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

Jasmin Christian Blanchette, Uwe Waldmann, and Daniel Wand

May 26, 2024

Abstract

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

Contents

1 Introduction

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

We refer to the following conference paper for details:

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

2 Utilities for Lambda-Free Orders

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

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

2.1 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 $(f \,^{\frown\frown}\, m)\ k \leq (f \,^{\frown\frown}\, n)\ k$
⟨*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 $(f \,^{\frown\frown}\, m)\ k < (f \,^{\frown\frown}\, n)\ k$
⟨*proof*⟩

2.2 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*: $P\ 0 \Longrightarrow (LEAST\ x :: enat.\ P\ x) = 0$
⟨*proof*⟩

2.3 Antisymmetric Relations

lemma *irrefl_trans_imp_antisym*: $irrefl\ r \Longrightarrow trans\ r \Longrightarrow antisym\ r$
⟨*proof*⟩

lemma *irreflp_transp_imp_antisymP*: $irreflp\ p \Longrightarrow transp\ p \Longrightarrow antisymp\ p$
⟨*proof*⟩

2.4 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*⟩

2.5 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*⟩

2.6 Well-Founded Relations

lemma *wf_app*: $wf\ r \Longrightarrow wf\ \{(x, y).\ (f\ x,\ f\ y) \in r\}$
⟨*proof*⟩

lemma *wfP_app*: $wfP\ p \Longrightarrow wfP\ (\lambda x\ y.\ p\ (f\ x)\ (f\ y))$
⟨*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* $\Longrightarrow$ *irrefl r* $\Longrightarrow$ *trans r* $\Longrightarrow$ *wf r*
⟨*proof*⟩

lemma *finite_irreflp_transp_imp_wfp*:
finite $\{(x,\ y).\ p\ x\ y\}$ $\Longrightarrow$ *irreflp p* $\Longrightarrow$ *transp p* $\Longrightarrow$ *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*⟩

2.7 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*⟩

2.8 Lists

lemma *rev_induct2*[*consumes 1*, *case_names Nil snoc*]:
length xs = *length ys* $\Longrightarrow$ *P* [] [] $\Longrightarrow$
($\bigwedge$*x xs y ys*. *length xs* = *length ys* $\Longrightarrow$ *P xs ys* $\Longrightarrow$ *P* (*xs* @ [*x*]) (*ys* @ [*y*])) $\Longrightarrow$ *P xs ys*
⟨*proof*⟩

lemma *hd_in_set*: *length xs* $\neq$ *0* $\Longrightarrow$ *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** $\forall i <$ *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_add,semiring_1}*
assumes $\forall i <$ *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 $\bigwedge x.\ x \in$ *set xs* $\Longrightarrow x \geq 0$
shows *prod_list xs* $\geq$ *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*⟩

2.9 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*: $m \le n - 1 \Longrightarrow n \ne \infty \Longrightarrow n \ne 0 \Longrightarrow m < n$ **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*]: $n \le m \Longrightarrow m = m - n + n$ **for** $m\ n :: enat$
⟨*proof*⟩

lemma *enat_the_enat_iden*[*simp*]: $n \ne \infty \Longrightarrow enat\ (the\_enat\ n) = n$
⟨*proof*⟩

lemma *the_enat_minus_nat*: $m \ne \infty \Longrightarrow the\_enat\ (m - enat\ n) = the\_enat\ m - n$
⟨*proof*⟩

lemma *enat_the_enat_le*: $enat\ (the\_enat\ x) \le x$
⟨*proof*⟩

lemma *enat_the_enat_minus_le*: $enat\ (the\_enat\ (x - y)) \le x$
⟨*proof*⟩

lemma *enat_le_imp_minus_le*: $k \le m \Longrightarrow k - n \le m$ **for** $k\ m\ n :: enat$
⟨*proof*⟩

lemma *add_diff_assoc2_enat*: $m \ge n \Longrightarrow m - n + p = m + p - n$ **for** $m\ n\ p :: enat$
⟨*proof*⟩

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

2.10 Multisets

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

end

3 Lambda-Free Higher-Order Terms

theory *Lambda_Free_Term*
imports *Lambda_Free_Util*
abbrevs $>s = >_s$

and $>h = >_{hd}$
and $\leq\geq h = \leq\geq_{hd}$
begin

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

3.1 Precedence on Symbols

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

lemma *gt_sym_antisym*: $f >_s g \Longrightarrow \neg g >_s f$
⟨*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* ⇒ *bool* **where**
is_Sym $\zeta \equiv \neg$ *is_Var* ζ

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 ⟹ 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*]: $syms\ (apps\ s\ ss) = syms\ s \cup (\bigcup s \in set\ ss.\ syms\ s)$ **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*]: $is\_App\ (apps\ s\ ss) \longleftrightarrow args\ (apps\ s\ ss) \neq []$ **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*]: $apps\ (Hd\ \zeta)\ ss = apps\ (Hd\ \xi)\ ts \longleftrightarrow \zeta = \xi \wedge ss = ts$
⟨*proof*⟩

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

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

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

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

lemma *tm_induct_apps*[*case_names apps*]:
assumes $\bigwedge\zeta\ ss.\ (\bigwedge s.\ s \in set\ ss \Longrightarrow P\ s) \Longrightarrow P\ (apps\ (Hd\ \zeta)\ ss)$
shows $P\ s$
⟨*proof*⟩

lemma
ground_fun: $ground\ s \Longrightarrow ground\ (fun\ s)$ **and**
ground_arg: $ground\ s \Longrightarrow ground\ (arg\ s)$
⟨*proof*⟩

lemma *ground_head*: $ground\ s \Longrightarrow is\_Sym\ (head\ s)$
⟨*proof*⟩

lemma *ground_args*: $t \in set\ (args\ s) \Longrightarrow ground\ s \Longrightarrow ground\ t$
⟨*proof*⟩

primrec *vars_mset* :: $('s, 'v)\ tm \Rightarrow 'v\ multiset$ **where**
$vars\_mset\ (Hd\ \zeta) = mset\_set\ (vars\_hd\ \zeta)$
| $vars\_mset\ (App\ s\ t) = vars\_mset\ s + vars\_mset\ t$

lemma *set_vars_mset*[*simp*]: $set\_mset\ (vars\_mset\ t) = vars\ t$
⟨*proof*⟩

lemma *vars_mset_empty_iff*[*iff*]: $vars\_mset\ s = \{\#\} \longleftrightarrow ground\ s$
⟨*proof*⟩

lemma *vars_mset_fun*[*intro*]: $vars\_mset\ (fun\ t) \subseteq\#\ vars\_mset\ t$
⟨*proof*⟩

lemma *vars_mset_arg*[*intro*]: $vars\_mset\ (arg\ t) \subseteq\#\ vars\_mset\ t$
⟨*proof*⟩

3.4 hsize

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

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

lemma *hsize_size*: hsize t * 2 = size t + 1
  ⟨proof⟩

lemma *hsize_pos*[simp]: hsize t > 0
  ⟨proof⟩

lemma *hsize_fun_lt*: is_App s ⟹ hsize (fun s) < hsize s
  ⟨proof⟩

lemma *hsize_arg_lt*: is_App s ⟹ hsize (arg s) < hsize s
  ⟨proof⟩

lemma *hsize_ge_num_args*: hsize s ≥ hsize s
  ⟨proof⟩

lemma *hsize_in_args*: s ∈ set (args t) ⟹ hsize s < hsize t
  ⟨proof⟩

lemma *hsize_apps*: hsize (apps t ts) = hsize t + sum_list (map hsize ts)
  ⟨proof⟩

lemma *hsize_args*: 1 + sum_list (map hsize (args t)) = hsize t
  ⟨proof⟩

3.5 Substitutions

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

lemma *subst_apps*[simp]: subst ϱ (apps s ts) = apps (subst ϱ s) (map (subst ϱ) ts)
  ⟨proof⟩

lemma *head_subst*[simp]: head (subst ϱ s) = head (subst ϱ (Hd (head s)))
  ⟨proof⟩

lemma *args_subst*[simp]:
  args (subst ϱ s) = (case head s of Var x ⇒ args (ϱ x) | Sym f ⇒ []) @ map (subst ϱ) (args s)
  ⟨proof⟩

lemma *ground_imp_subst_iden*: ground s ⟹ subst ϱ s = s
  ⟨proof⟩

lemma *vars_mset_subst*[simp]: vars_mset (subst ϱ s) = (∑# {#vars_mset (ϱ x). x ∈# vars_mset s#})
⟨proof⟩

lemma *vars_mset_subst_subseteq*:
  vars_mset t ⊇# vars_mset s ⟹ vars_mset (subst ϱ t) ⊇# vars_mset (subst ϱ s)
  ⟨proof⟩

lemma *vars_subst_subseteq*: vars t ⊇ vars s ⟹ vars (subst ϱ t) ⊇ vars (subst ϱ s)
  ⟨proof⟩

3.6 Subterms

inductive *sub* :: ('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool **where**

sub_refl: *sub s s*
| *sub_fun*: *sub s t* ⟹ *sub s* (*App u t*)
| *sub_arg*: *sub s t* ⟹ *sub s* (*App t u*)

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

lemma *in_vars_imp_sub*: *x* ∈ *vars s* ⟷ *sub* (*Hd* (*Var x*)) *s*
⟨*proof*⟩

lemma *sub_args*: *s* ∈ *set* (*args t*) ⟹ *sub s t*
⟨*proof*⟩

lemma *sub_size*: *sub s t* ⟹ *size s* ≤ *size t*
⟨*proof*⟩

lemma *sub_subst*: *sub s t* ⟹ *sub* (*subst* ϱ *s*) (*subst* ϱ *t*)
⟨*proof*⟩

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

lemma *proper_sub_Hd*[*simp*]: ¬ *proper_sub s* (*Hd* ζ)
⟨*proof*⟩

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* $\Longrightarrow$ *wary* (*arg t*)
⟨*proof*⟩

lemma *wary_args*: *s* $\in$ *set* (*args t*) $\Longrightarrow$ *wary t* $\Longrightarrow$ *wary s*
⟨*proof*⟩

lemma *wary_sub*: *sub s t* $\Longrightarrow$ *wary t* $\Longrightarrow$ *wary s*
⟨*proof*⟩

lemma *wary_inf_ary*: ($\bigwedge\zeta$. *arity_hd* $\zeta = \infty$) $\Longrightarrow$ *wary s*
⟨*proof*⟩

lemma *wary_num_args_le_arity_head*: *wary s* $\Longrightarrow$ *num_args s* $\leq$ *arity_hd* (*head s*)
⟨*proof*⟩

lemma *wary_apps*:
wary s $\Longrightarrow$ ($\bigwedge$*sa*. *sa* $\in$ *set ss* $\Longrightarrow$ *wary sa*) $\Longrightarrow$ *length ss* $\leq$ *arity s* $\Longrightarrow$ *wary* (*apps s ss*)
⟨*proof*⟩

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

lemma *arity_hd_head*: *wary s* $\Longrightarrow$ *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* $\lor$ *is_Sym* (*head s*) $\Longrightarrow$ *length* (*args s*) = *arity_hd* (*head s*) $\Longrightarrow$
($\forall$ *t* $\in$ *set* (*args s*). *wary_fo t*) $\Longrightarrow$ *wary_fo s*

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

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

end

3.8 Potential Heads of Ground Instances of Variables

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

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

lemma *ground_heads_arity*: $f \in$ *ground_heads* $\zeta \Longrightarrow$ *arity_sym* $f \geq$ *arity_hd* ζ
⟨*proof*⟩

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

lemma *sym_in_ground_heads*: *is_Sym* $\zeta \Longrightarrow$ *sym* $\zeta \in$ *ground_heads* ζ
⟨*proof*⟩

lemma *ground_hd_in_ground_heads*: *ground s* $\Longrightarrow$ *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* (ϱ *x*) $\wedge$ *arity* (ϱ *x*) $\geq$ *arity_var x* $\wedge$ *ground_heads* (*head* (ϱ *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* (ϱ *x*) $\wedge$ *arity* (ϱ *x*) $\in$ {*arity_var x*, ∞}
$\wedge$ *ground_heads* (*head* (ϱ *x*)) $\subseteq$ *ground_heads_var x*)

lemma *strict_imp_wary_subst*: *strict_wary_subst* $\varrho \Longrightarrow$ *wary_subst* ϱ
⟨*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*)) $\subseteq$ *ground_heads* (*head s*)
⟨*proof*⟩

lemmas *strict_wary_subst_ground_heads* = *wary_subst_ground_heads*[*OF strict_imp_wary_subst*]

definition *grounding_*ϱ :: $'v \Rightarrow$ ($'s$, $'v$) *tm* **where**
*grounding_*ϱ *x* = (*let s* = *Hd* (*Sym* (*SOME f. f* $\in$ *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* $\Rightarrow$ ($'s$, $'v$) *hd* $\Rightarrow$ *bool* (**infix** $>_{hd}$ *50*) **where**
$\xi >_{hd} \zeta \longleftrightarrow$ ($\forall g \in$ *ground_heads* ξ. $\forall f \in$ *ground_heads* ζ. $g >_s f$)

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

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

lemma *gt_hd_trans*: $\chi >_{hd} \xi \Longrightarrow \xi >_{hd} \zeta \Longrightarrow \chi >_{hd} \zeta$

⟨*proof*⟩

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

lemma *not_comp_hd_imp_Var*: $\neg\ \xi \le\ge_{hd} \zeta \Longrightarrow is\_Var\ \zeta \lor is\_Var\ \xi$
⟨*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 $p :: 'a \Rightarrow 'a \Rightarrow bool$
begin

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

lemma *wfP_iff_no_inf_chain*: $wfP\ (\lambda x\ y.\ p\ y\ x) \longleftrightarrow (\nexists f.\ inf\_chain\ f)$
⟨*proof*⟩

lemma *inf_chain_offset*: $inf\_chain\ f \Longrightarrow inf\_chain\ (\lambda j.\ f\ (j + i))$
⟨*proof*⟩

definition *bad* :: $'a \Rightarrow bool$ **where**
$bad\ x \longleftrightarrow (\exists f.\ inf\_chain\ f \land f\ 0 = x)$

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**
$worst\_chain\ 0 = (SOME\ x.\ bad\ x \land (\forall y.\ bad\ y \longrightarrow \neg\ gt\ x\ y))$
$|\ worst\_chain\ (Suc\ i) = (SOME\ x.\ bad\ x \land p\ (worst\_chain\ i)\ x \land$
$(\forall y.\ bad\ y \land p\ (worst\_chain\ i)\ y \longrightarrow \neg\ gt\ x\ y))$

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: $\neg\ gt\ (worst\_chain\ 0)\ x$
⟨*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* $\Longrightarrow$ ¬ *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: $\bigwedge z\ y\ x.\ p\ z\ y \Longrightarrow p\ y\ x \Longrightarrow p\ z\ x$
begin

lemma *worst_chain_not_gt*: ¬ *gt* (*worst_chain i*) (*worst_chain* (*Suc i*)) **for** *i*
⟨*proof*⟩

end

end

end

lemma *inf_chain_subset*: *inf_chain p f* $\Longrightarrow p \le q \Longrightarrow$ *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* ⇒ *'a* ⇒ *bool*) ⇒ *'a list* ⇒ *'a list* ⇒ *bool*
assumes
mono_strong: $(\forall y \in set\ ys.\ \forall x \in set\ xs.\ gt\ y\ x \longrightarrow gt'\ y\ x) \Longrightarrow ext\ gt\ ys\ xs \Longrightarrow ext\ gt'\ ys\ xs$ **and**
map: $finite\ A \Longrightarrow ys \in lists\ A \Longrightarrow xs \in lists\ A \Longrightarrow (\forall x \in A.\ \neg\ gt\ (f\ x)\ (f\ x)) \Longrightarrow$
$(\forall z \in A.\ \forall y \in A.\ \forall x \in A.\ gt\ (f\ z)\ (f\ y) \longrightarrow gt\ (f\ y)\ (f\ x) \longrightarrow gt\ (f\ z)\ (f\ x)) \Longrightarrow$
$(\forall y \in A.\ \forall x \in A.\ gt\ y\ x \longrightarrow gt\ (f\ y)\ (f\ x)) \Longrightarrow ext\ gt\ ys\ xs \Longrightarrow ext\ gt\ (map\ f\ ys)\ (map\ f\ xs)$
begin

lemma *mono*[*mono*]: $gt \le gt' \Longrightarrow ext\ gt \le ext\ gt'$
⟨*proof*⟩

end

locale *ext_irrefl* = *ext* +

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

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

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

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

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

sublocale *ext_irrefl_trans_strong* < *ext_irrefl_before_trans*
⟨*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*: $ext\ gt\ ys\ xs \Longrightarrow ext\ gt\ (x\ \#\ ys)\ (x\ \#\ xs)$
begin

lemma *compat_append_left*: $ext\ gt\ ys\ xs \Longrightarrow ext\ gt\ (zs\ @\ ys)\ (zs\ @\ xs)$
⟨*proof*⟩

end

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

lemma *compat_append_right*: $ext\ gt\ ys\ xs \Longrightarrow ext\ gt\ (ys\ @\ zs)\ (xs\ @\ zs)$
⟨*proof*⟩

end

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

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

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

lemma *compat_list*: $y \neq x \Longrightarrow gt\ y\ x \Longrightarrow ext\ gt\ (xs\ @\ y\ \#\ xs')\ (xs\ @\ x\ \#\ xs')$
⟨*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: ¬ ext gt xs xs
⟨proof⟩

lemma trans_gt: ext gt zs ys ⟹ ext gt ys xs ⟹ ext gt zs xs
⟨proof⟩

lemma hd_or_tl_gt: length ys = length xs ⟹ ext gt (y # ys) (x # xs) ⟹ gt y x ∨ ext gt ys xs
⟨proof⟩

lemma wf_same_length_if_total: wfP (λxs ys. length ys = n ∧ length xs = n ∧ ext gt ys xs)
⟨proof⟩

lemma wf_bounded_if_total: wfP (λxs ys. length ys ≤ n ∧ length xs ≤ n ∧ ext gt ys xs)
⟨proof⟩

end

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

lemma wf_bounded: wfP (λxs ys. length ys ≤ n ∧ length xs ≤ n ∧ ext gt ys xs)
⟨proof⟩

end

end

5.2 Lexicographic Extension

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

lemma *lexext_simps*[*simp*]:
lexext gt ys [] ⟷ *ys* ≠ []
¬ *lexext gt* [] *xs*
lexext gt (*y* # *ys*) (*x* # *xs*) ⟷ *gt y x* ∨ *x* = *y* ∧ *lexext gt ys xs*
⟨*proof*⟩

lemma *lexext_mono_strong*:
assumes
∀ *y* ∈ *set ys*. ∀ *x* ∈ *set xs*. *gt y x* ⟶ *gt′ y x* **and**
lexext gt ys xs
shows *lexext gt′ ys xs*
⟨*proof*⟩

lemma *lexext_map_strong*:
(∀ *y* ∈ *set ys*. ∀ *x* ∈ *set xs*. *gt y x* ⟶ *gt* (*f y*) (*f x*)) ⟹ *lexext gt ys xs* ⟹
lexext gt (*map f ys*) (*map f xs*)
⟨*proof*⟩

lemma *lexext_irrefl*:
assumes ∀ *x* ∈ *set xs*. ¬ *gt x x*
shows ¬ *lexext gt xs xs*
⟨*proof*⟩

lemma *lexext_trans_strong*:
assumes
∀ *z* ∈ *set zs*. ∀ *y* ∈ *set ys*. ∀ *x* ∈ *set xs*. *gt z y* ⟶ *gt y x* ⟶ *gt z x* **and**
lexext gt zs ys **and** *lexext gt ys xs*
shows *lexext gt zs xs*
⟨*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* ⟹ *lexext gt* (*xs* @ *y* # *xs′*) (*xs* @ *x* # *xs′*)
⟨*proof*⟩

lemma *lexext_singleton*: *lexext gt* [*y*] [*x*] ⟷ *gt y x*
⟨*proof*⟩

lemma *lexext_total*: (∀ *y* ∈ *B*. ∀ *x* ∈ *A*. *gt y x* ∨ *gt x y* ∨ *y* = *x*) ⟹ *ys* ∈ *lists B* ⟹ *xs* ∈ *lists A* ⟹
lexext gt ys xs ∨ *lexext gt xs ys* ∨ *ys* = *xs*
⟨*proof*⟩

lemma *lexext_hd_or_tl*: *lexext gt* (*y* # *ys*) (*x* # *xs*) ⟹ *gt y x* ∨ *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*: $gt\ y\ x \Longrightarrow lexext\_rev\ gt\ (xs\ @\ y\ \#\ xs')\ (xs\ @\ x\ \#\ xs')$
⟨*proof*⟩

lemma *lexext_rev_singleton*: $lexext\_rev\ gt\ [y]\ [x] \longleftrightarrow gt\ y\ x$
⟨*proof*⟩

lemma *lexext_rev_total*:
$(\forall y \in B.\ \forall x \in A.\ gt\ y\ x \lor gt\ x\ y \lor y = x) \Longrightarrow ys \in lists\ B \Longrightarrow xs \in lists\ A \Longrightarrow$
$lexext\_rev\ gt\ ys\ xs \lor lexext\_rev\ gt\ xs\ ys \lor ys = xs$
⟨*proof*⟩

lemma *lexext_rev_hd_or_tl*:
assumes
$length\ ys = length\ xs$ **and**
$lexext\_rev\ gt\ (y\ \#\ ys)\ (x\ \#\ xs)$
shows $gt\ y\ x \lor lexext\_rev\ gt\ ys\ xs$
⟨*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**
$lenext\ gts\ ys\ xs \longleftrightarrow length\ ys > length\ xs \lor length\ ys = length\ xs \land gts\ ys\ xs$

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

$lenext\_total$: $(gts\ ys\ xs \lor gts\ xs\ ys \lor ys = xs) \Longrightarrow$
$lenext\ gts\ ys\ xs \lor lenext\ gts\ xs\ ys \lor ys = xs$ **and**
$lenext\_hd\_or\_tl$: $(length\ ys = length\ xs \Longrightarrow gts\ (y \# ys)\ (x \# xs) \Longrightarrow gt\ y\ x \lor gts\ ys\ xs) \Longrightarrow$
$lenext\ gts\ (y \# ys)\ (x \# xs) \Longrightarrow gt\ y\ x \lor lenext\ gts\ ys\ xs$
⟨*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 $len\_lexext\_mono\_strong$:
$(\forall y \in set\ ys.\ \forall x \in set\ xs.\ gt\ y\ x \longrightarrow gt'\ y\ x) \Longrightarrow len\_lexext\ gt\ ys\ xs \Longrightarrow len\_lexext\ gt'\ ys\ xs$
⟨*proof*⟩

lemma $len\_lexext\_map\_strong$:
$(\forall y \in set\ ys.\ \forall x \in set\ xs.\ gt\ y\ x \longrightarrow gt\ (f\ y)\ (f\ x)) \Longrightarrow len\_lexext\ gt\ ys\ xs \Longrightarrow$
$len\_lexext\ gt\ (map\ f\ ys)\ (map\ f\ xs)$
⟨*proof*⟩

lemma len_lexext_irrefl: $(\forall x \in set\ xs.\ \neg\ gt\ x\ x) \Longrightarrow \neg\ len\_lexext\ gt\ xs\ xs$
⟨*proof*⟩

lemma $len\_lexext\_trans\_strong$:
$(\forall z \in set\ zs.\ \forall y \in set\ ys.\ \forall x \in set\ xs.\ gt\ z\ y \longrightarrow gt\ y\ x \longrightarrow gt\ z\ x) \Longrightarrow len\_lexext\ gt\ zs\ ys \Longrightarrow$
$len\_lexext\ gt\ ys\ xs \Longrightarrow len\_lexext\ gt\ zs\ xs$
⟨*proof*⟩

lemma len_lexext_snoc: $len\_lexext\ gt\ (xs\ @\ [x])\ xs$
⟨*proof*⟩

lemma $len\_lexext\_compat\_cons$: $len\_lexext\ gt\ ys\ xs \Longrightarrow len\_lexext\ gt\ (x \# ys)\ (x \# xs)$
⟨*proof*⟩

lemma $len\_lexext\_compat\_snoc$: $len\_lexext\ gt\ ys\ xs \Longrightarrow len\_lexext\ gt\ (ys\ @\ [x])\ (xs\ @\ [x])$
⟨*proof*⟩

lemma $len\_lexext\_compat\_list$: $gt\ y\ x \Longrightarrow len\_lexext\ gt\ (xs\ @\ y \# xs')\ (xs\ @\ x \# xs')$
⟨*proof*⟩

lemma $len\_lexext\_singleton[simp]$: $len\_lexext\ gt\ [y]\ [x] \longleftrightarrow gt\ y\ x$
⟨*proof*⟩

lemma len_lexext_total: $(\forall y \in B.\ \forall x \in A.\ gt\ y\ x \lor gt\ x\ y \lor y = x) \Longrightarrow ys \in lists\ B \Longrightarrow xs \in lists\ A \Longrightarrow$
$len\_lexext\ gt\ ys\ xs \lor len\_lexext\ gt\ xs\ ys \lor ys = xs$
⟨*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: $wfP\ (\lambda x\ y.\ gt\ y\ x) \Longrightarrow wfP\ (\lambda xs\ ys.\ len\_lexext\ gt\ ys\ xs)$
⟨*proof*⟩

lemma $len\_lexext\_hd\_or\_tl$: $len\_lexext\ gt\ (y \# ys)\ (x \# xs) \Longrightarrow gt\ y\ x \lor len\_lexext\ gt\ ys\ xs$
⟨*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*
⟨*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* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow bool$ **where**
len_lexext_rev gt ≡ *lenext* (*lexext_rev gt*)

lemma *len_lexext_rev_mono_strong*:
$(\forall y \in set\ ys.\ \forall x \in set\ xs.\ gt\ y\ x \longrightarrow gt'\ y\ x) \Longrightarrow len\_lexext\_rev\ gt\ ys\ xs \Longrightarrow len\_lexext\_rev\ gt'\ ys\ xs$
⟨*proof*⟩

lemma *len_lexext_rev_map_strong*:
$(\forall y \in set\ ys.\ \forall x \in set\ xs.\ gt\ y\ x \longrightarrow gt\ (f\ y)\ (f\ x)) \Longrightarrow len\_lexext\_rev\ gt\ ys\ xs \Longrightarrow$
$len\_lexext\_rev\ gt\ (map\ f\ ys)\ (map\ f\ xs)$
⟨*proof*⟩

lemma *len_lexext_rev_irrefl*: $(\forall x \in set\ xs.\ \neg\ gt\ x\ x) \Longrightarrow \neg\ len\_lexext\_rev\ gt\ xs\ xs$
⟨*proof*⟩

lemma *len_lexext_rev_trans_strong*:
$(\forall z \in set\ zs.\ \forall y \in set\ ys.\ \forall x \in set\ xs.\ gt\ z\ y \longrightarrow gt\ y\ x \longrightarrow gt\ z\ x) \Longrightarrow len\_lexext\_rev\ gt\ zs\ ys \Longrightarrow$
$len\_lexext\_rev\ gt\ ys\ xs \Longrightarrow len\_lexext\_rev\ gt\ zs\ xs$
⟨*proof*⟩

lemma *len_lexext_rev_snoc*: $len\_lexext\_rev\ gt\ (xs\ @\ [x])\ xs$
⟨*proof*⟩

lemma *len_lexext_rev_compat_cons*: $len\_lexext\_rev\ gt\ ys\ xs \Longrightarrow len\_lexext\_rev\ gt\ (x\ \#\ ys)\ (x\ \#\ xs)$
⟨*proof*⟩

lemma *len_lexext_rev_compat_snoc*: $len\_lexext\_rev\ gt\ ys\ xs \Longrightarrow len\_lexext\_rev\ gt\ (ys\ @\ [x])\ (xs\ @\ [x])$
⟨*proof*⟩

lemma *len_lexext_rev_compat_list*: $gt\ y\ x \Longrightarrow len\_lexext\_rev\ gt\ (xs\ @\ y\ \#\ xs')\ (xs\ @\ x\ \#\ xs')$
⟨*proof*⟩

lemma *len_lexext_rev_singleton*[*simp*]: $len\_lexext\_rev\ gt\ [y]\ [x] \longleftrightarrow gt\ y\ x$
⟨*proof*⟩

lemma *len_lexext_rev_total*: $(\forall y \in B.\ \forall x \in A.\ gt\ y\ x \lor gt\ x\ y \lor y = x) \Longrightarrow ys \in lists\ B \Longrightarrow$
$xs \in lists\ A \Longrightarrow len\_lexext\_rev\ gt\ ys\ xs \lor len\_lexext\_rev\ gt\ xs\ ys \lor ys = xs$

⟨*proof*⟩

lemma *len_lexext_rev_iff_len_lexext*: *len_lexext_rev gt ys xs* ⟷ *len_lexext gt* (*rev ys*) (*rev xs*)
⟨*proof*⟩

lemma *len_lexext_rev_wf*: *wfP* (λ*x y*. *gt y x*) ⟹ *wfP* (λ*xs ys*. *len_lexext_rev gt ys xs*)
⟨*proof*⟩

lemma *len_lexext_rev_hd_or_tl*:
len_lexext_rev gt (*y* # *ys*) (*x* # *xs*) ⟹ *gt y x* ∨ *len_lexext_rev gt ys xs*
⟨*proof*⟩

interpretation *len_lexext_rev*: *ext len_lexext_rev*
⟨*proof*⟩

interpretation *len_lexext_rev*: *ext_irrefl_trans_strong len_lexext_rev*
⟨*proof*⟩

interpretation *len_lexext_rev*: *ext_snoc len_lexext_rev*
⟨*proof*⟩

interpretation *len_lexext_rev*: *ext_compat_cons len_lexext_rev*
⟨*proof*⟩

interpretation *len_lexext_rev*: *ext_compat_snoc len_lexext_rev*
⟨*proof*⟩

interpretation *len_lexext_rev*: *ext_compat_list len_lexext_rev*
⟨*proof*⟩

interpretation *len_lexext_rev*: *ext_singleton len_lexext_rev*
⟨*proof*⟩

interpretation *len_lexext_rev*: *ext_total len_lexext_rev*
⟨*proof*⟩

interpretation *len_lexext_rev*: *ext_wf len_lexext_rev*
⟨*proof*⟩

interpretation *len_lexext_rev*: *ext_hd_or_tl len_lexext_rev*
⟨*proof*⟩

interpretation *len_lexext_rev*: *ext_wf_bounded len_lexext_rev*
⟨*proof*⟩

5.7 Dershowitz–Manna Multiset Extension

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

The following proof is based on that of $less\_multiset_{DM}\_imp\_mult$.

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

lemma *msetext_dersh_imp_mult*: *msetext_dersh gt ys xs* ⟹ (*mset xs*, *mset ys*) ∈ *mult* {(*x*, *y*). *gt y x*}
⟨*proof*⟩

lemma *mult_imp_msetext_dersh_rel*:
assumes

ys_a: *set_mset* (*mset ys*) ⊆ *A* **and** *xs_a*: *set_mset* (*mset xs*) ⊆ *A* **and**
in_mult: (*mset xs*, *mset ys*) ∈ *mult* {(*x*, *y*). *x* ∈ *A* ∧ *y* ∈ *A* ∧ *gt y x*} **and**
trans: ∀ *z* ∈ *A*. ∀ *y* ∈ *A*. ∀ *x* ∈ *A*. *gt z y* ⟶ *gt y x* ⟶ *gt z x*
shows *msetext_dersh gt ys xs*
⟨*proof*⟩

lemma *msetext_dersh_mono_strong*:
(∀ *y* ∈ *set ys*. ∀ *x* ∈ *set xs*. *gt y x* ⟶ *gt′ y x*) ⟹ *msetext_dersh gt ys xs* ⟹
msetext_dersh gt′ ys xs
⟨*proof*⟩

lemma *msetext_dersh_map_strong*:
assumes
compat_f: ∀ *y* ∈ *set ys*. ∀ *x* ∈ *set xs*. *gt y x* ⟶ *gt* (*f y*) (*f x*) **and**
ys_gt_xs: *msetext_dersh gt ys xs*
shows *msetext_dersh gt* (*map f ys*) (*map f xs*)
⟨*proof*⟩

lemma *msetext_dersh_trans*:
assumes
zs_a: *zs* ∈ *lists A* **and**
ys_a: *ys* ∈ *lists A* **and**
xs_a: *xs* ∈ *lists A* **and**
trans: ∀ *z* ∈ *A*. ∀ *y* ∈ *A*. ∀ *x* ∈ *A*. *gt z y* ⟶ *gt y x* ⟶ *gt z x* **and**
zs_gt_ys: *msetext_dersh gt zs ys* **and**
ys_gt_xs: *msetext_dersh gt ys xs*
shows *msetext_dersh gt zs xs*
⟨*proof*⟩

lemma *msetext_dersh_irrefl_from_trans*:
assumes
trans: ∀ *z* ∈ *set xs*. ∀ *y* ∈ *set xs*. ∀ *x* ∈ *set xs*. *gt z y* ⟶ *gt y x* ⟶ *gt z x* **and**
irrefl: ∀ *x* ∈ *set xs*. ¬ *gt x x*
shows ¬ *msetext_dersh gt xs xs*
⟨*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* ⟹ *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*] ⟷ *gt y x*
⟨*proof*⟩

lemma *msetext_dersh_wf*:
assumes *wf_gt*: *wfP* (λ*x y*. *gt y x*)
shows *wfP* (λ*xs ys*. *msetext_dersh gt ys xs*)
⟨*proof*⟩

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*
$M \neq N \wedge (\forall x.\ count\ M\ x > count\ N\ x \longrightarrow (\exists y.\ gt\ y\ x \wedge count\ N\ y > count\ M\ y)))$

lemma *msetext_huet_imp_count_gt*:
assumes *ys_gt_xs*: *msetext_huet gt ys xs*
shows $\exists x.\ count\ (mset\ ys)\ x > count\ (mset\ xs)\ x$
⟨*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: $(mset\ xs,\ mset\ ys) \in mult\ \{(x,\ y).\ gt\ y\ x\}$
shows *msetext_huet gt ys xs*
⟨*proof*⟩

theorem *msetext_huet_eq_dersh*: $irreflp\ gt \Longrightarrow transp\ gt \Longrightarrow msetext\_dersh\ gt = msetext\_huet\ gt$
⟨*proof*⟩

lemma *msetext_huet_mono_strong*:
$(\forall y \in set\ ys.\ \forall x \in set\ xs.\ gt\ y\ x \longrightarrow gt'\ y\ x) \Longrightarrow msetext\_huet\ gt\ ys\ xs \Longrightarrow msetext\_huet\ gt'\ ys\ xs$
⟨*proof*⟩

lemma *msetext_huet_map*:
assumes
fin: *finite A* **and**
ys_a: $ys \in lists\ A$ **and** *xs_a*: $xs \in lists\ A$ **and**
irrefl_f: $\forall x \in A.\ \neg\ gt\ (f\ x)\ (f\ x)$ **and**
trans_f: $\forall z \in A.\ \forall y \in A.\ \forall x \in A.\ gt\ (f\ z)\ (f\ y) \longrightarrow gt\ (f\ y)\ (f\ x) \longrightarrow gt\ (f\ z)\ (f\ x)$ **and**
compat_f: $\forall y \in A.\ \forall x \in A.\ gt\ y\ x \longrightarrow gt\ (f\ y)\ (f\ x)$ **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*: $(\forall x \in set\ xs.\ \neg\ gt\ x\ x) \Longrightarrow \neg\ msetext\_huet\ gt\ xs\ xs$

⟨*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: ∀ *x* ∈ *A*. ¬ *gt x x* **and**
trans: ∀ *z* ∈ *A*. ∀ *y* ∈ *A*. ∀ *x* ∈ *A*. *gt z y* ⟶ *gt y x* ⟶ *gt z x* **and**
zs_gt_ys: *msetext_huet gt zs ys* **and**
ys_gt_xs: *msetext_huet gt ys xs*
shows *msetext_huet gt zs xs*
⟨*proof*⟩

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

lemma *msetext_huet_compat_cons*: *msetext_huet gt ys xs* ⟹ *msetext_huet gt* (*x* # *ys*) (*x* # *xs*)
⟨*proof*⟩

lemma *msetext_huet_compat_snoc*: *msetext_huet gt ys xs* ⟹ *msetext_huet gt* (*ys* @ [*x*]) (*xs* @ [*x*])
⟨*proof*⟩

lemma *msetext_huet_compat_list*: *y* ≠ *x* ⟹ *gt y x* ⟹ *msetext_huet gt* (*xs* @ *y* # *xs'*) (*xs* @ *x* # *xs'*)
⟨*proof*⟩

lemma *msetext_huet_singleton*: *y* ≠ *x* ⟹ *msetext_huet gt* [*y*] [*x*] ⟷ *gt y x*
⟨*proof*⟩

lemma *msetext_huet_wf*: *wfP* (λ*x y*. *gt y x*) ⟹ *wfP* (λ*xs ys*. *msetext_huet gt ys xs*)
⟨*proof*⟩

lemma *msetext_huet_hd_or_tl*:
assumes
trans: ∀ *z y x*. *gt z y* ⟶ *gt y x* ⟶ *gt z x* **and**
total: ∀ *y x*. *gt y x* ∨ *gt x y* ∨ *y* = *x* **and**
len_eq: *length ys* = *length xs* **and**
yys_gt_xxs: *msetext_huet gt* (*y* # *ys*) (*x* # *xs*)
shows *gt y x* ∨ *msetext_huet gt ys xs*
⟨*proof*⟩

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

lemma *cwiseext_imp_len_lexext*:
assumes *cw*: *cwiseext gt ys xs*
shows *len_lexext gt ys xs*
⟨*proof*⟩

lemma *cwiseext_mono_strong*:
$(\forall y \in set\ ys.\ \forall x \in set\ xs.\ gt\ y\ x \longrightarrow gt'\ y\ x) \Longrightarrow cwiseext\ gt\ ys\ xs \Longrightarrow cwiseext\ gt'\ ys\ xs$
⟨*proof*⟩

lemma *cwiseext_map_strong*:
$(\forall y \in set\ ys.\ \forall x \in set\ xs.\ gt\ y\ x \longrightarrow gt\ (f\ y)\ (f\ x)) \Longrightarrow cwiseext\ gt\ ys\ xs \Longrightarrow$
$cwiseext\ gt\ (map\ f\ ys)\ (map\ f\ xs)$
⟨*proof*⟩

lemma *cwiseext_irrefl*: $(\forall x \in set\ xs.\ \neg\ gt\ x\ x) \Longrightarrow \neg\ cwiseext\ gt\ xs\ xs$
⟨*proof*⟩

lemma *cwiseext_trans_strong*:
assumes
$\forall z \in set\ zs.\ \forall y \in set\ ys.\ \forall x \in set\ xs.\ gt\ z\ y \longrightarrow gt\ y\ x \longrightarrow gt\ z\ x$ **and**
cwiseext gt zs ys **and** *cwiseext gt ys xs*
shows *cwiseext gt zs xs*
⟨*proof*⟩

lemma *cwiseext_compat_cons*: $cwiseext\ gt\ ys\ xs \Longrightarrow cwiseext\ gt\ (x\ \#\ ys)\ (x\ \#\ xs)$
⟨*proof*⟩

lemma *cwiseext_compat_snoc*: $cwiseext\ gt\ ys\ xs \Longrightarrow cwiseext\ gt\ (ys\ @\ [x])\ (xs\ @\ [x])$
⟨*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*: $cwiseext\ gt\ [y]\ [x] \longleftrightarrow gt\ y\ x$
⟨*proof*⟩

lemma *cwiseext_wf*: $wfP\ (\lambda x\ y.\ gt\ y\ x) \Longrightarrow wfP\ (\lambda xs\ ys.\ cwiseext\ gt\ ys\ xs)$
⟨*proof*⟩

lemma *cwiseext_hd_or_tl*: $cwiseext\ gt\ (y\ \#\ ys)\ (x\ \#\ xs) \Longrightarrow gt\ y\ x \vee cwiseext\ gt\ ys\ xs$
⟨*proof*⟩

locale *ext_cwiseext* = *ext_compat_list* + *ext_compat_cons*
begin

context
fixes *gt* :: $'a \Rightarrow 'a \Rightarrow bool$
assumes

gt_irrefl: ¬ *gt x x* **and**
trans_gt: *ext gt zs ys* ⟹ *ext gt ys xs* ⟹ *ext gt zs xs*
begin

lemma
assumes *ys_gtcw_xs*: *cwiseext gt ys xs*
shows *ext gt ys xs*
⟨*proof*⟩

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 $>t = >_t$
and $\geq t = \geq_t$
begin

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

locale *rpo_app* = *gt_sym* ($>_s$)
for *gt_sym* :: *'s* ⇒ *'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*]: $gt \le gt' \Longrightarrow ext\ gt \le ext\ gt'$
⟨*proof*⟩

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

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

end

end

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

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

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

7.1 Setup

locale *rpo_basis* = *ground_heads* $(>_s)$ *arity_sym arity_var*
for
gt_sym :: $'s \Rightarrow 's \Rightarrow bool$ (**infix** $>_s$ *50*) **and**
arity_sym :: $'s \Rightarrow enat$ **and**
arity_var :: $'v \Rightarrow enat$ +
fixes
extf :: $'s \Rightarrow (('s, 'v)\ tm \Rightarrow ('s, 'v)\ tm \Rightarrow bool) \Rightarrow ('s, 'v)\ tm\ list \Rightarrow ('s, 'v)\ tm\ list \Rightarrow bool$
assumes
extf_ext_trans_before_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*)
⟨*proof*⟩

lemma *extf_ext*: *ext* (*extf f*)
⟨*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* :: $('s, 'v)\ tm \Rightarrow ('s, 'v)\ tm \Rightarrow bool$ **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* ⟷ (*case s of App s1 s2* ⇒ *gt t s1* ∧ *gt t s2* | _ ⇒ *True*)

lemma *chksubs_mono*[*mono*]: $gt \le gt' \Longrightarrow chksubs\ gt \le chksubs\ gt'$
⟨*proof*⟩

inductive *gt* :: (*'s*, *'v*) *tm* ⇒ (*'s*, *'v*) *tm* ⇒ *bool* (**infix** $>_t$ *50*) **where**
gt_sub: $is\_App\ t \Longrightarrow (fun\ t >_t s \lor fun\ t = s) \lor (arg\ t >_t s \lor arg\ t = s) \Longrightarrow t >_t s$
| *gt_diff*: $head\ t >_{hd} head\ s \Longrightarrow chkvar\ t\ s \Longrightarrow chksubs\ (>_t)\ t\ s \Longrightarrow t >_t s$
| *gt_same*: $head\ t = head\ s \Longrightarrow chksubs\ (>_t)\ t\ s \Longrightarrow$
$(\forall f \in ground\_heads\ (head\ t).\ extf\ f\ (>_t)\ (args\ t)\ (args\ s)) \Longrightarrow t >_t s$

abbreviation *ge* :: (*'s*, *'v*) *tm* ⇒ (*'s*, *'v*) *tm* ⇒ *bool* (**infix** $\ge_t$ *50*) **where**
$t \ge_t s \equiv t >_t s \lor t = s$

inductive *gt_sub* :: (*'s*, *'v*) *tm* ⇒ (*'s*, *'v*) *tm* ⇒ *bool* **where**
gt_subI: $is\_App\ t \Longrightarrow fun\ t \ge_t s \lor arg\ t \ge_t s \Longrightarrow gt\_sub\ t\ s$

inductive *gt_diff* :: (*'s*, *'v*) *tm* ⇒ (*'s*, *'v*) *tm* ⇒ *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* ⇒ (*'s*, *'v*) *tm* ⇒ *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*: $t >_t s \longleftrightarrow gt\_sub\ t\ s \lor gt\_diff\ t\ s \lor gt\_same\ t\ s$
⟨*proof*⟩

7.3 Transitivity

lemma *gt_fun_imp*: $fun\ t >_t s \Longrightarrow t >_t s$
⟨*proof*⟩

lemma *gt_arg_imp*: $arg\ t >_t s \Longrightarrow t >_t s$
⟨*proof*⟩

lemma *gt_imp_vars*: $t >_t s \Longrightarrow vars\ t \supseteq vars\ s$
⟨*proof*⟩

theorem *gt_trans*: $u >_t t \Longrightarrow t >_t s \Longrightarrow u >_t s$
⟨*proof*⟩

7.4 Irreflexivity

theorem *gt_irrefl*: $\neg\ s >_t s$
⟨*proof*⟩

lemma *gt_antisym*: $t >_t s \Longrightarrow \neg\ s >_t t$
⟨*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: $proper\_sub\ s\ t \Longrightarrow t >_t s$
⟨*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**
$diff\_same$: $gt\_diff\ s'\ s \lor gt\_same\ s'\ s$
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 $chksubs\ (>_t)\ t\ s$
⟨*proof*⟩

theorem gt_subst:
assumes $wary\_\varrho$: $wary\_subst\ \varrho$
shows $t >_t s \Longrightarrow subst\ \varrho\ t >_t subst\ \varrho\ s$
⟨*proof*⟩

7.9 Totality on Ground Terms

theorem gt_total_ground:
assumes $extf\_total$: $\bigwedge f.\ ext\_total\ (extf\ f)$
shows $ground\ t \Longrightarrow ground\ s \Longrightarrow t >_t s \lor s >_t t \lor t = s$
⟨*proof*⟩

7.10 Well-foundedness

abbreviation $gtg :: ('s, 'v)\ tm \Rightarrow ('s, 'v)\ tm \Rightarrow bool$ (**infix** $>_{tg}$ *50*) **where**
$(>_{tg}) \equiv \lambda t\ s.\ ground\ t \land t >_t s$

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* ⇒ *enat* **and**
    arity_var :: *'v* ⇒ *enat* +
  assumes *extf_ext_snoc*: *ext_snoc* (*extf f*)
begin

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

8.2 Definition of the Order

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

lemma *chkargs_mono*[*mono*]: *gt* ≤ *gt'* ⟹ *chkargs gt* ≤ *chkargs gt'*
  ⟨*proof*⟩

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

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

8.3 Transitivity

lemma *gt_in_args_imp*: *ti* ∈ *set* (*args t*) ⟹ *ti* $>_t$ *s* ⟹ *t* $>_t$ *s*
  ⟨*proof*⟩

lemma *gt_imp_vars*: *t* $>_t$ *s* ⟹ *vars t* ⊇ *vars s*
⟨*proof*⟩

lemma *gt_trans*: *u* $>_t$ *t* ⟹ *t* $>_t$ *s* ⟹ *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*: ⋀*f*. *ext_snoc* (*extf f*)

begin

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

abbreviation
chkargs :: $(('s, 'v)\ tm \Rightarrow ('s, 'v)\ tm \Rightarrow bool) \Rightarrow ('s, 'v)\ tm \Rightarrow ('s, 'v)\ tm \Rightarrow bool$
where
chkargs ≡ *rpo_optim.chkargs*

abbreviation *gt_optim* :: $('s, 'v)\ tm \Rightarrow ('s, 'v)\ tm \Rightarrow bool$ (**infix** $>_{to}$ *50*) **where**
$(>_{to})$ ≡ *rpo_optim.gt ground_heads_var* $(>_s)$ *extf*

abbreviation *ge_optim* :: $('s, 'v)\ tm \Rightarrow ('s, 'v)\ tm \Rightarrow bool$ (**infix** $\geq_{to}$ *50*) **where**
$(\geq_{to})$ ≡ *rpo_optim.ge ground_heads_var* $(>_s)$ *extf*

theorem *gt_iff_optim*: $t >_t s \longleftrightarrow t >_{to} s$
⟨*proof*⟩

end

end

end

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

theory *Lambda_Encoding*
imports *Lambda_Free_Term*
begin

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

locale *lambda_encoding* =
fixes
lam :: *'s* **and**
db :: *nat* ⇒ *'s*
begin

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

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

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

lemma *vars_mset_subst_db*: *vars_mset* (*subst_db i x s*) = {#*y* ∈# *vars_mset s*. *y* ≠ *x*#}
⟨*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