Formalization of Recursive Path Orders for Lambda-Free Higher-Order Terms

Jasmin Christian Blanchette, Uwe Waldmann, and Daniel Wand

September 23, 2021

Abstract

This Isabelle/HOL formalization defines recursive path orders (RPOs) for higher-order terms without λ-abstraction and proves many useful properties about them. The main order fully coincides with the standard RPO on first-order terms also in the presence of currying, distinguishing it from previous work. An optimized variant is formalized as well. It appears promising as the basis of a higher-order superposition calculus.

Contents

1 Introduction

This Isabelle/HOL formalization defines recursive path orders (RPOs) for higher-order terms without λ-abstraction and proves many useful properties about them. The main order fully coincides with the standard RPO on first-order terms also in the presence of currying, distinguishing it from previous work. An optimized variant is formalized as well. It appears promising as the basis of a higher-order superposition calculus.

We refer to the following conference paper for details:

> Jasmin Christian Blanchette, Uwe Waldmann, Daniel Wand:
> A Lambda-Free Higher-Order Recursive Path Order.
> FoSSaCS 2017: 461-479
> https://www.cs.vu.nl/~jbe248/lambda_free_rpo_conf.pdf

2 Utilities for Lambda-Free Orders

theory *Lambda_Free_Util*
imports *HOL−Library.Extended_Nat HOL−Library.Multiset_Order*
begin

This theory gathers various lemmas that likely belong elsewhere in Isabelle or the *Archive of Formal Proofs*. Most (but certainly not all) of them are used to formalize orders on λ-free higher-order terms.

2.1 Finite Sets

lemma *finite_set_fold_singleton*[*simp*]: *Finite_Set.fold f z* $\{x\}$ $=$ *f x z*
proof −
  have *fold_graph f z* $\{x\}$ $(f\ x\ z)$
    by (*auto intro*: *fold_graph.intros*)
  moreover
  {
    fix *X y*
    have *fold_graph f z X y* $\Longrightarrow$ $(X = \{\} \longrightarrow y = z) \wedge (X = \{x\} \longrightarrow y = f\ x\ z)$
      by (*induct rule*: *fold_graph.induct*) *auto*
  }
  ultimately have (*THE y*. *fold_graph f z* $\{x\}$ *y*) $=$ *f x z*

by *blast*
thus *?thesis*
by (*simp add*: *Finite_Set.fold_def*)
qed

2.2 Function Power

lemma *funpow_lesseq_iter*:
fixes $f :: ('a::order) \Rightarrow 'a$
assumes *mono*: $\bigwedge k.\ k \leq f\ k$ **and** *m_le_n*: $m \leq n$
shows $(f \hat{\ } m)\ k \leq (f \hat{\ } n)\ k$
using *m_le_n* **by** (*induct n*) (*fastforce simp*: *le_Suc_eq intro*: *mono order_trans*)+

lemma *funpow_less_iter*:
fixes $f :: ('a::order) \Rightarrow 'a$
assumes *mono*: $\bigwedge k.\ k < f\ k$ **and** *m_lt_n*: $m < n$
shows $(f \hat{\ } m)\ k < (f \hat{\ } n)\ k$
using *m_lt_n* **by** (*induct n*) (*auto*, *blast intro*: *mono less_trans dest*: *less_antisym*)

2.3 Least Operator

lemma *Least_eq*[*simp*]: $(LEAST\ y.\ y = x) = x$ **and** $(LEAST\ y.\ x = y) = x$ **for** $x :: 'a::order$
by (*blast intro*: *Least_equality*)+

lemma *Least_in_nonempty_set_imp_ex*:
fixes $f :: 'b \Rightarrow ('a::wellorder)$
assumes
A_nemp: $A \neq \{\}$ **and**
P_least: $P\ (LEAST\ y.\ \exists x \in A.\ y = f\ x)$
shows $\exists x \in A.\ P\ (f\ x)$
proof −
obtain *a* **where** *a*: $a \in A$
using *A_nemp* **by** *fast*
have $\exists x.\ x \in A \wedge (LEAST\ y.\ \exists x.\ x \in A \wedge y = f\ x) = f\ x$
by (*rule LeastI*[*of* _ *f a*]) (*fast intro*: *a*)
thus *?thesis*
by (*metis P_least*)
qed

lemma *Least_eq_0_enat*: $P\ 0 \Longrightarrow (LEAST\ x :: enat.\ P\ x) = 0$
by (*simp add*: *Least_equality*)

2.4 Antisymmetric Relations

lemma *irrefl_trans_imp_antisym*: *irrefl r* $\Longrightarrow$ *trans r* $\Longrightarrow$ *antisym r*
unfolding *irrefl_def trans_def antisym_def* **by** *fast*

lemma *irreflp_transp_imp_antisymP*: *irreflp p* $\Longrightarrow$ *transp p* $\Longrightarrow$ *antisymp p*
by (*fact irrefl_trans_imp_antisym* [*to_pred*])

2.5 Acyclic Relations

lemma *finite_nonempty_ex_succ_imp_cyclic*:
assumes
fin: *finite A* **and**
nemp: $A \neq \{\}$ **and**
ex_y: $\forall x \in A.\ \exists y \in A.\ (y, x) \in r$
shows ¬ *acyclic r*
proof −
let $?R = \{(x, y).\ x \in A \wedge y \in A \wedge (x, y) \in r\}$

have *R_sub_r*: $?R \subseteq r$
by *auto*

```
  have ?R ⊆ A × A
    by auto
  hence fin_R: finite ?R
    by (auto intro: fin dest!: infinite_super)

  have ¬ acyclic ?R
    by (rule notI, drule finite_acyclic_wf[OF fin_R], unfold wf_eq_minimal, drule spec[of _ A],
      use ex_y nemp in blast)
  thus ?thesis
    using R_sub_r acyclic_subset by auto
qed
```

2.6 Reflexive, Transitive Closure

```
lemma relcomp_subset_left_imp_relcomp_trancl_subset_left:
  assumes sub: R O S ⊆ R
  shows R O S^* ⊆ R
proof
  fix x
  assume x ∈ R O S^*
  then obtain n where x ∈ R O S ^^ n
    using rtrancl_imp_relpow by fastforce
  thus x ∈ R
  proof (induct n)
    case (Suc m)
    thus ?case
      by (metis (no_types) O_assoc inf_sup_ord(3) le_iff_sup relcomp_distrib2 relpow.simps(2)
        relpow_commute sub subsetCE)
  qed auto
qed

lemma f_chain_in_rtrancl:
  assumes m_le_n: m ≤ n and f_chain: ∀ i ∈ {m..<n}. (f i, f (Suc i)) ∈ R
  shows (f m, f n) ∈ R^*
proof (rule relpow_imp_rtrancl, rule relpow_fun_conv[THEN iffD2], intro exI conjI)
  let ?g = λi. f (m + i)
  let ?k = n − m

  show ?g 0 = f m
    by simp
  show ?g ?k = f n
    using m_le_n by force
  show (∀ i < ?k. (?g i, ?g (Suc i)) ∈ R)
    by (simp add: f_chain)
qed

lemma f_rev_chain_in_rtrancl:
  assumes m_le_n: m ≤ n and f_chain: ∀ i ∈ {m..<n}. (f (Suc i), f i) ∈ R
  shows (f n, f m) ∈ R^*
  by (rule f_chain_in_rtrancl[OF m_le_n, of λi. f (n + m − i), simplified])
    (metis f_chain le_add_diff Suc_diff_Suc Suc_leI atLeastLessThan_iff diff_Suc_diff_eq1 diff_less
      le_add1 less_le_trans zero_less_Suc)
```

2.7 Well-Founded Relations

```
lemma wf_app: wf r ⟹ wf {(x, y). (f x, f y) ∈ r}
  unfolding wf_eq_minimal by (intro allI, drule spec[of _ f ` Q for Q]) fast

lemma wfP_app: wfP p ⟹ wfP (λx y. p (f x) (f y))
  unfolding wfP_def by (rule wf_app[of {(x, y). p x y} f, simplified])

lemma wf_exists_minimal:
  assumes wf: wf r and Q: Q x
  shows ∃ x. Q x ∧ (∀ y. (f y, f x) ∈ r ⟶ ¬ Q y)
```

using *wf_eq_minimal*[*THEN iffD1*, *OF wf_app*[*OF wf*], *rule_format*, *of* _ {*x*. *Q x*}, *simplified*, *OF Q*]
by *blast*

lemma *wfP_exists_minimal*:
assumes *wf*: *wfP p* **and** *Q*: *Q x*
shows ∃ *x*. *Q x* ∧ (∀ *y*. *p* (*f y*) (*f x*) ⟶ ¬ *Q y*)
by (*rule wf_exists_minimal*[*of* {(*x*, *y*). *p x y*} *Q x*, *OF wf*[*unfolded wfP_def*] *Q*, *simplified*])

lemma *finite_irrefl_trans_imp_wf*: *finite r* ⟹ *irrefl r* ⟹ *trans r* ⟹ *wf r*
by (*erule finite_acyclic_wf*) (*simp add*: *acyclic_irrefl*)

lemma *finite_irreflp_transp_imp_wfp*:
finite {(*x*, *y*). *p x y*} ⟹ *irreflp p* ⟹ *transp p* ⟹ *wfP p*
using *finite_irrefl_trans_imp_wf*[*of* {(*x*, *y*). *p x y*}]
unfolding *wfP_def transp_def irreflp_def trans_def irrefl_def mem_Collect_eq prod.case*
by *assumption*

lemma *wf_infinite_down_chain_compatible*:
assumes
wf_R: *wf R* **and**
inf_chain_RS: ∀ *i*. (*f* (*Suc i*), *f i*) ∈ *R* ∪ *S* **and**
O_subset: *R O S* ⊆ *R*
shows ∃ *k*. ∀ *i*. (*f* (*Suc* (*i* + *k*)), *f* (*i* + *k*)) ∈ *S*
proof (*rule ccontr*)
assume ∄ *k*. ∀ *i*. (*f* (*Suc* (*i* + *k*)), *f* (*i* + *k*)) ∈ *S*
hence ∀ *k*. ∃ *i*. (*f* (*Suc* (*i* + *k*)), *f* (*i* + *k*)) ∉ *S*
by *blast*
hence ∀ *k*. ∃ *i* > *k*. (*f* (*Suc i*), *f i*) ∉ *S*
by (*metis add.commute add_Suc less_add_Suc1*)
hence ∀ *k*. ∃ *i* > *k*. (*f* (*Suc i*), *f i*) ∈ *R*
using *inf_chain_RS* **by** *blast*
hence ∃ *i* > *k*. (*f* (*Suc i*), *f i*) ∈ *R* ∧ (∀ *j* > *k*. (*f* (*Suc j*), *f j*) ∈ *R* ⟶ *j* ≥ *i*) **for** *k*
using *wf_eq_minimal*[*THEN iffD1*, *OF wf_less*, *rule_format*,
of _ {*i*. *i* > *k* ∧ (*f* (*Suc i*), *f i*) ∈ *R*}, *simplified*]
by (*meson not_less*)
then obtain *j_of* **where**
j_of_gt: ⋀*k*. *j_of k* > *k* **and**
j_of_in_R: ⋀*k*. (*f* (*Suc* (*j_of k*)), *f* (*j_of k*)) ∈ *R* **and**
j_of_min: ⋀*k*. ∀ *j* > *k*. (*f* (*Suc j*), *f j*) ∈ *R* ⟶ *j* ≥ *j_of k*
by *moura*

have *j_of_min_s*: ⋀*k j*. *j* > *k* ⟹ *j* < *j_of k* ⟹ (*f* (*Suc j*), *f j*) ∈ *S*
using *j_of_min inf_chain_RS* **by** *fastforce*

define *g* :: *nat* ⇒ *'a* **where** ⋀*k*. *g k* = *f* (*Suc* ((*j_of* ^^ *k*) *0*))

have *between_g*[*simplified*]: (*f* ((*j_of* ^^ (*Suc i*)) *0*), *f* (*Suc* ((*j_of* ^^ *i*) *0*))) ∈ *S** **for** *i*
proof (*rule f_rev_chain_in_rtrancl*; *clarify?*)
show *Suc* ((*j_of* ^^ *i*) *0*) ≤ (*j_of* ^^ *Suc i*) *0*
using *j_of_gt* **by** (*simp add*: *Suc_leI*)
next
fix *ia*
assume *ia*: *ia* ∈ {*Suc* ((*j_of* ^^ *i*) *0*)..<(*j_of* ^^ *Suc i*) *0*}
have *ia_gt*: *ia* > (*j_of* ^^ *i*) *0*
using *ia* **by** *auto*
have *ia_lt*: *ia* < *j_of* ((*j_of* ^^ *i*) *0*)
using *ia* **by** *auto*
show (*f* (*Suc ia*), *f ia*) ∈ *S*
by (*rule j_of_min_s*[*OF ia_gt ia_lt*])
qed

have ⋀*i*. (*g* (*Suc i*), *g i*) ∈ *R*
unfolding *g_def funpow.simps comp_def*

by (*rule subsetD*[*OF relcomp_subset_left_imp_relcomp_trancl_subset_left*[*OF O_subset*]])
(*rule relcompI*[*OF j_of_in_R between_g*])
moreover have $\forall f.\ \exists i.\ (f\ (Suc\ i),\ f\ i) \notin R$
using *wf_R*[*unfolded wf_iff_no_infinite_down_chain*] **by** *blast*
ultimately show *False*
by *blast*
qed

2.8 Wellorders

lemma (**in** *wellorder*) *exists_minimal*:
fixes $x :: 'a$
assumes $P\ x$
shows $\exists x.\ P\ x \wedge (\forall y.\ P\ y \longrightarrow y \geq x)$
using *assms* **by** (*auto intro*: *LeastI Least_le*)

2.9 Lists

lemma *rev_induct2*[*consumes 1*, *case_names Nil snoc*]:
$length\ xs = length\ ys \Longrightarrow P\ []\ [] \Longrightarrow$
$(\bigwedge x\ xs\ y\ ys.\ length\ xs = length\ ys \Longrightarrow P\ xs\ ys \Longrightarrow P\ (xs\ @\ [x])\ (ys\ @\ [y])) \Longrightarrow P\ xs\ ys$
proof (*induct xs arbitrary*: *ys rule*: *rev_induct*)
case (*snoc x xs ys*)
thus *?case*
by (*induct ys rule*: *rev_induct*) *simp_all*
qed *auto*

lemma *hd_in_set*: $length\ xs \neq 0 \Longrightarrow hd\ xs \in set\ xs$
by (*cases xs*) *auto*

lemma *in_lists_iff_set*: $xs \in lists\ A \longleftrightarrow set\ xs \subseteq A$
by *fast*

lemma *butlast_append_Cons*[*simp*]: $butlast\ (xs\ @\ y\ \#\ ys) = xs\ @\ butlast\ (y\ \#\ ys)$
using *butlast_append*[*of xs y # ys*, *simplified*] **by** *simp*

lemma *rev_in_lists*[*simp*]: $rev\ xs \in lists\ A \longleftrightarrow xs \in lists\ A$
by *auto*

lemma *hd_le_sum_list*:
fixes *xs* :: *'a*::*ordered_ab_semigroup_monoid_add_imp_le list*
assumes $xs \neq []$ **and** $\forall i < length\ xs.\ xs\ !\ i \geq 0$
shows $hd\ xs \leq sum\_list\ xs$
using *assms*
by (*induct xs rule*: *rev_induct*, *simp_all*,
metis add_cancel_right_left add_increasing2 hd_append2 lessI less_SucI list.sel(*1*) *nth_append*
nth_append_length order_refl self_append_conv2 sum_list.Nil)

lemma *sum_list_ge_length_times*:
fixes *a* :: *'a*::{*ordered_ab_semigroup_add*,*semiring_1*}
assumes $\forall i < length\ xs.\ xs\ !\ i \geq a$
shows $sum\_list\ xs \geq of\_nat\ (length\ xs) * a$
using *assms*
proof (*induct xs*)
case (*Cons x xs*)
note *ih* = *this*(*1*) **and** *xxs_i_ge_a* = *this*(*2*)

have *xs_i_ge_a*: $\forall i < length\ xs.\ xs\ !\ i \geq a$
using *xxs_i_ge_a* **by** *auto*

have $x \geq a$
using *xxs_i_ge_a* **by** *auto*
thus *?case*
using *ih*[*OF xs_i_ge_a*] **by** (*simp add*: *ring_distribs ordered_ab_semigroup_add_class.add_mono*)

qed *auto*

lemma *prod_list_nonneg*:
fixes *xs* :: ('*a* :: {*ordered_semiring_0*,*linordered_nonzero_semiring*}) *list*
assumes ⋀*x*. *x* ∈ *set xs* ⟹ *x* ≥ *0*
shows *prod_list xs* ≥ *0*
using *assms* **by** (*induct xs*) *auto*

lemma *zip_append_0_upt*:
zip (*xs* @ *ys*) [*0*..<*length xs* + *length ys*] =
zip xs [*0*..<*length xs*] @ *zip ys* [*length xs*..<*length xs* + *length ys*]
proof (*induct ys arbitrary*: *xs*)
case (*Cons y ys*)
note *ih* = *this*
show *?case*
using *ih*[*of xs* @ [*y*]] **by** (*simp*, *cases ys*, *simp*, *simp add*: *upt_rec*)
qed *auto*

lemma *zip_eq_butlast_last*:
assumes *len_gt0*: *length xs* > *0* **and** *len_eq*: *length xs* = *length ys*
shows *zip xs ys* = *zip* (*butlast xs*) (*butlast ys*) @ [(*last xs*, *last ys*)]
using *len_eq len_gt0* **by** (*induct rule*: *list_induct2*) *auto*

2.10 Extended Natural Numbers

lemma *the_enat_0*[*simp*]: *the_enat 0* = *0*
by (*simp add*: *zero_enat_def*)

lemma *the_enat_1*[*simp*]: *the_enat 1* = *1*
by (*simp add*: *one_enat_def*)

lemma *enat_le_minus_1_imp_lt*: *m* ≤ *n* − *1* ⟹ *n* ≠ ∞ ⟹ *n* ≠ *0* ⟹ *m* < *n* **for** *m n* :: *enat*
by (*cases m*; *cases n*; *simp add*: *zero_enat_def one_enat_def*)

lemma *enat_diff_diff_eq*: *k* − *m* − *n* = *k* − (*m* + *n*) **for** *k m n* :: *enat*
by (*cases k*; *cases m*; *cases n*) *auto*

lemma *enat_sub_add_same*[*intro*]: *n* ≤ *m* ⟹ *m* = *m* − *n* + *n* **for** *m n* :: *enat*
by (*cases m*; *cases n*) *auto*

lemma *enat_the_enat_iden*[*simp*]: *n* ≠ ∞ ⟹ *enat* (*the_enat n*) = *n*
by *auto*

lemma *the_enat_minus_nat*: *m* ≠ ∞ ⟹ *the_enat* (*m* − *enat n*) = *the_enat m* − *n*
by *auto*

lemma *enat_the_enat_le*: *enat* (*the_enat x*) ≤ *x*
by (*cases x*; *simp*)

lemma *enat_the_enat_minus_le*: *enat* (*the_enat* (*x* − *y*)) ≤ *x*
by (*cases x*; *cases y*; *simp*)

lemma *enat_le_imp_minus_le*: *k* ≤ *m* ⟹ *k* − *n* ≤ *m* **for** *k m n* :: *enat*
by (*metis Groups.add_ac*(*2*) *enat_diff_diff_eq enat_ord_simps*(*3*) *enat_sub_add_same*
enat_the_enat_iden enat_the_enat_minus_le idiff_0_right idiff_infinity idiff_infinity_right
order_trans_rules(*23*) *plus_enat_simps*(*3*))

lemma *add_diff_assoc2_enat*: *m* ≥ *n* ⟹ *m* − *n* + *p* = *m* + *p* − *n* **for** *m n p* :: *enat*
by (*cases m*; *cases n*; *cases p*; *auto*)

lemma *enat_mult_minus_distrib*: *enat x* ∗ (*y* − *z*) = *enat x* ∗ *y* − *enat x* ∗ *z*
by (*cases y*; *cases z*; *auto simp*: *enat_0 right_diff_distrib'*)

2.11 Multisets

lemma *add_mset_lt_left_lt*: $a < b \Longrightarrow$ *add_mset a A* $<$ *add_mset b A*
unfolding *less_multiset*$_{HO}$ **by** *auto*

lemma *add_mset_le_left_le*: $a \leq b \Longrightarrow$ *add_mset a A* $\leq$ *add_mset b A* **for** $a :: 'a ::$ *linorder*
unfolding *less_multiset*$_{HO}$ **by** *auto*

lemma *add_mset_lt_right_lt*: $A < B \Longrightarrow$ *add_mset a A* $<$ *add_mset a B*
unfolding *less_multiset*$_{HO}$ **by** *auto*

lemma *add_mset_le_right_le*: $A \leq B \Longrightarrow$ *add_mset a A* $\leq$ *add_mset a B*
unfolding *less_multiset*$_{HO}$ **by** *auto*

lemma *add_mset_lt_lt_lt*:
assumes *a_lt_b*: $a < b$ **and** *A_le_B*: $A < B$
shows *add_mset a A* $<$ *add_mset b B*
by (*rule less_trans*[*OF add_mset_lt_left_lt*[*OF a_lt_b*] *add_mset_lt_right_lt*[*OF A_le_B*]])

lemma *add_mset_lt_lt_le*: $a < b \Longrightarrow A \leq B \Longrightarrow$ *add_mset a A* $<$ *add_mset b B*
using *add_mset_lt_lt_lt le_neq_trans* **by** *fastforce*

lemma *add_mset_lt_le_lt*: $a \leq b \Longrightarrow A < B \Longrightarrow$ *add_mset a A* $<$ *add_mset b B* **for** $a :: 'a ::$ *linorder*
using *add_mset_lt_lt_lt* **by** (*metis add_mset_lt_right_lt le_less*)

lemma *add_mset_le_le_le*:
fixes $a :: 'a ::$ *linorder*
assumes *a_le_b*: $a \leq b$ **and** *A_le_B*: $A \leq B$
shows *add_mset a A* $\leq$ *add_mset b B*
by (*rule order.trans*[*OF add_mset_le_left_le*[*OF a_le_b*] *add_mset_le_right_le*[*OF A_le_B*]])

declare *filter_eq_replicate_mset* [*simp*] *image_mset_subseteq_mono* [*intro*]

lemma *nonempty_subseteq_mset_eq_singleton*: $M \neq \{\#\} \Longrightarrow M \subseteq\# \{\#x\#\} \Longrightarrow M = \{\#x\#\}$
by (*cases M*) (*auto dest*: *subset_mset.diff_add*)

lemma *nonempty_subseteq_mset_iff_singleton*: $(M \neq \{\#\} \wedge M \subseteq\# \{\#x\#\} \wedge P) \longleftrightarrow M = \{\#x\#\} \wedge P$
by (*cases M*) (*auto dest*: *subset_mset.diff_add*)

lemma *count_gt_imp_in_mset*[*intro*]: *count M x* $> n \Longrightarrow x \in\# M$
using *count_greater_zero_iff* **by** *fastforce*

lemma *size_lt_imp_ex_count_lt*: *size M* $<$ *size N* $\Longrightarrow \exists x \in\# N.$ *count M x* $<$ *count N x*
by (*metis count_eq_zero_iff leD not_le_imp_less not_less_zero size_mset_mono subseteq_mset_def*)

lemma *filter_filter_mset*[*simp*]: $\{\#x \in\# \{\#x \in\# M.\ Q\ x\#\}.\ P\ x\#\} = \{\#x \in\# M.\ P\ x \wedge Q\ x\#\}$
by (*induct M*) *auto*

lemma *size_filter_unsat_elem*:
assumes $x \in\# M$ **and** $\neg\ P\ x$
shows *size* $\{\#x \in\# M.\ P\ x\#\} <$ *size M*
proof −
have *size* (*filter_mset P M*) $\neq$ *size M*
using *assms* **by** (*metis add.right_neutral add_diff_cancel_left' count_filter_mset mem_Collect_eq multiset_partition nonempty_has_size set_mset_def size_union*)
then show *?thesis*
by (*meson leD nat_neq_iff size_filter_mset_lesseq*)
qed

lemma *size_filter_ne_elem*: $x \in\# M \Longrightarrow$ *size* $\{\#y \in\# M.\ y \neq x\#\} <$ *size M*
by (*simp add*: *size_filter_unsat_elem*[*of x M* $\lambda y.\ y \neq x$])

lemma *size_eq_ex_count_lt*:
assumes

```
    sz_m_eq_n: size M = size N and
    m_eq_n: M ≠ N
  shows ∃x. count M x < count N x
proof −
  obtain x where count M x ≠ count N x
    using m_eq_n by (meson multiset_eqI)
  moreover
  {
    assume count M x < count N x
    hence ?thesis
      by blast
  }
  moreover
  {
    assume cnt_x: count M x > count N x

    have size {#y ∈# M. y = x#} + size {#y ∈# M. y ≠ x#} =
      size {#y ∈# N. y = x#} + size {#y ∈# N. y ≠ x#}
      using sz_m_eq_n multiset_partition by (metis size_union)
    hence sz_m_minus_x: size {#y ∈# M. y ≠ x#} < size {#y ∈# N. y ≠ x#}
      using cnt_x by simp
    then obtain y where count {#y ∈# M. y ≠ x#} y < count {#y ∈# N. y ≠ x#} y
      using size_lt_imp_ex_count_lt[OF sz_m_minus_x] by blast
    hence count M y < count N y
      by (metis count_filter_mset less_asym)
    hence ?thesis
      by blast
  }
  ultimately show ?thesis
    by fastforce
qed

lemma count_image_mset_lt_imp_lt_raw:
  assumes
    finite A and
    A = set_mset M ∪ set_mset N and
    count (image_mset f M) b < count (image_mset f N) b
  shows ∃x. f x = b ∧ count M x < count N x
  using assms
proof (induct A arbitrary: M N b rule: finite_induct)
  case (insert x F)
  note fin = this(1) and x_ni_f = this(2) and ih = this(3) and x_f_eq_m_n = this(4) and
    cnt_b = this(5)

  let ?Ma = {#y ∈# M. y ≠ x#}
  let ?Mb = {#y ∈# M. y = x#}
  let ?Na = {#y ∈# N. y ≠ x#}
  let ?Nb = {#y ∈# N. y = x#}

  have m_part: M = ?Mb + ?Ma and n_part: N = ?Nb + ?Na
    using multiset_partition by blast+

  have f_eq_ma_na: F = set_mset ?Ma ∪ set_mset ?Na
    using x_f_eq_m_n x_ni_f by auto

  show ?case
  proof (cases count (image_mset f ?Ma) b < count (image_mset f ?Na) b)
    case cnt_ba: True
    obtain xa where f xa = b and count ?Ma xa < count ?Na xa
      using ih[OF f_eq_ma_na cnt_ba] by blast
    thus ?thesis
      by (metis count_filter_mset not_less0)
  next
```

```
    case cnt_ba: False
    have fx_eq_b: f x = b
      using cnt_b cnt_ba by (subst (asm) m_part, subst (asm) n_part, auto, presburger)
    moreover have count M x < count N x
      using cnt_b cnt_ba by (subst (asm) m_part, subst (asm) n_part, auto simp: fx_eq_b)
    ultimately show ?thesis
      by blast
  qed
qed auto

lemma count_image_mset_lt_imp_lt:
  assumes cnt_b: count (image_mset f M) b < count (image_mset f N) b
  shows ∃ x. f x = b ∧ count M x < count N x
  by (rule count_image_mset_lt_imp_lt_raw[of set_mset M ∪ set_mset N, OF _ refl cnt_b]) auto

lemma count_image_mset_le_imp_lt_raw:
  assumes
    finite A and
    A = set_mset M ∪ set_mset N and
    count (image_mset f M) (f a) + count N a < count (image_mset f N) (f a) + count M a
  shows ∃ b. f b = f a ∧ count M b < count N b
  using assms
proof (induct A arbitrary: M N rule: finite_induct)
  case (insert x F)
  note fin = this(1) and x_ni_f = this(2) and ih = this(3) and x_f_eq_m_n = this(4) and
    cnt_lt = this(5)

  let ?Ma = {#y ∈# M. y ≠ x#}
  let ?Mb = {#y ∈# M. y = x#}
  let ?Na = {#y ∈# N. y ≠ x#}
  let ?Nb = {#y ∈# N. y = x#}

  have m_part: M = ?Mb + ?Ma and n_part: N = ?Nb + ?Na
    using multiset_partition by blast+

  have f_eq_ma_na: F = set_mset ?Ma ∪ set_mset ?Na
    using x_f_eq_m_n x_ni_f by auto

  show ?case
  proof (cases f x = f a)
    case fx_ne_fa: False

    have cnt_fma_fa: count (image_mset f ?Ma) (f a) = count (image_mset f M) (f a)
      using fx_ne_fa by (subst (2) m_part) auto
    have cnt_fna_fa: count (image_mset f ?Na) (f a) = count (image_mset f N) (f a)
      using fx_ne_fa by (subst (2) n_part) auto
    have cnt_ma_a: count ?Ma a = count M a
      using fx_ne_fa by (subst (2) m_part) auto
    have cnt_na_a: count ?Na a = count N a
      using fx_ne_fa by (subst (2) n_part) auto

    obtain b where fb_eq_fa: f b = f a and cnt_b: count ?Ma b < count ?Na b
      using ih[OF f_eq_ma_na] cnt_lt unfolding cnt_fma_fa cnt_fna_fa cnt_ma_a cnt_na_a by blast
    have fx_ne_fb: f x ≠ f b
      using fb_eq_fa fx_ne_fa by simp

    have cnt_ma_b: count ?Ma b = count M b
      using fx_ne_fb by (subst (2) m_part) auto
    have cnt_na_b: count ?Na b = count N b
      using fx_ne_fb by (subst (2) n_part) auto

    show ?thesis
      using fb_eq_fa cnt_b unfolding cnt_ma_b cnt_na_b by blast
```

```
  next
    case fx_eq_fa: True
    show ?thesis
    proof (cases x = a)
      case x_eq_a: True
      have count (image_mset f ?Ma) (f a) + count ?Na a
        < count (image_mset f ?Na) (f a) + count ?Ma a
        using cnt_lt x_eq_a by (subst (asm) (1 2) m_part, subst (asm) (1 2) n_part, auto)
      thus ?thesis
        using ih[OF f_eq_ma_na] by (metis count_filter_mset nat_neq_iff)
    next
      case x_ne_a: False
      show ?thesis
      proof (cases count M x < count N x)
        case True
        thus ?thesis
          using fx_eq_fa by blast
     next
        case False
        hence cnt_x: count M x ≥ count N x
          by fastforce
        have count M x + count (image_mset f ?Ma) (f a) + count ?Na a
          < count N x + count (image_mset f ?Na) (f a) + count ?Ma a
          using cnt_lt x_ne_a fx_eq_fa by (subst (asm) (1 2) m_part, subst (asm) (1 2) n_part, auto)
        hence count (image_mset f ?Ma) (f a) + count ?Na a
          < count (image_mset f ?Na) (f a) + count ?Ma a
          using cnt_x by linarith
        thus ?thesis
          using ih[OF f_eq_ma_na] by (metis count_filter_mset nat_neq_iff)
      qed
    qed
  qed
qed auto

lemma count_image_mset_le_imp_lt:
  assumes
    count (image_mset f M) (f a) ≤ count (image_mset f N) (f a) and
    count M a > count N a
  shows ∃ b. f b = f a ∧ count M b < count N b
  using assms by (auto intro: count_image_mset_le_imp_lt_raw[of set_mset M ∪ set_mset N])

lemma Max_in_mset: M ≠ {#} ⟹ Max_mset M ∈# M
  by simp

lemma Max_lt_imp_lt_mset:
  assumes n_nemp: N ≠ {#} and max: Max_mset M < Max_mset N (is ?max_M < ?max_N)
  shows M < N
proof (cases M = {#})
  case m_nemp: False

  have max_n_in_n: ?max_N ∈# N
    using n_nemp by simp
  have max_n_nin_m: ?max_N ∉# M
    using max Max_ge leD by auto

  have M ≠ N
    using max by auto
  moreover
  {
    fix y
    assume count N y < count M y
    hence y ∈# M
      by blast
```

hence *?max_M* $\geq$ *y*
by *simp*
hence *?max_N* $>$ *y*
using *max* **by** *auto*
hence $\exists x > y.$ *count M x* $<$ *count N x*
using *max_n_nin_m max_n_in_n* **by** *fastforce*
}
ultimately show *?thesis*
unfolding *less_multiset*$_{HO}$ **by** *blast*
qed (*auto simp*: *n_nemp*)

lemma *fold_mset_singleton*[*simp*]: *fold_mset f z* {#*x*#} = *f x z*
by (*simp add*: *fold_mset_def*)

end

3 Lambda-Free Higher-Order Terms

theory *Lambda_Free_Term*
imports *Lambda_Free_Util*
abbrevs $>s = >_s$
and $>h = >_{hd}$
and $\leq\geq h = \leq\geq_{hd}$
begin

This theory defines λ-free higher-order terms and related locales.

3.1 Precedence on Symbols

locale *gt_sym* =
fixes
gt_sym :: $'s \Rightarrow 's \Rightarrow$ *bool* (**infix** $>_s$ *50*)
assumes
gt_sym_irrefl: $\neg$ *f* $>_s$ *f* **and**
gt_sym_trans: *h* $>_s$ *g* $\Longrightarrow$ *g* $>_s$ *f* $\Longrightarrow$ *h* $>_s$ *f* **and**
gt_sym_total: *f* $>_s$ *g* $\vee$ *g* $>_s$ *f* $\vee$ *g* = *f* **and**
gt_sym_wf: *wfP* (λ*f g*. *g* $>_s$ *f*)
begin

lemma *gt_sym_antisym*: *f* $>_s$ *g* $\Longrightarrow$ $\neg$ *g* $>_s$ *f*
by (*metis gt_sym_irrefl gt_sym_trans*)

end

3.2 Heads

datatype (*plugins del*: *size*) (*syms_hd*: $'s$, *vars_hd*: $'v$) *hd* =
is_Var: *Var* (*var*: $'v$)
| *Sym* (*sym*: $'s$)

abbreviation *is_Sym* :: ($'s$, $'v$) *hd* $\Rightarrow$ *bool* **where**
is_Sym $\zeta \equiv \neg$ *is_Var* ζ

lemma *finite_vars_hd*[*simp*]: *finite* (*vars_hd* ζ)
by (*cases* ζ) *auto*

lemma *finite_syms_hd*[*simp*]: *finite* (*syms_hd* ζ)
by (*cases* ζ) *auto*

3.3 Terms

consts *head0* :: $'a$

datatype (*syms*: *'s*, *vars*: *'v*) *tm* =
  is_Hd: *Hd* (*head*: (*'s*, *'v*) *hd*)
| *App* (*fun*: (*'s*, *'v*) *tm*) (*arg*: (*'s*, *'v*) *tm*)
where
  head (*App s _*) = *head0 s*
| *fun* (*Hd ζ*) = *Hd ζ*
| *arg* (*Hd ζ*) = *Hd ζ*

overloading *head0* ≡ *head0* :: (*'s*, *'v*) *tm* ⇒ (*'s*, *'v*) *hd*
begin

primrec *head0* :: (*'s*, *'v*) *tm* ⇒ (*'s*, *'v*) *hd* **where**
  head0 (*Hd ζ*) = *ζ*
| *head0* (*App s _*) = *head0 s*

end

lemma *head_App*[*simp*]: *head* (*App s t*) = *head s*
  by (*cases s*) *auto*

declare *tm.sel*(*2*)[*simp del*]

lemma *head_fun*[*simp*]: *head* (*fun s*) = *head s*
  by (*cases s*) *auto*

abbreviation *ground* :: (*'s*, *'v*) *tm* ⇒ *bool* **where**
  ground t ≡ *vars t* = {}

abbreviation *is_App* :: (*'s*, *'v*) *tm* ⇒ *bool* **where**
  is_App s ≡ ¬ *is_Hd s*

lemma
  size_fun_lt: *is_App s* ⟹ *size* (*fun s*) < *size s* **and**
  size_arg_lt: *is_App s* ⟹ *size* (*arg s*) < *size s*
  by (*cases s*; *simp*)+

lemma
  finite_vars[*simp*]: *finite* (*vars s*) **and**
  finite_syms[*simp*]: *finite* (*syms s*)
  by (*induct s*) *auto*

lemma
  vars_head_subseteq: *vars_hd* (*head s*) ⊆ *vars s* **and**
  syms_head_subseteq: *syms_hd* (*head s*) ⊆ *syms s*
  by (*induct s*) *auto*

fun *args* :: (*'s*, *'v*) *tm* ⇒ (*'s*, *'v*) *tm list* **where**
  args (*Hd _*) = []
| *args* (*App s t*) = *args s* @ [*t*]

lemma *set_args_fun*: *set* (*args* (*fun s*)) ⊆ *set* (*args s*)
  by (*cases s*) *auto*

lemma *arg_in_args*: *is_App s* ⟹ *arg s* ∈ *set* (*args s*)
  by (*cases s rule*: *tm.exhaust*) *auto*

lemma
  vars_args_subseteq: *si* ∈ *set* (*args s*) ⟹ *vars si* ⊆ *vars s* **and**
  syms_args_subseteq: *si* ∈ *set* (*args s*) ⟹ *syms si* ⊆ *syms s*
  by (*induct s*) *auto*

lemma *args_Nil_iff_is_Hd*: *args s* = [] ⟷ *is_Hd s*
  by (*cases s*) *auto*

abbreviation *num_args* :: ('*s*, '*v*) *tm* ⇒ *nat* **where**
num_args s ≡ *length* (*args s*)

lemma *size_ge_num_args*: *size s* ≥ *num_args s*
by (*induct s*) *auto*

lemma *Hd_head_id*: *num_args s* = *0* ⟹ *Hd* (*head s*) = *s*
by (*metis args.cases args.simps*(*2*) *length_0_conv snoc_eq_iff_butlast tm.collapse*(*1*) *tm.disc*(*1*))

lemma *one_arg_imp_Hd*: *num_args s* = *1* ⟹ *s* = *App t u* ⟹ *t* = *Hd* (*head t*)
by (*simp add*: *Hd_head_id*)

lemma *size_in_args*: *s* ∈ *set* (*args t*) ⟹ *size s* < *size t*
by (*induct t*) *auto*

primrec *apps* :: ('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm list* ⇒ ('*s*, '*v*) *tm* **where**
apps s [] = *s*
| *apps s* (*t* # *ts*) = *apps* (*App s t*) *ts*

lemma
vars_apps[*simp*]: *vars* (*apps s ss*) = *vars s* ∪ (⋃ *s* ∈ *set ss*. *vars s*) **and**
syms_apps[*simp*]: *syms* (*apps s ss*) = *syms s* ∪ (⋃ *s* ∈ *set ss*. *syms s*) **and**
head_apps[*simp*]: *head* (*apps s ss*) = *head s* **and**
args_apps[*simp*]: *args* (*apps s ss*) = *args s* @ *ss* **and**
is_App_apps[*simp*]: *is_App* (*apps s ss*) ⟷ *args* (*apps s ss*) ≠ [] **and**
fun_apps_Nil[*simp*]: *fun* (*apps s* []) = *fun s* **and**
fun_apps_Cons[*simp*]: *fun* (*apps* (*App s sa*) *ss*) = *apps s* (*butlast* (*sa* # *ss*)) **and**
arg_apps_Nil[*simp*]: *arg* (*apps s* []) = *arg s* **and**
arg_apps_Cons[*simp*]: *arg* (*apps* (*App s sa*) *ss*) = *last* (*sa* # *ss*)
by (*induct ss arbitrary*: *s sa*) (*auto simp*: *args_Nil_iff_is_Hd*)

lemma *apps_append*[*simp*]: *apps s* (*ss* @ *ts*) = *apps* (*apps s ss*) *ts*
by (*induct ss arbitrary*: *s ts*) *auto*

lemma *App_apps*: *App* (*apps s ts*) *t* = *apps s* (*ts* @ [*t*])
by *simp*

lemma *tm_inject_apps*[*iff*, *induct_simp*]: *apps* (*Hd* ζ) *ss* = *apps* (*Hd* ξ) *ts* ⟷ ζ = ξ ∧ *ss* = *ts*
by (*metis args_apps head_apps same_append_eq tm.sel*(*1*))

lemma *tm_collapse_apps*[*simp*]: *apps* (*Hd* (*head s*)) (*args s*) = *s*
by (*induct s*) *auto*

lemma *tm_expand_apps*: *head s* = *head t* ⟹ *args s* = *args t* ⟹ *s* = *t*
by (*metis tm_collapse_apps*)

lemma *tm_exhaust_apps_sel*[*case_names apps*]: (*s* = *apps* (*Hd* (*head s*)) (*args s*) ⟹ *P*) ⟹ *P*
by (*atomize_elim*, *induct s*) *auto*

lemma *tm_exhaust_apps*[*case_names apps*]: (⋀ζ *ss*. *s* = *apps* (*Hd* ζ) *ss* ⟹ *P*) ⟹ *P*
by (*metis tm_collapse_apps*)

lemma *tm_induct_apps*[*case_names apps*]:
assumes ⋀ζ *ss*. (⋀*s*. *s* ∈ *set ss* ⟹ *P s*) ⟹ *P* (*apps* (*Hd* ζ) *ss*)
shows *P s*
using *assms*
by (*induct s taking*: *size rule*: *measure_induct_rule*) (*metis size_in_args tm_collapse_apps*)

lemma
ground_fun: *ground s* ⟹ *ground* (*fun s*) **and**
ground_arg: *ground s* ⟹ *ground* (*arg s*)
by (*induct s*) *auto*

lemma *ground_head*: *ground s* $\Longrightarrow$ *is_Sym* (*head s*)
by (*cases s rule*: *tm_exhaust_apps*) (*auto simp*: *is_Var_def*)

lemma *ground_args*: *t* $\in$ *set* (*args s*) $\Longrightarrow$ *ground s* $\Longrightarrow$ *ground t*
by (*induct s rule*: *tm_induct_apps*) *auto*

primrec *vars_mset* :: ('*s*, '*v*) *tm* $\Rightarrow$ '*v multiset* **where**
vars_mset (*Hd* ζ) = *mset_set* (*vars_hd* ζ)
| *vars_mset* (*App s t*) = *vars_mset s* + *vars_mset t*

lemma *set_vars_mset*[*simp*]: *set_mset* (*vars_mset t*) = *vars t*
by (*induct t*) *auto*

lemma *vars_mset_empty_iff*[*iff*]: *vars_mset s* = {#} $\longleftrightarrow$ *ground s*
by (*induct s*) (*auto simp*: *mset_set_empty_iff*)

lemma *vars_mset_fun*[*intro*]: *vars_mset* (*fun t*) $\subseteq\#$ *vars_mset t*
by (*cases t*) *auto*

lemma *vars_mset_arg*[*intro*]: *vars_mset* (*arg t*) $\subseteq\#$ *vars_mset t*
by (*cases t*) *auto*

3.4 hsize

The hsize of a term is the number of heads (Syms or Vars) in the term.

primrec *hsize* :: ('*s*, '*v*) *tm* $\Rightarrow$ *nat* **where**
hsize (*Hd* ζ) = *1*
| *hsize* (*App s t*) = *hsize s* + *hsize t*

lemma *hsize_size*: *hsize t* $*$ *2* = *size t* + *1*
by (*induct t*) *auto*

lemma *hsize_pos*[*simp*]: *hsize t* > *0*
by (*induction t*; *simp*)

lemma *hsize_fun_lt*: *is_App s* $\Longrightarrow$ *hsize* (*fun s*) < *hsize s*
by (*cases s*; *simp*)

lemma *hsize_arg_lt*: *is_App s* $\Longrightarrow$ *hsize* (*arg s*) < *hsize s*
by (*cases s*; *simp*)

lemma *hsize_ge_num_args*: *hsize s* $\geq$ *hsize s*
by (*induct s*) *auto*

lemma *hsize_in_args*: *s* $\in$ *set* (*args t*) $\Longrightarrow$ *hsize s* < *hsize t*
by (*induct t*) *auto*

lemma *hsize_apps*: *hsize* (*apps t ts*) = *hsize t* + *sum_list* (*map hsize ts*)
by (*induct ts arbitrary*:*t*; *simp*)

lemma *hsize_args*: *1* + *sum_list* (*map hsize* (*args t*)) = *hsize t*
by (*metis hsize.simps*(*1*) *hsize_apps tm_collapse_apps*)

3.5 Substitutions

primrec *subst* :: ('*v* $\Rightarrow$ ('*s*, '*v*) *tm*) $\Rightarrow$ ('*s*, '*v*) *tm* $\Rightarrow$ ('*s*, '*v*) *tm* **where**
subst ϱ (*Hd* ζ) = (*case* ζ *of Var x* $\Rightarrow$ ϱ *x* | *Sym f* $\Rightarrow$ *Hd* (*Sym f*))
| *subst* ϱ (*App s t*) = *App* (*subst* ϱ *s*) (*subst* ϱ *t*)

lemma *subst_apps*[*simp*]: *subst* ϱ (*apps s ts*) = *apps* (*subst* ϱ *s*) (*map* (*subst* ϱ) *ts*)
by (*induct ts arbitrary*: *s*) *auto*

lemma *head_subst*[*simp*]: *head* (*subst* ϱ *s*) = *head* (*subst* ϱ (*Hd* (*head s*)))
by (*cases s rule*: *tm_exhaust_apps*) (*auto split*: *hd.split*)

lemma *args_subst*[*simp*]:
args (*subst* ϱ *s*) = (*case head s of Var x* ⇒ *args* (ϱ *x*) | *Sym f* ⇒ []) @ *map* (*subst* ϱ) (*args s*)
by (*cases s rule*: *tm_exhaust_apps*) (*auto split*: *hd.split*)

lemma *ground_imp_subst_iden*: *ground s* ⟹ *subst* ϱ *s* = *s*
by (*induct s*) (*auto split*: *hd.split*)

lemma *vars_mset_subst*[*simp*]: *vars_mset* (*subst* ϱ *s*) = ($\sum_\#$ {#*vars_mset* (ϱ *x*). *x* ∈# *vars_mset s*#})
proof (*induct s*)
case (*Hd* ζ)
show *?case*
by (*cases* ζ) *auto*
qed *auto*

lemma *vars_mset_subst_subseteq*:
vars_mset t ⊇# *vars_mset s* ⟹ *vars_mset* (*subst* ϱ *t*) ⊇# *vars_mset* (*subst* ϱ *s*)
unfolding *vars_mset_subst*
by (*metis* (*no_types*) *add_diff_cancel_right′ diff_subset_eq_self image_mset_union sum_mset.union*
subset_mset.add_diff_inverse)

lemma *vars_subst_subseteq*: *vars t* ⊇ *vars s* ⟹ *vars* (*subst* ϱ *t*) ⊇ *vars* (*subst* ϱ *s*)
unfolding *set_vars_mset*[*symmetric*] *vars_mset_subst* **by** *auto*

3.6 Subterms

inductive *sub* :: ('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm* ⇒ *bool* **where**
sub_refl: *sub s s*
| *sub_fun*: *sub s t* ⟹ *sub s* (*App u t*)
| *sub_arg*: *sub s t* ⟹ *sub s* (*App t u*)

inductive-cases *sub_HdE*[*simplified*, *elim*]: *sub s* (*Hd* ξ)
inductive-cases *sub_AppE*[*simplified*, *elim*]: *sub s* (*App t u*)
inductive-cases *sub_Hd_HdE*[*simplified*, *elim*]: *sub* (*Hd* ζ) (*Hd* ξ)
inductive-cases *sub_Hd_AppE*[*simplified*, *elim*]: *sub* (*Hd* ζ) (*App t u*)

lemma *in_vars_imp_sub*: *x* ∈ *vars s* ⟷ *sub* (*Hd* (*Var x*)) *s*
by *induct* (*auto intro*: *sub.intros elim*: *hd.set_cases*(*2*))

lemma *sub_args*: *s* ∈ *set* (*args t*) ⟹ *sub s t*
by (*induct t*) (*auto intro*: *sub.intros*)

lemma *sub_size*: *sub s t* ⟹ *size s* ≤ *size t*
by *induct auto*

lemma *sub_subst*: *sub s t* ⟹ *sub* (*subst* ϱ *s*) (*subst* ϱ *t*)
proof (*induct t*)
case (*Hd* ζ)
thus *?case*
by (*cases* ζ; *blast intro*: *sub.intros*)
qed (*auto intro*: *sub.intros del*: *sub_AppE elim*!: *sub_AppE*)

abbreviation *proper_sub* :: ('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm* ⇒ *bool* **where**
proper_sub s t ≡ *sub s t* ∧ *s* ≠ *t*

lemma *proper_sub_Hd*[*simp*]: ¬ *proper_sub s* (*Hd* ζ)
using *sub.cases* **by** *blast*

lemma *proper_sub_subst*:
assumes *psub*: *proper_sub s t*
shows *proper_sub* (*subst* ϱ *s*) (*subst* ϱ *t*)
proof (*cases t*)

case *Hd*
thus *?thesis*
using *psub* **by** *simp*
next
case *t*: (*App t1 t2*)
have *sub s t1* ∨ *sub s t2*
using *t psub* **by** *blast*
hence *sub* (*subst ϱ s*) (*subst ϱ t1*) ∨ *sub* (*subst ϱ s*) (*subst ϱ t2*)
using *sub_subst* **by** *blast*
thus *?thesis*
unfolding *t* **by** (*auto intro*: *sub.intros dest*: *sub_size*)
qed

3.7 Maximum Arities

locale *arity* =
fixes
arity_sym :: *'s* ⇒ *enat* **and**
arity_var :: *'v* ⇒ *enat*
begin

primrec *arity_hd* :: (*'s*, *'v*) *hd* ⇒ *enat* **where**
arity_hd (*Var x*) = *arity_var x*
| *arity_hd* (*Sym f*) = *arity_sym f*

definition *arity* :: (*'s*, *'v*) *tm* ⇒ *enat* **where**
arity s = *arity_hd* (*head s*) − *num_args s*

lemma *arity_simps*[*simp*]:
arity (*Hd ζ*) = *arity_hd ζ*
arity (*App s t*) = *arity s* − *1*
by (*auto simp*: *arity_def enat_diff_diff_eq add.commute eSuc_enat plus_1_eSuc*(*1*))

lemma *arity_apps*[*simp*]: *arity* (*apps s ts*) = *arity s* − *length ts*
proof (*induct ts arbitrary*: *s*)
case (*Cons t ts*)
thus *?case*
by (*case_tac arity s*; *simp add*: *one_enat_def*)
qed *simp*

inductive *wary* :: (*'s*, *'v*) *tm* ⇒ *bool* **where**
wary_Hd[*intro*]: *wary* (*Hd ζ*)
| *wary_App*[*intro*]: *wary s* ⟹ *wary t* ⟹ *num_args s* < *arity_hd* (*head s*) ⟹ *wary* (*App s t*)

inductive-cases *wary_HdE*: *wary* (*Hd ζ*)
inductive-cases *wary_AppE*: *wary* (*App s t*)
inductive-cases *wary_binaryE*[*simplified*]: *wary* (*App* (*App s t*) *u*)

lemma *wary_fun*[*intro*]: *wary t* ⟹ *wary* (*fun t*)
by (*cases t*) (*auto elim*: *wary.cases*)

lemma *wary_arg*[*intro*]: *wary t* ⟹ *wary* (*arg t*)
by (*cases t*) (*auto elim*: *wary.cases*)

lemma *wary_args*: *s* ∈ *set* (*args t*) ⟹ *wary t* ⟹ *wary s*
by (*induct t arbitrary*: *s*, *simp*)
(*metis Un_iff args.simps*(*2*) *wary.cases insert_iff length_pos_if_in_set*
less_numeral_extra(*3*) *list.set*(*2*) *list.size*(*3*) *set_append tm.distinct*(*1*) *tm.inject*(*2*))

lemma *wary_sub*: *sub s t* ⟹ *wary t* ⟹ *wary s*
by (*induct rule*: *sub.induct*) (*auto elim*: *wary.cases*)

lemma *wary_inf_ary*: (⋀*ζ*. *arity_hd ζ* = ∞) ⟹ *wary s*
by *induct auto*

lemma *wary_num_args_le_arity_head*: *wary s* $\Longrightarrow$ *num_args s* $\leq$ *arity_hd* (*head s*)
by (*induct rule*: *wary.induct*) (*auto simp*: *zero_enat_def*[*symmetric*] *Suc_ile_eq*)

lemma *wary_apps*:
wary s $\Longrightarrow$ ($\bigwedge$*sa*. *sa* $\in$ *set ss* $\Longrightarrow$ *wary sa*) $\Longrightarrow$ *length ss* $\leq$ *arity s* $\Longrightarrow$ *wary* (*apps s ss*)
proof (*induct ss arbitrary*: *s*)
case (*Cons sa ss*)
note *ih* = *this*(*1*) **and** *wary_s* = *this*(*2*) **and** *wary_ss* = *this*(*3*) **and** *nargs_s_sa_ss* = *this*(*4*)
show *?case*
unfolding *apps.simps*
proof (*rule ih*)
have *wary sa*
using *wary_ss* **by** *simp*
moreover have *enat* (*num_args s*) < *arity_hd* (*head s*)
by (*metis* (*mono_tags*) *One_nat_def add.comm_neutral arity_def diff_add_zero enat_ord_simps*(*1*)
idiff_enat_enat less_one list.size(*4*) *nargs_s_sa_ss not_add_less2*
order.not_eq_order_implies_strict wary_num_args_le_arity_head wary_s)
ultimately show *wary* (*App s sa*)
by (*rule wary_App*[*OF wary_s*])
next
show $\bigwedge$*sa*. *sa* $\in$ *set ss* $\Longrightarrow$ *wary sa*
using *wary_ss* **by** *simp*
next
show *length ss* $\leq$ *arity* (*App s sa*)
proof (*cases arity s*)
case *enat*
thus *?thesis*
using *nargs_s_sa_ss* **by** (*simp add*: *one_enat_def*)
qed *simp*
qed
qed *simp*

lemma *wary_cases_apps*[*consumes 1*, *case_names apps*]:
assumes
wary_t: *wary t* **and**
apps: $\bigwedge\zeta$ *ss*. *t* = *apps* (*Hd* ζ) *ss* $\Longrightarrow$ ($\bigwedge$*sa*. *sa* $\in$ *set ss* $\Longrightarrow$ *wary sa*) $\Longrightarrow$ *length ss* $\leq$ *arity_hd* ζ $\Longrightarrow$ *P*
shows *P*
using *apps*
proof (*atomize_elim*, *cases t rule*: *tm_exhaust_apps*)
case *t*: (*apps* ζ *ss*)
show $\exists\zeta$ *ss*. *t* = *apps* (*Hd* ζ) *ss* $\wedge$ ($\forall$ *sa*. *sa* $\in$ *set ss* $\longrightarrow$ *wary sa*) $\wedge$ *enat* (*length ss*) $\leq$ *arity_hd* ζ
by (*rule exI*[*of* _ ζ], *rule exI*[*of* _ *ss*])
(*auto simp*: *t wary_args*[*OF* _ *wary_t*] *wary_num_args_le_arity_head*[*OF wary_t*, *unfolded t*, *simplified*])
qed

lemma *arity_hd_head*: *wary s* $\Longrightarrow$ *arity_hd* (*head s*) = *arity s* + *num_args s*
by (*simp add*: *arity_def enat_sub_add_same wary_num_args_le_arity_head*)

lemma *arity_head_ge*: *arity_hd* (*head s*) $\geq$ *arity s*
by (*induct s*) (*auto intro*: *enat_le_imp_minus_le*)

inductive *wary_fo* :: ('*s*, '*v*) *tm* $\Rightarrow$ *bool* **where**
wary_foI[*intro*]: *is_Hd s* $\vee$ *is_Sym* (*head s*) $\Longrightarrow$ *length* (*args s*) = *arity_hd* (*head s*) $\Longrightarrow$
($\forall$ *t* $\in$ *set* (*args s*). *wary_fo t*) $\Longrightarrow$ *wary_fo s*

lemma *wary_fo_args*: *s* $\in$ *set* (*args t*) $\Longrightarrow$ *wary_fo t* $\Longrightarrow$ *wary_fo s*
by (*induct t arbitrary*: *s rule*: *tm_induct_apps*, *simp*)
(*metis args.simps*(*1*) *args_apps self_append_conv2 wary_fo.cases*)

lemma *wary_fo_arg*: *wary_fo* (*App s t*) $\Longrightarrow$ *wary_fo t*
by (*erule wary_fo.cases*) *auto*

end

3.8 Potential Heads of Ground Instances of Variables

locale *ground_heads* = *gt_sym* ($>_s$) + *arity arity_sym arity_var*
  for
    gt_sym :: $'s \Rightarrow 's \Rightarrow$ *bool* (**infix** $>_s$ *50*) **and**
    arity_sym :: $'s \Rightarrow$ *enat* **and**
    arity_var :: $'v \Rightarrow$ *enat* +
  fixes
    ground_heads_var :: $'v \Rightarrow 's$ *set*
  assumes
    ground_heads_var_arity: $f \in$ *ground_heads_var* $x \Longrightarrow$ *arity_sym* $f \geq$ *arity_var* x **and**
    ground_heads_var_nonempty: *ground_heads_var* $x \neq \{\}$
begin

primrec *ground_heads* :: $('s, 'v)$ *hd* $\Rightarrow 's$ *set* **where**
  ground_heads (*Var* x) = *ground_heads_var* x
| *ground_heads* (*Sym* f) = $\{f\}$

lemma *ground_heads_arity*: $f \in$ *ground_heads* $\zeta \Longrightarrow$ *arity_sym* $f \geq$ *arity_hd* ζ
  by (*cases* ζ) (*auto simp*: *ground_heads_var_arity*)

lemma *ground_heads_nonempty*[*simp*]: *ground_heads* $\zeta \neq \{\}$
  by (*cases* ζ) (*auto simp*: *ground_heads_var_nonempty*)

lemma *sym_in_ground_heads*: *is_Sym* $\zeta \Longrightarrow$ *sym* $\zeta \in$ *ground_heads* ζ
  by (*metis ground_heads.simps*(*2*) *hd.collapse*(*2*) *hd.set_sel*(*1*) *hd.simps*(*16*))

lemma *ground_hd_in_ground_heads*: *ground* $s \Longrightarrow$ *sym* (*head* s) $\in$ *ground_heads* (*head* s)
  by (*simp add*: *ground_head sym_in_ground_heads*)

lemma *some_ground_head_arity*: *arity_sym* (*SOME* f. $f \in$ *ground_heads* (*Var* x)) $\geq$ *arity_var* x
  by (*simp add*: *ground_heads_var_arity ground_heads_var_nonempty some_in_eq*)

definition *wary_subst* :: $('v \Rightarrow ('s, 'v)$ *tm*$) \Rightarrow$ *bool* **where**
  wary_subst $\varrho \longleftrightarrow$
  $(\forall x.$ *wary* $(\varrho\ x) \wedge$ *arity* $(\varrho\ x) \geq$ *arity_var* $x \wedge$ *ground_heads* (*head* $(\varrho\ x)) \subseteq$ *ground_heads_var* $x)$

definition *strict_wary_subst* :: $('v \Rightarrow ('s, 'v)$ *tm*$) \Rightarrow$ *bool* **where**
  strict_wary_subst $\varrho \longleftrightarrow$
  $(\forall x.$ *wary* $(\varrho\ x) \wedge$ *arity* $(\varrho\ x) \in \{$*arity_var* $x, \infty\}$
   $\wedge$ *ground_heads* (*head* $(\varrho\ x)) \subseteq$ *ground_heads_var* $x)$

lemma *strict_imp_wary_subst*: *strict_wary_subst* $\varrho \Longrightarrow$ *wary_subst* ϱ
  unfolding *strict_wary_subst_def wary_subst_def* **using** *eq_iff* **by** *force*

lemma *wary_subst_wary*:
  assumes *wary_*ϱ: *wary_subst* ϱ **and** *wary_s*: *wary* s
  shows *wary* (*subst* ϱ s)
  using *wary_s*
proof (*induct* s *rule*: *tm.induct*)
  case (*App* s t)
  note *wary_st* = *this*(*3*)
  from *wary_st* **have** *wary_s*: *wary* s
    by (*rule wary_AppE*)
  from *wary_st* **have** *wary_t*: *wary* t
    by (*rule wary_AppE*)
  from *wary_st* **have** *nargs_s_lt*: *num_args* $s <$ *arity_hd* (*head* s)
    by (*rule wary_AppE*)

  note *wary_*ϱ*s* = *App*(*1*)[*OF wary_s*]
  note *wary_*ϱ*t* = *App*(*2*)[*OF wary_t*]

```
  note wary_ϱx = wary_ϱ[unfolded wary_subst_def, rule_format, THEN conjunct1]
  note ary_ϱx = wary_ϱ[unfolded wary_subst_def, rule_format, THEN conjunct2]

  have num_args (ϱ x) + num_args s < arity_hd (head (ϱ x)) if hd_s: head s = Var x for x
  proof −
    have ary_hd_s: arity_hd (head s) = arity_var x
      using hd_s arity_hd.simps(1) by presburger
    hence num_args s ≤ arity (ϱ x)
      by (metis (no_types) wary_num_args_le_arity_head ary_ϱx dual_order.trans wary_s)
    hence num_args s + num_args (ϱ x) ≤ arity_hd (head (ϱ x))
      by (metis (no_types) arity_hd_head[OF wary_ϱx] add_right_mono plus_enat_simps(1))
    thus ?thesis
      using ary_hd_s by (metis (no_types) add.commute add_diff_cancel_left' ary_ϱx arity_def
        idiff_enat_enat leD nargs_s_lt order.not_eq_order_implies_strict)
  qed
  hence nargs_ϱs: num_args (subst ϱ s) < arity_hd (head (subst ϱ s))
    using nargs_s_lt by (auto split: hd.split)

  show ?case
    by simp (rule wary_App[OF wary_ϱs wary_ϱt nargs_ϱs])
qed (auto simp: wary_ϱ[unfolded wary_subst_def] split: hd.split)

lemmas strict_wary_subst_wary = wary_subst_wary[OF strict_imp_wary_subst]

lemma wary_subst_ground_heads:
  assumes wary_ϱ: wary_subst ϱ
  shows ground_heads (head (subst ϱ s)) ⊆ ground_heads (head s)
proof (induct s rule: tm_induct_apps)
  case (apps ζ ss)
  show ?case
  proof (cases ζ)
    case x: (Var x)
    thus ?thesis
      using wary_ϱ wary_subst_def x by auto
  qed auto
qed

lemmas strict_wary_subst_ground_heads = wary_subst_ground_heads[OF strict_imp_wary_subst]

definition grounding_ϱ :: 'v ⇒ ('s, 'v) tm where
  grounding_ϱ x = (let s = Hd (Sym (SOME f. f ∈ ground_heads_var x)) in
     apps s (replicate (the_enat (arity s − arity_var x)) s))

lemma ground_grounding_ϱ: ground (subst grounding_ϱ s)
  by (induct s) (auto simp: Let_def grounding_ϱ_def elim: hd.set_cases(2) split: hd.split)

lemma strict_wary_grounding_ϱ: strict_wary_subst grounding_ϱ
  unfolding strict_wary_subst_def
proof (intro allI conjI)
  fix x

  define f where f = (SOME f. f ∈ ground_heads_var x)
  define s :: ('s, 'v) tm where s = Hd (Sym f)

  have wary_s: wary s
    unfolding s_def by (rule wary_Hd)
  have ary_s_ge_x: arity s ≥ arity_var x
    unfolding s_def f_def using some_ground_head_arity by simp
  have grϱ_x: grounding_ϱ x = apps s (replicate (the_enat (arity s − arity_var x)) s)
    unfolding grounding_ϱ_def Let_def f_def[symmetric] s_def[symmetric] by (rule refl)

  show wary (grounding_ϱ x)
    unfolding grϱ_x by (auto intro!: wary_s wary_apps[OF wary_s] enat_the_enat_minus_le)
```

show *arity* (*grounding_ϱ x*) ∈ {*arity_var x*, ∞}
unfolding *grϱ_x* **using** *ary_s_ge_x* **by** (*cases arity s*; *cases arity_var x*; *simp*)
show *ground_heads* (*head* (*grounding_ϱ x*)) ⊆ *ground_heads_var x*
unfolding *grϱ_x s_def f_def* **by** (*simp add*: *some_in_eq ground_heads_var_nonempty*)
qed

lemmas *wary_grounding_ϱ* = *strict_wary_grounding_ϱ*[*THEN strict_imp_wary_subst*]

definition *gt_hd* :: ('*s*, '*v*) *hd* ⇒ ('*s*, '*v*) *hd* ⇒ *bool* (**infix** >*hd* 50) **where**
ξ >*hd* ζ ⟷ (∀ *g* ∈ *ground_heads* ξ. ∀ *f* ∈ *ground_heads* ζ. *g* >*s* *f*)

definition *comp_hd* :: ('*s*, '*v*) *hd* ⇒ ('*s*, '*v*) *hd* ⇒ *bool* (**infix** ≤≥*hd* 50) **where**
ξ ≤≥*hd* ζ ⟷ ξ = ζ ∨ ξ >*hd* ζ ∨ ζ >*hd* ξ

lemma *gt_hd_irrefl*: ¬ ζ >*hd* ζ
unfolding *gt_hd_def* **using** *gt_sym_irrefl* **by** (*meson ex_in_conv ground_heads_nonempty*)

lemma *gt_hd_trans*: χ >*hd* ξ ⟹ ξ >*hd* ζ ⟹ χ >*hd* ζ
unfolding *gt_hd_def* **using** *gt_sym_trans* **by** (*meson ex_in_conv ground_heads_nonempty*)

lemma *gt_sym_imp_hd*: *g* >*s* *f* ⟹ *Sym g* >*hd* *Sym f*
unfolding *gt_hd_def* **by** *simp*

lemma *not_comp_hd_imp_Var*: ¬ ξ ≤≥*hd* ζ ⟹ *is_Var* ζ ∨ *is_Var* ξ
using *gt_sym_total* **by** (*cases* ζ; *cases* ξ; *auto simp*: *comp_hd_def gt_hd_def*)

end

end

4 Infinite (Non-Well-Founded) Chains

theory *Infinite_Chain*
imports *Lambda_Free_Util*
begin

This theory defines the concept of a minimal bad (or non-well-founded) infinite chain, as found in the term rewriting literature to prove the well-foundedness of syntactic term orders.

context
fixes *p* :: '*a* ⇒ '*a* ⇒ *bool*
begin

definition *inf_chain* :: (*nat* ⇒ '*a*) ⇒ *bool* **where**
inf_chain f ⟷ (∀ *i*. *p* (*f i*) (*f* (*Suc i*)))

lemma *wfP_iff_no_inf_chain*: *wfP* (λ*x y*. *p y x*) ⟷ (∄ *f*. *inf_chain f*)
unfolding *wfP_def wf_iff_no_infinite_down_chain inf_chain_def* **by** *simp*

lemma *inf_chain_offset*: *inf_chain f* ⟹ *inf_chain* (λ*j*. *f* (*j* + *i*))
unfolding *inf_chain_def* **by** *simp*

definition *bad* :: '*a* ⇒ *bool* **where**
bad x ⟷ (∃ *f*. *inf_chain f* ∧ *f* 0 = *x*)

lemma *inf_chain_bad*:
assumes *bad_f*: *inf_chain f*
shows *bad* (*f i*)
unfolding *bad_def* **by** (*rule exI*[*of* _ λ*j*. *f* (*j* + *i*)]) (*simp add*: *inf_chain_offset*[*OF bad_f*])

context
fixes *gt* :: '*a* ⇒ '*a* ⇒ *bool*
assumes *wf*: *wf* {(*x*, *y*). *gt y x*}
begin

```
primrec worst_chain :: nat ⇒ 'a where
  worst_chain 0 = (SOME x. bad x ∧ (∀ y. bad y ⟶ ¬ gt x y))
| worst_chain (Suc i) = (SOME x. bad x ∧ p (worst_chain i) x ∧
    (∀ y. bad y ∧ p (worst_chain i) y ⟶ ¬ gt x y))

declare worst_chain.simps[simp del]

context
  fixes x :: 'a
  assumes x_bad: bad x
begin

lemma
  bad_worst_chain_0: bad (worst_chain 0) and
  min_worst_chain_0: ¬ gt (worst_chain 0) x
proof −
  obtain y where bad y ∧ (∀ z. bad z ⟶ ¬ gt y z)
    using wf_exists_minimal[OF wf, of bad, OF x_bad] by force
  hence bad (worst_chain 0) ∧ (∀ z. bad z ⟶ ¬ gt (worst_chain 0) z)
    unfolding worst_chain.simps by (rule someI)
  thus bad (worst_chain 0) and ¬ gt (worst_chain 0) x
    using x_bad by blast+
qed

lemma
  bad_worst_chain_Suc: bad (worst_chain (Suc i)) and
  worst_chain_pred: p (worst_chain i) (worst_chain (Suc i)) and
  min_worst_chain_Suc: p (worst_chain i) x ⟹ ¬ gt (worst_chain (Suc i)) x
proof (induct i rule: less_induct)
  case (less i)

  have bad (worst_chain i)
  proof (cases i)
    case 0
    thus ?thesis
      using bad_worst_chain_0 by simp
  next
    case (Suc j)
    thus ?thesis
      using less(1) by blast
  qed
  then obtain fa where fa_bad: inf_chain fa and fa_0: fa 0 = worst_chain i
    unfolding bad_def by blast

  have ∃ s0. bad s0 ∧ p (worst_chain i) s0
  proof (intro exI conjI)
    let ?y0 = fa (Suc 0)

    show bad ?y0
      unfolding bad_def by (auto intro: exI[of _ λi. fa (Suc i)] inf_chain_offset[OF fa_bad])
    show p (worst_chain i) ?y0
      using fa_bad[unfolded inf_chain_def] fa_0 by metis
  qed
  then obtain y0 where y0: bad y0 ∧ p (worst_chain i) y0
    by blast

  obtain y1 where
    y1: bad y1 ∧ p (worst_chain i) y1 ∧ (∀ z. bad z ∧ p (worst_chain i) z ⟶ ¬ gt y1 z)
    using wf_exists_minimal[OF wf, of λy. bad y ∧ p (worst_chain i) y, OF y0] by force

  let ?y = worst_chain (Suc i)
```

```
  have conj: bad ?y ∧ p (worst_chain i) ?y ∧ (∀ z. bad z ∧ p (worst_chain i) z ⟶ ¬ gt ?y z)
    unfolding worst_chain.simps using y1 by (rule someI)

  show bad ?y
    by (rule conj[THEN conjunct1])
  show p (worst_chain i) ?y
    by (rule conj[THEN conjunct2, THEN conjunct1])
  show p (worst_chain i) x ⟹ ¬ gt ?y x
    using x_bad conj[THEN conjunct2, THEN conjunct2, rule_format] by meson
qed

lemma bad_worst_chain: bad (worst_chain i)
  by (cases i) (auto intro: bad_worst_chain_0 bad_worst_chain_Suc)

lemma worst_chain_bad: inf_chain worst_chain
  unfolding inf_chain_def using worst_chain_pred by metis

end

context
  fixes x :: 'a
  assumes
    x_bad: bad x and
    p_trans: ⋀z y x. p z y ⟹ p y x ⟹ p z x
begin

lemma worst_chain_not_gt: ¬ gt (worst_chain i) (worst_chain (Suc i)) for i
proof (cases i)
  case 0
  show ?thesis
    unfolding 0 by (rule min_worst_chain_0[OF inf_chain_bad[OF worst_chain_bad[OF x_bad]]])
next
  case Suc
  show ?thesis
    unfolding Suc
  by (rule min_worst_chain_Suc[OF inf_chain_bad[OF worst_chain_bad[OF x_bad]]])
    (rule p_trans[OF worst_chain_pred[OF x_bad] worst_chain_pred[OF x_bad]])
qed

end

end

end

lemma inf_chain_subset: inf_chain p f ⟹ p ≤ q ⟹ inf_chain q f
  unfolding inf_chain_def by blast

hide-fact (open) bad_worst_chain_0 bad_worst_chain_Suc

end
```

5 Extension Orders

```
theory Extension_Orders
imports Lambda_Free_Util Infinite_Chain HOL−Cardinals.Wellorder_Extension
begin
```

This theory defines locales for categorizing extension orders used for orders on λ-free higher-order terms and defines variants of the lexicographic and multiset orders.

5.1 Locales

locale *ext* =
  fixes *ext* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow bool$
  assumes
    mono_strong: $(\forall y \in set\ ys.\ \forall x \in set\ xs.\ gt\ y\ x \longrightarrow gt'\ y\ x) \Longrightarrow ext\ gt\ ys\ xs \Longrightarrow ext\ gt'\ ys\ xs$ **and**
    map: $finite\ A \Longrightarrow ys \in lists\ A \Longrightarrow xs \in lists\ A \Longrightarrow (\forall x \in A.\ \neg\ gt\ (f\ x)\ (f\ x)) \Longrightarrow$
      $(\forall z \in A.\ \forall y \in A.\ \forall x \in A.\ gt\ (f\ z)\ (f\ y) \longrightarrow gt\ (f\ y)\ (f\ x) \longrightarrow gt\ (f\ z)\ (f\ x)) \Longrightarrow$
      $(\forall y \in A.\ \forall x \in A.\ gt\ y\ x \longrightarrow gt\ (f\ y)\ (f\ x)) \Longrightarrow ext\ gt\ ys\ xs \Longrightarrow ext\ gt\ (map\ f\ ys)\ (map\ f\ xs)$
begin

lemma *mono*[*mono*]: $gt \leq gt' \Longrightarrow ext\ gt \leq ext\ gt'$
  using *mono_strong* **by** *blast*

end

locale *ext_irrefl* = *ext* +
  assumes *irrefl*: $(\forall x \in set\ xs.\ \neg\ gt\ x\ x) \Longrightarrow \neg\ ext\ gt\ xs\ xs$

locale *ext_trans* = *ext* +
  assumes *trans*: $zs \in lists\ A \Longrightarrow ys \in lists\ A \Longrightarrow xs \in lists\ A \Longrightarrow$
    $(\forall z \in A.\ \forall y \in A.\ \forall x \in A.\ gt\ z\ y \longrightarrow gt\ y\ x \longrightarrow gt\ z\ x) \Longrightarrow ext\ gt\ zs\ ys \Longrightarrow ext\ gt\ ys\ xs \Longrightarrow$
    ext gt zs xs

locale *ext_irrefl_before_trans* = *ext_irrefl* +
  assumes *trans_from_irrefl*: $finite\ A \Longrightarrow zs \in lists\ A \Longrightarrow ys \in lists\ A \Longrightarrow xs \in lists\ A \Longrightarrow$
    $(\forall x \in A.\ \neg\ gt\ x\ x) \Longrightarrow (\forall z \in A.\ \forall y \in A.\ \forall x \in A.\ gt\ z\ y \longrightarrow gt\ y\ x \longrightarrow gt\ z\ x) \Longrightarrow ext\ gt\ zs\ ys \Longrightarrow$
    $ext\ gt\ ys\ xs \Longrightarrow ext\ gt\ zs\ xs$

locale *ext_trans_before_irrefl* = *ext_trans* +
  assumes *irrefl_from_trans*: $(\forall z \in set\ xs.\ \forall y \in set\ xs.\ \forall x \in set\ xs.\ gt\ z\ y \longrightarrow gt\ y\ x \longrightarrow gt\ z\ x) \Longrightarrow$
    $(\forall x \in set\ xs.\ \neg\ gt\ x\ x) \Longrightarrow \neg\ ext\ gt\ xs\ xs$

locale *ext_irrefl_trans_strong* = *ext_irrefl* +
  assumes *trans_strong*: $(\forall z \in set\ zs.\ \forall y \in set\ ys.\ \forall x \in set\ xs.\ gt\ z\ y \longrightarrow gt\ y\ x \longrightarrow gt\ z\ x) \Longrightarrow$
    $ext\ gt\ zs\ ys \Longrightarrow ext\ gt\ ys\ xs \Longrightarrow ext\ gt\ zs\ xs$

sublocale *ext_irrefl_trans_strong* < *ext_irrefl_before_trans*
  by *standard* (*erule irrefl*, *metis in_listsD trans_strong*)

sublocale *ext_irrefl_trans_strong* < *ext_trans*
  by *standard* (*metis in_listsD trans_strong*)

sublocale *ext_irrefl_trans_strong* < *ext_trans_before_irrefl*
  by *standard* (*rule irrefl*)

locale *ext_snoc* = *ext* +
  assumes *snoc*: $ext\ gt\ (xs\ @\ [x])\ xs$

locale *ext_compat_cons* = *ext* +
  assumes *compat_cons*: $ext\ gt\ ys\ xs \Longrightarrow ext\ gt\ (x\ \#\ ys)\ (x\ \#\ xs)$
begin

lemma *compat_append_left*: $ext\ gt\ ys\ xs \Longrightarrow ext\ gt\ (zs\ @\ ys)\ (zs\ @\ xs)$
  by (*induct zs*) (*auto intro*: *compat_cons*)

end

locale *ext_compat_snoc* = *ext* +
  assumes *compat_snoc*: $ext\ gt\ ys\ xs \Longrightarrow ext\ gt\ (ys\ @\ [x])\ (xs\ @\ [x])$
begin

lemma *compat_append_right*: $ext\ gt\ ys\ xs \Longrightarrow ext\ gt\ (ys\ @\ zs)\ (xs\ @\ zs)$
  by (*induct zs arbitrary*: *xs ys rule*: *rev_induct*)

(*auto intro*: *compat_snoc simp del*: *append_assoc simp*: *append_assoc*[*symmetric*])

end

locale *ext_compat_list* = *ext* +
  assumes *compat_list*: $y \neq x \Longrightarrow gt\ y\ x \Longrightarrow ext\ gt\ (xs\ @\ y\ \#\ xs')\ (xs\ @\ x\ \#\ xs')$

locale *ext_singleton* = *ext* +
  assumes *singleton*: $y \neq x \Longrightarrow ext\ gt\ [y]\ [x] \longleftrightarrow gt\ y\ x$

locale *ext_compat_list_strong* = *ext_compat_cons* + *ext_compat_snoc* + *ext_singleton*
begin

lemma *compat_list*: $y \neq x \Longrightarrow gt\ y\ x \Longrightarrow ext\ gt\ (xs\ @\ y\ \#\ xs')\ (xs\ @\ x\ \#\ xs')$
  using *compat_append_left*[*of gt y* # *xs' x* # *xs' xs*]
    compat_append_right[*of gt*, *of* [*y*] [*x*] *xs'*] *singleton*[*of y x gt*]
  by *fastforce*

end

sublocale *ext_compat_list_strong* < *ext_compat_list*
  by *standard* (*fact compat_list*)

locale *ext_total* = *ext* +
  assumes *total*: $(\forall y \in A.\ \forall x \in A.\ gt\ y\ x \vee gt\ x\ y \vee y = x) \Longrightarrow ys \in lists\ A \Longrightarrow xs \in lists\ A \Longrightarrow$
    $ext\ gt\ ys\ xs \vee ext\ gt\ xs\ ys \vee ys = xs$

locale *ext_wf* = *ext* +
  assumes *wf*: $wfP\ (\lambda x\ y.\ gt\ y\ x) \Longrightarrow wfP\ (\lambda xs\ ys.\ ext\ gt\ ys\ xs)$

locale *ext_hd_or_tl* = *ext* +
  assumes *hd_or_tl*: $(\forall z\ y\ x.\ gt\ z\ y \longrightarrow gt\ y\ x \longrightarrow gt\ z\ x) \Longrightarrow (\forall y\ x.\ gt\ y\ x \vee gt\ x\ y \vee y = x) \Longrightarrow$
    $length\ ys = length\ xs \Longrightarrow ext\ gt\ (y\ \#\ ys)\ (x\ \#\ xs) \Longrightarrow gt\ y\ x \vee ext\ gt\ ys\ xs$

locale *ext_wf_bounded* = *ext_irrefl_before_trans* + *ext_hd_or_tl*
begin

context
  fixes *gt* :: $'a \Rightarrow 'a \Rightarrow bool$
  assumes
    gt_irrefl: $\bigwedge z.\ \neg\ gt\ z\ z$ **and**
    gt_trans: $\bigwedge z\ y\ x.\ gt\ z\ y \Longrightarrow gt\ y\ x \Longrightarrow gt\ z\ x$ **and**
    gt_total: $\bigwedge y\ x.\ gt\ y\ x \vee gt\ x\ y \vee y = x$ **and**
    gt_wf: $wfP\ (\lambda x\ y.\ gt\ y\ x)$
begin

lemma *irrefl_gt*: $\neg\ ext\ gt\ xs\ xs$
  using *irrefl gt_irrefl* **by** *simp*

lemma *trans_gt*: $ext\ gt\ zs\ ys \Longrightarrow ext\ gt\ ys\ xs \Longrightarrow ext\ gt\ zs\ xs$
  by (*rule trans_from_irrefl*[*of set zs* ∪ *set ys* ∪ *set xs zs ys xs gt*])
    (*auto intro*: *gt_trans simp*: *gt_irrefl*)

lemma *hd_or_tl_gt*: $length\ ys = length\ xs \Longrightarrow ext\ gt\ (y\ \#\ ys)\ (x\ \#\ xs) \Longrightarrow gt\ y\ x \vee ext\ gt\ ys\ xs$
  by (*rule hd_or_tl*) (*auto intro*: *gt_trans simp*: *gt_total*)

lemma *wf_same_length_if_total*: $wfP\ (\lambda xs\ ys.\ length\ ys = n \wedge length\ xs = n \wedge ext\ gt\ ys\ xs)$
proof (*induct n*)
  case *0*
  thus *?case*
    unfolding *wfP_def wf_def* **using** *irrefl* **by** *auto*
next
  case (*Suc n*)

```
  note ih = this(1)

  define gt_hd where ⋀ys xs. gt_hd ys xs ⟷ gt (hd ys) (hd xs)
  define gt_tl where ⋀ys xs. gt_tl ys xs ⟷ ext gt (tl ys) (tl xs)

  have hd_tl: gt_hd ys xs ∨ gt_tl ys xs
    if len_ys: length ys = Suc n and len_xs: length xs = Suc n and ys_gt_xs: ext gt ys xs
    for n ys xs
    using len_ys len_xs ys_gt_xs unfolding gt_hd_def gt_tl_def
    by (cases xs; cases ys) (auto simp: hd_or_tl_gt)

  show ?case
    unfolding wfP_iff_no_inf_chain
  proof (intro notI)
    let ?gtsn = λys xs. length ys = n ∧ length xs = n ∧ ext gt ys xs
    let ?gtsSn = λys xs. length ys = Suc n ∧ length xs = Suc n ∧ ext gt ys xs
    let ?gttlSn = λys xs. length ys = Suc n ∧ length xs = Suc n ∧ gt_tl ys xs

    assume ∃f. inf_chain ?gtsSn f
    then obtain xs where xs_bad: bad ?gtsSn xs
      unfolding inf_chain_def bad_def by blast

    let ?ff = worst_chain ?gtsSn gt_hd

    have wf_hd: wf {(xs, ys). gt_hd ys xs}
      unfolding gt_hd_def by (rule wfP_app[OF gt_wf, of hd, unfolded wfP_def])

    have inf_chain ?gtsSn ?ff
      by (rule worst_chain_bad[OF wf_hd xs_bad])
    moreover have ¬ gt_hd (?ff i) (?ff (Suc i)) for i
      by (rule worst_chain_not_gt[OF wf_hd xs_bad]) (blast intro: trans_gt)
    ultimately have tl_bad: inf_chain ?gttlSn ?ff
      unfolding inf_chain_def using hd_tl by blast

    have ¬ inf_chain ?gtsn (tl ∘ ?ff)
      using wfP_iff_no_inf_chain[THEN iffD1, OF ih] by blast
    hence tl_good: ¬ inf_chain ?gttlSn ?ff
      unfolding inf_chain_def gt_tl_def by force

    show False
      using tl_bad tl_good by sat
  qed
qed

lemma wf_bounded_if_total: wfP (λxs ys. length ys ≤ n ∧ length xs ≤ n ∧ ext gt ys xs)
  unfolding wfP_iff_no_inf_chain
proof (intro notI, induct n rule: less_induct)
  case (less n)
  note ih = this(1) and ex_bad = this(2)

  let ?gtsle = λys xs. length ys ≤ n ∧ length xs ≤ n ∧ ext gt ys xs

  obtain xs where xs_bad: bad ?gtsle xs
    using ex_bad unfolding inf_chain_def bad_def by blast

  let ?ff = worst_chain ?gtsle (λys xs. length ys > length xs)

  note wf_len = wf_app[OF wellorder_class.wf, of length, simplified]

  have ff_bad: inf_chain ?gtsle ?ff
    by (rule worst_chain_bad[OF wf_len xs_bad])
  have ffi_bad: ⋀i. bad ?gtsle (?ff i)
    by (rule inf_chain_bad[OF ff_bad])
```

```
  have len_le_n: ⋀i. length (?ff i) ≤ n
    using worst_chain_pred[OF wf_len xs_bad] by simp
  have len_le_Suc: ⋀i. length (?ff i) ≤ length (?ff (Suc i))
    using worst_chain_not_gt[OF wf_len xs_bad] not_le_imp_less by (blast intro: trans_gt)

  show False
  proof (cases ∃k. length (?ff k) = n)
    case False
    hence len_lt_n: ⋀i. length (?ff i) < n
      using len_le_n by (blast intro: le_neq_implies_less)
    hence nm1_le: n − 1 < n
      by fastforce

    let ?gtslt = λys xs. length ys ≤ n − 1 ∧ length xs ≤ n − 1 ∧ ext gt ys xs

    have inf_chain ?gtslt ?ff
      using ff_bad len_lt_n unfolding inf_chain_def
      by (metis (no_types, lifting) Suc_diff_1 le_antisym nat_neq_iff not_less0 not_less_eq_eq)
    thus False
      using ih[OF nm1_le] by blast
  next
    case True
    then obtain k where len_eq_n: length (?ff k) = n
      by blast

    let ?gtssl = λys xs. length ys = n ∧ length xs = n ∧ ext gt ys xs

    have len_eq_n: length (?ff (i + k)) = n for i
      by (induct i) (simp add: len_eq_n,
        metis (lifting) len_le_n len_le_Suc add_Suc dual_order.antisym)

    have inf_chain ?gtsle (λi. ?ff (i + k))
      by (rule inf_chain_offset[OF ff_bad])
    hence inf_chain ?gtssl (λi. ?ff (i + k))
      unfolding inf_chain_def using len_eq_n by presburger
    hence ¬ wfP (λxs ys. ?gtssl ys xs)
      using wfP_iff_no_inf_chain by blast
    thus False
      using wf_same_length_if_total[of n] by sat
  qed
qed

end

context
  fixes gt :: 'a ⇒ 'a ⇒ bool
  assumes
    gt_irrefl: ⋀z. ¬ gt z z and
    gt_wf: wfP (λx y. gt y x)
begin

lemma wf_bounded: wfP (λxs ys. length ys ≤ n ∧ length xs ≤ n ∧ ext gt ys xs)
proof −
  obtain Ge' where
    gt_sub_Ge': {(x, y). gt y x} ⊆ Ge' and
    Ge'_wo: Well_order Ge' and
    Ge'_fld: Field Ge' = UNIV
    using total_well_order_extension[OF gt_wf[unfolded wfP_def]] by blast

  define gt' where ⋀y x. gt' y x ⟷ y ≠ x ∧ (x, y) ∈ Ge'

  have gt_imp_gt': gt ≤ gt'
```

```
    by (auto simp: gt'_def gt_irrefl intro: gt_sub_Ge'[THEN subsetD])

  have gt'_irrefl: ⋀z. ¬ gt' z z
    unfolding gt'_def by simp

  have gt'_trans: ⋀z y x. gt' z y ⟹ gt' y x ⟹ gt' z x
    using Ge'_wo
    unfolding gt'_def well_order_on_def linear_order_on_def partial_order_on_def preorder_on_def
      trans_def antisym_def
    by blast

  have wf {(x, y). (x, y) ∈ Ge' ∧ x ≠ y}
    by (rule Ge'_wo[unfolded well_order_on_def set_diff_eq
      case_prod_eta[symmetric, of λxy. xy ∈ Ge' ∧ xy ∉ Id] pair_in_Id_conv, THEN conjunct2])
  moreover have ⋀y x. (x, y) ∈ Ge' ∧ x ≠ y ⟷ y ≠ x ∧ (x, y) ∈ Ge'
    by auto
  ultimately have gt'_wf: wfP (λx y. gt' y x)
    unfolding wfP_def gt'_def by simp

  have gt'_total: ⋀x y. gt' y x ∨ gt' x y ∨ y = x
    using Ge'_wo unfolding gt'_def well_order_on_def linear_order_on_def total_on_def Ge'_fld
    by blast

  have wfP (λxs ys. length ys ≤ n ∧ length xs ≤ n ∧ ext gt' ys xs)
    using wf_bounded_if_total gt'_total gt'_irrefl gt'_trans gt'_wf by blast
  thus ?thesis
    by (rule wfP_subset) (auto intro: mono[OF gt_imp_gt', THEN predicate2D])
qed

end

end
```

5.2 Lexicographic Extension

```
inductive lexext :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a list ⇒ bool for gt where
  lexext_Nil: lexext gt (y # ys) []
| lexext_Cons: gt y x ⟹ lexext gt (y # ys) (x # xs)
| lexext_Cons_eq: lexext gt ys xs ⟹ lexext gt (x # ys) (x # xs)

lemma lexext_simps[simp]:
  lexext gt ys [] ⟷ ys ≠ []
  ¬ lexext gt [] xs
  lexext gt (y # ys) (x # xs) ⟷ gt y x ∨ x = y ∧ lexext gt ys xs
proof
  show lexext gt ys [] ⟹ (ys ≠ [])
    by (metis lexext.cases list.distinct(1))
next
  show ys ≠ [] ⟹ lexext gt ys []
    by (metis lexext_Nil list.exhaust)
next
  show ¬ lexext gt [] xs
    using lexext.cases by auto
next
  show lexext gt (y # ys) (x # xs) = (gt y x ∨ x = y ∧ lexext gt ys xs)
  proof −
    have fwdd: lexext gt (y # ys) (x # xs) ⟶ gt y x ∨ x = y ∧ lexext gt ys xs
    proof
      assume lexext gt (y # ys) (x # xs)
      thus gt y x ∨ x = y ∧ lexext gt ys xs
        using lexext.cases by blast
    qed
    have backd: gt y x ∨ x = y ∧ lexext gt ys xs ⟶ lexext gt (y # ys) (x # xs)
      by (simp add: lexext_Cons lexext_Cons_eq)
```

```
    show lexext gt (y # ys) (x # xs) = (gt y x ∨ x = y ∧ lexext gt ys xs)
      using fwdd backd by blast
  qed
qed

lemma lexext_mono_strong:
  assumes
    ∀ y ∈ set ys. ∀ x ∈ set xs. gt y x ⟶ gt' y x and
    lexext gt ys xs
  shows lexext gt' ys xs
  using assms by (induct ys xs rule: list_induct2') auto

lemma lexext_map_strong:
  (∀ y ∈ set ys. ∀ x ∈ set xs. gt y x ⟶ gt (f y) (f x)) ⟹ lexext gt ys xs ⟹
   lexext gt (map f ys) (map f xs)
  by (induct ys xs rule: list_induct2') auto

lemma lexext_irrefl:
  assumes ∀ x ∈ set xs. ¬ gt x x
  shows ¬ lexext gt xs xs
  using assms by (induct xs) auto

lemma lexext_trans_strong:
  assumes
    ∀ z ∈ set zs. ∀ y ∈ set ys. ∀ x ∈ set xs. gt z y ⟶ gt y x ⟶ gt z x and
    lexext gt zs ys and lexext gt ys xs
  shows lexext gt zs xs
  using assms
proof (induct zs arbitrary: ys xs)
  case (Cons z zs)
  note zs_trans = this(1)
  show ?case
    using Cons(2−4)
  proof (induct ys arbitrary: xs rule: list.induct)
    case (Cons y ys)
    note ys_trans = this(1) and gt_trans = this(2) and zzs_gt_yys = this(3) and yys_gt_xs = this(4)
    show ?case
    proof (cases xs)
      case xs: (Cons x xs)
      thus ?thesis
      proof (unfold xs)
        note yys_gt_xxs = yys_gt_xs[unfolded xs]
        note gt_trans = gt_trans[unfolded xs]

        let ?case = lexext gt (z # zs) (x # xs)

        {
          assume gt z y and gt y x
          hence ?case
            using gt_trans by simp
        }
        moreover
        {
          assume gt z y and x = y
          hence ?case
            by simp
        }
        moreover
        {
          assume y = z and gt y x
          hence ?case
            by simp
        }
```

```
      moreover
      {
        assume
          y_eq_z: y = z and
          zs_gt_ys: lexext gt zs ys and
          x_eq_y: x = y and
          ys_gt_xs: lexext gt ys xs

        have lexext gt zs xs
          by (rule zs_trans[OF _ zs_gt_ys ys_gt_xs]) (meson gt_trans[simplified])
        hence ?case
          by (simp add: x_eq_y y_eq_z)
      }
      ultimately show ?case
        using zzs_gt_yys yys_gt_xxs by force
    qed
  qed auto
 qed auto
qed auto

lemma lexext_snoc: lexext gt (xs @ [x]) xs
 by (induct xs) auto

lemmas lexext_compat_cons = lexext_Cons_eq

lemma lexext_compat_snoc_if_same_length:
 assumes length ys = length xs and lexext gt ys xs
 shows lexext gt (ys @ [x]) (xs @ [x])
 using assms(2,1) by (induct rule: lexext.induct) auto

lemma lexext_compat_list: gt y x ⟹ lexext gt (xs @ y # xs') (xs @ x # xs')
 by (induct xs) auto

lemma lexext_singleton: lexext gt [y] [x] ⟷ gt y x
 by simp

lemma lexext_total: (∀ y ∈ B. ∀ x ∈ A. gt y x ∨ gt x y ∨ y = x) ⟹ ys ∈ lists B ⟹ xs ∈ lists A ⟹
 lexext gt ys xs ∨ lexext gt xs ys ∨ ys = xs
 by (induct ys xs rule: list_induct2') auto

lemma lexext_hd_or_tl: lexext gt (y # ys) (x # xs) ⟹ gt y x ∨ lexext gt ys xs
 by auto

interpretation lexext: ext lexext
 by standard (fact lexext_mono_strong, rule lexext_map_strong, metis in_listsD)

interpretation lexext: ext_irrefl_trans_strong lexext
 by standard (fact lexext_irrefl, fact lexext_trans_strong)

interpretation lexext: ext_snoc lexext
 by standard (fact lexext_snoc)

interpretation lexext: ext_compat_cons lexext
 by standard (fact lexext_compat_cons)

interpretation lexext: ext_compat_list lexext
 by standard (rule lexext_compat_list)

interpretation lexext: ext_singleton lexext
 by standard (rule lexext_singleton)

interpretation lexext: ext_total lexext
 by standard (fact lexext_total)
```

interpretation *lexext*: *ext_hd_or_tl lexext*
by *standard* (*rule lexext_hd_or_tl*)

interpretation *lexext*: *ext_wf_bounded lexext*
by *standard*

5.3 Reverse (Right-to-Left) Lexicographic Extension

abbreviation *lexext_rev* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a list* ⇒ *bool* **where**
lexext_rev gt ys xs ≡ *lexext gt* (*rev ys*) (*rev xs*)

lemma *lexext_rev_simps*[*simp*]:
lexext_rev gt ys [] ⟷ *ys* ≠ []
¬ *lexext_rev gt* [] *xs*
lexext_rev gt (*ys* @ [*y*]) (*xs* @ [*x*]) ⟷ *gt y x* ∨ *x* = *y* ∧ *lexext_rev gt ys xs*
by *simp*+

lemma *lexext_rev_cons_cons*:
assumes *length ys* = *length xs*
shows *lexext_rev gt* (*y* # *ys*) (*x* # *xs*) ⟷ *lexext_rev gt ys xs* ∨ *ys* = *xs* ∧ *gt y x*
using *assms*
proof (*induct arbitrary*: *y x rule*: *rev_induct2*)
case *Nil*
thus *?case*
by *simp*
next
case (*snoc y*′ *ys x*′ *xs*)
show *?case*
using *snoc*(*2*) **by** *auto*
qed

lemma *lexext_rev_mono_strong*:
assumes
∀ *y* ∈ *set ys*. ∀ *x* ∈ *set xs*. *gt y x* ⟶ *gt*′ *y x* **and**
lexext_rev gt ys xs
shows *lexext_rev gt*′ *ys xs*
using *assms* **by** (*simp add*: *lexext_mono_strong*)

lemma *lexext_rev_map_strong*:
(∀ *y* ∈ *set ys*. ∀ *x* ∈ *set xs*. *gt y x* ⟶ *gt* (*f y*) (*f x*)) ⟹ *lexext_rev gt ys xs* ⟹
lexext_rev gt (*map f ys*) (*map f xs*)
by (*simp add*: *lexext_map_strong rev_map*)

lemma *lexext_rev_irrefl*:
assumes ∀ *x* ∈ *set xs*. ¬ *gt x x*
shows ¬ *lexext_rev gt xs xs*
using *assms* **by** (*simp add*: *lexext_irrefl*)

lemma *lexext_rev_trans_strong*:
assumes
∀ *z* ∈ *set zs*. ∀ *y* ∈ *set ys*. ∀ *x* ∈ *set xs*. *gt z y* ⟶ *gt y x* ⟶ *gt z x* **and**
lexext_rev gt zs ys **and** *lexext_rev gt ys xs*
shows *lexext_rev gt zs xs*
using *assms*(*1*) *lexext_trans_strong*[*OF* _ *assms*(*2*,*3*), *unfolded set_rev*] **by** *sat*

lemma *lexext_rev_compat_cons_if_same_length*:
assumes *length ys* = *length xs* **and** *lexext_rev gt ys xs*
shows *lexext_rev gt* (*x* # *ys*) (*x* # *xs*)
using *assms* **by** (*simp add*: *lexext_compat_snoc_if_same_length*)

lemma *lexext_rev_compat_snoc*: *lexext_rev gt ys xs* ⟹ *lexext_rev gt* (*ys* @ [*x*]) (*xs* @ [*x*])
by (*simp add*: *lexext_compat_cons*)

lemma *lexext_rev_compat_list*: *gt y x* $\Longrightarrow$ *lexext_rev gt* (*xs* @ *y* # *xs'*) (*xs* @ *x* # *xs'*)
by (*induct xs' rule*: *rev_induct*) *auto*

lemma *lexext_rev_singleton*: *lexext_rev gt* [*y*] [*x*] $\longleftrightarrow$ *gt y x*
by *simp*

lemma *lexext_rev_total*:
($\forall$ *y* $\in$ *B*. $\forall$ *x* $\in$ *A*. *gt y x* $\vee$ *gt x y* $\vee$ *y* = *x*) $\Longrightarrow$ *ys* $\in$ *lists B* $\Longrightarrow$ *xs* $\in$ *lists A* $\Longrightarrow$
lexext_rev gt ys xs $\vee$ *lexext_rev gt xs ys* $\vee$ *ys* = *xs*
by (*rule lexext_total*[*of* _ _ _ *rev ys rev xs*, *simplified*])

lemma *lexext_rev_hd_or_tl*:
assumes
length ys = *length xs* **and**
lexext_rev gt (*y* # *ys*) (*x* # *xs*)
shows *gt y x* $\vee$ *lexext_rev gt ys xs*
using *assms lexext_rev_cons_cons* **by** *fastforce*

interpretation *lexext_rev*: *ext lexext_rev*
by *standard* (*fact lexext_rev_mono_strong*, *rule lexext_rev_map_strong*, *metis in_listsD*)

interpretation *lexext_rev*: *ext_irrefl_trans_strong lexext_rev*
by *standard* (*fact lexext_rev_irrefl*, *fact lexext_rev_trans_strong*)

interpretation *lexext_rev*: *ext_compat_snoc lexext_rev*
by *standard* (*fact lexext_rev_compat_snoc*)

interpretation *lexext_rev*: *ext_compat_list lexext_rev*
by *standard* (*rule lexext_rev_compat_list*)

interpretation *lexext_rev*: *ext_singleton lexext_rev*
by *standard* (*rule lexext_rev_singleton*)

interpretation *lexext_rev*: *ext_total lexext_rev*
by *standard* (*fact lexext_rev_total*)

interpretation *lexext_rev*: *ext_hd_or_tl lexext_rev*
by *standard* (*rule lexext_rev_hd_or_tl*)

interpretation *lexext_rev*: *ext_wf_bounded lexext_rev*
by *standard*

5.4 Generic Length Extension

definition *lenext* :: (*'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *bool*) $\Rightarrow$ *'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *bool* **where**
lenext gts ys xs $\longleftrightarrow$ *length ys* > *length xs* $\vee$ *length ys* = *length xs* $\wedge$ *gts ys xs*

lemma
lenext_mono_strong: (*gts ys xs* $\Longrightarrow$ *gts' ys xs*) $\Longrightarrow$ *lenext gts ys xs* $\Longrightarrow$ *lenext gts' ys xs* **and**
lenext_map_strong: (*length ys* = *length xs* $\Longrightarrow$ *gts ys xs* $\Longrightarrow$ *gts* (*map f ys*) (*map f xs*)) $\Longrightarrow$
lenext gts ys xs $\Longrightarrow$ *lenext gts* (*map f ys*) (*map f xs*) **and**
lenext_irrefl: $\neg$ *gts xs xs* $\Longrightarrow$ $\neg$ *lenext gts xs xs* **and**
lenext_trans: (*gts zs ys* $\Longrightarrow$ *gts ys xs* $\Longrightarrow$ *gts zs xs*) $\Longrightarrow$ *lenext gts zs ys* $\Longrightarrow$ *lenext gts ys xs* $\Longrightarrow$
lenext gts zs xs **and**
lenext_snoc: *lenext gts* (*xs* @ [*x*]) *xs* **and**
lenext_compat_cons: (*length ys* = *length xs* $\Longrightarrow$ *gts ys xs* $\Longrightarrow$ *gts* (*x* # *ys*) (*x* # *xs*)) $\Longrightarrow$
lenext gts ys xs $\Longrightarrow$ *lenext gts* (*x* # *ys*) (*x* # *xs*) **and**
lenext_compat_snoc: (*length ys* = *length xs* $\Longrightarrow$ *gts ys xs* $\Longrightarrow$ *gts* (*ys* @ [*x*]) (*xs* @ [*x*])) $\Longrightarrow$
lenext gts ys xs $\Longrightarrow$ *lenext gts* (*ys* @ [*x*]) (*xs* @ [*x*]) **and**
lenext_compat_list: *gts* (*xs* @ *y* # *xs'*) (*xs* @ *x* # *xs'*) $\Longrightarrow$
lenext gts (*xs* @ *y* # *xs'*) (*xs* @ *x* # *xs'*) **and**
lenext_singleton: *lenext gts* [*y*] [*x*] $\longleftrightarrow$ *gts* [*y*] [*x*] **and**
lenext_total: (*gts ys xs* $\vee$ *gts xs ys* $\vee$ *ys* = *xs*) $\Longrightarrow$
lenext gts ys xs $\vee$ *lenext gts xs ys* $\vee$ *ys* = *xs* **and**

lenext_hd_or_tl: (*length ys* = *length xs* ⟹ *gts* (*y* # *ys*) (*x* # *xs*) ⟹ *gt y x* ∨ *gts ys xs*) ⟹
lenext gts (*y* # *ys*) (*x* # *xs*) ⟹ *gt y x* ∨ *lenext gts ys xs*
unfolding *lenext_def* **by** *auto*

5.5 Length-Lexicographic Extension

abbreviation *len_lexext* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a list* ⇒ *bool* **where**
len_lexext gt ≡ *lenext* (*lexext gt*)

lemma *len_lexext_mono_strong*:
(∀ *y* ∈ *set ys*. ∀ *x* ∈ *set xs*. *gt y x* ⟶ *gt*′ *y x*) ⟹ *len_lexext gt ys xs* ⟹ *len_lexext gt*′ *ys xs*
by (*rule lenext_mono_strong*[*OF lexext_mono_strong*])

lemma *len_lexext_map_strong*:
(∀ *y* ∈ *set ys*. ∀ *x* ∈ *set xs*. *gt y x* ⟶ *gt* (*f y*) (*f x*)) ⟹ *len_lexext gt ys xs* ⟹
len_lexext gt (*map f ys*) (*map f xs*)
by (*rule lenext_map_strong*) (*metis lexext_map_strong*)

lemma *len_lexext_irrefl*: (∀ *x* ∈ *set xs*. ¬ *gt x x*) ⟹ ¬ *len_lexext gt xs xs*
by (*rule lenext_irrefl*[*OF lexext_irrefl*])

lemma *len_lexext_trans_strong*:
(∀ *z* ∈ *set zs*. ∀ *y* ∈ *set ys*. ∀ *x* ∈ *set xs*. *gt z y* ⟶ *gt y x* ⟶ *gt z x*) ⟹ *len_lexext gt zs ys* ⟹
len_lexext gt ys xs ⟹ *len_lexext gt zs xs*
by (*rule lenext_trans*[*OF lexext_trans_strong*])

lemma *len_lexext_snoc*: *len_lexext gt* (*xs* @ [*x*]) *xs*
by (*rule lenext_snoc*)

lemma *len_lexext_compat_cons*: *len_lexext gt ys xs* ⟹ *len_lexext gt* (*x* # *ys*) (*x* # *xs*)
by (*intro lenext_compat_cons lexext_compat_cons*)

lemma *len_lexext_compat_snoc*: *len_lexext gt ys xs* ⟹ *len_lexext gt* (*ys* @ [*x*]) (*xs* @ [*x*])
by (*intro lenext_compat_snoc lexext_compat_snoc_if_same_length*)

lemma *len_lexext_compat_list*: *gt y x* ⟹ *len_lexext gt* (*xs* @ *y* # *xs*′) (*xs* @ *x* # *xs*′)
by (*intro lenext_compat_list lexext_compat_list*)

lemma *len_lexext_singleton*[*simp*]: *len_lexext gt* [*y*] [*x*] ⟷ *gt y x*
by (*simp only*: *lenext_singleton lexext_singleton*)

lemma *len_lexext_total*: (∀ *y* ∈ *B*. ∀ *x* ∈ *A*. *gt y x* ∨ *gt x y* ∨ *y* = *x*) ⟹ *ys* ∈ *lists B* ⟹ *xs* ∈ *lists A* ⟹
len_lexext gt ys xs ∨ *len_lexext gt xs ys* ∨ *ys* = *xs*
by (*rule lenext_total*[*OF lexext_total*])

lemma *len_lexext_iff_lenlex*: *len_lexext gt ys xs* ⟷ (*xs*, *ys*) ∈ *lenlex* {(*x*, *y*). *gt y x*}
proof −
{
assume *length xs* = *length ys*
hence *lexext gt ys xs* ⟷ (*xs*, *ys*) ∈ *lex* {(*x*, *y*). *gt y x*}
by (*induct xs ys rule*: *list_induct2*) *auto*
}
thus *?thesis*
unfolding *lenext_def lenlex_conv* **by** *auto*
qed

lemma *len_lexext_wf*: *wfP* (λ*x y*. *gt y x*) ⟹ *wfP* (λ*xs ys*. *len_lexext gt ys xs*)
unfolding *wfP_def len_lexext_iff_lenlex* **by** (*simp add*: *wf_lenlex*)

lemma *len_lexext_hd_or_tl*: *len_lexext gt* (*y* # *ys*) (*x* # *xs*) ⟹ *gt y x* ∨ *len_lexext gt ys xs*
using *lenext_hd_or_tl lexext_hd_or_tl* **by** *metis*

interpretation *len_lexext*: *ext len_lexext*
by *standard* (*fact len_lexext_mono_strong*, *rule len_lexext_map_strong*, *metis in_listsD*)

interpretation *len_lexext*: *ext_irrefl_trans_strong len_lexext*
by *standard* (*fact len_lexext_irrefl*, *fact len_lexext_trans_strong*)

interpretation *len_lexext*: *ext_snoc len_lexext*
by *standard* (*fact len_lexext_snoc*)

interpretation *len_lexext*: *ext_compat_cons len_lexext*
by *standard* (*fact len_lexext_compat_cons*)

interpretation *len_lexext*: *ext_compat_snoc len_lexext*
by *standard* (*fact len_lexext_compat_snoc*)

interpretation *len_lexext*: *ext_compat_list len_lexext*
by *standard* (*rule len_lexext_compat_list*)

interpretation *len_lexext*: *ext_singleton len_lexext*
by *standard* (*rule len_lexext_singleton*)

interpretation *len_lexext*: *ext_total len_lexext*
by *standard* (*fact len_lexext_total*)

interpretation *len_lexext*: *ext_wf len_lexext*
by *standard* (*fact len_lexext_wf*)

interpretation *len_lexext*: *ext_hd_or_tl len_lexext*
by *standard* (*rule len_lexext_hd_or_tl*)

interpretation *len_lexext*: *ext_wf_bounded len_lexext*
by *standard*

5.6 Reverse (Right-to-Left) Length-Lexicographic Extension

abbreviation *len_lexext_rev* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow bool$ **where**
$len\_lexext\_rev\ gt \equiv lenext\ (lexext\_rev\ gt)$

lemma *len_lexext_rev_mono_strong*:
$(\forall y \in set\ ys.\ \forall x \in set\ xs.\ gt\ y\ x \longrightarrow gt'\ y\ x) \Longrightarrow len\_lexext\_rev\ gt\ ys\ xs \Longrightarrow len\_lexext\_rev\ gt'\ ys\ xs$
by (*rule lenext_mono_strong*) (*rule lexext_rev_mono_strong*)

lemma *len_lexext_rev_map_strong*:
$(\forall y \in set\ ys.\ \forall x \in set\ xs.\ gt\ y\ x \longrightarrow gt\ (f\ y)\ (f\ x)) \Longrightarrow len\_lexext\_rev\ gt\ ys\ xs \Longrightarrow$
$len\_lexext\_rev\ gt\ (map\ f\ ys)\ (map\ f\ xs)$
by (*rule lenext_map_strong*) (*rule lexext_rev_map_strong*)

lemma *len_lexext_rev_irrefl*: $(\forall x \in set\ xs.\ \neg\ gt\ x\ x) \Longrightarrow \neg\ len\_lexext\_rev\ gt\ xs\ xs$
by (*rule lenext_irrefl*) (*rule lexext_rev_irrefl*)

lemma *len_lexext_rev_trans_strong*:
$(\forall z \in set\ zs.\ \forall y \in set\ ys.\ \forall x \in set\ xs.\ gt\ z\ y \longrightarrow gt\ y\ x \longrightarrow gt\ z\ x) \Longrightarrow len\_lexext\_rev\ gt\ zs\ ys \Longrightarrow$
$len\_lexext\_rev\ gt\ ys\ xs \Longrightarrow len\_lexext\_rev\ gt\ zs\ xs$
by (*rule lenext_trans*) (*rule lexext_rev_trans_strong*)

lemma *len_lexext_rev_snoc*: $len\_lexext\_rev\ gt\ (xs\ @\ [x])\ xs$
by (*rule lenext_snoc*)

lemma *len_lexext_rev_compat_cons*: $len\_lexext\_rev\ gt\ ys\ xs \Longrightarrow len\_lexext\_rev\ gt\ (x\ \#\ ys)\ (x\ \#\ xs)$
by (*intro lenext_compat_cons lexext_rev_compat_cons_if_same_length*)

lemma *len_lexext_rev_compat_snoc*: $len\_lexext\_rev\ gt\ ys\ xs \Longrightarrow len\_lexext\_rev\ gt\ (ys\ @\ [x])\ (xs\ @\ [x])$
by (*intro lenext_compat_snoc lexext_rev_compat_snoc*)

lemma *len_lexext_rev_compat_list*: $gt\ y\ x \Longrightarrow len\_lexext\_rev\ gt\ (xs\ @\ y\ \#\ xs')\ (xs\ @\ x\ \#\ xs')$
by (*intro lenext_compat_list lexext_rev_compat_list*)

lemma *len_lexext_rev_singleton*[*simp*]: *len_lexext_rev gt* [*y*] [*x*] ⟷ *gt y x*
by (*simp only*: *lenext_singleton lexext_rev_singleton*)

lemma *len_lexext_rev_total*: (∀ *y* ∈ *B*. ∀ *x* ∈ *A*. *gt y x* ∨ *gt x y* ∨ *y* = *x*) ⟹ *ys* ∈ *lists B* ⟹
xs ∈ *lists A* ⟹ *len_lexext_rev gt ys xs* ∨ *len_lexext_rev gt xs ys* ∨ *ys* = *xs*
by (*rule lenext_total*[*OF lexext_rev_total*])

lemma *len_lexext_rev_iff_len_lexext*: *len_lexext_rev gt ys xs* ⟷ *len_lexext gt* (*rev ys*) (*rev xs*)
unfolding *lenext_def* **by** *simp*

lemma *len_lexext_rev_wf*: *wfP* (λ*x y*. *gt y x*) ⟹ *wfP* (λ*xs ys*. *len_lexext_rev gt ys xs*)
unfolding *len_lexext_rev_iff_len_lexext*
by (*rule wfP_app*[*of* λ*xs ys*. *len_lexext gt ys xs rev*, *simplified*]) (*rule len_lexext_wf*)

lemma *len_lexext_rev_hd_or_tl*:
len_lexext_rev gt (*y* # *ys*) (*x* # *xs*) ⟹ *gt y x* ∨ *len_lexext_rev gt ys xs*
using *lenext_hd_or_tl lexext_rev_hd_or_tl* **by** *metis*

interpretation *len_lexext_rev*: *ext len_lexext_rev*
by *standard* (*fact len_lexext_rev_mono_strong*, *rule len_lexext_rev_map_strong*, *metis in_listsD*)

interpretation *len_lexext_rev*: *ext_irrefl_trans_strong len_lexext_rev*
by *standard* (*fact len_lexext_rev_irrefl*, *fact len_lexext_rev_trans_strong*)

interpretation *len_lexext_rev*: *ext_snoc len_lexext_rev*
by *standard* (*fact len_lexext_rev_snoc*)

interpretation *len_lexext_rev*: *ext_compat_cons len_lexext_rev*
by *standard* (*fact len_lexext_rev_compat_cons*)

interpretation *len_lexext_rev*: *ext_compat_snoc len_lexext_rev*
by *standard* (*fact len_lexext_rev_compat_snoc*)

interpretation *len_lexext_rev*: *ext_compat_list len_lexext_rev*
by *standard* (*rule len_lexext_rev_compat_list*)

interpretation *len_lexext_rev*: *ext_singleton len_lexext_rev*
by *standard* (*rule len_lexext_rev_singleton*)

interpretation *len_lexext_rev*: *ext_total len_lexext_rev*
by *standard* (*fact len_lexext_rev_total*)

interpretation *len_lexext_rev*: *ext_wf len_lexext_rev*
by *standard* (*fact len_lexext_rev_wf*)

interpretation *len_lexext_rev*: *ext_hd_or_tl len_lexext_rev*
by *standard* (*rule len_lexext_rev_hd_or_tl*)

interpretation *len_lexext_rev*: *ext_wf_bounded len_lexext_rev*
by *standard*

5.7 Dershowitz–Manna Multiset Extension

definition *msetext_dersh* **where**
msetext_dersh gt ys xs = (*let N* = *mset ys*; *M* = *mset xs in*
(∃ *Y X*. *Y* ≠ {#} ∧ *Y* ⊆# *N* ∧ *M* = (*N* − *Y*) + *X* ∧ (∀ *x*. *x* ∈# *X* ⟶ (∃ *y*. *y* ∈# *Y* ∧ *gt y x*))))

The following proof is based on that of *less_multiset*$_{DM}$*_imp_mult*.

lemma *msetext_dersh_imp_mult_rel*:
assumes
ys_a: *ys* ∈ *lists A* **and** *xs_a*: *xs* ∈ *lists A* **and**
ys_gt_xs: *msetext_dersh gt ys xs*
shows (*mset xs*, *mset ys*) ∈ *mult* {(*x*, *y*). *x* ∈ *A* ∧ *y* ∈ *A* ∧ *gt y x*}

```
proof −
  obtain Y X where y_nemp: Y ≠ {#} and y_sub_ys: Y ⊆# mset ys and
    xs_eq: mset xs = mset ys − Y + X and ex_y: ∀ x. x ∈# X ⟶ (∃ y. y ∈# Y ∧ gt y x)
    using ys_gt_xs[unfolded msetext_dersh_def Let_def] by blast
  have ex_y': ∀ x. x ∈# X ⟶ (∃ y. y ∈# Y ∧ x ∈ A ∧ y ∈ A ∧ gt y x)
    using ex_y y_sub_ys xs_eq ys_a xs_a by (metis in_listsD mset_subset_eqD set_mset_mset union_iff)
  hence (mset ys − Y + X, mset ys − Y + Y) ∈ mult {(x, y). x ∈ A ∧ y ∈ A ∧ gt y x}
    using y_nemp y_sub_ys by (intro one_step_implies_mult) (auto simp: Bex_def trans_def)
  thus ?thesis
    using xs_eq y_sub_ys by (simp add: subset_mset.diff_add)
qed

lemma msetext_dersh_imp_mult: msetext_dersh gt ys xs ⟹ (mset xs, mset ys) ∈ mult {(x, y). gt y x}
  using msetext_dersh_imp_mult_rel[of _ UNIV] by auto

lemma mult_imp_msetext_dersh_rel:
  assumes
    ys_a: set_mset (mset ys) ⊆ A and xs_a: set_mset (mset xs) ⊆ A and
    in_mult: (mset xs, mset ys) ∈ mult {(x, y). x ∈ A ∧ y ∈ A ∧ gt y x} and
    trans: ∀ z ∈ A. ∀ y ∈ A. ∀ x ∈ A. gt z y ⟶ gt y x ⟶ gt z x
  shows msetext_dersh gt ys xs
  using in_mult ys_a xs_a unfolding mult_def msetext_dersh_def Let_def
proof induct
  case (base Ys)
  then obtain y M0 X where Ys = M0 + {#y#} and mset xs = M0 + X and ∀ a. a ∈# X ⟶ gt y a
    unfolding mult1_def by auto
  thus ?case
    by (auto intro: exI[of _ {#y#}] exI[of _ X])
next
  case (step Ys Zs)
  note ys_zs_in_mult1 = this(2) and ih = this(3) and zs_a = this(4) and xs_a = this(5)

  have Ys_a: set_mset Ys ⊆ A
    using ys_zs_in_mult1 zs_a unfolding mult1_def by auto

  obtain Y X where y_nemp: Y ≠ {#} and y_sub_ys: Y ⊆# Ys and xs_eq: mset xs = Ys − Y + X and
    ex_y: ∀ x. x ∈# X ⟶ (∃ y. y ∈# Y ∧ gt y x)
    using ih[OF Ys_a xs_a] by blast

  obtain z M0 Ya where zs_eq: Zs = M0 + {#z#} and ys_eq: Ys = M0 + Ya and
    z_gt: ∀ y. y ∈# Ya ⟶ y ∈ A ∧ z ∈ A ∧ gt z y
    using ys_zs_in_mult1[unfolded mult1_def] by auto

  let ?Za = Y − Ya + {#z#}
  let ?Xa = X + Ya + (Y − Ya) − Y

  have xa_sub_x_ya: set_mset ?Xa ⊆ set_mset (X + Ya)
    by (metis diff_subset_eq_self in_diffD subsetI subset_mset.diff_diff_right)

  have x_a: set_mset X ⊆ A
    using xs_a xs_eq by auto
  have ya_a: set_mset Ya ⊆ A
    by (simp add: subsetI z_gt)

  have ex_y': ∃ y. y ∈# Y − Ya + {#z#} ∧ gt y x if x_in: x ∈# X + Ya for x
  proof (cases x ∈# X)
    case True
    then obtain y where y_in: y ∈# Y and y_gt_x: gt y x
      using ex_y by blast
    show ?thesis
    proof (cases y ∈# Ya)
      case False
      hence y ∈# Y − Ya + {#z#}
```

```
      using y_in by fastforce
    thus ?thesis
      using y_gt_x by blast
  next
    case True
    hence y ∈ A and z ∈ A and gt z y
      using z_gt by blast+
    hence gt z x
      using trans y_gt_x x_a ya_a x_in by (meson subsetCE union_iff)
    thus ?thesis
      by auto
  qed
next
  case False
  hence x ∈# Ya
    using x_in by auto
  hence x ∈ A and z ∈ A and gt z x
    using z_gt by blast+
  thus ?thesis
    by auto
qed

show ?case
proof (rule exI[of _ ?Za], rule exI[of _ ?Xa], intro conjI)
  show Y − Ya + {#z#} ⊆# Zs
    using mset_subset_eq_mono_add subset_eq_diff_conv y_sub_ys ys_eq zs_eq by fastforce
next
  show mset xs = Zs − (Y − Ya + {#z#}) + (X + Ya + (Y − Ya) − Y)
    unfolding xs_eq ys_eq zs_eq by (auto simp: multiset_eq_iff)
next
  show ∀ x. x ∈# X + Ya + (Y − Ya) − Y ⟶ (∃ y. y ∈# Y − Ya + {#z#} ∧ gt y x)
    using ex_y' xa_sub_x_ya by blast
qed auto
qed

lemma msetext_dersh_mono_strong:
(∀ y ∈ set ys. ∀ x ∈ set xs. gt y x ⟶ gt' y x) ⟹ msetext_dersh gt ys xs ⟹
msetext_dersh gt' ys xs
unfolding msetext_dersh_def Let_def
by (metis mset_subset_eqD mset_subset_eq_add_right set_mset_mset)

lemma msetext_dersh_map_strong:
assumes
  compat_f: ∀ y ∈ set ys. ∀ x ∈ set xs. gt y x ⟶ gt (f y) (f x) and
  ys_gt_xs: msetext_dersh gt ys xs
shows msetext_dersh gt (map f ys) (map f xs)
proof −
obtain Y X where
  y_nemp: Y ≠ {#} and y_sub_ys: Y ⊆# mset ys and xs_eq: mset xs = mset ys − Y + X and
  ex_y: ∀ x. x ∈# X ⟶ (∃ y. y ∈# Y ∧ gt y x)
  using ys_gt_xs[unfolded msetext_dersh_def Let_def mset_map] by blast

have x_sub_xs: X ⊆# mset xs
  using xs_eq by simp

let ?fY = image_mset f Y
let ?fX = image_mset f X

show ?thesis
  unfolding msetext_dersh_def Let_def mset_map
proof (intro exI conjI)
  show image_mset f (mset xs) = image_mset f (mset ys) − ?fY + ?fX
    using xs_eq[THEN arg_cong, of image_mset f] y_sub_ys by (metis image_mset_Diff image_mset_union)
```

```
  next
    obtain y where y: ∀x. x ∈# X ⟶ y x ∈# Y ∧ gt (y x) x
      using ex_y by moura

    show ∀fx. fx ∈# ?fX ⟶ (∃fy. fy ∈# ?fY ∧ gt fy fx)
    proof (intro allI impI)
      fix fx
      assume fx ∈# ?fX
      then obtain x where fx: fx = f x and x_in: x ∈# X
        by auto
      hence y_in: y x ∈# Y and y_gt: gt (y x) x
        using y[rule_format, OF x_in] by blast+
      hence f (y x) ∈# ?fY ∧ gt (f (y x)) (f x)
        using compat_f y_sub_ys x_sub_xs x_in
        by (metis image_eqI in_image_mset mset_subset_eqD set_mset_mset)
      thus ∃fy. fy ∈# ?fY ∧ gt fy fx
        unfolding fx by auto
    qed
  qed (auto simp: y_nemp y_sub_ys image_mset_subseteq_mono)
qed

lemma msetext_dersh_trans:
  assumes
    zs_a: zs ∈ lists A and
    ys_a: ys ∈ lists A and
    xs_a: xs ∈ lists A and
    trans: ∀z ∈ A. ∀y ∈ A. ∀x ∈ A. gt z y ⟶ gt y x ⟶ gt z x and
    zs_gt_ys: msetext_dersh gt zs ys and
    ys_gt_xs: msetext_dersh gt ys xs
  shows msetext_dersh gt zs xs
proof (rule mult_imp_msetext_dersh_rel[OF _ _ _ trans])
  show set_mset (mset zs) ⊆ A
    using zs_a by auto
next
  show set_mset (mset xs) ⊆ A
    using xs_a by auto
next
  let ?Gt = {(x, y). x ∈ A ∧ y ∈ A ∧ gt y x}

  have (mset xs, mset ys) ∈ mult ?Gt
    by (rule msetext_dersh_imp_mult_rel[OF ys_a xs_a ys_gt_xs])
  moreover have (mset ys, mset zs) ∈ mult ?Gt
    by (rule msetext_dersh_imp_mult_rel[OF zs_a ys_a zs_gt_ys])
  ultimately show (mset xs, mset zs) ∈ mult ?Gt
    unfolding mult_def by simp
qed

lemma msetext_dersh_irrefl_from_trans:
  assumes
    trans: ∀z ∈ set xs. ∀y ∈ set xs. ∀x ∈ set xs. gt z y ⟶ gt y x ⟶ gt z x and
    irrefl: ∀x ∈ set xs. ¬ gt x x
  shows ¬ msetext_dersh gt xs xs
  unfolding msetext_dersh_def Let_def
proof clarify
  fix Y X
  assume y_nemp: Y ≠ {#} and y_sub_xs: Y ⊆# mset xs and xs_eq: mset xs = mset xs − Y + X and
    ex_y: ∀x. x ∈# X ⟶ (∃y. y ∈# Y ∧ gt y x)

  have x_eq_y: X = Y
    using y_sub_xs xs_eq by (metis diff_union_cancelL subset_mset.diff_add)

  let ?Gt = {(y, x). y ∈# Y ∧ x ∈# Y ∧ gt y x}
```

```
  have ?Gt ⊆ set_mset Y × set_mset Y
    by auto
  hence fin: finite ?Gt
    by (auto dest!: infinite_super)
  moreover have irefl ?Gt
    unfolding irefl_def using irefl y_sub_xs by (fastforce dest!: set_mset_mono)
  moreover have trans ?Gt
    unfolding trans_def using trans y_sub_xs by (fastforce dest!: set_mset_mono)
  ultimately have acyc: acyclic ?Gt
    by (rule finite_irefl_trans_imp_wf[THEN wf_acyclic])

  have fin_y: finite (set_mset Y)
    using y_sub_xs by simp
  hence cyc: ¬ acyclic ?Gt
    proof (rule finite_nonempty_ex_succ_imp_cyclic)
      show ∀ x ∈# Y. ∃ y ∈# Y. (y, x) ∈ ?Gt
        using ex_y[unfolded x_eq_y] by auto
    qed (auto simp: y_nemp)

  show False
    using acyc cyc by sat
qed

lemma msetext_dersh_snoc: msetext_dersh gt (xs @ [x]) xs
  unfolding msetext_dersh_def Let_def
proof (intro exI conjI)
  show mset xs = mset (xs @ [x]) − {#x#} + {#}
    by simp
qed auto

lemma msetext_dersh_compat_cons:
  assumes ys_gt_xs: msetext_dersh gt ys xs
  shows msetext_dersh gt (x # ys) (x # xs)
proof −
  obtain Y X where
    y_nemp: Y ≠ {#} and y_sub_ys: Y ⊆# mset ys and xs_eq: mset xs = mset ys − Y + X and
    ex_y: ∀ x. x ∈# X ⟶ (∃ y. y ∈# Y ∧ gt y x)
    using ys_gt_xs[unfolded msetext_dersh_def Let_def mset_map] by blast

  show ?thesis
    unfolding msetext_dersh_def Let_def
  proof (intro exI conjI)
    show Y ⊆# mset (x # ys)
      using y_sub_ys
      by (metis add_mset_add_single mset.simps(2) mset_subset_eq_add_left
        subset_mset.add_increasing2)
  next
    show mset (x # xs) = mset (x # ys) − Y + X
    proof −
      have X + (mset ys − Y) = mset xs
        by (simp add: union_commute xs_eq)
      hence mset (x # xs) = X + (mset (x # ys) − Y)
        by (metis add_mset_add_single mset.simps(2) mset_subset_eq_multiset_union_diff_commute
          union_mset_add_mset_right y_sub_ys)
      thus ?thesis
        by (simp add: union_commute)
    qed
  qed (auto simp: y_nemp ex_y)
qed

lemma msetext_dersh_compat_snoc: msetext_dersh gt ys xs ⟹ msetext_dersh gt (ys @ [x]) (xs @ [x])
  using msetext_dersh_compat_cons[of gt ys xs x] unfolding msetext_dersh_def by simp
```

lemma *msetext_dersh_compat_list*:
assumes *y_gt_x*: *gt y x*
shows *msetext_dersh gt* (*xs* @ *y* # *xs'*) (*xs* @ *x* # *xs'*)
unfolding *msetext_dersh_def Let_def*
proof (*intro exI conjI*)
show *mset* (*xs* @ *x* # *xs'*) = *mset* (*xs* @ *y* # *xs'*) − {#*y*#} + {#*x*#}
by *auto*
qed (*auto intro*: *y_gt_x*)

lemma *msetext_dersh_singleton*: *msetext_dersh gt* [*y*] [*x*] ⟷ *gt y x*
unfolding *msetext_dersh_def Let_def*
by (*auto dest*: *nonempty_subseteq_mset_eq_singleton simp*: *nonempty_subseteq_mset_iff_singleton*)

lemma *msetext_dersh_wf*:
assumes *wf_gt*: *wfP* (λ*x y*. *gt y x*)
shows *wfP* (λ*xs ys*. *msetext_dersh gt ys xs*)
proof (*rule wfP_subset*, *rule wfP_app*[*of* λ*xs ys*. (*xs*, *ys*) ∈ *mult* {(*x*, *y*). *gt y x*} *mset*])
show *wfP* (λ*xs ys*. (*xs*, *ys*) ∈ *mult* {(*x*, *y*). *gt y x*})
using *wf_gt* **unfolding** *wfP_def* **by** (*auto intro*: *wf_mult*)
next
show (λ*xs ys*. *msetext_dersh gt ys xs*) ≤ (λ*x y*. (*mset x*, *mset y*) ∈ *mult* {(*x*, *y*). *gt y x*})
using *msetext_dersh_imp_mult* **by** *blast*
qed

interpretation *msetext_dersh*: *ext msetext_dersh*
by *standard* (*fact msetext_dersh_mono_strong*, *rule msetext_dersh_map_strong*, *metis in_listsD*)

interpretation *msetext_dersh*: *ext_trans_before_irrefl msetext_dersh*
by *standard* (*fact msetext_dersh_trans*, *fact msetext_dersh_irrefl_from_trans*)

interpretation *msetext_dersh*: *ext_snoc msetext_dersh*
by *standard* (*fact msetext_dersh_snoc*)

interpretation *msetext_dersh*: *ext_compat_cons msetext_dersh*
by *standard* (*fact msetext_dersh_compat_cons*)

interpretation *msetext_dersh*: *ext_compat_snoc msetext_dersh*
by *standard* (*fact msetext_dersh_compat_snoc*)

interpretation *msetext_dersh*: *ext_compat_list msetext_dersh*
by *standard* (*rule msetext_dersh_compat_list*)

interpretation *msetext_dersh*: *ext_singleton msetext_dersh*
by *standard* (*rule msetext_dersh_singleton*)

interpretation *msetext_dersh*: *ext_wf msetext_dersh*
by *standard* (*fact msetext_dersh_wf*)

5.8 Huet–Oppen Multiset Extension

definition *msetext_huet* **where**
msetext_huet gt ys xs = (*let N* = *mset ys*; *M* = *mset xs in*
M ≠ *N* ∧ (∀ *x*. *count M x* > *count N x* ⟶ (∃ *y*. *gt y x* ∧ *count N y* > *count M y*)))

lemma *msetext_huet_imp_count_gt*:
assumes *ys_gt_xs*: *msetext_huet gt ys xs*
shows ∃ *x*. *count* (*mset ys*) *x* > *count* (*mset xs*) *x*
proof −
obtain *x* **where** *count* (*mset ys*) *x* ≠ *count* (*mset xs*) *x*
using *ys_gt_xs*[*unfolded msetext_huet_def Let_def*] **by** (*fastforce intro*: *multiset_eqI*)
moreover
{
assume *count* (*mset ys*) *x* < *count* (*mset xs*) *x*
hence *?thesis*

```
    using ys_gt_xs[unfolded msetext_huet_def Let_def] by blast
  }
  moreover
  {
    assume count (mset ys) x > count (mset xs) x
    hence ?thesis
      by fast
  }
  ultimately show ?thesis
    by fastforce
qed

lemma msetext_huet_imp_dersh:
  assumes huet: msetext_huet gt ys xs
  shows msetext_dersh gt ys xs
proof (unfold msetext_dersh_def Let_def, intro exI conjI)
  let ?X = mset xs − mset ys
  let ?Y = mset ys − mset xs

  show ?Y ≠ {#}
    by (metis msetext_huet_imp_count_gt[OF huet] empty_iff in_diff_count set_mset_empty)
  show ?Y ⊆# mset ys
    by auto
  show mset xs = mset ys − ?Y + ?X
    by (rule multiset_eqI) simp
  show ∀ x. x ∈# ?X ⟶ (∃ y. y ∈# ?Y ∧ gt y x)
    using huet[unfolded msetext_huet_def Let_def, THEN conjunct2] by (meson in_diff_count)
qed
```

The following proof is based on that of $mult\_imp\_less\_multiset_{HO}$.

```
lemma mult_imp_msetext_huet:
  assumes
    irefl: irreflp gt and trans: transp gt and
    in_mult: (mset xs, mset ys) ∈ mult {(x, y). gt y x}
  shows msetext_huet gt ys xs
  using in_mult unfolding mult_def msetext_huet_def Let_def
proof (induct rule: tranclp_induct)
  case (base Ys)
  thus ?case
    using irefl unfolding irreflp_def msetext_huet_def Let_def mult1_def
    by (auto 0 3 split: if_splits)
next
  case (step Ys Zs)

  have asym[unfolded antisym_def, simplified]: antisymp gt
    by (rule irreflp_transp_imp_antisymP[OF irefl trans])

  from step(3) have mset xs ≠ Ys and
    **: ⋀x. count Ys x < count (mset xs) x ⟹ (∃ y. gt y x ∧ count (mset xs) y < count Ys y)
    by blast+
  from step(2) obtain M0 a K where
    *: Zs = M0 + {#a#} Ys = M0 + K a ∉# K ⋀b. b ∈# K ⟹ gt a b
    using irefl unfolding mult1_def irreflp_def by force
  have mset xs ≠ Zs
  proof (cases K = {#})
    case True
    thus ?thesis
      using ‹mset xs ≠ Ys› ** *(1,2) irefl[unfolded irreflp_def]
      by (metis One_nat_def add.comm_neutral count_single diff_union_cancelL lessI
        minus_multiset.rep_eq not_add_less2 plus_multiset.rep_eq union_commute zero_less_diff)
  next
    case False
    thus ?thesis
```

```
  proof −
    obtain aa :: 'a ⇒ 'a where
      f1: ∀ a. ¬ count Ys a < count (mset xs) a ∨ gt (aa a) a ∧
        count (mset xs) (aa a) < count Ys (aa a)
      using ∗∗ by moura
    have f2: K + M0 = Ys
      using ∗(2) union_ac(2) by blast
    have f3: ⋀aa. count Zs aa = count M0 aa + count {#a#} aa
      by (simp add: ∗(1))
    have f4: ⋀a. count Ys a = count K a + count M0 a
      using f2 by auto
    have f5: count K a = 0
      by (meson ∗(3) count_inI)
    have Zs − M0 = {#a#}
      using ∗(1) add_diff_cancel_left' by blast
    then have f6: count M0 a < count Zs a
      by (metis in_diff_count union_single_eq_member)
    have ⋀m. count m a = 0 + count m a
      by simp
    moreover
    { assume aa a ≠ a
      then have mset xs = Zs ∧ count Zs (aa a) < count K (aa a) + count M0 (aa a) ⟶
        count K (aa a) + count M0 (aa a) < count Zs (aa a)
        using f5 f3 f2 f1 ∗(4) asym by (auto dest!: antisympD) }
    ultimately show ?thesis
      using f6 f5 f4 f1 by (metis less_imp_not_less)
  qed
qed
moreover
{
  assume count Zs a ≤ count (mset xs) a
  with ‹a ∉# K› have count Ys a < count (mset xs) a unfolding ∗(1,2)
    by (auto simp add: not_in_iff)
    with ∗∗ obtain z where z: gt z a count (mset xs) z < count Ys z
      by blast
    with ∗ have count Ys z ≤ count Zs z
      using asym
      by (auto simp: intro: count_inI dest: antisympD)
    with z have ∃ z. gt z a ∧ count (mset xs) z < count Zs z by auto
}
note count_a = this
{
  fix y
  assume count_y: count Zs y < count (mset xs) y
  have ∃ x. gt x y ∧ count (mset xs) x < count Zs x
  proof (cases y = a)
    case True
    with count_y count_a show ?thesis by auto
  next
    case False
    show ?thesis
    proof (cases y ∈# K)
      case True
      with ∗(4) have gt a y by simp
      then show ?thesis
        by (cases count Zs a ≤ count (mset xs) a,
          blast dest: count_a trans[unfolded transp_def, rule_format], auto dest: count_a)
    next
      case False
      with ‹y ≠ a› have count Zs y = count Ys y unfolding ∗(1,2)
        by (simp add: not_in_iff)
      with count_y ∗∗ obtain z where z: gt z y count (mset xs) z < count Ys z by auto
      show ?thesis
```

```
      proof (cases z ∈# K)
        case True
        with *(4) have gt a z by simp
        with z(1) show ?thesis
          by (cases count Zs a ≤ count (mset xs) a)
            (blast dest: count_a not_le_imp_less trans[unfolded transp_def, rule_format])+
      next
        case False
        with ‹a ∉# K› have count Ys z ≤ count Zs z unfolding *
          by (auto simp add: not_in_iff)
        with z show ?thesis by auto
      qed
    qed
  qed
  ultimately show ?case
    unfolding msetext_huet_def Let_def by blast
qed

theorem msetext_huet_eq_dersh: irreflp gt ⟹ transp gt ⟹ msetext_dersh gt = msetext_huet gt
  using msetext_huet_imp_dersh msetext_dersh_imp_mult mult_imp_msetext_huet by fast

lemma msetext_huet_mono_strong:
  (∀ y ∈ set ys. ∀ x ∈ set xs. gt y x ⟶ gt' y x) ⟹ msetext_huet gt ys xs ⟹ msetext_huet gt' ys xs
  unfolding msetext_huet_def
  by (metis less_le_trans mem_Collect_eq not_le not_less0 set_mset_mset[unfolded set_mset_def])

lemma msetext_huet_map:
  assumes
    fin: finite A and
    ys_a: ys ∈ lists A and xs_a: xs ∈ lists A and
    irrefl_f: ∀ x ∈ A. ¬ gt (f x) (f x) and
    trans_f: ∀ z ∈ A. ∀ y ∈ A. ∀ x ∈ A. gt (f z) (f y) ⟶ gt (f y) (f x) ⟶ gt (f z) (f x) and
    compat_f: ∀ y ∈ A. ∀ x ∈ A. gt y x ⟶ gt (f y) (f x) and
    ys_gt_xs: msetext_huet gt ys xs
  shows msetext_huet gt (map f ys) (map f xs) (is msetext_huet _ ?fys ?fxs)
proof −
  have irrefl: ∀ x ∈ A. ¬ gt x x
    using irrefl_f compat_f by blast

  have
    ms_xs_ne_ys: mset xs ≠ mset ys and
    ex_gt: ∀ x. count (mset ys) x < count (mset xs) x ⟶
      (∃ y. gt y x ∧ count (mset xs) y < count (mset ys) y)
    using ys_gt_xs[unfolded msetext_huet_def Let_def] by blast+

  have ex_y: ∃ y. gt (f y) (f x) ∧ count (mset ?fxs) (f y) < count (mset (map f ys)) (f y)
    if cnt_x: count (mset xs) x > count (mset ys) x for x
  proof −
    have x_in_a: x ∈ A
      using cnt_x xs_a dual_order.strict_trans2 by fastforce

    obtain y where y_gt_x: gt y x and cnt_y: count (mset ys) y > count (mset xs) y
      using cnt_x ex_gt by blast
    have y_in_a: y ∈ A
      using cnt_y ys_a dual_order.strict_trans2 by fastforce

    have wf_gt_f: wfP (λy x. y ∈ A ∧ x ∈ A ∧ gt (f y) (f x))
      by (rule finite_irreflp_transp_imp_wfp)
        (auto elim: trans_f[rule_format] simp: fin irrefl_f Collect_case_prod_Sigma irreflp_def
          transp_def)

    obtain yy where
```

```
      fyy_gt_fx: gt (f yy) (f x) and
      cnt_yy: count (mset ys) yy > count (mset xs) yy and
      max_yy: ∀ y ∈ A. yy ∈ A ⟶ gt (f y) (f yy) ⟶ gt (f y) (f x) ⟶
        count (mset xs) y ≥ count (mset ys) y
      using wfP_eq_minimal[THEN iffD1, OF wf_gt_f, rule_format,
          of y {y. gt (f y) (f x) ∧ count (mset xs) y < count (mset ys) y}, simplified]
        y_gt_x cnt_y
      by (metis compat_f not_less x_in_a y_in_a)
    have yy_in_a: yy ∈ A
      using cnt_yy ys_a dual_order.strict_trans2 by fastforce

    {
      assume count (mset ?fxs) (f yy) ≥ count (mset ?fys) (f yy)
      then obtain u where fu_eq_fyy: f u = f yy and cnt_u: count (mset xs) u > count (mset ys) u
        using count_image_mset_le_imp_lt cnt_yy mset_map by (metis (mono_tags))
      have u_in_a: u ∈ A
        using cnt_u xs_a dual_order.strict_trans2 by fastforce

      obtain v where v_gt_u: gt v u and cnt_v: count (mset ys) v > count (mset xs) v
        using cnt_u ex_gt by blast
      have v_in_a: v ∈ A
        using cnt_v ys_a dual_order.strict_trans2 by fastforce

      have fv_gt_fu: gt (f v) (f u)
        using v_gt_u compat_f v_in_a u_in_a by blast
      hence fv_gt_fyy: gt (f v) (f yy)
        by (simp only: fu_eq_fyy)

      have gt (f v) (f x)
        using fv_gt_fyy fyy_gt_fx v_in_a yy_in_a x_in_a trans_f by blast
      hence False
        using max_yy[rule_format, of v] fv_gt_fyy v_in_a yy_in_a cnt_v by linarith
    }
    thus ?thesis
      using fyy_gt_fx leI by blast
  qed

  show ?thesis
    unfolding msetext_huet_def Let_def
  proof (intro conjI allI impI)
    {
      assume len_eq: length xs = length ys
      obtain x where cnt_x: count (mset xs) x > count (mset ys) x
        using len_eq ms_xs_ne_ys by (metis size_eq_ex_count_lt size_mset)
      hence mset ?fxs ≠ mset ?fys
        using ex_y by fastforce
    }
    thus mset ?fxs ≠ mset (map f ys)
      by (metis length_map size_mset)
  next
    fix fx
    assume cnt_fx: count (mset ?fxs) fx > count (mset ?fys) fx
    then obtain x where fx: fx = f x and cnt_x: count (mset xs) x > count (mset ys) x
      using count_image_mset_lt_imp_lt mset_map by (metis (mono_tags))
    thus ∃ fy. gt fy fx ∧ count (mset ?fxs) fy < count (mset (map f ys)) fy
      using ex_y[OF cnt_x] by blast
  qed
qed

lemma msetext_huet_irrefl: (∀ x ∈ set xs. ¬ gt x x) ⟹ ¬ msetext_huet gt xs xs
  unfolding msetext_huet_def by simp

lemma msetext_huet_trans_from_irrefl:
```

```
  assumes
    fin: finite A and
    zs_a: zs ∈ lists A and ys_a: ys ∈ lists A and xs_a: xs ∈ lists A and
    irrefl: ∀ x ∈ A. ¬ gt x x and
    trans: ∀ z ∈ A. ∀ y ∈ A. ∀ x ∈ A. gt z y ⟶ gt y x ⟶ gt z x and
    zs_gt_ys: msetext_huet gt zs ys and
    ys_gt_xs: msetext_huet gt ys xs
  shows msetext_huet gt zs xs
proof −
  have wf_gt: wfP (λy x. y ∈ A ∧ x ∈ A ∧ gt y x)
    by (rule finite_irreflp_transp_imp_wfp)
      (auto elim: trans[rule_format] simp: fin irrefl Collect_case_prod_Sigma irreflp_def
        transp_def)

  show ?thesis
    unfolding msetext_huet_def Let_def
  proof (intro conjI allI impI)
    obtain x where cnt_x: count (mset zs) x > count (mset ys) x
      using msetext_huet_imp_count_gt[OF zs_gt_ys] by blast
    have x_in_a: x ∈ A
      using cnt_x zs_a dual_order.strict_trans2 by fastforce

    obtain xx where
      cnt_xx: count (mset zs) xx > count (mset ys) xx and
      max_xx: ∀ y ∈ A. xx ∈ A ⟶ gt y xx ⟶ count (mset ys) y ≥ count (mset zs) y
      using wfP_eq_minimal[THEN iffD1, OF wf_gt, rule_format,
          of x {y. count (mset ys) y < count (mset zs) y}, simplified]
        cnt_x
      by force
    have xx_in_a: xx ∈ A
      using cnt_xx zs_a dual_order.strict_trans2 by fastforce

    show mset xs ≠ mset zs
    proof (cases count (mset ys) xx ≥ count (mset xs) xx)
      case True
      thus ?thesis
        using cnt_xx by fastforce
    next
      case False
      hence count (mset ys) xx < count (mset xs) xx
        by fastforce
      then obtain z where z_gt_xx: gt z xx and cnt_z: count (mset ys) z > count (mset xs) z
        using ys_gt_xs[unfolded msetext_huet_def Let_def] by blast
      have z_in_a: z ∈ A
        using cnt_z ys_a dual_order.strict_trans2 by fastforce

      have count (mset zs) z ≤ count (mset ys) z
        using max_xx[rule_format, of z] z_in_a xx_in_a z_gt_xx by blast
      moreover
      {
        assume count (mset zs) z < count (mset ys) z
        then obtain u where u_gt_z: gt u z and cnt_u: count (mset ys) u < count (mset zs) u
          using zs_gt_ys[unfolded msetext_huet_def Let_def] by blast
        have u_in_a: u ∈ A
          using cnt_u zs_a dual_order.strict_trans2 by fastforce
        have u_gt_xx: gt u xx
          using trans u_in_a z_in_a xx_in_a u_gt_z z_gt_xx by blast
        have False
          using max_xx[rule_format, of u] u_in_a xx_in_a u_gt_xx cnt_u by fastforce
      }
      ultimately have count (mset zs) z = count (mset ys) z
        by fastforce
      thus ?thesis
```

```
    using cnt_z by fastforce
  qed
next
  fix x
  assume cnt_x_xz: count (mset zs) x < count (mset xs) x
  have x_in_a: x ∈ A
    using cnt_x_xz xs_a dual_order.strict_trans2 by fastforce

  let ?case = ∃ y. gt y x ∧ count (mset zs) y > count (mset xs) y

  {
    assume cnt_x: count (mset zs) x < count (mset ys) x
    then obtain y where y_gt_x: gt y x and cnt_y: count (mset zs) y > count (mset ys) y
      using zs_gt_ys[unfolded msetext_huet_def Let_def] by blast
    have y_in_a: y ∈ A
      using cnt_y zs_a dual_order.strict_trans2 by fastforce

    obtain yy where
      yy_gt_x: gt yy x and
      cnt_yy: count (mset zs) yy > count (mset ys) yy and
      max_yy: ∀ y ∈ A. yy ∈ A ⟶ gt y yy ⟶ gt y x ⟶ count (mset ys) y ≥ count (mset zs) y
      using wfP_eq_minimal[THEN iffD1, OF wf_gt, rule_format,
          of y {y. gt y x ∧ count (mset ys) y < count (mset zs) y}, simplified]
        y_gt_x cnt_y
      by force
    have yy_in_a: yy ∈ A
      using cnt_yy zs_a dual_order.strict_trans2 by fastforce

    have ?case
    proof (cases count (mset ys) yy ≥ count (mset xs) yy)
      case True
      thus ?thesis
        using yy_gt_x cnt_yy by fastforce
    next
      case False
      hence count (mset ys) yy < count (mset xs) yy
        by fastforce
      then obtain z where z_gt_yy: gt z yy and cnt_z: count (mset ys) z > count (mset xs) z
        using ys_gt_xs[unfolded msetext_huet_def Let_def] by blast
      have z_in_a: z ∈ A
        using cnt_z ys_a dual_order.strict_trans2 by fastforce
      have z_gt_x: gt z x
        using trans z_in_a yy_in_a x_in_a z_gt_yy yy_gt_x by blast

      have count (mset zs) z ≤ count (mset ys) z
        using max_yy[rule_format, of z] z_in_a yy_in_a z_gt_yy z_gt_x by blast
      moreover
      {
        assume count (mset zs) z < count (mset ys) z
        then obtain u where u_gt_z: gt u z and cnt_u: count (mset ys) u < count (mset zs) u
          using zs_gt_ys[unfolded msetext_huet_def Let_def] by blast
        have u_in_a: u ∈ A
          using cnt_u zs_a dual_order.strict_trans2 by fastforce
        have u_gt_yy: gt u yy
          using trans u_in_a z_in_a yy_in_a u_gt_z z_gt_yy by blast
        have u_gt_x: gt u x
          using trans u_in_a z_in_a x_in_a u_gt_z z_gt_x by blast
        have False
          using max_yy[rule_format, of u] u_in_a yy_in_a u_gt_yy u_gt_x cnt_u by fastforce
      }
      ultimately have count (mset zs) z = count (mset ys) z
        by fastforce
      thus ?thesis
```

```
    using z_gt_x cnt_z by fastforce
  qed
}
moreover
{
  assume count (mset zs) x ≥ count (mset ys) x
  hence count (mset ys) x < count (mset xs) x
    using cnt_x_xz by fastforce
  then obtain y where y_gt_x: gt y x and cnt_y: count (mset ys) y > count (mset xs) y
    using ys_gt_xs[unfolded msetext_huet_def Let_def] by blast
  have y_in_a: y ∈ A
    using cnt_y ys_a dual_order.strict_trans2 by fastforce

  obtain yy where
    yy_gt_x: gt yy x and
    cnt_yy: count (mset ys) yy > count (mset xs) yy and
    max_yy: ∀ y ∈ A. yy ∈ A ⟶ gt y yy ⟶ gt y x ⟶ count (mset xs) y ≥ count (mset ys) y
    using wfP_eq_minimal[THEN iffD1, OF wf_gt, rule_format,
        of y {y. gt y x ∧ count (mset xs) y < count (mset ys) y}, simplified]
      y_gt_x cnt_y
    by force
  have yy_in_a: yy ∈ A
    using cnt_yy ys_a dual_order.strict_trans2 by fastforce

  have ?case
  proof (cases count (mset zs) yy ≥ count (mset ys) yy)
    case True
    thus ?thesis
      using yy_gt_x cnt_yy by fastforce
  next
    case False
    hence count (mset zs) yy < count (mset ys) yy
      by fastforce
    then obtain z where z_gt_yy: gt z yy and cnt_z: count (mset zs) z > count (mset ys) z
      using zs_gt_ys[unfolded msetext_huet_def Let_def] by blast
    have z_in_a: z ∈ A
      using cnt_z zs_a dual_order.strict_trans2 by fastforce
    have z_gt_x: gt z x
      using trans z_in_a yy_in_a x_in_a z_gt_yy yy_gt_x by blast

    have count (mset ys) z ≤ count (mset xs) z
      using max_yy[rule_format, of z] z_in_a yy_in_a z_gt_yy z_gt_x by blast
    moreover
    {
      assume count (mset ys) z < count (mset xs) z
      then obtain u where u_gt_z: gt u z and cnt_u: count (mset xs) u < count (mset ys) u
        using ys_gt_xs[unfolded msetext_huet_def Let_def] by blast
      have u_in_a: u ∈ A
        using cnt_u ys_a dual_order.strict_trans2 by fastforce
      have u_gt_yy: gt u yy
        using trans u_in_a z_in_a yy_in_a u_gt_z z_gt_yy by blast
      have u_gt_x: gt u x
        using trans u_in_a z_in_a x_in_a u_gt_z z_gt_x by blast
      have False
        using max_yy[rule_format, of u] u_in_a yy_in_a u_gt_yy u_gt_x cnt_u by fastforce
    }
    ultimately have count (mset ys) z = count (mset xs) z
      by fastforce
    thus ?thesis
      using z_gt_x cnt_z by fastforce
  qed
}
ultimately show ∃ y. gt y x ∧ count (mset xs) y < count (mset zs) y
```

```
    by fastforce
  qed
qed

lemma msetext_huet_snoc: msetext_huet gt (xs @ [x]) xs
  unfolding msetext_huet_def Let_def by simp

lemma msetext_huet_compat_cons: msetext_huet gt ys xs ⟹ msetext_huet gt (x # ys) (x # xs)
  unfolding msetext_huet_def Let_def by auto

lemma msetext_huet_compat_snoc: msetext_huet gt ys xs ⟹ msetext_huet gt (ys @ [x]) (xs @ [x])
  unfolding msetext_huet_def Let_def by auto

lemma msetext_huet_compat_list: y ≠ x ⟹ gt y x ⟹ msetext_huet gt (xs @ y # xs') (xs @ x # xs')
  unfolding msetext_huet_def Let_def by auto

lemma msetext_huet_singleton: y ≠ x ⟹ msetext_huet gt [y] [x] ⟷ gt y x
  unfolding msetext_huet_def by simp

lemma msetext_huet_wf: wfP (λx y. gt y x) ⟹ wfP (λxs ys. msetext_huet gt ys xs)
  by (erule wfP_subset[OF msetext_dersh_wf]) (auto intro: msetext_huet_imp_dersh)

lemma msetext_huet_hd_or_tl:
  assumes
    trans: ∀ z y x. gt z y ⟶ gt y x ⟶ gt z x and
    total: ∀ y x. gt y x ∨ gt x y ∨ y = x and
    len_eq: length ys = length xs and
    yys_gt_xxs: msetext_huet gt (y # ys) (x # xs)
  shows gt y x ∨ msetext_huet gt ys xs
proof −
  let ?Y = mset (y # ys)
  let ?X = mset (x # xs)

  let ?Ya = mset ys
  let ?Xa = mset xs

  have Y_ne_X: ?Y ≠ ?X and
    ex_gt_Y: ⋀xa. count ?X xa > count ?Y xa ⟹ ∃ ya. gt ya xa ∧ count ?Y ya > count ?X ya
    using yys_gt_xxs[unfolded msetext_huet_def Let_def] by auto
  obtain yy where
    yy: ⋀xa. count ?X xa > count ?Y xa ⟹ gt (yy xa) xa ∧ count ?Y (yy xa) > count ?X (yy xa)
    using ex_gt_Y by metis

  have cnt_Y_pres: count ?Ya xa > count ?Xa xa if count ?Y xa > count ?X xa and xa ≠ y for xa
    using that by (auto split: if_splits)
  have cnt_X_pres: count ?Xa xa > count ?Ya xa if count ?X xa > count ?Y xa and xa ≠ x for xa
    using that by (auto split: if_splits)

  {
    assume y_eq_x: y = x
    have ?Xa ≠ ?Ya
      using y_eq_x Y_ne_X by simp
    moreover have ⋀xa. count ?Xa xa > count ?Ya xa ⟹ ∃ ya. gt ya xa ∧ count ?Ya ya > count ?Xa ya
    proof −
      fix xa :: 'a
      assume a1: count (mset ys) xa < count (mset xs) xa
      from ex_gt_Y obtain aa :: 'a ⇒ 'a where
        f3: ∀ a. ¬ count (mset (y # ys)) a < count (mset (x # xs)) a ∨ gt (aa a) a ∧
          count (mset (x # xs)) (aa a) < count (mset (y # ys)) (aa a)
        by (metis (full_types))
      then have f4: ⋀a. count (mset (x # xs)) (aa a) < count (mset (x # ys)) (aa a) ∨
          ¬ count (mset (x # ys)) a < count (mset (x # xs)) a
        using y_eq_x by meson
```

```
    have ⋀a as aa. count (mset ((a::'a) # as)) aa = count (mset as) aa ∨ aa = a
      by fastforce
    then have xa = x ∨ count (mset (x # xs)) (aa xa) < count (mset (x # ys)) (aa xa)
      using f4 a1 by (metis (no_types))
    then show ∃ a. gt a xa ∧ count (mset xs) a < count (mset ys) a
      using f3 y_eq_x a1 by (metis (no_types) Suc_less_eq count_add_mset mset.simps(2))
  qed
  ultimately have msetext_huet gt ys xs
    unfolding msetext_huet_def Let_def by simp
}
moreover
{
  assume x_gt_y: gt x y and y_ngt_x: ¬ gt y x
  hence y_ne_x: y ≠ x
    by fast

  obtain z where z_cnt: count ?X z > count ?Y z
    using size_eq_ex_count_lt[of ?Y ?X] size_mset size_mset len_eq Y_ne_X by auto

  have Xa_ne_Ya: ?Xa ≠ ?Ya
  proof (cases z = x)
    case True
    hence yy z ≠ y
      using y_ngt_x yy z_cnt by blast
    hence count ?Ya (yy z) > count ?Xa (yy z)
      using cnt_Y_pres yy z_cnt by blast
    thus ?thesis
      by auto
  next
    case False
    hence count ?Xa z > count ?Ya z
      using z_cnt cnt_X_pres by blast
    thus ?thesis
      by auto
  qed

  have ∃ ya. gt ya xa ∧ count ?Ya ya > count ?Xa ya
    if xa_cnta: count ?Xa xa > count ?Ya xa for xa
  proof (cases xa = y)
    case xa_eq_y: True

    {
      assume count ?Ya x > count ?Xa x
      moreover have gt x xa
        unfolding xa_eq_y by (rule x_gt_y)
      ultimately have ?thesis
        by fast
    }
    moreover
    {
      assume count ?Xa x ≥ count ?Ya x
      hence x_cnt: count ?X x > count ?Y x
        by (simp add: y_ne_x)
      hence yyx_gt_x: gt (yy x) x and yyx_cnt: count ?Y (yy x) > count ?X (yy x)
        using yy by blast+

      have yyx_ne_y: yy x ≠ y
        using y_ngt_x yyx_gt_x by auto

      have gt (yy x) xa
        unfolding xa_eq_y using trans yyx_gt_x x_gt_y by blast
      moreover have count ?Ya (yy x) > count ?Xa (yy x)
        using cnt_Y_pres yyx_cnt yyx_ne_y by blast
```

```
      ultimately have ?thesis
        by blast
    }
    ultimately show ?thesis
      by fastforce
  next
    case False
    hence xa_cnt: count ?X xa > count ?Y xa
      using xa_cnta by fastforce

    show ?thesis
    proof (cases yy xa = y ∧ count ?Ya y ≤ count ?Xa y)
      case yyxa_ne_y_or: False

      have yyxa_gt_xa: gt (yy xa) xa and yyxa_cnt: count ?Y (yy xa) > count ?X (yy xa)
        using yy[OF xa_cnt] by blast+

      have count ?Ya (yy xa) > count ?Xa (yy xa)
        using cnt_Y_pres yyxa_cnt yyxa_ne_y_or by fastforce
      thus ?thesis
        using yyxa_gt_xa by blast
    next
      case True
      note yyxa_eq_y = this[THEN conjunct1] and y_cnt = this[THEN conjunct2]

      {
        assume count ?Ya x > count ?Xa x
        moreover have gt x xa
          using trans x_gt_y xa_cnt yy yyxa_eq_y by blast
        ultimately have ?thesis
          by fast
      }
      moreover
      {
        assume count ?Xa x ≥ count ?Ya x
        hence x_cnt: count ?X x > count ?Y x
          by (simp add: y_ne_x)
        hence yyx_gt_x: gt (yy x) x and yyx_cnt: count ?Y (yy x) > count ?X (yy x)
          using yy by blast+

        have yyx_ne_y: yy x ≠ y
          using y_ngt_x yyx_gt_x by auto

        have gt (yy x) xa
          using trans x_gt_y xa_cnt yy yyx_gt_x yyxa_eq_y by blast
        moreover have count ?Ya (yy x) > count ?Xa (yy x)
          using cnt_Y_pres yyx_cnt yyx_ne_y by blast
        ultimately have ?thesis
          by blast
      }
      ultimately show ?thesis
        by fastforce
    qed
  qed
  hence msetext_huet gt ys xs
    unfolding msetext_huet_def Let_def using Xa_ne_Ya by fast
 }
 ultimately show ?thesis
   using total by blast
qed

interpretation msetext_huet: ext msetext_huet
  by standard (fact msetext_huet_mono_strong, fact msetext_huet_map)
```

interpretation *msetext_huet*: *ext_irrefl_before_trans msetext_huet*
by *standard* (*fact msetext_huet_irrefl*, *fact msetext_huet_trans_from_irrefl*)

interpretation *msetext_huet*: *ext_snoc msetext_huet*
by *standard* (*fact msetext_huet_snoc*)

interpretation *msetext_huet*: *ext_compat_cons msetext_huet*
by *standard* (*fact msetext_huet_compat_cons*)

interpretation *msetext_huet*: *ext_compat_snoc msetext_huet*
by *standard* (*fact msetext_huet_compat_snoc*)

interpretation *msetext_huet*: *ext_compat_list msetext_huet*
by *standard* (*fact msetext_huet_compat_list*)

interpretation *msetext_huet*: *ext_singleton msetext_huet*
by *standard* (*fact msetext_huet_singleton*)

interpretation *msetext_huet*: *ext_wf msetext_huet*
by *standard* (*fact msetext_huet_wf*)

interpretation *msetext_huet*: *ext_hd_or_tl msetext_huet*
by *standard* (*rule msetext_huet_hd_or_tl*)

interpretation *msetext_huet*: *ext_wf_bounded msetext_huet*
by *standard*

5.9 Componentwise Extension

definition *cwiseext* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a list* ⇒ *'a list* ⇒ *bool* **where**
cwiseext gt ys xs ⟷ *length ys* = *length xs*
∧ (∀ *i* < *length ys*. *gt* (*ys* ! *i*) (*xs* ! *i*) ∨ *ys* ! *i* = *xs* ! *i*)
∧ (∃ *i* < *length ys*. *gt* (*ys* ! *i*) (*xs* ! *i*))

lemma *cwiseext_imp_len_lexext*:
assumes *cw*: *cwiseext gt ys xs*
shows *len_lexext gt ys xs*
proof −
have *len_eq*: *length ys* = *length xs*
using *cw*[*unfolded cwiseext_def*] **by** *sat*
moreover have *lexext gt ys xs*
proof −
obtain *j* **where**
j_len: *j* < *length ys* **and**
j_gt: *gt* (*ys* ! *j*) (*xs* ! *j*)
using *cw*[*unfolded cwiseext_def*] **by** *blast*
then obtain *j0* **where**
j0_len: *j0* < *length ys* **and**
j0_gt: *gt* (*ys* ! *j0*) (*xs* ! *j0*) **and**
j0_min: ⋀*i*. *i* < *j0* ⟹ ¬ *gt* (*ys* ! *i*) (*xs* ! *i*)
using *wf_eq_minimal*[*THEN iffD1*, *OF wf_less*, *rule_format*, *of* _ {*i*. *gt* (*ys* ! *i*) (*xs* ! *i*)},
simplified, *OF j_gt*]
by (*metis less_trans nat_neq_iff*)

have *j0_eq*: ⋀*i*. *i* < *j0* ⟹ *ys* ! *i* = *xs* ! *i*
using *cw*[*unfolded cwiseext_def*] **by** (*metis j0_len j0_min less_trans*)

have *lexext gt* (*drop j0 ys*) (*drop j0 xs*)
using *lexext_Cons*[*of gt* _ _ *drop* (*Suc j0*) *ys drop* (*Suc j0*) *xs*, *OF j0_gt*]
by (*metis Cons_nth_drop_Suc j0_len len_eq*)
thus *?thesis*
using *cw len_eq j0_len j0_min*
proof (*induct j0 arbitrary*: *ys xs*)

```
      case (Suc k)
      note ih0 = this(1) and gts_dropSk = this(2) and cw = this(3) and len_eq = this(4) and
        Sk_len = this(5) and Sk_min = this(6)

      have Sk_eq: ⋀i. i < Suc k ⟹ ys ! i = xs ! i
        using cw[unfolded cwiseext_def] by (metis Sk_len Sk_min less_trans)

      have k_len: k < length ys
        using Sk_len by simp
      have k_min: ⋀i. i < k ⟹ ¬ gt (ys ! i) (xs ! i)
        using Sk_min by simp

      have k_eq: ⋀i. i < k ⟹ ys ! i = xs ! i
        using Sk_eq by simp

      note ih = ih0[OF _ cw len_eq k_len k_min]

      show ?case
      proof (cases k < length ys)
        case k_lt_ys: True
        note k_lt_xs = k_lt_ys[unfolded len_eq]

        obtain x where x: x = xs ! k
          by simp
        hence y: x = ys ! k
          using Sk_eq[of k] by simp

        have dropk_xs: drop k xs = x # drop (Suc k) xs
          using k_lt_xs x by (simp add: Cons_nth_drop_Suc)
        have dropk_ys: drop k ys = x # drop (Suc k) ys
          using k_lt_ys y by (simp add: Cons_nth_drop_Suc)

        show ?thesis
          by (rule ih, unfold dropk_xs dropk_ys, rule lexext_Cons_eq[OF gts_dropSk])
      next
        case False
        hence drop k xs = [] and drop k ys = []
          using len_eq by simp_all
        hence lexext gt [] []
          using gts_dropSk by simp
        hence lexext gt (drop k ys) (drop k xs)
          by simp
        thus ?thesis
          by (rule ih)
      qed
    qed simp
  qed
  ultimately show ?thesis
    unfolding lenext_def by sat
qed

lemma cwiseext_mono_strong:
  (∀ y ∈ set ys. ∀ x ∈ set xs. gt y x ⟶ gt' y x) ⟹ cwiseext gt ys xs ⟹ cwiseext gt' ys xs
  unfolding cwiseext_def by (induct, force, fast)

lemma cwiseext_map_strong:
  (∀ y ∈ set ys. ∀ x ∈ set xs. gt y x ⟶ gt (f y) (f x)) ⟹ cwiseext gt ys xs ⟹
   cwiseext gt (map f ys) (map f xs)
  unfolding cwiseext_def by auto

lemma cwiseext_irrefl: (∀ x ∈ set xs. ¬ gt x x) ⟹ ¬ cwiseext gt xs xs
  unfolding cwiseext_def by (blast intro: nth_mem)
```

lemma *cwiseext_trans_strong*:
assumes
$\forall z \in set\ zs.\ \forall y \in set\ ys.\ \forall x \in set\ xs.\ gt\ z\ y \longrightarrow gt\ y\ x \longrightarrow gt\ z\ x$ **and**
cwiseext gt zs ys **and** *cwiseext gt ys xs*
shows *cwiseext gt zs xs*
using *assms* **unfolding** *cwiseext_def* **by** (*metis* (*mono_tags*) *nth_mem*)

lemma *cwiseext_compat_cons*: *cwiseext gt ys xs* $\Longrightarrow$ *cwiseext gt* (*x* # *ys*) (*x* # *xs*)
unfolding *cwiseext_def*
proof (*elim conjE*, *intro conjI*)
assume
length ys = *length xs* **and**
$\forall i < length\ ys.\ gt\ (ys\ !\ i)\ (xs\ !\ i) \vee ys\ !\ i = xs\ !\ i$
thus $\forall i < length\ (x \# ys).\ gt\ ((x \# ys)\ !\ i)\ ((x \# xs)\ !\ i) \vee (x \# ys)\ !\ i = (x \# xs)\ !\ i$
by (*simp add*: *nth_Cons'*)
next
assume $\exists i < length\ ys.\ gt\ (ys\ !\ i)\ (xs\ !\ i)$
thus $\exists i < length\ (x \# ys).\ gt\ ((x \# ys)\ !\ i)\ ((x \# xs)\ !\ i)$
by *fastforce*
qed *auto*

lemma *cwiseext_compat_snoc*: *cwiseext gt ys xs* $\Longrightarrow$ *cwiseext gt* (*ys* @ [*x*]) (*xs* @ [*x*])
unfolding *cwiseext_def*
proof (*elim conjE*, *intro conjI*)
assume
length ys = *length xs* **and**
$\forall i < length\ ys.\ gt\ (ys\ !\ i)\ (xs\ !\ i) \vee ys\ !\ i = xs\ !\ i$
thus $\forall i < length\ (ys\ @\ [x]).$
$gt\ ((ys\ @\ [x])\ !\ i)\ ((xs\ @\ [x])\ !\ i) \vee (ys\ @\ [x])\ !\ i = (xs\ @\ [x])\ !\ i$
by (*simp add*: *nth_append*)
next
assume
length ys = *length xs* **and**
$\exists i < length\ ys.\ gt\ (ys\ !\ i)\ (xs\ !\ i)$
thus $\exists i < length\ (ys\ @\ [x]).\ gt\ ((ys\ @\ [x])\ !\ i)\ ((xs\ @\ [x])\ !\ i)$
by (*metis length_append_singleton less_Suc_eq nth_append*)
qed *auto*

lemma *cwiseext_compat_list*:
assumes *y_gt_x*: *gt y x*
shows *cwiseext gt* (*xs* @ *y* # *xs'*) (*xs* @ *x* # *xs'*)
unfolding *cwiseext_def*
proof (*intro conjI*)
show $\forall i < length\ (xs\ @\ y \# xs').\ gt\ ((xs\ @\ y \# xs')\ !\ i)\ ((xs\ @\ x \# xs')\ !\ i)$
$\vee\ (xs\ @\ y \# xs')\ !\ i = (xs\ @\ x \# xs')\ !\ i$
using *y_gt_x* **by** (*simp add*: *nth_Cons' nth_append*)
next
show $\exists i < length\ (xs\ @\ y \# xs').\ gt\ ((xs\ @\ y \# xs')\ !\ i)\ ((xs\ @\ x \# xs')\ !\ i)$
using *y_gt_x* **by** (*metis add_diff_cancel_right' append_is_Nil_conv diff_less length_append*
length_greater_0_conv list.simps(*3*) *nth_append_length*)
qed *auto*

lemma *cwiseext_singleton*: *cwiseext gt* [*y*] [*x*] $\longleftrightarrow$ *gt y x*
unfolding *cwiseext_def* **by** *auto*

lemma *cwiseext_wf*: *wfP* ($\lambda x\ y.\ gt\ y\ x$) $\Longrightarrow$ *wfP* ($\lambda xs\ ys.\ cwiseext\ gt\ ys\ xs$)
by (*auto intro*: *cwiseext_imp_len_lexext wfP_subset*[*OF len_lexext_wf*])

lemma *cwiseext_hd_or_tl*: *cwiseext gt* (*y* # *ys*) (*x* # *xs*) $\Longrightarrow$ *gt y x* $\vee$ *cwiseext gt ys xs*
unfolding *cwiseext_def*
proof (*elim conjE*, *intro disj_imp*[*THEN iffD2*, *rule_format*] *conjI*)
assume
$\exists i < length\ (y \# ys).\ gt\ ((y \# ys)\ !\ i)\ ((x \# xs)\ !\ i)$ **and**

```
    ¬ gt y x
  thus ∃ i < length ys. gt (ys ! i) (xs ! i)
    by (metis (no_types) One_nat_def diff_le_self diff_less dual_order.strict_trans2
      length_Cons less_Suc_eq linorder_neqE_nat not_less0 nth_Cons')
qed auto

locale ext_cwiseext = ext_compat_list + ext_compat_cons
begin

context
  fixes gt :: 'a ⇒ 'a ⇒ bool
  assumes
    gt_irrefl: ¬ gt x x and
    trans_gt: ext gt zs ys ⟹ ext gt ys xs ⟹ ext gt zs xs
begin

lemma
  assumes ys_gtcw_xs: cwiseext gt ys xs
  shows ext gt ys xs
proof −
  have length ys = length xs
    by (rule ys_gtcw_xs[unfolded cwiseext_def, THEN conjunct1])
  thus ?thesis
    using ys_gtcw_xs
  proof (induct rule: list_induct2)
    case Nil
    thus ?case
      unfolding cwiseext_def by simp
  next
    case (Cons y ys x xs)
    note len_ys_eq_xs = this(1) and ih = this(2) and yys_gtcw_xxs = this(3)

    have xys_gts_xxs: ext gt (x # ys) (x # xs) if ys_ne_xs: ys ≠ xs
    proof −
      have ys_gtcw_xs: cwiseext gt ys xs
        using yys_gtcw_xxs unfolding cwiseext_def
      proof (elim conjE, intro conjI)
        assume
          ∀ i < length (y # ys). gt ((y # ys) ! i) ((x # xs) ! i) ∨ (y # ys) ! i = (x # xs) ! i
        hence ge: ∀ i < length ys. gt (ys ! i) (xs ! i) ∨ ys ! i = xs ! i
          by auto
        thus ∃ i < length ys. gt (ys ! i) (xs ! i)
          using ys_ne_xs len_ys_eq_xs nth_equalityI by blast
      qed auto
      hence ext gt ys xs
        by (rule ih)
      thus ext gt (x # ys) (x # xs)
        by (rule compat_cons)
    qed

    have gt y x ∨ y = x
      using yys_gtcw_xxs unfolding cwiseext_def by fastforce
    moreover
    {
      assume y_eq_x: y = x
      have ?case
      proof (cases ys = xs)
        case True
        hence False
          using y_eq_x gt_irrefl yys_gtcw_xxs unfolding cwiseext_def by presburger
        thus ?thesis
          by sat
      next
```

```
      case False
      thus ?thesis
        using y_eq_x xys_gts_xxs by simp
    qed
  }
  moreover
  {
    assume y ≠ x and gt y x
    hence yys_gts_xys: ext gt (y # ys) (x # ys)
      using compat_list[of _ _ gt []] by simp

    have ?case
    proof (cases ys = xs)
      case ys_eq_xs: True
      thus ?thesis
        using yys_gts_xys by simp
    next
      case False
      thus ?thesis
        using yys_gts_xys xys_gts_xxs trans_gt by blast
    qed
  }
  ultimately show ?case
    by sat
 qed
qed

end

end

interpretation cwiseext: ext cwiseext
  by standard (fact cwiseext_mono_strong, rule cwiseext_map_strong, metis in_listsD)

interpretation cwiseext: ext_irrefl_trans_strong cwiseext
  by standard (fact cwiseext_irrefl, fact cwiseext_trans_strong)

interpretation cwiseext: ext_compat_cons cwiseext
  by standard (fact cwiseext_compat_cons)

interpretation cwiseext: ext_compat_snoc cwiseext
  by standard (fact cwiseext_compat_snoc)

interpretation cwiseext: ext_compat_list cwiseext
  by standard (rule cwiseext_compat_list)

interpretation cwiseext: ext_singleton cwiseext
  by standard (rule cwiseext_singleton)

interpretation cwiseext: ext_wf cwiseext
  by standard (rule cwiseext_wf)

interpretation cwiseext: ext_hd_or_tl cwiseext
  by standard (rule cwiseext_hd_or_tl)

interpretation cwiseext: ext_wf_bounded cwiseext
  by standard

end
```

6 The Applicative Recursive Path Order for Lambda-Free Higher-Order Terms

theory *Lambda_Free_RPO_App*
imports *Lambda_Free_Term Extension_Orders*
abbrevs $>t = >_t$
and $\geq t = \geq_t$
begin

This theory defines the applicative recursive path order (RPO), a variant of RPO for λ-free higher-order terms. It corresponds to the order obtained by applying the standard first-order RPO on the applicative encoding of higher-order terms and assigning the lowest precedence to the application symbol.

locale *rpo_app* = *gt_sym* $(>_s)$
for *gt_sym* :: $'s \Rightarrow 's \Rightarrow$ *bool* (**infix** $>_s$ *50*) +
fixes *ext* :: $(('s, 'v)\ tm \Rightarrow ('s, 'v)\ tm \Rightarrow bool) \Rightarrow ('s, 'v)\ tm\ list \Rightarrow ('s, 'v)\ tm\ list \Rightarrow bool$
assumes
ext_ext_trans_before_irefl: *ext_trans_before_irefl ext* **and**
ext_ext_compat_list: *ext_compat_list ext*
begin

lemma *ext_mono*[*mono*]: $gt \leq gt' \Longrightarrow ext\ gt \leq ext\ gt'$
by (*simp add*: *ext.mono ext_ext_compat_list*[*unfolded ext_compat_list_def*, *THEN conjunct1*])

inductive *gt* :: $('s, 'v)\ tm \Rightarrow ('s, 'v)\ tm \Rightarrow bool$ (**infix** $>_t$ *50*) **where**
gt_sub: $is\_App\ t \Longrightarrow (fun\ t >_t s \vee fun\ t = s) \vee (arg\ t >_t s \vee arg\ t = s) \Longrightarrow t >_t s$
| *gt_sym_sym*: $g >_s f \Longrightarrow Hd\ (Sym\ g) >_t Hd\ (Sym\ f)$
| *gt_sym_app*: $Hd\ (Sym\ g) >_t s1 \Longrightarrow Hd\ (Sym\ g) >_t s2 \Longrightarrow Hd\ (Sym\ g) >_t App\ s1\ s2$
| *gt_app_app*: $ext\ (>_t)\ [t1, t2]\ [s1, s2] \Longrightarrow App\ t1\ t2 >_t s1 \Longrightarrow App\ t1\ t2 >_t s2 \Longrightarrow$
$App\ t1\ t2 >_t App\ s1\ s2$

abbreviation *ge* :: $('s, 'v)\ tm \Rightarrow ('s, 'v)\ tm \Rightarrow bool$ (**infix** $\geq_t$ *50*) **where**
$t \geq_t s \equiv t >_t s \vee t = s$

end

end

7 The Graceful Recursive Path Order for Lambda-Free Higher-Order Terms

theory *Lambda_Free_RPO_Std*
imports *Lambda_Free_Term Extension_Orders Nested_Multisets_Ordinals.Multiset_More*
abbrevs $>t = >_t$
and $\geq t = \geq_t$
begin

This theory defines the graceful recursive path order (RPO) for λ-free higher-order terms.

7.1 Setup

locale *rpo_basis* = *ground_heads* $(>_s)$ *arity_sym arity_var*
for
gt_sym :: $'s \Rightarrow 's \Rightarrow$ *bool* (**infix** $>_s$ *50*) **and**
arity_sym :: $'s \Rightarrow$ *enat* **and**
arity_var :: $'v \Rightarrow$ *enat* +
fixes
extf :: $'s \Rightarrow (('s, 'v)\ tm \Rightarrow ('s, 'v)\ tm \Rightarrow bool) \Rightarrow ('s, 'v)\ tm\ list \Rightarrow ('s, 'v)\ tm\ list \Rightarrow bool$
assumes
extf_ext_trans_before_irefl: *ext_trans_before_irefl* (*extf f*) **and**
extf_ext_compat_cons: *ext_compat_cons* (*extf f*) **and**
extf_ext_compat_list: *ext_compat_list* (*extf f*)

begin

lemma *extf_ext_trans*: *ext_trans* (*extf f*)
  by (*rule ext_trans_before_irrefl.axioms*(*1*)[*OF extf_ext_trans_before_irrefl*])

lemma *extf_ext*: *ext* (*extf f*)
  by (*rule ext_trans.axioms*(*1*)[*OF extf_ext_trans*])

lemmas *extf_mono_strong* = *ext.mono_strong*[*OF extf_ext*]
lemmas *extf_mono* = *ext.mono*[*OF extf_ext*, *mono*]
lemmas *extf_map* = *ext.map*[*OF extf_ext*]
lemmas *extf_trans* = *ext_trans.trans*[*OF extf_ext_trans*]
lemmas *extf_irrefl_from_trans* =
  ext_trans_before_irrefl.irrefl_from_trans[*OF extf_ext_trans_before_irrefl*]
lemmas *extf_compat_append_left* = *ext_compat_cons.compat_append_left*[*OF extf_ext_compat_cons*]
lemmas *extf_compat_list* = *ext_compat_list.compat_list*[*OF extf_ext_compat_list*]

definition *chkvar* :: ('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm* ⇒ *bool* **where**
  [*simp*]: *chkvar t s* ⟷ *vars_hd* (*head s*) ⊆ *vars t*

end

locale *rpo* = *rpo_basis* _ _ *arity_sym arity_var*
  for
    arity_sym :: '*s* ⇒ *enat* **and**
    arity_var :: '*v* ⇒ *enat*
begin

7.2 Inductive Definitions

definition
  chksubs :: (('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm* ⇒ *bool*) ⇒ ('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm* ⇒ *bool*
where
  [*simp*]: *chksubs gt t s* ⟷ (*case s of App s1 s2* ⇒ *gt t s1* ∧ *gt t s2* | _ ⇒ *True*)

lemma *chksubs_mono*[*mono*]: *gt* ≤ *gt'* ⟹ *chksubs gt* ≤ *chksubs gt'*
  by (*auto simp*: *tm.case_eq_if*) *force*+

inductive *gt* :: ('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm* ⇒ *bool* (**infix** >$_t$ *50*) **where**
  gt_sub: *is_App t* ⟹ (*fun t* >$_t$ *s* ∨ *fun t* = *s*) ∨ (*arg t* >$_t$ *s* ∨ *arg t* = *s*) ⟹ *t* >$_t$ *s*
| *gt_diff*: *head t* >$_{hd}$ *head s* ⟹ *chkvar t s* ⟹ *chksubs* (>$_t$) *t s* ⟹ *t* >$_t$ *s*
| *gt_same*: *head t* = *head s* ⟹ *chksubs* (>$_t$) *t s* ⟹
    (∀*f* ∈ *ground_heads* (*head t*). *extf f* (>$_t$) (*args t*) (*args s*)) ⟹ *t* >$_t$ *s*

abbreviation *ge* :: ('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm* ⇒ *bool* (**infix** ≥$_t$ *50*) **where**
  t ≥$_t$ *s* ≡ *t* >$_t$ *s* ∨ *t* = *s*

inductive *gt_sub* :: ('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm* ⇒ *bool* **where**
  gt_subI: *is_App t* ⟹ *fun t* ≥$_t$ *s* ∨ *arg t* ≥$_t$ *s* ⟹ *gt_sub t s*

inductive *gt_diff* :: ('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm* ⇒ *bool* **where**
  gt_diffI: *head t* >$_{hd}$ *head s* ⟹ *chkvar t s* ⟹ *chksubs* (>$_t$) *t s* ⟹ *gt_diff t s*

inductive *gt_same* :: ('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm* ⇒ *bool* **where**
  gt_sameI: *head t* = *head s* ⟹ *chksubs* (>$_t$) *t s* ⟹
    (∀*f* ∈ *ground_heads* (*head t*). *extf f* (>$_t$) (*args t*) (*args s*)) ⟹ *gt_same t s*

lemma *gt_iff_sub_diff_same*: *t* >$_t$ *s* ⟷ *gt_sub t s* ∨ *gt_diff t s* ∨ *gt_same t s*
  by (*subst gt.simps*) (*auto simp*: *gt_sub.simps gt_diff.simps gt_same.simps*)

7.3 Transitivity

lemma *gt_fun_imp*: *fun t* >$_t$ *s* ⟹ *t* >$_t$ *s*
  by (*cases t*) (*auto intro*: *gt_sub*)

```
lemma gt_arg_imp: arg t >t s ⟹ t >t s
  by (cases t) (auto intro: gt_sub)

lemma gt_imp_vars: t >t s ⟹ vars t ⊇ vars s
proof (simp only: atomize_imp,
    rule measure_induct_rule[of λ(t, s). size t + size s
      λ(t, s). t >t s ⟶ vars t ⊇ vars s (t, s), simplified prod.case],
    simp only: split_paired_all prod.case atomize_imp[symmetric])
  fix t s
  assume
    ih: ⋀ta sa. size ta + size sa < size t + size s ⟹ ta >t sa ⟹ vars ta ⊇ vars sa and
    t_gt_s: t >t s
  show vars t ⊇ vars s
    using t_gt_s
  proof cases
    case gt_sub
    thus ?thesis
      using ih[of fun t s] ih[of arg t s]
      by (meson add_less_cancel_right subsetD size_arg_lt size_fun_lt subsetI tm.set_sel(5,6))
  next
    case gt_diff
    show ?thesis
    proof (cases s)
      case Hd
      thus ?thesis
        using gt_diff(2) by (auto elim: hd.set_cases(2))
    next
      case (App s1 s2)
      thus ?thesis
        using gt_diff(3) ih[of t s1] ih[of t s2] by simp
    qed
  next
    case gt_same
    show ?thesis
    proof (cases s)
      case Hd
      thus ?thesis
        using gt_same(1) vars_head_subseteq by fastforce
    next
      case (App s1 s2)
      thus ?thesis
        using gt_same(2) ih[of t s1] ih[of t s2] by simp
    qed
  qed
qed

theorem gt_trans: u >t t ⟹ t >t s ⟹ u >t s
proof (simp only: atomize_imp,
    rule measure_induct_rule[of λ(u, t, s). {#size u, size t, size s#}
        λ(u, t, s). u >t t ⟶ t >t s ⟶ u >t s (u, t, s),
      simplified prod.case],
    simp only: split_paired_all prod.case atomize_imp[symmetric])
  fix u t s
  assume
    ih: ⋀ua ta sa. {#size ua, size ta, size sa#} < {#size u, size t, size s#} ⟹
      ua >t ta ⟹ ta >t sa ⟹ ua >t sa and
    u_gt_t: u >t t and t_gt_s: t >t s

  have chkvar: chkvar u s
    by clarsimp (meson u_gt_t t_gt_s gt_imp_vars hd.set_sel(2) vars_head_subseteq subsetCE)

  have chk_u_s_if: chksubs (>t) u s if chk_t_s: chksubs (>t) t s
```

```
proof (cases s)
  case (App s1 s2)
  thus ?thesis
    using chk_t_s by (auto intro: ih[of _ _ s1, OF _ u_gt_t] ih[of _ _ s2, OF _ u_gt_t])
qed auto

have
  fun_u_lt_etc: is_App u ⟹ {#size (fun u), size t, size s#} < {#size u, size t, size s#} and
  arg_u_lt_etc: is_App u ⟹ {#size (arg u), size t, size s#} < {#size u, size t, size s#}
  by (simp_all add: size_fun_lt size_arg_lt)

have u_gt_s_if_ui: is_App u ⟹ fun u ≥t t ∨ arg u ≥t t ⟹ u >t s
  using ih[of fun u t s, OF fun_u_lt_etc] ih[of arg u t s, OF arg_u_lt_etc] gt_arg_imp
    gt_fun_imp t_gt_s by blast

show u >t s
  using t_gt_s
proof cases
  case gt_sub_t_s: gt_sub

  have u_gt_s_if_chk_u_t: ?thesis if chk_u_t: chksubs (>t) u t
    using gt_sub_t_s(1)
  proof (cases t)
    case t: (App t1 t2)
    show ?thesis
      using ih[of u t1 s] ih[of u t2 s] gt_sub_t_s(2) chk_u_t unfolding t by auto
  qed auto

  show ?thesis
    using u_gt_t by cases (auto intro: u_gt_s_if_ui u_gt_s_if_chk_u_t)
next
  case gt_diff_t_s: gt_diff
  show ?thesis
    using u_gt_t
  proof cases
    case gt_diff_u_t: gt_diff
    have head u >hd head s
      using gt_diff_u_t(1) gt_diff_t_s(1) by (auto intro: gt_hd_trans)
    thus ?thesis
      by (rule gt_diff[OF _ chkvar chk_u_s_if[OF gt_diff_t_s(3)]])
  next
    case gt_same_u_t: gt_same
    have head u >hd head s
      using gt_diff_t_s(1) gt_same_u_t(1) by auto
    thus ?thesis
      by (rule gt_diff[OF _ chkvar chk_u_s_if[OF gt_diff_t_s(3)]])
  qed (auto intro: u_gt_s_if_ui)
next
  case gt_same_t_s: gt_same
  show ?thesis
    using u_gt_t
  proof cases
    case gt_diff_u_t: gt_diff
    have head u >hd head s
      using gt_diff_u_t(1) gt_same_t_s(1) by simp
    thus ?thesis
      by (rule gt_diff[OF _ chkvar chk_u_s_if[OF gt_same_t_s(2)]])
  next
    case gt_same_u_t: gt_same
    have hd_u_s: head u = head s
      using gt_same_u_t(1) gt_same_t_s(1) by simp

    let ?S = set (args u) ∪ set (args t) ∪ set (args s)
```

```
    have gt_trans_args: ∀ua ∈ ?S. ∀ta ∈ ?S. ∀sa ∈ ?S. ua >t ta ⟶ ta >t sa ⟶ ua >t sa
    proof clarify
      fix sa ta ua
      assume
        ua_in: ua ∈ ?S and ta_in: ta ∈ ?S and sa_in: sa ∈ ?S and
        ua_gt_ta: ua >t ta and ta_gt_sa: ta >t sa
      show ua >t sa
        by (auto intro!: ih[OF Max_lt_imp_lt_mset ua_gt_ta ta_gt_sa])
          (meson ua_in ta_in sa_in Un_iff max.strict_coboundedI1 max.strict_coboundedI2
            size_in_args)+
    qed

    have ∀f ∈ ground_heads (head u). extf f (>t) (args u) (args s)
    proof (clarify, rule extf_trans[OF _ _ _ gt_trans_args])
      fix f
      assume f_in_grounds: f ∈ ground_heads (head u)
      show extf f (>t) (args u) (args t)
        using f_in_grounds gt_same_u_t(3) by blast
      show extf f (>t) (args t) (args s)
        using f_in_grounds gt_same_t_s(3) unfolding gt_same_u_t(1) by blast
    qed auto
    thus ?thesis
      by (rule gt_same[OF hd_u_s chk_u_s_if[OF gt_same_t_s(2)]])
  qed (auto intro: u_gt_s_if_ui)
 qed
qed
```

7.4 Irreflexivity

```
theorem gt_irrefl: ¬ s >t s
proof (standard, induct s rule: measure_induct_rule[of size])
  case (less s)
  note ih = this(1) and s_gt_s = this(2)

  show False
    using s_gt_s
  proof cases
    case _: gt_sub
    note is_app = this(1) and si_ge_s = this(2)
    have s_gt_fun_s: s >t fun s and s_gt_arg_s: s >t arg s
      using is_app by (simp_all add: gt_sub)

    have fun s >t s ∨ arg s >t s
      using si_ge_s is_app s_gt_arg_s s_gt_fun_s by auto
    moreover
    {
      assume fun_s_gt_s: fun s >t s
      have fun s >t fun s
        by (rule gt_trans[OF fun_s_gt_s s_gt_fun_s])
      hence False
        using ih[of fun s] is_app size_fun_lt by blast
    }
    moreover
    {
      assume arg_s_gt_s: arg s >t s
      have arg s >t arg s
        by (rule gt_trans[OF arg_s_gt_s s_gt_arg_s])
      hence False
        using ih[of arg s] is_app size_arg_lt by blast
    }
    ultimately show False
      by sat
  next
```

```
    case gt_diff
    thus False
      by (cases head s) (auto simp: gt_hd_irrefl)
  next
    case gt_same
    note in_grounds = this(3)

    obtain si where si_in_args: si ∈ set (args s) and si_gt_si: si >_t si
      using in_grounds
      by (metis (full_types) all_not_in_conv extf_irrefl_from_trans ground_heads_nonempty gt_trans)
    have size si < size s
      by (rule size_in_args[OF si_in_args])
    thus False
      by (rule ih[OF _ si_gt_si])
  qed
qed

lemma gt_antisym: t >_t s ⟹ ¬ s >_t t
  using gt_irrefl gt_trans by blast
```

7.5 Subterm Property

```
lemma
  gt_sub_fun: App s t >_t s and
  gt_sub_arg: App s t >_t t
  by (auto intro: gt_sub)

theorem gt_proper_sub: proper_sub s t ⟹ t >_t s
  by (induct t) (auto intro: gt_sub_fun gt_sub_arg gt_trans)
```

7.6 Compatibility with Functions

```
lemma gt_compat_fun:
  assumes t'_gt_t: t' >_t t
  shows App s t' >_t App s t
proof −
  have t'_ne_t: t' ≠ t
    using gt_antisym t'_gt_t by blast
  have extf_args_single: ∀f ∈ ground_heads (head s). extf f (>_t) (args s @ [t']) (args s @ [t])
    by (simp add: extf_compat_list t'_gt_t t'_ne_t)
  show ?thesis
    by (rule gt_same) (auto simp: gt_sub gt_sub_fun t'_gt_t intro!: extf_args_single)
qed

theorem gt_compat_fun_strong:
  assumes t'_gt_t: t' >_t t
  shows apps s (t' # us) >_t apps s (t # us)
proof (induct us rule: rev_induct)
  case Nil
  show ?case
    using t'_gt_t by (auto intro!: gt_compat_fun)
next
  case (snoc u us)
  note ih = snoc

  let ?v' = apps s (t' # us @ [u])
  let ?v = apps s (t # us @ [u])

  show ?case
  proof (rule gt_same)
    show chksubs (>_t) ?v' ?v
      using ih by (auto intro: gt_sub gt_sub_arg)
  next
    show ∀f ∈ ground_heads (head ?v'). extf f (>_t) (args ?v') (args ?v)
```

```
    by (metis args_apps extf_compat_list gt_irrefl t'_gt_t)
  qed simp
qed
```

7.7 Compatibility with Arguments

```
theorem gt_diff_same_compat_arg:
  assumes
    extf_compat_snoc: ⋀f. ext_compat_snoc (extf f) and
    diff_same: gt_diff s' s ∨ gt_same s' s
  shows App s' t >t App s t
proof −
  {
    assume s' >t s
    hence App s' t >t s
      using gt_sub_fun gt_trans by blast
    moreover have App s' t >t t
      by (simp add: gt_sub_arg)
    ultimately have chksubs (>t) (App s' t) (App s t)
      by auto
  }
  note chk_s't_st = this

  show ?thesis
    using diff_same
  proof
    assume gt_diff s' s
    hence
      s'_gt_s: s' >t s and
      hd_s'_gt_s: head s' >hd head s and
      chkvar_s'_s: chkvar s' s and
      chk_s'_s: chksubs (>t) s' s
      using gt_diff.cases by (auto simp: gt_iff_sub_diff_same)

    have chkvar_s't_st: chkvar (App s' t) (App s t)
      using chkvar_s'_s by auto
    show ?thesis
      by (rule gt_diff[OF _ chkvar_s't_st chk_s't_st[OF s'_gt_s]])
        (simp add: hd_s'_gt_s[simplified])
  next
    assume gt_same s' s
    hence
      s'_gt_s: s' >t s and
      hd_s'_eq_s: head s' = head s and
      chk_s'_s: chksubs (>t) s' s and
      gts_args: ∀f ∈ ground_heads (head s'). extf f (>t) (args s') (args s)
      using gt_same.cases by (auto simp: gt_iff_sub_diff_same, metis)

    have gts_args_t:
      ∀f ∈ ground_heads (head (App s' t)). extf f (>t) (args (App s' t)) (args (App s t))
      using gts_args ext_compat_snoc.compat_append_right[OF extf_compat_snoc] by simp

    show ?thesis
      by (rule gt_same[OF _ chk_s't_st[OF s'_gt_s] gts_args_t]) (simp add: hd_s'_eq_s)
  qed
qed
```

7.8 Stability under Substitution

```
lemma gt_imp_chksubs_gt:
  assumes t_gt_s: t >t s
  shows chksubs (>t) t s
proof −
  have is_App s ⟹ t >t fun s ∧ t >t arg s
```

```
    using t_gt_s by (meson gt_sub gt_trans)
  thus ?thesis
    by (simp add: tm.case_eq_if)
qed

theorem gt_subst:
  assumes wary_ϱ: wary_subst ϱ
  shows t >_t s ⟹ subst ϱ t >_t subst ϱ s
proof (simp only: atomize_imp,
    rule measure_induct_rule[of λ(t, s). {#size t, size s#}
        λ(t, s). t >_t s ⟶ subst ϱ t >_t subst ϱ s (t, s),
      simplified prod.case],
    simp only: split_paired_all prod.case atomize_imp[symmetric])
  fix t s
  assume
    ih: ⋀ta sa. {#size ta, size sa#} < {#size t, size s#} ⟹ ta >_t sa ⟹
      subst ϱ ta >_t subst ϱ sa and
    t_gt_s: t >_t s

  {
    assume chk_t_s: chksubs (>_t) t s
    have chksubs (>_t) (subst ϱ t) (subst ϱ s)
    proof (cases s)
      case s: (Hd ζ)
      show ?thesis
      proof (cases ζ)
        case ζ: (Var x)
        have psub_x_t: proper_sub (Hd (Var x)) t
          using ζ s t_gt_s gt_imp_vars gt_irrefl in_vars_imp_sub by fastforce
        show ?thesis
          unfolding ζ s
          by (rule gt_imp_chksubs_gt[OF gt_proper_sub[OF proper_sub_subst]]) (rule psub_x_t)
      qed (auto simp: s)
    next
      case s: (App s1 s2)
      have t >_t s1 and t >_t s2
        using s chk_t_s by auto
      thus ?thesis
        using s by (auto intro!: ih[of t s1] ih[of t s2])
    qed
  }
  note chk_ϱt_ϱs_if = this

  show subst ϱ t >_t subst ϱ s
    using t_gt_s
  proof cases
    case gt_sub_t_s: gt_sub
    obtain t1 t2 where t: t = App t1 t2
      using gt_sub_t_s(1) by (metis tm.collapse(2))
    show ?thesis
      using gt_sub ih[of t1 s] ih[of t2 s] gt_sub_t_s(2) t by auto
  next
    case gt_diff_t_s: gt_diff
    have head (subst ϱ t) >_hd head (subst ϱ s)
      by (meson wary_subst_ground_heads gt_diff_t_s(1) gt_hd_def subsetCE wary_ϱ)
    moreover have chkvar (subst ϱ t) (subst ϱ s)
      unfolding chkvar_def using vars_subst_subseteq[OF gt_imp_vars[OF t_gt_s]] vars_head_subseteq
      by force
    ultimately show ?thesis
      by (rule gt_diff[OF _ _ chk_ϱt_ϱs_if[OF gt_diff_t_s(3)]])
  next
    case gt_same_t_s: gt_same
```

```
    have hd_ϱt_eq_ϱs: head (subst ϱ t) = head (subst ϱ s)
      using gt_same_t_s(1) by simp

    {
      fix f
      assume f_in_grounds: f ∈ ground_heads (head (subst ϱ t))

      let ?S = set (args t) ∪ set (args s)

      have extf_args_s_t: extf f (>t) (args t) (args s)
        using gt_same_t_s(3) f_in_grounds wary_ϱ wary_subst_ground_heads by blast
      have extf f (>t) (map (subst ϱ) (args t)) (map (subst ϱ) (args s))
      proof (rule extf_map[of ?S, OF _ _ _ _ _ _ extf_args_s_t])
        have sz_a: ∀ ta ∈ ?S. ∀ sa ∈ ?S. {#size ta, size sa#} < {#size t, size s#}
          by (fastforce intro: Max_lt_imp_lt_mset dest: size_in_args)
        show ∀ ta ∈ ?S. ∀ sa ∈ ?S. ta >t sa ⟶ subst ϱ ta >t subst ϱ sa
          using ih sz_a size_in_args by fastforce
      qed (auto intro!: gt_irefl elim!: gt_trans)
      hence extf f (>t) (args (subst ϱ t)) (args (subst ϱ s))
        by (auto simp: gt_same_t_s(1) intro: extf_compat_append_left)
    }
    hence ∀ f ∈ ground_heads (head (subst ϱ t)).
      extf f (>t) (args (subst ϱ t)) (args (subst ϱ s))
      by blast
    thus ?thesis
      by (rule gt_same[OF hd_ϱt_eq_ϱs chk_ϱt_ϱs_if[OF gt_same_t_s(2)]])
  qed
qed
```

7.9 Totality on Ground Terms

```
theorem gt_total_ground:
  assumes extf_total: ⋀f. ext_total (extf f)
  shows ground t ⟹ ground s ⟹ t >t s ∨ s >t t ∨ t = s
proof (simp only: atomize_imp,
    rule measure_induct_rule[of λ(t, s). {# size t, size s #}
      λ(t, s). ground t ⟶ ground s ⟶ t >t s ∨ s >t t ∨ t = s (t, s), simplified prod.case],
    simp only: split_paired_all prod.case atomize_imp[symmetric])
  fix t s :: ('s, 'v) tm
  assume
    ih: ⋀ta sa. {# size ta, size sa #} < {# size t, size s #} ⟹ ground ta ⟹ ground sa ⟹
      ta >t sa ∨ sa >t ta ∨ ta = sa and
    gr_t: ground t and gr_s: ground s

  let ?case = t >t s ∨ s >t t ∨ t = s

  have chksubs (>t) t s ∨ s >t t
    unfolding chksubs_def tm.case_eq_if using ih[of t fun s] ih[of t arg s] mset_lt_single_iff
    by (metis add_mset_lt_right_lt gr_s gr_t ground_arg ground_fun gt_sub size_arg_lt size_fun_lt)
  moreover have chksubs (>t) s t ∨ t >t s
    unfolding chksubs_def tm.case_eq_if using ih[of fun t s] ih[of arg t s]
    by (metis add_mset_lt_left_lt gr_s gr_t ground_arg ground_fun gt_sub size_arg_lt size_fun_lt)
  moreover
  {
    assume
      chksubs_t_s: chksubs (>t) t s and
      chksubs_s_t: chksubs (>t) s t

    obtain g where g: head t = Sym g
      using gr_t by (metis ground_head hd.collapse(2))
    obtain f where f: head s = Sym f
      using gr_s by (metis ground_head hd.collapse(2))

    have chkvar_t_s: chkvar t s and chkvar_s_t: chkvar s t
```

```
    using g f by simp_all

  {
    assume g_gt_f: g >_s f
    have t >_t s
      by (rule gt_diff[OF _ chkvar_t_s chksubs_t_s]) (simp add: g f gt_sym_imp_hd[OF g_gt_f])
  }
  moreover
  {
    assume f_gt_g: f >_s g
    have s >_t t
      by (rule gt_diff[OF _ chkvar_s_t chksubs_s_t]) (simp add: g f gt_sym_imp_hd[OF f_gt_g])
  }
  moreover
  {
    assume g_eq_f: g = f
    hence hd_t: head t = head s
      using g f by auto

    let ?ts = args t
    let ?ss = args s

    have gr_ts: ∀ ta ∈ set ?ts. ground ta
      using ground_args[OF _ gr_t] by blast
    have gr_ss: ∀ sa ∈ set ?ss. ground sa
      using ground_args[OF _ gr_s] by blast

    {
      assume ts_eq_ss: ?ts = ?ss
      have t = s
        using hd_t ts_eq_ss by (rule tm_expand_apps)
    }
    moreover
    {
      assume ts_gt_ss: extf g (>_t) ?ts ?ss
      have t >_t s
        by (rule gt_same[OF hd_t chksubs_t_s]) (auto simp: g ts_gt_ss)
    }
    moreover
    {
      assume ss_gt_ts: extf g (>_t) ?ss ?ts
      have s >_t t
        by (rule gt_same[OF hd_t[symmetric] chksubs_s_t]) (auto simp: f[folded g_eq_f] ss_gt_ts)
    }
    ultimately have ?case
      using ih gr_ss gr_ts
        ext_total.total[OF extf_total, rule_format, of set ?ts ∪ set ?ss (>_t) ?ts ?ss g]
      by (metis Un_iff in_listsI less_multiset_doubletons size_in_args)
  }
  ultimately have ?case
    using gt_sym_total by blast
}
ultimately show ?case
  by fast
qed
```

7.10 Well-foundedness

```
abbreviation gtg :: ('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool (infix >_tg 50) where
  (>_tg) ≡ λt s. ground t ∧ t >_t s

theorem gt_wf:
  assumes extf_wf: ⋀f. ext_wf (extf f)
  shows wfP (λs t. t >_t s)
```

```
proof −
  have ground_wfP: wfP (λs t. t >_tg s)
    unfolding wfP_iff_no_inf_chain
  proof
    assume ∃f. inf_chain (>_tg) f
    then obtain t where t_bad: bad (>_tg) t
      unfolding inf_chain_def bad_def by blast

    let ?ff = worst_chain (>_tg) (λt s. size t > size s)
    let ?U_of = λi. if is_App (?ff i) then {fun (?ff i)} ∪ set (args (?ff i)) else {}

    note wf_sz = wf_app[OF wellorder_class.wf, of size, simplified]

    define U where U = (⋃i. ?U_of i)

    have gr: ⋀i. ground (?ff i)
      using worst_chain_bad[OF wf_sz t_bad, unfolded inf_chain_def] by fast
    have gr_fun: ⋀i. ground (fun (?ff i))
      by (rule ground_fun[OF gr])
    have gr_args: ⋀i s. s ∈ set (args (?ff i)) ⟹ ground s
      by (rule ground_args[OF _ gr])
    have gr_u: ⋀u. u ∈ U ⟹ ground u
      unfolding U_def by (auto dest: gr_args) (metis (lifting) empty_iff gr_fun)

    have ¬ bad (>_tg) u if u_in: u ∈ ?U_of i for u i
    proof
      let ?ti = ?ff i

      assume u_bad: bad (>_tg) u
      have sz_u: size u < size ?ti
      proof (cases ?ff i)
        case Hd
        thus ?thesis
          using u_in size_in_args by fastforce
      next
        case App
        thus ?thesis
          using u_in size_in_args insert_iff size_fun_lt by fastforce
      qed

      show False
      proof (cases i)
        case 0
        thus False
          using sz_u min_worst_chain_0[OF wf_sz u_bad] by simp
      next
        case Suc
        hence ?ff (i − 1) >_t ?ff i
          using worst_chain_pred[OF wf_sz t_bad] by simp
        moreover have ?ff i >_t u
        proof −
          have u_in: u ∈ ?U_of i
            using u_in by blast
          have ffi_ne_u: ?ff i ≠ u
            using sz_u by fastforce
          hence is_App (?ff i) ⟹ ¬ sub u (?ff i) ⟹ ?ff i >_t u
            using u_in gt_sub sub_args by auto
          thus ?ff i >_t u
            using ffi_ne_u u_in gt_proper_sub sub_args by fastforce
        qed
        ultimately have ?ff (i − 1) >_t u
          by (rule gt_trans)
        thus False
```

```
    using Suc sz_u min_worst_chain_Suc[OF wf_sz u_bad] gr by fastforce
  qed
qed
hence u_good: ⋀u. u ∈ U ⟹ ¬ bad (>_tg) u
  unfolding U_def by blast

have bad_diff_same: inf_chain (λt s. ground t ∧ (gt_diff t s ∨ gt_same t s)) ?ff
  unfolding inf_chain_def
proof (intro allI conjI)
  fix i

  show ground (?ff i)
    by (rule gr)

  have gt: ?ff i >_t ?ff (Suc i)
    using worst_chain_pred[OF wf_sz t_bad] by blast

  have ¬ gt_sub (?ff i) (?ff (Suc i))
  proof
    assume a: gt_sub (?ff i) (?ff (Suc i))
    hence fi_app: is_App (?ff i) and
      fun_or_arg_fi_ge: fun (?ff i) ≥_t ?ff (Suc i) ∨ arg (?ff i) ≥_t ?ff (Suc i)
      unfolding gt_sub.simps by blast+
    have fun (?ff i) ∈ ?U_of i
      unfolding U_def using fi_app by auto
    moreover have arg (?ff i) ∈ ?U_of i
      unfolding U_def using fi_app arg_in_args by force
    ultimately obtain uij where uij_in: uij ∈ U and uij_cases: uij ≥_t ?ff (Suc i)
      unfolding U_def using fun_or_arg_fi_ge by blast

    have ⋀n. ?ff n >_t ?ff (Suc n)
      by (rule worst_chain_pred[OF wf_sz t_bad, THEN conjunct2])
    hence uij_gt_i_plus_3: uij >_t ?ff (Suc (Suc i))
      using gt_trans uij_cases by blast

    have inf_chain (>_tg) (λj. if j = 0 then uij else ?ff (Suc (i + j)))
      unfolding inf_chain_def
      by (auto intro!: gr gr_u[OF uij_in] uij_gt_i_plus_3 worst_chain_pred[OF wf_sz t_bad])
    hence bad (>_tg) uij
      unfolding bad_def by fastforce
    thus False
      using u_good[OF uij_in] by sat
  qed
  thus gt_diff (?ff i) (?ff (Suc i)) ∨ gt_same (?ff i) (?ff (Suc i))
    using gt unfolding gt_iff_sub_diff_same by sat
qed

have wf {(s, t). ground s ∧ ground t ∧ sym (head t) >_s sym (head s)}
  using gt_sym_wf unfolding wfP_def wf_iff_no_infinite_down_chain by fast
moreover have {(s, t). ground t ∧ gt_diff t s}
  ⊆ {(s, t). ground s ∧ ground t ∧ sym (head t) >_s sym (head s)}
proof (clarsimp, intro conjI)
  fix s t
  assume gr_t: ground t and gt_diff_t_s: gt_diff t s
  thus gr_s: ground s
    using gt_iff_sub_diff_same gt_imp_vars by fastforce

  show sym (head t) >_s sym (head s)
    using gt_diff_t_s ground_head[OF gr_s] ground_head[OF gr_t]
    by (cases; cases head s; cases head t) (auto simp: gt_hd_def)
qed
ultimately have wf_diff: wf {(s, t). ground t ∧ gt_diff t s}
  by (rule wf_subset)
```

have *diff_O_same*: $\{(s, t).\ ground\ t \wedge gt\_diff\ t\ s\}\ O\ \{(s, t).\ ground\ t \wedge gt\_same\ t\ s\}$
$\subseteq \{(s, t).\ ground\ t \wedge gt\_diff\ t\ s\}$
unfolding *gt_diff.simps gt_same.simps*
by *clarsimp* (*metis chksubs_def empty_subsetI gt_diff*[*unfolded chkvar_def*] *gt_imp_chksubs_gt gt_same gt_trans*)

have *diff_same_as_union*: $\{(s, t).\ ground\ t \wedge (gt\_diff\ t\ s \vee gt\_same\ t\ s)\} =$
$\{(s, t).\ ground\ t \wedge gt\_diff\ t\ s\} \cup \{(s, t).\ ground\ t \wedge gt\_same\ t\ s\}$
by *auto*

obtain *k* **where** *bad_same*: *inf_chain* $(\lambda t\ s.\ ground\ t \wedge gt\_same\ t\ s)$ $(\lambda i.\ ?ff\ (i + k))$
using *wf_infinite_down_chain_compatible*[*OF wf_diff _ diff_O_same*, *of ?ff*] *bad_diff_same*
unfolding *inf_chain_def diff_same_as_union*[*symmetric*] **by** *auto*
hence *hd_sym*: $\bigwedge i.\ is\_Sym\ (head\ (?ff\ (i + k)))$
unfolding *inf_chain_def* **by** (*simp add*: *ground_head*)

define *f* **where** $f = sym\ (head\ (?ff\ k))$

have *hd_eq_f*: $head\ (?ff\ (i + k)) = Sym\ f$ **for** *i*
proof (*induct i*)
case *0*
thus *?case*
by (*auto simp*: *f_def hd.collapse*(*2*)[*OF hd_sym*, *of 0*, *simplified*])
next
case (*Suc ia*)
thus *?case*
using *bad_same* **unfolding** *inf_chain_def gt_same.simps* **by** *simp*
qed

let $?gtu = \lambda t\ s.\ t \in U \wedge t >_t s$

have $t \in set\ (args\ (?ff\ i)) \Longrightarrow t \in ?U\_of\ i$ **for** *t i*
unfolding *U_def*
by (*cases is_App* (*?ff i*), *simp_all*,
metis (*lifting*) *neq_iff size_in_args sub.cases sub_args tm.discI*(*2*))
moreover have $\bigwedge i.\ extf\ f\ (>_t)\ (args\ (?ff\ (i + k)))\ (args\ (?ff\ (Suc\ i + k)))$
using *bad_same hd_eq_f* **unfolding** *inf_chain_def gt_same.simps* **by** *auto*
ultimately have $\bigwedge i.\ extf\ f\ ?gtu\ (args\ (?ff\ (i + k)))\ (args\ (?ff\ (Suc\ i + k)))$
using *extf_mono_strong*[*of _ _* $(>_t)$ $\lambda t\ s.\ t \in U \wedge t >_t s$] **unfolding** *U_def* **by** *blast*
hence *inf_chain* $(extf\ f\ ?gtu)$ $(\lambda i.\ args\ (?ff\ (i + k)))$
unfolding *inf_chain_def* **by** *blast*
hence *nwf_ext*: $\neg\ wfP\ (\lambda xs\ ys.\ extf\ f\ ?gtu\ ys\ xs)$
unfolding *wfP_iff_no_inf_chain* **by** *fast*

have *gtu_le_gtg*: $?gtu \leq (>_{tg})$
by (*auto intro*!: *gr_u*)

have *wfP* $(\lambda s\ t.\ ?gtu\ t\ s)$
unfolding *wfP_iff_no_inf_chain*
proof (*intro notI*, *elim exE*)
fix *f*
assume *bad_f*: *inf_chain ?gtu f*
hence *bad_f0*: *bad ?gtu* (*f 0*)
by (*rule inf_chain_bad*)

have $f\ 0 \in U$
using *bad_f* **unfolding** *inf_chain_def* **by** *blast*
hence *good_f0*: $\neg\ bad\ ?gtu\ (f\ 0)$
using *u_good bad_f inf_chain_bad inf_chain_subset*[*OF _ gtu_le_gtg*] **by** *blast*

show *False*
using *bad_f0 good_f0* **by** *sat*

qed
hence *wf_ext*: *wfP* (λ*xs ys*. *extf f ?gtu ys xs*)
by (*rule ext_wf.wf*[*OF extf_wf*, *rule_format*])

show *False*
using *nwf_ext wf_ext* **by** *blast*
qed

let *?subst* = *subst grounding_ϱ*

have *wfP* (λ*s t*. *?subst t* >$_{tg}$ *?subst s*)
by (*rule wfP_app*[*OF ground_wfP*])
hence *wfP* (λ*s t*. *?subst t* >$_t$ *?subst s*)
by (*simp add*: *ground_grounding_ϱ*)
thus *?thesis*
by (*auto intro*: *wfP_subset gt_subst*[*OF wary_grounding_ϱ*])
qed

end

end

8 The Optimized Graceful Recursive Path Order for Lambda-Free Higher-Order Terms

theory *Lambda_Free_RPO_Optim*
imports *Lambda_Free_RPO_Std*
begin

This theory defines the optimized variant of the graceful recursive path order (RPO) for λ-free higher-order terms.

8.1 Setup

locale *rpo_optim* = *rpo_basis* _ _ *arity_sym arity_var*
for
arity_sym :: *'s* ⇒ *enat* **and**
arity_var :: *'v* ⇒ *enat* +
assumes *extf_ext_snoc*: *ext_snoc* (*extf f*)
begin

lemmas *extf_snoc* = *ext_snoc.snoc*[*OF extf_ext_snoc*]

8.2 Definition of the Order

definition
chkargs :: (('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm* ⇒ *bool*) ⇒ ('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm* ⇒ *bool*
where
[*simp*]: *chkargs gt t s* ⟷ (∀ *s'* ∈ *set* (*args s*). *gt t s'*)

lemma *chkargs_mono*[*mono*]: *gt* ≤ *gt'* ⟹ *chkargs gt* ≤ *chkargs gt'*
by *force*

inductive *gt* :: ('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm* ⇒ *bool* (**infix** >$_t$ *50*) **where**
gt_arg: *ti* ∈ *set* (*args t*) ⟹ *ti* >$_t$ *s* ∨ *ti* = *s* ⟹ *t* >$_t$ *s*
| *gt_diff*: *head t* >$_{hd}$ *head s* ⟹ *chkvar t s* ⟹ *chkargs* (>$_t$) *t s* ⟹ *t* >$_t$ *s*
| *gt_same*: *head t* = *head s* ⟹ *chkargs* (>$_t$) *t s* ⟹
(∀ *f* ∈ *ground_heads* (*head t*). *extf f* (>$_t$) (*args t*) (*args s*)) ⟹ *t* >$_t$ *s*

abbreviation *ge* :: ('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm* ⇒ *bool* (**infix** ≥$_t$ *50*) **where**
t ≥$_t$ *s* ≡ *t* >$_t$ *s* ∨ *t* = *s*

8.3 Transitivity

lemma *gt_in_args_imp*: $ti \in set\ (args\ t) \Longrightarrow ti >_t s \Longrightarrow t >_t s$
by (*cases t*) (*auto intro*: *gt_arg*)

lemma *gt_imp_vars*: $t >_t s \Longrightarrow vars\ t \supseteq vars\ s$
proof (*simp only*: *atomize_imp*,
rule measure_induct_rule[*of* $\lambda(t, s).\ size\ t + size\ s$
$\lambda(t, s).\ t >_t s \longrightarrow vars\ t \supseteq vars\ s\ (t, s)$, *simplified prod.case*],
simp only: *split_paired_all prod.case atomize_imp*[*symmetric*])
fix *t s*
assume
ih: $\bigwedge ta\ sa.\ size\ ta + size\ sa < size\ t + size\ s \Longrightarrow ta >_t sa \Longrightarrow vars\ ta \supseteq vars\ sa$ **and**
t_gt_s: $t >_t s$
show $vars\ t \supseteq vars\ s$
using *t_gt_s*
proof *cases*
case (*gt_arg ti*)
thus *?thesis*
using *ih*[*of ti s*]
by (*metis size_in_args vars_args_subseteq add_mono_thms_linordered_field*(*1*) *order_trans*)
next
case *gt_diff*
show *?thesis*
proof (*cases s*)
case *Hd*
thus *?thesis*
using *gt_diff*(*2*) **by** (*auto elim*: *hd.set_cases*(*2*))
next
case (*App s1 s2*)
thus *?thesis*
using *gt_diff ih*
by *simp* (*metis* (*no_types*) *add.assoc gt.simps*[*unfolded chkargs_def chkvar_def*] *less_add_Suc1*)
qed
next
case *gt_same*
thus *?thesis*
proof (*cases s rule*: *tm_exhaust_apps*)
case *s*: (*apps* ζ *ss*)
thus *?thesis*
using *gt_same* **unfolding** *chkargs_def s*
by (*auto intro*!: *vars_head_subseteq*)
(*metis ih*[*of t*] *insert_absorb insert_subset nat_add_left_cancel_less s size_in_args*
tm_collapse_apps tm_inject_apps)
qed
qed
qed

lemma *gt_trans*: $u >_t t \Longrightarrow t >_t s \Longrightarrow u >_t s$
proof (*simp only*: *atomize_imp*,
rule measure_induct_rule[*of* $\lambda(u, t, s).\ \{\#size\ u, size\ t, size\ s\#\}$
$\lambda(u, t, s).\ u >_t t \longrightarrow t >_t s \longrightarrow u >_t s\ (u, t, s)$,
simplified prod.case],
simp only: *split_paired_all prod.case atomize_imp*[*symmetric*])
fix *u t s*
assume
ih: $\bigwedge ua\ ta\ sa.\ \{\#size\ ua, size\ ta, size\ sa\#\} < \{\#size\ u, size\ t, size\ s\#\} \Longrightarrow$
$ua >_t ta \Longrightarrow ta >_t sa \Longrightarrow ua >_t sa$ **and**
u_gt_t: $u >_t t$ **and** *t_gt_s*: $t >_t s$

have *chkvar*: *chkvar u s*
by *clarsimp* (*meson u_gt_t t_gt_s gt_imp_vars hd.set_sel*(*2*) *vars_head_subseteq subsetCE*)

have *chk_u_s_if*: *chkargs* $(>_t)$ *u s* **if** *chk_t_s*: *chkargs* $(>_t)$ *t s*

```
proof (clarsimp simp only: chkargs_def)
  fix s'
  assume s' ∈ set (args s)
  thus u >_t s'
    using chk_t_s by (auto intro!: ih[of _ _ s', OF _ u_gt_t] size_in_args)
qed

have u_gt_s_if_ui: ui ≥_t t ⟹ u >_t s if ui_in: ui ∈ set (args u) for ui
  using ih[of ui t s, simplified, OF size_in_args[OF ui_in] _ t_gt_s]
    gt_in_args_imp[OF ui_in, of s] t_gt_s by blast

show u >_t s
  using t_gt_s
proof cases
  case gt_arg_t_s: (gt_arg ti)
  have u_gt_s_if_chk_u_t: ?thesis if chk_u_t: chkargs (>_t) u t
    using ih[of u ti s] gt_arg_t_s chk_u_t size_in_args by force
  show ?thesis
    using u_gt_t by cases (auto intro: u_gt_s_if_ui u_gt_s_if_chk_u_t)
next
  case gt_diff_t_s: gt_diff
  show ?thesis
    using u_gt_t
  proof cases
    case gt_diff_u_t: gt_diff
    have head u >_hd head s
      using gt_diff_u_t(1) gt_diff_t_s(1) by (auto intro: gt_hd_trans)
    thus ?thesis
      by (rule gt_diff[OF _ chkvar chk_u_s_if[OF gt_diff_t_s(3)]])
  next
    case gt_same_u_t: gt_same
    have head u >_hd head s
      using gt_diff_t_s(1) gt_same_u_t(1) by auto
    thus ?thesis
      by (rule gt_diff[OF _ chkvar chk_u_s_if[OF gt_diff_t_s(3)]])
  qed (auto intro: u_gt_s_if_ui)
next
  case gt_same_t_s: gt_same
  show ?thesis
    using u_gt_t
  proof cases
    case gt_diff_u_t: gt_diff
    have head u >_hd head s
      using gt_diff_u_t(1) gt_same_t_s(1) by simp
    thus ?thesis
      by (rule gt_diff[OF _ chkvar chk_u_s_if[OF gt_same_t_s(2)]])
  next
    case gt_same_u_t: gt_same
    have hd_u_s: head u = head s
      using gt_same_u_t(1) gt_same_t_s(1) by simp

    let ?S = set (args u) ∪ set (args t) ∪ set (args s)

    have gt_trans_args: ∀ua ∈ ?S. ∀ta ∈ ?S. ∀sa ∈ ?S. ua >_t ta ⟶ ta >_t sa ⟶ ua >_t sa
    proof clarify
      fix sa ta ua
      assume
        ua_in: ua ∈ ?S and ta_in: ta ∈ ?S and sa_in: sa ∈ ?S and
        ua_gt_ta: ua >_t ta and ta_gt_sa: ta >_t sa
      show ua >_t sa
        by (auto intro!: ih[OF Max_lt_imp_lt_mset ua_gt_ta ta_gt_sa])
          (meson ua_in ta_in sa_in Un_iff max.strict_coboundedI1 max.strict_coboundedI2
            size_in_args)+
```

```
    qed

    have ∀f ∈ ground_heads (head u). extf f (>t) (args u) (args s)
    proof (clarify, rule extf_trans[OF _ _ _ gt_trans_args])
      fix f
      assume f_in_grounds: f ∈ ground_heads (head u)
      show extf f (>t) (args u) (args t)
        using f_in_grounds gt_same_u_t(3) by blast
      show extf f (>t) (args t) (args s)
        using f_in_grounds gt_same_t_s(3) unfolding gt_same_u_t(1) by blast
    qed auto
    thus ?thesis
      by (rule gt_same[OF hd_u_s chk_u_s_if[OF gt_same_t_s(2)]])
  qed (auto intro: u_gt_s_if_ui)
 qed
qed

lemma gt_sub_fun: App s t >t s
 by (rule gt_same) (auto intro: extf_snoc gt_arg[of _ App s t])

end
```

8.4 Conditional Equivalence with Unoptimized Version

```
context rpo
begin

context
 assumes extf_ext_snoc: ⋀f. ext_snoc (extf f)
begin

lemma rpo_optim: rpo_optim ground_heads_var (>s) extf arity_sym arity_var
 unfolding rpo_optim_def rpo_optim_axioms_def using rpo_basis_axioms extf_ext_snoc by auto

abbreviation
 chkargs :: (('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool) ⇒ ('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool
where
 chkargs ≡ rpo_optim.chkargs

abbreviation gt_optim :: ('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool (infix >to 50) where
 (>to) ≡ rpo_optim.gt ground_heads_var (>s) extf

abbreviation ge_optim :: ('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool (infix ≥to 50) where
 (≥to) ≡ rpo_optim.ge ground_heads_var (>s) extf

theorem gt_iff_optim: t >t s ⟷ t >to s
proof (rule measure_induct_rule[of λ(t, s). size t + size s
     λ(t, s). t >t s ⟷ t >to s (t, s), simplified prod.case],
   simp only: split_paired_all prod.case)
 fix t s :: ('s, 'v) tm
 assume ih: ⋀ta sa. size ta + size sa < size t + size s ⟹ ta >t sa ⟷ ta >to sa

 show t >t s ⟷ t >to s
 proof
   assume t_gt_s: t >t s

   have chkargs_if_chksubs: chkargs (>to) t s if chksubs: chksubs (>t) t s
     unfolding rpo_optim.chkargs_def[OF rpo_optim]
   proof (cases s, simp_all, intro conjI ballI)
     fix s1 s2
     assume s: s = App s1 s2

     have t_gt_s2: t >t s2
       using chksubs s by simp
```

```
  show t >to s2
    by (rule ih[THEN iffD1, OF _ t_gt_s2]) (simp add: s)

  {
    fix s1i
    assume s1i_in: s1i ∈ set (args s1)

    have t >t s1
      using chksubs s by simp
    moreover have s1 >t s1i
      using s1i_in gt_proper_sub size_in_args sub_args by fastforce
    ultimately have t_gt_s1i: t >t s1i
      by (rule gt_trans)

    have sz_s1i: size s1i < size s
      using size_in_args[OF s1i_in] s by simp

    show t >to s1i
      by (rule ih[THEN iffD1, OF _ t_gt_s1i]) (simp add: sz_s1i)
  }
qed

show t >to s
  using t_gt_s
proof cases
  case gt_sub
  note t_app = this(1) and ti_geo_s = this(2)

  obtain t1 t2 where t: t = App t1 t2
    using t_app by (metis tm.collapse(2))

  have t_gto_t1: t >to t1
    unfolding t by (rule rpo_optim.gt_sub_fun[OF rpo_optim])
  have t_gto_t2: t >to t2
    unfolding t by (rule rpo_optim.gt_arg[OF rpo_optim, of t2]) simp+

  {
    assume t1_gt_s: t1 >t s
    have t1 >to s
      by (rule ih[THEN iffD1, OF _ t1_gt_s]) (simp add: t)
    hence ?thesis
      by (rule rpo_optim.gt_trans[OF rpo_optim t_gto_t1])
  }
  moreover
  {
    assume t2_gt_s: t2 >t s
    have t2 >to s
      by (rule ih[THEN iffD1, OF _ t2_gt_s]) (simp add: t)
    hence ?thesis
      by (rule rpo_optim.gt_trans[OF rpo_optim t_gto_t2])
  }
  ultimately show ?thesis
    using t ti_geo_s t_gto_t1 t_gto_t2 by auto
next
  case gt_diff
  note hd_t_gt_s = this(1) and chkvar = this(2) and chksubs = this(3)
  show ?thesis
    by (rule rpo_optim.gt_diff[OF rpo_optim hd_t_gt_s chkvar chkargs_if_chksubs[OF chksubs]])
next
  case gt_same
  note hd_t_eq_s = this(1) and chksubs = this(2) and extf = this(3)

  have extf_gto: ∀ f ∈ ground_heads (head t). extf f (>to) (args t) (args s)
```

```
    proof (rule ballI, rule extf_mono_strong[of _ _ (>t), rule_format])
      fix f
      assume f_in_ground: f ∈ ground_heads (head t)

      {
        fix ta sa
        assume ta_in: ta ∈ set (args t) and sa_in: sa ∈ set (args s) and ta_gt_sa: ta >t sa

        show ta >to sa
          by (rule ih[THEN iffD1, OF _ ta_gt_sa])
            (simp add: ta_in sa_in add_less_mono size_in_args)
      }

      show extf f (>t) (args t) (args s)
        using f_in_ground extf by simp
    qed

    show ?thesis
      by (rule rpo_optim.gt_same[OF rpo_optim hd_t_eq_s chkargs_if_chksubs[OF chksubs] extf_gto])
  qed
next
  assume t_gto_s: t >to s

  have chksubs_if_chkargs: chksubs (>t) t s if chkargs: chkargs (>to) t s
    unfolding chksubs_def
  proof (cases s, simp_all, rule conjI)
    fix s1 s2
    assume s: s = App s1 s2

    have s >to s1
      unfolding s by (rule rpo_optim.gt_sub_fun[OF rpo_optim])
    hence t_gto_s1: t >to s1
      by (rule rpo_optim.gt_trans[OF rpo_optim t_gto_s])
    show t >t s1
      by (rule ih[THEN iffD2, OF _ t_gto_s1]) (simp add: s)

    have t_gto_s2: t >to s2
      using chkargs unfolding rpo_optim.chkargs_def[OF rpo_optim] s by simp
    show t >t s2
      by (rule ih[THEN iffD2, OF _ t_gto_s2]) (simp add: s)
  qed

  show t >t s
  proof (cases rule: rpo_optim.gt.cases[OF rpo_optim t_gto_s,
      case_names gto_arg gto_diff gto_same])
    case (gto_arg ti)
    hence ti_in: ti ∈ set (args t) and ti_geo_s: ti ≥to s
      by auto
    obtain ζ ts where t: t = apps (Hd ζ) ts
      by (fact tm_exhaust_apps)

    {
      assume ti_gto_s: ti >to s
      hence ti_gt_s: ti >t s
        using ih[of ti s] size_in_args ti_in by auto
      moreover have t >t ti
        using sub_args[OF ti_in] gt_proper_sub size_in_args[OF ti_in] by blast
      ultimately have ?thesis
        using gt_trans by blast
    }
    moreover
    {
      assume ti = s
```

```
      hence ?thesis
        using sub_args[OF ti_in] gt_proper_sub size_in_args[OF ti_in] by blast
    }
    ultimately show ?thesis
      using ti_geo_s by blast
  next
    case gto_diff
    hence hd_t_gt_s: head t >_hd head s and chkvar: chkvar t s and
      chkargs: chkargs (>_to) t s
      by blast+

    have chksubs (>_t) t s
      by (rule chksubs_if_chkargs[OF chkargs])
    thus ?thesis
      by (rule gt_diff[OF hd_t_gt_s chkvar])
  next
    case gto_same
    hence hd_t_eq_s: head t = head s and chkargs: chkargs (>_to) t s and
      extf_gto: ∀f ∈ ground_heads (head t). extf f (>_to) (args t) (args s)
      by blast+

    have chksubs: chksubs (>_t) t s
      by (rule chksubs_if_chkargs[OF chkargs])

    have extf: ∀f ∈ ground_heads (head t). extf f (>_t) (args t) (args s)
    proof (rule ballI, rule extf_mono_strong[of _ _ (>_to), rule_format])
      fix f
      assume f_in_ground: f ∈ ground_heads (head t)

      {
        fix ta sa
        assume ta_in: ta ∈ set (args t) and sa_in: sa ∈ set (args s) and ta_gto_sa: ta >_to sa

        show ta >_t sa
          by (rule ih[THEN iffD2, OF _ ta_gto_sa])
            (simp add: ta_in sa_in add_less_mono size_in_args)
      }

      show extf f (>_to) (args t) (args s)
        using f_in_ground extf_gto by simp
    qed

    show ?thesis
      by (rule gt_same[OF hd_t_eq_s chksubs extf])
  qed
 qed
qed

end

end

end
```

9 An Encoding of Lambdas in Lambda-Free Higher-Order Logic

```
theory Lambda_Encoding
imports Lambda_Free_Term
begin
```

This theory defines an encoding of λ-expressions as λ-free higher-order terms.

```
locale lambda_encoding =
  fixes
```

lam :: *'s* **and**
db :: *nat* ⇒ *'s*
begin

definition *is_db* :: *'s* ⇒ *bool* **where**
is_db f ⟷ (∃ *i*. *f* = *db i*)

fun *subst_db* :: *nat* ⇒ *'v* ⇒ (*'s*, *'v*) *tm* ⇒ (*'s*, *'v*) *tm* **where**
subst_db i x (*Hd* ζ) = *Hd* (*if* ζ = *Var x then Sym* (*db i*) *else* ζ)
| *subst_db i x* (*App s t*) =
App (*subst_db i x s*) (*subst_db* (*if head s* = *Sym lam then i* + *1 else i*) *x t*)

definition *raw_db_subst* :: *nat* ⇒ *'v* ⇒ *'v* ⇒ (*'s*, *'v*) *tm* **where**
raw_db_subst i x = (λ*y*. *Hd* (*if y* = *x then Sym* (*db i*) *else Var y*))

lemma *vars_mset_subst_db*: *vars_mset* (*subst_db i x s*) = {#*y* ∈# *vars_mset s*. *y* ≠ *x*#}
by (*induct s arbitrary*: *i*) (*auto elim*: *hd.set_cases*)

lemma *head_subst_db*: *head* (*subst_db i x s*) = *head* (*subst* (*raw_db_subst i x*) *s*)
by (*induct s arbitrary*: *i*) (*auto simp*: *raw_db_subst_def split*: *hd.split*)

lemma *args_subst_db*:
args (*subst_db i x s*) = *map* (*subst_db* (*if head s* = *Sym lam then i* + *1 else i*) *x*) (*args s*)
by (*induct s arbitrary*: *i*) *auto*

lemma *var_mset_subst_db_subseteq*:
vars_mset s ⊆# *vars_mset t* ⟹ *vars_mset* (*subst_db i x s*) ⊆# *vars_mset* (*subst_db i x t*)
by (*simp add*: *vars_mset_subst_db multiset_filter_mono*)

end

end

10 Recursive Path Orders for Lambda-Free Higher-Order Terms

theory *Lambda_Free_RPOs*
imports *Lambda_Free_RPO_App Lambda_Free_RPO_Optim Lambda_Encoding*
begin

locale *simple_rpo_instances*
begin

definition *arity_sym* :: *nat* ⇒ *enat* **where**
arity_sym n = ∞

definition *arity_var* :: *nat* ⇒ *enat* **where**
arity_var n = ∞

definition *ground_head_var* :: *nat* ⇒ *nat set* **where**
ground_head_var x = *UNIV*

definition *gt_sym* :: *nat* ⇒ *nat* ⇒ *bool* **where**
gt_sym g f ⟷ *g* > *f*

sublocale *app*: *rpo_app gt_sym len_lexext*
by *unfold_locales* (*auto simp*: *gt_sym_def intro*: *wf_less*[*folded wfP_def*])

sublocale *std*: *rpo ground_head_var gt_sym* λ*f*. *len_lexext arity_sym arity_var*
by *unfold_locales* (*auto simp*: *arity_var_def arity_sym_def ground_head_var_def*)

sublocale *optim*: *rpo_optim ground_head_var gt_sym* λ*f*. *len_lexext arity_sym arity_var*
by *unfold_locales*

end

end