

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

Jasmin Christian Blanchette, Uwe Waldmann, and Daniel Wand

October 11, 2017

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	2
2	Utilities for Lambda-Free Orders	2
2.1	Finite Sets	2
2.2	Function Power	2
2.3	Least Operator	3
2.4	Antisymmetric Relations	3
2.5	Acyclic Relations	3
2.6	Reflexive, Transitive Closure	3
2.7	Well-Founded Relations	3
2.8	Wellorders	4
2.9	Lists	4
2.10	Extended Natural Numbers	5
2.11	Multisets	5
3	Lambda-Free Higher-Order Terms	7
3.1	Precedence on Symbols	7
3.2	Heads	7
3.3	Terms	8
3.4	Substitutions	10
3.5	Subterms	10
3.6	Maximum Arities	11
3.7	Potential Heads of Ground Instances of Variables	12
4	Infinite (Non-Well-Founded) Chains	14
5	Extension Orders	15
5.1	Locales	15
5.2	Lexicographic Extension	17
5.3	Reverse (Right-to-Left) Lexicographic Extension	19
5.4	Generic Length Extension	20
5.5	Length-Lexicographic Extension	21
5.6	Reverse (Right-to-Left) Length-Lexicographic Extension	22
5.7	Dershowitz–Manna Multiset Extension	23
5.8	Huet–Oppen Multiset Extension	25
5.9	Componentwise Extension	27

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	29
7.1	Setup	29
7.2	Inductive Definitions	30
7.3	Transitivity	30
7.4	Irreflexivity	30
7.5	Subterm Property	31
7.6	Compatibility with Functions	31
7.7	Compatibility with Arguments	31
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	32
8.1	Setup	32
8.2	Definition of the Order	32
8.3	Transitivity	32
8.4	Conditional Equivalence with Unoptimized Version	32
9	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 our FoSSaCS 2017 paper for details.¹

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.

```
hide-const (open) Complex.arg
```

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)  $\Rightarrow$  'a
  assumes mono:  $\bigwedge k. k \leq f k$  and m_le_n:  $m \leq n$ 
  shows  $(f \text{ ^^ } m) k \leq (f \text{ ^^ } n) k$ 
  ⟨proof⟩
```

```
lemma funpow_less_iter:
  fixes f :: ('a::order)  $\Rightarrow$  'a
  assumes mono:  $\bigwedge k. k < f k$  and m_lt_n:  $m < n$ 
  shows  $(f \text{ ^^ } m) k < (f \text{ ^^ } n) k$ 
  ⟨proof⟩
```

¹https://www21.in.tum.de/~blanchet/lambda_free_rpo_conf.pdf

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 $f :: 'b \Rightarrow ('a::wellorder)$
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>

lemma *Least_eq_0_enat*: $P 0 \implies (LEAST x :: enat. P x) = 0$
<proof>

2.4 Antisymmetric Relations

lemma *irrefl_trans_imp_antisym*: $irrefl\ r \implies trans\ r \implies antisym\ r$
<proof>

lemma *irreflp_transp_imp_antisymP*: $irreflp\ p \implies transp\ p \implies antisymp\ p$
<proof>

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 $\neg acyclic\ r$
<proof>

2.6 Reflexive, Transitive Closure

lemma *relcomp_subset_left_imp_relcomp_trancl_subset_left*:
assumes $sub: R\ O\ S \subseteq R$
shows $R\ O\ S^* \subseteq R$
<proof>

lemma *f_chain_in_rtrancl*:
assumes $m_le_n: m \leq n$ and $f_chain: \forall i \in \{m..<n\}. (f\ i, f\ (Suc\ i)) \in R$
shows $(f\ m, f\ n) \in R^*$
<proof>

lemma *f_rev_chain_in_rtrancl*:
assumes $m_le_n: m \leq n$ and $f_chain: \forall i \in \{m..<n\}. (f\ (Suc\ i), f\ i) \in R$
shows $(f\ n, f\ m) \in R^*$
<proof>

2.7 Well-Founded Relations

lemma *wf_app*: $wf\ r \implies wf\ \{(x, y). (f\ x, f\ y) \in r\}$
<proof>

lemma *wfP_app*: $wfP\ p \implies wfP\ (\lambda x\ y. p\ (f\ x)\ (f\ y))$
<proof>

lemma *wf_exists_minimal*:
assumes $wf: wf\ r$ and $Q: Q\ x$
shows $\exists x. Q\ x \wedge (\forall y. (f\ y, f\ x) \in r \longrightarrow \neg Q\ y)$
<proof>

lemma *wfP_exists_minimal*:
assumes *wf*: *wfP* *p* **and** *Q*: *Q* *x*
shows $\exists x. Q\ x \wedge (\forall y. p\ (f\ y)\ (f\ x) \longrightarrow \neg Q\ y)$
<proof>

lemma *finite_irrefl_trans_imp_wf*: *finite* *r* \implies *irrefl* *r* \implies *trans* *r* \implies *wf* *r*
<proof>

lemma *finite_irreflp_transp_imp_wfp*:
finite $\{(x, y). p\ x\ y\} \implies$ *irreflp* *p* \implies *transp* *p* \implies *wfP* *p*
<proof>

lemma *wf_infinite_down_chain_compatible*:
assumes
wf *R*: *wf* *R* **and**
inf_chain_RS: $\forall i. (f\ (Suc\ i), f\ i) \in R \cup S$ **and**
O_subset: $R\ O\ S \subseteq R$
shows $\exists k. \forall i. (f\ (Suc\ (i + k)), f\ (i + k)) \in S$
<proof>

2.8 Wellorders

lemma (*in_wellorder*) *exists_minimal*:
fixes *x* :: 'a
assumes *P* *x*
shows $\exists x. P\ x \wedge (\forall y. P\ y \longrightarrow y \geq x)$
<proof>

2.9 Lists

lemma *rev_induct2*[*consumes* 1, *case_names* *Nil* *snoc*]:
length *xs* = *length* *ys* \implies *P* [] [] \implies
 $(\bigwedge x\ xs\ y\ ys. \text{length}\ xs = \text{length}\ ys \implies P\ xs\ ys \implies P\ (xs\ @\ [x])\ (ys\ @\ [y])) \implies P\ xs\ ys$
<proof>

lemma *hd_in_set*: *length* *xs* $\neq 0 \implies$ *hd* *xs* \in *set* *xs*
<proof>

lemma *in_lists_iff_set*: *xs* \in *lists* *A* \longleftrightarrow *set* *xs* \subseteq *A*
<proof>

lemma *butlast_append_Cons*[*simp*]: *butlast* (*xs* @ *y* # *ys*) = *xs* @ *butlast* (*y* # *ys*)
<proof>

lemma *rev_in_lists*[*simp*]: *rev* *xs* \in *lists* *A* \longleftrightarrow *xs* \in *lists* *A*
<proof>

lemma *hd_le_sum_list*:
fixes *xs* :: 'a::ordered_ab_semigroup_monoid_add_imp_le *list*
assumes *xs* $\neq []$ **and** $\forall i < \text{length}\ xs. xs\ !\ i \geq 0$
shows *hd* *xs* \leq *sum_list* *xs*
<proof>

lemma *sum_list_ge_length_times*:
fixes *a* :: 'a::ordered_ab_semigroup_monoid_add_semiring_1
assumes $\forall i < \text{length}\ xs. xs\ !\ i \geq a$
shows *sum_list* *xs* \geq *of_nat* (*length* *xs*) * *a*
<proof>

lemma *prod_list_nonneg*:
fixes *xs* :: ('a :: {ordered_semiring_0, linordered_nonzero_semiring}) *list*
assumes $\bigwedge x. x \in \text{set}\ xs \implies x \geq 0$
shows *prod_list* *xs* ≥ 0
<proof>

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⟩

lemma *zip_eq_butlast_last*:
assumes *len_gt0*: $length\ xs > 0$ **and** *len_eq*: $length\ xs = length\ ys$
shows $zip\ xs\ ys = zip\ (butlast\ xs)\ (butlast\ ys) @ [(last\ xs, last\ ys)]$
 ⟨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*: $m \leq n - 1 \implies n \neq \infty \implies n \neq 0 \implies m < n$ **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]*: $n \leq m \implies m = m - n + n$ **for** $m\ n :: enat$
 ⟨proof⟩

lemma *enat_the_enat_iden[simp]*: $n \neq \infty \implies enat\ (the_enat\ n) = n$
 ⟨proof⟩

lemma *the_enat_minus_nat*: $m \neq \infty \implies the_enat\ (m - enat\ n) = the_enat\ m - n$
 ⟨proof⟩

lemma *enat_the_enat_le*: $enat\ (the_enat\ x) \leq x$
 ⟨proof⟩

lemma *enat_the_enat_minus_le*: $enat\ (the_enat\ (x - y)) \leq x$
 ⟨proof⟩

lemma *enat_le_imp_minus_le*: $k \leq m \implies k - n \leq m$ **for** $k\ m\ n :: enat$
 ⟨proof⟩

lemma *add_diff_assoc2_enat*: $m \geq n \implies m - n + p = m + p - n$ **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*: $a < b \implies add_mset\ a\ A < add_mset\ b\ A$
 ⟨proof⟩

lemma *add_mset_le_left_le*: $a \leq b \implies add_mset\ a\ A \leq add_mset\ b\ A$ **for** $a :: 'a :: linorder$
 ⟨proof⟩

lemma *add_mset_lt_right_lt*: $A < B \implies add_mset\ a\ A < add_mset\ a\ B$
 ⟨proof⟩

lemma *add_mset_le_right_le*: $A \leq B \implies add_mset\ a\ A \leq add_mset\ a\ B$
 ⟨proof⟩

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

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

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

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

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

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**
 $A = set_mset\ M \cup set_mset\ N$ **and**
 $count\ (image_mset\ f\ M)\ b < count\ (image_mset\ f\ N)\ b$
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 $\exists x. f\ x = b \wedge count\ M\ x < count\ N\ x$
 $\langle proof \rangle$

lemma $count_image_mset_le_imp_lt_raw:$
assumes

finite A and
 $A = \text{set_mset } M \cup \text{set_mset } N$ **and**
 $\text{count } (\text{image_mset } f M) (f a) + \text{count } N a < \text{count } (\text{image_mset } f N) (f a) + \text{count } M a$
shows $\exists b. f b = f a \wedge \text{count } M b < \text{count } N b$
 ⟨*proof*⟩

lemma *count_image_mset_le_imp_lt*:
assumes
 $\text{count } (\text{image_mset } f M) (f a) \leq \text{count } (\text{image_mset } f N) (f a)$ **and**
 $\text{count } M a > \text{count } N a$
shows $\exists b. f b = f a \wedge \text{count } M b < \text{count } N b$
 ⟨*proof*⟩

lemma *Max_in_mset*: $M \neq \{\#\} \implies \text{Max_mset } M \in\# M$
 ⟨*proof*⟩

lemma *Max_lt_imp_lt_mset*:
assumes $n_nemp: N \neq \{\#\}$ **and** $max: \text{Max_mset } M < \text{Max_mset } N$ (**is** $?max_M < ?max_N$)
shows $M < N$
 ⟨*proof*⟩

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

end

3 Lambda-Free Higher-Order Terms

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

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

3.1 Precedence on Symbols

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

lemma *gt_sym_antisym*: $f >_s g \implies \neg g >_s f$
 ⟨*proof*⟩

end

3.2 Heads

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

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

lemma *finite_vars_hd*[simp]: *finite (vars_hd ζ)*
 ⟨*proof*⟩

lemma *finite_syms_hd*[simp]: *finite (syms_hd ζ)*
 ⟨*proof*⟩

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*
 ⟨*proof*⟩

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

lemma *head_fun*[simp]: *head (fun s) = head s*
 ⟨*proof*⟩

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*
 ⟨*proof*⟩

lemma

finite_vars[simp]: *finite* (*vars* *s*) **and**
finite_syms[simp]: *finite* (*syms* *s*)
 ⟨*proof*⟩

lemma

vars_head_subseteq: *vars_hd* (*head* *s*) ⊆ *vars* *s* **and**
syms_head_subseteq: *syms_hd* (*head* *s*) ⊆ *syms* *s*
 ⟨*proof*⟩

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*)
 ⟨*proof*⟩

lemma *arg_in_args*: $is_App\ s \implies arg\ s \in set\ (args\ s)$
 ⟨proof⟩

lemma
vars_args_subseteq: $si \in set\ (args\ s) \implies vars\ si \subseteq vars\ s$ **and**
syms_args_subseteq: $si \in set\ (args\ s) \implies syms\ si \subseteq syms\ s$
 ⟨proof⟩

lemma *args_Nil_iff_is_Hd*: $args\ s = [] \iff is_Hd\ s$
 ⟨proof⟩

abbreviation *num_args* :: $('s, 'v)\ tm \Rightarrow nat$ **where**
num_args $s \equiv length\ (args\ s)$

lemma *size_ge_num_args*: $size\ s \geq num_args\ s$
 ⟨proof⟩

lemma *Hd_head_id*: $num_args\ s = 0 \implies Hd\ (head\ s) = s$
 ⟨proof⟩

lemma *one_arg_imp_Hd*: $num_args\ s = 1 \implies s = App\ t\ u \implies t = Hd\ (head\ t)$
 ⟨proof⟩

lemma *size_in_args*: $s \in set\ (args\ t) \implies size\ s < size\ t$
 ⟨proof⟩

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

lemma
vars_apps[simp]: $vars\ (apps\ s\ ss) = vars\ s \cup (\bigcup s \in set\ ss.\ vars\ s)$ **and**
syms_apps[simp]: $syms\ (apps\ s\ ss) = syms\ s \cup (\bigcup s \in set\ ss.\ syms\ s)$ **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]: $is_App\ (apps\ s\ ss) \iff args\ (apps\ s\ ss) \neq []$ **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]*: $apps\ (Hd\ \zeta)\ ss = apps\ (Hd\ \xi)\ ts \iff \zeta = \xi \wedge ss = ts$
 ⟨proof⟩

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

lemma *tm_expand_apps*: $head\ s = head\ t \implies args\ s = args\ t \implies s = t$
 ⟨proof⟩

lemma *tm_exhaust_apps_sel[case_names apps]*: $(s = apps\ (Hd\ (head\ s))\ (args\ s) \implies P) \implies P$
 ⟨proof⟩

lemma *tm_exhaust_apps[case_names apps]*: $(\bigwedge \zeta\ ss.\ s = apps\ (Hd\ \zeta)\ ss \implies P) \implies P$
 ⟨proof⟩

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

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

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

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

primrec *vars_mset* :: $(\text{'s}, \text{'v}) \text{tm} \Rightarrow \text{'v multiset}$ **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*]: $\text{set_mset } (\text{vars_mset } t) = \text{vars } t$
<proof>

lemma *vars_mset_empty_iff*[*iff*]: $\text{vars_mset } s = \{\#\} \iff \text{ground } s$
<proof>

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

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

3.4 Substitutions

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

lemma *subst_apps*[*simp*]: $\text{subst } \rho (\text{apps } s \text{ts}) = \text{apps } (\text{subst } \rho s) (\text{map } (\text{subst } \rho) \text{ts})$
<proof>

lemma *head_subst*[*simp*]: $\text{head } (\text{subst } \rho s) = \text{head } (\text{subst } \rho (\text{Hd } (\text{head } s)))$
<proof>

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

lemma *ground_imp_subst_iden*: $\text{ground } s \implies \text{subst } \rho s = s$
<proof>

lemma *vars_mset_subst*[*simp*]: $\text{vars_mset } (\text{subst } \rho s) = (\bigcup\# \{\#\text{vars_mset } (\rho x). x \in\# \text{vars_mset } s\#\})$
<proof>

lemma *vars_mset_subst_subseteq*:
 $\text{vars_mset } t \supseteq\# \text{vars_mset } s \implies \text{vars_mset } (\text{subst } \rho t) \supseteq\# \text{vars_mset } (\text{subst } \rho s)$
<proof>

lemma *vars_subst_subseteq*: $\text{vars } t \supseteq \text{vars } s \implies \text{vars } (\text{subst } \rho t) \supseteq \text{vars } (\text{subst } \rho s)$
<proof>

3.5 Subterms

inductive *sub* :: $(\text{'s}, \text{'v}) \text{tm} \Rightarrow (\text{'s}, \text{'v}) \text{tm} \Rightarrow \text{bool}$ **where**

$sub_refl: sub\ s\ s$
 $|\ sub_fun: sub\ s\ t \implies sub\ s\ (App\ u\ t)$
 $|\ sub_arg: sub\ s\ t \implies sub\ s\ (App\ t\ u)$

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

lemma $in_vars_imp_sub: x \in vars\ s \iff sub\ (Hd\ (Var\ x))\ s$
 $\langle proof \rangle$

lemma $sub_args: s \in set\ (args\ t) \implies sub\ s\ t$
 $\langle proof \rangle$

lemma $sub_size: sub\ s\ t \implies size\ s \leq size\ t$
 $\langle proof \rangle$

lemma $sub_subst: sub\ s\ t \implies sub\ (subst\ \rho\ s)\ (subst\ \rho\ t)$
 $\langle proof \rangle$

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

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

lemma $proper_sub_subst:$
assumes $p\ sub: proper_sub\ s\ t$
shows $proper_sub\ (subst\ \rho\ s)\ (subst\ \rho\ t)$
 $\langle proof \rangle$

3.6 Maximum Arities

locale $arity =$

fixes

$arity_sym :: 's \Rightarrow enat$ **and**

$arity_var :: 'v \Rightarrow enat$

begin

primrec $arity_hd :: ('s, 'v)\ hd \Rightarrow enat$ **where**

$arity_hd\ (Var\ x) = arity_var\ x$

$|\ arity_hd\ (Sym\ f) = arity_sym\ f$

definition $arity :: ('s, 'v)\ tm \Rightarrow enat$ **where**

$arity\ s = arity_hd\ (head\ s) - num_args\ s$

lemma $arity_simps[simp]:$

$arity\ (Hd\ \zeta) = arity_hd\ \zeta$

$arity\ (App\ s\ t) = arity\ s - 1$

$\langle proof \rangle$

lemma $arity_apps[simp]: arity\ (apps\ s\ ts) = arity\ s - length\ ts$

$\langle proof \rangle$

inductive $wary :: ('s, 'v)\ tm \Rightarrow bool$ **where**

$wary_Hd[intro]: wary\ (Hd\ \zeta)$

$|\ wary_App[intro]: wary\ s \implies wary\ t \implies num_args\ s < arity_hd\ (head\ s) \implies wary\ (App\ s\ t)$

inductive-cases $wary_HdE: wary\ (Hd\ \zeta)$

inductive-cases $wary_AppE: wary\ (App\ s\ t)$

inductive-cases $wary_binaryE[simplified]: wary\ (App\ (App\ s\ t)\ u)$

lemma $wary_fun[intro]: wary\ t \implies wary\ (fun\ t)$

<proof>

lemma *wary_arg*[*intro*]: $wary\ t \implies wary\ (arg\ t)$
<proof>

lemma *wary_args*: $s \in set\ (args\ t) \implies wary\ t \implies wary\ s$
<proof>

lemma *wary_sub*: $sub\ s\ t \implies wary\ t \implies wary\ s$
<proof>

lemma *wary_inf_ary*: $(\bigwedge \zeta. arity_hd\ \zeta = \infty) \implies wary\ s$
<proof>

lemma *wary_num_args_le_arity_head*: $wary\ s \implies num_args\ s \leq arity_hd\ (head\ s)$
<proof>

lemma *wary_apps*:
 $wary\ s \implies (\bigwedge sa. sa \in set\ ss \implies wary\ sa) \implies length\ ss \leq arity\ s \implies wary\ (apps\ s\ ss)$
<proof>

lemma *wary_cases_apps*[*consumes 1, case_names apps*]:
assumes
 wary_t: $wary\ t$ **and**
 apps: $\bigwedge \zeta. t = apps\ (Hd\ \zeta)\ ss \implies (\bigwedge sa. sa \in set\ ss \implies wary\ sa) \implies length\ ss \leq arity_hd\ \zeta \implies P$
shows *P*
<proof>

lemma *arity_hd_head*: $wary\ s \implies arity_hd\ (head\ s) = arity\ s + num_args\ s$
<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*]: $is_Hd\ s \vee is_Sym\ (head\ s) \implies length\ (args\ s) = arity_hd\ (head\ s) \implies$
 $(\forall t \in set\ (args\ s). wary_fo\ t) \implies wary_fo\ s$

lemma *wary_fo_args*: $s \in set\ (args\ t) \implies wary_fo\ t \implies wary_fo\ s$
<proof>

lemma *wary_fo_arg*: $wary_fo\ (App\ s\ t) \implies wary_fo\ t$
<proof>

end

3.7 Potential Heads of Ground Instances of Variables

locale *ground_heads* = *gt_sym op >_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: $f \in ground_heads_var\ x \implies arity_sym\ f \geq arity_var\ x$ **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*: $f \in \text{ground_heads } \zeta \implies \text{arity_sym } f \geq \text{arity_hd } \zeta$
 ⟨proof⟩

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

lemma *sym_in_ground_heads*: $\text{is_Sym } \zeta \implies \text{sym } \zeta \in \text{ground_heads } \zeta$
 ⟨proof⟩

lemma *ground_hd_in_ground_heads*: $\text{ground } s \implies \text{sym } (\text{head } s) \in \text{ground_heads } (\text{head } s)$
 ⟨proof⟩

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

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

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

lemma *strict_imp_wary_subst*: $\text{strict_wary_subst } \rho \implies \text{wary_subst } \rho$
 ⟨proof⟩

lemma *wary_subst_wary*:
assumes $\text{wary_}\rho$: $\text{wary_subst } \rho$ **and** $\text{wary_}s$: $\text{wary } s$
shows $\text{wary } (\text{subst } \rho s)$
 ⟨proof⟩

lemmas $\text{strict_wary_subst_wary} = \text{wary_subst_wary}[\text{OF } \text{strict_imp_wary_subst}]$

lemma *wary_subst_ground_heads*:
assumes $\text{wary_}\rho$: $\text{wary_subst } \rho$
shows $\text{ground_heads } (\text{head } (\text{subst } \rho s)) \subseteq \text{ground_heads } (\text{head } s)$
 ⟨proof⟩

lemmas $\text{strict_wary_subst_ground_heads} = \text{wary_subst_ground_heads}[\text{OF } \text{strict_imp_wary_subst}]$

definition *grounding_* ρ :: $'v \Rightarrow ('s, 'v) \text{tm}$ **where**
 $\text{grounding_}\rho x = (\text{let } s = \text{Hd } (\text{Sym } (\text{SOME } f. f \in \text{ground_heads_var } x)) \text{ in}$
 $\text{apps } s (\text{replicate } (\text{the_enat } (\text{arity } s - \text{arity_var } x)) s))$

lemma *ground_grounding_* ρ : $\text{ground } (\text{subst } \text{grounding_}\rho s)$
 ⟨proof⟩

lemma *strict_wary_grounding_* ρ : $\text{strict_wary_subst } \text{grounding_}\rho$
 ⟨proof⟩

lemmas $\text{wary_grounding_}\rho = \text{strict_wary_grounding_}\rho[\text{THEN } \text{strict_imp_wary_subst}]$

definition *gt_hd* :: $('s, 'v) \text{hd} \Rightarrow ('s, 'v) \text{hd} \Rightarrow \text{bool}$ (**infix** $>_{hd}$ 50) **where**
 $\xi >_{hd} \zeta \longleftrightarrow (\forall g \in \text{ground_heads } \xi. \forall f \in \text{ground_heads } \zeta. g >_s f)$

definition *comp_hd* :: $('s, 'v) \text{hd} \Rightarrow ('s, 'v) \text{hd} \Rightarrow \text{bool}$ (**infix** \leq_{hd} 50) **where**
 $\xi \leq_{hd} \zeta \longleftrightarrow \xi = \zeta \vee \xi >_{hd} \zeta \vee \zeta >_{hd} \xi$

lemma *gt_hd_irrefl*: $\neg \zeta >_{hd} \zeta$
 ⟨proof⟩

lemma *gt_hd_trans*: $\chi >_{hd} \xi \implies \xi >_{hd} \zeta \implies \chi >_{hd} \zeta$
 ⟨proof⟩

lemma *gt_sym_imp_hd*: $g >_s f \implies \text{Sym } g >_{hd} \text{Sym } f$
 ⟨proof⟩

lemma *not_comp_hd_imp_Var*: $\neg \xi \leq_{hd} \zeta \implies \text{is_Var } \zeta \vee \text{is_Var } \xi$
 ⟨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 $p :: 'a \Rightarrow 'a \Rightarrow \text{bool}$
begin

definition *inf_chain* :: $(\text{nat} \Rightarrow 'a) \Rightarrow \text{bool}$ **where**
inf_chain $f \longleftrightarrow (\forall i. p (f i) (f (\text{Suc } i)))$

lemma *wfP_iff_no_inf_chain*: $\text{wfP } (\lambda x y. p y x) \longleftrightarrow (\nexists f. \text{inf_chain } f)$
 ⟨proof⟩

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

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

lemma *inf_chain_bad*:
assumes *bad_f*: $\text{inf_chain } f$
shows *bad* $(f i)$
 ⟨proof⟩

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

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

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

context
fixes $x :: 'a$
assumes *x_bad*: $\text{bad } x$
begin

lemma
bad_worst_chain_0: $\text{bad } (\text{worst_chain } 0)$ **and**
min_worst_chain_0: $\neg gt (\text{worst_chain } 0) x$
 ⟨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  $\implies \neg$  gt (worst_chain (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:  $\neg$  gt (worst_chain i) (worst_chain (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:  $(\forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } y x \longrightarrow \text{gt}' y x) \implies \text{ext } \text{gt } ys \ xs \implies \text{ext } \text{gt}' ys \ xs$  and
    map: finite A  $\implies ys \in \text{lists } A \implies xs \in \text{lists } A \implies (\forall x \in A. \neg \text{gt } (f x) (f x)) \implies$ 
       $(\forall z \in A. \forall y \in A. \forall x \in A. \text{gt } (f z) (f y) \longrightarrow \text{gt } (f y) (f x) \longrightarrow \text{gt } (f z) (f x)) \implies$ 
       $(\forall y \in A. \forall x \in A. \text{gt } y x \longrightarrow \text{gt } (f y) (f x)) \implies \text{ext } \text{gt } ys \ xs \implies \text{ext } \text{gt } (\text{map } f \ ys) (\text{map } f \ xs)$ 
begin

```

```

lemma mono[mono]: gt  $\leq$  gt'  $\implies \text{ext } \text{gt} \leq \text{ext } \text{gt}'$ 
<proof>

```

end

```

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

```

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*: $(\forall z \in set\ xs. \forall y \in set\ xs. \forall x \in set\ xs. gt\ z\ y \longrightarrow gt\ y\ x \longrightarrow gt\ z\ x) \implies$
 $(\forall x \in set\ xs. \neg\ gt\ x\ x) \implies \neg\ ext\ gt\ xs\ xs$

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

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*: $ext\ gt\ ys\ xs \implies ext\ gt\ (x\ \#\ ys)\ (x\ \#\ xs)$
begin

lemma *compat_append_left*: $ext\ gt\ ys\ xs \implies ext\ gt\ (zs\ @\ ys)\ (zs\ @\ xs)$
<proof>

end

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

lemma *compat_append_right*: $ext\ gt\ ys\ xs \implies ext\ gt\ (ys\ @\ zs)\ (xs\ @\ zs)$
<proof>

end

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

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

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

lemma *compat_list*: $y \neq x \implies gt\ y\ x \implies ext\ gt\ (xs\ @\ y\ \#\ xs')\ (xs\ @\ x\ \#\ xs')$
<proof>

end

sublocale *ext_compat_list_strong* < *ext_compat_list*
 ⟨*proof*⟩

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

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

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

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: $\bigwedge z\ y\ x. gt\ z\ y \implies gt\ y\ x \implies gt\ z\ x$ **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*: $\neg ext\ gt\ xs\ xs$
 ⟨*proof*⟩

lemma *trans_gt*: $ext\ gt\ zs\ ys \implies ext\ gt\ ys\ xs \implies ext\ gt\ zs\ xs$
 ⟨*proof*⟩

lemma *hd_or_tl_gt*: $length\ ys = length\ xs \implies ext\ gt\ (y\ \#\ ys)\ (x\ \#\ xs) \implies gt\ y\ x \vee ext\ gt\ ys\ xs$
 ⟨*proof*⟩

lemma *wf_same_length_if_total*: $wfP\ (\lambda xs\ ys. length\ ys = n \wedge length\ xs = n \wedge ext\ gt\ ys\ xs)$
 ⟨*proof*⟩

lemma *wf_bounded_if_total*: $wfP\ (\lambda xs\ ys. length\ ys \leq n \wedge length\ xs \leq n \wedge ext\ gt\ ys\ xs)$
 ⟨*proof*⟩

end

context

fixes *gt* :: 'a \Rightarrow 'a \Rightarrow bool

assumes

gt_irrefl: $\bigwedge z. \neg gt\ z\ z$ **and**

gt_wf: $wfP\ (\lambda x\ y. gt\ y\ x)$

begin

lemma *wf_bounded*: $wfP\ (\lambda xs\ ys. length\ ys \leq n \wedge length\ xs \leq n \wedge ext\ gt\ ys\ xs)$
 ⟨*proof*⟩

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]*:
 $lexext\ gt\ ys\ [] \longleftrightarrow ys \neq []$
 $\neg lexext\ gt\ []\ xs$
 $lexext\ gt\ (y\ \#\ ys)\ (x\ \#\ xs) \longleftrightarrow gt\ y\ x \vee x = y \wedge lexext\ gt\ ys\ xs$
<proof>

lemma *lexext_mono_strong*:
assumes
 $\forall y \in set\ ys. \forall x \in set\ xs. gt\ y\ x \longrightarrow gt'\ y\ x$ **and**
 $lexext\ gt\ ys\ xs$
shows $lexext\ gt'\ ys\ xs$
<proof>

lemma *lexext_map_strong*:
 $(\forall y \in set\ ys. \forall x \in set\ xs. gt\ y\ x \longrightarrow gt\ (f\ y)\ (f\ x)) \Longrightarrow lexext\ gt\ ys\ xs \Longrightarrow$
 $lexext\ gt\ (map\ f\ ys)\ (map\ f\ xs)$
<proof>

lemma *lexext_irrefl*:
assumes $\forall x \in set\ xs. \neg gt\ x\ x$
shows $\neg lexext\ gt\ xs\ xs$
<proof>

lemma *lexext_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**
 $lexext\ gt\ zs\ ys$ **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*: $gt\ y\ x \Longrightarrow lexext\ gt\ (xs\ @\ y\ \#\ xs')\ (xs\ @\ x\ \#\ xs')$
<proof>

lemma *lexext_singleton*: $lexext\ gt\ [y]\ [x] \longleftrightarrow gt\ y\ x$
<proof>

lemma *lexext_total*: $(\forall y \in B. \forall x \in A. gt\ y\ x \vee gt\ x\ y \vee y = x) \Longrightarrow ys \in lists\ B \Longrightarrow xs \in lists\ A \Longrightarrow$
 $lexext\ gt\ ys\ xs \vee lexext\ gt\ xs\ ys \vee ys = xs$
<proof>

lemma *lexext_hd_or_tl*: $lexext\ gt\ (y\ \#\ ys)\ (x\ \#\ xs) \Longrightarrow gt\ y\ x \vee lexext\ gt\ ys\ xs$
<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 ⇒ '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*
<proof>

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*
<proof>

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*
<proof>

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*)
<proof>

lemma *lexext_rev_irrefl*:
assumes ∀ *x* ∈ *set* *xs*. ¬ *gt* *x* *x*
shows ¬ *lexext_rev* *gt* *xs* *xs*
<proof>

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*
<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* ⇒ *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*
 ⟨proof⟩

interpretation *lexext_rev*: *ext_compat_list lexext_rev*
 ⟨proof⟩

interpretation *lexext_rev*: *ext_singleton lexext_rev*
 ⟨proof⟩

interpretation *lexext_rev*: *ext_total lexext_rev*
 ⟨proof⟩

interpretation *lexext_rev*: *ext_hd_or_tl lexext_rev*
 ⟨proof⟩

interpretation *lexext_rev*: *ext_wf_bounded lexext_rev*
 ⟨proof⟩

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

$lenext\ gts\ ys\ xs \vee lenext\ gts\ xs\ ys \vee ys = xs$ **and**
 $lenext_hd_or_tl: (length\ ys = length\ xs \implies gts\ (y\ \# \ ys)\ (x\ \# \ xs) \implies gt\ y\ x \vee gts\ ys\ xs) \implies$
 $lenext\ gts\ (y\ \# \ ys)\ (x\ \# \ xs) \implies gt\ y\ x \vee lenext\ gts\ ys\ xs$
 <proof>

5.5 Length-Lexicographic Extension

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

lemma $len_lexext_mono_strong:$
 $(\forall y \in set\ ys. \forall x \in set\ xs. gt\ y\ x \longrightarrow gt'\ y\ x) \implies len_lexext\ gt\ ys\ xs \implies len_lexext\ gt'\ ys\ xs$
 <proof>

lemma $len_lexext_map_strong:$
 $(\forall y \in set\ ys. \forall x \in set\ xs. gt\ y\ x \longrightarrow gt\ (f\ y)\ (f\ x)) \implies len_lexext\ gt\ ys\ xs \implies$
 $len_lexext\ gt\ (map\ f\ ys)\ (map\ f\ xs)$
 <proof>

lemma $len_lexext_irrefl: (\forall x \in set\ xs. \neg gt\ x\ x) \implies \neg len_lexext\ gt\ xs\ xs$
 <proof>

lemma $len_lexext_trans_strong:$
 $(\forall z \in set\ zs. \forall y \in set\ ys. \forall x \in set\ xs. gt\ z\ y \longrightarrow gt\ y\ x \longrightarrow gt\ z\ x) \implies len_lexext\ gt\ zs\ ys \implies$
 $len_lexext\ gt\ ys\ xs \implies len_lexext\ gt\ zs\ xs$
 <proof>

lemma $len_lexext_snoc: len_lexext\ gt\ (xs\ @\ [x])\ xs$
 <proof>

lemma $len_lexext_compat_cons: len_lexext\ gt\ ys\ xs \implies len_lexext\ gt\ (x\ \# \ ys)\ (x\ \# \ xs)$
 <proof>

lemma $len_lexext_compat_snoc: len_lexext\ gt\ ys\ xs \implies len_lexext\ gt\ (ys\ @\ [x])\ (xs\ @\ [x])$
 <proof>

lemma $len_lexext_compat_list: gt\ y\ x \implies len_lexext\ gt\ (xs\ @\ y\ \# \ xs')\ (xs\ @\ x\ \# \ xs')$
 <proof>

lemma $len_lexext_singleton[simp]: len_lexext\ gt\ [y]\ [x] \longleftrightarrow gt\ y\ x$
 <proof>

lemma $len_lexext_total: (\forall y \in B. \forall x \in A. gt\ y\ x \vee gt\ x\ y \vee y = x) \implies ys \in lists\ B \implies xs \in lists\ A \implies$
 $len_lexext\ gt\ ys\ xs \vee len_lexext\ gt\ xs\ ys \vee ys = xs$
 <proof>

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

lemma $len_lexext_wf: wfP\ (\lambda x\ y. gt\ y\ x) \implies wfP\ (\lambda xs\ ys. len_lexext\ gt\ ys\ xs)$
 <proof>

lemma $len_lexext_hd_or_tl: len_lexext\ gt\ (y\ \# \ ys)\ (x\ \# \ xs) \implies gt\ y\ x \vee len_lexext\ gt\ ys\ xs$
 <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 $len_lexext: ext_compat_cons\ len_lexext$
 $\langle proof \rangle$

interpretation $len_lexext: ext_compat_snoc\ len_lexext$
 $\langle proof \rangle$

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

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

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

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

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

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

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 $len_lexext_rev_mono_strong:$
 $(\forall y \in set\ ys. \forall x \in set\ xs. gt\ y\ x \longrightarrow gt'\ y\ x) \Longrightarrow len_lexext_rev\ gt\ ys\ xs \Longrightarrow len_lexext_rev\ gt'\ ys\ xs$
 $\langle proof \rangle$

lemma $len_lexext_rev_map_strong:$
 $(\forall y \in set\ ys. \forall x \in set\ xs. gt\ y\ x \longrightarrow gt\ (f\ y)\ (f\ x)) \Longrightarrow len_lexext_rev\ gt\ ys\ xs \Longrightarrow$
 $len_lexext_rev\ gt\ (map\ f\ ys)\ (map\ f\ xs)$
 $\langle proof \rangle$

lemma $len_lexext_rev_irrefl: (\forall x \in set\ xs. \neg gt\ x\ x) \Longrightarrow \neg len_lexext_rev\ gt\ xs\ xs$
 $\langle proof \rangle$

lemma $len_lexext_rev_trans_strong:$
 $(\forall z \in set\ zs. \forall y \in set\ ys. \forall x \in set\ xs. gt\ z\ y \longrightarrow gt\ y\ x \longrightarrow gt\ z\ x) \Longrightarrow len_lexext_rev\ gt\ zs\ ys \Longrightarrow$
 $len_lexext_rev\ gt\ ys\ xs \Longrightarrow len_lexext_rev\ gt\ zs\ xs$
 $\langle proof \rangle$

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

lemma $len_lexext_rev_compat_cons: len_lexext_rev\ gt\ ys\ xs \Longrightarrow len_lexext_rev\ gt\ (x\ \#\ ys)\ (x\ \#\ xs)$
 $\langle proof \rangle$

lemma $len_lexext_rev_compat_snoc: len_lexext_rev\ gt\ ys\ xs \Longrightarrow len_lexext_rev\ gt\ (ys\ @\ [x])\ (xs\ @\ [x])$
 $\langle proof \rangle$

lemma $len_lexext_rev_compat_list: gt\ y\ x \Longrightarrow len_lexext_rev\ gt\ (xs\ @\ y\ \#\ xs')\ (xs\ @\ x\ \#\ xs')$
 $\langle proof \rangle$

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

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

lemma *len_lexext_rev_iff_len_lexext*: $\text{len_lexext_rev gt ys xs} \longleftrightarrow \text{len_lexext gt (rev ys) (rev xs)}$
 ⟨proof⟩

lemma *len_lexext_rev_wf*: $\text{wfP } (\lambda x y. \text{gt } y x) \implies \text{wfP } (\lambda xs ys. \text{len_lexext_rev gt ys xs})$
 ⟨proof⟩

lemma *len_lexext_rev_hd_or_tl*:
 $\text{len_lexext_rev gt (y \# ys) (x \# xs)} \implies \text{gt } y x \vee \text{len_lexext_rev gt ys xs}$
 ⟨proof⟩

interpretation *len_lexext_rev*: $\text{ext len_lexext_rev}$
 ⟨proof⟩

interpretation *len_lexext_rev*: $\text{ext_irrefl_trans_strong len_lexext_rev}$
 ⟨proof⟩

interpretation *len_lexext_rev*: $\text{ext_snoc len_lexext_rev}$
 ⟨proof⟩

interpretation *len_lexext_rev*: $\text{ext_compat_cons len_lexext_rev}$
 ⟨proof⟩

interpretation *len_lexext_rev*: $\text{ext_compat_snoc len_lexext_rev}$
 ⟨proof⟩

interpretation *len_lexext_rev*: $\text{ext_compat_list len_lexext_rev}$
 ⟨proof⟩

interpretation *len_lexext_rev*: $\text{ext_singleton len_lexext_rev}$
 ⟨proof⟩

interpretation *len_lexext_rev*: $\text{ext_total len_lexext_rev}$
 ⟨proof⟩

interpretation *len_lexext_rev*: $\text{ext_wf len_lexext_rev}$
 ⟨proof⟩

interpretation *len_lexext_rev*: $\text{ext_hd_or_tl len_lexext_rev}$
 ⟨proof⟩

interpretation *len_lexext_rev*: $\text{ext_wf_bounded len_lexext_rev}$
 ⟨proof⟩

5.7 Dershowitz–Manna Multiset Extension

definition *msetext_dersh* **where**

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

The following proof is based on that of *less_multiset_{DM}_imp_mult*.

lemma *msetext_dersh_imp_mult_rel*:

assumes

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

$\text{ys_gt_xs: } \text{msetext_dersh gt ys xs}$

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

⟨proof⟩

lemma *msetext_dersh_imp_mult*: $\text{msetext_dersh gt ys xs} \implies (\text{mset } xs, \text{mset } ys) \in \text{mult } \{(x, y). \text{gt } y x\}$
 ⟨proof⟩

lemma *mult_imp_msetext_dersh_rel*:

assumes

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

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

lemma *msetext_dersh_mono_strong*:
 $(\forall y \in set\ ys. \forall x \in set\ xs. gt\ y\ x \longrightarrow gt'\ y\ x) \implies msetext_dersh\ gt\ ys\ xs \implies$
 $msetext_dersh\ gt'\ ys\ xs$
 <proof>

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)*
 <proof>

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 = (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 $\exists x. \text{count } (mset \text{ ys}) \ x > \text{count } (mset \text{ xs}) \ x$

<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 ⇒ transp gt ⇒ msetext_dersh gt = msetext_huet gt*
<proof>

lemma *msetext_huet_mono_strong:*

$(\forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } y \ x \rightarrow \text{gt}' \ y \ x) \Rightarrow \text{msetext_huet } \text{gt } \text{ys } \text{xs} \Rightarrow \text{msetext_huet } \text{gt}' \ \text{ys } \text{xs}$

<proof>

lemma *msetext_huet_map:*

assumes

fin: finite A and

ys_a: ys ∈ lists A and xs_a: xs ∈ lists A and

irrefl_f: ∀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>

lemma *msetext_huet_irrefl: (∀x ∈ set xs. ¬ gt x x) ⇒ ¬ msetext_huet gt xs xs*
<proof>

lemma *msetext_huet_trans_from_irrefl*:

assumes

fin: *finite A* **and**

zs_a: *zs* \in *lists A* **and** *ys_a*: *ys* \in *lists A* **and** *xs_a*: *xs* \in *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 gt y x \implies msetext_huet gt (xs @ y # xs') (xs @ x # xs')*

<proof>

lemma *msetext_huet_singleton*: *y \neq x \implies msetext_huet gt [y] [x] \longleftrightarrow gt y x*

<proof>

lemma *msetext_huet_wf*: *wfP ($\lambda x y. \text{gt } y \ x$) \implies wfP ($\lambda xs ys. \text{msetext_huet } \text{gt } ys \ xs$)*

<proof>

lemma *msetext_huet_hd_or_tl*:

assumes

trans: $\forall z y x. \text{gt } z \ y \longrightarrow \text{gt } y \ x \longrightarrow \text{gt } z \ x$ **and**

total: $\forall y x. \text{gt } y \ x \vee \text{gt } x \ y \vee y = x$ **and**

len_eq: *length ys = length xs* **and**

yys_gt_xxs: *msetext_huet gt (y # ys) (x # xs)*

shows *gt y x \vee msetext_huet gt ys xs*

<proof>

interpretation *msetext_huet*: *ext msetext_huet*

<proof>

interpretation *msetext_huet*: *ext_irrefl_before_trans msetext_huet*

<proof>

interpretation *msetext_huet*: *ext_snoc msetext_huet*

<proof>

interpretation *msetext_huet*: *ext_compat_cons msetext_huet*

<proof>

interpretation *msetext_huet*: *ext_compat_snoc msetext_huet*

<proof>

interpretation *msetext_huet*: *ext_compat_list msetext_huet*

<proof>

interpretation *msetext_huet*: *ext_singleton msetext_huet*

<proof>

interpretation *msetext_huet*: *ext_wf msetext_huet*

<proof>

interpretation *msetext_huet*: *ext_hd_or_tl msetext_huet*
(*proof*)

interpretation *msetext_huet*: *ext_wf_bounded msetext_huet*
(*proof*)

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*)

lemma *cwiseext_mono_strong*:
(∀ y ∈ set ys. ∀ x ∈ set xs. *gt y x* → *gt' y x*) ⇒ *cwiseext gt ys xs* ⇒ *cwiseext gt' ys xs*
(*proof*)

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)
(*proof*)

lemma *cwiseext_irrefl*: (∀ x ∈ set xs. ¬ *gt x x*) ⇒ ¬ *cwiseext gt xs xs*
(*proof*)

lemma *cwiseext_trans_strong*:
assumes
∀ z ∈ set zs. ∀ y ∈ set ys. ∀ x ∈ set xs. *gt z y* → *gt y x* → *gt z x* **and**
cwiseext gt zs ys **and** *cwiseext gt ys xs*
shows *cwiseext gt zs xs*
(*proof*)

lemma *cwiseext_compat_cons*: *cwiseext gt ys xs* ⇒ *cwiseext gt (x # ys) (x # xs)*
(*proof*)

lemma *cwiseext_compat_snoc*: *cwiseext gt ys xs* ⇒ *cwiseext gt (ys @ [x]) (xs @ [x])*
(*proof*)

lemma *cwiseext_compat_list*:
assumes *y_gt_x*: *gt y x*
shows *cwiseext gt (xs @ y # xs') (xs @ x # xs')*
(*proof*)

lemma *cwiseext_singleton*: *cwiseext gt [y] [x]* ↔ *gt y x*
(*proof*)

lemma *cwiseext_wf*: *wfP (λx y. gt y x)* ⇒ *wfP (λxs ys. cwiseext gt ys xs)*
(*proof*)

lemma *cwiseext_hd_or_tl*: *cwiseext gt (y # ys) (x # xs)* ⇒ *gt y x* ∨ *cwiseext gt ys xs*
(*proof*)

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  $\implies$  ext gt ys xs  $\implies$  ext gt zs xs
begin
lemma
  assumes ys_gtcw_xs: cwiseext gt ys xs
  shows ext gt ys xs
  <proof>

end

end

interpretation cwiseext: ext cwiseext
  <proof>

interpretation cwiseext: ext_irrefl_trans_strong cwiseext
  <proof>

interpretation cwiseext: ext_compat_cons cwiseext
  <proof>

interpretation cwiseext: ext_compat_snoc cwiseext
  <proof>

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
  ≥t = ≥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 op >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_irrefl: ext_trans_before_irrefl ext and
    ext_ext_compat_list: ext_compat_list ext
begin

```

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

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

abbreviation *ge* :: ('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool (**infix** ≥_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*
abbrevs
 >_t = >_t
 ≥_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 op >_s arity_sym arity_var*
for
 | *gt_sym* :: 's ⇒ 's ⇒ bool (**infix** >_s 50) **and**
 | *arity_sym* :: 's ⇒ enat **and**
 | *arity_var* :: 'v ⇒ enat +
fixes
 | *extf* :: 's ⇒ (('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool) ⇒ ('s, 'v) tm list ⇒ ('s, 'v) tm list ⇒ bool
assumes
 | *extf_ext_trans_before_irrefl*: *ext_trans_before_irrefl* (*extf*) **and**
 | *extf_ext_compat_cons*: *ext_compat_cons* (*extf*) **and**
 | *extf_ext_compat_list*: *ext_compat_list* (*extf*)
begin

lemma *extf_ext_trans*: *ext_trans* (*extf*)
 ⟨*proof*⟩

lemma *extf_ext*: *ext* (*extf*)
 ⟨*proof*⟩

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*]: $chkvar\ t\ s \longleftrightarrow vars_hd\ (head\ s) \subseteq vars\ t$

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

$[simp]: chksubs\ gt\ t\ s \iff (case\ s\ of\ App\ s1\ s2 \Rightarrow\ gt\ t\ s1 \wedge\ gt\ t\ s2 \mid _ \Rightarrow\ True)$

lemma $chksubs_mono[mono]: gt \leq gt' \implies chksubs\ gt \leq chksubs\ gt'$

<proof>

inductive $gt :: ('s, 'v) tm \Rightarrow ('s, 'v) tm \Rightarrow bool$ (**infix** $>_t$ 50) **where**

$gt_sub: is_App\ t \implies (fun\ t >_t\ s \vee fun\ t = s) \vee (arg\ t >_t\ s \vee arg\ t = s) \implies t >_t\ s$
 $| gt_diff: head\ t >_{hd}\ head\ s \implies chkvar\ t\ s \implies chksubs\ (op >_t)\ t\ s \implies t >_t\ s$
 $| gt_same: head\ t = head\ s \implies chksubs\ (op >_t)\ t\ s \implies$
 $(\forall f \in ground_heads\ (head\ t). extf\ f\ (op >_t)\ (args\ t)\ (args\ s)) \implies t >_t\ s$

abbreviation $ge :: ('s, 'v) tm \Rightarrow ('s, 'v) tm \Rightarrow bool$ (**infix** \geq_t 50) **where**

$t \geq_t\ s \equiv t >_t\ s \vee t = s$

inductive $gt_sub :: ('s, 'v) tm \Rightarrow ('s, 'v) tm \Rightarrow bool$ **where**

$gt_subI: is_App\ t \implies fun\ t \geq_t\ s \vee arg\ t \geq_t\ s \implies gt_sub\ t\ s$

inductive $gt_diff :: ('s, 'v) tm \Rightarrow ('s, 'v) tm \Rightarrow bool$ **where**

$gt_diffI: head\ t >_{hd}\ head\ s \implies chkvar\ t\ s \implies chksubs\ (op >_t)\ t\ s \implies gt_diff\ t\ s$

inductive $gt_same :: ('s, 'v) tm \Rightarrow ('s, 'v) tm \Rightarrow bool$ **where**

$gt_sameI: head\ t = head\ s \implies chksubs\ (op >_t)\ t\ s \implies$
 $(\forall f \in ground_heads\ (head\ t). extf\ f\ (op >_t)\ (args\ t)\ (args\ s)) \implies gt_same\ t\ s$

lemma $gt_iff_sub_diff_same: t >_t\ s \iff gt_sub\ t\ s \vee gt_diff\ t\ s \vee gt_same\ t\ s$

<proof>

7.3 Transitivity

lemma $gt_fun_imp: fun\ t >_t\ s \implies t >_t\ s$

<proof>

lemma $gt_arg_imp: arg\ t >_t\ s \implies t >_t\ s$

<proof>

lemma $gt_imp_vars: t >_t\ s \implies vars\ t \supseteq vars\ s$

<proof>

theorem $gt_trans: u >_t\ t \implies t >_t\ s \implies u >_t\ s$

<proof>

7.4 Irreflexivity

theorem $gt_irrefl: \neg s >_t\ s$

<proof>

lemma $gt_antisym: t >_t\ s \implies \neg s >_t\ t$

<proof>

7.5 Subterm Property

lemma

gt_sub_fun: $App\ s\ t >_t\ s$ and
gt_sub_arg: $App\ s\ t >_t\ t$
<proof>

theorem *gt_proper_sub*: $proper_sub\ s\ t \implies t >_t\ s$
<proof>

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>

theorem *gt_compat_fun_strong*:

assumes t'_gt_t : $t' >_t\ t$
shows $apps\ s\ (t' \# us) >_t\ apps\ s\ (t \# us)$
<proof>

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 $App\ s'\ t >_t\ App\ s\ t$
<proof>

7.8 Stability under Substitution

lemma *gt_imp_chksubs_gt*:

assumes t_gt_s : $t >_t\ s$
shows $chksubs\ (op >_t)\ t\ s$
<proof>

theorem *gt_subst*:

assumes *wary_ρ*: $wary_subst\ \rho$
shows $t >_t\ s \implies subst\ \rho\ t >_t\ subst\ \rho\ s$
<proof>

7.9 Totality on Ground Terms

theorem *gt_total_ground*:

assumes *extf_total*: $\bigwedge f. ext_total\ (extf\ f)$
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**
 $op >_{tg} \equiv \lambda t\ s. ground\ t \wedge t >_t\ s$

theorem *gt_wf*:

assumes *extf_wf*: $\bigwedge f. ext_wf\ (extf\ f)$
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  $\iff$  ( $\forall s' \in \text{set } (\text{args } s).$  gt t s')
```

```
lemma chkargs_mono[mono]: gt  $\leq$  gt'  $\implies$  chkargs gt  $\leq$  chkargs gt'
  <proof>
```

inductive gt :: ('s, 'v) tm \Rightarrow ('s, 'v) tm \Rightarrow bool (infix $>_t$ 50) **where**

```
gt_arg: ti  $\in$  set (args t)  $\implies$  ti  $>_t$  s  $\vee$  ti = s  $\implies$  t  $>_t$  s
| gt_diff: head t  $>_{hd}$  head s  $\implies$  chkvar t s  $\implies$  chkargs (op  $>_t$ ) t s  $\implies$  t  $>_t$  s
| gt_same: head t = head s  $\implies$  chkargs (op  $>_t$ ) t s  $\implies$ 
  ( $\forall f \in \text{ground\_heads } (\text{head } t).$  extf f (op  $>_t$ ) (args t) (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  $\in$  set (args t)  $\implies$  ti  $>_t$  s  $\implies$  t  $>_t$  s
  <proof>
```

```
lemma gt_imp_vars: t  $>_t$  s  $\implies$  vars t  $\supseteq$  vars s
  <proof>
```

```
lemma gt_trans: u  $>_t$  t  $\implies$  t  $>_t$  s  $\implies$  u  $>_t$  s
  <proof>
```

```
lemma gt_sub_fun: App s t  $>_t$  s
  <proof>
```

end

8.4 Conditional Equivalence with Unoptimized Version

```
context rpo
begin
```

context

```
assumes extf_ext_snoc:  $\bigwedge f.$  ext_snoc (extf f)
```

begin

lemma *rpo_optim*: *rpo_optim ground_heads_var (op >_s) extf arity_sym arity_var*
⟨*proof*⟩

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**

op >_{to} ≡ *rpo_optim.gt ground_heads_var (op >_s) extf*

abbreviation *ge_optim* :: (*'s, 'v*) *tm* ⇒ (*'s, 'v*) *tm* ⇒ *bool* (**infix** ≥_{to} 50) **where**

op ≥_{to} ≡ *rpo_optim.ge ground_heads_var (op >_s) extf*

theorem *gt_iff_optim*: *t >_t s* ⟷ *t >_{to} s*

⟨*proof*⟩

end

end

end

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

theory *Lambda_Free_RPOs*

imports *Lambda_Free_RPO_App Lambda_Free_RPO_Optim*

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