' \cap$  vars_clause_list (subst_cls_lists Cs (renamings_apart Cs)) = {}
      unfolding  $\sigma'$ _def using Suc_Max_notin_set by auto
    then show ?thesis
      unfolding  $\sigma$ _def  $\sigma'$ _def by auto
  qed
  ultimately have vars_clause (subst_cls C  $\sigma$ )
     $\cap$  vars_clause_list (subst_cls_lists Cs (renamings_apart Cs)) = {}
    by auto
  then show ?case
    unfolding  $\sigma$ _def  $\sigma'$ _def unfolding subst_cls_lists_def
    by (simp add: Let_def subst_cls_lists_def)
qed auto

lemma vars_partitioned_renamings_apart: vars_partitioned (subst_cls_lists Cs (renamings_apart Cs))
proof (induction Cs)
  case (Cons C Cs)
  {
    fix i :: nat and j :: nat
    assume ij:
      i < Suc (length Cs)
      j < i
    have vars_clause (subst_cls_lists (C # Cs) (renamings_apart (C # Cs))) ! i  $\cap$ 
      vars_clause (subst_cls_lists (C # Cs) (renamings_apart (C # Cs))) ! j =
      {}
    proof (cases i; cases j)
      fix j' :: nat
      assume i'j':
        i = 0
  }

```

```

    j = Suc j'
  then show vars_clause (subst_cls_lists (C # Cs) (renamings_apart (C # Cs)) ! i) ∩
    vars_clause (subst_cls_lists (C # Cs) (renamings_apart (C # Cs)) ! j) =
    {}
  using ij by auto
next
fix i' :: nat
assume i'j':
  i = Suc i'
  j = 0
have disjoint_C_Cs: vars_clause (subst_cls_lists (C # Cs) (renamings_apart (C # Cs)) ! 0) ∩
  vars_clause_list ((subst_cls_lists Cs (renamings_apart Cs))) = {}
  using vars_clause_hd_partitioned_from_tl[of C # Cs]
  by (simp add: Let_def subst_cls_lists_def)
{
  fix x
  assume asm: x ∈ vars_clause (subst_cls_lists Cs (renamings_apart Cs) ! i')
  then have (subst_cls_lists Cs (renamings_apart Cs) ! i')
    ∈ set (subst_cls_lists Cs (renamings_apart Cs))
    using i'j' ij unfolding subst_cls_lists_def
    by (metis Suc_less_SucD length_map len_renamings_apart length_zip min_less_iff_conj
      nth_mem)
  moreover from asm have
    x ∈ vars_clause (subst_cls_lists Cs (renamings_apart Cs) ! i')
    using i'j' ij
    unfolding subst_cls_lists_def by simp
  ultimately have ∃ D ∈ set (subst_cls_lists Cs (renamings_apart Cs)). x ∈ vars_clause D
    by auto
}
then have vars_clause (subst_cls_lists Cs (renamings_apart Cs) ! i')
  ⊆ Union (set (map vars_clause ((subst_cls_lists Cs (renamings_apart Cs)))))
  by auto
then have vars_clause (subst_cls_lists (C # Cs) (renamings_apart (C # Cs)) ! 0) ∩
  vars_clause (subst_cls_lists Cs (renamings_apart Cs) ! i') =
  {} using disjoint_C_Cs unfolding vars_clause_list_def by auto
moreover
have subst_cls_lists Cs (renamings_apart Cs) ! i' =
  subst_cls_lists (C # Cs) (renamings_apart (C # Cs)) ! i
  using i'j' ij unfolding subst_cls_lists_def by (simp add: Let_def)
ultimately
show vars_clause (subst_cls_lists (C # Cs) (renamings_apart (C # Cs)) ! i) ∩
  vars_clause (subst_cls_lists (C # Cs) (renamings_apart (C # Cs)) ! j) =
  {}
  using i'j' by (simp add: Int_commute)
next
fix i' :: nat and j' :: nat
assume i'j':
  i = Suc i'
  j = Suc j'
have i' < length (subst_cls_lists Cs (renamings_apart Cs))
  using ij i'j' unfolding subst_cls_lists_def by (auto simp: len_renamings_apart)
moreover
have j' < length (subst_cls_lists Cs (renamings_apart Cs))
  using ij i'j' unfolding subst_cls_lists_def by (auto simp: len_renamings_apart)
moreover
have i' ≠ j'
  using ⟨i = Suc i'⟩ ⟨j = Suc j'⟩ ij by blast
ultimately
have vars_clause (subst_cls_lists Cs (renamings_apart Cs) ! i') ∩
  vars_clause (subst_cls_lists Cs (renamings_apart Cs) ! j') =
  {}
  using Cons unfolding vars_partitioned_def by auto
then show vars_clause (subst_cls_lists (C # Cs) (renamings_apart (C # Cs)) ! i) ∩

```



```

vars_clause (subst_cls_lists (C # Cs) (renamings_apart (C # Cs)) ! j) =
  {}
unfolding i'j'
by (simp add: subst_cls_lists_def Let_def)
next
assume
  ⟨i = 0⟩ and
  ⟨j = 0⟩
then show ⟨vars_clause (subst_cls_lists (C # Cs) (renamings_apart (C # Cs)) ! i) ∩
  vars_clause (subst_cls_lists (C # Cs) (renamings_apart (C # Cs)) ! j) =
  {}⟩ using ij by auto
qed
}
then show ?case
unfolding vars_partitioned_def
by (metis (no_types, lifting) Int_commute Suc_lessI len_renamings_apart length_map
  length_nth_simps(2) length_zip min.idem nat.inject not_less_eq subst_cls_lists_def)
qed auto

interpretation substitution (·) Var :: _ ⇒ ('f, nat) term (∘s) renamings_apart Fun undefined
proof (standard)
show  $\bigwedge A. A \cdot \text{Var} = A$ 
by auto
next
show  $\bigwedge A \tau \sigma. A \cdot \tau \circ_s \sigma = A \cdot \tau \cdot \sigma$ 
by auto
next
show  $\bigwedge \sigma \tau. (\bigwedge A. A \cdot \sigma = A \cdot \tau) \implies \sigma = \tau$ 
by (simp add: subst_term_eqI)
next
fix C :: ('f, nat) term clause
fix  $\sigma$ 
assume is_ground_cls (subst_cls C  $\sigma$ )
then have ground_atms_ $\sigma$ :  $\bigwedge v. v \in \text{vars\_clause } C \implies \text{is\_ground\_atm } (\sigma v)$ 
by (meson is_ground_cls is_ground_on_var)

define some_ground_trm :: ('f, nat) term where some_ground_trm = (Fun undefined [])
have ground_trm: is_ground_atm some_ground_trm
unfolding is_ground_atm_def some_ground_trm_def by auto
define  $\tau$  where  $\tau = (\lambda v. \text{if } v \in \text{vars\_clause } C \text{ then } \sigma v \text{ else some\_ground\_trm})$ 
then have  $\tau_\sigma$ :  $\forall v \in \text{vars\_clause } C. \sigma v = \tau v$ 
unfolding  $\tau$ _def by auto

have all_ground_ $\tau$ : is_ground_atm ( $\tau v$ ) for v
proof (cases v ∈ vars_clause C)
case True
then show ?thesis
using ground_atms_ $\sigma$   $\tau_\sigma$  by auto
next
case False
then show ?thesis
unfolding  $\tau$ _def using ground_trm by auto
qed
have is_ground_subst  $\tau$ 
unfolding is_ground_subst_def
proof
fix A
show is_ground_atm (subst_atm_abbrev A  $\tau$ )
proof (induction A)
case (Var v)
then show ?case using all_ground_ $\tau$  by auto
next
case (Fun f As)

```

```

    then show ?case using all_ground_τ
      by (simp add: is_ground_atm_def)
  qed
qed
moreover have  $\forall v \in \text{vars\_clause } C. \sigma v = \tau v$ 
  using  $\tau_\sigma$  unfolding vars_clause_list_def
  by blast
then have  $\text{subst\_cls } C \sigma = \text{subst\_cls } C \tau$ 
  using same_on_vars_clause by auto
ultimately show  $\exists \tau. \text{is\_ground\_subst } \tau \wedge \text{subst\_cls } C \tau = \text{subst\_cls } C \sigma$ 
  by auto
next
fix Cs :: ('f, nat) term clause list
show length (renamings_apart Cs) = length Cs
  using len_renamings_apart by auto
next
fix Cs :: ('f, nat) term clause list
fix ρ :: nat ⇒ ('f, nat) Term.term
assume ρ_renaming: ρ ∈ set (renamings_apart Cs)
{
  have inj_is_renaming:
     $\bigwedge \sigma :: ('f, \text{nat}) \text{ subst. } (\bigwedge x. \text{is\_Var } (\sigma x)) \implies \text{inj } \sigma \implies \text{is\_renaming } \sigma$ 
  proof -
    fix σ :: ('f, nat) subst
    fix x
    assume is_var_σ:  $\bigwedge x. \text{is\_Var } (\sigma x)$ 
    assume inj_σ: inj σ
    define σ' where σ' = var_map_of_subst σ
    have σ:  $\sigma = \text{Var} \circ \sigma'$ 
      unfolding σ'_def var_map_of_subst_def using is_var_σ by auto

    from is_var_σ inj_σ have inj σ'
      unfolding is_renaming_def unfolding subst_domain_def inj_on_def σ'_def var_map_of_subst_def
      by (metis term.collapse(1))
    then have inv σ' ∘ σ' = id
      using inv_o_cancel[of σ'] by simp
    then have Var ∘ (inv σ' ∘ σ') = Var
      by simp
    then have  $\forall x. (\text{Var} \circ (\text{inv } \sigma' \circ \sigma')) x = \text{Var } x$ 
      by metis
    then have  $\forall x. ((\text{Var} \circ \sigma') \circ_s (\text{Var} \circ (\text{inv } \sigma')) x = \text{Var } x$ 
      unfolding subst_compose_def by auto
    then have  $\sigma \circ_s (\text{Var} \circ (\text{inv } \sigma')) = \text{Var}$ 
      using σ by auto
    then show is_renaming σ
      unfolding is_renaming_def by blast
  qed
  then have  $\forall \sigma \in (\text{set } (\text{renamings\_apart } Cs)). \text{is\_renaming } \sigma$ 
    using renamings_apart_is_Var renamings_apart_inj by blast
}
then show is_renaming ρ
  using ρ_renaming by auto
next
fix Cs :: ('f, nat) term clause list
have vars_partitioned (subst_cls_lists Cs (renamings_apart Cs))
  using vars_partitioned_renamings_apart by auto
then show var_disjoint (subst_cls_lists Cs (renamings_apart Cs))
  using vars_partitioned_var_disjoint by auto
next
show  $\bigwedge \sigma \text{ As Bs. Fun undefined As } \cdot \sigma = \text{Fun undefined Bs} \iff \text{map } (\lambda A. A \cdot \sigma) \text{ As} = \text{Bs}$ 
  by simp
next
show wfP (strictly_generalizes_atm :: ('f, 'v) term ⇒ _ ⇒ _)

```

```

unfolding wfP_def
by (rule wf_subset[OF wf_subsumes])
  (auto simp: strictly_generalizes_atm_def generalizes_atm_def term_subsumable.subsumes_def
    subsumeseq_term.simps)
qed

```

```

fun pairs :: 'a list  $\Rightarrow$  ('a  $\times$  'a) list where
  pairs (x # y # xs) = (x, y) # pairs (y # xs) |
  pairs _ = []

```

```

derive compare term
derive compare literal

```

```

lemma class_linorder_compare: class.linorder (le_of_comp compare) (lt_of_comp compare)
apply standard
  apply (simp_all add: lt_of_comp_def le_of_comp_def split: order.splits)
  apply (metis comparator.sym comparator_compare invert_order.simps(1) order.distinct(5))
  apply (metis comparator_compare comparator_def order.distinct(5))
  apply (metis comparator.sym comparator_compare invert_order.simps(1) order.distinct(5))
by (metis comparator.sym comparator_compare invert_order.simps(2) order.distinct(5))

```

```

context begin
interpretation compare_linorder: linorder
  le_of_comp compare
  lt_of_comp compare
by (rule class_linorder_compare)

```

```

definition Pairs where
  Pairs AAA = concat (compare_linorder.sorted_list_of_set
    ((pairs  $\circ$  compare_linorder.sorted_list_of_set) '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 (induct xs rule: pairs.induct)
case (1 x y xs)
then show ?case
  unfolding pairs.simps list.set ball_Un ball_simps simp_thms fst_conv snd_conv by metis
qed simp_all

```

```

lemma in_pair_in_set:
  assumes (A,B)  $\in$  set ((pairs As))
  shows A  $\in$  set As  $\wedge$  B  $\in$  set As
  using assms
proof (induction As)
case (Cons A As)
  note Cons_outer = this
  show ?case
  proof (cases As)
  case Nil
  then show ?thesis
  using Cons_outer by auto
  next
  case (Cons B As')
  then show ?thesis using Cons_outer by auto
  qed
qed auto

```

```

lemma in_pairs_sorted_list_of_set_in_set:
  assumes
    finite AAA
     $\forall AA \in AAA. \text{finite } AA$ 
    AB_pairs  $\in$  (pairs  $\circ$  compare_linorder.sorted_list_of_set) 'AAA and
    (A :: _ :: compare, B)  $\in$  set AB_pairs

```

shows $\exists AA. AA \in AAA \wedge A \in AA \wedge B \in AA$
proof –
from *assms* **have** $AB_pairs \in (pairs \circ compare_linorder.sorted_list_of_set) \text{ 'AAA}$
by *auto*
then obtain *AA* **where**
 $AA_p: AA \in AAA \wedge (pairs \circ compare_linorder.sorted_list_of_set) AA = AB_pairs$
by *auto*
have $(A, B) \in set (pairs (compare_linorder.sorted_list_of_set AA))$
using $AA_p[]$ *assms*(4) **by** *auto*
then have $A \in set (compare_linorder.sorted_list_of_set AA)$ **and**
 $B \in set (compare_linorder.sorted_list_of_set AA)$
using *in_pair_in_set*[of *A*] **by** *auto*
then show *?thesis*
using *assms*(2) *AA_p* **by** *auto*
qed

lemma *unifiers_Pairs*:

assumes
finite *AAA* **and**
 $\forall AA \in AAA. finite\ AA$
shows $unifiers (set (Pairs\ AAA)) = \{\sigma. is_unifiers\ \sigma\ AAA\}$
proof (*rule*; *rule*)
fix $\sigma :: ('a, 'b)\ subst$
assume *asm*: $\sigma \in unifiers (set (Pairs\ AAA))$
have $\bigwedge AA. AA \in AAA \implies card (AA \cdot_{set} \sigma) \leq Suc\ 0$
proof –
fix $AA :: ('a, 'b)\ term\ set$
assume *asm'*: $AA \in AAA$
then have $\forall p \in set (pairs (compare_linorder.sorted_list_of_set\ AA)).$
 $subst_atm_abbrev (fst\ p)\ \sigma = subst_atm_abbrev (snd\ p)\ \sigma$
using *assms* *asm* **unfolding** *Pairs_def* **by** *auto*
then have $\forall A \in AA. \forall B \in AA. subst_atm_abbrev\ A\ \sigma = subst_atm_abbrev\ B\ \sigma$
using *assms* *asm'* **unfolding** *unifies_all_pairs_iff*
using *compare_linorder.sorted_list_of_set* **by** *blast*
then show $card (AA \cdot_{set} \sigma) \leq Suc\ 0$
by (*smt imageE* *card.empty* *card_Suc_eq* *card_mono* *finite.intros*(1) *finite_insert* *le_SucI*
singletonI *subsetI*)
qed
then show $\sigma \in \{\sigma. is_unifiers\ \sigma\ AAA\}$
using *assms* **by** (*auto* *simp*: *is_unifiers_def* *is_unifier_def* *subst_atms_def*)
next
fix $\sigma :: ('a, 'b)\ subst$
assume *asm*: $\sigma \in \{\sigma. is_unifiers\ \sigma\ AAA\}$

{
fix $AB_pairs\ A\ B$
assume
 $AB_pairs \in set (compare_linorder.sorted_list_of_set$
 $((pairs \circ compare_linorder.sorted_list_of_set) \text{ 'AAA}))$ **and**
 $(A, B) \in set\ AB_pairs$
then have $\exists AA. AA \in AAA \wedge A \in AA \wedge B \in AA$
using *assms* **by** (*simp* *add*: *in_pairs_sorted_list_of_set_in_set*)
then obtain *AA* **where**
 $a: AA \in AAA\ A \in AA\ B \in AA$
by *blast*
from *a* *assms* *asm* **have** $card_AA_sigma: card (AA \cdot_{set} \sigma) \leq Suc\ 0$
unfolding *is_unifiers_def* *is_unifier_def* *subst_atms_def* **by** *auto*
have $subst_atm_abbrev\ A\ \sigma = subst_atm_abbrev\ B\ \sigma$
proof (*cases* $card (AA \cdot_{set} \sigma) = Suc\ 0$)
case *True*
moreover
have $subst_atm_abbrev\ A\ \sigma \in AA \cdot_{set} \sigma$
using *a* *assms* *asm* $card_AA_sigma$ **by** *auto*

```

moreover
have subst_atm_abbrev  $B \sigma \in AA \cdot_{set} \sigma$ 
  using a assms asm card_AA_σ by auto
ultimately
show ?thesis
  using a assms asm card_AA_σ by (metis (no_types, lifting) card_Suc_eq singletonD)
next
case False
then have  $card (AA \cdot_{set} \sigma) = 0$ 
  using a assms asm card_AA_σ
  by arith
then show ?thesis
  using a assms asm card_AA_σ by auto
qed
}
then show  $\sigma \in unifiers (set (Pairs AAA))$ 
  unfolding Pairs_def unifiers_def by auto
qed

end

```

```

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

```

```

interpretation mgu ( $\cdot$ ) Var ::  $\_ \Rightarrow ('f :: compare, nat) term (\circ_s) Fun undefined$ 
  renamings_apart mgu_sets

```

proof

```

fix AAA :: ( $'a :: compare, nat$ ) term set set and  $\sigma :: ('a, nat) subst$ 
assume fin: finite AAA  $\forall AA \in AAA. finite AA$  and mgu_sets AAA = Some σ
then have is_imgu σ (set (Pairs AAA))
  using unify_sound unfolding mgu_sets_def by blast
then show is_mgu σ AAA
  unfolding is_imgu_def is_mgu_def unifiers_Pairs[OF fin] by auto

```

next

```

fix AAA :: ( $'a :: compare, nat$ ) term set set and  $\sigma :: ('a, nat) subst$ 
assume fin: finite AAA  $\forall AA \in AAA. finite AA$  and is_unifiers σ AAA
then have  $\sigma \in unifiers (set (Pairs AAA))$ 
  unfolding is_mgu_def unifiers_Pairs[OF fin] by auto
then show  $\exists \tau. mgu\_sets AAA = Some \tau$ 
  using unify_complete unfolding mgu_sets_def by blast
qed

```

```

derive linorder prod

```

```

derive linorder list

```

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

```

record 'f weights =

```

```

  w ::  $'f \times nat \Rightarrow nat$ 
  w0 :: nat
  pr_strict ::  $'f \times nat \Rightarrow 'f \times nat \Rightarrow bool$ 
  least ::  $'f \Rightarrow bool$ 
  scf ::  $'f \times nat \Rightarrow nat \Rightarrow nat$ 

```

```

class weighted =

```

```

  fixes weights ::  $'a weights$ 
  assumes weights_adm:
    admissible_kbo
    (w weights) (w0 weights) (pr_strict weights) ( $((pr\_strict weights) ==)$ ) (least weights) (scf weights)
  and pr_strict_total:  $fi = gj \vee pr\_strict weights fi gj \vee pr\_strict weights gj fi$ 
  and pr_strict_asymp: asymp (pr_strict weights)
  and scf_ok:  $i < n \implies scf weights (f, n) i \leq 1$ 

```

```

instantiation unit :: weighted begin

```

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

instance

by (*intro_classes*, *unfold_locales*) (auto simp: *weights_unit_def SN_iff_wf async_simps irreflp_def intro!*: *wf_subset[OF wf_inv_image[OF wf], of_snd]*)

end

global-interpretation *KBO*:

admissible_kbo
w (*weights* :: 'f :: weighted weights) *w0* (*weights* :: 'f :: weighted weights)
pr_strict *weights* ((*pr_strict* *weights*)⁼⁼) *least* *weights* *scf* *weights*
defines *weight* = *KBO.weight*
and *kbo* = *KBO.kbo*
by (*simp add: weights_adm*)

lemma *kbo_code*[*code*]: *kbo* *s* *t* =

(let *wt* = *weight* *t*; *ws* = *weight* *s* in
if *vars_term_ms* (*KBO.SCF* *t*) \subseteq # *vars_term_ms* (*KBO.SCF* *s*) \wedge *wt* \leq *ws*
then
(if *wt* < *ws* then (*True*, *True*)
else
(case *s* of
Var *y* \Rightarrow (*False*, case *t* of Var *x* \Rightarrow *True* | Fun *g* *ts* \Rightarrow *ts* = [] \wedge *least* *weights* *g*)
| Fun *f* *ss* \Rightarrow
(case *t* of
Var *x* \Rightarrow (*True*, *True*)
| Fun *g* *ts* \Rightarrow
if *pr_strict* *weights* (*f*, *length* *ss*) (*g*, *length* *ts*) then (*True*, *True*)
else if (*f*, *length* *ss*) = (*g*, *length* *ts*) then *lex_ext_unbounded* *kbo* *ss* *ts*
else (*False*, *False*))))
else (*False*, *False*))
by (*subst* *KBO.kbo_simps*) (auto simp: *Let_def split: term_splits*)

definition *less_kbo* *s* *t* = *fst* (*kbo* *t* *s*)

lemma *less_kbo_gtotal*: *ground* *s* \Longrightarrow *ground* *t* \Longrightarrow *s* = *t* \vee *less_kbo* *s* *t* \vee *less_kbo* *t* *s*
unfolding *less_kbo_def* **using** *KBO.S_ground_total* **by** (*metis* *pr_strict_total subset_UNIV*)

lemma *less_kbo_subst*:

fixes σ :: ('f :: weighted, 'v) *subst*
shows *less_kbo* *s* *t* \Longrightarrow *less_kbo* (*s* \cdot σ) (*t* \cdot σ)
unfolding *less_kbo_def* **by** (*rule* *KBO.S_subst*)

lemma *wfP_less_kbo*: *wfP* *less_kbo*

proof –

have *SN* {(*x*, *y*). *fst* (*kbo* *x* *y*)}
using *pr_strict_async* **by** (*fastforce* *simp: async_simps irreflp_def intro!*: *KBO.S_SN scf_ok*)
then show *thesis*
unfolding *SN_iff_wf wfP_def* **by** (*rule* *wf_subset*) (auto simp: *less_kbo_def*)

qed

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

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

lemma *less_trm_extension*: {(*s*, *t*). *less_kbo* *s* *t*} \subseteq *leq_term*

unfolding *leq_term_def*
by (*rule* *someI2_ex[OF total_well_order_extension[OF wfP_less_kbo[unfolded wfP_def]]]*) auto

lemma *less_trm_well_order*: *well_order* *leq_term*

unfolding *leq_term_def*
by (*rule* *someI2_ex[OF total_well_order_extension[OF wfP_less_kbo[unfolded wfP_def]]]*) auto

```

definition less_eq_term :: ('a :: weighted, 'b) term ⇒ _ ⇒ bool where
  less_eq_term = in_rel leq_term
definition less_term :: ('a :: weighted, 'b) term ⇒ _ ⇒ bool where
  less_term s t = strict (≤) s t

lemma leq_term_minus_Id: leq_term - Id = {(x,y). x < y}
using less_trm_well_order
unfolding well_order_on_def linear_order_on_def partial_order_on_def antisym_def less_term_def less_eq_term_def
by auto

lemma less_term_alt: (<) = in_rel (leq_term - Id)
by (simp add: in_rel_Collect_case_prod_eq leq_term_minus_Id)

instance
proof (standard, goal_cases less_less_eq refl trans antisym total)
  case (less_less_eq x y)
  then show ?case unfolding less_term_def ..
next
case (refl x)
  then show ?case using less_trm_well_order
  unfolding well_order_on_def linear_order_on_def partial_order_on_def preorder_on_def refl_on_def
  less_eq_term_def by auto
next
case (trans x y z)
  then show ?case using less_trm_well_order
  unfolding well_order_on_def linear_order_on_def partial_order_on_def preorder_on_def trans_def
  less_eq_term_def by auto
next
case (antisym x y)
  then show ?case using less_trm_well_order
  unfolding well_order_on_def linear_order_on_def partial_order_on_def antisym_def
  less_eq_term_def by auto
next
case (total x y)
  then show ?case using less_trm_well_order
  unfolding well_order_on_def linear_order_on_def partial_order_on_def preorder_on_def refl_on_def
  Relation.total_on_def less_eq_term_def by (cases x = y) auto
qed

end

instantiation term :: (weighted, type) wellorder begin
instance
using less_trm_well_order[unfolded well_order_on_def wf_def leq_term_minus_Id, THEN conjunct2]
by intro_classes (atomize, auto)
end

lemma ground_less_less_kbo: ground s ⇒ ground t ⇒ s < t ⇒ less_kbo s t
using less_kbo_gtotal[of s t] less_trm_extension
by (auto simp: less_term_def less_eq_term_def)

lemma less_kbo_less: less_kbo s t ⇒ s < t
using less_trm_extension
by (auto simp: less_term_alt less_kbo_def KBO.S_irrefl)

lemma is_ground_atm_ground: is_ground_atm t ⇔ ground t
unfolding is_ground_atm_def
by (induct t) (fastforce simp: in_set_conv_nth list_eq_iff_nth_eq)+

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 Naïve 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  $\Rightarrow False$ 
  | Some  $\rho \Rightarrow subsumes\_list\ Ls\ (remove1\ K\ Ks)\ \rho$ ))
```

```
lemma atm_of_map_literal[simp]: atm_of (map_literal f l) = f (atm_of l)
by (cases l; simp)
```

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

```
lemma extends_subst_refl[simp]: extends_subst  $\sigma\ \sigma$ 
unfolding extends_subst_def by auto
```

```
lemma extends_subst_trans: extends_subst  $\sigma\ \tau \implies extends\_subst\ \tau\ \rho \implies extends\_subst\ \sigma\ \rho$ 
unfolding extends_subst_def dom_def by (metis mem_Collect_eq)
```

```
lemma extends_subst_dom: extends_subst  $\sigma\ \tau \implies dom\ \sigma \subseteq dom\ \tau$ 
unfolding extends_subst_def dom_def by auto
```

```
lemma extends_subst_extends: extends_subst  $\sigma\ \tau \implies x \in dom\ \sigma \implies \tau\ x = \sigma\ x$ 
unfolding extends_subst_def dom_def by auto
```

```
lemma extends_subst_fun_upd_new:
   $\sigma\ x = None \implies extends\_subst\ (\sigma(x \mapsto t))\ \tau \longleftrightarrow extends\_subst\ \sigma\ \tau \wedge \tau\ x = Some\ t$ 
unfolding extends_subst_def dom_fun_upd subst_of_map_def
by (force simp add: dom_def split: option.splits)
```

```
lemma extends_subst_fun_upd_matching:
   $\sigma\ x = Some\ t \implies extends\_subst\ (\sigma(x \mapsto t))\ \tau \longleftrightarrow extends\_subst\ \sigma\ \tau$ 
unfolding extends_subst_def dom_fun_upd subst_of_map_def
by (auto simp add: dom_def split: option.splits)
```

```
lemma extends_subst_empty[simp]: extends_subst Map.empty  $\tau$ 
unfolding extends_subst_def by auto
```

```
lemma extends_subst_cong_term:
   $extends\_subst\ \sigma\ \tau \implies vars\_term\ t \subseteq dom\ \sigma \implies t \cdot subst\_of\_map\ Var\ \sigma = t \cdot subst\_of\_map\ Var\ \tau$ 
by (force simp: extends_subst_def subst_of_map_def split: option.splits intro!: term_subst_eq)
```

```
lemma extends_subst_cong_lit:
   $extends\_subst\ \sigma\ \tau \implies vars\_lit\ L \subseteq dom\ \sigma \implies L \cdot lit\ subst\_of\_map\ Var\ \sigma = L \cdot lit\ subst\_of\_map\ Var\ \tau$ 
by (cases L) (auto simp: extends_subst_cong_term)
```

```
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*
unfolding *vars_clause_def* **by** *auto*

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

unfolding *subsumes_modulo_def*

proof (*safe, goal_cases left_right right_left*)

case (*left_right* τ)

then show *?case*

by (*intro* *bexI[of _ L ·lit subst_of_map Var τ]*)

exI[of _ $\lambda x. \text{if } x \in \text{vars_lit } L \cup \text{dom } \sigma \text{ then } \tau \text{ else None}], \text{intro conjI exI[of _ } \tau]$)

(auto 0 3 simp: extends_subst_def dom_def split: if_splits

simp: insert_subset_eq_iff subst_lit_def intro!: extends_subst_cong_lit)

next

case (*right_left* $K \tau \tau'$)

then show *?case*

by (*intro* *bexI[of _ L ·lit subst_of_map Var τ] exI[of _ τ'], intro conjI exI[of _ τ]*)

(auto simp: insert_subset_eq_iff subst_lit_def extends_subst_cong_lit

intro: extends_subst_trans)

qed

lemma *decompose_Some_var_terms: decompose (Fun f ss) (Fun g ts) = Some eqs \implies*
 $f = g \wedge \text{length } ss = \text{length } ts \wedge \text{eqs} = \text{zip } ss \ ts \wedge$

$(\bigcup (t, u) \in \text{set } ((\text{Fun } f \ ss, \text{Fun } g \ ts) \# P). \text{vars_term } t) =$

$(\bigcup (t, u) \in \text{set } (\text{eqs} @ P). \text{vars_term } t)$

by (*drule decompose_Some*)

(fastforce simp: in_set_zip in_set_conv_nth Bex_def image_iff)

lemma *match_term_list_sound: match_term_list tus $\sigma = \text{Some } \tau \implies$*

extends_subst $\sigma \tau \wedge \text{dom } \tau = (\bigcup (t, u) \in \text{set } tus. \text{vars_term } t) \cup \text{dom } \sigma \wedge$

($\forall (t, u) \in \text{set } tus. t \cdot \text{subst_of_map } \text{Var } \tau = u)$)

proof (*induct tus σ rule: match_term_list.induct*)

case ($2 \ x \ t \ P \ \sigma$)

then show *?case*

by (*auto 0 3 simp: extends_subst_fun_upd_new extends_subst_fun_upd_matching*

subst_of_map_def dest: extends_subst_extends simp del: fun_upd_apply

split: if_splits option.splits)

next

case ($3 \ f \ ss \ g \ ts \ P \ \sigma$)

from $3(2)$ **obtain** *eqs where* *decompose (Fun f ss) (Fun g ts) = Some eqs*

match_term_list (eqs @ P) $\sigma = \text{Some } \tau$ by (auto split: option.splits)

with $3(1)[\text{OF this}]$ **show** *?case*

proof (*elim decompose_Some_var_terms[where $P = P, \text{elim_format}] \text{conjE, intro conjI, goal_cases extend dom subst}$*)

case *subst*

from *subst(3,5,6,7) show ?case*

by (*auto 0 6 simp: in_set_conv_nth list_eq_iff_nth_eq Ball_def*)

qed *auto*

qed *auto*

lemma *match_term_list_complete: match_term_list tus $\sigma = \text{None} \implies$*

extends_subst $\sigma \tau \implies \text{dom } \tau = (\bigcup (t, u) \in \text{set } tus. \text{vars_term } t) \cup \text{dom } \sigma \implies$

($\exists (t, u) \in \text{set } tus. t \cdot \text{subst_of_map } \text{Var } \tau \neq u)$)

proof (*induct tus σ arbitrary: τ rule: match_term_list.induct*)

case ($2 \ x \ t \ P \ \sigma$)

then show *?case*

by (*auto simp: extends_subst_fun_upd_new extends_subst_fun_upd_matching*

subst_of_map_def dest: extends_subst_extends simp del: fun_upd_apply

split: if_splits option.splits)

next

case ($3 \ f \ ss \ g \ ts \ P \ \sigma$)

```

show ?case
proof (cases decompose (Fun f ss) (Fun g ts) = None)
  case False
  with  $\exists(2)$  obtain eqs where decompose (Fun f ss) (Fun g ts) = Some eqs
    match_term_list (eqs @ P)  $\sigma = \text{None}$  by (auto split: option.splits)
  with  $\exists(1)$ [OF this  $\exists(3)$  trans[OF  $\exists(4)$  arg_cong[of _ _  $\lambda x. x \cup \text{dom } \sigma$ ]]] show ?thesis
  proof (elim decompose_Some_var_terms[where P = P, elim_format] conjE, goal_cases subst)
    case subst
    from subst(1)[OF subst(6)] subst(4,5) show ?case
    by (auto 0  $\exists$  simp: in_set_conv_nth list_eq_iff_nth_eq Ball_def)
  qed
qed auto
qed auto

```

```

lemma unique_extends_subst:
  assumes extends_extends_subst  $\sigma \tau$  extends_subst  $\sigma \rho$  and
    dom: dom  $\tau = \text{vars\_term } t \cup \text{dom } \sigma$  dom  $\rho = \text{vars\_term } t \cup \text{dom } \sigma$  and
    eq:  $t \cdot \text{subst\_of\_map Var } \rho = t \cdot \text{subst\_of\_map Var } \tau$ 
  shows  $\rho = \tau$ 
proof
  fix x
  consider (a)  $x \in \text{dom } \sigma$  | (b)  $x \in \text{vars\_term } t$  | (c)  $x \notin \text{dom } \tau$  using assms by auto
  then show  $\rho x = \tau x$ 
  proof cases
    case a
    then show ?thesis using extends_unfolding extends_subst_def by auto
  next
  case b
  with eq show ?thesis
  proof (induct t)
    case (Var x)
    with trans[OF dom(1) dom(2)[symmetric]] show ?case
    by (auto simp: subst_of_map_def split: option.splits)
  qed auto
  next
  case c
  then have  $\rho x = \text{None}$   $\tau x = \text{None}$  using dom by auto
  then show ?thesis by simp
  qed
qed

```

```

lemma subsumes_list_alt:
  subsumes_list Ls Ks  $\sigma \longleftrightarrow$  subsumes_list_modulo Ls Ks  $\sigma$ 
proof (induction Ls Ks  $\sigma$  rule: subsumes_list.induct[case_names Nil Cons])
  case (Cons L Ls Ks  $\sigma$ )
  show ?case
  unfolding subsumes_list_modulo_Cons subsumes_list.simps
  proof ((intro bex_cong[OF refl] ext iffI; elim exE conjE), goal_cases LR RL)
  case (LR K)
  show ?case
  by (insert LR; cases K; cases L; auto simp: Cons.IH split: option.splits dest!: match_term_list_sound)
  next
  case (RL K  $\tau$ )
  then show ?case
  proof (cases match_term_list [(atm_of L, atm_of K)]  $\sigma$ )
  case None
  with RL show ?thesis
  by (auto simp: Cons.IH dest!: match_term_list_complete)
  next
  case (Some  $\tau'$ )
  with RL show ?thesis
  using unique_extends_subst[of  $\sigma \tau \tau'$  atm_of L]
  by (auto simp: Cons.IH dest!: match_term_list_sound)

```

qed
 qed
 qed (auto simp: subsumes_modulo_def subst_cls_def vars_clause_def intro: extends_subst_refl)

lemma *subsumes_subsumes_list*[code_unfold]:
subsumes (mset Ls) (mset Ks) = subsumes_list Ls Ks Map.empty
unfolding *subsumes_list_alt*[of Ls Ks Map.empty]
proof
assume *subsumes (mset Ls) (mset Ks)*
then obtain σ **where** *subst_cls (mset Ls) $\sigma \subseteq \#$ mset Ks* **unfolding** *subsumes_def* **by** *blast*
moreover define τ **where** $\tau = (\lambda x. \text{if } x \in \text{vars_clause } (mset Ls) \text{ then Some } (\sigma x) \text{ else None})$
ultimately show *subsumes_list_modulo Ls Ks Map.empty*
unfolding *subsumes_modulo_def*
by (*subst (asm) same_on_vars_clause*[of σ *subst_of_map* Var τ])
 (*auto intro!*: *exI*[of τ] *simp: subst_of_map_def*[*abs_def*] *split: if_splits*)
 qed (auto simp: subsumes_modulo_def subst_lit_def subsumes_def)

lemma *strictly_subsumes_subsumes_list*[code_unfold]:
strictly_subsumes (mset Ls) (mset Ks) =
(subsumes_list Ls Ks Map.empty \wedge \neg subsumes_list Ks Ls Map.empty)
unfolding *strictly_subsumes_def* *subsumes_subsumes_list* **by** *simp*

lemma *subsumes_list_filterD*: *subsumes_list Ls (filter P Ks) $\sigma \implies$ subsumes_list Ls Ks σ*
proof (*induction Ls arbitrary: Ks σ*)
case (*Cons L Ls*)
from *Cons.premis* **show** ?*case*
by (*auto dest!*: *Cons.IH* *simp: filter_remove1*[*symmetric*] *split: option.splits*)
 qed *simp*

lemma *subsumes_list_filterI*:
assumes *match: ($\bigwedge L K \sigma \tau. L \in \text{set } Ls \implies$*
match_term_list [(atm_of L, atm_of K)] $\sigma = \text{Some } \tau \implies \text{is_pos } L = \text{is_pos } K \implies P K$)
shows *subsumes_list Ls Ks $\sigma \implies$ subsumes_list Ls (filter P Ks) σ*
using *assms* **proof** (*induction Ls Ks σ rule: subsumes_list.induct*[*case_names Nil Cons*])
case (*Cons L Ls Ks σ*)
from *Cons.premis* **show** ?*case*
unfolding *subsumes_list.simps* *set_filter* *bex_simps* *conj_assoc*
by (*elim* *bexE* *conjE*)
 (*rule* *exI*, *rule* *conjI*, *assumption*,
auto split: option.splits simp: filter_remove1[*symmetric*] *intro!*: *Cons.IH*)
 qed *simp*

lemma *subsumes_list_Cons_filter_iff*:
assumes *sorted_wrt: sorted_wrt leq (L # Ls)* **and** *trans: transp leq*
and *match: ($\bigwedge L K \sigma \tau.$*
match_term_list [(atm_of L, atm_of K)] $\sigma = \text{Some } \tau \implies \text{is_pos } L = \text{is_pos } K \implies \text{leq } L K$)
shows *subsumes_list (L # Ls) (filter (leq L) Ks) $\sigma \longleftrightarrow$ subsumes_list (L # Ls) Ks σ*
apply (*rule iffI*[*OF* *subsumes_list_filterD* *subsumes_list_filterI*]; *assumption?*)
unfolding *list.set insert_iff*
apply (*elim* *disjE*)
subgoal by (*auto split: option.splits elim!*: *match*)
subgoal for $L K \sigma \tau$
using *sorted_wrt* **unfolding** *List.sorted_wrt.simps*(2)
apply (*elim* *conjE*)
apply (*drule* *bspec*, *assumption*)
apply (*erule* *transpD*[*OF* *trans*])
apply (*erule* *match*)
by *auto*
 done

definition *leq_head* :: ($'f::\text{linorder}, 'v$) *term* \Rightarrow ($'f, 'v$) *term* \Rightarrow *bool* **where**
leq_head t u = (case (root t, root u) of
 (*None, _*) \Rightarrow *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
  unfolding transp_def leq_lit_def leq_head_def by (force split: option.splits literal.splits)

```

```

lemma reflp_leq_lit[simp]: reflp_on leq_lit A
  unfolding reflp_on_def leq_lit_def leq_head_def by (auto split: option.splits literal.splits)

```

```

lemma total_leq_lit[simp]: total_on leq_lit A
  unfolding total_on_def leq_lit_def leq_head_def by (auto split: option.splits literal.splits)

```

```

lemma leq_head_subst[simp]: leq_head t (t · σ)
  by (induct t) (auto simp: leq_head_def)

```

```

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
  by (cases L; cases K)
  (auto simp: leq_lit_def dest!: match_term_list_sound split: option.splits)

```

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 ⇒ subsumes_list Ls Ks σ = subsumes_list_filter Ls Ks σ

```

```

proof (induction Ls arbitrary: Ks σ)
  case (Cons L Ls)
  from Cons.prem1 have subsumes_list (L # Ls) Ks σ = subsumes_list (L # Ls) (filter (leq_lit L) Ks) σ
  by (intro subsumes_list_Cons_filter_iff[symmetric]) (auto dest: leq_lit_match)
  also have subsumes_list (L # Ls) (filter (leq_lit L) Ks) σ = subsumes_list_filter (L # Ls) Ks σ
  using Cons.prem1 by (auto simp: Cons.IH split: option.splits)
  finally show ?case .
qed simp

```

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 ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a list where
  quicksort _ [] = []
| quicksort R (x # xs) =
  quicksort R (filter (λy. R y x) xs) @ [x] @ quicksort R (filter (λy. ¬ R y x) xs)

```

We can easily show that this QuickSort is correct:

```

theorem mset_quicksort [simp]: mset (quicksort R xs) = mset xs
  by (induction R xs rule: quicksort.induct) simp_all

```

```

corollary set_quicksort [simp]: set (quicksort R xs) = set xs
  by (induction R xs rule: quicksort.induct) auto

theorem sorted_wrt_quicksort:
  assumes transp R and total_on R (set xs) and reflp_on R (set xs)
  shows sorted_wrt R (quicksort R xs)
using assms
proof (induction R xs rule: quicksort.induct)
  case (2 R x xs)
  have total: R a b if ¬ R b a a ∈ set (x#xs) b ∈ set (x#xs) for a b
    using 2.prem1 that unfolding total_on_def reflp_on_def by (cases a = b) auto

  have sorted_wrt R (quicksort R (filter (λy. R y x) xs))
    sorted_wrt R (quicksort R (filter (λy. ¬ R y x) xs))
  using 2.prem2 by (intro 2.IH; auto simp: total_on_def reflp_on_def)+
  then show ?case
    by (auto simp: sorted_wrt_append ‹transp R›
      intro: transpD[OF ‹transp R›] dest!: total)
qed auto

```

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 σ = subsumes_list_filter (quicksort leq_lit Ls) Ks σ
  by (rule trans[OF box_equals[OF _ subsumes_list_alt[symmetric] subsumes_list_alt[symmetric]]
    sorted_wrt_subsumes_list_subsumes_list_filter]
    (auto simp: sorted_wrt_quicksort))

```

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 = λ_. {#} and
  subst_atm = (·) and
  id_subst = Var :: _ ⇒ (′f :: {weighted, compare_order}, nat) term and
  comp_subst = (∘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_clss = RP.remdups_clss
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 insert_key = linorder.insert_key (le_of_comp compare)
by (unfold_locales)
  (auto simp: less_kbo_subst is_ground_atm_ground less_kbo_less intro: ground_less_less_kbo)

```

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*: $remove1_mset\ a\ A \subseteq\# B \iff A \subseteq\# add_mset\ a\ B$
by (metis add_mset_add_single subset_eq_diff_conv)

lemma *Bex_cong*: $(\bigwedge b. b \in B \implies P\ b = Q\ b) \implies Bex\ B\ P = Bex\ B\ Q$
by auto

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 \implies False
 | Some $\sigma \implies$ subsumes_list (remove1 L' D) C σ)
 else False))

unfolding *RP.is_reducible_lit_def* *subsumes_list_alt* *subsumes_modulo_def*
apply (rule *Bex_cong*)+
subgoal for D L'
apply (split if_splits option.splits)+
apply safe
subgoal for σ
using *term_subst_eq*[of _ subst_of_map Var ($\lambda x. if\ x \in vars_lit\ L'\ then\ Some\ (\sigma\ x)\ else\ None$) σ]
by (cases L; cases L';
 auto simp add: subst_lit_def subst_of_map_def)

```

    dest!: match_term_list_complete[of _ _  $\lambda x$ . if  $x \in \text{vars\_lit } L'$  then  $\text{Some } (\sigma x)$  else  $\text{None}$ ]
  subgoal for  $\sigma$ 
    using term_subst_eq[of _ subst_of_map Var ( $\lambda x$ . if  $x \in \text{vars\_lit } L'$  then  $\text{Some } (\sigma x)$  else  $\text{None}$ )  $\sigma$ ]
    by (cases L; cases L';
        auto simp add: subst_lit_def subst_of_map_def
        dest!: match_term_list_complete[of _ _  $\lambda x$ . if  $x \in \text{vars\_lit } L'$  then  $\text{Some } (\sigma x)$  else  $\text{None}$ ])
  subgoal for  $\sigma$ 
    by (cases L; cases L'; simp add: subst_lit_def)
  subgoal for  $\sigma$ 
    by (cases L; cases L'; simp add: subst_lit_def)
  subgoal for  $\sigma \tau$ 
    using same_on_vars_clause[of mset (remove1 L' D) subst_of_map Var
      ( $\lambda x$ . if  $x \in \text{vars\_clause } (\text{remove1\_mset } L' (\text{mset } D)) \cup \text{dom } \sigma$  then  $\text{Some } (\tau x)$  else  $\text{None}$ )  $\tau$ ]
    apply (cases L; cases L'; auto simp add: subst_lit_def dom_def subst_of_map_def
      dest!: match_term_list_sound split: option.splits if_splits
      intro!: exI[of _  $\lambda x$ . if  $x \in \text{vars\_clause } (\text{remove1\_mset } L' (\text{mset } D)) \cup \text{dom } \sigma$  then  $\text{Some } (\tau x)$  else  $\text{None}$ ])
    by (auto 0 4 simp: extends_subst_def subst_of_map_def split: option.splits dest!: term_subst_eq_rev)
  subgoal for  $\sigma \tau$ 
    by (cases L; cases L'; auto simp add: subst_lit_def subst_of_map_def extends_subst_def
      dest!: match_term_list_sound intro!: exI[of _ subst_of_map Var  $\tau$ ] term_subst_eq)
  subgoal for  $\sigma \tau$ 
    using same_on_vars_clause[of mset (remove1 L' D) subst_of_map Var
      ( $\lambda x$ . if  $x \in \text{vars\_clause } (\text{remove1\_mset } L' (\text{mset } D)) \cup \text{dom } \sigma$  then  $\text{Some } (\tau x)$  else  $\text{None}$ )  $\tau$ ]
    apply (cases L; cases L'; auto simp add: subst_lit_def dom_def subst_of_map_def
      dest!: match_term_list_sound split: option.splits if_splits
      intro!: exI[of _  $\lambda x$ . if  $x \in \text{vars\_clause } (\text{remove1\_mset } L' (\text{mset } D)) \cup \text{dom } \sigma$  then  $\text{Some } (\tau x)$  else  $\text{None}$ ])
    by (auto 0 4 simp: extends_subst_def subst_of_map_def split: option.splits dest!: term_subst_eq_rev)
  subgoal for  $\sigma \tau$ 
    by (cases L; cases L'; auto simp add: subst_lit_def subst_of_map_def extends_subst_def
      dest!: match_term_list_sound intro!: exI[of _ subst_of_map Var  $\tau$ ] term_subst_eq)
  subgoal for  $\sigma \tau$ 
    by (cases L; cases L'; simp add: subst_lit_def)
  subgoal for  $\sigma \tau$ 
    by (cases L; cases L'; simp add: subst_lit_def)
done
done

```

declare

```

Pairs_def[folded sorted_list_of_set_def, code]
linorder.sorted_list_of_set_sort_remdups[OF class_linorder_compare,
  folded sorted_list_of_set_def sort_key_def, code]
linorder.sort_key_def[OF class_linorder_compare, folded sort_key_def insert_key_def, code]
linorder.insert_key.simps[OF class_linorder_compare, folded insert_key_def, code]

```

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* =

```

( $w = \text{Suc} \circ \text{prod\_encode}$ ,  $w0 = 1$ ,  $\text{pr\_strict} = \lambda(f, n) (g, m). f > g \vee f = g \wedge n > m$ ,  $\text{least} = \lambda n. n = 0$ ,  $\text{scf} =$ 
 $\lambda\_.. 1$ )

```

instance

```

by (intro_classes, unfold_locales)
(auto simp: weights_nat_def SN_iff_wf async.simps irreflp_def prod_encode_def
  intro!: wf_subset[OF wf_lex_prod])

```

end

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

```

prover N = (case deterministic_RP (St0 N 0) of
  None  $\Rightarrow$  True
  | Some R  $\Rightarrow$  []  $\notin$  set R)

```

theorem *prover_complete_refutation*: $prover\ N \longleftrightarrow\ satisfiable\ (RP.grounded_N0\ N)$
unfolding *prover_def St0_def*
using *RP.deterministic_RP_complete[of N 0] RP.deterministic_RP_refutation[of N 0]*
by (*force simp: grounding_of_cls_def grounding_of_cls_def ex_ground_subst*
split: option.splits if_splits)

definition *string_literal_of_nat* :: $nat \Rightarrow String.literal$ **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* $\equiv Fun\ 42$

abbreviation *a* $\equiv Fun\ 0\ []$

abbreviation *b* $\equiv Fun\ 1\ []$

abbreviation *c* $\equiv Fun\ 2\ []$

abbreviation *X* $\equiv Var\ 0$

abbreviation *Y* $\equiv Var\ 1$

abbreviation *Z* $\equiv Var\ 2$

value *prover*
 $((([Neg\ (p[X,Y,Z]),\ Pos\ (p[Y,Z,X])],\ 1),$
 $([Pos\ (p[c,a,b])],\ 1),$
 $([Neg\ (p[b,c,a])],\ 1)]$
 $::\ ((nat,\ nat)\ Term.term\ literal\ list\ \times\ nat)\ list)$

value *prover*
 $((([Pos\ (p[X,Y])],\ 1),\ ([Neg\ (p[X,X])],\ 1)]$
 $::\ ((nat,\ nat)\ Term.term\ literal\ list\ \times\ nat)\ list)$

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

definition *mk_MSC015_1* :: $nat \Rightarrow ((nat,\ nat)\ Term.term\ literal\ list\ \times\ nat)\ list$ **where**
mk_MSC015_1 $n =$
 $(let$
 $\quad init = ([Pos\ (p\ (replicate\ n\ a))],\ 1);$
 $\quad rules = map\ (\lambda i.\ ([Neg\ (p\ (map\ Var\ [0\ ..<\ n - i - 1]\ @\ a\ \# replicate\ i\ b)),$
 $\quad Pos\ (p\ (map\ Var\ [0\ ..<\ n - i - 1]\ @\ b\ \# replicate\ i\ a))],\ 1))\ [0\ ..<\ n];$
 $\quad goal = ([Neg\ (p\ (replicate\ n\ b))],\ 1)$
 $\quad in\ init\ \# rules\ @\ [goal])$

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

$p\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a\ a$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11\ x12\ x13.$
 $\quad \neg\ p\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11\ x12\ x13\ a\ \vee$
 $\quad p\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11\ x12\ x13\ b)$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11\ x12.$
 $\quad \neg\ p\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11\ x12\ a\ b\ \vee$
 $\quad p\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11\ x12\ b\ a)$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11.$
 $\quad \neg\ p\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11\ a\ b\ b\ \vee$
 $\quad p\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ x11\ b\ a\ a)$
 $(\forall x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10.$
 $\quad \neg\ p\ x1\ x2\ x3\ x4\ x5\ x6\ x7\ x8\ x9\ x10\ a\ b\ b\ b\ \vee$


```

  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 b ∨
  p x1 x2 x3 x4 x5 x6 x7 x8 x9 b a 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 b b ∨
  p x1 x2 x3 x4 x5 x6 x7 x8 b a a 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 b b b ∨
  p x1 x2 x3 x4 x5 x6 x7 b a a 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 b b ∨
  p x1 x2 x3 x4 x5 x6 b a a 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 b b b ∨
  p x1 x2 x3 x4 x5 b a a 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 b b ∨
  p x1 x2 x3 x4 b a 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 b b b ∨
  p x1 x2 x3 b a 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 b b b b ∨
  p x1 x2 b a 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 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 b b b b ∨
  p b a 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 b b

```

shows *False*
using *assms bymetis*

end