

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

Jasmin Christian Blanchette, Uwe Waldmann, and Daniel Wand

September 13, 2023

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	1
2	Utilities for Lambda-Free Orders	1
2.1	Finite Sets	1
2.2	Function Power	1
2.3	Least Operator	2
2.4	Antisymmetric Relations	2
2.5	Ayclic Relations	2
2.6	Reflexive, Transitive Closure	2
2.7	Well-Founded Relations	2
2.8	Wellorders	3
2.9	Lists	3
2.10	Extended Natural Numbers	4
2.11	Multisets	4
3	Lambda-Free Higher-Order Terms	6
3.1	Precedence on Symbols	6
3.2	Heads	6
3.3	Terms	7
3.4	hsize	9
3.5	Substitutions	10
3.6	Subterms	10
3.7	Maximum Arities	11
3.8	Potential Heads of Ground Instances of Variables	12
4	Infinite (Non-Well-Founded) Chains	13
5	Extension Orders	15
5.1	Locales	15
5.2	Lexicographic Extension	17
5.3	Reverse (Right-to-Left) Lexicographic Extension	18
5.4	Generic Length Extension	20
5.5	Length-Lexicographic Extension	20
5.6	Reverse (Right-to-Left) Length-Lexicographic Extension	21
5.7	Dershowitz–Manna Multiset Extension	23
5.8	Huet–Oppen Multiset Extension	24
5.9	Componentwise Extension	26

6	The Applicative Recursive Path Order for Lambda-Free Higher-Order Terms	28
7	The Graceful Recursive Path Order for Lambda-Free Higher-Order Terms	28
7.1	Setup	29
7.2	Inductive Definitions	29
7.3	Transitivity	30
7.4	Irreflexivity	30
7.5	Subterm Property	30
7.6	Compatibility with Functions	30
7.7	Compatibility with Arguments	30
7.8	Stability under Substitution	31
7.9	Totality on Ground Terms	31
7.10	Well-foundedness	31
8	The Optimized Graceful Recursive Path Order for Lambda-Free Higher-Order Terms	31
8.1	Setup	31
8.2	Definition of the Order	31
8.3	Transitivity	32
8.4	Conditional Equivalence with Unoptimized Version	32
9	An Encoding of Lambdas in Lambda-Free Higher-Order Logic	32
10	Recursive Path Orders for Lambda-Free Higher-Order Terms	33

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⟩
```

2.2 Function Power

```
lemma funpow_lesseq_iter:
fixes f :: ('a::order) ⇒ 'a
assumes mono: ∀k. k ≤ f k ∧ m_le_n: m ≤ n
shows (f ▷ m) k ≤ (f ▷ n) k
⟨proof⟩
```

```

lemma funpow_less_iter:
  fixes f :: ('a::order) ⇒ 'a
  assumes mono: ∀k. k < f k and m_lt_n: m < n
  shows (f ^ m) k < (f ^ n) k
  ⟨proof⟩

```

2.3 Least Operator

```

lemma Least_eq[simp]: (LEAST y. y = x) = x and (LEAST y. x = y) = x for x :: 'a::order
  ⟨proof⟩

```

```

lemma Least_in_nonempty_set_imp_ex:
  fixes f :: 'b ⇒ ('a::wellorder)
  assumes
    A_nemp: A ≠ {}
    P_least: P (LEAST y. ∃x ∈ A. y = f x)
  shows ∃x ∈ A. P (f x)
  ⟨proof⟩

```

```

lemma Least_eq_0_enat: P 0 ⇒ (LEAST x :: enat. P x) = 0
  ⟨proof⟩

```

2.4 Antisymmetric Relations

```

lemma irrefl_transp_imp_antisym: irrefl r ⇒ trans r ⇒ antisym r
  ⟨proof⟩

```

```

lemma irreflp_transp_imp_antisymP: irreflp p ⇒ transp p ⇒ antisymp p
  ⟨proof⟩

```

2.5 Acyclic Relations

```

lemma finite_nonempty_ex_succ_imp_cyclic:
  assumes
    fin: finite A and
    nemp: A ≠ {} and
    ex_y: ∀x ∈ A. ∃y ∈ A. (y, x) ∈ r
  shows ¬ acyclic r
  ⟨proof⟩

```

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⟩

```

```

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⟩

```

```

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*
  ⟨proof⟩

```

2.7 Well-Founded Relations

```

lemma wf_app: wf r ⇒ wf {(x, y). (f x, f y) ∈ r}
  ⟨proof⟩

```

```

lemma wfP_app: wfP p ⇒ wfP (λx y. p (f x) (f y))
  ⟨proof⟩

```

```

lemma wf_exists_minimal:
  assumes wf: wf r and Q: Q x
  shows ∃ x. Q x ∧ (∀ y. (f y, f x) ∈ r → ¬ Q y)
  ⟨proof⟩

lemma wfP_exists_minimal:
  assumes wf: wfP p and Q: Q x
  shows ∃ x. Q x ∧ (∀ y. p (f y) (f x) → ¬ Q y)
  ⟨proof⟩

lemma finite_irrefl_trans_imp_wf: finite r ==> irrefl r ==> trans r ==> wf r
  ⟨proof⟩

lemma finite_irreflp_transp_imp_wfp:
  finite {(x, y). p x y} ==> irreflp p ==> transp p ==> wfP p
  ⟨proof⟩

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 ∩ S ⊆ R
  shows ∃ k. ∀ i. (f (Suc (i + k)), f (i + k)) ∈ S
  ⟨proof⟩

```

2.8 Wellorders

```

lemma (in wellorder) exists_minimal:
  fixes x :: 'a
  assumes P x
  shows ∃ x. P x ∧ (∀ y. P y → y ≥ x)
  ⟨proof⟩

```

2.9 Lists

```

lemma rev_induct2[consumes 1, case_names Nil snoc]:
  length xs = length ys ==> P [] [] ==>
  (¬ ∃ x ys. length xs = length ys ==> P xs ys ==> P (xs @ [x]) (ys @ [y])) ==> P xs ys
  ⟨proof⟩

```

```

lemma hd_in_set: length xs ≠ 0 ==> hd xs ∈ set xs
  ⟨proof⟩

```

```

lemma in_lists_iff_set: xs ∈ lists A ↔ set xs ⊆ A
  ⟨proof⟩

```

```

lemma butlast_append_Cons[simp]: butlast (xs @ y # ys) = xs @ butlast (y # ys)
  ⟨proof⟩

```

```

lemma rev_in_lists[simp]: rev xs ∈ lists A ↔ xs ∈ lists A
  ⟨proof⟩

```

```

lemma hd_le_sum_list:
  fixes xs :: 'a::ordered_ab_semigroup_monoid_add_imp_le list
  assumes xs ≠ [] and ∀ i < length xs. xs ! i ≥ 0
  shows hd xs ≤ sum_list xs
  ⟨proof⟩

```

```

lemma sum_list_ge_length_times:
  fixes a :: 'a::{ordered_ab_semigroup_add, semiring_1}
  assumes ∀ i < length xs. xs ! i ≥ a
  shows sum_list xs ≥ of_nat (length xs) * a
  ⟨proof⟩

```

```

lemma prod_list_nonneg:
  fixes xs :: "('a :: {ordered_semiring_0, linordered_nonzero_semiring}) list"
  assumes "x ∈ set xs ⟹ x ≥ 0"
  shows "prod_list xs ≥ 0"
  ⟨proof⟩

lemma zip_append_0_up:
  zip (xs @ ys) [0..

```

```

lemma zip_eq_butlast_last:
  assumes "length xs > 0" and "length xs = length ys"
  shows "zip xs ys = zip (butlast xs) (butlast ys) @ [(last xs, last ys)]"
  ⟨proof⟩

```

2.10 Extended Natural Numbers

```

lemma the_enat_0[simp]: the_enat 0 = 0
  ⟨proof⟩

```

```

lemma the_enat_1[simp]: the_enat 1 = 1
  ⟨proof⟩

```

```

lemma enat_le_minus_1_imp_lt: m ≤ n - 1 ⟹ n ≠ ∞ ⟹ n ≠ 0 ⟹ m < n for m n :: enat
  ⟨proof⟩

```

```

lemma enat_diff_diff_eq: k - m - n = k - (m + n) for k m n :: enat
  ⟨proof⟩

```

```

lemma enat_sub_add_same[intro]: n ≤ m ⟹ m = m - n + n for m n :: enat
  ⟨proof⟩

```

```

lemma enat_the_enat_iden[simp]: n ≠ ∞ ⟹ enat (the_enat n) = n
  ⟨proof⟩

```

```

lemma the_enat_minus_nat: m ≠ ∞ ⟹ the_enat (m - enat n) = the_enat m - n
  ⟨proof⟩

```

```

lemma enat_the_enat_le: enat (the_enat x) ≤ x
  ⟨proof⟩

```

```

lemma enat_the_enat_minus_le: enat (the_enat (x - y)) ≤ x
  ⟨proof⟩

```

```

lemma enat_le_imp_minus_le: k ≤ m ⟹ k - n ≤ m for k m n :: enat
  ⟨proof⟩

```

```

lemma add_diff_assoc2_enat: m ≥ n ⟹ m - n + p = m + p - n for m n p :: enat
  ⟨proof⟩

```

```

lemma enat_mult_minus_distrib: enat x * (y - z) = enat x * y - enat x * z
  ⟨proof⟩

```

2.11 Multisets

```

lemma add_mset_lt_left_lt: a < b ⟹ add_mset a A < add_mset b A
  ⟨proof⟩

```

```

lemma add_mset_le_left_le: a ≤ b ⟹ add_mset a A ≤ add_mset b A for a :: 'a :: linorder
  ⟨proof⟩

```

```

lemma add_mset_lt_right_lt: A < B ⟹ add_mset a A < add_mset a B

```

$\langle proof \rangle$

lemma *add_mset_le_right_le*: $A \leq B \implies add_mset a A \leq add_mset a B$
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

lemma *add_mset_lt_lt_le*: $a < b \implies A \leq B \implies add_mset a A < add_mset b B$
 $\langle proof \rangle$

lemma *add_mset_lt_le_lt*: $a \leq b \implies A < B \implies add_mset a A < add_mset b B$ **for** $a :: 'a :: linorder$
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

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

lemma *nonempty_subseteq_mset_eq_singleton*: $M \neq \{\#\} \implies M \subseteq \{\#x\# \mid P\} \implies M = \{\#x\# \mid P\}$
 $\langle proof \rangle$

lemma *nonempty_subseteq_mset_iff_singleton*: $(M \neq \{\#\} \wedge M \subseteq \{\#x\# \mid P\}) \longleftrightarrow M = \{\#x\# \mid P\}$
 $\langle proof \rangle$

lemma *count_gt_imp_in_mset*[intro]: $count M x > n \implies x \in M$
 $\langle proof \rangle$

lemma *size_lt_imp_ex_count_lt*: $size M < size N \implies \exists x \in M. count M x < count N x$
 $\langle proof \rangle$

lemma *filter_filter_mset*[simp]: $\{#x \in M. P x\# \mid Q x\# \} = \{#x \in M. P x \wedge Q x\# \}$
 $\langle proof \rangle$

lemma *size_filter_unsat_elem*:
 assumes $x \in M$ **and** $\neg P x$
 shows $size \{#x \in M. P x\# \} < size M$
 $\langle proof \rangle$

lemma *size_filter_ne_elem*: $x \in M \implies size \{#y \in M. y \neq x\# \} < size M$
 $\langle proof \rangle$

lemma *size_eq_ex_count_lt*:
 assumes
 sz_m_eq_n: $size M = size N$ **and**
 m_eq_n: $M \neq N$
 shows $\exists x. count M x < count N x$
 $\langle proof \rangle$

lemma *count_image_mset_lt_imp_lt_raw*:
 assumes
 finite_A **and**
 A_set_mset: $A = set_mset M \cup set_mset N$ **and**
 count_image_mset_f: $count (image_mset f M) b < count (image_mset f N) b$
 shows $\exists x. f x = b \wedge count M x < count N x$
 $\langle proof \rangle$

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
⟨proof⟩

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
⟨proof⟩

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
⟨proof⟩

lemma Max_in_mset: M ≠ {#} ==> Max_mset M ∈# M
⟨proof⟩

lemma Max_lt_imp_lt_mset:
assumes n_neq: N ≠ {#} and max: Max_mset M < Max_mset N (is ?max_M < ?max_N)
shows M < N
⟨proof⟩

lemma fold_mset_singleton[simp]: fold_mset f z {#x#} = f x z
⟨proof⟩

end

```

3 Lambda-Free Higher-Order Terms

```

theory Lambda_Free_Term
imports Lambda_Free_Util
abbrevs >s = >s
and >h = >hd
and ≤≥h = ≤≥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 ⇒ 's ⇒ bool (infix >s 50)
assumes
gt_sym_irrefl: ¬ f >s f and
gt_sym_trans: h >s g ==> g >s f ==> h >s f and
gt_sym_total: f >s g ∨ g >s f ∨ g = f and
gt_sym_wf: wfP (λf g. g >s f)
begin

lemma gt_sym_antisym: f >s g ==> ¬ g >s f
⟨proof⟩

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  $\zeta$ 

```

```

lemma finite_vars_hd[simp]: finite (vars_hd  $\zeta$ )
   $\langle$ proof $\rangle$ 

```

```

lemma finite_syms_hd[simp]: finite (syms_hd  $\zeta$ )
   $\langle$ proof $\rangle$ 

```

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  $\zeta$ ) = Hd  $\zeta$ 
| arg (Hd  $\zeta$ ) = Hd  $\zeta$ 

```

```

overloading head0  $\equiv$  head0 :: ('s, 'v) tm  $\Rightarrow$  ('s, 'v) hd
begin

```

```

primrec head0 :: ('s, 'v) tm  $\Rightarrow$  ('s, 'v) hd where
  head0 (Hd  $\zeta$ ) =  $\zeta$ 
| head0 (App s _) = head0 s

```

```

end

```

```

lemma head_App[simp]: head (App s t) = head s
   $\langle$ proof $\rangle$ 

```

```

declare tm.sel(2)[simp del]

```

```

lemma head_fun[simp]: head (fun s) = head s
   $\langle$ proof $\rangle$ 

```

```

abbreviation ground :: ('s, 'v) tm  $\Rightarrow$  bool where
  ground t  $\equiv$  vars t = {}

```

```

abbreviation is_App :: ('s, 'v) tm  $\Rightarrow$  bool where
  is_App s  $\equiv$   $\neg$  is_Hd s

```

```

lemma
  size_fun_lt: is_App s  $\implies$  size (fun s) < size s and
  size_arg_lt: is_App s  $\implies$  size (arg s) < size s
   $\langle$ proof $\rangle$ 

```

```

lemma
  finite_vars[simp]: finite (vars s) and
  finite_syms[simp]: finite (syms s)
   $\langle$ proof $\rangle$ 

```

```

lemma
  vars_head_subseteq: vars_hd (head s)  $\subseteq$  vars s and
  syms_head_subseteq: syms_hd (head s)  $\subseteq$  syms s
   $\langle$ proof $\rangle$ 

```

```

fun args :: ('s, 'v) tm  $\Rightarrow$  ('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)
  ⟨proof⟩

lemma arg_in_args: is_App s ⇒ arg s ∈ set (args s)
  ⟨proof⟩

lemma
  vars_args_subseteq: si ∈ set (args s) ⇒ vars si ⊆ vars s and
  syms_args_subseteq: si ∈ set (args s) ⇒ syms si ⊆ syms s
  ⟨proof⟩

lemma args_Nil_iff_is_Hd: args s = [] ↔ is_Hd s
  ⟨proof⟩

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

lemma size_ge_num_args: size s ≥ num_args s
  ⟨proof⟩

lemma Hd_head_id: num_args s = 0 ⇒ Hd (head s) = s
  ⟨proof⟩

lemma one_arg_imp_Hd: num_args s = 1 ⇒ s = App t u ⇒ t = Hd (head t)
  ⟨proof⟩

lemma size_in_args: s ∈ set (args t) ⇒ size s < size t
  ⟨proof⟩

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)
  ⟨proof⟩

lemma apps_append[simp]: apps s (ss @ ts) = apps (apps s ss) ts
  ⟨proof⟩

lemma App_apps: App (apps s ts) t = apps s (ts @ [t])
  ⟨proof⟩

lemma tm_inject_apps[iff, induct_simp]: apps (Hd ζ) ss = apps (Hd ξ) ts ↔ ζ = ξ ∧ ss = ts
  ⟨proof⟩

lemma tm_collapse_apps[simp]: apps (Hd (head s)) (args s) = s
  ⟨proof⟩

lemma tm_expand_apps: head s = head t ⇒ args s = args t ⇒ s = t
  ⟨proof⟩

lemma tm_exhaust_apps_sel[case_names apps]: (s = apps (Hd (head s)) (args s) ⇒ P) ⇒ P

```

$\langle proof \rangle$

lemma *tm_exhaust_apps*[*case_names apps*]: $(\bigwedge \zeta \text{ ss. } s = \text{apps}(\text{Hd } \zeta) \text{ ss} \implies P) \implies P$
 $\langle proof \rangle$

lemma *tm_induct_apps*[*case_names apps*]:
assumes $\bigwedge \zeta \text{ ss. } (\bigwedge s. s \in \text{set ss} \implies P s) \implies P (\text{apps}(\text{Hd } \zeta) \text{ ss})$
shows $P s$
 $\langle proof \rangle$

lemma
ground_fun: $\text{ground } s \implies \text{ground } (\text{fun } s)$ **and**
ground_arg: $\text{ground } s \implies \text{ground } (\text{arg } s)$
 $\langle proof \rangle$

lemma *ground_head*: $\text{ground } s \implies \text{is_Sym } (\text{head } s)$
 $\langle proof \rangle$

lemma *ground_args*: $t \in \text{set } (\text{args } s) \implies \text{ground } s \implies \text{ground } t$
 $\langle proof \rangle$

primrec *vars_mset* :: $('s, 'v) \text{ tm} \Rightarrow 'v \text{ multiset where}$
 $\text{vars_mset } (\text{Hd } \zeta) = \text{mset_set } (\text{vars_hd } \zeta)$
 $\mid \text{vars_mset } (\text{App } s t) = \text{vars_mset } s + \text{vars_mset } t$

lemma *set_vars_mset*[*simp*]: $\text{set_mset } (\text{vars_mset } t) = \text{vars_mset } t$
 $\langle proof \rangle$

lemma *vars_mset_empty_iff*[*iff*]: $\text{vars_mset } s = \{\#\} \longleftrightarrow \text{ground } s$
 $\langle proof \rangle$

lemma *vars_mset_fun*[*intro*]: $\text{vars_mset } (\text{fun } t) \subseteq \# \text{ vars_mset } t$
 $\langle proof \rangle$

lemma *vars_mset_arg*[*intro*]: $\text{vars_mset } (\text{arg } t) \subseteq \# \text{ vars_mset } t$
 $\langle proof \rangle$

3.4 hsize

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

primrec *hsize* :: $('s, 'v) \text{ tm} \Rightarrow \text{nat where}$
 $\text{hsize } (\text{Hd } \zeta) = 1$
 $\mid \text{hsize } (\text{App } s t) = \text{hsize } s + \text{hsize } t$

lemma *hsize_size*: $\text{hsize } t * 2 = \text{size } t + 1$
 $\langle proof \rangle$

lemma *hsize_pos*[*simp*]: $\text{hsize } t > 0$
 $\langle proof \rangle$

lemma *hsize_fun_lt*: $\text{is_App } s \implies \text{hsize } (\text{fun } s) < \text{hsize } s$
 $\langle proof \rangle$

lemma *hsize_arg_lt*: $\text{is_App } s \implies \text{hsize } (\text{arg } s) < \text{hsize } s$
 $\langle proof \rangle$

lemma *hsize_ge_num_args*: $\text{hsize } s \geq \text{hsize } s$
 $\langle proof \rangle$

lemma *hsize_in_args*: $s \in \text{set } (\text{args } t) \implies \text{hsize } s < \text{hsize } t$
 $\langle proof \rangle$

lemma *hsize_apps*: $\text{hsize } (\text{apps } t ts) = \text{hsize } t + \text{sum_list } (\text{map } \text{hsize } ts)$

$\langle proof \rangle$

lemma *hsizem_args*: $1 + \text{sum_list}(\text{map hsize}(\text{args } t)) = \text{hsize } t$
 $\langle proof \rangle$

3.5 Substitutions

primrec *subst* :: $('v \Rightarrow ('s, 'v) \text{ tm}) \Rightarrow ('s, 'v) \text{ tm} \Rightarrow ('s, 'v) \text{ tm}$ **where**
 $\text{subst } \varrho (\text{Hd } \zeta) = (\text{case } \zeta \text{ of Var } x \Rightarrow \varrho x \mid \text{Sym } f \Rightarrow \text{Hd}(\text{Sym } f))$
 $\mid \text{subst } \varrho (\text{App } s t) = \text{App}(\text{subst } \varrho s)(\text{subst } \varrho t)$

lemma *subst_apps[simp]*: $\text{subst } \varrho (\text{apps } s ts) = \text{apps}(\text{subst } \varrho s)(\text{map}(\text{subst } \varrho) ts)$
 $\langle proof \rangle$

lemma *head_subst[simp]*: $\text{head}(\text{subst } \varrho s) = \text{head}(\text{subst } \varrho (\text{Hd}(\text{head } s)))$
 $\langle proof \rangle$

lemma *args_subst[simp]*:
 $\text{args}(\text{subst } \varrho s) = (\text{case head } s \text{ of Var } x \Rightarrow \text{args}(\varrho x) \mid \text{Sym } f \Rightarrow []) @ \text{map}(\text{subst } \varrho)(\text{args } s)$
 $\langle proof \rangle$

lemma *ground_imp_subst_iden*: $\text{ground } s \Rightarrow \text{subst } \varrho s = s$
 $\langle proof \rangle$

lemma *vars_mset_subst[simp]*: $\text{vars_mset}(\text{subst } \varrho s) = (\sum_{x \in \text{vars_mset } s} \#\{\text{vars_mset}(\varrho x)\})$
 $\langle proof \rangle$

lemma *vars_mset_subst_subseteq*:
 $\text{vars_mset } t \supseteq \text{vars_mset } s \Rightarrow \text{vars_mset}(\text{subst } \varrho t) \supseteq \text{vars_mset}(\text{subst } \varrho s)$
 $\langle proof \rangle$

lemma *vars_subst_subseteq*: $\text{vars } t \supseteq \text{vars } s \Rightarrow \text{vars}(\text{subst } \varrho t) \supseteq \text{vars}(\text{subst } \varrho s)$
 $\langle proof \rangle$

3.6 Subterms

inductive *sub* :: $('s, 'v) \text{ tm} \Rightarrow ('s, 'v) \text{ tm} \Rightarrow \text{bool}$ **where**
 $\text{sub_refl}: \text{sub } s s$
 $\mid \text{sub_fun}: \text{sub } s t \Rightarrow \text{sub } s (\text{App } u t)$
 $\mid \text{sub_arg}: \text{sub } s t \Rightarrow \text{sub } s (\text{App } t u)$

inductive-cases *sub_HdE[simplified, elim]*: $\text{sub } s (\text{Hd } \xi)$
inductive-cases *sub_AppE[simplified, elim]*: $\text{sub } s (\text{App } t u)$
inductive-cases *sub_Hd_HdE[simplified, elim]*: $\text{sub } (\text{Hd } \zeta) (\text{Hd } \xi)$
inductive-cases *sub_Hd_AppE[simplified, elim]*: $\text{sub } (\text{Hd } \zeta) (\text{App } t u)$

lemma *in_vars_imp_sub*: $x \in \text{vars } s \leftrightarrow \text{sub}(\text{Hd}(\text{Var } x)) s$
 $\langle proof \rangle$

lemma *sub_args*: $s \in \text{set}(\text{args } t) \Rightarrow \text{sub } s t$
 $\langle proof \rangle$

lemma *sub_size*: $\text{sub } s t \Rightarrow \text{size } s \leq \text{size } t$
 $\langle proof \rangle$

lemma *sub_subst*: $\text{sub } s t \Rightarrow \text{sub}(\text{subst } \varrho s)(\text{subst } \varrho t)$
 $\langle proof \rangle$

abbreviation *proper_sub* :: $('s, 'v) \text{ tm} \Rightarrow ('s, 'v) \text{ tm} \Rightarrow \text{bool}$ **where**
 $\text{proper_sub } s t \equiv \text{sub } s t \wedge s \neq t$

lemma *proper_sub_Hd[simp]*: $\neg \text{proper_sub } s (\text{Hd } \zeta)$
 $\langle proof \rangle$

```

lemma proper_sub_subst:
  assumes psub: proper_sub s t
  shows proper_sub (subst ρ s) (subst ρ t)
  ⟨proof⟩

```

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
  ⟨proof⟩

lemma arity_apps[simp]: arity (apps s ts) = arity s − length ts
  ⟨proof⟩

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)
  ⟨proof⟩

lemma wary_arg[intro]: wary t ⇒ wary (arg t)
  ⟨proof⟩

lemma wary_args: s ∈ set (args t) ⇒ wary t ⇒ wary s
  ⟨proof⟩

lemma wary_sub: sub s t ⇒ wary t ⇒ wary s
  ⟨proof⟩

lemma wary_inf_ary: (¬ ∃ ζ. arity_hd ζ = ∞) ⇒ wary s
  ⟨proof⟩

lemma wary_num_args_le_arity_head: wary s ⇒ num_args s ≤ arity_hd (head s)
  ⟨proof⟩

lemma wary_apps:
  wary s ⇒ (∀ sa. sa ∈ set ss ⇒ wary sa) ⇒ length ss ≤ arity s ⇒ wary (apps s ss)
  ⟨proof⟩

lemma wary_cases_apps[consumes 1, case_names apps]:
  assumes
    wary_t: wary t and
    apps: ∀ ζ ss. t = apps (Hd ζ) ss ⇒ (∀ sa. sa ∈ set ss ⇒ wary sa) ⇒ length ss ≤ arity_hd ζ ⇒ P
  shows P
  ⟨proof⟩

```

```

lemma arity_hd_head: wary s  $\Rightarrow$  arity_hd (head s) = arity s + num_args s
  ⟨proof⟩

lemma arity_head_ge: arity_hd (head s)  $\geq$  arity s
  ⟨proof⟩

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

lemma wary_fo_args: s  $\in$  set (args t)  $\Rightarrow$  wary_fo t  $\Rightarrow$  wary_fo s
  ⟨proof⟩

lemma wary_fo_arg: wary_fo (App s t)  $\Rightarrow$  wary_fo t
  ⟨proof⟩

end

```

3.8 Potential Heads of Ground Instances of Variables

```

locale ground_heads = gt_sym (>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
    ground_heads_var :: 'v  $\Rightarrow$  's set
  assumes
    ground_heads_var_arity: f  $\in$  ground_heads_var x  $\Rightarrow$  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  $\zeta$   $\Rightarrow$  arity_sym f  $\geq$  arity_hd  $\zeta$ 
  ⟨proof⟩

lemma ground_heads_nonempty[simp]: ground_heads  $\zeta$   $\neq$  {}
  ⟨proof⟩

lemma sym_in_ground_heads: is_Sym  $\zeta$   $\Rightarrow$  sym  $\zeta$   $\in$  ground_heads  $\zeta$ 
  ⟨proof⟩

lemma ground_hd_in_ground_heads: ground s  $\Rightarrow$  sym (head s)  $\in$  ground_heads (head s)
  ⟨proof⟩

lemma some_ground_head_arity: arity_sym (SOME f. f  $\in$  ground_heads (Var x))  $\geq$  arity_var x
  ⟨proof⟩

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  $\varrho$   $\Rightarrow$  wary_subst  $\varrho$ 
  ⟨proof⟩

```

```

lemma wary_subst_wary:
  assumes wary_ρ: wary_subst ρ and wary_s: wary s
  shows wary (subst ρ s)
  ⟨proof⟩

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⟩

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)
  ⟨proof⟩

lemma strict_wary_grounding_ρ: strict_wary_subst grounding_ρ
  ⟨proof⟩

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: ¬ ξ >_hd ξ
  ⟨proof⟩

lemma gt_hd_trans: χ >_hd ξ ⇒ ξ >_hd ζ ⇒ χ >_hd ζ
  ⟨proof⟩

lemma gt_sym_imp_hd: g >_s f ⇒ Sym g >_hd Sym f
  ⟨proof⟩

lemma not_comp_hd_imp_Var: ¬ ξ ≤_hd ζ ⇒ is_Var ζ ∨ is_Var ξ
  ⟨proof⟩

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 ( $\lambda x y. p y x$ )  $\longleftrightarrow$  ( $\nexists f. \text{inf\_chain } f$ )
  ⟨proof⟩

lemma inf_chain_offset: inf_chain f  $\implies$  inf_chain ( $\lambda j. f (j + i)$ )
  ⟨proof⟩

definition bad :: 'a  $\Rightarrow$  bool where
  bad x  $\longleftrightarrow$  ( $\exists f. \text{inf\_chain } f \wedge f 0 = x$ )

lemma inf_chain_bad:
  assumes bad_f: inf_chain f
  shows bad (f i)
  ⟨proof⟩

context
  fixes gt :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool
  assumes wf: wf {(x, y). gt y x}
begin

primrec worst_chain :: nat  $\Rightarrow$  'a where
  worst_chain 0 = (SOME x. bad x  $\wedge$  ( $\forall y. \text{bad } y \longrightarrow \neg \text{gt } x y$ ))
| worst_chain (Suc i) = (SOME x. bad x  $\wedge$  p (worst_chain i) x  $\wedge$ 
  ( $\forall y. \text{bad } y \wedge p (\text{worst\_chain } i) y \longrightarrow \neg \text{gt } x y$ ))

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:  $\neg \text{gt} (\text{worst\_chain } 0) x$ 
  ⟨proof⟩

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  $\implies$   $\neg \text{gt} (\text{worst\_chain } (\text{Suc } i)) x$ 
  ⟨proof⟩

lemma bad_worst_chain: bad (worst_chain i)
  ⟨proof⟩

lemma worst_chain_bad: inf_chain worst_chain
  ⟨proof⟩

end

context
  fixes x :: 'a
  assumes
    x_bad: bad x and
    p_trans:  $\bigwedge z y x. p z y \implies p y x \implies p z x$ 
begin

lemma worst_chain_not_gt:  $\neg \text{gt} (\text{worst\_chain } i) (\text{worst\_chain } (\text{Suc } i))$  for i
  ⟨proof⟩

end

end

```

```

end

lemma inf_chain_subset: inf_chain p f  $\implies$  p  $\leq$  q  $\implies$  inf_chain q f
  ⟨proof⟩

```

```
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:  $(\forall y \in \text{set } ys. \forall x \in \text{set } xs. gt y x \longrightarrow gt' y x) \implies ext gt ys xs \implies ext gt' ys xs$  and
    map_finite A  $\implies$  ys  $\in$  lists A  $\implies$  xs  $\in$  lists A  $\implies$   $(\forall x \in A. \neg gt(f x)(f x)) \implies$ 
     $(\forall z \in A. \forall y \in A. \forall x \in A. gt(f z)(f y) \longrightarrow gt(f y)(f x) \longrightarrow gt(f z)(f x)) \implies$ 
     $(\forall y \in A. \forall x \in A. gt y x \longrightarrow gt(f y)(f x)) \implies ext gt ys xs \implies ext gt(\text{map } f ys)(\text{map } f xs)$ 
begin
```

```
lemma mono[mono]:  $gt \leq gt' \implies ext gt \leq ext gt'$ 
  ⟨proof⟩
```

```
end
```

```
locale ext_irrefl = ext +
  assumes irrefl:  $(\forall x \in \text{set } xs. \neg gt x x) \implies \neg ext gt xs xs$ 
```

```
locale ext_trans = ext +
  assumes trans:  $zs \in \text{lists } A \implies ys \in \text{lists } A \implies xs \in \text{lists } A \implies$ 
 $(\forall z \in A. \forall y \in A. \forall x \in A. gt z y \longrightarrow gt y x \longrightarrow gt z x) \implies ext gt zs ys \implies ext gt ys xs \implies ext gt zs xs$ 
```

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

```
locale ext_trans_before_irrefl = ext_trans +
  assumes irrefl_from_trans:  $(\forall z \in \text{set } zs. \forall y \in \text{set } ys. \forall x \in \text{set } xs. gt z y \longrightarrow gt y x \longrightarrow gt z x) \implies$ 
 $(\forall x \in \text{set } xs. \neg gt x x) \implies \neg ext gt xs xs$ 
```

```
locale ext_irrefl_trans_strong = ext_irrefl +
  assumes trans_strong:  $(\forall z \in \text{set } zs. \forall y \in \text{set } ys. \forall x \in \text{set } xs. gt z y \longrightarrow gt y x \longrightarrow gt z x) \implies$ 
 $ext gt zs ys \implies ext gt ys xs \implies ext gt zs xs$ 
```

```
sublocale ext_irrefl_trans_strong < ext_irrefl_before_trans
  ⟨proof⟩
```

```
sublocale ext_irrefl_trans_strong < ext_trans
  ⟨proof⟩
```

```
sublocale ext_irrefl_trans_strong < ext_trans_before_irrefl
```

```

⟨proof⟩

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

locale ext_compat_cons = ext +
assumes compat_cons: ext gt ys xs ==> ext gt (x # ys) (x # xs)
begin

lemma compat_append_left: ext gt ys xs ==> ext gt (zs @ ys) (zs @ xs)
⟨proof⟩

end

locale ext_compat_snoc = ext +
assumes compat_snoc: ext gt ys xs ==> ext gt (ys @ [x]) (xs @ [x])
begin

lemma compat_append_right: ext gt ys xs ==> ext gt (ys @ zs) (xs @ zs)
⟨proof⟩

end

locale ext_compat_list = ext +
assumes compat_list: y ≠ x ==> gt y x ==> ext gt (xs @ y # xs') (xs @ x # xs')

locale ext_singleton = ext +
assumes singleton: y ≠ x ==> ext gt [y] [x] ↔ gt y x

locale ext_compat_list_strong = ext_compat_cons + ext_compat_snoc + ext_singleton
begin

lemma compat_list: y ≠ x ==> gt y x ==> ext gt (xs @ y # xs') (xs @ x # xs')
⟨proof⟩

end

sublocale ext_compat_list_strong < ext_compat_list
⟨proof⟩

locale ext_total = ext +
assumes total: (∀ y ∈ A. ∀ x ∈ A. gt y x ∨ gt x y ∨ y = x) ==> ys ∈ lists A ==> xs ∈ lists A ==>
ext gt ys xs ∨ ext gt xs ys ∨ ys = xs

locale ext_wf = ext +
assumes wf: wfP (λx y. gt y x) ==> wfP (λxs ys. ext gt ys xs)

locale ext_hd_or_tl = ext +
assumes hd_or_tl: (∀ z y x. gt z y → gt y x → gt z x) ==> (∀ y x. gt y x ∨ gt x y ∨ y = x) ==>
length ys = length xs ==> ext gt (y # ys) (x # xs) ==> gt y x ∨ ext gt ys xs

locale ext_wf_bounded = ext_irrefl_before_trans + ext_hd_or_tl
begin

context
fixes gt :: 'a ⇒ 'a ⇒ bool
assumes
gt_irrefl: ∀z. ¬ gt z z and
gt_trans: ∀z y x. gt z y ==> gt y x ==> gt z x and
gt_total: ∀y x. gt y x ∨ gt x y ∨ y = x and
gt_wf: wfP (λx y. gt y x)
begin

```

```

lemma irrefl_gt:  $\neg \text{ext } gt \text{ xs xs}$ 
   $\langle \text{proof} \rangle$ 

lemma trans_gt:  $\text{ext } gt \text{ zs ys} \implies \text{ext } gt \text{ ys xs} \implies \text{ext } gt \text{ zs xs}$ 
   $\langle \text{proof} \rangle$ 

lemma hd_or_tl_gt:  $\text{length } ys = \text{length } xs \implies \text{ext } gt \text{ (y \# ys) (x \# xs)} \implies gt \text{ y x} \vee \text{ext } gt \text{ ys xs}$ 
   $\langle \text{proof} \rangle$ 

lemma wf_same_length_if_total:  $wfP (\lambda xs \text{ ys}. \text{length } ys = n \wedge \text{length } xs = n \wedge \text{ext } gt \text{ ys xs})$ 
   $\langle \text{proof} \rangle$ 

lemma wf_bounded_if_total:  $wfP (\lambda xs \text{ ys}. \text{length } ys \leq n \wedge \text{length } xs \leq n \wedge \text{ext } gt \text{ ys xs})$ 
   $\langle \text{proof} \rangle$ 

end

context
  fixes gt :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool
  assumes
    gt_irrefl:  $\bigwedge z. \neg gt \text{ z z}$  and
    gt_wf:  $wfP (\lambda x \text{ y}. gt \text{ y x})$ 
begin

lemma wf_bounded:  $wfP (\lambda xs \text{ ys}. \text{length } ys \leq n \wedge \text{length } xs \leq n \wedge \text{ext } gt \text{ ys xs})$ 
   $\langle \text{proof} \rangle$ 

end

end

```

5.2 Lexicographic Extension

```

inductive lexext :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool for gt where
  lexext_Nil: lexext gt (y # ys) []
| lexext_Cons: gt y x  $\implies$  lexext gt (y # ys) (x # xs)
| lexext_Cons_eq: lexext gt ys xs  $\implies$  lexext gt (x # ys) (x # xs)

lemma lexext.simps[simp]:
  lexext gt ys []  $\longleftrightarrow$  ys  $\neq$  []
   $\neg$  lexext gt [] xs
  lexext gt (y # ys) (x # xs)  $\longleftrightarrow$  gt y x  $\vee$  x = y  $\wedge$  lexext gt ys xs
   $\langle \text{proof} \rangle$ 

lemma lexext_mono_strong:
  assumes
     $\forall y \in \text{set } ys. \forall x \in \text{set } xs. gt \text{ y x} \longrightarrow gt' \text{ y x}$  and
    lexext gt ys xs
  shows lexext gt' ys xs
   $\langle \text{proof} \rangle$ 

lemma lexext_map_strong:
   $(\forall y \in \text{set } ys. \forall x \in \text{set } xs. gt \text{ y x} \longrightarrow gt (f y) (f x)) \implies \text{lexext gt ys xs} \implies$ 
  lexext gt (map f ys) (map f xs)
   $\langle \text{proof} \rangle$ 

lemma lexext_irrefl:
  assumes  $\forall x \in \text{set } xs. \neg gt \text{ x x}$ 
  shows  $\neg \text{lexext gt xs xs}$ 
   $\langle \text{proof} \rangle$ 

lemma lexext_trans_strong:
  assumes
     $\forall z \in \text{set } zs. \forall y \in \text{set } ys. \forall x \in \text{set } xs. gt \text{ z y} \longrightarrow gt \text{ y x} \longrightarrow gt \text{ z x}$  and

```

```

lexext gt zs ys and lexext gt ys xs
shows lexext gt zs xs
⟨proof⟩

lemma lexext_snoc: lexext gt (xs @ [x]) xs
⟨proof⟩

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])
⟨proof⟩

lemma lexext_compat_list: gt y x  $\implies$  lexext gt (xs @ y # xs') (xs @ x # xs')
⟨proof⟩

lemma lexext_singleton: lexext gt [y] [x]  $\longleftrightarrow$  gt y x
⟨proof⟩

lemma lexext_total: ( $\forall y \in B. \forall x \in A. gt y x \vee gt x y \vee y = x$ )  $\implies$  ys  $\in$  lists B  $\implies$  xs  $\in$  lists A  $\implies$ 
  lexext gt ys xs  $\vee$  lexext gt xs ys  $\vee$  ys = xs
⟨proof⟩

lemma lexext_hd_or_tl: lexext gt (y # ys) (x # xs)  $\implies$  gt y x  $\vee$  lexext gt ys xs
⟨proof⟩

interpretation lexext: ext lexext
⟨proof⟩

interpretation lexext: ext_irrefl_trans_strong lexext
⟨proof⟩

interpretation lexext: ext_snoc lexext
⟨proof⟩

interpretation lexext: ext_compat_cons lexext
⟨proof⟩

interpretation lexext: ext_compat_list lexext
⟨proof⟩

interpretation lexext: ext_singleton lexext
⟨proof⟩

interpretation lexext: ext_total lexext
⟨proof⟩

interpretation lexext: ext_hd_or_tl lexext
⟨proof⟩

interpretation lexext: ext_wf_bounded lexext
⟨proof⟩

```

5.3 Reverse (Right-to-Left) Lexicographic Extension

abbreviation lexext_rev :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a list \Rightarrow 'a list \Rightarrow bool **where**
 $lexext_rev\ gt\ ys\ xs \equiv lexext\ gt\ (rev\ ys)\ (rev\ xs)$

```

lemma lexext_rev_simp[simp]:
  lexext_rev gt ys []  $\longleftrightarrow$  ys  $\neq$  []
   $\neg$  lexext_rev gt [] xs
  lexext_rev gt (ys @ [y]) (xs @ [x])  $\longleftrightarrow$  gt y x  $\vee$  x = y  $\wedge$  lexext_rev gt ys xs
⟨proof⟩

```

```

lemma lexext_rev_cons_cons:
  assumes length ys = length xs
  shows lexext_rev gt (y # ys) (x # xs)  $\longleftrightarrow$  lexext_rev gt ys xs  $\vee$  ys = xs  $\wedge$  gt y x
  (proof)

lemma lexext_rev_mono_strong:
  assumes
     $\forall y \in \text{set } ys. \forall x \in \text{set } xs. gt y x \longrightarrow gt' y x$  and
    lexext_rev gt ys xs
  shows lexext_rev gt' ys xs
  (proof)

lemma lexext_rev_map_strong:
  ( $\forall y \in \text{set } ys. \forall x \in \text{set } xs. gt y x \longrightarrow gt (f y) (f x)) \implies lexext_rev gt ys xs \implies$ 
  lexext_rev gt (map f ys) (map f xs)
  (proof)

lemma lexext_rev_irrefl:
  assumes  $\forall x \in \text{set } xs. \neg gt x x$ 
  shows  $\neg lexext\_rev\ gt\ xs\ xs$ 
  (proof)

lemma lexext_rev_trans_strong:
  assumes
     $\forall z \in \text{set } zs. \forall y \in \text{set } ys. \forall x \in \text{set } xs. gt z y \longrightarrow gt y x \longrightarrow gt z x$  and
    lexext_rev gt zs ys and lexext_rev gt ys xs
  shows lexext_rev gt zs xs
  (proof)

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)
  (proof)

lemma lexext_rev_compat_snoc: lexext_rev gt ys xs  $\implies$  lexext_rev gt (ys @ [x]) (xs @ [x])
  (proof)

lemma lexext_rev_compat_list: gt y x  $\implies$  lexext_rev gt (xs @ y # xs') (xs @ x # xs')
  (proof)

lemma lexext_rev_singleton: lexext_rev gt [y] [x]  $\longleftrightarrow$  gt y x
  (proof)

lemma lexext_rev_total:
  ( $\forall y \in B. \forall x \in A. gt y x \vee gt x y \vee y = x) \implies ys \in lists B \implies xs \in lists A \implies$ 
  lexext_rev gt ys xs  $\vee$  lexext_rev gt xs ys  $\vee$  ys = xs
  (proof)

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
  (proof)

interpretation lexext_rev: ext lexext_rev
  (proof)

interpretation lexext_rev: ext_irrefl_trans_strong lexext_rev
  (proof)

interpretation lexext_rev: ext_compat_snoc lexext_rev

```

$\langle proof \rangle$

interpretation `lexext_rev`: `ext_compat_list lexext_rev`
 $\langle proof \rangle$

interpretation `lexext_rev`: `ext_singleton lexext_rev`
 $\langle proof \rangle$

interpretation `lexext_rev`: `ext_total lexext_rev`
 $\langle proof \rangle$

interpretation `lexext_rev`: `ext_hd_or_tl lexext_rev`
 $\langle proof \rangle$

interpretation `lexext_rev`: `ext_wf_bounded lexext_rev`
 $\langle proof \rangle$

5.4 Generic Length Extension

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

lemma

`lenext_mono_strong`: $(gts \text{ ys xs} \implies gts' \text{ ys xs}) \implies \text{lenext } gts \text{ ys xs} \implies \text{lenext } gts' \text{ ys xs}$ **and**
`lenext_map_strong`: $(\text{length ys} = \text{length xs} \implies gts \text{ ys xs} \implies gts (\text{map f ys}) (\text{map f xs})) \implies$
 $\text{lenext } gts \text{ ys xs} \implies \text{lenext } gts (\text{map f ys}) (\text{map f xs})$ **and**
`lenext_irrefl`: $\neg gts \text{ ys xs} \implies \neg \text{lenext } gts \text{ ys xs}$ **and**
`lenext_trans`: $(gts \text{ zs ys} \implies gts \text{ ys xs} \implies gts \text{ zs xs}) \implies \text{lenext } gts \text{ zs ys} \implies \text{lenext } gts \text{ ys xs} \implies$
 $\text{lenext } gts \text{ zs xs}$ **and**
`lenext_snoc`: $\text{lenext } gts (\text{xs @ [x]}) \text{ xs}$ **and**
`lenext_compat_cons`: $(\text{length ys} = \text{length xs} \implies gts \text{ ys xs} \implies gts (x \# ys) (x \# xs)) \implies$
 $\text{lenext } gts \text{ ys xs} \implies \text{lenext } gts (x \# ys) (x \# xs)$ **and**
`lenext_compat_snoc`: $(\text{length ys} = \text{length xs} \implies gts \text{ ys xs} \implies gts (ys @ [x]) (xs @ [x])) \implies$
 $\text{lenext } gts \text{ ys xs} \implies \text{lenext } gts (ys @ [x]) (xs @ [x])$ **and**
`lenext_compat_list`: $gts (\text{xs @ y} \# \text{xs}') (\text{xs @ x} \# \text{xs}') \implies$
 $\text{lenext } gts (\text{xs @ y} \# \text{xs}') (\text{xs @ x} \# \text{xs}')$ **and**
`lenext_singleton`: $\text{lenext } gts [y] [x] \longleftrightarrow gts [y] [x]$ **and**
`lenext_total`: $(gts \text{ ys xs} \vee gts \text{ xs ys} \vee ys = xs) \implies$
 $\text{lenext } gts \text{ ys xs} \vee \text{lenext } gts \text{ xs ys} \vee ys = xs$ **and**
`lenext_hd_or_tl`: $(\text{length ys} = \text{length xs} \implies gts (y \# ys) (x \# xs) \implies gt y x \vee gts \text{ ys xs}) \implies$
 $\text{lenext } gts (y \# ys) (x \# xs) \implies gt y x \vee \text{lenext } gts \text{ ys xs}$
 $\langle proof \rangle$

5.5 Length-Lexicographic Extension

abbreviation `len_lexext` :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$ **where**
 $\text{len_lexext } gt \equiv \text{lenext } (\text{lexext } gt)$

lemma `len_lexext_mono_strong`:

$(\forall y \in \text{set ys}. \forall x \in \text{set xs}. gt y x \longrightarrow gt' y x) \implies \text{len_lexext } gt \text{ ys xs} \implies \text{len_lexext } gt' \text{ ys xs}$
 $\langle proof \rangle$

lemma `len_lexext_map_strong`:

$(\forall y \in \text{set ys}. \forall x \in \text{set xs}. gt y x \longrightarrow gt (f y) (f x)) \implies \text{len_lexext } gt \text{ ys xs} \implies$
 $\text{len_lexext } gt (\text{map f ys}) (\text{map f xs})$
 $\langle proof \rangle$

lemma `len_lexext_irrefl`: $(\forall x \in \text{set xs}. \neg gt x x) \implies \neg \text{len_lexext } gt \text{ xs xs}$
 $\langle proof \rangle$

lemma `len_lexext_trans_strong`:

$(\forall z \in \text{set zs}. \forall y \in \text{set ys}. \forall x \in \text{set xs}. gt z y \longrightarrow gt y x \longrightarrow gt z x) \implies \text{len_lexext } gt \text{ zs ys} \implies$
 $\text{len_lexext } gt \text{ ys xs} \implies \text{len_lexext } gt \text{ zs xs}$
 $\langle proof \rangle$

lemma *len_lexext_snoc*: $\text{len_lexext gt} (\text{xs} @ [x]) \text{ xs}$
(proof)

lemma *len_lexext_compat_cons*: $\text{len_lexext gt} \text{ ys xs} \implies \text{len_lexext gt} (\text{x} \# \text{ys}) (\text{x} \# \text{xs})$
(proof)

lemma *len_lexext_compat_snoc*: $\text{len_lexext gt} \text{ ys xs} \implies \text{len_lexext gt} (\text{ys} @ [x]) (\text{xs} @ [x])$
(proof)

lemma *len_lexext_compat_list*: $\text{gt} \text{ y x} \implies \text{len_lexext gt} (\text{xs} @ \text{y} \# \text{xs}') (\text{xs} @ \text{x} \# \text{xs}')$
(proof)

lemma *len_lexext_singleton[simp]*: $\text{len_lexext gt} [\text{y}] [\text{x}] \longleftrightarrow \text{gt} \text{ y x}$
(proof)

lemma *len_lexext_total*: $(\forall y \in B. \forall x \in A. \text{gt} \text{ y x} \vee \text{gt} \text{ x y} \vee y = x) \implies \text{ys} \in \text{lists } B \implies \text{xs} \in \text{lists } A \implies \text{len_lexext gt} \text{ ys xs} \vee \text{len_lexext gt} \text{ xs ys} \vee \text{ys} = \text{xs}$
(proof)

lemma *len_lexext_iff_lenlex*: $\text{len_lexext gt} \text{ ys xs} \longleftrightarrow (\text{xs}, \text{ys}) \in \text{lenlex} \{(x, y). \text{gt} \text{ y x}\}$
(proof)

lemma *len_lexext_wf*: $\text{wfP} (\lambda x y. \text{gt} \text{ y x}) \implies \text{wfP} (\lambda \text{xs} \text{ ys}. \text{len_lexext gt} \text{ ys xs})$
(proof)

lemma *len_lexext_hd_or_tl*: $\text{len_lexext gt} (\text{y} \# \text{ys}) (\text{x} \# \text{xs}) \implies \text{gt} \text{ y x} \vee \text{len_lexext gt} \text{ ys xs}$
(proof)

interpretation *len_lexext*: *ext len_lexext*
(proof)

interpretation *len_lexext*: *ext_irrefl_trans_strong len_lexext*
(proof)

interpretation *len_lexext*: *ext_snoc len_lexext*
(proof)

interpretation *len_lexext*: *ext_compat_cons len_lexext*
(proof)

interpretation *len_lexext*: *ext_compat_snoc len_lexext*
(proof)

interpretation *len_lexext*: *ext_compat_list len_lexext*
(proof)

interpretation *len_lexext*: *ext_singleton len_lexext*
(proof)

interpretation *len_lexext*: *ext_total len_lexext*
(proof)

interpretation *len_lexext*: *ext_wf len_lexext*
(proof)

interpretation *len_lexext*: *ext_hd_or_tl len_lexext*
(proof)

interpretation *len_lexext*: *ext_wf_bounded len_lexext*
(proof)

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

abbreviation `len_lexext_rev` :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$ **where**
 $\text{len_lexext_rev } gt \equiv \text{lenext} (\text{lexext_rev } gt)$

lemma `len_lexext_rev_mono_strong`:
 $(\forall y \in \text{set } ys. \forall x \in \text{set } xs. gt y x \rightarrow gt' y x) \Rightarrow \text{len_lexext_rev } gt ys xs \Rightarrow \text{len_lexext_rev } gt' ys xs$
 $\langle \text{proof} \rangle$

lemma `len_lexext_rev_map_strong`:
 $(\forall y \in \text{set } ys. \forall x \in \text{set } xs. gt y x \rightarrow gt (f y) (f x)) \Rightarrow \text{len_lexext_rev } gt ys xs \Rightarrow$
 $\text{len_lexext_rev } gt (\text{map } f ys) (\text{map } f xs)$
 $\langle \text{proof} \rangle$

lemma `len_lexext_rev_irrefl`: $(\forall x \in \text{set } xs. \neg gt x x) \Rightarrow \neg \text{len_lexext_rev } gt xs xs$
 $\langle \text{proof} \rangle$

lemma `len_lexext_rev_trans_strong`:
 $(\forall z \in \text{set } zs. \forall y \in \text{set } ys. \forall x \in \text{set } xs. gt z y \rightarrow gt y x \rightarrow gt z x) \Rightarrow \text{len_lexext_rev } gt zs ys \Rightarrow$
 $\text{len_lexext_rev } gt ys xs \Rightarrow \text{len_lexext_rev } gt zs xs$
 $\langle \text{proof} \rangle$

lemma `len_lexext_rev_snoc`: $\text{len_lexext_rev } gt (xs @ [x]) xs$
 $\langle \text{proof} \rangle$

lemma `len_lexext_rev_compat_cons`: $\text{len_lexext_rev } gt ys xs \Rightarrow \text{len_lexext_rev } gt (x \# ys) (x \# xs)$
 $\langle \text{proof} \rangle$

lemma `len_lexext_rev_compat_snoc`: $\text{len_lexext_rev } gt ys xs \Rightarrow \text{len_lexext_rev } gt (ys @ [x]) (xs @ [x])$
 $\langle \text{proof} \rangle$

lemma `len_lexext_rev_compat_list`: $gt y x \Rightarrow \text{len_lexext_rev } gt (xs @ y \# xs') (xs @ x \# xs')$
 $\langle \text{proof} \rangle$

lemma `len_lexext_rev_singleton[simp]`: $\text{len_lexext_rev } gt [y] [x] \longleftrightarrow gt y x$
 $\langle \text{proof} \rangle$

lemma `len_lexext_rev_total`: $(\forall y \in B. \forall x \in A. gt y x \vee gt x y \vee y = x) \Rightarrow ys \in \text{lists } B \Rightarrow$
 $xs \in \text{lists } A \Rightarrow \text{len_lexext_rev } gt ys xs \vee \text{len_lexext_rev } gt xs ys \vee ys = xs$
 $\langle \text{proof} \rangle$

lemma `len_lexext_rev_iff_len_lexext`: $\text{len_lexext_rev } gt ys xs \longleftrightarrow \text{len_lexext } gt (\text{rev } ys) (\text{rev } xs)$
 $\langle \text{proof} \rangle$

lemma `len_lexext_rev_wf`: $wfP (\lambda x y. gt y x) \Rightarrow wfP (\lambda xs ys. \text{len_lexext_rev } gt ys xs)$
 $\langle \text{proof} \rangle$

lemma `len_lexext_rev_hd_or_tl`:
 $\text{len_lexext_rev } gt (y \# ys) (x \# xs) \Rightarrow gt y x \vee \text{len_lexext_rev } gt ys xs$
 $\langle \text{proof} \rangle$

interpretation `len_lexext_rev`: `ext len_lexext_rev`
 $\langle \text{proof} \rangle$

interpretation `len_lexext_rev`: `ext_irrefl_trans_strong len_lexext_rev`
 $\langle \text{proof} \rangle$

interpretation `len_lexext_rev`: `ext_snoc len_lexext_rev`
 $\langle \text{proof} \rangle$

interpretation `len_lexext_rev`: `ext_compatible_cons len_lexext_rev`
 $\langle \text{proof} \rangle$

interpretation `len_lexext_rev`: `ext_compatible_snoc len_lexext_rev`

$\langle proof \rangle$

interpretation $len_lexext_rev: ext_compat_list len_lexext_rev$
 $\langle proof \rangle$

interpretation $len_lexext_rev: ext_singleton len_lexext_rev$
 $\langle proof \rangle$

interpretation $len_lexext_rev: ext_total len_lexext_rev$
 $\langle proof \rangle$

interpretation $len_lexext_rev: ext_wf len_lexext_rev$
 $\langle proof \rangle$

interpretation $len_lexext_rev: ext_hd_or_tl len_lexext_rev$
 $\langle proof \rangle$

interpretation $len_lexext_rev: ext_wf_bounded len_lexext_rev$
 $\langle proof \rangle$

5.7 Dershowitz–Manna Multiset Extension

definition $msetext_dersh$ where

$msetext_dersh gt ys xs = (let N = mset ys; M = mset xs in$
 $(\exists Y X. Y \neq \{\#\} \wedge Y \subseteq \# N \wedge M = (N - Y) + X \wedge (\forall x. x \in \# X \longrightarrow (\exists y. y \in \# Y \wedge gt y x))))$

The following proof is based on that of $less_multiset_{DM_imp_mult}$.

lemma $msetext_dersh_imp_mult_rel:$

assumes

$ys_a: ys \in lists A$ **and** $xs_a: xs \in lists A$ **and**

$ys_gt_xs: msetext_dersh gt ys xs$

shows $(mset ys, mset xs) \in mult \{(x, y). x \in A \wedge y \in A \wedge gt y x\}$

$\langle proof \rangle$

lemma $msetext_dersh_imp_mult: msetext_dersh gt ys xs \implies (mset ys, mset xs) \in mult \{(x, y). gt y x\}$
 $\langle proof \rangle$

lemma $mult_imp_msetext_dersh_rel:$

assumes

$ys_a: set_mset (mset ys) \subseteq A$ **and** $xs_a: set_mset (mset xs) \subseteq A$ **and**

$in_mult: (mset ys, mset xs) \in mult \{(x, y). x \in A \wedge y \in A \wedge gt y x\}$ **and**

$trans: \forall z \in A. \forall y \in A. \forall x \in A. gt z y \longrightarrow gt y x \longrightarrow gt z x$

shows $msetext_dersh gt ys xs$

$\langle proof \rangle$

lemma $msetext_dersh_mono_strong:$

$(\forall y \in set ys. \forall x \in set xs. gt y x \longrightarrow gt' y x) \implies msetext_dersh gt ys xs \implies$

$msetext_dersh gt' ys xs$

$\langle proof \rangle$

lemma $msetext_dersh_map_strong:$

assumes

$compat_f: \forall y \in set ys. \forall x \in set xs. gt y x \longrightarrow gt (f y) (f x)$ **and**

$ys_gt_xs: msetext_dersh gt ys xs$

shows $msetext_dersh gt (map f ys) (map f xs)$

$\langle proof \rangle$

lemma $msetext_dersh_trans:$

assumes

$zs_a: zs \in lists A$ **and**

$ys_a: ys \in lists A$ **and**

$xs_a: xs \in lists A$ **and**

$trans: \forall z \in A. \forall y \in A. \forall x \in A. gt z y \longrightarrow gt y x \longrightarrow gt z x$ **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⟩

lemma  $msetext\_dersh\_irrefl\_from\_trans:$ 
assumes
 $trans: \forall z \in set\ xs. \forall y \in set\ xs. \forall x \in set\ xs. gt\ z\ y \longrightarrow gt\ y\ x \longrightarrow gt\ z\ x$  and
 $irrefl: \forall x \in set\ xs. \neg gt\ x\ x$ 
shows  $\neg msetext\_dersh\ gt\ xs\ xs$ 
⟨proof⟩

lemma  $msetext\_dersh\_snoc: msetext\_dersh\ gt\ (xs @ [x])\ xs$ 
⟨proof⟩

lemma  $msetext\_dersh\_compat\_cons:$ 
assumes  $ys\_gt\_xs: msetext\_dersh\ gt\ ys\ xs$ 
shows  $msetext\_dersh\ gt\ (x \# ys) (x \# xs)$ 
⟨proof⟩

lemma  $msetext\_dersh\_compat\_snoc: msetext\_dersh\ gt\ ys\ xs \implies msetext\_dersh\ gt\ (ys @ [x]) (xs @ [x])$ 
⟨proof⟩

lemma  $msetext\_dersh\_compat\_list:$ 
assumes  $y\_gt\_x: gt\ y\ x$ 
shows  $msetext\_dersh\ gt\ (xs @ y \# xs') (xs @ x \# xs')$ 
⟨proof⟩

lemma  $msetext\_dersh\_singleton: msetext\_dersh\ gt\ [y]\ [x] \longleftrightarrow gt\ y\ x$ 
⟨proof⟩

lemma  $msetext\_dersh\_wf:$ 
assumes  $wf\_gt: wfP(\lambda x\ y. gt\ y\ x)$ 
shows  $wfP(\lambda xs\ ys. msetext\_dersh\ gt\ ys\ xs)$ 
⟨proof⟩

interpretation  $msetext\_dersh: ext\ msetext\_dersh$ 
⟨proof⟩

interpretation  $msetext\_dersh: ext\_trans\_before\_irrefl\ msetext\_dersh$ 
⟨proof⟩

interpretation  $msetext\_dersh: ext\_snoc\ msetext\_dersh$ 
⟨proof⟩

interpretation  $msetext\_dersh: ext\_compat\_cons\ msetext\_dersh$ 
⟨proof⟩

interpretation  $msetext\_dersh: ext\_compat\_snoc\ msetext\_dersh$ 
⟨proof⟩

interpretation  $msetext\_dersh: ext\_compat\_list\ msetext\_dersh$ 
⟨proof⟩

interpretation  $msetext\_dersh: ext\_singleton\ msetext\_dersh$ 
⟨proof⟩

interpretation  $msetext\_dersh: ext\_wf\ msetext\_dersh$ 
⟨proof⟩

```

5.8 Huet–Oppen Multiset Extension

```

definition  $msetext\_huet$  where
 $msetext\_huet\ gt\ ys\ xs = (\text{let } N = mset\ ys; M = mset\ xs \text{ in}$ 
 $M \neq N \wedge (\forall x. count\ M\ x > count\ N\ x \longrightarrow (\exists y. gt\ y\ x \wedge count\ N\ y > count\ M\ y)))$ 

```

```

lemma msetext_huet_imp_count_gt:
  assumes ys_gt_xs: msetext_huet gt ys xs
  shows  $\exists x. \text{count}(\text{mset } ys) x > \text{count}(\text{mset } xs) x$ 
  (proof)

```

```

lemma msetext_huet_imp_dersh:
  assumes huet: msetext_huet gt ys xs
  shows msetext_dersh gt ys xs
  (proof)

```

The following proof is based on that of *mult_imp_less_multiset_{HO}*.

```

lemma mult_imp_msetext_huet:
  assumes
    irrefl: 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
  (proof)

```

```

theorem msetext_huet_eq_dersh: irreflp gt  $\implies$  transp gt  $\implies$  msetext_dersh gt = msetext_huet gt
  (proof)

```

```

lemma msetext_huet_mono_strong:
   $(\forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } y x \longrightarrow \text{gt}' y x) \implies \text{msetext\_huet } gt \text{ ys xs} \implies \text{msetext\_huet } gt' \text{ ys xs}$ 
  (proof)

```

```

lemma msetext_huet_map:
  assumes
    fin: finite A and
    ys_a: ys ∈ lists A and xs_a: xs ∈ lists A and
    irrefl_f:  $\forall x \in A. \neg \text{gt } (f x) (f x)$  and
    trans_f:  $\forall z \in A. \forall y \in A. \forall x \in A. \text{gt } (f z) (f y) \longrightarrow \text{gt } (f y) (f x) \longrightarrow \text{gt } (f z) (f x)$  and
    compat_f:  $\forall y \in A. \forall x \in A. \text{gt } y x \longrightarrow \text{gt } (f y) (f x)$  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)

```

```

lemma msetext_huet_irrefl:  $(\forall x \in \text{set } xs. \neg \text{gt } x x) \implies \neg \text{msetext\_huet } gt \text{ xs xs}$ 
  (proof)

```

```

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:  $\forall x \in A. \neg \text{gt } x x$  and
    trans:  $\forall z \in A. \forall y \in A. \forall x \in A. \text{gt } z y \longrightarrow \text{gt } y x \longrightarrow \text{gt } z x$  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)

```

```

lemma msetext_huet_snoc: msetext_huet gt (xs @ [x]) xs
  (proof)

```

```

lemma msetext_huet_compat_cons: msetext_huet gt ys xs  $\implies$  msetext_huet gt (x # ys) (x # xs)
  (proof)

```

```

lemma msetext_huet_compat_snoc: msetext_huet gt ys xs  $\implies$  msetext_huet gt (ys @ [x]) (xs @ [x])
  (proof)

```

```

lemma msetext_huet_compat_list:  $y \neq x \implies \text{gt } y x \implies \text{msetext\_huet } gt (xs @ y \# xs') (xs @ x \# xs')$ 
  (proof)

```

```

lemma msetext_huet_singleton:  $y \neq x \implies \text{msetext\_huet } gt [y] [x] \longleftrightarrow gt y x$ 
   $\langle \text{proof} \rangle$ 

lemma msetext_huet_wf:  $\text{wfP } (\lambda x y. gt y x) \implies \text{wfP } (\lambda xs ys. \text{msetext\_huet } gt ys xs)$ 
   $\langle \text{proof} \rangle$ 

lemma msetext_huet_hd_or_tl:
  assumes
    trans:  $\forall z y x. gt z y \longrightarrow gt y x \longrightarrow gt z x$  and
    total:  $\forall y x. gt y x \vee gt x y \vee y = x$  and
    len_eq:  $\text{length } ys = \text{length } xs$  and
    yys_gt_xxs:  $\text{msetext\_huet } gt (y \# ys) (x \# xs)$ 
  shows  $gt y x \vee \text{msetext\_huet } gt ys xs$ 
   $\langle \text{proof} \rangle$ 

interpretation msetext_huet: ext msetext_huet
   $\langle \text{proof} \rangle$ 

interpretation msetext_huet: ext_irrefl_before_trans msetext_huet
   $\langle \text{proof} \rangle$ 

interpretation msetext_huet: ext_snoc msetext_huet
   $\langle \text{proof} \rangle$ 

interpretation msetext_huet: ext_compat_cons msetext_huet
   $\langle \text{proof} \rangle$ 

interpretation msetext_huet: ext_compat_snoc msetext_huet
   $\langle \text{proof} \rangle$ 

interpretation msetext_huet: ext_compat_list msetext_huet
   $\langle \text{proof} \rangle$ 

interpretation msetext_huet: ext_singleton msetext_huet
   $\langle \text{proof} \rangle$ 

interpretation msetext_huet: ext_wf msetext_huet
   $\langle \text{proof} \rangle$ 

interpretation msetext_huet: ext_hd_or_tl msetext_huet
   $\langle \text{proof} \rangle$ 

interpretation msetext_huet: ext_wf_bounded msetext_huet
   $\langle \text{proof} \rangle$ 

```

5.9 Componentwise Extension

```

definition cwiseext ::  $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$  where
  cwiseext gt ys xs  $\longleftrightarrow \text{length } ys = \text{length } xs$ 
   $\wedge (\forall i < \text{length } ys. gt (ys ! i) (xs ! i) \vee ys ! i = xs ! i)$ 
   $\wedge (\exists i < \text{length } ys. gt (ys ! i) (xs ! i))$ 

lemma cwiseext_imp_len_lexext:
  assumes cw: cwiseext gt ys xs
  shows len_lexext gt ys xs
   $\langle \text{proof} \rangle$ 

lemma cwiseext_mono_strong:
   $(\forall y \in \text{set } ys. \forall x \in \text{set } xs. gt y x \longrightarrow gt' y x) \implies \text{cwiseext } gt ys xs \implies \text{cwiseext } gt' ys xs$ 
   $\langle \text{proof} \rangle$ 

lemma cwiseext_map_strong:
   $(\forall y \in \text{set } ys. \forall x \in \text{set } xs. gt y x \longrightarrow gt (f y) (f x)) \implies \text{cwiseext } gt ys xs \implies$ 
   $\text{cwiseext } gt (\text{map } f ys) (\text{map } f xs)$ 

```

```
 $\langle proof \rangle$ 
```

```
lemma cwiseext_irrefl: ( $\forall x \in \text{set } xs. \neg gt x x \implies \neg \text{cwiseext } gt xs xs$ )  
 $\langle proof \rangle$ 
```

```
lemma cwiseext_trans_strong:  
assumes  
 $\forall z \in \text{set } zs. \forall y \in \text{set } ys. \forall x \in \text{set } xs. gt z y \longrightarrow gt y x \longrightarrow gt z x$  and  
 $\text{cwiseext } gt zs ys$  and  $\text{cwiseext } gt ys xs$   
shows  $\text{cwiseext } gt zs xs$   
 $\langle proof \rangle$ 
```

```
lemma cwiseext_compat_cons:  $\text{cwiseext } gt ys xs \implies \text{cwiseext } gt (x \# ys) (x \# xs)$   
 $\langle proof \rangle$ 
```

```
lemma cwiseext_compat_snoc:  $\text{cwiseext } gt ys xs \implies \text{cwiseext } gt (ys @ [x]) (xs @ [x])$   
 $\langle proof \rangle$ 
```

```
lemma cwiseext_compat_list:  
assumes  $y_gt_x: gt y x$   
shows  $\text{cwiseext } gt (xs @ y \# xs') (xs @ x \# xs')$   
 $\langle proof \rangle$ 
```

```
lemma cwiseext_singleton:  $\text{cwiseext } gt [y] [x] \longleftrightarrow gt y x$   
 $\langle proof \rangle$ 
```

```
lemma cwiseext_wf:  $wfP (\lambda x y. gt y x) \implies wfP (\lambda xs ys. \text{cwiseext } gt ys xs)$   
 $\langle proof \rangle$ 
```

```
lemma cwiseext_hd_or_tl:  $\text{cwiseext } gt (y \# ys) (x \# xs) \implies gt y x \vee \text{cwiseext } gt ys xs$   
 $\langle proof \rangle$ 
```

```
locale ext_cwiseext = ext_compat_list + ext_compat_cons  
begin
```

```
context  
fixes  $gt :: 'a \Rightarrow 'a \Rightarrow \text{bool}$   
assumes  
 $gt_{\text{irrefl}}: \neg gt x x$  and  
 $\text{trans\_}gt: \text{ext } gt zs ys \implies \text{ext } gt ys xs \implies \text{ext } gt zs xs$   
begin
```

```
lemma  
assumes  $ys\_gt_{\text{cw}}\_xs: \text{cwiseext } gt ys xs$   
shows  $\text{ext } gt ys xs$   
 $\langle proof \rangle$ 
```

```
end
```

```
end
```

```
interpretation cwiseext: ext cwiseext  
 $\langle proof \rangle$ 
```

```
interpretation cwiseext: ext_irrefl_trans_strong cwiseext  
 $\langle proof \rangle$ 
```

```
interpretation cwiseext: ext_compat_cons cwiseext  
 $\langle proof \rangle$ 
```

```
interpretation cwiseext: ext_compat_snoc cwiseext  
 $\langle proof \rangle$ 
```

```

interpretation cwiseext: ext_compat_list cwiseext
  ⟨proof⟩

interpretation cwiseext: ext_singleton cwiseext
  ⟨proof⟩

interpretation cwiseext: ext_wf cwiseext
  ⟨proof⟩

interpretation cwiseext: ext_hd_or_tl cwiseext
  ⟨proof⟩

interpretation cwiseext: ext_wf_bounded cwiseext
  ⟨proof⟩

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 ≥_t = ≥_t
begin

This theory defines the applicative recursive path order (RPO), a variant of RPO for  $\lambda$ -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 ⇒ 's ⇒ bool (infix >_s 50) +
  fixes ext :: (('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool) ⇒ ('s, 'v) tm list ⇒ ('s, 'v) tm list ⇒ bool
  assumes
    ext_ext_trans_before_irrefl: ext_trans_before_irrefl ext and
    ext_ext_compat_list: ext_compat_list ext
begin
```

```

lemma ext_mono[mono]: gt ≤ gt' ⇒ ext gt ≤ ext gt'
  ⟨proof⟩

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_sym_sym: g >_s f ⇒ Hd (Sym g) >_t Hd (Sym f)
  | gt_sym_app: Hd (Sym g) >_t s1 ⇒ Hd (Sym g) >_t s2 ⇒ Hd (Sym g) >_t App s1 s2
  | gt_app_app: ext (>_t) [t1, t2] [s1, s2] ⇒ App t1 t2 >_t s1 ⇒ App t1 t2 >_t s2 ⇒
    App t1 t2 >_t App s1 s2

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

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 ≥_t = ≥_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 :: ' $s \Rightarrow s \Rightarrow \text{bool}$  (infix  $>_s$  50) and
  arity_sym :: ' $s \Rightarrow \text{enat}$  and
  arity_var :: ' $v \Rightarrow \text{enat}$  +
fixes
  extf :: ' $s \Rightarrow ((s, v) \text{ tm} \Rightarrow (s, v) \text{ tm} \Rightarrow \text{bool}) \Rightarrow (s, v) \text{ tm list} \Rightarrow (s, v) \text{ tm list} \Rightarrow \text{bool}$ 
assumes
  extf_ext_trans_before_irrefl: ext_trans_before_irrefl (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)
   $\langle \text{proof} \rangle$ 

lemma extf_ext: ext (extf f)
   $\langle \text{proof} \rangle$ 

lemmas extf_mono_strong = ext_mono_strong[ $\text{OF extf\_ext}$ ]
lemmas extf_mono = ext_mono[ $\text{OF extf\_ext, mono}$ ]
lemmas extf_map = ext_map[ $\text{OF extf\_ext}$ ]
lemmas extf_trans = ext_trans.trans[ $\text{OF extf\_ext\_trans}$ ]
lemmas extf_irrefl_from_trans =
  ext_trans_before_irrefl_irrefl_from_trans[ $\text{OF extf\_ext\_trans\_before\_irrefl}$ ]
lemmas extf_compat_append_left = ext_compat_cons.compat_append_left[ $\text{OF extf\_ext\_compat\_cons}$ ]
lemmas extf_compat_list = ext_compat_list.compat_list[ $\text{OF extf\_ext\_compat\_list}$ ]

definition chkvar :: ' $(s, v) \text{ tm} \Rightarrow (s, v) \text{ tm} \Rightarrow \text{bool}$  where
  [ $\text{simp}$ ]: chkvar t s  $\longleftrightarrow$  vars_hd (head s)  $\subseteq$  vars t

end

```

```

locale rpo = rpo_basis __ arity_sym arity_var
for
  arity_sym :: ' $s \Rightarrow \text{enat}$  and
  arity_var :: ' $v \Rightarrow \text{enat}$ 
begin

```

7.2 Inductive Definitions

```

definition
  chksubs :: ' $((s, v) \text{ tm} \Rightarrow (s, v) \text{ tm} \Rightarrow \text{bool}) \Rightarrow (s, v) \text{ tm} \Rightarrow (s, v) \text{ tm} \Rightarrow \text{bool}$ 
where
  [ $\text{simp}$ ]: chksubs gt t s  $\longleftrightarrow$  (case s of App s1 s2  $\Rightarrow$  gt t s1  $\wedge$  gt t s2 | _  $\Rightarrow$  True)

```

```

lemma chksubs_mono[mono]: gt  $\leq$  gt'  $\Longrightarrow$  chksubs gt  $\leq$  chksubs gt'
   $\langle \text{proof} \rangle$ 

```

```

inductive gt :: ' $(s, v) \text{ tm} \Rightarrow (s, v) \text{ tm} \Rightarrow \text{bool}$  (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_diff: head t  $>_{hd}$  head s  $\Longrightarrow$  chkvar t s  $\Longrightarrow$  chksubs ( $>_t$ ) t s  $\Longrightarrow$  t  $>_t$  s
  | gt_same: head t = head s  $\Longrightarrow$  chksubs ( $>_t$ ) t s  $\Longrightarrow$ 
    ( $\forall f \in \text{ground\_heads} (\text{head } t)$ . extf f ( $>_t$ ) (args t) (args s))  $\Longrightarrow$  t  $>_t$  s

```

```

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

```

```

inductive gt_sub :: ('s, 'v) tm  $\Rightarrow$  ('s, 'v) tm  $\Rightarrow$  bool where
  gt_subI: is_App t  $\Longrightarrow$  fun t  $\geq_t$  s  $\vee$  arg t  $\geq_t$  s  $\Longrightarrow$  gt_sub t s

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

inductive gt_same :: ('s, 'v) tm  $\Rightarrow$  ('s, 'v) tm  $\Rightarrow$  bool where
  gt_sameI: head t = head s  $\Longrightarrow$  chksubs ( $>_t$ ) t s  $\Longrightarrow$ 
    ( $\forall f \in ground\_heads (head t)$ . extf f ( $>_t$ ) (args t) (args s))  $\Longrightarrow$  gt_same t s

lemma gt_iff_sub_diff_same: t  $>_t$  s  $\longleftrightarrow$  gt_sub t s  $\vee$  gt_diff t s  $\vee$  gt_same t s
   $\langle proof \rangle$ 

```

7.3 Transitivity

```

lemma gt_fun_imp: fun t  $>_t$  s  $\Longrightarrow$  t  $>_t$  s
   $\langle proof \rangle$ 

lemma gt_arg_imp: arg t  $>_t$  s  $\Longrightarrow$  t  $>_t$  s
   $\langle proof \rangle$ 

lemma gt_imp_vars: t  $>_t$  s  $\Longrightarrow$  vars t  $\supseteq$  vars s
   $\langle proof \rangle$ 

theorem gt_trans: u  $>_t$  t  $\Longrightarrow$  t  $>_t$  s  $\Longrightarrow$  u  $>_t$  s
   $\langle proof \rangle$ 

```

7.4 Irreflexivity

```

theorem gt_irrefl:  $\neg$  s  $>_t$  s
   $\langle proof \rangle$ 

lemma gt_antisym: t  $>_t$  s  $\Longrightarrow$   $\neg$  s  $>_t$  t
   $\langle proof \rangle$ 

```

7.5 Subterm Property

```

lemma
  gt_sub_fun: App s t  $>_t$  s and
  gt_sub_arg: App s t  $>_t$  t
   $\langle proof \rangle$ 

theorem gt_proper_sub: proper_sub s t  $\Longrightarrow$  t  $>_t$  s
   $\langle proof \rangle$ 

```

7.6 Compatibility with Functions

```

lemma gt_compat_fun:
  assumes t'_gt_t: t'  $>_t$  t
  shows App s t'  $>_t$  App s t
   $\langle proof \rangle$ 

theorem gt_compat_fun_strong:
  assumes t'_gt_t: t'  $>_t$  t
  shows apps s (t' # us)  $>_t$  apps s (t # us)
   $\langle proof \rangle$ 

```

7.7 Compatibility with Arguments

```

theorem gt_diff_same_compat_arg:
  assumes
    extf_compat_snoc:  $\bigwedge f$ . ext_compat_snoc (extf f) and
    diff_same: gt_diff s' s  $\vee$  gt_same s' s
  shows App s' t  $>_t$  App s t

```

$\langle proof \rangle$

7.8 Stability under Substitution

```
lemma gt_imp_chksubs_gt:
  assumes t_gt_s:  $t >_t s$ 
  shows chksubs ( $>_t$ ) t s
⟨proof⟩

theorem gt_subst:
  assumes wary_ρ: wary_subst ρ
  shows  $t >_t s \implies \text{subst } \rho t >_t \text{subst } \rho s$ 
⟨proof⟩
```

7.9 Totality on Ground Terms

```
theorem gt_total_ground:
  assumes extf_total:  $\bigwedge f. \text{ext\_total } (\text{extf } f)$ 
  shows ground t  $\implies$  ground s  $\implies$   $t >_t s \vee s >_t t \vee t = s$ 
⟨proof⟩
```

7.10 Well-foundedness

```
abbreviation gtg :: ('s, 'v) tm  $\Rightarrow$  ('s, 'v) tm  $\Rightarrow$  bool (infix  $>_{tg}$  50) where
```

```
 $(>_{tg}) \equiv \lambda t s. \text{ground } t \wedge t >_t s$ 
```

```
theorem gt_wf:
  assumes extf_wf:  $\bigwedge f. \text{ext\_wf } (\text{extf } f)$ 
  shows wfP ( $\lambda s t. t >_t s$ )
⟨proof⟩
```

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  $\Rightarrow$  enat and
  arity_var :: 'v  $\Rightarrow$  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  $\Rightarrow$  ('s, 'v) tm  $\Rightarrow$  bool)  $\Rightarrow$  ('s, 'v) tm  $\Rightarrow$  ('s, 'v) tm  $\Rightarrow$  bool
where
```

```
[simp]: chkargs gt t s  $\longleftrightarrow$   $(\forall s' \in \text{set } (\text{args } s). \text{gt } t s')$ 
```

```

lemma chkargs_mono[mono]:  $gt \leq gt' \implies \text{chkargs } gt \leq \text{chkargs } gt'$ 
   $\langle \text{proof} \rangle$ 

inductive gt :: ('s, 'v) tm  $\Rightarrow$  ('s, 'v) tm  $\Rightarrow$  bool (infix  $>_t$  50) where
  | gt_arg:  $ti \in \text{set } (\text{args } t) \implies ti >_t s \vee ti = s \implies t >_t s$ 
  | gt_diff:  $\text{head } t >_{hd} \text{head } s \implies \text{chkvar } t s \implies \text{chkargs } (>_t) t s \implies t >_t s$ 
  | gt_same:  $\text{head } t = \text{head } s \implies \text{chkargs } (>_t) t s \implies$ 
     $(\forall f \in \text{ground\_heads } (\text{head } t). \text{extff } (>_t) (\text{args } t) (\text{args } s)) \implies t >_t s$ 

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

```

8.3 Transitivity

```

lemma gt_in_args_imp:  $ti \in \text{set } (\text{args } t) \implies ti >_t s \implies t >_t s$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma gt_imp_vars:  $t >_t s \implies \text{vars } t \supseteq \text{vars } s$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma gt_trans:  $u >_t t \implies t >_t s \implies u >_t s$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma gt_sub_fun:  $\text{App } s t >_t s$ 
   $\langle \text{proof} \rangle$ 

```

```
end
```

8.4 Conditional Equivalence with Unoptimized Version

```

context rpo
begin

```

```

context
  assumes extf_ext_snoc:  $\bigwedge f. \text{ext\_snoc } (\text{extff } f)$ 
begin

```

```

lemma rpo_optim:  $rpo\_optim \text{ ground\_heads\_var } (>_s) \text{ extf arity\_sym arity\_var}$ 
   $\langle \text{proof} \rangle$ 

```

```

abbreviation
  chkargs :: (('s, 'v) tm  $\Rightarrow$  ('s, 'v) tm  $\Rightarrow$  bool)  $\Rightarrow$  ('s, 'v) tm  $\Rightarrow$  bool
where
  chkargs  $\equiv rpo\_optim.\text{chkargs}$ 

```

```

abbreviation gt_optim :: ('s, 'v) tm  $\Rightarrow$  ('s, 'v) tm  $\Rightarrow$  bool (infix  $>_{to}$  50) where
   $(>_{to}) \equiv rpo\_optim.gt \text{ ground\_heads\_var } (>_s) \text{ extf}$ 

```

```

abbreviation ge_optim :: ('s, 'v) tm  $\Rightarrow$  ('s, 'v) tm  $\Rightarrow$  bool (infix  $\geq_{to}$  50) where
   $(\geq_{to}) \equiv rpo\_optim.ge \text{ ground\_heads\_var } (>_s) \text{ extf}$ 

```

```

theorem gt_iff_optim:  $t >_t s \longleftrightarrow t >_{to} s$ 
   $\langle \text{proof} \rangle$ 

```

```
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 ↔ (exists 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 ⇒ ('s, 'v) tm where
  raw_db_subst i x = (λy. 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#}
  ⟨proof⟩

lemma head_subst_db: head (subst_db i x s) = head (subst (raw_db_subst i x) s)
  ⟨proof⟩

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)
  ⟨proof⟩

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)
  ⟨proof⟩

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
  ⟨proof⟩

sublocale std: rpo ground_head_var gt_sym λf. len_lexext arity_sym arity_var
  ⟨proof⟩

sublocale optim: rpo_optim ground_head_var gt_sym λf. len_lexext arity_sym arity_var
  ⟨proof⟩

end

end
```