

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

Anders Schlichtkrull, Jasmin Christian Blanchette, and Dmitriy Traytel

March 17, 2025

## 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

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>A Fair Ordered Resolution Prover for First-Order Clauses with Weights</b>	<b>1</b>
<b>3</b>	<b>A Deterministic Ordered Resolution Prover for First-Order Clauses</b>	<b>15</b>
3.1	Library . . . . .	16
3.2	Prover . . . . .	16
<b>4</b>	<b>Integration of IsaFoR Terms and the Knuth–Bendix Order</b>	<b>43</b>
<b>5</b>	<b>An Executable Algorithm for Clause Subsumption</b>	<b>56</b>
5.1	Naive Implementation of Clause Subsumption . . . . .	56
5.2	Optimized Implementation of Clause Subsumption . . . . .	60
5.3	Definition of Deterministic QuickSort . . . . .	60
<b>6</b>	<b>An Executable Simple Ordered Resolution Prover for First-Order Clauses</b>	<b>61</b>

## 1 Introduction

Bachmair and Ganzinger’s “Resolution Theorem Proving” chapter in the *Handbook of Automated Reasoning* is the standard reference on the topic. It defines a general framework for propositional and first-order resolution-based theorem proving. Resolution forms the basis for superposition, the calculus implemented in many popular automatic theorem provers.

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

We refer to the following conference paper for details:

Anders Schlichtkrull, Jasmin Christian Blanchette, Dmitriy Traytel:  
A verified prover based on ordered resolution.  
CPP 2019: 152-165  
[http://matryoshka.gforge.inria.fr/pubs/fun\\_rp\\_paper.pdf](http://matryoshka.gforge.inria.fr/pubs/fun_rp_paper.pdf)

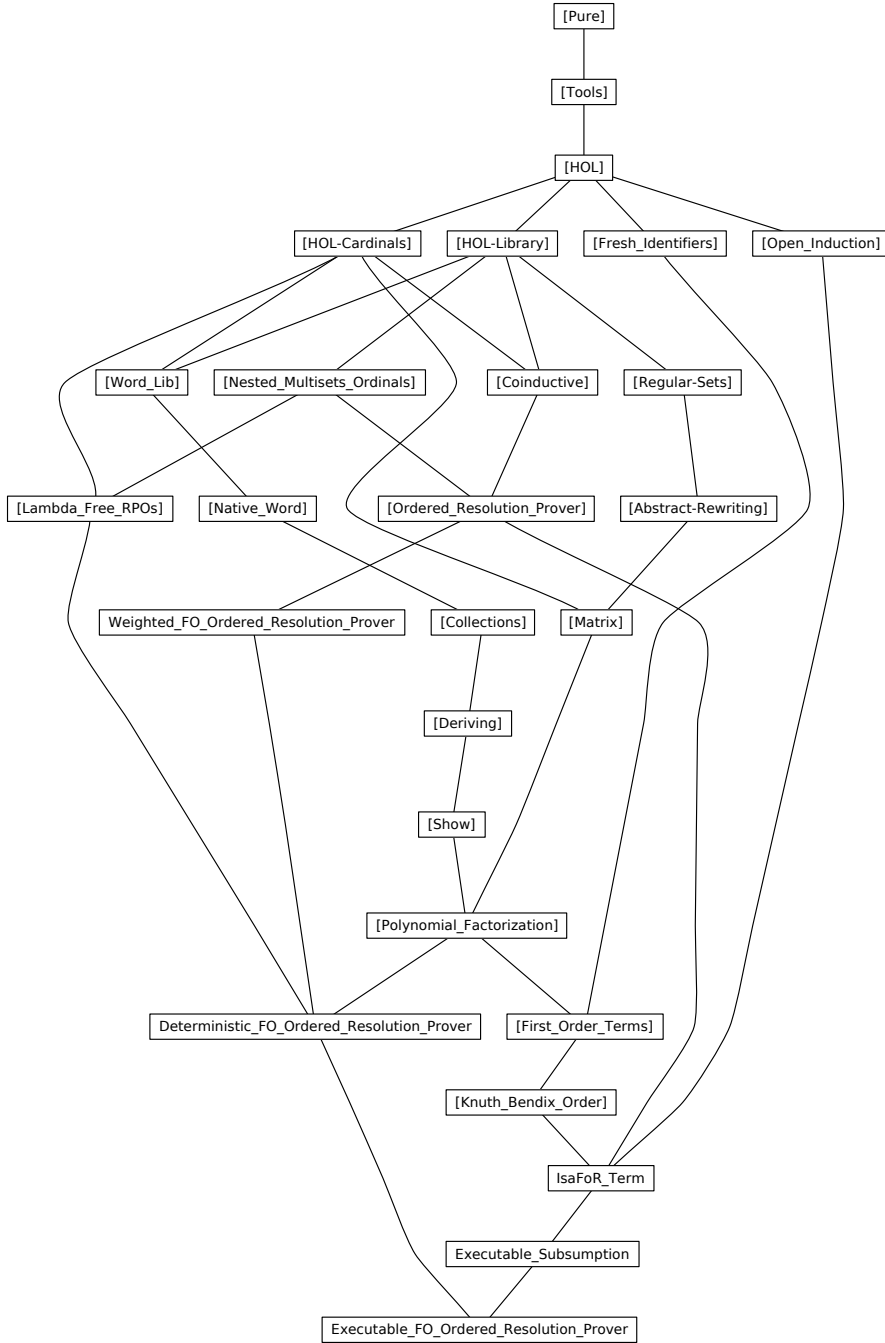


Figure 1: Theory dependency graph

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

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

```

theory Weighted_FO_Ordered_Resolution_Prover
  imports Ordered_Resolution_Prover.FO_Ordered_Resolution_Prover
begin

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

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

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

abbreviation class_of_wstate :: 'a wstate ⇒ 'a clause set where
  class_of_wstate St ≡ class_of_state (state_of_wstate St)

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

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

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

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

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

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

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

lemma of_wstate_split[simp]:
  (wN_of_wstate St, wP_of_wstate St, wQ_of_wstate St, n_of_wstate St) = St
by (cases St) auto

```

**abbreviation** *grounding\_of\_wstate* :: 'a wstate  $\Rightarrow$  'a clause set **where**  
*grounding\_of\_wstate* St  $\equiv$  *grounding\_of\_state* (state\_of\_wstate St)

**abbreviation** *Liminf\_wstate* :: 'a wstate llist  $\Rightarrow$  'a state **where**  
*Liminf\_wstate* Sts  $\equiv$  *Liminf\_state* (lmap state\_of\_wstate Sts)

**lemma** *timestamp\_le\_weight*:  $n \leq \text{weight } (C, n)$   
**by** (*induct* n, *simp*, *metis* *weight\_mono*[of k *Suc* k **for** k] *Suc\_le\_eq* *le\_less* *le\_trans*)

**inductive** *weighted\_RP* :: 'a wstate  $\Rightarrow$  'a wstate  $\Rightarrow$  bool (**infix**  $\langle \rightsquigarrow_w \rangle$  50) **where**  
*tautology\_deletion*:  $\text{Neg } A \in \# C \Longrightarrow \text{Pos } A \in \# C \Longrightarrow (N + \{\#(C, i)\}, P, Q, n) \rightsquigarrow_w (N, P, Q, n)$   
| *forward\_subsumption*:  $D \in \# \text{image\_mset fst } (P + Q) \Longrightarrow \text{subsumes } D C \Longrightarrow$   
 $(N + \{\#(C, i)\}, P, Q, n) \rightsquigarrow_w (N, P, Q, n)$   
| *backward\_subsumption\_P*:  $D \in \# \text{image\_mset fst } N \Longrightarrow C \in \# \text{image\_mset fst } P \Longrightarrow$   
 $\text{strictly\_subsumes } D C \Longrightarrow (N, P, Q, n) \rightsquigarrow_w (N, \{\#(E, k) \in \# P. E \neq C\}, Q, n)$   
| *backward\_subsumption\_Q*:  $D \in \# \text{image\_mset fst } N \Longrightarrow \text{strictly\_subsumes } D C \Longrightarrow$   
 $(N, P, Q + \{\#(C, i)\}, n) \rightsquigarrow_w (N, P, Q, n)$   
| *forward\_reduction*:  $D + \{\#L'\} \in \# \text{image\_mset fst } (P + Q) \Longrightarrow -L = L' \cdot l \sigma \Longrightarrow D \cdot \sigma \subseteq \# C \Longrightarrow$   
 $(N + \{\#(C + \{\#L'\}, i)\}, P, Q, n) \rightsquigarrow_w (N + \{\#(C, i)\}, P, Q, n)$   
| *backward\_reduction\_P*:  $D + \{\#L'\} \in \# \text{image\_mset fst } N \Longrightarrow -L = L' \cdot l \sigma \Longrightarrow D \cdot \sigma \subseteq \# C \Longrightarrow$   
 $(\forall j. (C + \{\#L'\}, j) \in \# P \longrightarrow j \leq i) \Longrightarrow$   
 $(N, P + \{\#(C + \{\#L'\}, i)\}, Q, n) \rightsquigarrow_w (N, P + \{\#(C, i)\}, Q, n)$   
| *backward\_reduction\_Q*:  $D + \{\#L'\} \in \# \text{image\_mset fst } N \Longrightarrow -L = L' \cdot l \sigma \Longrightarrow D \cdot \sigma \subseteq \# C \Longrightarrow$   
 $(N, P, Q + \{\#(C + \{\#L'\}, i)\}, n) \rightsquigarrow_w (N, P + \{\#(C, i)\}, Q, n)$   
| *clause\_processing*:  $(N + \{\#(C, i)\}, P, Q, n) \rightsquigarrow_w (N, P + \{\#(C, i)\}, Q, n)$   
| *inference\_computation*:  $(\forall (D, j) \in \# P. \text{weight } (C, i) \leq \text{weight } (D, j)) \Longrightarrow$   
 $N = \text{mset\_set } ((\lambda D. (D, n)) \text{ `concls\_of$   
 $(\text{inference\_system.inferences\_between } (\text{ord\_FO\_}\Gamma S) (\text{set\_mset } (\text{image\_mset fst } Q)) C)) \Longrightarrow$   
 $(\{\#\}, P + \{\#(C, i)\}, Q, n) \rightsquigarrow_w (N, \{\#(D, j) \in \# P. D \neq C\}, Q + \{\#(C, i)\}, \text{Suc } n)$

**lemma** *weighted\_RP\_imp\_RP*:  $St \rightsquigarrow_w St' \Longrightarrow \text{state\_of\_wstate } St \rightsquigarrow \text{state\_of\_wstate } St'$

**proof** (*induction* rule: *weighted\_RP.induct*)

**case** (*backward\_subsumption\_P* D N C P Q n)

**show** ?case

**by** (rule *arg\_cong2*[*THEN* *iffD1*, of  $\_ \_ \_ \_ (\rightsquigarrow)$ , *OF*  $\_ \_ \_ \_ \text{RP.backward\_subsumption\_P}$ [of D *fst* 'set\_mset N C *fst* 'set\_mset P - {C} *fst* 'set\_mset Q]])  
*(use backward\_subsumption\_P in auto)*

**next**

**case** (*inference\_computation* P C i N n Q)

**show** ?case

**by** (rule *arg\_cong2*[*THEN* *iffD1*, of  $\_ \_ \_ \_ (\rightsquigarrow)$ , *OF*  $\_ \_ \_ \_ \text{RP.inference\_computation}$ [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*:  $\neg (\{\#\}, \{\#\}, Q, n) \rightsquigarrow_w St$   
**by** (*auto elim: weighted\_RP.cases*)

**context**

**fixes**

*Sts* :: 'a wstate llist

**assumes**

*full\_deriv*: *full\_chain* ( $\rightsquigarrow_w$ ) *Sts* **and**

*empty\_P0*: *P\_of\_wstate* (lhd *Sts*) = {} **and**

*empty\_Q0*: *Q\_of\_wstate* (lhd *Sts*) = {}

**begin**

**lemma** *finite\_Sts0*: *finite* (*class\_of\_wstate* (lhd *Sts*))

**by** (*cases* lhd *Sts*) *auto*

**lemmas** *deriv* = *full\_chain\_imp\_chain*[*OF full\_deriv*]

**lemmas** *lhd\_lmap\_Sts* = *lmap\_sel(1)*[*OF chain\_not\_null*[*OF deriv*]]

**lemma** *deriv\_RP*: *chain* ( $\rightsquigarrow$ ) (*lmap state\_of\_wstate Sts*)

**using** *deriv\_weighted\_RP\_imp\_RP* **by** (*metis chain\_lmap*)

**lemma** *finite\_Sts0\_RP*: *finite* (*class\_of\_state* (*lhd* (*lmap state\_of\_wstate Sts*)))

**using** *finite\_Sts0\_chain\_length\_pos*[*OF deriv*] **by** *auto*

**lemma** *empty\_P0\_RP*: *P\_of\_state* (*lhd* (*lmap state\_of\_wstate Sts*)) = {}

**using** *empty\_P0\_chain\_length\_pos*[*OF deriv*] **by** *auto*

**lemma** *empty\_Q0\_RP*: *Q\_of\_state* (*lhd* (*lmap state\_of\_wstate Sts*)) = {}

**using** *empty\_Q0\_chain\_length\_pos*[*OF deriv*] **by** *auto*

**lemmas** *Sts\_thms* = *deriv\_RP finite\_Sts0\_RP empty\_P0\_RP empty\_Q0\_RP*

**theorem** *weighted\_RP\_model*:

*St*  $\rightsquigarrow_w$  *St'*  $\implies I \models_s$  *grounding\_of\_wstate St'*  $\iff I \models_s$  *grounding\_of\_wstate St*

**using** *RP\_model Sts\_thms weighted\_RP\_imp\_RP* **by** (*simp only: comp\_def*)

**abbreviation** *S\_gQ* :: '*a clause*  $\implies$  '*a clause* **where**

*S\_gQ*  $\equiv$  *S\_Q* (*lmap state\_of\_wstate Sts*)

**interpretation** *sq*: *selection S\_gQ*

**unfolding** *S\_Q\_def* **using** *S\_M\_selects\_subseteq S\_M\_selects\_neg\_lits selection\_axioms*  
**by** *unfold\_locales auto*

**interpretation** *gd*: *ground\_resolution\_with\_selection S\_gQ*

**by** *unfold\_locales*

**interpretation** *src*: *standard\_redundancy\_criterion\_reductive gd.ord\_Γ*

**by** *unfold\_locales*

**interpretation** *src*: *standard\_redundancy\_criterion\_counterex\_reducing gd.ord\_Γ*

*ground\_resolution\_with\_selection.INTERP S\_gQ*

**by** *unfold\_locales*

**lemmas** *ord\_Γ\_saturated\_upto\_def* = *src.saturated\_upto\_def*

**lemmas** *ord\_Γ\_saturated\_upto\_complete* = *src.saturated\_upto\_complete*

**lemmas** *ord\_Γ\_contradiction\_Rf* = *src.contradiction\_Rf*

**theorem** *weighted\_RP\_sound*:

**assumes** {#}  $\in$  *class\_of\_state* (*Liminf\_wstate Sts*)

**shows**  $\neg$  *satisfiable* (*grounding\_of\_wstate* (*lhd Sts*))

**by** (*rule RP\_sound*[*OF deriv\_RP assms, unfolded lhd\_lmap\_Sts*])

**abbreviation** *RP\_filtered\_measure* :: ('*a wclause*  $\implies$  *bool*)  $\implies$  '*a wstate*  $\implies$  *nat*  $\times$  *nat*  $\times$  *nat* **where**

*RP\_filtered\_measure*  $\equiv$   $\lambda p$  (*N*, *P*, *Q*, *n*).

(*sum\_mset* (*image\_mset* ( $\lambda(C, i). Suc$  (*size C*)) {#*Di*  $\in$  # *N* + *P* + *Q*. *p Di*#}),

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

**abbreviation** *RP\_combined\_measure* :: *nat*  $\implies$  '*a wstate*  $\implies$  *nat*  $\times$  (*nat*  $\times$  *nat*  $\times$  *nat*)  $\times$  (*nat*  $\times$  *nat*  $\times$  *nat*) **where**

*RP\_combined\_measure*  $\equiv$   $\lambda w$  *St*.

(*w* + 1 - *n\_of\_wstate St*, *RP\_filtered\_measure* ( $\lambda(C, i). i \leq w$ ) *St*,

*RP\_filtered\_measure* ( $\lambda Ci. True$ ) *St*)

**abbreviation** (*input*) *RP\_filtered\_relation* :: ((*nat*  $\times$  *nat*  $\times$  *nat*)  $\times$  (*nat*  $\times$  *nat*  $\times$  *nat*)) *set* **where**

*RP\_filtered\_relation*  $\equiv$  *natLess*  $\langle *lex* \rangle$  *natLess*  $\langle *lex* \rangle$  *natLess*

**abbreviation** (*input*) *RP\_combined\_relation* :: ((*nat*  $\times$  ((*nat*  $\times$  (*nat*  $\times$  *nat*  $\times$  *nat*)  $\times$  (*nat*  $\times$  *nat*  $\times$  *nat*)))  $\times$  (*nat*  $\times$  ((*nat*  $\times$  (*nat*  $\times$  *nat*  $\times$  *nat*)  $\times$  (*nat*  $\times$  *nat*  $\times$  *nat*)))))) *set* **where**

*RP\_combined\_relation*  $\equiv$  *natLess*  $\langle *lex* \rangle$  *RP\_filtered\_relation*  $\langle *lex* \rangle$  *RP\_filtered\_relation*

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

**lemma**

$\text{wf\_RP\_filtered\_relation}$ :  $\text{wf RP\_filtered\_relation}$  **and**  
 $\text{wf\_RP\_combined\_relation}$ :  $\text{wf RP\_combined\_relation}$   
**unfolding**  $\text{natLess\_def}$  **using**  $\text{wf\_less wf\_mult}$  **by**  $\text{auto}$

**lemma**  $\text{multiset\_sum\_of\_Suc\_f\_monotone}$ :  $N \subset\# M \implies (\sum x \in\# N. \text{Suc } (f x)) < (\sum x \in\# M. \text{Suc } (f x))$

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

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

**next**

**case** ( $\text{add } x N$ )  
**from**  $\text{this}(2)$  **have**  $(\sum y \in\# N. \text{Suc } (f y)) < (\sum y \in\# M - \{x\}. \text{Suc } (f y))$   
**using**  $\text{add}(1)[\text{of } M - \{x\}]$  **by** ( $\text{simp add: insert\_union\_subset\_iff}$ )  
**moreover have**  $\text{add\_mset } x (\text{remove1\_mset } x M) = M$   
**by** ( $\text{meson add.premis add\_mset\_remove\_trivial\_If mset\_subset\_insertD}$ )  
**ultimately show**  $?case$   
**by** ( $\text{metis (no\_types) add.commute add\_less\_cancel\_right sum\_mset.insert}$ )

**qed**

**lemma**  $\text{multiset\_sum\_monotone\_f'}$ :

**assumes**  $CC \subset\# DD$   
**shows**  $(\sum (C, i) \in\# CC. \text{Suc } (f C)) < (\sum (C, i) \in\# DD. \text{Suc } (f C))$   
**using**  $\text{multiset\_sum\_of\_Suc\_f\_monotone}[OF \text{ assms, of } f \circ \text{fst}]$   
**by** ( $\text{metis (mono\_tags) comp\_apply image\_mset\_cong2 split\_beta}$ )

**lemma**  $\text{filter\_mset\_strict\_subset}$ :

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

**proof** –

**have**  $\text{subsetq: } \{E \in\# M. p E\} \subseteq\# M$   
**by**  $\text{auto}$   
**have**  $\text{count } \{E \in\# M. p E\} x = 0$   
**using**  $\text{assms}$  **by**  $\text{auto}$   
**moreover have**  $0 < \text{count } M x$   
**using**  $\text{assms}$  **by**  $\text{auto}$   
**ultimately have**  $\text{lt\_count: } \text{count } \{y \in\# M. p y\} x < \text{count } M x$   
**by**  $\text{auto}$   
**then show**  $?thesis$   
**using**  $\text{subsetq}$  **by** ( $\text{metis less\_not\_refl2 subset\_mset.le\_neq\_trans}$ )

**qed**

**lemma**  $\text{weighted\_RP\_measure\_decreasing\_N}$ :

**assumes**  $St \rightsquigarrow_w St'$  **and**  $(C, l) \in\# \text{wN\_of\_wstate } St$   
**shows**  $(\text{RP\_filtered\_measure } (\lambda Ci. \text{True}) St', \text{RP\_filtered\_measure } (\lambda Ci. \text{True}) St)$   
 $\in \text{RP\_filtered\_relation}$   
**using**  $\text{assms}$  **proof** ( $\text{induction rule: weighted\_RP.induct}$ )  
**case** ( $\text{backward\_subsumption\_P } D N C' P Q n$ )

**then obtain**  $i'$  **where**  $(C', i') \in\# P$   
**by** *auto*  
**then have**  $\{\#(E, k) \in\# P. E \neq C'\#\} \subset\# P$   
**using** *filter\_mset\_strict\_subset*[of  $(C', i') P \lambda X. \neg \text{fst } X = C'$ ]  
**by** (*metis* (*mono\_tags*, *lifting*) *filter\_mset\_cong fst\_conv prod.case\_eq\_if*)  
**then have**  $(\sum (C, i) \in\# \{\#(E, k) \in\# P. E \neq C'\#\}. \text{Suc } (\text{size } C)) < (\sum (C, i) \in\# P. \text{Suc } (\text{size } C))$   
**using** *multiset\_sum\_monotone\_f'*[of  $\{\#(E, k) \in\# P. E \neq C'\#\} P \text{size}$ ] **by** *metis*  
**then show** *?case*  
**unfolding** *natLess\_def* **by** *auto*  
**qed** (*auto simp: natLess\_def*)

**lemma** *weighted\_RP\_measure\_decreasing\_P*:

**assumes**  $St \rightsquigarrow_w St'$  **and**  $(C, i) \in\# wP\_of\_wstate St$   
**shows**  $(RP\_combined\_measure (weight (C, i)) St', RP\_combined\_measure (weight (C, i)) St)$   
 $\in RP\_combined\_relation$   
**using** *assms proof* (*induction rule: weighted\_RP.induct*)  
**case** (*backward\_subsumption\_P D N C' P Q n*)

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

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

**then have**  $P'_sub\_P: P' \subset\# P$   
**unfolding**  $P'_def$  **using** *filter\_mset\_strict\_subset*[of  $(C', i') P \lambda Dj. \text{fst } Dj \neq C'$ ]  
**by** (*metis* (*no\_types*, *lifting*) *filter\_mset\_cong fst\_conv prod.case\_eq\_if*)

**have**  $P'_subeq\_P\_filter$ :  
 $\{\#(Ca, ia) \in\# P'. ia \leq weight (C, i)\#\} \subseteq\# \{\#(Ca, ia) \in\# P. ia \leq weight (C, i)\#\}$   
**using**  $P'_sub\_P$  **by** (*auto intro: multiset\_filter\_mono*)

**have**  $\text{fst3 } (RP\_combined\_measure (weight (C, i)) St')$   
 $\leq \text{fst3 } (RP\_combined\_measure (weight (C, i)) St)$   
**unfolding**  $St'_def St\_def$  **by** *auto*  
**moreover have**  $(\sum (C, i) \in\# \{\#(Ca, ia) \in\# P'. ia \leq weight (C, i)\#\}. \text{Suc } (\text{size } C))$   
 $\leq (\sum x \in\# \{\#(Ca, ia) \in\# P. ia \leq weight (C, i)\#\}. \text{case } x \text{ of } (C, i) \Rightarrow \text{Suc } (\text{size } C))$   
**using**  $P'_subeq\_P\_filter$  **by** (*rule sum\_image\_mset\_mono*)  
**then have**  $\text{fst3 } (snd3 (RP\_combined\_measure (weight (C, i)) St'))$   
 $\leq \text{fst3 } (snd3 (RP\_combined\_measure (weight (C, i)) St))$   
**unfolding**  $St'_def St\_def$  **by** *auto*  
**moreover have**  $\text{snd3 } (snd3 (RP\_combined\_measure (weight (C, i)) St'))$   
 $\leq \text{snd3 } (snd3 (RP\_combined\_measure (weight (C, i)) St))$   
**unfolding**  $St'_def St\_def$  **by** *auto*  
**moreover from**  $P'_subeq\_P\_filter$  **have**  $\text{size } \{\#(Ca, ia) \in\# P'. ia \leq weight (C, i)\#\}$   
 $\leq \text{size } \{\#(Ca, ia) \in\# P. ia \leq weight (C, i)\#\}$   
**by** (*simp add: size\_mset\_mono*)  
**then have**  $\text{trd3 } (snd3 (RP\_combined\_measure (weight (C, i)) St'))$   
 $< \text{trd3 } (snd3 (RP\_combined\_measure (weight (C, i)) St))$   
**unfolding**  $St'_def St\_def$  **unfolding**  $\text{fst\_def snd\_def}$  **by** *auto*  
**moreover from**  $P'_sub\_P$  **have**  $(\sum (C, i) \in\# P'. \text{Suc } (\text{size } C)) < (\sum (C, i) \in\# P. \text{Suc } (\text{size } C))$   
**using** *multiset\_sum\_monotone\_f'*[of  $\{\#(E, k) \in\# P. E \neq C'\#\} P \text{size}$ ] **unfolding**  $P'_def$  **by** *metis*  
**then have**  $\text{fst3 } (trd3 (RP\_combined\_measure (weight (C, i)) St'))$   
 $< \text{fst3 } (trd3 (RP\_combined\_measure (weight (C, i)) St))$   
**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**  $weight (C, i) + 1 - n > weight (C, i) + 1 - \text{Suc } n$

```

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

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

have fin: finite concls
  unfolding concls_def using finite_ord_FO_resolution_inferences_between by auto

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

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

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

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

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

```



```

case (backward_reduction_P D L' N L 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**  $i < \text{llength } xs$  **and**  $x \in \# (\text{lnth } xs \ i)$   
**shows**  $x \in \text{Sup\_llist } (\text{lmap } \text{set\_mset } (\text{ldrop } (\text{enat } i) \ xs))$   
**using** *assms* **by** (*metis* (*no\_types*) *ldrop\_eq\_LConsD* *ldroprn\_0* *llist.simps(13)* *contra\_subsetD*  
*ldrop\_enat* *ldroprn\_Suc\_conv* *ldroprn\_lnth\_0* *lnth\_lmap* *lnth\_subset\_Sup\_llist*)

**lemma** *persistent\_wclause\_in\_P\_if\_persistent\_clause\_in\_P*:

**assumes**  $C \in \text{Liminf\_llist } (\text{lmap } P\_of\_state (\text{lmap } \text{state\_of\_wstate } Sts))$   
**shows**  $\exists i. (C, i) \in \text{Liminf\_llist } (\text{lmap } (\text{set\_mset} \circ wP\_of\_wstate) Sts)$

**proof** –

**obtain**  $t\_C$  **where**  $t\_C\_p$ :  
 $\text{enat } t\_C < \text{llength } Sts$   
 $\wedge t. t\_C \leq t \implies t < \text{llength } Sts \implies C \in P\_of\_state (\text{state\_of\_wstate } (\text{lnth } Sts \ t))$   
**using** *assms* **unfolding** *Liminf\_llist\_def* **by** *auto*  
**then obtain**  $i$  **where**  $i\_p$ :  
 $(C, i) \in \# wP\_of\_wstate (\text{lnth } Sts \ t\_C)$   
**using**  $t\_C\_p$  **by** (*cases* *lnth* *Sts*  $t\_C$ ) *force*

**have**  $C_i \text{ in\_nth\_wP} : \exists i. (C, i) \in \# wP\_of\_wstate (\text{lnth } Sts \ (t\_C + t))$  **if**  $t\_C + t < \text{llength } Sts$   
**for**  $t$   
**using** *that*  $t\_C\_p(2)$ [*of*  $t\_C + \_$ ] **by** (*cases* *lnth* *Sts*  $(t\_C + t)$ ) *force*

**define**  $\text{in\_Sup\_wP} :: \text{nat} \implies \text{bool}$  **where**

$\text{in\_Sup\_wP} = (\lambda i. (C, i) \in \text{Sup\_llist } (\text{lmap } (\text{set\_mset} \circ wP\_of\_wstate) (\text{ldrop } t\_C \ Sts)))$

**have**  $\text{in\_Sup\_wP } i$

**using**  $i\_p$  *assms(1)* *in\_lnth\_in\_Supremum* *ldrop*[*of*  $t\_C$  *lmap*  $wP\_of\_wstate \ Sts \ (C, i)$ ]  $t\_C\_p$   
**by** (*simp* *add*: *in\_Sup\_wP\_def* *llist.map\_comp*)

**then obtain**  $j$  **where**  $j\_p$ :  $\text{is\_least } \text{in\_Sup\_wP } j$

**unfolding** *in\_Sup\_wP\_def*[*symmetric*] **using** *least\_exists* **by** *metis*

**then have**  $\forall i. (C, i) \in \text{Sup\_llist } (\text{lmap } (\text{set\_mset} \circ wP\_of\_wstate) (\text{ldrop } t\_C \ Sts)) \implies j \leq i$

**unfolding** *is\_least\_def* *in\_Sup\_wP\_def* **using** *not\_less* **by** *blast*

**then have**  $j$  *smallest*:

$\wedge i t. \text{enat } (t\_C + t) < \text{llength } Sts \implies (C, i) \in \# wP\_of\_wstate (\text{lnth } Sts \ (t\_C + t)) \implies j \leq i$

**unfolding** *comp\_def*

**by** (*smt* (*verit*) *add commute* *ldrop\_enat* *ldrop\_eq\_LConsD* *ldrop\_ldrop* *ldroprn\_Suc\_conv* *ldroprn\_plus\_enat\_simps(1)* *lnth\_ldroprn\_Sup\_llist\_def* *UN\_I* *ldrop\_lmap* *llength\_lmap* *lnth\_lmap* *mem\_Collect\_eq*)

**from**  $j\_p$  **have**  $\exists t\_Cj. t\_Cj < \text{llength } (\text{ldrop } (\text{enat } t\_C) \ Sts)$

$\wedge (C, j) \in \# wP\_of\_wstate (\text{lnth } (\text{ldrop } t\_C \ Sts) \ t\_Cj)$

**unfolding** *in\_Sup\_wP\_def* *Sup\_llist\_def* *is\_least\_def* **by** *simp*

**then obtain**  $t\_Cj$  **where**  $j\_p$ :

$(C, j) \in \# wP\_of\_wstate (\text{lnth } Sts \ (t\_C + t\_Cj))$

$\text{enat } (t\_C + t\_Cj) < \text{llength } Sts$

**by** (*smt* (*verit*) *add commute* *ldrop\_enat* *ldrop\_eq\_LConsD* *ldrop\_ldrop* *ldroprn\_Suc\_conv* *ldroprn\_plus\_enat\_simps(1)* *lhd\_ldroprn*)

**have**  $C_i$  *stays*:

$t\_C + t\_Cj + t < \text{llength } Sts \implies (C, j) \in \# wP\_of\_wstate (\text{lnth } Sts \ (t\_C + t\_Cj + t))$  **for**  $t$

**proof** (*induction*  $t$ )

**case**  $0$

**then show** *?case*

**using**  $j\_p$  **by** (*simp* *add*: *add commute*)

**next**

**case** (*Suc*  $t$ )

**have**  $\text{any\_Ck\_in\_wP} : j \leq k$  **if**  $(C, k) \in \# wP\_of\_wstate (\text{lnth } Sts \ (t\_C + t\_Cj + t))$  **for**  $k$

**using** *that*  $j\_p$   $j$  *smallest* *Suc*

**by** (*smt* (*verit*) *Suc\_ile\_eq* *add commute* *add.left\_commute* *add\_Suc* *less\_imp\_le* *plus\_enat\_simps(1)* *the\_enat.simps*)

**from** *Suc* **have**  $Cj \text{ in\_wP} : (C, j) \in \# wP\_of\_wstate (\text{lnth } Sts \ (t\_C + t\_Cj + t))$

**by** (*metis* (*no\_types*, *opaque\_lifting*) *Suc\_ile\_eq* *add commute* *add\_Suc\_right* *less\_imp\_le*)

**moreover have**  $C \in P\_of\_state (\text{state\_of\_wstate } (\text{lnth } Sts \ (\text{Suc } (t\_C + t\_Cj + t))))$

**using**  $t\_C\_p(2)$  *Suc.prem*s **by** *auto*

**then have**  $\exists k. (C, k) \in \# wP\_of\_wstate (\text{lnth } Sts \ (\text{Suc } (t\_C + t\_Cj + t)))$

**by** (*smt* (*verit*) *Suc.prem*s  $C_i \text{ in\_nth\_wP}$  *add commute* *add.left\_commute* *add\_Suc\_right* *enat\_ord\_code(4)*)

**ultimately have**  $(C, j) \in \# wP\_of\_wstate (lnth\ Sts (Suc (t\_C + t\_Cj + t)))$   
**using** *preserve\_min\_P\_Sts Cj\_in\_wP any\_Ck\_in\_wP Suc.prem* **by force**  
**then have**  $(C, j) \in \# lnth (lmap\ wP\_of\_wstate\ Sts) (Suc (t\_C + t\_Cj + t))$   
**using** *Suc.prem* **by auto**  
**then show** *?case*  
**by** (*smt (verit) Suc.prem add commute add\_Suc\_right lnth\_lmap*)  
**qed**  
**then have**  $(\bigwedge t. t\_C + t\_Cj \leq t \implies t < llength (lmap (set\_mset \circ wP\_of\_wstate) Sts) \implies (C, j) \in \# wP\_of\_wstate (lnth\ Sts\ t))$   
**using** *Ci\_stays[of\_ - (t\_C + t\_Cj)]* **by** (*metis le\_add\_diff\_inverse llength\_lmap*)  
**then have**  $(C, j) \in Liminf\_llist (lmap (set\_mset \circ wP\_of\_wstate) Sts)$   
**unfolding** *Liminf\\_llist\_def* **using** *j\_p* **by auto**  
**then show**  $\exists i. (C, i) \in Liminf\_llist (lmap (set\_mset \circ wP\_of\_wstate) Sts)$   
**by auto**  
**qed**

**lemma** *lfinite\_not\_LNil\_nth\_llast*:  
**assumes** *lfinite Sts and Sts  $\neq$  LNil*  
**shows**  $\exists i < llength\ Sts. lnth\ Sts\ i = llast\ Sts \wedge (\forall j < llength\ Sts. j \leq i)$   
**using** *assms* **proof** (*induction rule: lfinite.induct*)  
**case** (*lfinite\_LConsI xs x*)  
**then show** *?case*  
**proof** (*cases xs = LNil*)  
**case** *True*  
**show** *?thesis*  
**using** *True zero\_enat\_def* **by auto**  
**next**  
**case** *False*  
**then obtain** *i* **where**  
 $i\_p: enat\ i < llength\ xs \wedge lnth\ xs\ i = llast\ xs \wedge (\forall j < llength\ xs. j \leq enat\ i)$   
**using** *lfinite\_LConsI* **by auto**  
**then have**  $enat (Suc\ i) < llength (LCons\ x\ xs)$   
**by** (*simp add: Suc\_ile\_eq*)  
**moreover from** *i\_p* **have**  $lnth (LCons\ x\ xs) (Suc\ i) = llast (LCons\ x\ xs)$   
**by** (*metis gr\_implies\_not\_zero llast\_LCons llength\_inull lnth\_Suc\_LCons*)  
**moreover from** *i\_p* **have**  $\forall j < llength (LCons\ x\ xs). j \leq enat (Suc\ i)$   
**by** (*metis antisym\_conv2 eSuc\_enat eSuc\_ile\_mono ileI1 iless\_Suc\_eq llength\_LCons*)  
**ultimately show** *?thesis*  
**by auto**  
**qed**  
**qed auto**

**lemma** *fair\_if\_finite*:  
**assumes** *fin: lfinite Sts*  
**shows** *fair\_state\_seq (lmap state\_of\_wstate Sts)*  
**proof** (*rule ccontr*)  
**assume** *unfair:  $\neg$  fair\_state\_seq (lmap state\_of\_wstate Sts)*  
  
**have** *no\_inf\_from\_last:  $\forall y. \neg llast\ Sts \rightsquigarrow_w y$*   
**using** *fin full\_chain\_iff\_chain[of ( $\rightsquigarrow_w$ ) Sts] full\_deriv* **by auto**

**from** *unfair* **obtain** *C* **where**  
 $C \in Liminf\_llist (lmap\ N\_of\_state (lmap\ state\_of\_wstate\ Sts))$   
 $\cup Liminf\_llist (lmap\ P\_of\_state (lmap\ state\_of\_wstate\ Sts))$   
**unfolding** *fair\_state\_seq\_def Liminf\_state\_def* **by auto**  
**then obtain** *i* **where** *i\_p*:  
 $enat\ i < llength\ Sts$   
 $\bigwedge j. i \leq j \implies enat\ j < llength\ Sts \implies$   
 $C \in N\_of\_state (state\_of\_wstate (lnth\ Sts\ j)) \cup P\_of\_state (state\_of\_wstate (lnth\ Sts\ j))$   
**unfolding** *Liminf\\_llist\_def* **by auto**

**have** *C\_in\_llast*:  
 $C \in N\_of\_state (state\_of\_wstate (llast\ Sts)) \cup P\_of\_state (state\_of\_wstate (llast\ Sts))$

```

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

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

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

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

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

```

qed

**lemma** *N\_of\_state\_state\_of\_wstate\_wN\_of\_wstate:*

**assumes**  $C \in N\_of\_state (state\_of\_wstate St)$

**shows**  $\exists i. (C, i) \in\# wN\_of\_wstate St$

**by** (*smt* (*verit*) *N\_of\_state.elims* *assms* *eq\_fst\_iff* *fstI* *fst\_conv* *image\_iff* *of\_wstate\_split* *set\_image\_mset* *state\_of\_wstate.simps*)

**lemma** *in\_wN\_of\_wstate\_in\_N\_of\_wstate:*  $(C, i) \in\# wN\_of\_wstate St \implies C \in N\_of\_wstate St$

**by** (*metis* (*mono\_guards\_query\_query*) *N\_of\_state.simps* *fst\_conv* *image\_eqI* *of\_wstate\_split* *set\_image\_mset* *state\_of\_wstate.simps*)

**lemma** *in\_wP\_of\_wstate\_in\_P\_of\_wstate:*  $(C, i) \in\# wP\_of\_wstate St \implies C \in P\_of\_wstate St$

**by** (*metis* (*mono\_guards\_query\_query*) *P\_of\_state.simps* *fst\_conv* *image\_eqI* *of\_wstate\_split* *set\_image\_mset* *state\_of\_wstate.simps*)

**lemma** *in\_wQ\_of\_wstate\_in\_Q\_of\_wstate:*  $(C, i) \in\# wQ\_of\_wstate St \implies C \in Q\_of\_wstate St$

**by** (*metis* (*mono\_guards\_query\_query*) *Q\_of\_state.simps* *fst\_conv* *image\_eqI* *of\_wstate\_split* *set\_image\_mset* *state\_of\_wstate.simps*)

**lemma** *n\_of\_wstate\_weighted\_RP\_increasing:*  $St \rightsquigarrow_w St' \implies n\_of\_wstate St \leq n\_of\_wstate St'$

**by** (*induction rule: weighted\_RP.induct*) *auto*

**lemma** *nth\_of\_wstate\_monotonic:*

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

**shows**  $n\_of\_wstate (lnth Sts i) \leq n\_of\_wstate (lnth Sts j)$

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

**case** (*Suc*  $x$ )

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

**by** *auto*

**then have**  $n\_of\_wstate (lnth Sts (i + 1)) \leq n\_of\_wstate (lnth Sts j)$

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

**moreover have**  $i < j$

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

**then have** *Suc*  $i < llength Sts$

**using** *Suc* **by** (*metis* *enat\_ord\_simps(2)* *le\_less\_Suc\_eq* *less\_le\_trans* *not\_le*)

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

**using** *deriv chain\_lnth\_rel*[*of* ( $\rightsquigarrow_w$ ) *Sts*  $i$ ] **by** *auto*

**then have**  $n\_of\_wstate (lnth Sts i) \leq n\_of\_wstate (lnth Sts (i + 1))$

**using** *n\_of\_wstate\_weighted\_RP\_increasing*[*of* *lnth Sts*  $i$  *lnth Sts*  $(i + 1)$ ] **by** *auto*

**ultimately show** *?case*

**by** *auto*

qed *auto*

**lemma** *infinite\_chain\_relation\_measure:*

**assumes**

*measure\_decreasing:*  $\bigwedge St St'. P St \implies R St St' \implies (m St', m St) \in mR$  **and**

*non\_infer\_chain:*  $chain R (ldrop (enat k) Sts)$  **and**

*inf:*  $llength Sts = \infty$  **and**

$P: \bigwedge i. P (lnth (ldrop (enat k) Sts) i)$

**shows**  $chain (\lambda x y. (x, y) \in mR)^{-1-1} (lmap m (ldrop (enat k) Sts))$

**proof** (*rule* *lnth\_rel\_chain*)

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

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

**next**

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

**using** *inf* **by** (*auto simp: llength\_eq\_infty\_conv\_lfinite*)

{

**fix**  $j :: nat$

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

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

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

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

```

from ldrop_inf have  $R \text{ St } \text{St}'$ 
  unfolding St_def St'_def
  using non_infer_chain infinite_chain_lnth_rel[of ldrop (enat k) Sts R j] by auto
then have  $(m \text{ St}', m \text{ St}) \in mR$ 
  using measure_decreasing P' by auto
then have  $(\text{lnth } (\text{lmap } m (\text{ldrop } (\text{enat } k) \text{ Sts})) (j + 1), \text{lnth } (\text{lmap } m (\text{ldrop } (\text{enat } k) \text{ Sts})) j) \in mR$ 
  unfolding St_def St'_def using lnth_lmap
  by (simp add: local.ldrop_inf)
}
then show  $\forall j. \text{enat } (j + 1) < \text{llength } (\text{lmap } m (\text{ldrop } (\text{enat } k) \text{ Sts})) \longrightarrow$ 
   $(\lambda x y. (x, y) \in mR)^{-1-1} (\text{lnth } (\text{lmap } m (\text{ldrop } (\text{enat } k) \text{ Sts})) j)$ 
   $(\text{lnth } (\text{lmap } m (\text{ldrop } (\text{enat } k) \text{ Sts})) (j + 1))$ 
  by blast
qed

theorem weighted_RP_fair: fair_state_seq (lmap state_of_wstate Sts)
proof (rule ccontr)
  assume asm:  $\neg \text{fair\_state\_seq } (\text{lmap } \text{state\_of\_wstate } \text{Sts})$ 
  then have inff:  $\neg \text{lfinite } \text{Sts}$  using fair_if_finite
  by auto
  then have inf:  $\text{llength } \text{Sts} = \infty$ 
  using llength_eq_infty_conv_lfinite by auto
from asm obtain C where
   $C \in \text{Liminf\_llist } (\text{lmap } N\_of\_state (\text{lmap } \text{state\_of\_wstate } \text{Sts}))$ 
   $\cup \text{Liminf\_llist } (\text{lmap } P\_of\_state (\text{lmap } \text{state\_of\_wstate } \text{Sts}))$ 
  unfolding fair_state_seq_def Liminf_state_def by auto
then show False
proof
  assume  $C \in \text{Liminf\_llist } (\text{lmap } N\_of\_state (\text{lmap } \text{state\_of\_wstate } \text{Sts}))$ 
  then obtain x where  $\text{enat } x < \text{llength } \text{Sts}$ 
   $\forall xa. x \leq xa \wedge \text{enat } xa < \text{llength } \text{Sts} \longrightarrow C \in N\_of\_state (\text{state\_of\_wstate } (\text{lnth } \text{Sts } xa))$ 
  unfolding Liminf_llist_def by auto
  then have  $\exists k. \forall j. k \leq j \longrightarrow (\exists i. (C, i) \in\# wN\_of\_wstate (\text{lnth } \text{Sts } j))$ 
  unfolding Liminf_llist_def by (force simp add: inf_N_of_state_state_of_wstate_wN_of_wstate)
  then obtain k_p where k_p:
   $\bigwedge j. k \leq j \implies \exists i. (C, i) \in\# wN\_of\_wstate (\text{lnth } \text{Sts } j)$ 
  unfolding Liminf_llist_def
  by auto
  have chain_drop_Sts:  $\text{chain } (\sim_w) (\text{ldrop } k \text{ Sts})$ 
  using deriv_inf_inff by (simp add: inf_chain_ldroprn_chain_ldrop_enat)
  have in_N_j:  $\bigwedge j. \exists i. (C, i) \in\# wN\_of\_wstate (\text{lnth } (\text{ldrop } k \text{ Sts}) j)$ 
  using k_p by (simp add: add commute inf)
  then have  $\text{chain } (\lambda x y. (x, y) \in \text{RP\_filtered\_relation})^{-1-1} (\text{lmap } (\text{RP\_filtered\_measure } (\lambda Ci. \text{True}))) (\text{ldrop } k \text{ Sts})$ 
  using inff_inf_weighted_RP_measure_decreasing_N_chain_drop_Sts
   $\text{infinite\_chain\_relation\_measure}$ [of  $\lambda \text{St}. \exists i. (C, i) \in\# wN\_of\_wstate \text{St } (\sim_w)$ ] by blast
  then show False
  using wfP_iff_no_infinite_down_chain_llist[of  $\lambda x y. (x, y) \in \text{RP\_filtered\_relation}$ ]
   $\text{wf\_RP\_filtered\_relation inff}$ 
  by (metis (no_types, lifting) inf_llist_lnth_ldrop_enat_inf_llist_lfinite_inf_llist
   $\text{lfinite\_lmap wfPUNIVI wf\_induct\_rule}$ )
next
  assume asm:  $C \in \text{Liminf\_llist } (\text{lmap } P\_of\_state (\text{lmap } \text{state\_of\_wstate } \text{Sts}))$ 
  from asm obtain i where i_p:
   $\text{enat } i < \text{llength } \text{Sts}$ 
   $\bigwedge j. i \leq j \wedge \text{enat } j < \text{llength } \text{Sts} \implies C \in P\_of\_state (\text{state\_of\_wstate } (\text{lnth } \text{Sts } j))$ 
  unfolding Liminf_llist_def by auto
  then obtain i where  $(C, i) \in \text{Liminf\_llist } (\text{lmap } (\text{set\_mset } \circ wP\_of\_wstate) \text{Sts})$ 
  using persistent_wclause_in_P_if_persistent_clause_in_P[of C] using asm inf by auto
  then have  $\exists l. \forall k \geq l. (C, i) \in (\text{set\_mset } \circ wP\_of\_wstate) (\text{lnth } \text{Sts } k)$ 
  unfolding Liminf_llist_def using inff_inf by auto
  then obtain k where k_p:

```

```

  (∀ 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_ldroprn_chain inf inff by (simp add: inf_chain_ldroprn_chain ldrops_enat)
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 (verit) inf_llist_lnth ldrops_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 weighted_RP_fair empty_Q0_RP, 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 weighted_RP_fair empty_Q0_RP, simplified lhd_lmap_Sts] .

end

end

locale weighted_FO_resolution_prover_with_size_timestamp_factors =
  FO_resolution_prover S subst_atm id_subst comp_subst renamings_apart atm_of_atms mgu less_atm
  for
    S :: ('a :: 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  $\langle \rightsquigarrow_w \rangle$  50)

end

end

### 3 A Deterministic Ordered Resolution Prover for First-Order Clauses

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

theory *Deterministic\_FO\_Ordered\_Resolution\_Prover*

imports

*Polynomial\_Factorization.Missing\_List*

*Weighted\_FO\_Ordered\_Resolution\_Prover*

*Lambda\_Free\_RPOs.Lambda\_Free\_Util*

begin

#### 3.1 Library

lemma *apfst\_fst\_snd*:  $apfst\ f\ x = (f\ (fst\ x),\ snd\ x)$

by (rule *apfst\_conv*[of *\_fst* *x* *snd* *x* for *x*, *unfolded* *prod.collapse*])

lemma *apfst\_comp\_rpair\_const*:  $apfst\ f \circ (\lambda x. (x, y)) = (\lambda x. (x, y)) \circ f$

by (*simp* *add: comp\_def*)

lemma *length\_remove1\_less*[*termination\_simp*]:  $x \in set\ xs \implies length\ (remove1\ x\ xs) < length\ xs$

by (*induct* *xs*) *auto*

lemma *map\_filter\_neq\_eq\_filter\_map*:

$map\ f\ (filter\ (\lambda y. f\ x \neq f\ y)\ xs) = filter\ (\lambda z. f\ x \neq z)\ (map\ f\ xs)$

by (*induct* *xs*) *auto*

lemma *mset\_map\_remdups\_gen*:

$mset\ (map\ f\ (remdups\_gen\ f\ xs)) = mset\ (remdups\_gen\ (\lambda x. x)\ (map\ f\ xs))$

by (*induct* *f* *xs* *rule: remdups\_gen.induct*) (*auto simp: map\_filter\_neq\_eq\_filter\_map*)

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

proof –

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

proof (*induct* *f* *xs* *rule: remdups\_gen.induct*)

case (*?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 *funpow\_fixpoint*:  $f\ x = x \implies (f \hat{\sim} n)\ x = x$

by (*induct* *n*) *auto*

lemma *rtranclp\_imp\_eq\_image*:  $(\forall x\ y. R\ x\ y \longrightarrow f\ x = f\ y) \implies R^{**}\ x\ y \implies f\ x = f\ y$

by (*erule* *rtranclp.induct*) *auto*

lemma *tranclp\_imp\_eq\_image*:  $(\forall x\ y. R\ x\ y \longrightarrow f\ x = f\ y) \implies R^{++}\ x\ y \implies f\ x = f\ y$



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** class\_of\_dstate :: 'a dstate ⇒ 'a clause set **where**

class\_of\_dstate St ≡ class\_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 <math>\rightsquigarrow\_w^\*</math> 50) **where**

( $\rightsquigarrow_w^*$ ) ≡ ( $\rightsquigarrow_w$ )\*\*

**abbreviation** trancl\_weighted\_RP (infix <math>\rightsquigarrow\_w^+</math> 50) **where**

( $\rightsquigarrow_w^+$ ) ≡ ( $\rightsquigarrow_w$ )\*\*

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

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

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

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

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

**fun** *reduce\_all2* :: 'a lclause  $\Rightarrow$  'a dclause list  $\Rightarrow$  'a dclause list  $\times$  'a dclause list **where**  
*reduce\_all2* \_ [] = ([], [])  
| *reduce\_all2* D (Ci # Cs) =  
(let  
(C, i) = Ci;  
C' = *reduce* [D] [] C  
in  
(if C' = C then *apsnd* else *apfst*) (Cons (C', i)) (*reduce\_all2* D Cs))

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

**lemma** *remove\_all\_mset\_minus*:  $\text{mset } ys \subseteq \# \text{mset } xs \implies \text{mset } (\text{remove\_all } xs \ ys) = \text{mset } xs - \text{mset } ys$

**proof** (*induction* ys *arbitrary*: xs)

**case** (Cons y ys)

**show** ?case

**proof** (*cases* y  $\in$  set xs)

**case** y\_in: True

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

**using** Cons(2) **by** (*simp* add: *insert\_subset\_eq\_iff*)

**show** ?thesis

**using** y\_in Cons subs **by** auto

**next**

**case** False

**then show** ?thesis

**using** Cons **by** auto

**qed**

**qed** auto

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

**definition** *resolvable* :: 'a  $\Rightarrow$  'a lclause  $\Rightarrow$  'a lclause  $\Rightarrow$  'a lclause  $\Rightarrow$  bool **where**  
*resolvable* A D CA Ls  $\longleftrightarrow$   
(let  $\sigma =$  (*mgu* {*insert* A (*atms\_of* (*mset* Ls))}) in  
 $\sigma \neq \text{None}$   
 $\wedge$  Ls  $\neq$  []  
 $\wedge$  *maximal\_wrt* (A  $\cdot$  a *the*  $\sigma$ ) ((*add\_mset* (Neg A) (*mset* D))  $\cdot$  *the*  $\sigma$ )  
 $\wedge$  *strictly\_maximal\_wrt* (A  $\cdot$  a *the*  $\sigma$ ) ((*mset* CA - *mset* Ls)  $\cdot$  *the*  $\sigma$ )  
 $\wedge$  ( $\forall L \in \text{set } Ls. \text{is\_pos } L$ ))

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

**definition** *resolve* :: 'a lclause  $\Rightarrow$  'a lclause  $\Rightarrow$  'a lclause list **where**

```

resolve C D =
  concat (map ( $\lambda L$ .
    (case L of
      Pos A  $\Rightarrow$  []
    | Neg A  $\Rightarrow$ 
      if maximal_wrt A (mset D) then
        resolve_on A (remove1 L D) C
      else
        [])) D)

```

**definition** *resolve\_rename* :: 'a lclause  $\Rightarrow$  'a lclause  $\Rightarrow$  'a lclause list **where**

```

resolve_rename C D =
  (let  $\sigma s = \text{renamings\_apart } [mset D, mset C]$  in
    resolve (map ( $\lambda L$ . L  $\cdot$ l last  $\sigma s$ ) C) (map ( $\lambda L$ . L  $\cdot$ l hd  $\sigma s$ ) D))

```

**definition** *resolve\_rename\_either\_way* :: 'a lclause  $\Rightarrow$  'a lclause  $\Rightarrow$  'a lclause list **where**

```

resolve_rename_either_way C D = resolve_rename C D @ resolve_rename D C

```

**fun** *select\_min\_weight\_clause* :: 'a dclause  $\Rightarrow$  'a dclause list  $\Rightarrow$  'a dclause **where**

```

select_min_weight_clause Ci [] = Ci
| select_min_weight_clause Ci (Dj # Djs) =
  select_min_weight_clause
    (if weight (apfst mset Dj) < weight (apfst mset Ci) then Dj else Ci) Djs

```

**lemma** *select\_min\_weight\_clause\_in*: *select\_min\_weight\_clause* P0 P  $\in$  set (P0 # P)

**by** (induct P arbitrary: P0) auto

**function** *remdups\_clss* :: 'a dclause list  $\Rightarrow$  'a dclause list **where**

```

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

```

**declare** *remdups\_clss.simps(2)* [simp del]

**fun** *deterministic\_RP\_step* :: 'a dstate  $\Rightarrow$  'a dstate **where**

```

deterministic_RP_step (N, P, Q, n) =
  (if  $\exists Ci \in \text{set } (P @ Q)$ . fst Ci = [] then
    ([], [], remdups_clss P @ Q, n + length (remdups_clss P))
  else
    (case N of
      []  $\Rightarrow$ 
      (case P of
        []  $\Rightarrow$  (N, P, Q, n)
      | P0 # P'  $\Rightarrow$ 
        let
          (C, i) = select_min_weight_clause P0 P';
          N = map ( $\lambda D$ . (D, n)) (remdups_gen mset (resolve_rename C C
            @ concat (map (resolve_rename_either_way C  $\circ$  fst) Q)));
          P = filter ( $\lambda(D, j)$ . mset D  $\neq$  mset C) P;
          Q = (C, i) # Q;
          n = Suc n
        in
          (N, P, Q, n))

```

```

| (C, i) # N =>
  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 ~<sub>w</sub> St'  
**using** *wrp.final\_weighted\_RP*  
**by** (cases St) (auto intro: *wrp.final\_weighted\_RP* simp: *is\_final\_dstate.simps*)

**lemma** *is\_final\_dstate\_funpow\_imp\_deterministic\_RP\_neq\_None*:  
*is\_final\_dstate* ((*deterministic\_RP\_step* ^ k) St) ==> *deterministic\_RP* St ≠ None  
**proof** (induct k arbitrary: St)  
**case** (Suc k)  
**note** *ih* = *this*(1) **and** *final\_Sk* = *this*(2)[*simplified, unfolded funpow\_swap1*]  
**show** ?case  
**using** *ih*[OF *final\_Sk*] **by** (subst *deterministic\_RP.simps*) (simp add: *prod.case\_eq\_if*)  
**qed** (subst *deterministic\_RP.simps*, simp add: *prod.case\_eq\_if*)

**lemma** *is\_reducible\_lit\_mono\_cls*:  
*mset* C ⊆# *mset* C' ==> *is\_reducible\_lit* Ds C L ==> *is\_reducible\_lit* Ds C' L  
**unfolding** *is\_reducible\_lit\_def* **by** (blast intro: *subset\_mset.trans*)

**lemma** *is\_reducible\_lit\_mset\_iff*:  
*mset* C = *mset* C' ==> *is\_reducible\_lit* Ds C' L ↔ *is\_reducible\_lit* Ds C L  
**by** (metis *is\_reducible\_lit\_mono\_cls subset\_mset.order\_refl*)

**lemma** *is\_reducible\_lit\_remove1\_Cons\_iff*:  
**assumes** L ∈ set C'  
**shows** *is\_reducible\_lit* Ds (C @ remove1 L (M # C')) L ↔  
*is\_reducible\_lit* Ds (M # C @ remove1 L C') L  
**using** *assms* **by** (subst *is\_reducible\_lit\_mset\_iff*, auto)

**lemma** *reduce\_mset\_eq*: *mset* C = *mset* C' ==> *reduce* Ds C E = *reduce* Ds C' E  
**proof** (induct E arbitrary: C C')  
**case** (Cons L E)  
**note** *ih* = *this*(1) **and** *mset\_eq* = *this*(2)  
**have**  
*mset\_lc\_eq*: *mset* (L # C) = *mset* (L # C') **and**  
*mset\_ce\_eq*: *mset* (C @ E) = *mset* (C' @ E)  
**using** *mset\_eq* **by** simp+

**show** ?case  
**using** ih[OF mset\_eq] ih[OF mset\_lc\_eq] **by** (simp add: is\_reducible\_lit\_mset\_iff[OF mset\_ce\_eq])  
**qed** simp

**lemma** reduce\_rotate[simp]: 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)  $\subseteq$ # 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:  
 $L \in \text{set } C' \implies \text{is\_reducible\_lit } Ds (C @ \text{remove1 } L C') L \implies \text{reduce } Ds C C' \neq C'$   
**proof** (induct C' arbitrary: C)  
**case** (Cons M C')  
**note** ih = this(1) **and** l\_in = this(2) **and** l\_red = this(3)

**show** ?case  
**proof** (cases is\_reducible\_lit Ds (C @ C') M)  
**case** True  
**then show** ?thesis  
**by** simp (metis mset.simps(2) mset\_reduce\_subset multi\_self\_add\_other\_not\_self  
subset\_mset.eq\_iff subset\_mset\_imp\_subset\_add\_mset)  
**next**  
**case** m\_irred: False  
**have**  
 $L \in \text{set } C'$  **and**  
 $\text{is\_reducible\_lit } Ds (M \# C @ \text{remove1 } L C') L$   
**using** l\_in l\_red m\_irred is\_reducible\_lit\_remove1\_Cons\_iff **by** auto  
**then show** ?thesis  
**by** (simp add: ih[of M # C] m\_irred)  
**qed**  
**qed** simp

**lemma** is\_reducible\_imp\_is\_reducible\_lit:  
 $\text{reduce } Ds C C' \neq C' \implies \exists L \in \text{set } C'. \text{is\_reducible\_lit } Ds (C @ \text{remove1 } L C') L$   
**proof** (induct C' arbitrary: C)  
**case** (Cons M C')  
**note** ih = this(1) **and** mc'\_red = this(2)

**show** ?case  
**proof** (cases is\_reducible\_lit Ds (C @ C') M)  
**case** m\_irred: False  
**show** ?thesis  
**using** ih[of M # C] mc'\_red[simplified, simplified m\_irred, simplified] m\_irred  
is\_reducible\_lit\_remove1\_Cons\_iff  
**by** auto  
**qed** simp  
**qed** simp

**lemma** is\_irreducible\_iff\_nexists\_is\_reducible\_lit:  
 $\text{reduce } Ds C C' = C' \iff \neg (\exists L \in \text{set } C'. \text{is\_reducible\_lit } Ds (C @ \text{remove1 } L C') L)$   
**using** is\_reducible\_imp\_is\_reducible\_lit is\_reducible\_lit\_imp\_is\_reducible **by** blast

**lemma** is\_irreducible\_mset\_iff: mset E = mset E'  $\implies \text{reduce } Ds C E = E \iff \text{reduce } Ds C E' = E'$   
**unfolding** is\_irreducible\_iff\_nexists\_is\_reducible\_lit  
**by** (metis (full\_types) is\_reducible\_lit\_mset\_iff mset\_remove1\_set\_mset\_mset\_union\_code)

```

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

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

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

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

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

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

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

```

```

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

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

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

  have wstate_of_dstate (N, P @ P', Q, n)
    ~_w wstate_of_dstate (N, P @ filter (λ(E, l). mset E ≠ mset (fst ?Dj)) ?P'', Q, n)
    by (rule arg_cong2[THEN iffD1, of _ _ _ _ (~_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 _ _ λx. image_mset (apfst mset) x])
  apply (metis (no_types, opaque_lifting) strictly_subsume_def)
  apply (subst (3) p')
  apply (subst list.size)
  apply (metis (no_types, lifting) less_Suc0 less_add_same_cancel1 linorder_neqE_nat
    not_add_less1 sum_length_filter_compl trans_less_add1)
  using p_nsubs by fast
  ultimately show ?thesis
    by (rule converse_rtranclp_into_rtranclp)
next
  case nsubs: False
  show ?thesis
    apply (rule arg_cong2[THEN iffD1, of _ _ _ _ (~_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 ( $N, P, Q @ Dj \# Q', n$ )  $\rightsquigarrow_w^*$ 
    wstate_of_dstate ( $N, P, Q @ \text{filter } (Not \circ \text{strictly\_subsume } [C] \circ \text{fst}) [Dj] @ Q', n$ )
  proof (cases  $\text{strictly\_subsume } [C] (\text{fst } Dj)$ )
    case subs: True
    have wstate_of_dstate ( $N, P, Q @ Dj \# Q', n$ )  $\rightsquigarrow_w$  wstate_of_dstate ( $N, P, Q @ Q', n$ )
      by (rule arg_cong2[THEN iffD1, of _ _ _ _ ( $\rightsquigarrow_w$ ), 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  $Q @ \text{filter } (Not \circ \text{strictly\_subsume } [C] \circ \text{fst}) [Dj]$ ] by force
  qed simp

```

**lemma** reduce\_clause\_in\_P:

```

assumes
  c_in:  $C \in \text{fst } ' \text{set } N$  and
  p_irred:  $\forall (E, k) \in \text{set } (P @ P'). k > j \longrightarrow \text{is\_irreducible } [C] E$ 
shows wstate_of_dstate ( $N, P @ (D @ D', j) \# P', Q, n$ )
   $\rightsquigarrow_w^*$  wstate_of_dstate ( $N, P @ (D @ \text{reduce } [C] D D', j) \# P', Q, n$ )
proof (induct  $D'$  arbitrary:  $D$ )
  case ih: (Cons  $L D'$ )
  show ?case
  proof (cases  $\text{is\_reducible\_lit } [C] (D @ D') L$ )
    case l_red: True
    then obtain  $L' :: 'a \text{ literal}$  and  $\sigma :: 's$  where
      l'_in:  $L' \in \text{set } C$  and
      not_l:  $- L = L' \cdot l \sigma$  and
      subs:  $\text{mset } (\text{remove1 } L' C) \cdot \sigma \subseteq \# \text{mset } (D @ D')$ 
      unfolding  $\text{is\_reducible\_lit\_def}$  by force

    have ldd'_red:  $\text{is\_reducible } [C] (L \# D @ D')$ 
      apply (rule  $\text{is\_reducible\_lit\_imp\_is\_reducible}$ )
      using l_red by auto

    have lt_imp_neq:  $\forall (E, k) \in \text{set } (P @ P'). j < k \longrightarrow \text{mset } E \neq \text{mset } (L \# D @ D')$ 
      using p_irred ldd'_red  $\text{is\_irreducible\_mset\_iff}$  by fast

    have wstate_of_dstate ( $N, P @ (D @ L \# D', j) \# P', Q, n$ )
       $\rightsquigarrow_w$  wstate_of_dstate ( $N, P @ (D @ D', j) \# P', Q, n$ )
      apply (rule arg_cong2[THEN iffD1, of _ _ _ _ ( $\rightsquigarrow_w$ ), OF _ _
        wrp.backward_reduction_P[of mset  $C - \{\#L\# \}$   $L'$  mset (map (apfst mset)  $N$ )  $L \sigma$ 
          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:  $C \in \text{fst } ' \text{set } N$  and
  p_irred:  $\forall (E, k) \in \text{set } P. k > j \longrightarrow \text{is\_irreducible } [C] E$  and
  d'_red:  $\text{reduce } [C] D D' \neq D'$ 
shows wstate_of_dstate ( $N, P, Q @ (D @ D', j) \# Q', n$ )

```



```

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

  have wstate_of_dstate (N, P, Q @ (D @ L # D', j) # Q', n)
   $\rightsquigarrow_w$  wstate_of_dstate (N, (D @ D', j) # P, Q @ Q', n)
  by (rule arg_cong2[THEN iffD1, of _ _ _ _ ( $\rightsquigarrow_w$ ), OF _ _
    wrp.backward_reduction_Q[of mset C - {#L'#} L' mset (map (apfst mset) N) L  $\sigma$ 
    mset (D @ D') mset (map (apfst mset) P) mset (map (apfst mset) (Q @ Q')) j n]],
    use l'_in not_l subs c_in in auto)
  then show ?thesis
  using l_red p_irred reduce_clause_in_P[OF c_in, of [] P j D D' Q @ Q' n] by simp
next
case l_nred: False
then have d'_red: reduce [C] (D @ [L]) D'  $\neq$  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  $\in$  fst ' set N and
  p_irred:  $\forall (E, k) \in$  set P. is_irreducible [C] E
shows wstate_of_dstate (N, P @ P', Q, n)  $\rightsquigarrow_w^*$  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'  $\neq$  []
  using suc_l by auto

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

  obtain Dj :: 'a dclause where
    dj_in: Dj  $\in$  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  $\forall k \in$  snd ' set P'. k  $\leq$  j
  unfolding j_def using p'_nnil by simp
  then have j_max:  $\forall (E, k) \in$  set P'. j  $\geq$  k
  unfolding image_def by fastforce

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

```

```

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

```

**lemma** reduce\_clauses\_in\_Q:

```

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

```

**lemma** eligible\_iff:

```

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

```

**lemma** ord\_resolve\_one\_side\_prem:

```

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

```

**lemma** ord\_resolve\_rename\_one\_side\_prem:

```

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

```

**abbreviation** Bin\_ord\_resolve :: 'a clause ⇒ 'a clause ⇒ 'a clause set **where**

```

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

```

**abbreviation** Bin\_ord\_resolve\_rename :: 'a clause ⇒ 'a clause ⇒ 'a clause set **where**

```

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

```

**lemma** resolve\_on\_eq\_UNION\_Bin\_ord\_resolve:

```

mset ' set (resolve_on A D CA) =

```

```

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

```

```

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

```

```

    then show ?thesis
      by auto
  qed
qed
then obtain Al where Al_p: mset Al = AA subseq (map Pos Al) CA
  by auto

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

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

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

```

```

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

```

**lemma** poss\_in\_map\_clauseD:

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

**proof** (induct AA arbitrary: C)

**case** (add A AA)

**note** ih = this(1) **and** aaa\_sub = this(2)

**have** Pos A  $\in\# \text{map\_clause } f\ C$

**using** aaa\_sub **by** auto

**then obtain** A0 **where**

pa0\_in: Pos A0  $\in\# C$  **and**

a: A = f A0

**by** clarify (metis literal.distinct(1) literal.exhaust literal.inject(1) literal.simps(9,10))

**have** poss AA  $\subseteq\# \text{map\_clause } f\ (C - \{\#Pos\ A0\# \})$

**using** pa0\_in aaa\_sub[unfolded a] **by** (simp add: image\_mset\_remove1\_mset\_if insert\_subset\_eq\_iff)

**then obtain** AA0 **where**

paa0\_sub: poss AA0  $\subseteq\# C - \{\#Pos\ A0\# \}$  **and**

aa: AA = image\_mset f AA0

**using** ih **by** meson

**have** poss (add\_mset A0 AA0)  $\subseteq\# C$

**using** pa0\_in paa0\_sub **by** (simp add: insert\_subset\_eq\_iff)

**moreover have** add\_mset A AA = image\_mset f (add\_mset A0 AA0)

**unfolding** a aa **by** simp

**ultimately show** ?case

**by** blast

**qed** simp

**lemma** poss\_subset\_filterD:

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

**unfolding** subst\_atm\_mset\_def subst\_lit\_def **by** (rule poss\_in\_map\_clauseD)

**lemma** neg\_in\_map\_literalD: Neg A  $\in\# \text{map\_literal } f\ 'D \implies \exists A0. Neg\ A0 \in\# D \wedge A = f\ A0$

**unfolding** image\_def **by** (clarify, case\_tac x, auto)

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

**unfolding** subst\_lit\_def image\_def **by** (rule neg\_in\_map\_literalD) simp

**lemma** resolve\_rename\_eq\_Bin\_ord\_resolve\_rename:

$\text{mset } ' \text{set } (\text{resolve\_rename } C\ D) = \text{Bin\_ord\_resolve\_rename } (\text{mset } C)\ (\text{mset } D)$

**proof** (intro order\_antisym subsetI)

**let** ?qs = renamings\_apart [mset D, mset C]

**define**  $\varrho' :: 's$  **where**

$\varrho' = \text{hd } ?qs$

```

define  $\rho :: 's$  where
   $\rho = last \ ?\rho s$ 

have  $tl\_qs: tl \ ?\rho s = [\rho]$ 
  unfolding  $\rho\_def$ 
  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  $e\_in: E \in mset \ 'set \ (resolve\_rename \ C \ D)$ 

  from  $e\_in$  obtain  $AA :: 'a \ multiset$  and  $A :: 'a$  and  $\sigma :: 's$  where
     $aa\_sub: poss \ AA \subseteq\# \ mset \ C \cdot \rho$  and
     $a\_in: Neg \ A \in\# \ mset \ D \cdot \rho'$  and
     $res\_e: ord\_resolve \ S \ [mset \ C \cdot \rho] \ \{\#L \cdot l \ \rho'. \ L \in\# \ mset \ D\# \} \ [AA] \ [A] \ \sigma \ E$ 
    unfolding  $\rho'\_def \ \rho\_def$ 
    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
     $aa0\_sub: poss \ AA0 \subseteq\# \ mset \ C$  and
     $aa: AA = AA0 \cdot am \ \rho$ 
    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
     $a0\_in: Neg \ A0 \in \ set \ D$  and
     $a: A = A0 \cdot a \ \rho'$ 
    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  $E \in Bin\_ord\_resolve\_rename \ (mset \ C) \ (mset \ D)$ 
    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 ( $rule\_tac \ x = \sigma \ in \ exI$ )
    unfolding  $aa \ a \ \rho'\_def[symmetric] \ \rho\_def[symmetric] \ tl\_qs$  by ( $simp \ add: \ subst\_cls\_def$ )
}
{
  fix  $E$ 
  assume  $e\_in: E \in Bin\_ord\_resolve\_rename \ (mset \ C) \ (mset \ D)$ 
  show  $E \in mset \ 'set \ (resolve\_rename \ C \ D)$ 
    using  $e\_in$ 
    unfolding  $resolve\_rename\_def \ Let\_def \ resolve\_eq \ Bin\_ord\_resolve \ ord\_resolve\_rename.simps$ 
    apply  $clarsimp$ 
    apply ( $rule\_tac \ x = AA \cdot am \ \rho \ in \ exI$ )

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma concls_of_inferences_between_eq_Bin_ord_resolve_rename:
  concls_of (inference_system.inferences_between (ord_FO_Γ S) Q C) =
  Bin_ord_resolve_rename C C ∪ (∪ D ∈ Q. Bin_ord_resolve_rename C D ∪ Bin_ord_resolve_rename D C)
by (subst inferences_between_eq_UNION)
  (auto simp: image_Un image_UN concls_of_inferences_between_singleton_eq_Bin_ord_resolve_rename)

```

```

lemma resolve_rename_either_way_eq_concls_of_inferences_between:
  mset ' set (resolve_rename C C) ∪ (∪ D ∈ Q. mset ' set (resolve_rename_either_way C D)) =
  concls_of (inference_system.inferences_between (ord_FO_Γ S) (mset ' Q) (mset C))
by (simp add: resolve_rename_either_way_def image_Un resolve_rename_eq_Bin_ord_resolve_rename
  concls_of_inferences_between_eq_Bin_ord_resolve_rename UN_Un_distrib)

```



**lemma** *compute\_inferences*:

**assumes**

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

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

**shows**

*wstate\_of\_dstate* ( $[], P, Q, n$ )  $\rightsquigarrow_w$

*wstate\_of\_dstate* ( $\text{map } (\lambda D. (D, n)) (\text{remdups\_gen mset } (\text{resolve\_rename } C \ C \ @$

$\text{concat } (\text{map } (\text{resolve\_rename\_either\_way } C \circ \text{fst}) Q)))$ ,

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

(**is**  $\_ \rightsquigarrow_w \text{ wstate\_of\_dstate } (?N, \_)$ )

**proof** –

**have** *ms\_ci\_in*:  $(\text{mset } C, i) \in \# \text{ image\_mset } (\text{apfst mset}) (\text{mset } P)$

**using** *ci\_in* **by** *force*

**show** *?thesis*

**apply** (*rule arg\_cong2*[*THEN iffD1*, *of*  $\_ \_ \_ \_ (\rightsquigarrow_w)$ , *OF*  $\_ \_ \_$

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

$\text{mset } (\text{map } (\text{apfst mset}) ?N) \ n \ \text{mset } (\text{map } (\text{apfst mset}) Q)$ ]])

**apply** (*simp add*: *add\_mset\_remove\_trivial\_eq*[*THEN iffD2*, *OF ms\_ci\_in*, *symmetric*])

**using** *ms\_ci\_in*

**apply** (*simp add*: *ci\_in image\_mset\_remove1\_mset\_if*)

**apply** (*smt apfst\_conv case\_prodE case\_prodI2 case\_prod\_conv filter\_mset\_cong*

*image\_mset\_filter\_swap mset\_filter*)

**apply** (*metis ci\_min in\_diffD*)

**apply** (*simp only*: *list.map\_comp apfst\_comp\_rpair\_const*)

**apply** (*simp only*: *list.map\_comp*[*symmetric*])

**apply** (*subst mset\_map*)

**apply** (*unfold mset\_map\_remdups\_gen mset\_remdups\_gen\_ident*)

**apply** (*subst image\_mset\_mset\_set*)

**apply** (*simp add*: *inj\_on\_def*)

**apply** (*subst mset\_set\_eq\_iff*)

**apply** *simp*

**apply** (*simp add*: *finite\_ord\_FO\_resolution\_inferences\_between*)

**apply** (*rule arg\_cong*[*of*  $\_ \_ \lambda N. (\lambda D. (D, n)) ' N$ ])

**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*:  $\neg \text{ is\_final\_dstate } St$  **and**

*step*:  $St' = \text{deterministic\_RP\_step } St$

**shows** *wstate\_of\_dstate*  $St \rightsquigarrow_w^+ \text{ wstate\_of\_dstate } St'$

**proof** –

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

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

**by** (*cases St*) *blast*

**note** *step* = *step*[*unfolded st deterministic\_RP\_step.simps*, *simplified*]

**show** *?thesis*

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

**case** *nil\_in*: *True*

**note** *step* = *step*[*simplified nil\_in*, *simplified*]

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

**using** *nil\_in* **by** (*force simp*: *image\_def*)

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

*wstate\_of\_dstate*  $(N, P, Q, n)$

$\rightsquigarrow_w^* \text{ wstate\_of\_dstate } ([], [], \text{remdups\_class } P \ @ \ Q, n + \text{length } (\text{remdups\_class } P))$

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

**case** *0*

```

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

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

have wstate_of_dstate ((E @ C, i) # N', P, Q, n)
  ~*_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 _ _ _ _ (λ_. ~*_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  $\sigma$  in  $\langle \text{auto simp: comp\_def} \rangle$ )
  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)  $\rightsquigarrow_w^*$  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')
   $\rightsquigarrow_w$  wstate_of_dstate (N', (C', i) # P', Q', n') for P' Q' n'
  by (rule arg_cong2[THEN iffD1, of _ _ _ _ ( $\rightsquigarrow_w$ ), 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  $\circ$  strictly_subsume [[]]  $\circ$  fst) P = [] and
    filter_q: filter (Not  $\circ$  strictly_subsume [[]]  $\circ$  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)
     $\rightsquigarrow_w^*$  wstate_of_dstate (([], i) # N', [], Q, n)
    by (rule arg_cong2[THEN iffD1, of _ _ _ _ ( $\rightsquigarrow_w^*$ ), OF _ _
      remove_strictly_subsumed_clauses_in_P[of [] _ [], unfolded append_Nil],
      OF refl])
    (auto simp: filter_p)
  also have ...  $\rightsquigarrow_w^*$  wstate_of_dstate (([], i) # N', [], [], n)
    by (rule arg_cong2[THEN iffD1, of _ _ _ _ ( $\rightsquigarrow_w^*$ ), 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 ( $\rightsquigarrow_w$ ))]
  also have wstate_of_dstate (N', [([], i)], [], n)
     $\rightsquigarrow_w^*$  wstate_of_dstate ([[], [([], i)], [], n)
    by (rule empty_N_if_Nil_in_P_or_Q) simp
  also have ...  $\rightsquigarrow_w$  wstate_of_dstate ([[], [], [([], i)], Suc n)
    by (rule arg_cong2[THEN iffD1, of _ _ _ _ ( $\rightsquigarrow_w$ ), 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'  $\vee$  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)  $\rightsquigarrow_w$  wstate_of_dstate (N', P, Q, n)

```

```

proof (cases is_tautology C')
case True
then obtain A :: 'a where
  neg_a: Neg A ∈ 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*:  $is\_final\_dstate\ St \implies deterministic\_RP\_step\ St = St$   
**by** (*cases St*) (*auto simp: deterministic\_RP\_step.simps is\_final\_dstate.simps*)

**lemma** *deterministic\_RP\_SomeD*:

**assumes** *deterministic\_RP*  $(N, P, Q, n) = Some\ R$   
**shows**  $\exists N' P' Q' n'. (\exists k. (deterministic\_RP\_step \rightsquigarrow k)\ (N, P, Q, n) = (N', P', Q', n'))$   
 $\wedge is\_final\_dstate\ (N', P', Q', n') \wedge R = map\ fst\ Q'$

**proof** (*induct rule: deterministic\_RP.raw\_induct[OF \_ assms]*)

**case** (*1 self\_call St R*)

**note** *ih = this(1) and step = this(2)*

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

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

**by** (*cases St*) *blast*

**note** *step = step[unfolded st, simplified]*

**show** *?case*

**proof** (*cases is\_final\_dstate (N, P, Q, n)*)

**case** *True*

**then have**  $(deterministic\_RP\_step \rightsquigarrow 0)\ (N, P, Q, n) = (N, P, Q, n)$

$\wedge is\_final\_dstate\ (N, P, Q, n) \wedge R = map\ fst\ Q$

**using** *step* **by** *simp*

**then show** *?thesis*

**unfolding** *st* **by** *blast*

**next**

**case** *nonfinal: False*

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

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

$(deterministic\_RP\_step \rightsquigarrow k)\ (deterministic\_RP\_step\ (N, P, Q, n)) = (N', P', Q', n')$  **and**

$is\_final\_dstate\ (N', P', Q', n')$

$R = map\ fst\ Q'$

**using** *ih[OF step]* **by** *blast*

**then show** *?thesis*

**unfolding** *st* *funpow\_Suc\_right[symmetric, THEN fun\_cong, unfolded comp\_apply]* **by** *blast*

**qed**

**qed**

**context**

**fixes**

$N0 :: 'a\ dclause\ list$  **and**

$n0 :: nat$  **and**

$R :: 'a\ lclause\ list$

**begin**

**abbreviation** *St0*  $:: 'a\ dstate$  **where**

$St0 \equiv (N0, [], [], n0)$

**abbreviation** *grounded\_N0* **where**

$grounded\_N0 \equiv grounding\_of\_cls\ (set\ (map\ (mset\ o\ fst)\ N0))$

**abbreviation** *grounded\_R*  $:: 'a\ clause\ set$  **where**

$grounded\_R \equiv grounding\_of\_cls\ (set\ (map\ mset\ R))$

**primcorec** *derivation\_from*  $:: 'a\ dstate \Rightarrow 'a\ dstate\ llist$  **where**

*derivation\_from St =*

$LCons\ St$  (if  $is\_final\_dstate\ St$  then  $LNil$  else  $derivation\_from\ (deterministic\_RP\_step\ St)$ )

**abbreviation**  $Sts :: 'a\ dstate\ llist\ where$   
 $Sts \equiv derivation\_from\ St0$

**abbreviation**  $wSts :: 'a\ wstate\ llist\ where$   
 $wSts \equiv lmap\ wstate\_of\_dstate\ Sts$

**lemma**  $full\_deriv\_wSts\_trancl\_weighted\_RP: full\_chain\ (\rightsquigarrow_w^+)\ wSts$

**proof** –

**have**  $Sts' = derivation\_from\ St0' \implies full\_chain\ (\rightsquigarrow_w^+)\ (lmap\ wstate\_of\_dstate\ Sts')$   
**for**  $St0'\ Sts'$

**proof** (coinduction arbitrary:  $St0'\ Sts'$  rule:  $full\_chain.coinduct$ )

**case**  $sts': full\_chain$

**show**  $?case$

**proof** (cases  $is\_final\_dstate\ St0'$ )

**case**  $True$

**then have**  $ltl\ (lmap\ wstate\_of\_dstate\ Sts') = LNil$

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

**then have**  $lmap\ wstate\_of\_dstate\ Sts' = LCons\ (wstate\_of\_dstate\ St0')\ LNil$

**unfolding**  $sts'$  **by** ( $subst\ derivation\_from.code, subst\ (asm)\ derivation\_from.code, auto$ )

**moreover have**  $\bigwedge St''. \neg wstate\_of\_dstate\ St0' \rightsquigarrow_w\ St''$

**using**  $True$  **by** ( $rule\ is\_final\_dstate\_imp\_not\_weighted\_RP$ )

**ultimately show**  $?thesis$

**by** ( $meson\ tranclpD$ )

**next**

**case**  $nfinal: False$

**have**  $lmap\ wstate\_of\_dstate\ Sts' =$

$LCons\ (wstate\_of\_dstate\ St0')\ (lmap\ wstate\_of\_dstate\ (ltl\ Sts'))$

**unfolding**  $sts'$  **by** ( $subst\ derivation\_from.code$ )  $simp$

**moreover have**  $ltl\ Sts' = derivation\_from\ (deterministic\_RP\_step\ St0')$

**unfolding**  $sts'$  **using**  $nfinal$  **by** ( $subst\ derivation\_from.code$ )  $simp$

**moreover have**  $wstate\_of\_dstate\ St0' \rightsquigarrow_w^+\ wstate\_of\_dstate\ (lhd\ (ltl\ Sts'))$

**unfolding**  $sts'$  **using**  $nonfinal\_deterministic\_RP\_step[OF\ nfinal\ refl]$   $nfinal$

**by** ( $subst\ derivation\_from.code$ )  $simp$

**ultimately show**  $?thesis$

**by**  $fastforce$

**qed**

**qed**

**then show**  $?thesis$

**by**  $blast$

**qed**

**lemmas**  $deriv\_wSts\_trancl\_weighted\_RP = full\_chain\_imp\_chain[OF\ full\_deriv\_wSts\_trancl\_weighted\_RP]$

**definition**  $sswSts :: 'a\ wstate\ llist\ where$

$sswSts = (SOME\ wSts').$

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

**lemma**  $sswSts:$

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

**unfolding**  $sswSts\_def$

**by** ( $rule\ someI\_ex[OF\ full\_chain\_tranclp\_imp\_exists\_full\_chain[OF\ full\_deriv\_wSts\_trancl\_weighted\_RP]]$ )

**lemmas**  $full\_deriv\_sswSts\_weighted\_RP = sswSts[THEN\ conjunct1]$

**lemmas**  $emb\_sswSts = sswSts[THEN\ conjunct2, THEN\ conjunct1]$

**lemmas**  $lfinite\_sswSts\_iff = emb\_lfinite[OF\ emb\_sswSts]$

**lemmas**  $lhd\_sswSts = sswSts[THEN\ conjunct2, THEN\ conjunct2, THEN\ conjunct1]$

**lemmas**  $llast\_sswSts = sswSts[THEN\ conjunct2, THEN\ conjunct2, THEN\ conjunct2]$

**lemmas**  $deriv\_sswSts\_weighted\_RP = full\_chain\_imp\_chain[OF\ full\_deriv\_sswSts\_weighted\_RP]$

**lemma** *not\_innull\_sswSts*:  $\neg \text{innull } \text{sswSts}$   
**using** *deriv\_sswSts\_weighted\_RP* **by** (*cases rule: chain.cases*) *auto*

**lemma** *empty\_ssgP0*: *wrp.P\_of\_wstate* (*lhd sswSts*) = {}  
**unfolding** *lhd\_sswSts* **by** (*subst derivation\_from.code*) *simp*

**lemma** *empty\_ssgQ0*: *wrp.Q\_of\_wstate* (*lhd sswSts*) = {}  
**unfolding** *lhd\_sswSts* **by** (*subst derivation\_from.code*) *simp*

**lemmas** *sswSts\_thms* = *full\_deriv\_sswSts\_weighted\_RP empty\_ssgP0 empty\_ssgQ0*

**abbreviation** *S\_ssgQ* :: 'a clause  $\Rightarrow$  'a clause **where**  
*S\_ssgQ*  $\equiv$  *wrp.S\_gQ sswSts*

**abbreviation** *ord\_Γ* :: 'a inference set **where**  
*ord\_Γ*  $\equiv$  *ground\_resolution\_with\_selection.ord\_Γ S\_ssgQ*

**abbreviation** *Rf* :: 'a clause set  $\Rightarrow$  'a clause set **where**  
*Rf*  $\equiv$  *standard\_redundancy\_criterion.Rf*

**abbreviation** *Ri* :: 'a clause set  $\Rightarrow$  'a inference set **where**  
*Ri*  $\equiv$  *standard\_redundancy\_criterion.Ri ord\_Γ*

**abbreviation** *saturated\_upto* :: 'a clause set  $\Rightarrow$  bool **where**  
*saturated\_upto*  $\equiv$  *redundancy\_criterion.saturated\_upto ord\_Γ Rf Ri*

**context**  
**assumes** *drp\_some*: *deterministic\_RP St0* = *Some R*  
**begin**

**lemma** *lfinite\_Sts*: *lfinite Sts*  
**proof** (*induct rule: deterministic\_RP.raw\_induct[OF drp\_some]*)  
**case** (*1 self\_call St St'*)  
**note** *ih* = *this(1)* **and** *step* = *this(2)*  
**show** ?*case*  
**using** *step* **by** (*subst derivation\_from.code, auto intro: ih*)  
**qed**

**lemma** *lfinite\_wSts*: *lfinite wSts*  
**by** (*rule lfinite\_lmap[THEN iffD2, OF lfinite\_Sts]*)

**lemmas** *lfinite\_sswSts* = *lfinite\_sswSts\_iff[THEN iffD2, OF lfinite\_wSts]*

**theorem**  
*deterministic\_RP\_saturated*: *saturated\_upto grounded\_R* (**is** ?*saturated*) **and**  
*deterministic\_RP\_model*:  $I \models s \text{ grounded\_} N0 \longleftrightarrow I \models s \text{ grounded\_} R$  (**is** ?*model*)  
**proof** –  
**obtain** *N' P' Q'* :: 'a dclause list **and** *n' k* :: nat **where**  
*k\_steps*: (*deterministic\_RP\_step*  $\overset{\sim}{\sim} k$ ) *St0* = (*N', P', Q', n'*) (**is**  $\_ = ?Stk$ ) **and**  
*final*: *is\_final\_dstate* (*N', P', Q', n'*) **and**  
*r*:  $R = \text{map fst } Q'$   
**using** *deterministic\_RP\_SomeD[OF drp\_some]* **by** *blast*

**have** *wrp*: *wstate\_of\_dstate St0*  $\rightsquigarrow_w^*$  *wstate\_of\_dstate (llast Sts)*  
**using** *lfinite\_chain\_imp\_rtranclp\_lhd\_llast*  
**by** (*metis (no\_types) deriv\_sswSts\_weighted\_RP derivation\_from.disc\_iff derivation\_from.simps(2)*)  
*lfinite\_Sts lfinite\_sswSts llast\_lmap llist.map\_sel(1) sswSts*)

**have** *last\_sts*: *llast Sts* = ?*Stk*  
**proof** –  
**have** (*deterministic\_RP\_step*  $\overset{\sim}{\sim} k'$ ) *St0'* = ?*Stk*  $\implies$  *llast (derivation\_from St0')* = ?*Stk*  
**for** *St0' k'*  
**proof** (*induct k' arbitrary: St0'*)



```

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

have ?saturated ∧ ?model
proof (cases [] ∈ set R)
  case True
  then have emp_in: {#} ∈ grounded_R
    unfolding grounding_of_cls_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] grounding_of_cls_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_cls_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_cls_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] 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 ( $\sim_w$ )  $\lambda St. I \models s$  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:
   $\neg$  satisfiable grounded_NO  $\longleftrightarrow$   $\{\#\} \in$  grounded_R (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume ?rhs
  then have  $\neg$  satisfiable grounded_R
    unfolding true_cls_def true_cls_def by force
  then show ?lhs
    using deterministic_RP_model[THEN iffD1] by blast
next
  assume ?lhs
  then have  $\neg$  satisfiable grounded_R
    using deterministic_RP_model[THEN iffD2] by blast
  then show ?rhs
    unfolding wrp.ord_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_NO
proof (rule ccontr)
  assume unsat:  $\neg$  satisfiable grounded_NO

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

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

```

```

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

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

```

## 4 Integration of IsaFoR Terms and the Knuth–Bendix Order

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

```

theory IsaFoR_Term
  imports
    Deriving.Derive
    Ordered_Resolution_Prover.Abstract_Substitution
    First_Order_Terms.Unification
    First_Order_Terms.Subsumption
    HOL_Cardinals.Wellorder_Extension
    Open_Induction.Restricted_Predicates
    Knuth_Bendix_Order.KBO
begin

```

**hide-const** (open) *mg*

**abbreviation** *subst\_apply\_literal* ::

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

**definition** *subst\_apply\_clause* ::

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

**abbreviation** *vars\_lit* ::  $(f, v)$  term literal  $\Rightarrow v$  set **where**

$\text{vars\_lit } L \equiv \text{vars\_term } (\text{atm\_of } L)$

**definition** *vars\_clause* ::  $(f, v)$  term clause  $\Rightarrow v$  set **where**

$\text{vars\_clause } C = \text{Union } (\text{set\_mset } (\text{image\_mset } \text{vars\_lit } C))$

**definition** *vars\_clause\_list* ::  $(f, v)$  term clause list  $\Rightarrow v$  set **where**

$\text{vars\_clause\_list } Cs = \text{Union } (\text{vars\_clause } \text{' set } Cs)$

**definition** *vars\_partitioned* ::  $(f, v)$  term clause list  $\Rightarrow \text{bool}$  **where**

$\text{vars\_partitioned } Cs \longleftrightarrow$

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

**lemma** *vars\_clause\_mono*:  $S \subseteq\# C \Longrightarrow \text{vars\_clause } S \subseteq \text{vars\_clause } C$

**unfolding** *vars\_clause\_def* **by** *auto*

**interpretation** *substitution\_ops*  $(\cdot) \text{ Var } (\circ_s) \cdot$

**lemma** *is\_ground\_atm\_is\_ground\_on\_var*:

**assumes** *is\_ground\_atm*  $(A \cdot \sigma)$  **and**  $v \in \text{vars\_term } A$

**shows** *is\_ground\_atm*  $(\sigma v)$

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

**case**  $(\text{Var } x)$

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

**next**

**case**  $(\text{Fun } f \text{ ts})$

**then show** *?case* **unfolding** *is\_ground\_atm\_def*

**by** *auto*

**qed**

**lemma** *is\_ground\_lit\_is\_ground\_on\_var*:

**assumes** *ground\_lit*: *is\_ground\_lit*  $(\text{subst\_lit } L \sigma)$  **and**  $v \text{ in } L$ :  $v \in \text{vars\_lit } L$

**shows** *is\_ground\_atm*  $(\sigma v)$

**proof** –

**let**  $?A = \text{atm\_of } L$

**from**  $v \text{ in } L$  **have**  $A\_p$ :  $v \in \text{vars\_term } ?A$

**by** *auto*

**then have** *is\_ground\_atm*  $(?A \cdot \sigma)$

**using** *ground\_lit* **unfolding** *is\_ground\_lit\_def* **by** *auto*

**then show** *?thesis*

**using**  $A\_p$  *is\_ground\_atm\_is\_ground\_on\_var* **by** *metis*

**qed**

**lemma** *is\_ground\_cls\_is\_ground\_on\_var*:

**assumes**

*ground\_clause*: *is\_ground\_cls*  $(\text{subst\_cls } C \sigma)$  **and**

$v \text{ in } C$ :  $v \in \text{vars\_clause } C$

**shows** *is\_ground\_atm*  $(\sigma v)$

**proof** –

**from**  $v \text{ in } C$  **obtain**  $L$  **where**  $L\_p$ :  $L \in\# C$   $v \in \text{vars\_lit } L$

**unfolding** *vars\_clause\_def* **by** *auto*

**then have** *is\_ground\_lit*  $(\text{subst\_lit } L \sigma)$

**using** *ground\_clause* **unfolding** *is\_ground\_cls\_def* *subst\_cls\_def* **by** *auto*

**then show** *?thesis*  
**using** *L\_p is\_ground\_lit\_is\_ground\_on\_var* **by** *metis*  
**qed**

**lemma** *is\_ground\_cls\_list\_is\_ground\_on\_var*:  
**assumes** *ground\_list: is\_ground\_cls\_list (subst\_cls\_list Cs 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**  $\forall v \in \text{vars\_lit } L. \sigma v = \tau v$   
**shows**  $\text{subst\_lit } L \sigma = \text{subst\_lit } L \tau$   
**using** *assms*  
**proof** (*induction L*)  
**case** (*Pos x*)  
**then have**  $\forall v \in \text{vars\_term } x. \sigma v = \tau v \implies \text{subst\_atm\_abbrev } x \sigma = \text{subst\_atm\_abbrev } x \tau$   
**using** *term\_subst\_eq* **by** *metis+*  
**then show** *?case*  
**unfolding** *subst\_lit\_def* **using** *Pos* **by** *auto*  
**next**  
**case** (*Neg x*)  
**then have**  $\forall v \in \text{vars\_term } x. \sigma v = \tau v \implies \text{subst\_atm\_abbrev } x \sigma = \text{subst\_atm\_abbrev } x \tau$   
**using** *term\_subst\_eq* **by** *metis+*  
**then show** *?case*  
**unfolding** *subst\_lit\_def* **using** *Neg* **by** *auto*  
**qed**

**lemma** *in\_list\_of\_mset\_in\_S*:  
**assumes**  $i < \text{length } (\text{list\_of\_mset } S)$   
**shows**  $\text{list\_of\_mset } S ! i \in \# S$   
**proof** –  
**from** *assms* **have**  $\text{list\_of\_mset } S ! i \in \text{set } (\text{list\_of\_mset } S)$   
**by** *auto*  
**then have**  $\text{list\_of\_mset } S ! i \in \# \text{mset } (\text{list\_of\_mset } S)$   
**by** (*meson in\_multiset\_in\_set*)  
**then show** *?thesis*  
**by** *auto*  
**qed**

**lemma** *same\_on\_vars\_clause*:  
**assumes**  $\forall v \in \text{vars\_clause } S. \sigma v = \tau v$   
**shows**  $\text{subst\_cls } S \sigma = \text{subst\_cls } S \tau$   
**by** (*smt assms image\_eqI image\_mset\_cong2 mem\_simps(9) same\_on\_vars\_lit set\_image\_mset subst\_cls\_def vars\_clause\_def*)

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

**proof** *unfold\_locales*

**show**  $\bigwedge A. A \cdot \text{Var} = A$   
**by** *auto*

**next**

**show**  $\bigwedge A \tau \sigma. A \cdot \tau \circ_s \sigma = A \cdot \tau \cdot \sigma$   
**by** *auto*

**next**

**show**  $\bigwedge \sigma \tau. (\bigwedge A. A \cdot \sigma = A \cdot \tau) \implies \sigma = \tau$   
**by** (*simp add: subst\_term\_eqI*)

```

next
  fix C :: ('f, nat) term clause
  fix  $\sigma$ 
  assume is_ground_cls (subst_cls C  $\sigma$ )
  then have ground_atms_ $\sigma$ :  $\bigwedge v. v \in \text{vars\_clause } C \implies \text{is\_ground\_atm } (\sigma v)$ 
    by (meson is_ground_cls is_ground_on_var)

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

  have all_ground_ $\tau$ : is_ground_atm ( $\tau v$ ) for v
  proof (cases v  $\in$  vars_clause C)
    case True
    then show ?thesis
      using ground_atms_ $\sigma$   $\tau_\sigma$  by auto
  next
    case False
    then show ?thesis
      unfolding  $\tau$ _def using ground_trm by auto
  qed
  have is_ground_subst  $\tau$ 
    unfolding is_ground_subst_def
  proof
    fix A
    show is_ground_atm (subst_atm_abbrev A  $\tau$ )
    proof (induction A)
      case (Var v)
      then show ?case using all_ground_ $\tau$  by auto
    next
      case (Fun f As)
      then show ?case using all_ground_ $\tau$ 
        by (simp add: is_ground_atm_def)
    qed
  qed
  moreover have  $\forall v \in \text{vars\_clause } C. \sigma v = \tau v$ 
    using  $\tau_\sigma$  unfolding vars_clause_list_def
    by blast
  then have subst_cls C  $\sigma = \text{subst\_cls } C \tau$ 
    using same_on_vars_clause by auto
  ultimately show  $\exists \tau. \text{is\_ground\_subst } \tau \wedge \text{subst\_cls } C \tau = \text{subst\_cls } C \sigma$ 
    by auto
next
  show wfP (strictly_generalizes_atm :: ('f, 'v) term  $\Rightarrow$  _  $\Rightarrow$  _)
    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

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

```

**by auto**  
**have**  $\forall i < \text{length } Cs. \forall S. S \subseteq\# Cs ! i \longrightarrow \text{subst\_cls } S (\sigma s ! i) = \text{subst\_cls } S \sigma$   
**proof** (*rule, rule, rule, rule*)  
**fix**  $i :: \text{nat}$  **and**  $S :: ('a, 'b) \text{ term literal multiset}$   
**assume**  
 $i < \text{length } Cs$  **and**  
 $S \subseteq\# Cs ! i$   
**then have**  $\forall v \in \text{vars\_clause } S. (\sigma s ! i) v = \sigma v$   
**using** *vars\_clause\_mono*[of  $S Cs ! i$ ]  $\sigma_p$  **by auto**  
**then show**  $\text{subst\_cls } S (\sigma s ! i) = \text{subst\_cls } S \sigma$   
**using** *same\_on\_vars\_clause* **by auto**  
**qed**  
**then show**  $\exists \tau. \forall i < \text{length } Cs. \forall S. S \subseteq\# Cs ! i \longrightarrow \text{subst\_cls } S (\sigma s ! i) = \text{subst\_cls } S \tau$   
**by auto**  
**qed**

**lemma** *vars\_in\_instance\_in\_range\_term*:  
 $\text{vars\_term } (\text{subst\_atm\_abbrev } A \sigma) \subseteq \text{Union } (\text{image } \text{vars\_term } (\text{range } \sigma))$   
**by** (*induction A*) **auto**

**lemma** *vars\_in\_instance\_in\_range\_lit*:  $\text{vars\_lit } (\text{subst\_lit } L \sigma) \subseteq \text{Union } (\text{image } \text{vars\_term } (\text{range } \sigma))$   
**proof** (*induction L*)  
**case** (*Pos A*)  
**have**  $\text{vars\_term } (A \cdot \sigma) \subseteq \text{Union } (\text{image } \text{vars\_term } (\text{range } \sigma))$   
**using** *vars\_in\_instance\_in\_range\_term*[of  $A \sigma$ ] **by blast**  
**then show** *?case* **by auto**  
**next**  
**case** (*Neg A*)  
**have**  $\text{vars\_term } (A \cdot \sigma) \subseteq \text{Union } (\text{image } \text{vars\_term } (\text{range } \sigma))$   
**using** *vars\_in\_instance\_in\_range\_term*[of  $A \sigma$ ] **by blast**  
**then show** *?case* **by auto**  
**qed**

**lemma** *vars\_in\_instance\_in\_range\_cls*:  
 $\text{vars\_clause } (\text{subst\_cls } C \sigma) \subseteq \text{Union } (\text{image } \text{vars\_term } (\text{range } \sigma))$   
**unfolding** *vars\_clause\_def subst\_cls\_def* **using** *vars\_in\_instance\_in\_range\_lit*[of  $\sigma$ ] **by auto**

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

**definition** *var\_map\_of\_subst* ::  $('f, \text{nat}) \text{ subst} \Rightarrow \text{nat} \Rightarrow \text{nat}$  **where**  
 $\text{var\_map\_of\_subst } \sigma v = \text{the\_Var } (\sigma v)$

**lemma** *len\_renamings\_apart*:  $\text{length } (\text{renamings\_apart } Cs) = \text{length } Cs$   
**by** (*induction Cs*) (*auto simp: Let\_def*)

**lemma** *renamings\_apart\_is\_Var*:  $\forall \sigma \in \text{set } (\text{renamings\_apart } Cs). \forall x. \text{is\_Var } (\sigma x)$   
**by** (*induction Cs*) (*auto simp: Let\_def*)

**lemma** *renamings\_apart\_inj*:  $\forall \sigma \in \text{set } (\text{renamings\_apart } Cs). \text{inj } \sigma$   
**proof** (*induction Cs*)  
**case** (*Cons a Cs*)  
**then have**  $\text{inj } (\lambda v. \text{Var } (\text{Suc } (v + \text{Max } (\text{vars\_clause\_list } (\text{subst\_cls\_lists } Cs (\text{renamings\_apart } Cs)) \cup \{0\}))))$   
**by** (*meson add\_right\_imp\_eq injI nat.inject term.inject*(1))  
**then show** *?case*  
**using** *Cons* **by** (*auto simp: Let\_def*)  
**qed auto**

**lemma** *finite\_vars\_clause*[*simp*]:  $\text{finite } (\text{vars\_clause } x)$

```

unfolding vars_clause_def by auto

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

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

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

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

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

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

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

```



```

then show vars_clause (subst_cls_lists (C # Cs) (renamings_apart (C # Cs)) ! i)  $\cap$ 
  vars_clause (subst_cls_lists (C # Cs) (renamings_apart (C # Cs)) ! j) =
  {}
using ij by auto
next
fix i' :: nat
assume i'j':
  i = Suc i'
  j = 0
have disjoint_C_Cs: vars_clause (subst_cls_lists (C # Cs) (renamings_apart (C # Cs)) ! 0)  $\cap$ 
  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  $\in$  vars_clause (subst_cls_lists Cs (renamings_apart Cs) ! i')
  then have (subst_cls_lists Cs (renamings_apart Cs) ! i')
     $\in$  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  $\in$  vars_clause (subst_cls_lists Cs (renamings_apart Cs) ! i')
    using i'j' ij
    unfolding subst_cls_lists_def by simp
  ultimately have  $\exists D \in$  set (subst_cls_lists Cs (renamings_apart Cs)). x  $\in$  vars_clause D
    by auto
}
then have vars_clause (subst_cls_lists Cs (renamings_apart Cs) ! i')
   $\subseteq$  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)  $\cap$ 
  vars_clause (subst_cls_lists Cs (renamings_apart Cs) ! i') =
  {} using disjoint_C_Cs unfolding vars_clause_list_def by auto
moreover
have subst_cls_lists Cs (renamings_apart Cs) ! i' =
  subst_cls_lists (C # Cs) (renamings_apart (C # Cs)) ! i
  using i'j' ij unfolding subst_cls_lists_def by (simp add: Let_def)
ultimately
show vars_clause (subst_cls_lists (C # Cs) (renamings_apart (C # Cs)) ! i)  $\cap$ 
  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'  $\neq$  j'
  using  $\langle i = \text{Suc } i' \rangle \langle j = \text{Suc } j' \rangle$  ij by blast
ultimately
have vars_clause (subst_cls_lists Cs (renamings_apart Cs) ! i')  $\cap$ 
  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)  $\cap$ 
  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_renamings (·) Var :: _ ⇒ ('f, nat) term (o_s) renamings_apart Fun undefined
proof unfold_locales
  fix Cs :: ('f, nat) term clause list
  show length (renamings_apart Cs) = length Cs
  using len_renamings_apart by auto
next
  fix Cs :: ('f, nat) term clause list
  fix ρ :: nat ⇒ ('f, nat) Term.term
  assume ρ_renaming: ρ ∈ set (renamings_apart Cs)
  {
  have inj_is_renaming:
     $\bigwedge \sigma :: ('f, nat) \text{ subst. } (\bigwedge x. \text{is\_Var } (\sigma x)) \implies \text{inj } \sigma \implies \text{is\_renaming } \sigma$ 
  proof -
    fix  $\sigma :: ('f, nat) \text{ subst}$ 
    fix  $x$ 
    assume is_var_σ:  $\bigwedge x. \text{is\_Var } (\sigma x)$ 
    assume inj_σ:  $\text{inj } \sigma$ 
    define σ' where σ' = var_map_of_subst σ
    have σ:  $\sigma = \text{Var} \circ \sigma'$ 
      unfolding σ'_def var_map_of_subst_def using is_var_σ by auto

    from is_var_σ inj_σ have inj_σ'
      unfolding is_renaming_def unfolding subst_domain_def inj_on_def σ'_def var_map_of_subst_def
      by (metis term.collapse(1))
    then have inv_σ' ∘ σ' = id
      using inv_o_cancel[of σ'] by simp
    then have  $\text{Var} \circ (\text{inv } \sigma' \circ \sigma') = \text{Var}$ 
      by simp
    then have  $\forall x. (\text{Var} \circ (\text{inv } \sigma' \circ \sigma')) x = \text{Var } x$ 
      by metis
    then have  $\forall x. ((\text{Var} \circ \sigma') \circ_s (\text{Var} \circ (\text{inv } \sigma')) x = \text{Var } x$ 
      unfolding subst_compose_def by auto
    then have  $\sigma \circ_s (\text{Var} \circ (\text{inv } \sigma')) = \text{Var}$ 
      using σ by auto
    then show is_renaming σ
      unfolding is_renaming_def by blast
    qed
  }
  then show is_renaming ρ
    using ρ_renaming by auto
next
  fix Cs :: ('f, nat) term clause list
  have vars_partitioned (subst_cls_lists Cs (renamings_apart Cs))

```

```

    using vars_partitioned_renamings_apart by auto
  then show var_disjoint (subst_cls_lists Cs (renamings_apart Cs))
    using vars_partitioned_var_disjoint by auto
next
show  $\bigwedge \sigma \text{ As Bs. Fun undefined As} \cdot \sigma = \text{Fun undefined Bs} \longleftrightarrow \text{map } (\lambda A. A \cdot \sigma) \text{ As} = \text{Bs}$ 
  by simp
qed

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

derive compare term
derive compare literal

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

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

definition Pairs where
  Pairs AAA = concat (compare_linorder.sorted_list_of_set
    ((pairs  $\circ$  compare_linorder.sorted_list_of_set) 'AAA))

lemma unifies_all_pairs_iff:
  ( $\forall p \in \text{set (pairs xs). fst } p \cdot \sigma = \text{snd } p \cdot \sigma$ )  $\longleftrightarrow$  ( $\forall a \in \text{set xs. } \forall b \in \text{set xs. } a \cdot \sigma = b \cdot \sigma$ )
proof (induct xs rule: pairs.induct)
  case (1 x y xs)
  then show ?case
    unfolding pairs.simps list.set ball_Un ball_simps simp_thms fst_conv snd_conv by metis
qed simp_all

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

lemma in_pairs_sorted_list_of_set_in_set:
  assumes
    finite AAA
     $\forall AA \in \text{AAA. finite AA}$ 

```

```

  AB_pairs ∈ (pairs ∘ compare_linorder.sorted_list_of_set) ‘ AAA and
  (A :: _ :: compare, B) ∈ set AB_pairs
shows ∃ AAA. 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 ∧ AAA. 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 ∃ AAA. 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  $\sigma \in AA \cdot_{set} \sigma$ 
  using a assms asm card_AA_ $\sigma$  by auto
moreover
have subst_atm_abbrev B  $\sigma \in AA \cdot_{set} \sigma$ 
  using a assms asm card_AA_ $\sigma$  by auto
ultimately
show ?thesis
  using a assms asm card_AA_ $\sigma$  by (metis (no_types, lifting) card_Suc_eq singletonD)
next
case False
then have card (AA  $\cdot_{set} \sigma$ ) = 0
  using a assms asm card_AA_ $\sigma$ 
  by arith
then show ?thesis
  using a assms asm card_AA_ $\sigma$  by auto
qed
}
then show  $\sigma \in unifiers$  (set (Pairs AAA))
  unfolding Pairs_def unifiers_def by auto
qed

end

```

**definition** mgu\_sets AAA = map\_option subst\_of (unify (Pairs AAA) [])

**lemma** mgu\_sets\_is\_imgu:

**fixes** AAA :: ('a :: compare, nat) term set set **and**  $\sigma$  :: ('a, nat) subst  
**assumes** fin: finite AAA  $\forall AA \in AAA$ . finite AA **and** mgu\_sets AAA = Some  $\sigma$   
**shows** is\_imgu  $\sigma$  AAA

**proof** –

**have** Unifiers.is\_imgu  $\sigma$  (set (Pairs AAA))  
**using** assms unify\_sound **unfolding** mgu\_sets\_def **by** blast  
**thus** ?thesis  
**unfolding** Unifiers.is\_imgu\_def is\_imgu\_def unifiers\_Pairs[OF fin]  
**by** simp

**qed**

**interpretation** mgu ( $\cdot$ ) Var ::  $\_ \Rightarrow$  ('f :: compare, nat) term ( $\circ_s$ ) renamings\_apart  
 Fun undefined mgu\_sets

**proof** unfold\_locales

**fix** AAA :: ('a :: compare, nat) term set set **and**  $\sigma$  :: ('a, nat) subst  
**assume** fin: finite AAA  $\forall AA \in AAA$ . finite AA **and** mgu\_sets AAA = Some  $\sigma$   
**thus** is\_mgu  $\sigma$  AAA  
**using** mgu\_sets\_is\_imgu **by** auto

**next**

**fix** AAA :: ('a :: compare, nat) term set set **and**  $\sigma$  :: ('a, nat) subst  
**assume** fin: finite AAA  $\forall AA \in AAA$ . finite AA **and** is\_unifiers  $\sigma$  AAA  
**then have**  $\sigma \in unifiers$  (set (Pairs AAA))  
**unfolding** is\_mgu\_def unifiers\_Pairs[OF fin] **by** auto  
**then show**  $\exists \tau$ . mgu\_sets AAA = Some  $\tau$   
**using** unify\_complete **unfolding** mgu\_sets\_def **by** blast

**qed**

**interpretation** imgu ( $\cdot$ ) Var ::  $\_ \Rightarrow$  ('f :: compare, nat) term ( $\circ_s$ ) renamings\_apart  
 Fun undefined mgu\_sets

**proof** unfold\_locales

**fix** AAA :: ('a :: compare, nat) term set set **and**  $\sigma$  :: ('a, nat) subst  
**assume** fin: finite AAA  $\forall AA \in AAA$ . finite AA **and** mgu\_sets AAA = Some  $\sigma$   
**thus** is\_imgu  $\sigma$  AAA  
**by** (rule mgu\_sets\_is\_imgu)

**qed**

**derive** linorder prod

derive linorder list

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

```

record 'f weights =
  w :: 'f × nat ⇒ nat
  w0 :: nat
  pr_strict :: 'f × nat ⇒ 'f × nat ⇒ bool
  least :: 'f ⇒ bool
  scf :: 'f × nat ⇒ nat ⇒ nat

class weighted =
  fixes weights :: 'a weights
  assumes weights_adm:
    admissible_kbo
    (w weights) (w0 weights) (pr_strict weights) ((pr_strict weights)==) (least weights) (scf weights)
  and pr_strict_total: fi = gj ∨ pr_strict weights fi gj ∨ pr_strict weights gj fi
  and pr_strict_asymp: asymp (pr_strict weights)
  and scf_ok: i < n ⇒ scf weights (f, n) i ≤ 1

instantiation unit :: weighted begin

```

```

definition weights_unit :: unit weights where weights_unit =
  (λw = Suc ∘ snd, w0 = 1, pr_strict = λ(⟦, n) (⟦, m). n > m, least = λ_. True, scf = λ_ . 1)

```

```

instance
  by (intro_classes, unfold_locales) (auto simp: weights_unit_def SN_iff_wf irreflp_def
    intro: asympI intro!: wf_subset[OF wf_inv_image[OF wf], of _ snd])
end

```

**global-interpretation** KBO:

```

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

```

```

lemma kbo_code[code]: kbo s t =
  (let wt = weight t; ws = weight s in
  if vars_term_ms (KBO.SCF t) ⊆# vars_term_ms (KBO.SCF s) ∧ wt ≤ ws
  then
    (if wt < ws then (True, True)
    else
      (case s of
        Var y ⇒ (False, case t of Var x ⇒ True | Fun g ts ⇒ ts = [] ∧ least weights g)
      | Fun f ss ⇒
        (case t of
          Var x ⇒ (True, True)
          | Fun g ts ⇒
            if pr_strict weights (f, length ss) (g, length ts) then (True, True)
            else if (f, length ss) = (g, length ts) then lex_ext_unbounded kbo ss ts
            else (False, False))))
    else (False, False))
by (subst KBO.kbo.simps) (auto simp: Let_def split: term.splits)

```

**definition** less\_kbo s t = fst (kbo t s)

```

lemma less_kbo_gtotal: ground s ⇒ ground t ⇒ s = t ∨ less_kbo s t ∨ less_kbo t s
  unfolding less_kbo_def using KBO.S_ground_total by (metis pr_strict_total subset_UNIV)

```

```

lemma less_kbo_subst:
  fixes σ :: ('f :: weighted, 'v) subst
  shows less_kbo s t ⇒ less_kbo (s · σ) (t · σ)

```

```

unfolding less_kbo_def by (rule KBO.S_subst)

lemma wfp_less_kbo: wfp less_kbo
proof -
  have SN {(x, y). fst (kbo x y)}
    using pr_strict_asymp by (fastforce simp: asympI irreflp_def intro!: KBO.S_SN scf_ok)
  then show ?thesis
    unfolding SN_iff_wfp wfp_def by (rule wf_subset) (auto simp: less_kbo_def)
qed

instantiation term :: (weighted, type) linorder begin

definition leq_term = (SOME leq. {(s,t). less_kbo s t} ⊆ leq ∧ Well_order leq ∧ Field leq = UNIV)

lemma less_trm_extension: {(s,t). less_kbo s t} ⊆ leq_term
  unfolding leq_term_def
  by (rule someI2_ex[OF total_well_order_extension[OF wfp_less_kbo[unfolded wfp_def]]]) auto

lemma less_trm_well_order: well_order leq_term
  unfolding leq_term_def
  by (rule someI2_ex[OF total_well_order_extension[OF wfp_less_kbo[unfolded wfp_def]]]) auto

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

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

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

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

```

```

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

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

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

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

end

```

## 5 An Executable Algorithm for Clause Subsumption

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

```

theory Executable_Subsumption
imports
  IsaFoR_Term
  First_Order_Terms.Matching
begin

```

### 5.1 Naive Implementation of Clause Subsumption

```

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

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

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

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

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

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

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

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

```



**lemma** *extends\_subst\_fun\_upd\_matching*:

$\sigma x = \text{Some } t \implies \text{extends\_subst } (\sigma(x \mapsto t)) \tau \longleftrightarrow \text{extends\_subst } \sigma \tau$   
**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*:

$\text{extends\_subst } \sigma \tau \implies \text{vars\_term } t \subseteq \text{dom } \sigma \implies t \cdot \text{subst\_of\_map } \text{Var } \sigma = t \cdot \text{subst\_of\_map } \text{Var } \tau$   
**by** (*force simp: extends\_subst\_def subst\_of\_map\_def split: option.splits intro!: term\_subst\_eq*)

**lemma** *extends\_subst\_cong\_lit*:

$\text{extends\_subst } \sigma \tau \implies \text{vars\_lit } L \subseteq \text{dom } \sigma \implies L \cdot \text{lit } \text{subst\_of\_map } \text{Var } \sigma = L \cdot \text{lit } \text{subst\_of\_map } \text{Var } \tau$   
**by** (*cases L*) (*auto simp: extends\_subst\_cong\_term*)

**definition** *subsumes\_modulo*  $C D \sigma =$

$(\exists \tau. \text{dom } \tau = \text{vars\_clause } C \cup \text{dom } \sigma \wedge \text{extends\_subst } \sigma \tau \wedge \text{subst\_cls } C (\text{subst\_of\_map } \text{Var } \tau) \subseteq\# D)$

**abbreviation** *subsumes\_list\_modulo* **where**

*subsumes\_list\_modulo*  $Ls Ks \sigma \equiv \text{subsumes\_modulo } (\text{mset } Ls) (\text{mset } Ks) \sigma$

**lemma** *vars\_clause\_add\_mset[simp]*: *vars\_clause* (*add\_mset*  $L C$ ) = *vars\_lit*  $L \cup \text{vars\_clause } C$

**unfolding** *vars\_clause\_def* **by** *auto*

**lemma** *subsumes\_list\_modulo\_Cons*: *subsumes\_list\_modulo* ( $L \# Ls$ )  $Ks \sigma \longleftrightarrow$

$(\exists K \in \text{set } Ks. \exists \tau. \text{extends\_subst } \sigma \tau \wedge \text{dom } \tau = \text{vars\_lit } L \cup \text{dom } \sigma \wedge L \cdot \text{lit } (\text{subst\_of\_map } \text{Var } \tau) = K$   
 $\wedge \text{subsumes\_list\_modulo } Ls (\text{remove1 } K Ks) \tau)$

**unfolding** *subsumes\_modulo\_def*

**proof** (*safe, goal\_cases left\_right right\_left*)

**case** (*left\_right*  $\tau$ )

**then show** *?case*

**by** (*intro* *beXI*[*of*  $L \cdot \text{lit } \text{subst\_of\_map } \text{Var } \tau$ ]

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

(*auto*  $0 \ 3$  *simp: extends\_subst\_def dom\_def split: if\_splits*)

*simp: insert\_subset\_eq\_iff subst\_lit\_def intro!: extends\_subst\_cong\_lit*)

**next**

**case** (*right\_left*  $K \tau \tau'$ )

**then show** *?case*

**by** (*intro* *beXI*[*of*  $L \cdot \text{lit } \text{subst\_of\_map } \text{Var } \tau$ ] *exI*[*of*  $\tau'$ ], *intro* *conjI* *exI*[*of*  $\tau$ ])

(*auto simp: insert\_subset\_eq\_iff subst\_lit\_def extends\_subst\_cong\_lit*)

*intro: extends\_subst\_trans*)

**qed**

**lemma** *decompose\_Some\_var\_terms*: *decompose* (*Fun*  $f ss$ ) (*Fun*  $g ts$ ) = *Some eqs*  $\implies$

$f = g \wedge \text{length } ss = \text{length } ts \wedge \text{eqs} = \text{zip } ss ts \wedge$

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

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

**by** (*drule* *decompose\_Some*)

(*fastforce simp: in\_set\_zip in\_set\_conv\_nth Bex\_def image\_iff*)

**lemma** *match\_term\_list\_sound*: *match\_term\_list*  $tus \sigma = \text{Some } \tau \implies$

$\text{extends\_subst } \sigma \tau \wedge \text{dom } \tau = (\bigcup (t, u) \in \text{set } tus. \text{vars\_term } t) \cup \text{dom } \sigma \wedge$

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

**proof** (*induct*  $tus \sigma$  *rule: match\_term\_list.induct*)

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

**then show** *?case*

**by** (*auto*  $0 \ 3$  *simp: extends\_subst\_fun\_upd\_new extends\_subst\_fun\_upd\_matching*)

*subst\_of\_map\_def dest: extends\_subst\_extends simp del: fun\_upd\_apply*

*split: if\_splits option.splits*)

**next**

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

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

```

  match_term_list (eqs @ P) 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*:  $\text{match\_term\_list } tus \sigma = \text{None} \implies$   
 $\text{extends\_subst } \sigma \tau \implies \text{dom } \tau = (\bigcup (t, u) \in \text{set } tus. \text{vars\_term } t) \cup \text{dom } \sigma \implies$   
 $(\exists (t, u) \in \text{set } tus. t \cdot \text{subst\_of\_map } \text{Var } \tau \neq u)$

```

proof (induct tus σ arbitrary: τ rule: match_term_list.induct)
  case (2 x t P 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  $\sigma$   $\longleftrightarrow$  subsumes_list_modulo Ls Ks  $\sigma$ 
proof (induction Ls Ks  $\sigma$  rule: subsumes_list.induct[case_names Nil Cons])
case (Cons L Ls Ks  $\sigma$ )
show ?case
  unfolding subsumes_list_modulo_Cons subsumes_list.simps
proof ((intro bex_cong[OF refl] ext iffI; elim exE conjE), goal_cases LR RL)
  case (LR K)
  show ?case
    by (insert LR; cases K; cases L; auto simp: Cons.IH split: option.splits dest!: match_term_list_sound)
next
case (RL K  $\tau$ )
then show ?case
proof (cases match_term_list [(atm_of L, atm_of K)]  $\sigma$ )
  case None
    with RL show ?thesis
      by (auto simp: Cons.IH dest!: match_term_list_complete)
  next
  case (Some  $\tau'$ )
    with RL show ?thesis
      using unique_extends_subst[of  $\sigma$   $\tau$   $\tau'$  atm_of L]
      by (auto simp: Cons.IH dest!: match_term_list_sound)
qed
qed
qed (auto simp: subsumes_modulo_def subst_cls_def vars_clause_def intro: extends_subst_refl)

```

```

lemma subsumes_subsumes_list[code_unfold]:
  subsumes (mset Ls) (mset Ks) = subsumes_list Ls Ks Map.empty
unfolding subsumes_list_alt[of Ls Ks Map.empty]
proof
  assume subsumes (mset Ls) (mset Ks)
  then obtain  $\sigma$  where subst_cls (mset Ls)  $\sigma \subseteq\#$  mset Ks unfolding subsumes_def by blast
  moreover define  $\tau$  where  $\tau = (\lambda x. \text{if } x \in \text{vars\_clause } (mset Ls) \text{ then Some } (\sigma x) \text{ else None})$ 
  ultimately show subsumes_list_modulo Ls Ks Map.empty
  unfolding subsumes_modulo_def
  by (subst (asm) same_on_vars_clause[of  $\_$   $\sigma$  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  $\wedge$   $\neg$  subsumes_list Ks Ls Map.empty)
unfolding strictly_subsumes_def subsumes_subsumes_list by simp

```

```

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

```

```

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

```

**lemma** *subsumes\_list\_Cons\_filter\_iff*:  
**assumes** *sorted\_wrt*: *sorted\_wrt* *leq* (*L* # *Ls*) **and** *trans*: *transp* *leq*  
**and** *match*:  $(\bigwedge L K \sigma \tau. \text{match\_term\_list } [(atm\_of\ L, atm\_of\ K)] \sigma = \text{Some } \tau \implies is\_pos\ L = is\_pos\ K \implies leq\ L\ K)$   
**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*  $\Rightarrow (f, 'v)$  *term*  $\Rightarrow bool$  **where**  
*leq\_head* *t* *u* = (*case* (*root* *t*, *root* *u*) *of*  
 (*None*, *\_*)  $\Rightarrow True$   
 | (*\_*, *None*)  $\Rightarrow False$   
 | (*Some* *f*, *Some* *g*)  $\Rightarrow f \leq g$ )

**definition** *leq\_lit* *L* *K* = (*case* (*K*, *L*) *of*  
 (*Neg* *\_*, *Pos* *\_*)  $\Rightarrow True$   
 | (*Pos* *\_*, *Neg* *\_*)  $\Rightarrow False$   
 | *\_*  $\Rightarrow leq\_head\ (atm\_of\ L)\ (atm\_of\ K)$ )

**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* *A* *leq\_lit*  
**unfolding** *reflp\_on\_def* *leq\_lit\_def* *leq\_head\_def* **by** (*auto* *split*: *option.splits* *literal.splits*)

**lemma** *total\_leq\_lit[simp]*: *totalp\_on* *A* *leq\_lit*  
**unfolding** *totalp\_on\_def* *leq\_lit\_def* *leq\_head\_def* **by** (*auto* *split*: *option.splits* *literal.splits*)

**lemma** *leq\_head\_subst[simp]*: *leq\_head* *t* (*t*  $\cdot$   $\sigma$ )  
**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*)]  $\sigma = \text{Some } \tau \implies is\_pos\ L = is\_pos\ K \implies leq\_lit\ L\ K$   
**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*  $\sigma = True$   
| *subsumes\_list\_filter* (*L* # *Ls*) *Ks*  $\sigma =$   
 (*let* *Ks* = *filter* (*leq\_lit* *L*) *Ks* *in*  
 $(\exists K \in set\ Ks. is\_pos\ K = is\_pos\ L \wedge$   
 (*case* *match\_term\_list* [(*atm\_of* *L*, *atm\_of* *K*)]  $\sigma$  *of*  
*None*  $\Rightarrow False$   
 | *Some*  $\varrho \Rightarrow subsumes\_list\_filter\ Ls\ (remove1\ K\ Ks)\ \varrho$ ))

**lemma** *sorted\_wrt\_subsumes\_list\_subsumes\_list\_filter*:  
*sorted\_wrt* *leq\_lit* *Ls*  $\implies subsumes\_list\ Ls\ Ks\ \sigma = subsumes\_list\_filter\ Ls\ Ks\ \sigma$

**proof** (*induction* *Ls* *arbitrary*: *Ks*  $\sigma$ )

**case** (*Cons* *L* *Ls*)

**from** *Cons.prem*s **have** *subsumes\_list* (*L* # *Ls*) *Ks*  $\sigma = subsumes\_list\ (L\ \#\ Ls)\ (filter\ (leq\_lit\ L)\ Ks)\ \sigma$   
**by** (*intro* *subsumes\_list\_Cons\_filter\_iff*[*symmetric*]) (*auto* *dest*: *leq\_lit\_match*)

**also** **have** *subsumes\_list* (*L* # *Ls*) (*filter* (*leq\_lit* *L*) *Ks*)  $\sigma = subsumes\_list\_filter\ (L\ \#\ Ls)\ Ks\ \sigma$

```

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

```

We can easily show that this QuickSort is correct:

```

theorem mset_quickSort [simp]: mset (quicksort R xs) = mset xs
  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 totalp_on (set xs) R and reflp_on (set xs) R
  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  $\neg R b a$   $a \in \text{set } (x\#xs)$   $b \in \text{set } (x\#xs)$  for a b
    using 2.premis that unfolding totalp_on_def reflp_on_def by (cases a = b) auto
  have sorted_wrt R (quicksort R (filter ( $\lambda y. R y x$ ) xs))
    sorted_wrt R (quicksort R (filter ( $\lambda y. \neg R y x$ ) xs))
    using 2.premis by (intro 2.IH; auto simp: totalp_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  $\sigma$  = subsumes_list_filter (quicksort leq_lit Ls) Ks  $\sigma$ 
  by (rule trans[OF box_equals[OF subsumes_list_alt[symmetric] subsumes_list_alt[symmetric]]
    sorted_wrt_subsumes_list_subsumes_list_filter])
  (auto simp: sorted_wrt_quickSort)

```

end

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

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

```

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

```

```

    IsaFoR_Term
begin
global-interpretation RP: deterministic_FO_resolution_prover where
  S =  $\lambda\_.$  {#} and
  subst_atm = ( $\cdot$ ) and
  id_subst = Var ::  $\_ \Rightarrow$  ('f :: {weighted, compare_order}, nat) term and
  comp_subst = ( $\circ_s$ ) and
  renamings_apart = renamings_apart and
  atm_of_atms = Fun undefined and
  mgu = mgu_sets and
  less_atm = less_kbo and
  size_atm = size and
  timestamp_factor = 1 and
  size_factor = 1
defines deterministic_RP = RP.deterministic_RP
and deterministic_RP_step = RP.deterministic_RP_step
and is_final_dstate = RP.is_final_dstate
and is_reducible_lit = RP.is_reducible_lit
and is_tautology = RP.is_tautology
and maximal_wrt = RP.maximal_wrt
and reduce = RP.reduce
and reduce_all = RP.reduce_all
and reduce_all2 = RP.reduce_all2
and remdups_cls = RP.remdups_cls
and resolve = RP.resolve
and resolve_on = RP.resolve_on
and resolvable = RP.resolvable
and resolvent = RP.resolvent
and resolve_rename = RP.resolve_rename
and resolve_rename_either_way = RP.resolve_rename_either_way
and select_min_weight_clause = RP.select_min_weight_clause
and strictly_maximal_wrt = RP.strictly_maximal_wrt
and strictly_subsume = RP.strictly_subsume
and subsume = RP.subsume
and weight = RP.weight
and St0 = RP.St0
and sorted_list_of_set = linorder.sorted_list_of_set (le_of_comp compare)
and sort_key = linorder.sort_key (le_of_comp compare)
and insort_key = linorder.insort_key (le_of_comp compare)
by (unfold_locales)
  (auto simp: less_kbo_subst is_ground_atm_ground less_kbo_less intro: ground_less_less_kbo)

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

lemma remove1_mset_subset_eq: remove1_mset a A  $\subseteq\#$  B  $\longleftrightarrow$  A  $\subseteq\#$  add_mset a B
by (metis add_mset_add_single_subset_eq_diff_conv)

```



**export-code** *St0* in *SML*  
**export-code** *deterministic\_RP* in *SML module-name RP*

**instantiation** *nat* :: *weighted begin*

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

**instance**

**by** (*intro\_classes, unfold\_locales*)  
*(auto simp: weights\_nat\_def SN\_iff\_wf irreflp\_def prod\_encode\_def*  
*intro: asympI intro!: wf\_subset[OF wf\_lex\_prod])*

**end**

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

*prover N* = (case *deterministic\_RP (St0 N 0)* of  
*None*  $\Rightarrow$  *True*  
| *Some R*  $\Rightarrow$   $\exists \notin \text{set } R$ )

**theorem** *prover\_complete\_refutation*: *prover N*  $\longleftrightarrow$  *satisfiable (RP.grounded\_N0 N)*

**unfolding** *prover\_def St0\_def*

**using** *RP.deterministic\_RP\_complete[of N 0] RP.deterministic\_RP\_refutation[of N 0]*

**by** (*force simp: grounding\_of\_cls\_def grounding\_of\_cls\_def ex\_ground\_subst*  
*split: option.splits if\_splits*)

**definition** *string\_literal\_of\_nat* :: *nat*  $\Rightarrow$  *String.literal* **where**

*string\_literal\_of\_nat n* = *String.implode (show n)*

**export-code** *prover Fun Var Pos Neg string\_literal\_of\_nat 0::nat Suc* in *SML module-name RPx*

**abbreviation** *p*  $\equiv$  *Fun 42*

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

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

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

**abbreviation** *X*  $\equiv$  *Var 0*

**abbreviation** *Y*  $\equiv$  *Var 1*

**abbreviation** *Z*  $\equiv$  *Var 2*

**value** *prover*

$(\lambda([[\text{Neg } (\mathbf{p}[X, Y, Z]), \text{Pos } (\mathbf{p}[Y, Z, X])], 1],$   
 $([\text{Pos } (\mathbf{p}[\mathbf{c}, \mathbf{a}, \mathbf{b}])], 1),$   
 $([\text{Neg } (\mathbf{p}[\mathbf{b}, \mathbf{c}, \mathbf{a}])], 1)])$   
::  $((\text{nat}, \text{nat}) \text{Term.term literal list} \times \text{nat}) \text{list}$ )

**value** *prover*

$(\lambda([\text{Pos } (\mathbf{p}[X, Y])], 1), ([\text{Neg } (\mathbf{p}[X, X])], 1))$   
::  $((\text{nat}, \text{nat}) \text{Term.term literal list} \times \text{nat}) \text{list}$ )

**value** *prover*  $(\lambda([\text{Neg } (\mathbf{p}[X, Y, Z]), \text{Pos } (\mathbf{p}[Y, Z, X])], 1))$

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

**definition** *mk\_MSC015\_1* :: *nat*  $\Rightarrow$   $((\text{nat}, \text{nat}) \text{Term.term literal list} \times \text{nat}) \text{list}$  **where**

*mk\_MSC015\_1 n* =  
*(let*  
*init* =  $([\text{Pos } (\mathbf{p} (\text{replicate } n \ \mathbf{a}))], 1)$ ;  
*rules* =  $\text{map } (\lambda i. ([\text{Neg } (\mathbf{p} (\text{map } \text{Var } [0 \ ..< n - i - 1] @ \mathbf{a} \ \# \ \text{replicate } i \ \mathbf{b}))],$   
 $\text{Pos } (\mathbf{p} (\text{map } \text{Var } [0 \ ..< n - i - 1] @ \mathbf{b} \ \# \ \text{replicate } i \ \mathbf{a}))], 1)) [0 \ ..< n]$ ;  
*goal* =  $([\text{Neg } (\mathbf{p} (\text{replicate } n \ \mathbf{b}))], 1)$   
*in* *init*  $\#$  *rules*  $@$  [*goal*])

**value** *prover* (*mk\_MSC015\_1 1*)



```

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

```

**lemma**

**assumes**

```

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

```

**shows** *False*

**using** *assms* **by** *metis*

**end**