DTU Library

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

**Schlichtkrull, Anders; Blanchette, Jasmin Christian; Traytel, Dmitriy**

*Published in:*
Archive of Formal Proofs

*Publication date:*
2018

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

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

Anders Schlichtkrull, Jasmin Christian Blanchette, and Dmitriy Traytel

November 29, 2018

**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

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.

Due to a dependency on the Knuth–Bendix order from the IsaFoR library, which has not yet been moved to the AFP, the final part of our development is currently hosted in the IsaFoL repository.[1]

[1] https://bitbucket.org/isafol/isafol/src/master/Functional_Ordered_Resolution_Prover/

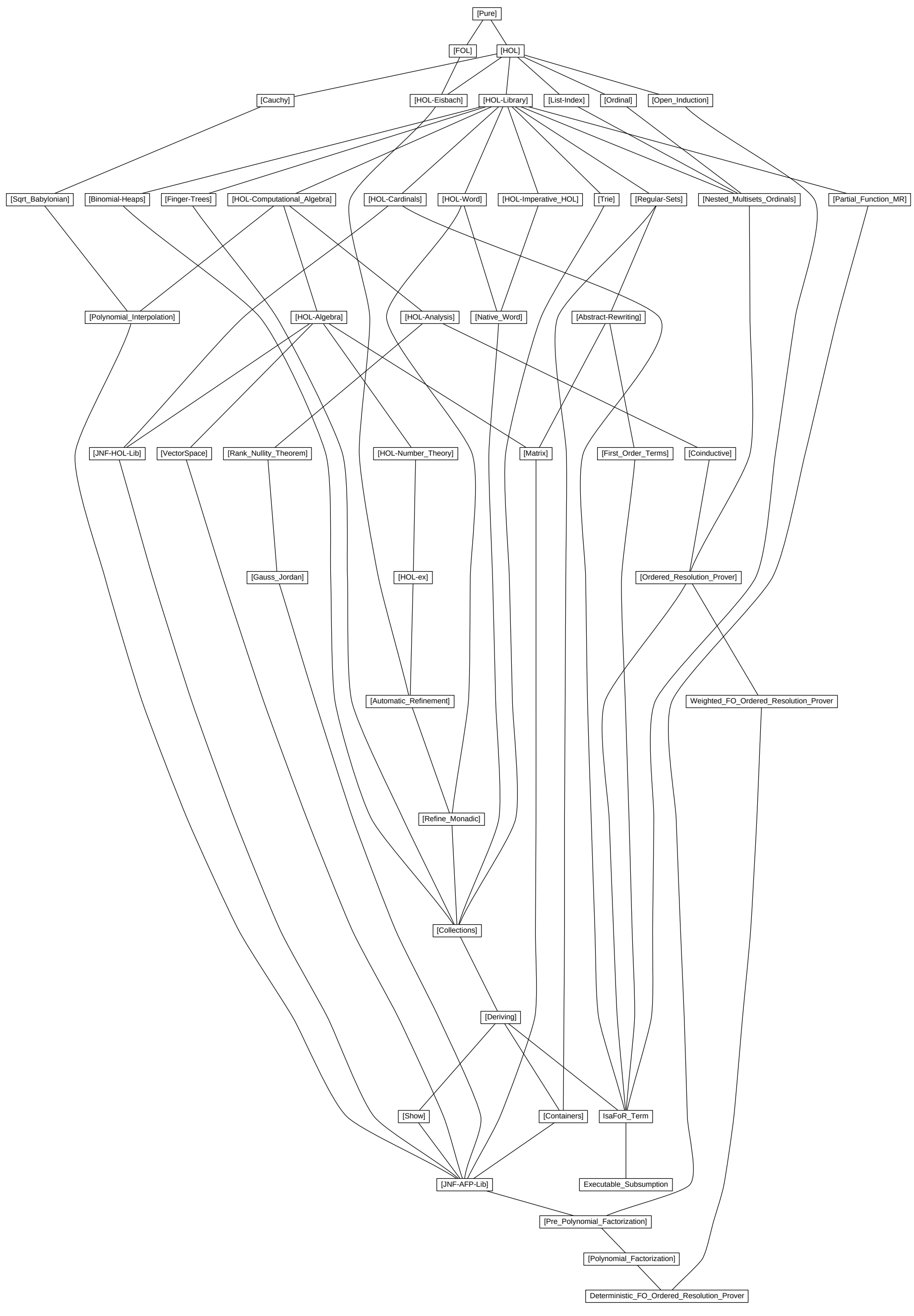


Figure 1: Theory dependency graph

# 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
```

## 2.1 Library

```
lemma ldrop_Suc_conv_ltl: ldrop (enat (Suc k)) xs = ltl (ldrop (enat k) xs)
  by (metis eSuc_enat ldrop_eSuc_conv_ltl)


lemma lhd_ldrop':
  assumes enat k < llength xs
  shows lhd (ldrop (enat k) xs) = lnth xs k
  using assms by (simp add: lhd_ldrop)


lemma filter_mset_empty_if_finite_and_filter_set_empty:
  assumes
    {x ∈ X. P x} = {} and
    finite X
  shows {#x ∈# mset_set X. P x#} = {#}
proof –
  have empty_empty: ⋀Y. set_mset Y = {} ⟹ Y = {#}
    by auto
  from assms have set_mset {#x ∈# mset_set X. P x#} = {}
    by auto
  then show ?thesis
    by (rule empty_empty)
qed


lemma inf_chain_ltl_chain: chain R xs ⟹ llength xs = ∞ ⟹ chain R (ltl xs)
  unfolding chain.simps[of R xs] llength_eq_infty_conv_lfinite
  by (metis lfinite_code(1) lfinite_ltl llist.sel(3))


lemma inf_chain_ldrop_chain:
  assumes
    chain: chain R xs and
    inf: ¬ lfinite xs
  shows chain R (ldrop (enat k) xs)
proof (induction k)
  case 0
  then show ?case
    using zero_enat_def chain by auto
next
  case (Suc k)
  have llength (ldrop (enat k) xs) = ∞
    using inf by (simp add: not_lfinite_llength)
  with Suc have chain R (ltl (ldrop (enat k) xs))
    using inf_chain_ltl_chain[of R (ldrop (enat k) xs)] by auto
  then show ?case
    using ldrop_Suc_conv_ltl[of k xs] by auto
qed
```

## 2.2 Prover

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

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

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

**abbreviation** *clss_of_wstate* :: *'a wstate* ⇒ *'a clause set* **where**
  *clss_of_wstate St* ≡ *clss_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* ⇒ *'a clause set* **where**
  *grounding_of_wstate St* ≡ *grounding_of_state* (*state_of_wstate St*)

**abbreviation** *Liminf_wstate* :: *'a wstate llist* ⇒ *'a state* **where**
  *Liminf_wstate Sts* ≡ *Liminf_state* (*lmap state_of_wstate Sts*)

**lemma** *timestamp_le_weight*: *n* ≤ *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* ⇒ *'a wstate* ⇒ *bool* (**infix** $\leadsto_w$ *50*) **where**

```
  tautology_deletion: Neg A ∈# C ⟹ Pos A ∈# C ⟹ (N + {#(C, i)#}, P, Q, n) ⇝w (N, P, Q, n)
| forward_subsumption: D ∈# image_mset fst (P + Q) ⟹ subsumes D C ⟹
    (N + {#(C, i)#}, P, Q, n) ⇝w (N, P, Q, n)
| backward_subsumption_P: D ∈# image_mset fst N ⟹ C ∈# image_mset fst P ⟹
    strictly_subsumes D C ⟹ (N, P, Q, n) ⇝w (N, {#(E, k) ∈# P. E ≠ C#}, Q, n)
| backward_subsumption_Q: D ∈# image_mset fst N ⟹ strictly_subsumes D C ⟹
    (N, P, Q + {#(C, i)#}, n) ⇝w (N, P, Q, n)
| forward_reduction: D + {#L'#} ∈# image_mset fst (P + Q) ⟹ − L = L' ·l σ ⟹ D · σ ⊆# C ⟹
    (N + {#(C + {#L#}, i)#}, P, Q, n) ⇝w (N + {#(C, i)#}, P, Q, n)
| backward_reduction_P: D + {#L'#} ∈# image_mset fst N ⟹ − L = L' ·l σ ⟹ D · σ ⊆# C ⟹
    (∀j. (C + {#L#}, j) ∈# P ⟶ j ≤ i) ⟹
    (N, P + {#(C + {#L#}, i)#}, Q, n) ⇝w (N, P + {#(C, i)#}, Q, n)
| backward_reduction_Q: D + {#L'#} ∈# image_mset fst N ⟹ − L = L' ·l σ ⟹ D · σ ⊆# C ⟹
    (N, P, Q + {#(C + {#L#}, i)#}, n) ⇝w (N, P + {#(C, i)#}, Q, n)
| clause_processing: (N + {#(C, i)#}, P, Q, n) ⇝w (N, P + {#(C, i)#}, Q, n)
| inference_computation: (∀(D, j) ∈# P. weight (C, i) ≤ weight (D, j)) ⟹
    N = mset_set ((λD. (D, n)) ` concls_of
      (inference_system.inferences_between (ord_FO_Γ S) (set_mset (image_mset fst Q)) C)) ⟹
    ({#}, P + {#(C, i)#}, Q, n) ⇝w (N, {#(D, j) ∈# P. D ≠ C#}, Q + {#(C, i)#}, Suc n)

lemma weighted_RP_imp_RP: St ⇝w St' ⟹ state_of_wstate St ⇝ 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 _ _ _ _ (⇝), 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 _ _ _ _ (⇝), OF _ _
           RP.inference_computation[of fst ` set_mset N fst ` set_mset Q C
             fst ` set_mset P − {C}]],
         use inference_computation(2) finite_ord_FO_resolution_inferences_between in
           ⟨auto simp: comp_def image_comp inference_system.inferences_between_def⟩)
qed (use RP.intros in simp_all)

lemma final_weighted_RP: ¬ ({#}, {#}, Q, n) ⇝w St
  by (auto elim: weighted_RP.cases)

context
  fixes
    Sts :: 'a wstate llist
  assumes
    full_deriv: full_chain (⇝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))
  unfolding clss_of_state_def by (cases lhd Sts) auto

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

lemma deriv_RP: chain (⇝) (lmap state_of_wstate Sts)
  using deriv weighted_RP_imp_RP by (metis chain_lmap)

lemma finite_Sts0_RP: finite (clss_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*:
$St \leadsto_w St' \Longrightarrow I \models s$ *grounding_of_wstate* $St' \longleftrightarrow I \models s$ *grounding_of_wstate St*
**using** *RP_model Sts_thms weighted_RP_imp_RP* **by** (*simp only*: *comp_def*)

**abbreviation** *S_gQ* :: *'a clause* ⇒ *'a clause* **where**
*S_gQ* ≡ *S_Q* (*lmap state_of_wstate Sts*)

**interpretation** *sq*: *selection S_gQ*
**unfolding** *S_Q_def*[*OF deriv_RP empty_Q0_RP*]
**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** {#} ∈ *clss_of_state* (*Liminf_wstate Sts*)
**shows** ¬ *satisfiable* (*grounding_of_wstate* (*lhd Sts*))
**by** (*rule RP_sound*[*OF deriv_RP empty_Q0_RP assms*, *unfolded lhd_lmap_Sts*])

**abbreviation** *RP_filtered_measure* :: (*'a wclause* ⇒ *bool*) ⇒ *'a wstate* ⇒ *nat* × *nat* × *nat* **where**
*RP_filtered_measure* ≡ λ*p* (*N*, *P*, *Q*, *n*).
(*sum_mset* (*image_mset* (λ(*C*, *i*). *Suc* (*size C*)) {#*Di* ∈# *N* + *P* + *Q*. *p Di*#}),
*size* {#*Di* ∈# *N*. *p Di*#}, *size* {#*Di* ∈# *P*. *p Di*#})

**abbreviation** *RP_combined_measure* :: *nat* ⇒ *'a wstate* ⇒ *nat* × (*nat* × *nat* × *nat*) × (*nat* × *nat* × *nat*) **where**
*RP_combined_measure* ≡ λ*w St*.
(*w* + *1* − *n_of_wstate St*, *RP_filtered_measure* (λ(*C*, *i*). *i* ≤ *w*) *St*,
*RP_filtered_measure* (λ*Ci*. *True*) *St*)

**abbreviation** (*input*) *RP_filtered_relation* :: ((*nat* × *nat* × *nat*) × (*nat* × *nat* × *nat*)) *set* **where**
*RP_filtered_relation* ≡ *natLess* <∗*lex*∗> *natLess* <∗*lex*∗> *natLess*

**abbreviation** (*input*) *RP_combined_relation* :: ((*nat* × ((*nat* × *nat* × *nat*) × (*nat* × *nat* × *nat*))) ×
(*nat* × ((*nat* × *nat* × *nat*) × (*nat* × *nat* × *nat*)))) *set* **where**
*RP_combined_relation* ≡ *natLess* <∗*lex*∗> *RP_filtered_relation* <∗*lex*∗> *RP_filtered_relation*

**abbreviation** (*fst3* :: *'b* ∗ *'c* ∗ *'d* ⇒ *'b*) ≡ *fst*
**abbreviation** (*snd3* :: *'b* ∗ *'c* ∗ *'d* ⇒ *'c*) ≡ λ*x*. *fst* (*snd x*)
**abbreviation** (*trd3* :: *'b* ∗ *'c* ∗ *'d* ⇒ *'d*) ≡ λ*x*. *snd* (*snd x*)

**lemma**
*wf_RP_filtered_relation*: *wf RP_filtered_relation* **and**
*wf_RP_combined_relation*: *wf RP_combined_relation*
**unfolding** *natLess_def* **using** *wf_less wf_mult* **by** *auto*

```
lemma multiset_sum_of_Suc_f_monotone: N ⊂# M ⟹ (∑x ∈# N. Suc (f x)) < (∑x ∈# M. Suc (f x))
proof (induction N arbitrary: M)
  case empty
  then obtain y where y ∈# M
    by force
  then have (∑x ∈# M. 1) = (∑x ∈# M − {#y#} + {#y#}. 1)
    by auto
  also have ... = (∑x ∈# M − {#y#}. 1) + (∑x ∈# {#y#}. 1)
    by (metis image_mset_union sum_mset.union)
  also have ... > (0 :: nat)
    by auto
  finally have 0 < (∑x ∈# M. Suc (f x))
    by (fastforce intro: gr_zeroI)
  then show ?case
    using empty by auto
next
  case (add x N)
  from this(2) have (∑y ∈# N. Suc (f y)) < (∑y ∈# M − {#x#}. Suc (f y))
    using add(1)[of M − {#x#}] by (simp add: insert_union_subset_iff)
  moreover have add_mset x (remove1_mset x M) = M
    by (meson add.prems add_mset_remove_trivial_If mset_subset_insertD)
  ultimately show ?case
    by (metis (no_types) add.commute add_less_cancel_right sum_mset.insert)
qed

lemma multiset_sum_monotone_f':
  assumes CC ⊂# DD
  shows (∑(C, i) ∈# CC. Suc (f C)) < (∑(C, i) ∈# DD. Suc (f C))
  using multiset_sum_of_Suc_f_monotone[OF assms, of f ∘ fst]
  by (metis (mono_tags) comp_apply image_mset_cong2 split_beta)

lemma filter_mset_strict_subset:
  assumes x ∈# M and ¬ p x
  shows {#y ∈# M. p y#} ⊂# M
proof −
  have subseteq: {#E ∈# M. p E#} ⊆# M
    by auto
  have count {#E ∈# M. p E#} x = 0
    using assms by auto
  moreover have 0 < count M x
    using assms by auto
  ultimately have lt_count: count {#y ∈# M. p y#} x < count M x
    by auto
  then show ?thesis
    using subseteq by (metis less_not_refl2 subset_mset.le_neq_trans)
qed

lemma weighted_RP_measure_decreasing_N:
  assumes St ↝w St' and (C, l) ∈# wN_of_wstate St
  shows (RP_filtered_measure (λCi. True) St', RP_filtered_measure (λCi. True) St)
    ∈ RP_filtered_relation
using assms proof (induction rule: weighted_RP.induct)
  case (backward_subsumption_P D N C' P Q n)
  then obtain i' where (C', i') ∈# P
    by auto
  then have {#(E, k) ∈# P. E ≠ C'#} ⊂# P
    using filter_mset_strict_subset[of (C', i') P λX. ¬fst X = C']
    by (metis (mono_tags, lifting) filter_mset_cong fst_conv prod.case_eq_if)
  then have (∑(C, i) ∈# {#(E, k) ∈# P. E ≠ C'#}. Suc (size C)) < (∑(C, i) ∈# P. Suc (size C))
    using multiset_sum_monotone_f'[of {#(E, k) ∈# P. E ≠ C'#} P size] by metis
  then show ?case
    unfolding natLess_def by auto
```

**qed** (*auto simp*: *natLess_def*)

**lemma** *weighted_RP_measure_decreasing_P*:
**assumes** *St* ↝$_w$ *St'* **and** (*C*, *i*) ∈# *wP_of_wstate St*
**shows** (*RP_combined_measure* (*weight* (*C*, *i*)) *St'*, *RP_combined_measure* (*weight* (*C*, *i*)) *St*)
∈ *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*) ∈# *P*. *E* ≠ *C'*#}
**define** *St'* **where** *St'* = (*N*, *P'*, *Q*, *n*)

**from** *backward_subsumption_P* **obtain** *i'* **where** (*C'*, *i'*) ∈# *P*
**by** *auto*
**then have** *P'_sub_P*: *P'* ⊂# *P*
**unfolding** *P'_def* **using** *filter_mset_strict_subset*[*of* (*C'*, *i'*) *P* λ*Dj*. *fst Dj* ≠ *C'*]
**by** (*metis* (*no_types*, *lifting*) *filter_mset_cong fst_conv prod.case_eq_if*)

**have** *P'_subeq_P_filter*:
{#(*Ca*, *ia*) ∈# *P'*. *ia* ≤ *weight* (*C*, *i*)#} ⊆# {#(*Ca*, *ia*) ∈# *P*. *ia* ≤ *weight* (*C*, *i*)#}
**using** *P'_sub_P* **by** (*auto intro*: *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** (∑(*C*, *i*) ∈# {#(*Ca*, *ia*) ∈# *P'*. *ia* ≤ *weight* (*C*, *i*)#}. *Suc* (*size C*))
≤ (∑*x* ∈# {#(*Ca*, *ia*) ∈# *P*. *ia* ≤ *weight* (*C*, *i*)#}. *case x of* (*C*, *i*) ⇒ *Suc* (*size C*))
**using** *P'_subeq_P_filter* **by** (*rule sum_image_mset_mono*)
**then have** *fst3* (*snd3* (*RP_combined_measure* (*weight* (*C*, *i*)) *St'*))
≤ *fst3* (*snd3* (*RP_combined_measure* (*weight* (*C*, *i*)) *St*))
**unfolding** *St'_def St_def* **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* **by** *auto*
**moreover from** *P'_subeq_P_filter* **have** *size* {#(*Ca*, *ia*) ∈# *P'*. *ia* ≤ *weight* (*C*, *i*)#}
≤ *size* {#(*Ca*, *ia*) ∈# *P*. *ia* ≤ *weight* (*C*, *i*)#}
**by** (*simp add*: *size_mset_mono*)
**then have** *trd3* (*snd3* (*RP_combined_measure* (*weight* (*C*, *i*)) *St'*))
≤ *trd3* (*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** (∑(*C*, *i*) ∈# *P'*. *Suc* (*size C*)) < (∑(*C*, *i*) ∈# *P*. *Suc* (*size C*))
**using** *multiset_sum_monotone_f'*[*of* {#(*E*, *k*) ∈# *P*. *E* ≠ *C'*#} *P size*] **unfolding** *P'_def* **by** *metis*
**then have** *fst3* (*trd3* (*RP_combined_measure* (*weight* (*C*, *i*)) *St'*))
< *fst3* (*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* − *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 σ C′ P i′ Q n)
 show ?case
 proof (cases C = C′ + {#L#})
  case True_outer: True
  then have C_i_in: (C, i) ∈# P + {#(C, i′)#}
    using backward_reduction_P by auto
  then have max: ⋀k. (C, k) ∈# P + {#(C, i′)#} ⟹ k ≤ i′
    using backward_reduction_P unfolding True_outer[symmetric] by auto
  then have count (P + {#(C, i′)#}) (C, i′) ≥ 1
```

```
      by auto
    moreover
    {
      assume asm: count (P + {#(C, i')#}) (C, i') = 1
      then have nin_P: (C, i') ∉# 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 ∀ j. (C, j) ∈# P + {#(C, i')#} ⟶ 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 ↝w St' (C, j) ∈# wP_of_wstate St' and
    (C, i) ∈# wP_of_wstate St and
    ∀ k. (C, k) ∈# wP_of_wstate St ⟶ i ≤ k
  shows (C, i) ∈# 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) ∈# wP_of_wstate (lnth Sts k) and
    (C, j) ∈# wP_of_wstate (lnth Sts (Suc k)) and
    ∀ j. (C, j) ∈# wP_of_wstate (lnth Sts k) ⟶ i ≤ j
  shows (C, i) ∈# wP_of_wstate (lnth Sts (Suc k))
  using deriv assms chain_lnth_rel preserve_min_P by metis

lemma in_lnth_in_Supremum_ldrop:
  assumes i < llength xs and x ∈# (lnth xs i)
  shows x ∈ Sup_llist (lmap set_mset (ldrop (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 ∈ Liminf_llist (lmap P_of_state (lmap state_of_wstate Sts))
  shows ∃ i. (C, i) ∈ Liminf_llist (lmap (set_mset ∘ wP_of_wstate) Sts)
proof -
```

```
obtain t_C where t_C_p:
  enat t_C < llength Sts
  ⋀t. t_C ≤ t ⟹ t < llength Sts ⟹ C ∈ P_of_state (state_of_wstate (lnth Sts t))
  using assms unfolding Liminf_llist_def by auto
then obtain i where i_p:
  (C, i) ∈# wP_of_wstate (lnth Sts t_C)
  using t_C_p by (cases lnth Sts t_C) force

have Ci_in_nth_wP: ∃ i. (C, i) ∈# wP_of_wstate (lnth Sts (t_C + t)) if t_C + t < llength Sts
  for t
  using that t_C_p(2)[of t_C + _] by (cases lnth Sts (t_C + t)) force

define in_Sup_wP :: nat ⇒ bool where
  in_Sup_wP = (λi. (C, i) ∈ Sup_llist (lmap (set_mset ∘ wP_of_wstate) (ldrop t_C Sts)))

have 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: is_least in_Sup_wP j
  unfolding in_Sup_wP_def[symmetric] using least_exists by metis
then have ∀ i. (C, i) ∈ Sup_llist (lmap (set_mset ∘ wP_of_wstate) (ldrop t_C Sts)) ⟶ j ≤ i
  unfolding is_least_def in_Sup_wP_def using not_less by blast
then have j_smallest:
  ⋀i t. enat (t_C + t) < llength Sts ⟹ (C, i) ∈# wP_of_wstate (lnth Sts (t_C + t)) ⟹ j ≤ 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 ∃ t_Cj. t_Cj < llength (ldrop (enat t_C) Sts)
  ∧ (C, j) ∈# wP_of_wstate (lnth (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) ∈# wP_of_wstate (lnth Sts (t_C + t_Cj))
  enat (t_C + t_Cj) < 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 < llength Sts ⟹ (C,j) ∈# wP_of_wstate (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 any_Ck_in_wP: j ≤ k if (C, k) ∈# wP_of_wstate (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) ∈# wP_of_wstate (lnth Sts (t_C + t_Cj + t))
    by (metis (no_types, hide_lams) Suc_ile_eq add.commute add_Suc_right less_imp_le)
  moreover have C ∈ P_of_state (state_of_wstate (lnth Sts (Suc (t_C + t_Cj + t))))
    using t_C_p(2) Suc.prems by auto
  then have ∃ k. (C, k) ∈# wP_of_wstate (lnth Sts (Suc (t_C + t_Cj + t)))
    by (smt Suc.prems Ci_in_nth_wP add.commute add.left_commute add_Suc_right enat_ord_code(4))
  ultimately have (C, j) ∈# wP_of_wstate (lnth Sts (Suc (t_C + t_Cj + t)))
    using preserve_min_P_Sts Cj_in_wP any_Ck_in_wP Suc.prems by force
  then have (C, j) ∈# lnth (lmap wP_of_wstate Sts) (Suc (t_C + t_Cj + t))
    using Suc.prems by auto
  then show ?case
    by (smt Suc.prems add.commute add_Suc_right lnth_lmap)
qed
then have (⋀t. t_C + t_Cj ≤ t ⟹ t < llength (lmap (set_mset ∘ wP_of_wstate) Sts) ⟹
  (C, j) ∈# 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) ∈ Liminf_llist (lmap (set_mset ∘ wP_of_wstate) Sts)
    unfolding Liminf_llist_def using j_p by auto
  then show ∃ i. (C, i) ∈ Liminf_llist (lmap (set_mset ∘ wP_of_wstate) Sts)
    by auto
qed

lemma lfinite_not_LNil_nth_llast:
  assumes lfinite Sts and Sts ≠ LNil
  shows ∃ i < llength Sts. lnth Sts i = llast Sts ∧ (∀ j < llength Sts. j ≤ 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 ∧ lnth xs i = llast xs ∧ (∀ j < llength xs. j ≤ 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_lnull lnth_Suc_LCons)
    moreover from i_p have ∀ j < llength (LCons x xs). j ≤ 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_finite:
  assumes fin: lfinite Sts
  shows fair_state_seq (lmap state_of_wstate Sts)
proof (rule ccontr)
  assume unfair: ¬ fair_state_seq (lmap state_of_wstate Sts)

  have no_inf_from_last: ∀ y. ¬ llast Sts ⇝w y
    using fin full_chain_iff_chain[of (⇝w) Sts] full_deriv by auto

  from unfair obtain C where
    C ∈ Liminf_llist (lmap N_of_state (lmap state_of_wstate Sts))
      ∪ 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
    ⋀j. i ≤ j ⟹ enat j < llength Sts ⟹
     C ∈ N_of_state (state_of_wstate (lnth Sts j)) ∪ P_of_state (state_of_wstate (lnth Sts j))
    unfolding Liminf_llist_def by auto

  have C_in_llast:
    C ∈ N_of_state (state_of_wstate (llast Sts)) ∪ P_of_state (state_of_wstate (llast Sts))
  proof −
    obtain l where
      l_p: enat l < llength Sts ∧ lnth Sts l = llast Sts ∧ (∀ j < llength Sts. j ≤ enat l)
      using fin lfinite_not_LNil_nth_llast i_p(1) by fastforce
    then have
      C ∈ N_of_state (state_of_wstate (lnth Sts l)) ∪ 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)) ≠ {}
    then obtain D j where (D, j) ∈# N
      unfolding N_def by (cases llast Sts) auto
    then have llast Sts ⇝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 ∃ St'. llast Sts ⇝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 ∈ P_of_state (state_of_wstate (llast Sts))
      using C_in_llast by auto
    then obtain D j where (D, j) ∈# P
      unfolding P_def by (cases llast Sts) auto
    then have weight (D, j) ∈ weight ` set_mset P
      by auto
    then have ∃ w. is_least (λw. w ∈ (weight ` set_mset P)) w
      using least_exists by auto
    then have ∃ D j. (∀ (D', j') ∈# P. weight (D, j) ≤ weight (D', j')) ∧ (D, j) ∈# P
      using assms linorder_not_less unfolding is_least_def by (auto 6 0)
    then obtain D j where
      min: (∀ (D', j') ∈# P. weight (D, j) ≤ weight (D', j')) and
      Dj_in_p: (D, j) ∈# P
      by auto
    from min have min: (∀ (D', j') ∈# P − {#(D, j)#}. weight (D, j) ≤ weight (D', j'))
      using mset_subset_diff_self[OF Dj_in_p] by auto

    define N' where
      N' = mset_set ((λD'. (D', n)) ` concls_of (inference_system.inferences_between (ord_FO_Γ S)
        (set_mset (image_mset fst Q)) D))

    have llast Sts ⇝w (N', {#(D', j') ∈# P − {#(D, j)#}. D' ≠ 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 ∃ St'. llast Sts ⇝w St'
      by auto
  }
  ultimately have ∃ St'. llast Sts ⇝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 ∈ N_of_state (state_of_wstate St)
  shows ∃ i. (C, i) ∈# 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) ∈# wN_of_wstate St ⟹ C ∈ 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) ∈# wP_of_wstate St ⟹ C ∈ 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) ∈# wQ_of_wstate St ⟹ C ∈ 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 ↝w St' ⟹ n_of_wstate St ≤ n_of_wstate St'
  by (induction rule: weighted_RP.induct) auto

lemma nth_of_wstate_monotonic:
  assumes j < llength Sts and i ≤ j
  shows n_of_wstate (lnth Sts i) ≤ 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)) ≤ 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 ↝w lnth Sts (Suc i)
    using deriv chain_lnth_rel[of (↝w) Sts i] by auto
  then have n_of_wstate (lnth Sts i) ≤ 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: ⋀St St'. P St ⟹ R St St' ⟹ (m St', m St) ∈ mR and
    non_infer_chain: chain R (ldrop (enat k) Sts) and
    inf: llength Sts = ∞ and
    P: ⋀i. P (lnth (ldrop (enat k) Sts) i)
  shows chain (λx y. (x, y) ∈ mR)⁻¹⁻¹ (lmap m (ldrop (enat k) Sts))
proof (rule lnth_rel_chain)
  show ¬ lnull (lmap m (ldrop (enat k) Sts))
    using assms by auto
next
  from inf have ldrop_inf: llength (ldrop (enat k) Sts) = ∞ ∧ ¬ 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 ∧ P St'
      unfolding St_def St'_def using P by auto
    from ldrop_inf have R St 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 St', m St) ∈ mR
      using measure_decreasing P' by auto
    then have (lnth (lmap m (ldrop (enat k) Sts)) (j + 1), lnth (lmap m (ldrop (enat k) Sts)) j)
      ∈ mR
      unfolding St_def St'_def using lnth_lmap
      by (smt enat.distinct(1) enat_add_left_cancel enat_ord_simps(4) inf ldrop_lmap llength_lmap
```

```
        lnth_ldrop plus_enat_simps(3))
  }
  then show ∀ j. enat (j + 1) < llength (lmap m (ldrop (enat k) Sts)) ⟶
    (λx y. (x, y) ∈ mR)⁻¹⁻¹ (lnth (lmap m (ldrop (enat k) Sts)) j)
      (lnth (lmap m (ldrop (enat k) Sts)) (j + 1))
    by blast
qed

theorem weighted_RP_fair: fair_state_seq (lmap state_of_wstate Sts)
proof (rule ccontr)
  assume asm: ¬ fair_state_seq (lmap state_of_wstate Sts)
  then have inff: ¬ lfinite Sts using fair_if_finite
    by auto
  then have inf: llength Sts = ∞
    using llength_eq_infty_conv_lfinite by auto
  from asm obtain C where
    C ∈ Liminf_llist (lmap N_of_state (lmap state_of_wstate Sts))
      ∪ Liminf_llist (lmap P_of_state (lmap state_of_wstate Sts))
    unfolding fair_state_seq_def Liminf_state_def by auto
  then show False
  proof
    assume C ∈ Liminf_llist (lmap N_of_state (lmap state_of_wstate Sts))
    then obtain x where enat x < llength Sts
      ∀ xa. x ≤ xa ∧ enat xa < llength Sts ⟶ C ∈ N_of_state (state_of_wstate (lnth Sts xa))
      unfolding Liminf_llist_def by auto
    then have ∃ k. ∀ j. k ≤ j ⟶ (∃ i. (C, i) ∈# wN_of_wstate (lnth Sts j))
      unfolding Liminf_llist_def by (force simp add: inf N_of_state_state_of_wstate_wN_of_wstate)
    then obtain k where k_p:
      ⋀j. k ≤ j ⟹ ∃ i. (C, i) ∈# wN_of_wstate (lnth Sts j)
      unfolding Liminf_llist_def
      by auto
    have chain_drop_Sts: chain (↝w) (ldrop k Sts)
      using deriv inf inff inf_chain_ldrop_chain by auto
    have in_N_j: ⋀j. ∃ i. (C, i) ∈# wN_of_wstate (lnth (ldrop k Sts) j)
      using k_p by (simp add: add.commute inf)
    then have chain (λx y. (x, y) ∈ RP_filtered_relation)⁻¹⁻¹ (lmap (RP_filtered_measure (λCi. True))
      (ldrop k Sts))
      using inff inf weighted_RP_measure_decreasing_N chain_drop_Sts
        infinite_chain_relation_measure[of λSt. ∃ i. (C, i) ∈# wN_of_wstate St (↝w)] by blast
    then show False
      using wfP_iff_no_infinite_down_chain_llist[of λx y. (x, y) ∈ RP_filtered_relation]
        wf_RP_filtered_relation inff
      by (metis (no_types, lifting) inf_llist_lnth ldrop_enat_inf_llist lfinite_inf_llist
        lfinite_lmap wfPUNIVI wf_induct_rule)
  next
    assume asm: C ∈ Liminf_llist (lmap P_of_state (lmap state_of_wstate Sts))
    from asm obtain i where i_p:
      enat i < llength Sts
      ⋀j. i ≤ j ∧ enat j < llength Sts ⟹ C ∈ P_of_state (state_of_wstate (lnth Sts j))
      unfolding Liminf_llist_def by auto
    then obtain i where (C, i) ∈ Liminf_llist (lmap (set_mset ∘ wP_of_wstate) Sts)
      using persistent_wclause_in_P_if_persistent_clause_in_P[of C] using asm inf by auto
    then have ∃ l. ∀ k ≥ l. (C, i) ∈ (set_mset ∘ wP_of_wstate) (lnth Sts k)
      unfolding Liminf_llist_def using inff inf by auto
    then obtain k where k_p:
      (∀ k'≥k. (C, i) ∈ (set_mset ∘ wP_of_wstate) (lnth Sts k'))
      by blast
    have Ci_in: ∀ k'. (C, i) ∈ (set_mset ∘ wP_of_wstate) (lnth (ldrop k Sts) k')
      using k_p lnth_ldrop[of k _ Sts] inf inff by force
    then have Ci_inn: ∀ k'. (C, i) ∈# (wP_of_wstate) (lnth (ldrop k Sts) k')
      by auto
    have chain (↝w) (ldrop k Sts)
      using deriv inf_chain_ldrop_chain inf inff by auto
```

```
    then have chain (λx y. (x, y) ∈ RP_combined_relation)^-1^-1
      (lmap (RP_combined_measure (weight (C, i))) (ldrop k Sts))
      using inff inf Ci_in weighted_RP_measure_decreasing_P
        infinite_chain_relation_measure[of λSt. (C, i) ∈# wP_of_wstate St (↝w)
          RP_combined_measure (weight (C, i)) ]
      by auto
    then show False
      using wfP_iff_no_infinite_down_chain_llist[of λx y. (x, y) ∈ RP_combined_relation]
        wf_RP_combined_relation inff
      by (smt inf_llist_lnth 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 Sts))
  using RP_saturated_if_fair[OF deriv_RP empty_Q0_RP weighted_RP_fair, unfolded llist.map_comp]
  by simp

corollary weighted_RP_complete:
  ¬ satisfiable (grounding_of_wstate (lhd Sts)) ⟹ {#} ∈ Q_of_state (Liminf_wstate Sts)
  using RP_complete_if_fair[OF deriv_RP empty_Q0_RP weighted_RP_fair, simplified lhd_lmap_Sts]
  by simp

end

end

locale weighted_FO_resolution_prover_with_size_timestamp_factors =
  FO_resolution_prover S subst_atm id_subst comp_subst renamings_apart atm_of_atms mgu less_atm
  for
    S :: ('a :: wellorder) clause ⇒ 'a clause and
    subst_atm :: 'a ⇒ 's ⇒ 'a and
    id_subst :: 's and
    comp_subst :: 's ⇒ 's ⇒ 's and
    renamings_apart :: 'a literal multiset list ⇒ 's list and
    atm_of_atms :: 'a list ⇒ 'a and
    mgu :: 'a set set ⇒ 's option and
    less_atm :: 'a ⇒ 'a ⇒ bool +
  fixes
    size_atm :: 'a ⇒ nat and
    size_factor :: nat and
    timestamp_factor :: nat
  assumes
    timestamp_factor_pos: timestamp_factor > 0
begin

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

lemma weight_mono: i < j ⟹ weight (C, i) < 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 ↝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
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 ∘ (λx. (x, y)) = (λx. (x, y)) ∘ f
  by (simp add: comp_def)


lemma length_remove1_less[termination_simp]: x ∈ set xs ⟹ length (remove1 x xs) < length xs
  by (induct xs) auto


lemma subset_mset_imp_subset_add_mset: A ⊆# B ⟹ A ⊆# add_mset x B
  by (metis add_mset_diff_bothsides diff_subset_eq_self multiset_inter_def subset_mset.inf.absorb2)


lemma subseq_mset_subseteq_mset: subseq xs ys ⟹ mset xs ⊆# mset ys
proof (induct xs arbitrary: ys)
  case (Cons x xs)
  note Outer_Cons = this
  then show ?case
  proof (induct ys)
    case (Cons y ys)
    have subseq xs ys
      by (metis Cons.prems(2) subseq_Cons' subseq_Cons2_iff)
    then show ?case
      using Cons by (metis mset.simps(2) mset_subset_eq_add_mset_cancel subseq_Cons2_iff
          subset_mset_imp_subset_add_mset)
  qed simp
qed simp

lemma map_filter_neq_eq_filter_map:
  map f (filter (λy. f x ≠ f y) xs) = filter (λz. f x ≠ z) (map f xs)
  by (induct xs) auto

lemma mset_map_remdups_gen:
  mset (map f (remdups_gen f xs)) = mset (remdups_gen (λx. 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 (λx. x) xs) = mset_set (set xs)
proof -
  have f = (λx. x) ⟹ mset (remdups_gen f xs) = mset_set (set xs) for f
  proof (induct f xs rule: remdups_gen.induct)
    case (2 f x 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** *wf_app*: *wf r* $\Longrightarrow$ *wf* $\{(x, y).\ (f\ x, f\ y) \in r\}$
**unfolding** *wf_eq_minimal* **by** (*intro allI*, *drule spec*[*of* _ *f* ' *Q* **for** *Q*]) *auto*

**lemma** *wfP_app*: *wfP p* $\Longrightarrow$ *wfP* $(\lambda x\ y.\ p\ (f\ x)\ (f\ y))$
**unfolding** *wfP_def* **by** (*rule wf_app*[*of* $\{(x, y).\ p\ x\ y\}$ *f*, *simplified*])

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

**lemma** *rtranclp_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 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* × *nat*
**type-synonym** *'a dstate* = *'a dclause list* × *'a dclause list* × *'a dclause list* × *nat*

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

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

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

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

**abbreviation** *clss_of_dstate* :: *'a dstate* ⇒ *'a clause set* **where**
*clss_of_dstate St* ≡ *clss_of_state* (*state_of_dstate St*)

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

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

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

```
  (⇝w*) ≡ (⇝w)**

abbreviation trancl_weighted_RP (infix ⇝w+ 50) where
  (⇝w+) ≡ (⇝w)++

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

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

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

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

definition is_reducible_lit :: 'a lclause list ⇒ 'a lclause ⇒ 'a literal ⇒ bool where
  is_reducible_lit Ds C L ⟷
   (∃ D ∈ set Ds. ∃ L' ∈ set D. ∃ σ. − L = L' ·l σ ∧ mset (remove1 L' D) · σ ⊆# mset C)

primrec reduce :: 'a lclause list ⇒ 'a lclause ⇒ 'a lclause ⇒ '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 ⇒ 'a lclause ⇒ bool where
  is_irreducible Ds C ≡ reduce Ds [] C = C

abbreviation is_reducible :: 'a lclause list ⇒ 'a lclause ⇒ bool where
  is_reducible Ds C ≡ reduce Ds [] C ≠ C

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

fun reduce_all2 :: 'a lclause ⇒ 'a dclause list ⇒ 'a dclause list × '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 ⇒ 'b list ⇒ 'b list where
  remove_all xs [] = xs
| remove_all xs (y # ys) = (if y ∈ set xs then remove_all (remove1 y xs) ys else remove_all xs ys)

lemma remove_all_mset_minus: mset ys ⊆# mset xs ⟹ mset (remove_all xs ys) = mset xs − mset ys
proof (induction ys arbitrary: xs)
  case (Cons y ys)
  show ?case
  proof (cases y ∈ set xs)
    case y_in: True
    then have subs: mset ys ⊆# mset (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 ⇒ 'a ⇒'a lclause ⇒ 'a lclause ⇒ 'a lclause where
  resolvent D A CA Ls =
   map (λM. M ·l (the (mgu {insert A (atms_of (mset Ls))}))) (remove_all CA Ls @ D)

definition resolvable :: 'a ⇒ 'a lclause ⇒ 'a lclause ⇒ 'a lclause ⇒ bool where
  resolvable A D CA Ls ⟷
   (let σ = (mgu {insert A (atms_of (mset Ls))}) in
        σ ≠ None
      ∧ Ls ≠ []
      ∧ maximal_wrt (A ·a the σ) ((add_mset (Neg A) (mset D)) · the σ)
      ∧ strictly_maximal_wrt (A ·a the σ) ((mset CA − mset Ls) · the σ)
      ∧ (∀ L ∈ set Ls. is_pos L))

definition resolve_on :: 'a ⇒ 'a lclause ⇒ 'a lclause ⇒ '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 ⇒ 'a lclause ⇒ 'a lclause list 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 ⇒ 'a lclause ⇒ 'a lclause list 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 ⇒ 'a lclause ⇒ 'a lclause list where
  resolve_rename_either_way C D = resolve_rename C D @ resolve_rename D C

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

lemma select_min_weight_clause_in: select_min_weight_clause P0 P ∈ set (P0 # P)
  by (induct P arbitrary: P0) auto

function remdups_clss :: 'a dclause list ⇒ 'a dclause list 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 ⇒ 'a dstate 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.order.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]: reduce Ds (C @ [L]) E = reduce Ds (L # C) E
  by (rule reduce_mset_eq) simp

lemma mset_reduce_subset: mset (reduce Ds C E) ⊆# mset E
  by (induct E arbitrary: C) (auto intro: subset_mset_imp_subset_add_mset)

lemma reduce_idem: reduce Ds C (reduce Ds C E) = 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 ∈ set C′ ⟹ is_reducible_lit Ds (C @ remove1 L C′) L ⟹ reduce Ds C C′ ≠ 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 ∈ set C′ and
      is_reducible_lit Ds (M # C @ 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:
  reduce Ds C C′ ≠ C′ ⟹ ∃L ∈ set C′. is_reducible_lit Ds (C @ 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:
  reduce Ds C C' = C' ⟷ ¬ (∃L ∈ set C'. is_reducible_lit Ds (C @ remove1 L C') L)
  using is_reducible_imp_is_reducible_lit is_reducible_lit_imp_is_reducible by blast

lemma is_irreducible_mset_iff: mset E = mset E' ⟹ reduce Ds C E = E ⟷ 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 = select_min_weight_clause P0 P
  shows weight (apfst mset Ci) = Min ((weight ∘ apfst mset) ` set (P0 # P))
  using assms
proof (induct P arbitrary: P0 Ci)
  case (Cons P1 P)
  note ih = this(1) and ci = this(2)

  show ?case
  proof (cases weight (apfst mset P1) < weight (apfst mset P0))
    case True
    then have min: Min ((weight ∘ apfst mset) ` set (P0 # P1 # P)) =
      Min ((weight ∘ apfst mset) ` 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 Min ((weight ∘ apfst mset) ` set (P0 # P1 # P)) =
      Min ((weight ∘ apfst mset) ` set (P1 # P0 # P))
      by (rule arg_cong[of _ _ Min]) auto
    then have min: Min ((weight ∘ apfst mset) ` set (P0 # P1 # P)) =
      Min ((weight ∘ apfst mset) ` 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_clss_Nil_iff: remdups_clss Cs = [] ⟷ Cs = []
  by (cases Cs, simp, hypsubst, subst remdups_clss.simps(2), simp add: Let_def)

lemma empty_N_if_Nil_in_P_or_Q:
  assumes nil_in: [] ∈ fst ` set (P @ Q)
  shows wstate_of_dstate (N, P, Q, n) ⇝w* wstate_of_dstate ([], P, Q, n)
proof (induct N)
  case ih: (Cons N0 N)
  have wstate_of_dstate (N0 # N, P, Q, n) ⇝w wstate_of_dstate (N, P, Q, n)
    by (rule arg_cong2[THEN iffD1, of _ _ _ _ (⇝w), OF _ _
          wrp.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 ∈ fst ‘ set N and
    p_nsubs: ∀ D ∈ fst ‘ set P. ¬ strictly_subsume [C] D
  shows wstate_of_dstate (N, P @ P′, Q, n)
    ↝w* wstate_of_dstate (N, P @ filter (Not ∘ strictly_subsume [C] ∘ fst) P′, Q, n)
  using p_nsubs
proof (induct length P′ arbitrary: P P′ rule: less_induct)
  case less
  note ih = this(1) and p_nsubs = this(2)

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

    let ?Dj = 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] mset_filter 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 _ _ _ _ (↝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 _ _ _ _ (↝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 _ _ λf. image_mset (apfst mset) (mset (filter f (tl P′)))])
          apply (rule ext)
          apply (simp add: comp_def strictly_subsume_def)
          apply force
         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 _ _ _ _ (⇝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)
    ⇝w* wstate_of_dstate (N, P, Q @ filter (Not ∘ strictly_subsume [C] ∘ fst) Q', n)
proof (induct Q' arbitrary: Q)
  case ih: (Cons Dj Q')
  have wstate_of_dstate (N, P, Q @ Dj # Q', n) ⇝w*
    wstate_of_dstate (N, P, Q @ filter (Not ∘ strictly_subsume [C] ∘ fst) [Dj] @ Q', n)
  proof (cases strictly_subsume [C] (fst Dj))
    case subs: True
    have wstate_of_dstate (N, P, Q @ Dj # Q', n) ⇝w wstate_of_dstate (N, P, Q @ Q', n)
      by (rule arg_cong2[THEN iffD1, of _ _ _ _ (⇝w), OF _ _
            wrp.backward_subsumption_Q[of mset C mset (map (apfst mset) N) mset (fst Dj)
              mset (map (apfst mset) P) mset (map (apfst mset) (Q @ Q')) snd Dj n]])
        (use c_in subs in ‹auto simp: apfst_fst_snd strictly_subsume_def›)
    then show ?thesis
      by auto
  qed simp
  then show ?case
    using ih[of Q @ filter (Not ∘ strictly_subsume [C] ∘ fst) [Dj]] by force
qed simp

lemma reduce_clause_in_P:
  assumes
    c_in: C ∈ fst ` set N and
    p_irred: ∀(E, k) ∈ set (P @ P'). k > j ⟶ is_irreducible [C] E
  shows wstate_of_dstate (N, P @ (D @ D', j) # P', Q, n)
    ⇝w* wstate_of_dstate (N, P @ (D @ reduce [C] D D', j) # P', Q, n)
proof (induct D' arbitrary: D)
  case ih: (Cons L D')
  show ?case
  proof (cases is_reducible_lit [C] (D @ D') L)
    case l_red: True
    then obtain L' :: 'a literal and σ :: 's where
      l'_in: L' ∈ set C and
      not_l: − L = L' ·l σ and
      subs: mset (remove1 L' C) · σ ⊆# mset (D @ D')
      unfolding is_reducible_lit_def by force

    have ldd'_red: is_reducible [C] (L # D @ D')
      apply (rule is_reducible_lit_imp_is_reducible)
      using l_red by auto

    have lt_imp_neq: ∀(E, k) ∈ set (P @ P'). j < k ⟶ mset E ≠ mset (L # D @ D')
      using p_irred ldd'_red is_irreducible_mset_iff by fast

    have wstate_of_dstate (N, P @ (D @ L # D', j) # P', Q, n)
      ⇝w wstate_of_dstate (N, P @ (D @ D', j) # P', Q, n)
```

```
      apply (rule arg_cong2[THEN iffD1, of _ _ _ _ (⇝w), OF _ _
            wrp.backward_reduction_P[of mset C − {#L'#} L' mset (map (apfst mset) N) L σ
              mset (D @ D') mset (map (apfst mset) (P @ P')) j mset (map (apfst mset) Q) n]])
      using l'_in not_l subs c_in lt_imp_neq by (simp_all add: case_prod_beta) force+
    then show ?thesis
      using ih[of D] l_red by simp
  next
    case False
    then show ?thesis
      using ih[of D @ [L]] by simp
  qed
qed simp

lemma reduce_clause_in_Q:
  assumes
    c_in: C ∈ fst ‘ set N and
    p_irred: ∀(E, k) ∈ set P. k > j ⟶ is_irreducible [C] E and
    d'_red: reduce [C] D D' ≠ D'
  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 σ :: 's where
      l'_in: L' ∈ set C and
      not_l: − L = L' ·l σ and
      subs: mset (remove1 L' C) · σ ⊆# 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 σ
              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 σ
    assume ord_resolve S [mset CA] (add_mset (Neg A) (mset D)) [AA] [A] σ E
    then obtain Cs where res': E = sum_list Cs · σ + mset D · σ
      length Cs = Suc 0
      mset CA = Cs ! 0 + poss AA
      AA ≠ {#}
      Some σ = mgu {insert A (set_mset AA)}
      eligible S σ [A] (add_mset (Neg A) (mset D))
      strictly_maximal_wrt (A ·a σ) (Cs ! 0 · σ)
      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 · σ + mset D · σ
      mset CA = C + poss AA
      AA ≠ {#}
      Some σ = mgu {insert A (set_mset AA)}
      eligible S σ [A] (add_mset (Neg A) (mset D))
      strictly_maximal_wrt (A ·a σ) (C · σ)
      S (C + poss AA) = {#}
      unfolding ord_resolve.simps by auto
    from this(1) have
      E = C · σ + mset D · σ
      unfolding C_def using res'(2) by (cases Cs) auto
    note res' = this res(2−7)
    have ∃Al. mset Al = AA ∧ 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 ∈# poss AA )
        case True
        then have pos_L: is_pos L
          by auto
        have rem: ⋀A'. Pos A' ∈# poss AA ⟹
          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 ∃Al. mset Al = remove1_mset (atm_of L) AA ∧ 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 ∧ 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 ∃Al. mset Al = AA ∧ 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 ∃y. mgu {insert A (atms_of (mset Ls))} = Some y
    unfolding Ls_def using res(4) Al_p by (metis atms_of_poss mset_map)
  moreover have Ls ≠ []
    using Al_p(1) Ls_def res'(3) by auto
  moreover have σ_p: the (mgu {insert A (set Al)}) = σ
    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 σ_p Al_p by auto
  moreover have ∀L ∈ set Ls. 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 ⊆# mset CA
  using ls_subq_ca subseq_mset_subseteq_mset[of Ls CA] by simp
have {#x ·l σ. x ∈# mset CA − mset Ls#} + {#M ·l σ. M ∈# mset D#} = C · σ + mset D · σ
  using diff unfolding subst_cls_def by simp
then have {#x ·l σ. x ∈# mset CA − mset Ls#} + {#M ·l σ. M ∈# mset D#} = E
  using res'(1) by auto
then have {#M ·l σ. M ∈# mset (remove_all CA Ls)#} + {#M ·l σ . M ∈# 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 ∈ mset ‘ set (resolve_on A D CA)
  unfolding resolve_on_def by auto
```

```
  }
  then show {E. ∃AA σ. ord_resolve S [mset CA] ({#Neg A#} + mset D) [AA] [A] σ E}
    ⊆ mset ` set (resolve_on A D CA)
    by auto
qed

lemma set_resolve_eq_UNION_set_resolve_on:
  set (resolve C D) =
   (⋃L ∈ set D.
      (case L of
         Pos _ ⇒ {}
      | Neg A ⇒ 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 bexI)
   apply (rule conjI)
    apply blast
   apply clarify
   apply (rule conjI)
    apply clarify
    apply (rule_tac x = AA in exI)
    apply (rule_tac x = σ in exI)
    apply (frule ord_resolve.simps[THEN iffD1])
    apply force
   apply (drule ord_resolve.simps[THEN iffD1])
   apply (clarsimp simp: eligible_iff simp del: subst_cls_add_mset subst_cls_union)
   apply (drule maximal_wrt_subst)
   apply sat
  apply (drule ord_resolve.simps[THEN iffD1])
  using set_mset_mset by fastforce

lemma poss_in_map_clauseD:
  poss AA ⊆# map_clause f C ⟹ ∃AA0. poss AA0 ⊆# C ∧ AA = {#f A. A ∈# AA0#}
proof (induct AA arbitrary: C)
  case (add A AA)
  note ih = this(1) and aaa_sub = this(2)

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

  have poss AA ⊆# 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 ⊆# C − {#Pos A0#} and
    aa: AA = image_mset f AA0
    using ih by meson

  have poss (add_mset A0 AA0) ⊆# 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 ⊆# {#L ·l ϱ. L ∈# mset C#} ⟹ ∃ AA0. poss AA0 ⊆# mset C ∧ AA = AA0 ·am ϱ
  unfolding subst_atm_mset_def subst_lit_def by (rule poss_in_map_clauseD)

lemma neg_in_map_literalD: Neg A ∈ map_literal f ` D ⟹ ∃ A0. Neg A0 ∈ D ∧ A = f A0
  unfolding image_def by (clarify, case_tac x, auto)

lemma neg_in_filterD: Neg A ∈# {#L ·l ϱ'. L ∈# mset D#} ⟹ ∃ A0. Neg A0 ∈# mset D ∧ A = A0 ·a ϱ'
  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 ?ϱs = renamings_apart [mset D, mset C]
  define ϱ' :: 's where
    ϱ' = hd ?ϱs
  define ϱ :: 's where
    ϱ = last ?ϱs

  have tl_ϱs: tl ?ϱs = [ϱ]
    unfolding ϱ_def
    using renamings_apart_length Nitpick.size_list_simp(2) Suc_length_conv last.simps
    by (smt length_greater_0_conv list.sel(3))

  {
    fix E
    assume e_in: E ∈ mset ` set (resolve_rename C D)

    from e_in obtain AA :: 'a multiset and A :: 'a and σ :: 's where
      aa_sub: poss AA ⊆# mset C · ϱ and
      a_in: Neg A ∈# mset D · ϱ' and
      res_e: ord_resolve S [mset C · ϱ] {#L ·l ϱ'. L ∈# mset D#} [AA] [A] σ E
      unfolding ϱ'_def ϱ_def
      apply atomize_elim
      using e_in unfolding resolve_rename_def Let_def resolve_eq_Bin_ord_resolve
      apply clarsimp
      apply (frule ord_resolve_one_side_prem)
      apply (frule ord_resolve.simps[THEN iffD1])
      apply (rule_tac x = AA in exI)
      apply (clarsimp simp: subst_cls_def)
      apply (rule_tac x = A in exI)
      by (metis (full_types) Melem_subst_cls set_mset_mset subst_cls_def union_single_eq_member)

    obtain AA0 :: 'a multiset where
      aa0_sub: poss AA0 ⊆# mset C and
      aa: AA = AA0 ·am ϱ
      using aa_sub
      apply atomize_elim
      apply (rule ord_resolve.cases[OF res_e])
      by (rule poss_subset_filterD[OF aa_sub[unfolded subst_cls_def]])

    obtain A0 :: 'a where
      a0_in: Neg A0 ∈ set D and
      a: A = A0 ·a ϱ'
      apply atomize_elim
      apply (rule ord_resolve.cases[OF res_e])
      using neg_in_filterD[OF a_in[unfolded subst_cls_def]] by simp

    show E ∈ Bin_ord_resolve_rename (mset C) (mset D)
      unfolding ord_resolve_rename.simps
```

```
      using res_e
      apply clarsimp
      apply (rule_tac x = AA0 in exI)
      apply (intro conjI)
       apply (rule aa0_sub)
      apply (rule_tac x = A0 in exI)
      apply (intro conjI)
       apply (rule a0_in)
      apply (rule_tac x = σ in exI)
      unfolding aa a ϱ'_def[symmetric] ϱ_def[symmetric] tl_ϱs by (simp add: subst_cls_def)
  }
  {
    fix E
    assume e_in: E ∈ Bin_ord_resolve_rename (mset C) (mset D)
    show E ∈ mset ‘ set (resolve_rename C D)
      using e_in
      unfolding resolve_rename_def Let_def resolve_eq_Bin_ord_resolve ord_resolve_rename.simps
      apply 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_prems_of γ = {#THE D. D ∈# side_prems_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 ∈# prems_of γ} =
   {γ ∈ ord_FO_Γ S. ∃D ∈ DD ∪ {C}. prems_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_congls_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) ∈ set P and
    ci_min: ∀(D, j) ∈# mset (map (apfst mset) P). weight (mset C, i) ≤ weight (D, j)
  shows
    wstate_of_dstate ([], P, Q, n) ⇝w
     wstate_of_dstate (map (λD. (D, n)) (remdups_gen mset (resolve_rename C C @
         concat (map (resolve_rename_either_way C ∘ fst) Q))),
       filter (λ(D, j). mset D ≠ mset C) P, (C, i) # Q, Suc n)
    (is _ ⇝w wstate_of_dstate (?N, _))
proof −
  have ms_ci_in: (mset C, i) ∈# image_mset (apfst mset) (mset P)
    using ci_in by force

  show ?thesis
    apply (rule arg_cong2[THEN iffD1, of _ _ _ _ (⇝w), OF _ _
          wrp.inference_computation[of mset (map (apfst mset) P) − {#(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 _ _ λN. (λD. (D, n)) ‘ N])
    apply (simp only: map_concat list.map_comp image_comp)
    using resolve_rename_either_way_eq_congls_of_inferences_between[of C fst ‘ set Q, symmetric]
    by (simp add: image_comp comp_def image_UN)
qed

lemma nonfinal_deterministic_RP_step:
  assumes
    nonfinal: ¬ is_final_dstate St and
    step: St′ = deterministic_RP_step St
  shows wstate_of_dstate St ⇝w⁺ wstate_of_dstate St′
```

```
proof –
  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 deterministic_RP_step.simps, simplified]

  show ?thesis
  proof (cases ∃ Ci ∈ set P ∪ set Q. fst Ci = [])
    case nil_in: True
    note step = step[simplified nil_in, simplified]

    have nil_in': [] ∈ fst ` set (P @ Q)
      using nil_in by (force simp: image_def)

    have star: [] ∈ fst ` set (P @ Q) ⟹
      wstate_of_dstate (N, P, Q, n)
      ↝w* wstate_of_dstate ([], [], remdups_clss P @ Q, n + length (remdups_clss P))
    proof (induct length (remdups_clss 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) ↝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 ≠ []
        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) ∈ set P
        unfolding ci using p_cons select_min_weight_clause_in[of hd P tl P] by simp
      have ci_min: ∀(D, j) ∈# mset (map (apfst mset) P). weight (mset C, i) ≤ weight (D, j)
        by (subst p_cons) (simp add: select_min_weight_clause_min_weight[OF ci, simplified])

      let ?P' = filter (λ(D, j). mset D ≠ mset C) P

      have ms_p'_ci_q_eq: mset (remdups_clss ?P' @ (C, i) # Q) = mset (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: length (remdups_clss P) = length (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) ↝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
        ... ↝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 ... ↝w* wstate_of_dstate ([], [], remdups_clss P @ Q, n + length (remdups_clss P))
        apply (rule arg_cong2[THEN iffD1, of _ _ _ _ (↝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)
        ↝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 σ :: ′s where
      D ∈ set (map fst P @ map fst Q) and
      D′ = remove1 L′ D and
      L′ ∈ set D and
      − L = L′ ·l σ and
      mset D′ · σ ⊆# mset (E @ C)
      using l_red unfolding is_reducible_lit_def comp_def by blast
    then have σ:
      mset D′ + {#L′#} ∈ set (map (mset ∘ fst) (P @ Q))
      − L = L′ ·l σ ∧ mset D′ · σ ⊆# 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)
        ↝w wstate_of_dstate ((E @ C, i) # N′, P, Q, n)
      by (rule arg_cong2[THEN iffD1, of _ _ _ _ (↝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]

    have wstate_of_dstate ((C', i) # N', P, Q, n) ⇝w wstate_of_dstate (N', P, Q, n)
    proof (cases is_tautology C')
      case True
      then obtain A :: 'a where
        neg_a: Neg A ∈ set C' and pos_a: Pos A ∈ set C'
        unfolding is_tautology_def by blast
      show ?thesis
        by (rule arg_cong2[THEN iffD1, of _ _ _ _ (⇝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 ∈ 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 _ _ _ _ (⇝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 × 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 × nat) list where
      Q'' = filter (Not ∘ strictly_subsume [C'] ∘ fst) Q'
    define P'' :: ('a literal list × nat) list where
      P'' = filter (Not ∘ strictly_subsume [C'] ∘ 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)
            ⇝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 ... ⇝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 ... ⇝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 ... ⇝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 (⇝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 ⟹ 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 ∃N′ P′ Q′ n′. (∃k. (deterministic_RP_step ^^ k) (N, P, Q, n) = (N′, P′, Q′, n′))
    ∧ is_final_dstate (N′, P′, Q′, n′) ∧ 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 ^^ 0) (N, P, Q, n) = (N, P, Q, n)
      ∧ is_final_dstate (N, P, Q, n) ∧ 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 ^^ 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 ≡ (N0, [], [], n0)

abbreviation grounded_N0 where
  grounded_N0 ≡ grounding_of_clss (set (map (mset ∘ fst) N0))

abbreviation grounded_R :: 'a clause set where
  grounded_R ≡ grounding_of_clss (set (map mset R))

primcorec derivation_from :: 'a dstate ⇒ 'a dstate llist where
  derivation_from St =
   LCons St (if is_final_dstate St then LNil else derivation_from (deterministic_RP_step St))

abbreviation Sts :: 'a dstate llist where
  Sts ≡ derivation_from St0

abbreviation wSts :: 'a wstate llist where
  wSts ≡ lmap wstate_of_dstate Sts

lemma full_deriv_wSts_trancl_weighted_RP: full_chain (↝w⁺) wSts
proof −
  have Sts' = derivation_from St0' ⟹ full_chain (↝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 ⋀St''. ¬ wstate_of_dstate St0' ↝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' ↝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* ($\leadsto_w$) *wSts'* ∧ *emb wSts wSts'* ∧ *lhd wSts'* = *lhd wSts* ∧ *llast wSts'* = *llast wSts*)

**lemma** *sswSts*:
*full_chain* ($\leadsto_w$) *sswSts* ∧ *emb wSts sswSts* ∧ *lhd sswSts* = *lhd wSts* ∧ *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_lnull_sswSts*: ¬ *lnull 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* ⇒ *'a clause* **where**
*S_ssgQ* ≡ *wrp.S_gQ sswSts*

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

**abbreviation** *Rf* :: *'a clause set* ⇒ *'a clause set* **where**
*Rf* ≡ *standard_redundancy_criterion.Rf*

**abbreviation** *Ri* :: *'a clause set* ⇒ *'a inference set* **where**
*Ri* ≡ *standard_redundancy_criterion.Ri ord_Γ*

**abbreviation** *saturated_upto* :: *'a clause set* ⇒ *bool* **where**
*saturated_upto* ≡ *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 ⊨s grounded_N0 ⟷ I ⊨s grounded_R (is ?model)
proof -
  obtain N' P' Q' :: 'a dclause list and n' k :: nat where
    k_steps: (deterministic_RP_step ^^ 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 ↝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 ^^ k') St0' = ?Stk ⟹ 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: llast_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] llast_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_N0
    by (simp add: clss_of_state_def comp_def)

  have ?saturated ∧ ?model
  proof (cases [] ∈ set R)
    case True
    then have emp_in: {#} ∈ grounded_R
      unfolding grounding_of_clss_def grounding_of_cls_def by (auto intro: ex_ground_subst)

    have grounded_R ⊆ wrp.grounding_of_wstate (llast sswSts)
      unfolding r llast_sswSts
      by (simp add: last_sts llast_lmap[OF lfinite_Sts] clss_of_state_def grounding_of_clss_def)
```

```
    then have gr_last_st: grounded_R ⊆ wrp.grounding_of_wstate (wstate_of_dstate (llast Sts))
      by (simp add: lfinite_Sts llast_lmap llast_sswSts)

    have gr_r_fls: ¬ I ⊨s grounded_R
      using emp_in unfolding true_clss_def by force
    then have gr_last_fls: ¬ I ⊨s wrp.grounding_of_wstate (wstate_of_dstate (llast Sts))
      using gr_last_st unfolding true_clss_def by auto

    have ?saturated
      unfolding wrp.ord_Γ_saturated_upto_def[OF sswSts_thms]
        wrp.ord_Γ_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 (↝w) λSt. I ⊨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] clss_of_state_def 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 (↝w) λSt. I ⊨s 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:
  ¬ satisfiable grounded_N0 ⟷ {#} ∈ grounded_R (is ?lhs ⟷ ?rhs)
proof
  assume ?rhs
  then have ¬ satisfiable grounded_R
    unfolding true_clss_def true_cls_def by force
  then show ?lhs
    using deterministic_RP_model[THEN iffD1] by blast
next
  assume ?lhs
  then have ¬ satisfiable grounded_R
    using deterministic_RP_model[THEN iffD2] by blast
  then show ?rhs
    unfolding wrp.ord_Γ_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_N0
proof (rule ccontr)
  assume unsat: ¬ satisfiable grounded_N0

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

  have bot_in_ss: {#} ∈ 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: {#} ∈ 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 ^^ k) St0))
  proof -
    have enat k < llength Sts' ⟹ Sts' = derivation_from St0' ⟹
      {#} ∈ wrp.Q_of_wstate (lnth (lmap wstate_of_dstate Sts') k) ⟹
      {#} ∈ Q_of_state (state_of_dstate ((deterministic_RP_step ^^ 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 ≠ None
    by (rule is_final_dstate_funpow_imp_deterministic_RP_neq_None[of Suc k],
        cases (deterministic_RP_step ^^ 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
```

# 4 Integration of IsaFoR Terms

This theory implements the abstract interface for atoms and substitutions using the IsaFoR library (part of the AFP entry *First_Order_Terms*).

```
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
begin

hide-const (open) mgu

abbreviation subst_apply_literal ::
  ('f, 'v) term literal ⇒ ('f, 'v, 'w) gsubst ⇒ ('f, 'w) term literal (infixl ·lit 60) where
  L ·lit σ ≡ map_literal (λA. A · σ) L

definition subst_apply_clause ::
  ('f, 'v) term clause ⇒ ('f, 'v, 'w) gsubst ⇒ ('f, 'w) term clause (infixl ·cls 60) where
  C ·cls σ = image_mset (λL. L ·lit σ) C

abbreviation vars_lit :: ('f, 'v) term literal ⇒ 'v set where
  vars_lit L ≡ vars_term (atm_of L)

definition vars_clause :: ('f, 'v) term clause ⇒ 'v set where
  vars_clause C = Union (set_mset (image_mset vars_lit C))

definition vars_clause_list :: ('f, 'v) term clause list ⇒ 'v set where
  vars_clause_list Cs = Union (vars_clause ` set Cs)

definition vars_partitioned :: ('f,'v) term clause list ⇒ bool where
  vars_partitioned Cs ⟷
   (∀ i < length Cs. ∀ j < length Cs. i ≠ j ⟶ (vars_clause (Cs ! i) ∩ vars_clause (Cs ! j)) = {})

lemma vars_clause_mono: S ⊆# C ⟹ vars_clause S ⊆ vars_clause C
  unfolding vars_clause_def by auto

interpretation substitution_ops (·) Var (∘s) .

lemma is_ground_atm_is_ground_on_var:
  assumes is_ground_atm (A · σ) and v ∈ vars_term A
  shows is_ground_atm (σ v)
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 (subst_lit L σ) and v_in_L: v ∈ vars_lit L
  shows is_ground_atm (σ v)
```

```
proof -
  let ?A = atm_of L
  from v_in_L have A_p: v ∈ vars_term ?A
    by auto
  then have is_ground_atm (?A · σ)
    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 (subst_cls C σ) and
    v_in_C: v ∈ vars_clause C
  shows is_ground_atm (σ v)
proof -
  from v_in_C obtain L where L_p: L ∈# C v ∈ vars_lit L
    unfolding vars_clause_def by auto
  then have is_ground_lit (subst_lit L σ)
    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 σ)
    and v_in_Cs: v ∈ vars_clause_list Cs
  shows is_ground_atm (σ v)
proof -
  from v_in_Cs obtain C where C_p: C ∈ set Cs v ∈ vars_clause C
    unfolding vars_clause_list_def by auto
  then have is_ground_cls (subst_cls C σ)
    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 ∀ v ∈ vars_lit L. σ v = τ v
  shows subst_lit L σ = subst_lit L τ
  using assms
proof (induction L)
  case (Pos x)
  then have ∀ v ∈ vars_term x. σ v = τ v ⟹ subst_atm_abbrev x σ = subst_atm_abbrev x τ
    using term_subst_eq by metis+
  then show ?case
    unfolding subst_lit_def using Pos by auto
next
  case (Neg x)
  then have ∀ v ∈ vars_term x. σ v = τ v ⟹ subst_atm_abbrev x σ = subst_atm_abbrev x τ
    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 < length (list_of_mset S)
  shows list_of_mset S ! i ∈# S
proof -
  from assms have list_of_mset S ! i ∈ set (list_of_mset S)
    by auto
  then have list_of_mset S ! i ∈# mset (list_of_mset S)
    by (meson in_multiset_in_set)
```

```
  then show ?thesis
    by auto
qed

lemma same_on_vars_clause:
  assumes ∀ v ∈ vars_clause S. σ v = τ v
  shows subst_cls S σ = subst_cls S τ
  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 σs :: ‹('b ⇒ ('a, 'b) term) list›
  assume length σs = length Cs
  with assms[unfolded vars_partitioned_def] Fun_More.fun_merge[of map vars_clause Cs nth σs]
  obtain σ where
    σ_p: ∀ i < length (map vars_clause Cs). ∀ x ∈ map vars_clause Cs ! i. σ x = (σs ! i) x
    by auto
  have ∀ i < length Cs. ∀ S. S ⊆# Cs ! i ⟶ subst_cls S (σs ! i) = subst_cls S σ
  proof (rule, rule, rule, rule)
    fix i :: nat and S :: ('a, 'b) term literal multiset
    assume
      i < length Cs and
      S ⊆# Cs ! i
    then have ∀ v ∈ vars_clause S. (σs ! i) v = σ v
      using vars_clause_mono[of S Cs ! i] σ_p by auto
    then show subst_cls S (σs ! i) = subst_cls S σ
      using same_on_vars_clause by auto
  qed
  then show ∃ τ. ∀ i<length Cs. ∀ S. S ⊆# Cs ! i ⟶ subst_cls S (σs ! i) = subst_cls S τ
    by auto
qed

lemma vars_in_instance_in_range_term:
  vars_term (subst_atm_abbrev A σ) ⊆ Union (image vars_term (range σ))
  by (induction A) auto

lemma vars_in_instance_in_range_lit: vars_lit (subst_lit L σ) ⊆ Union (image vars_term (range σ))
proof (induction L)
  case (Pos A)
  have vars_term (A · σ) ⊆ Union (image vars_term (range σ))
    using vars_in_instance_in_range_term[of A σ] by blast
  then show ?case by auto
next
  case (Neg A)
  have vars_term (A · σ) ⊆ Union (image vars_term (range σ))
    using vars_in_instance_in_range_term[of A σ] by blast
  then show ?case by auto
qed

lemma vars_in_instance_in_range_cls:
  vars_clause (subst_cls C σ) ⊆ Union (image vars_term (range σ))
  unfolding vars_clause_def subst_cls_def using vars_in_instance_in_range_lit[of _ σ] by auto

primrec renamings_apart :: ('f, nat) term clause list ⇒ (('f, nat) subst) list where
  renamings_apart [] = []
| renamings_apart (C # Cs) =
   (let σs = renamings_apart Cs in
     (λv. Var (v + Max (vars_clause_list (subst_cls_lists Cs σs) ∪ {0}) + 1)) # σs)
```

**definition** *var_map_of_subst* :: (*'f*, *nat*) *subst* ⇒ *nat* ⇒ *nat* **where**
*var_map_of_subst σ v* = *the_Var* (*σ v*)

**lemma** *len_renamings_apart*: *length* (*renamings_apart Cs*) = *length Cs*
**by** (*induction Cs*) (*auto simp*: *Let_def*)

**lemma** *renamings_apart_is_Var*: ∀ *σ* ∈ *set* (*renamings_apart Cs*). ∀ *x*. *is_Var* (*σ x*)
**by** (*induction Cs*) (*auto simp*: *Let_def*)

**lemma** *renamings_apart_inj*: ∀ *σ* ∈ *set* (*renamings_apart Cs*). *inj σ*
**proof** (*induction Cs*)
**case** (*Cons a Cs*)
**then have** *inj* (*λv*. *Var* (*Suc* (*v* + *Max* (*vars_clause_list*
(*subst_cls_lists Cs* (*renamings_apart Cs*)) ∪ {*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* ⟹ *Suc* (*v* + *Max* (*insert 0 X*)) ∉ *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* ≠[]
**shows** *vars_clause* (*hd* (*subst_cls_lists Cs* (*renamings_apart Cs*)))
∩ *vars_clause_list* (*tl* (*subst_cls_lists Cs* (*renamings_apart Cs*))) = {}
**using** *assms*
**proof** (*induction Cs*)
**case** (*Cons C Cs*)
**define** *σ'* :: *nat* ⇒ *nat*
**where** *σ'* = (*λv*. (*Suc* (*v* + *Max* ((*vars_clause_list* (*subst_cls_lists Cs*
(*renamings_apart Cs*))) ∪ {*0*}))))
**define** *σ* :: *nat* ⇒ (*'a*, *nat*) *term*
**where** *σ* = (*λv*. *Var* (*σ' v*))

**have** *vars_clause* (*subst_cls C σ*) ⊆ *UNION* (*range σ*) *vars_term*
**using** *vars_in_instance_in_range_cls*[*of C hd* (*renamings_apart* (*C* # *Cs*))] *σ_def σ'_def*
**by** (*auto simp*: *Let_def*)
**moreover have** *UNION* (*range σ*) *vars_term*
∩ *vars_clause_list* (*subst_cls_lists Cs* (*renamings_apart Cs*)) = {}
**proof** −
**have** *range σ'* ∩ *vars_clause_list* (*subst_cls_lists Cs* (*renamings_apart Cs*)) = {}
**unfolding** *σ'_def* **using** *Suc_Max_notin_set* **by** *auto*
**then show** *?thesis*
**unfolding** *σ_def σ'_def* **by** *auto*
**qed**
**ultimately have** *vars_clause* (*subst_cls C σ*)
∩ *vars_clause_list* (*subst_cls_lists Cs* (*renamings_apart Cs*)) = {}
**by** *auto*
**then show** *?case*

```
    unfolding σ_def σ'_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) ∩
        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 disjoin_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 disjoin_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 ⋀A. A · Var = A
    by auto
next
  show ⋀A τ σ. A · τ ∘s σ = A · τ · σ
    by auto
next
  show ⋀σ τ. (⋀A. A · σ = A · τ) ⟹ σ = τ
    by (simp add: subst_term_eqI)
next
  fix C :: ('f, nat) term clause
  fix σ
  assume is_ground_cls (subst_cls C σ)
  then have ground_atms_σ: ⋀v. v ∈ vars_clause C ⟹ is_ground_atm (σ 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 τ where τ = (λv. if v ∈ vars_clause C then σ v else some_ground_trm)
  then have τ_σ: ∀ v ∈ vars_clause C. σ v = τ v
    unfolding τ_def by auto

  have all_ground_τ: is_ground_atm (τ v) for v
```

```
  proof (cases v ∈ vars_clause C)
    case True
    then show ?thesis
      using ground_atms_σ τ_σ by auto
  next
    case False
    then show ?thesis
      unfolding τ_def using ground_trm by auto
  qed
  have is_ground_subst τ
    unfolding is_ground_subst_def
  proof
    fix A
    show is_ground_atm (subst_atm_abbrev A τ)
    proof (induction A)
      case (Var v)
      then show ?case using all_ground_τ 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 ∀ v ∈ vars_clause C. σ v = τ v
    using τ_σ unfolding vars_clause_list_def
    by blast
  then have subst_cls C σ = subst_cls C τ
    using same_on_vars_clause by auto
  ultimately show ∃ τ. is_ground_subst τ ∧ subst_cls C τ = subst_cls C σ
    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:
      ⋀σ :: ('f, nat) subst. (⋀x. is_Var (σ x)) ⟹ inj σ ⟹ is_renaming σ
    proof –
      fix σ :: ('f, nat) subst
      fix x
      assume is_var_σ: ⋀x. is_Var (σ x)
      assume inj_σ: inj σ
      define σ' where σ' = var_map_of_subst σ
      have σ: σ = Var ∘ σ'
        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 ∀ x. (Var ∘ (inv σ' ∘ σ')) x = Var x
        by metis
      then have ∀ x. ((Var ∘ σ') ∘s (Var ∘ (inv σ'))) x = Var x
        unfolding subst_compose_def by auto
      then have σ ∘s (Var ∘ (inv σ')) = Var
        using σ by auto
```

```
      then show is_renaming σ
        unfolding is_renaming_def by blast
    qed
    then have ∀ σ ∈ (set (renamings_apart Cs)). is_renaming σ
      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 ⋀σ As Bs. Fun undefined As · σ = Fun undefined Bs ⟷ map (λA. A · σ) As = 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 ⇒ ('a × '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 ∘ compare_linorder.sorted_list_of_set) ` AAA))

lemma unifies_all_pairs_iff:
  (∀ p ∈ set (pairs xs). fst p · σ = snd p · σ) ⟷ (∀ a ∈ set xs. ∀ b ∈ set xs. a · σ = b · σ)
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) ∈ set ((pairs As))
  shows A ∈ set As ∧ B ∈ 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
    ∀ AA ∈ AAA. finite AA
    AB_pairs ∈ (pairs ∘ compare_linorder.sorted_list_of_set) ` AAA and
    (A :: _ :: compare, B) ∈ set AB_pairs
  shows ∃ AA. AA ∈ AAA ∧ A ∈ AA ∧ B ∈ AA
proof –
  from assms have AB_pairs ∈ (pairs ∘ compare_linorder.sorted_list_of_set) ` AAA
    by auto
  then obtain AA where
    AA_p: AA ∈ AAA ∧ (pairs ∘ compare_linorder.sorted_list_of_set) AA = AB_pairs
    by auto
  have (A, B) ∈ set (pairs (compare_linorder.sorted_list_of_set AA))
    using AA_p[] assms(4) by auto
  then have A ∈ set (compare_linorder.sorted_list_of_set AA) and
    B ∈ 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
    ∀ AA ∈ AAA. finite AA
  shows unifiers (set (Pairs AAA)) = {σ. is_unifiers σ AAA}
proof (rule; rule)
  fix σ :: ('a, 'b) subst
  assume asm: σ ∈ unifiers (set (Pairs AAA))
  have ⋀AA. AA ∈ AAA ⟹ card (AA ·set σ) ≤ Suc 0
  proof –
    fix AA :: ('a, 'b) term set
    assume asm': AA ∈ AAA
    then have ∀ p ∈ set (pairs (compare_linorder.sorted_list_of_set AA)).
      subst_atm_abbrev (fst p) σ = subst_atm_abbrev (snd p) σ
      using assms asm unfolding Pairs_def by auto
    then have ∀ A ∈ AA. ∀ B ∈ AA. subst_atm_abbrev A σ = subst_atm_abbrev B σ
      using assms asm' unfolding unifies_all_pairs_iff
      using compare_linorder.sorted_list_of_set by blast
    then show card (AA ·set σ) ≤ Suc 0
      by (smt imageE card.empty card_Suc_eq card_mono finite.intros(1) finite_insert le_SucI
          singletonI subsetI)
  qed
  then show σ ∈ {σ. is_unifiers σ AAA}
    using assms by (auto simp: is_unifiers_def is_unifier_def subst_atms_def)
next
  fix σ :: ('a, 'b) subst
  assume asm: σ ∈ {σ. is_unifiers σ AAA}
```

```
  {
    fix AB_pairs A B
    assume
      AB_pairs ∈ set (compare_linorder.sorted_list_of_set
        ((pairs ∘ compare_linorder.sorted_list_of_set) ` AAA)) and
      (A, B) ∈ set AB_pairs
    then have ∃ AA. AA ∈ AAA ∧ A ∈ AA ∧ B ∈ AA
      using assms by (simp add: in_pairs_sorted_list_of_set_in_set)
    then obtain AA where
     a: AA ∈ AAA A ∈ AA B ∈ AA
      by blast
    from a assms asm have card_AA_σ: card (AA ·set σ) ≤ Suc 0
      unfolding is_unifiers_def is_unifier_def subst_atms_def by auto
    have subst_atm_abbrev A σ = subst_atm_abbrev B σ
    proof (cases card (AA ·set σ) = Suc 0)
      case True
      moreover
      have subst_atm_abbrev A σ ∈ AA ·set σ
        using a assms asm card_AA_σ by auto
      moreover
      have subst_atm_abbrev B σ ∈ AA ·set σ
        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 ·set σ) = 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 σ ∈ 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 (·) Var :: _ ⇒ ('f :: compare, nat) term (∘s) Fun undefined
  renamings_apart mgu_sets
proof
  fix AAA :: ('a :: compare, nat) term set set and σ :: ('a, nat) subst
  assume fin: finite AAA ∀ AA ∈ 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 σ :: ('a, nat) subst
  assume fin: finite AAA ∀ AA ∈ AAA. finite AA and is_unifiers σ AAA
  then have σ ∈ unifiers (set (Pairs AAA))
    unfolding is_mgu_def unifiers_Pairs[OF fin] by auto
  then show ∃ τ. mgu_sets AAA = Some τ
    using unify_complete unfolding mgu_sets_def by blast
qed

derive linorder prod
derive linorder list
```

**end**

# 5 An Executable Algorithm for Clause Subsumption

This theory provides a functional implementation of clause subsumption, building on the IsaFoR library (part of the AFP entry *First_Order_Terms*).

**theory** *Executable_Subsumption*
  **imports** *IsaFoR_Term First_Order_Terms.Matching*
**begin**

## 5.1 Naive Implementation of Clause Subsumption

**fun** *subsumes_list* **where**
  *subsumes_list* [] *Ks* $\sigma$ = *True*
| *subsumes_list* (*L* # *Ls*) *Ks* $\sigma$ =
    ($\exists$ *K* $\in$ *set Ks*. *is_pos K* = *is_pos L* $\wedge$
      (*case match_term_list* [(*atm_of L*, *atm_of K*)] $\sigma$ *of*
        *None* $\Rightarrow$ *False*
      | *Some* $\varrho$ $\Rightarrow$ *subsumes_list Ls* (*remove1 K Ks*) $\varrho$))

**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* $\sigma$ $\tau$ $\Longrightarrow$ *extends_subst* $\tau$ $\varrho$ $\Longrightarrow$ *extends_subst* $\sigma$ $\varrho$
  **unfolding** *extends_subst_def dom_def* **by** (*metis mem_Collect_eq*)

**lemma** *extends_subst_dom*: *extends_subst* $\sigma$ $\tau$ $\Longrightarrow$ *dom* $\sigma$ $\subseteq$ *dom* $\tau$
  **unfolding** *extends_subst_def dom_def* **by** *auto*

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

**lemma** *extends_subst_fun_upd_new*:
  $\sigma$ *x* = *None* $\Longrightarrow$ *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* $\Longrightarrow$ *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* $\sigma$ $\tau$ $\Longrightarrow$ *vars_term t* $\subseteq$ *dom* $\sigma$ $\Longrightarrow$ *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* $\sigma$ $\tau$ $\Longrightarrow$ *vars_lit L* $\subseteq$ *dom* $\sigma$ $\Longrightarrow$ *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 σ ≡ subsumes_modulo (mset Ls) (mset Ks) σ

lemma vars_clause_add_mset[simp]: vars_clause (add_mset L C) = vars_lit L ∪ vars_clause C
  unfolding vars_clause_def by auto

lemma subsumes_list_modulo_Cons: subsumes_list_modulo (L # Ls) Ks σ ⟷
  (∃ K ∈ set Ks. ∃ τ. extends_subst σ τ ∧ dom τ = vars_lit L ∪ dom σ ∧ L ·lit (subst_of_map Var τ) = K
    ∧ subsumes_list_modulo Ls (remove1 K Ks) τ)
  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 _ λx. if x ∈ vars_lit L ∪ dom σ then τ x else None], intro conjI exI[of _ τ])
      (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 τ τ')
  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 ⟹
   f = g ∧ length ss = length ts ∧ eqs = zip ss ts ∧
   (⋃(t, u)∈set ((Fun f ss, Fun g ts) # P). vars_term t) =
   (⋃(t, u)∈set (eqs @ P). 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 σ = Some τ ⟹
  extends_subst σ τ ∧ dom τ = (⋃(t, u)∈set tus. vars_term t) ∪ dom σ ∧
  (∀(t,u)∈set tus. t · subst_of_map Var τ = u)
proof (induct tus σ rule: match_term_list.induct)
  case (2 x t P σ)
  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 σ)
  from 3(2) obtain eqs where decompose (Fun f ss) (Fun g ts) = Some eqs
    match_term_list (eqs @ P) σ = Some τ by (auto split: option.splits)
  with 3(1)[OF this] show ?case
   proof (elim decompose_Some_var_terms[where P = P, elim_format] 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 σ = None ⟹
   extends_subst σ τ ⟹ dom τ = (⋃(t, u)∈set tus. vars_term t) ∪ dom σ ⟹
   (∃(t,u)∈set tus. t · subst_of_map Var τ ≠ u)
proof (induct tus σ arbitrary: τ rule: match_term_list.induct)
  case (2 x t P σ)
  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 σ)
  show ?case
  proof (cases decompose (Fun f ss) (Fun g ts) = None)
    case False
    with 3(2) obtain eqs where decompose (Fun f ss) (Fun g ts) = Some eqs
      match_term_list (eqs @ P) σ = None by (auto split: option.splits)
    with 3(1)[OF this 3(3) trans[OF 3(4) arg_cong[of _ _ λx. x ∪ dom σ]]] 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 3 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 σ τ extends_subst σ ϱ and
    dom: dom τ = vars_term t ∪ dom σ dom ϱ = vars_term t ∪ dom σ and
    eq: t · subst_of_map Var ϱ = t · subst_of_map Var τ
  shows ϱ = τ
proof
  fix x
  consider (a) x ∈ dom σ | (b) x ∈ vars_term t | (c) x ∉ dom τ using assms by auto
  then show ϱ x = τ 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 ϱ x = None τ x = None using dom by auto
    then show ?thesis by simp
  qed
qed

lemma subsumes_list_alt:
  subsumes_list Ls Ks σ ⟷ subsumes_list_modulo Ls Ks σ
proof (induction Ls Ks σ rule: subsumes_list.induct[case_names Nil Cons])
  case (Cons L Ls Ks σ)
  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 τ)
    then show ?case
    proof (cases match_term_list [(atm_of L, atm_of K)] σ)
      case None
      with RL show ?thesis
        by (auto simp: Cons.IH dest!: match_term_list_complete)
    next
      case (Some τ')
      with RL show ?thesis
        using unique_extends_subst[of σ τ τ' 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** $\sigma$ **where** *subst_cls* (*mset Ls*) $\sigma$ ⊆# *mset Ks* **unfolding** *subsumes_def* **by** *blast*
**moreover define** $\tau$ **where** $\tau$ = ($\lambda x$. *if* $x \in$ *vars_clause* (*mset Ls*) *then Some* ($\sigma$ $x$) *else None*)
**ultimately show** *subsumes_list_modulo Ls Ks Map.empty*
**unfolding** *subsumes_modulo_def*
**by** (*subst* (*asm*) *same_on_vars_clause*[*of _ $\sigma$ subst_of_map Var $\tau$*])
(*auto intro*!: *exI*[*of _ $\tau$*] *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* ∧ ¬ *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*) $\sigma \Longrightarrow$ *subsumes_list Ls Ks* $\sigma$
**proof** (*induction Ls arbitrary*: *Ks* $\sigma$)
**case** (*Cons L Ls*)
**from** *Cons.prems* **show** *?case*
**by** (*auto dest*!: *Cons.IH simp*: *filter_remove1*[*symmetric*] *split*: *option.splits*)
**qed** *simp*

**lemma** *subsumes_list_filterI*:
**assumes** *match*: ($\bigwedge L\ K\ \sigma\ \tau$. $L \in$ *set Ls* $\Longrightarrow$
*match_term_list* [(*atm_of L*, *atm_of K*)] $\sigma$ = *Some* $\tau \Longrightarrow$ *is_pos L* = *is_pos K* $\Longrightarrow$ *P K*)
**shows** *subsumes_list Ls Ks* $\sigma \Longrightarrow$ *subsumes_list Ls* (*filter P Ks*) $\sigma$
**using** *assms* **proof** (*induction Ls Ks* $\sigma$ *rule*: *subsumes_list.induct*[*case_names Nil Cons*])
**case** (*Cons L Ls Ks* $\sigma$)
**from** *Cons.prems* **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*)] $\sigma$ = *Some* $\tau \Longrightarrow$ *is_pos L* = *is_pos K* $\Longrightarrow$ *leq L K*)
**shows** *subsumes_list* (*L* # *Ls*) (*filter* (*leq L*) *Ks*) $\sigma \longleftrightarrow$ *subsumes_list* (*L* # *Ls*) *Ks* $\sigma$
**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*::*linorder*, '*v*) *term* ⇒ ('*f*, '*v*) *term* ⇒ *bool* **where**
*leq_head t u* = (*case* (*root t*, *root u*) *of*

```
    (None, _) ⇒ True
  | (_, None) ⇒ False
  | (Some f, Some g) ⇒ f ≤ g)
definition leq_lit L K = (case (K, L) of
    (Neg _, Pos _) ⇒ True
  | (Pos _, Neg _) ⇒ False
  | _ ⇒ leq_head (atm_of L) (atm_of K))

lemma transp_leq_lit[simp]: transp leq_lit
  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.prems 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.prems 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.prems 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.prems 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
```