Formalization of the Embedding Path Order for Lambda-Free Higher-Order Terms

Alexander Bentkamp

March 17, 2025

Abstract

This Isabelle/HOL formalization defines the Embedding Path Order (EPO) for higher-order terms without λ-abstraction and proves many useful properties about it. In contrast to the lambda-free recursive path orders, it does not fully coincide with RPO on first-order terms, but it is compatible with arbitrary higher-order contexts.

Contents

1 Introduction

This Isabelle/HOL formalization defines the Embedding Path Order (EPO) for higher-order terms without λ-abstraction and proves many useful properties about it. In contrast to the lambda-free recursive path orders, it does not fully coincide with RPO on first-order terms, but it is compatible with arbitrary higher-order contexts.

2 The Embedding Relation for Lambda-Free Higher-Order Terms

theory *Embeddings*
  imports *Lambda_Free_RPOs.Lambda_Free_Term Lambda_Free_RPOs.Extension_Orders*
begin

2.1 Positions of terms

datatype *dir* = *Left* | *Right*

fun *position_of* :: $('s,'v)$ *tm* ⇒ *dir list* ⇒ *bool* **where**
  position_of_Nil: *position_of* _ [] = *True* |
  position_of_Hd: *position_of* (*Hd* _) (_#_) = *False* |
  position_of_left: *position_of* (*App t s*) (*Left* # *ds*) = *position_of t ds* |
  position_of_right: *position_of* (*App t s*) (*Right* # *ds*) = *position_of s ds*

definition *opp* :: *dir* ⇒ *dir* **where**
  opp d = (*if d* = *Right then Left else Right*)

lemma *opp_simps*[*simp*]:
  opp Right = *Left*
  opp Left = *Right*
  ⟨*proof*⟩

lemma *shallower_pos*: *position_of t* (*p* @ *q* @ [*dq*]) ⟹ *position_of t* (*p* @ [*dp*])
⟨*proof*⟩

lemma *no_position_replicate_num_args*: ¬ *position_of t* (*replicate* (*num_args t*) *Left* @ [*d*])
⟨*proof*⟩

lemma *shorten_position*:*position_of t* (*p* @ *q*) ⟹ *position_of t p*
⟨*proof*⟩

2.2 Embedding step

Embedding step at a given position. If the position is not present, default to identity.

fun *emb_step_at* :: *dir list* ⇒ *dir* ⇒ $('s, 'v)$ *tm* ⇒ $('s, 'v)$ *tm* **where**
  emb_step_at_left:*emb_step_at* [] *Left* (*App t s*) = *t*
| *emb_step_at_right*:*emb_step_at* [] *Right* (*App t s*) = *s*
| *emb_step_at_left_context*:*emb_step_at* (*Left* # *p*) *dir* (*App t s*) = *App* (*emb_step_at p dir t*) *s*
| *emb_step_at_right_context*:*emb_step_at* (*Right* # *p*) *dir* (*App t s*) = *App t* (*emb_step_at p dir s*)
| *emb_step_at_head*:*emb_step_at* _ _ (*Hd h*) = *Hd h*

abbreviation *emb_step_at'* *p t* == *emb_step_at* (*butlast p*) (*last p*) *t*

lemmas *emb_step_at_induct* = *emb_step_at.induct*[*case_names left right left_context right_context head*]

lemma *emb_step_at_is_App*:*emb_step_at p d u* ≠ *u* ⟹ *is_App u*
  ⟨*proof*⟩

Definition of an embedding step without using positions.

inductive *emb_step* (**infix** ‹$\to_{emb}$› *50*) **where**
  left: (*App t1 t2*) $\to_{emb}$ *t1* |
  right: (*App t1 t2*) $\to_{emb}$ *t2* |
  context_left: *t* $\to_{emb}$ *s* ⟹ (*App t u*) $\to_{emb}$ (*App s u*) |
  context_right: *t* $\to_{emb}$ *s* ⟹ (*App u t*) $\to_{emb}$ (*App u s*)

The two definitions of an embedding step are equivalent:

lemma *emb_step_equiv*: *emb_step t s* ⟷ (∃ *p d*. *emb_step_at p d t* = *s*) ∧ *t* ≠ *s*
⟨*proof*⟩

lemma *emb_step_fun*: *is_App t* ⟹ *t* $\to_{emb}$ (*fun t*)
  ⟨*proof*⟩

lemma *emb_step_arg*: $is\_App\ t \Longrightarrow t \rightarrow_{emb} (arg\ t)$
⟨*proof*⟩

lemma *emb_step_hsize*: $t \rightarrow_{emb} s \Longrightarrow hsize\ t > hsize\ s$
⟨*proof*⟩

lemma *emb_step_vars*: $t \rightarrow_{emb} s \Longrightarrow vars\ s \subseteq vars\ t$
⟨*proof*⟩

lemma *emb_step_equiv'*: $emb\_step\ t\ s \longleftrightarrow (\exists p.\ p \neq [] \wedge emb\_step\_at'\ p\ t = s) \wedge t \neq s$
⟨*proof*⟩

lemma *position_if_emb_step_at*: $emb\_step\_at\ p\ d\ t = u \Longrightarrow t \neq u \Longrightarrow position\_of\ t\ (p\ @\ [d])$
⟨*proof*⟩

lemma *emb_step_at_if_position*:
assumes
$position\_of\ t\ (p\ @\ [d])$
shows $t \rightarrow_{emb} emb\_step\_at\ p\ d\ t$
⟨*proof*⟩

2.3 Embedding relation

Definition of an embedding as a sequence of embedding steps at given positions:

fun *emb_at* :: $(dir\ list \times dir)\ list \Rightarrow ('s, 'v)\ tm \Rightarrow ('s, 'v)\ tm$ **where**
emb_at_Nil: $emb\_at\ []\ t = t$ |
emb_at_Cons: $emb\_at\ ((p,d)\ \#\ ps)\ t = emb\_step\_at\ p\ d\ (emb\_at\ ps\ t)$

Definition of an embedding without using positions:

inductive *emb* (**infix** ‹$\trianglerighteq_{emb}$› *50*) **where**
refl: $t \trianglerighteq_{emb} t$ |
step: $t \trianglerighteq_{emb} u \Longrightarrow u \rightarrow_{emb} s \Longrightarrow t \trianglerighteq_{emb} s$

abbreviation *emb_neq* (**infix** ‹$\triangleright_{emb}$› *50*) **where** $emb\_neq\ t\ s \equiv t \trianglerighteq_{emb} s \wedge t \neq s$

The two definitions coincide:

lemma *emb_equiv*: $(t \trianglerighteq_{emb} s) = (\exists ps.\ emb\_at\ ps\ t = s)$
⟨*proof*⟩

lemma *emb_at_trans*: $emb\_at\ ps\ t = u \Longrightarrow emb\_at\ qs\ u = s \Longrightarrow emb\_at\ (qs\ @\ ps)\ t = s$
⟨*proof*⟩

lemma *emb_trans*: $t \trianglerighteq_{emb} u \Longrightarrow u \trianglerighteq_{emb} s \Longrightarrow t \trianglerighteq_{emb} s$
⟨*proof*⟩

lemma *emb_step_is_emb*: $t \rightarrow_{emb} s \Longrightarrow t \trianglerighteq_{emb} s$
⟨*proof*⟩

lemma *emb_hsize*: $t \trianglerighteq_{emb} s \Longrightarrow hsize\ t \geq hsize\ s$
⟨*proof*⟩

lemma *emb_prepend_step*: $t \rightarrow_{emb} u \Longrightarrow u \trianglerighteq_{emb} s \Longrightarrow t \trianglerighteq_{emb} s$
⟨*proof*⟩

lemma *sub_emb*: $sub\ s\ t \Longrightarrow t \trianglerighteq_{emb} s$
⟨*proof*⟩

lemma *sequence_emb_steps*: $t \trianglerighteq_{emb} s \longleftrightarrow (\exists us.\ us{\neq}[] \wedge hd\ us = t \wedge last\ us = s \wedge (\forall i.\ Suc\ i < length\ us \longrightarrow us\ !\ i \rightarrow_{emb} us\ !\ Suc\ i))$
⟨*proof*⟩

lemma *emb_induct_reverse* [*consumes 1*, *case_names refl step*]:
assumes
emb: $t \trianglerighteq_{emb} s$ **and**
refl: $\bigwedge t.\ P\ t\ t$ **and**
step: $\bigwedge t\ u\ s.\ t \rightarrow_{emb} u \Longrightarrow u \trianglerighteq_{emb} s \Longrightarrow P\ u\ s \Longrightarrow P\ t\ s$
shows
$P\ t\ s$
⟨*proof*⟩

lemma *emb_cases_reverse* [*consumes 1*, *case_names refl step*]:
$t \trianglerighteq_{emb} s \Longrightarrow (\bigwedge t'.\ t = t' \Longrightarrow s = t' \Longrightarrow P) \Longrightarrow (\bigwedge t'\ u\ s'.\ t = t' \Longrightarrow s = s' \Longrightarrow t' \rightarrow_{emb} u \Longrightarrow u \trianglerighteq_{emb} s' \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *emb_vars*: $t \trianglerighteq_{emb} s \Longrightarrow vars\ s \subseteq vars\ t$
⟨*proof*⟩

lemma *ground_emb*: $t \trianglerighteq_{emb} s \Longrightarrow ground\ t \Longrightarrow ground\ s$
⟨*proof*⟩

lemma *arg_emb*: $s \in set\ (args\ t) \Longrightarrow t \trianglerighteq_{emb} s$
⟨*proof*⟩

lemma *emb_step_at_subst*:
assumes
position_of t $(p\ @\ [d])$
shows
emb_step_at $p\ d\ (subst\ \varrho\ t) = subst\ \varrho\ (emb\_step\_at\ p\ d\ t)$
⟨*proof*⟩

lemma *emb_step_subst*: $t \rightarrow_{emb} s \Longrightarrow subst\ \varrho\ t \rightarrow_{emb} subst\ \varrho\ s$
⟨*proof*⟩

lemma *emb_subst*: $t\ \trianglerighteq_{emb} s \Longrightarrow subst\ \varrho\ t \trianglerighteq_{emb} subst\ \varrho\ s$
⟨*proof*⟩

lemma *emb_hsize_neq*:
assumes
$t \trianglerighteq_{emb} s\ t \neq s$
shows
$hsize\ t > hsize\ s$
⟨*proof*⟩

2.4 How are positions preserved under embedding steps?

Disjunct positions are preserved: For example, [L,R] is a position of f a (g b). When performing an embedding step at [R,R] to obtain f a b, the position [L,R] still exists. (More precisely, it even contains the same subterm, namely a.)

lemma *pos_emb_step_at_disjunct*:
assumes
$take\ (length\ q)\ p \neq q$
$take\ (length\ p)\ q \neq p$
shows
position_of $t\ (p\ @\ [d1]) \longleftrightarrow$ *position_of* $(emb\_step\_at\ q\ d2\ t)\ (p\ @\ [d1])$
⟨*proof*⟩

Even if only the last element of a position differs from the position of an embedding step, that position is preserved. For example, [L] is a position of f (g b). After performing an embedding step at [R,R] to obtain f b, the position [L] still exists. (More precisely, it even contains the same subterm, namely f.)

lemma *pos_emb_step_at_opp*:
position_of $t\ (p@[d1]) \longleftrightarrow$ *position_of* $(emb\_step\_at\ (p\ @\ [opp\ d1]\ @\ q)\ d2\ t)\ (p@[d1])$
⟨*proof*⟩

Positions are preserved under embedding steps below them:

lemma *pos_emb_step_at_nested*:
shows $position\_of\ (emb\_step\_at\ (p\ @\ [d1]\ @\ q)\ d2\ t)\ (p\ @\ [d1]) \longleftrightarrow position\_of\ t\ (p\ @\ [d1])$
$\langle proof \rangle$

2.5 Swapping embedding steps

The order of embedding steps at disjunct position can be changed freely:

lemma *swap_disjunct_emb_step_at*:
assumes
$length\ p \leq length\ q \Longrightarrow take\ (length\ p)\ q \neq p$ $length\ q \leq length\ p \Longrightarrow take\ (length\ q)\ p \neq q$
shows
$emb\_step\_at\ q\ d2\ (emb\_step\_at\ p\ d1\ t) = emb\_step\_at\ p\ d1\ (emb\_step\_at\ q\ d2\ t)$
$\langle proof \rangle$

An embedding step inside the branch that is removed in a second embedding step is useless. For example, the embedding f (g b) ->emb f b ->emb f can achieved using a single step f (g b) ->emb f.

lemma *merge_emb_step_at*:
$emb\_step\_at\ p\ d1\ (emb\_step\_at\ (p\ @\ [opp\ d1]\ @\ q)\ d2\ t) = emb\_step\_at\ p\ d1\ t$
$\langle proof \rangle$

When swapping two embedding steps of a position below another, one of the positions has to be slightly changed:

lemma *swap_nested_emb_step_at*:
$emb\_step\_at\ (p\ @\ q)\ d2\ (emb\_step\_at\ p\ d1\ t) = emb\_step\_at\ p\ d1\ (emb\_step\_at\ (p\ @\ [d1]\ @\ q)\ d2\ t)$
$\langle proof \rangle$

2.6 Performing embedding steps in order of a given priority

We want to perform all embedding steps first that modify the head or the number of arguments of a term. To this end we define the function prio_emb_step that performs the embedding step with the highest priority possible. The priority is given by a function "prio" from positions to nats, where the lowest number has the highest priority.

definition $prio\_emb\_pos :: (dir\ list \Rightarrow nat) \Rightarrow ('s,'v)\ tm \Rightarrow ('s,'v)\ tm \Rightarrow dir\ list$ **where**
$prio\_emb\_pos\ prio\ t\ s = (ARG\_MIN\ prio\ p.\ p \neq [] \wedge position\_of\ t\ p \wedge emb\_step\_at'\ p\ t \trianglerighteq_{emb} s)$

definition $prio\_emb\_step :: (dir\ list \Rightarrow nat) \Rightarrow ('s,'v)\ tm \Rightarrow ('s,'v)\ tm \Rightarrow ('s,'v)\ tm$ **where**
$prio\_emb\_step\ prio\ t\ s = emb\_step\_at'\ (prio\_emb\_pos\ prio\ t\ s)\ t$

lemma *prio_emb_posI*:
$t \trianglerighteq_{emb} s \Longrightarrow t \neq s \Longrightarrow prio\_emb\_pos\ prio\ t\ s \neq [] \wedge position\_of\ t\ (prio\_emb\_pos\ prio\ t\ s) \wedge emb\_step\_at'\ (prio\_emb\_pos\ prio\ t\ s)\ t \trianglerighteq_{emb} s$
$\langle proof \rangle$

lemma *prio_emb_pos_le*:
assumes $p \neq []$ $position\_of\ t\ p$ $emb\_step\_at'\ p\ t \trianglerighteq_{emb} s$
shows $prio\ (prio\_emb\_pos\ prio\ t\ s) \leq prio\ p$
$\langle proof \rangle$

We want an embedding step sequence in which the priority numbers monotonely increase. We can get such a sequence if the priority function assigns greater values to deeper positions.

lemma *prio_emb_pos_increase*:
assumes
$t \trianglerighteq_{emb} s$ $t \neq s$ $prio\_emb\_step\ prio\ t\ s \neq s$ **and**
valid_prio: $\bigwedge p\ q\ dp\ dq.\ prio\ (p\ @\ [dp]) > prio\ (q\ @\ [dq]) \Longrightarrow take\ (length\ p)\ q \neq p$
shows
$prio\ (prio\_emb\_pos\ prio\ t\ s) \leq prio\ (prio\_emb\_pos\ prio\ (prio\_emb\_step\ prio\ t\ s)\ s)$
(**is** $prio\ ?p1 \leq prio\ ?p2$)
$\langle proof \rangle$

lemma *sequence_prio_emb_steps*:
assumes
$t \trianglerighteq_{emb} s$
shows
$\exists\, us.\ us{\neq}[] \wedge hd\ us = t \wedge last\ us = s \wedge$
$(\forall\, i.\ Suc\ i < length\ us \longrightarrow (prio\_emb\_step\ prio\ (us\ !\ i)\ s = us\ !\ Suc\ i \wedge us\ !\ i \rightarrow_{emb} us\ !\ Suc\ i))$
⟨*proof*⟩

2.7 Embedding steps under arguments

We want to perform positions that modify the head and the umber of arguments first. Formally these positions can be characterized as "list_all (op = Left) p". We show here that embeddings at other positions do not modify the head, the number of arguments. Moreover, for each argument, the argument after the step is an embedding of the argument before the step.

lemma *emb_step_under_args_head*:
assumes
$\neg\ list\_all\ (\lambda x.\ x = Left)\ p$
shows
$head\ (emb\_step\_at\ p\ d\ t) = head\ t$
⟨*proof*⟩

lemma *emb_step_under_args_num_args*:
assumes
$\neg\ list\_all\ (\lambda x.\ x = Left)\ p$
shows
$num\_args\ (emb\_step\_at\ p\ d\ t) = num\_args\ t$
⟨*proof*⟩

lemma *emb_step_under_args_emb_step*:
assumes
$\neg\ list\_all\ (\lambda x.\ x = Left)\ p$
$position\_of\ t\ (p\ @\ [d])$
obtains i **where**
$i < num\_args\ t$
$args\ t\ !\ i \rightarrow_{emb} args\ (emb\_step\_at\ p\ d\ t)\ !\ i$ **and**
$\bigwedge j.\ j < num\_args\ t \Longrightarrow i \neq j \Longrightarrow args\ t\ !\ j = args\ (emb\_step\_at\ p\ d\ t)\ !\ j$
⟨*proof*⟩

lemma *emb_step_under_args_emb*:
assumes $\neg\ list\_all\ (\lambda x.\ x = Left)\ p$
$position\_of\ t\ (p\ @\ [d])$
shows
$\forall\, i.\ i < num\_args\ t \longrightarrow args\ t\ !\ i \trianglerighteq_{emb} args\ (emb\_step\_at\ p\ d\ t)\ !\ i$
⟨*proof*⟩

lemma *position_Left_only_subst*:
assumes $list\_all\ (\lambda x.\ x = Left)\ p$
and $position\_of\ (subst\ \varrho\ w)\ (p\ @\ [d])$
and $num\_args\ (subst\ \varrho\ w)\ = num\_args\ w$
shows $position\_of\ w\ (p\ @\ [d])$
⟨*proof*⟩

2.8 Rearranging embedding steps: first above, then below arguments

lemma *perform_emb_above_vars0*:
assumes
$subst\ \varrho\ s \trianglerighteq_{emb} u$
obtains w **where**
$s \trianglerighteq_{emb} w$
$subst\ \varrho\ w \trianglerighteq_{emb} u$
$\forall\, w'.\ w \rightarrow_{emb} w' \longrightarrow \neg\ subst\ \varrho\ w' \trianglerighteq_{emb} u$
⟨*proof*⟩

lemma *emb_only_below_vars*:
assumes
$subst\ \varrho\ s \trianglerighteq_{emb}\ u$
$s \trianglerighteq_{emb}\ w$
$is\_Sym\ (head\ w)$
$subst\ \varrho\ w \trianglerighteq_{emb}\ u$
$\forall w'.\ w \rightarrow_{emb} w' \longrightarrow \neg\ subst\ \varrho\ w' \trianglerighteq_{emb}\ u$
obtains *ws* **where**
$ws \neq []$
$hd\ ws = subst\ \varrho\ w$
$last\ ws = u$
$\forall i.\ Suc\ i < length\ ws \longrightarrow$
$(\exists p\ d.\ emb\_step\_at\ p\ d\ (ws\ !\ i) = ws\ !\ Suc\ i \wedge \neg\ list\_all\ (\lambda x.\ x = Left)\ p)$
$\forall i.\ i < length\ ws \longrightarrow head\ (ws\ !\ i) = head\ w \wedge num\_args\ (ws\ !\ i) = num\_args\ w$
$\forall i.\ i < length\ ws \longrightarrow (\forall k.\ k < num\_args\ w \longrightarrow args\ (subst\ \varrho\ w)\ !\ k \trianglerighteq_{emb}\ args\ (ws\ !\ i)\ !\ k)$
⟨*proof*⟩

lemma *perform_emb_above_vars*:
assumes
$subst\ \varrho\ s \trianglerighteq_{emb}\ u$
obtains *w* **where**
$s \trianglerighteq_{emb}\ w$
$subst\ \varrho\ w \trianglerighteq_{emb}\ u$
$is\_Sym\ (head\ w) \Longrightarrow head\ w = head\ u \wedge num\_args\ w = num\_args\ u \wedge (\forall k.\ k<num\_args\ w \longrightarrow args\ (subst\ \varrho\ w)\ !\ k \trianglerighteq_{emb}\ args\ u\ !\ k)$
⟨*proof*⟩

end

3 The Chop Operation on Lambda-Free Higher-Order Terms

theory *Chop*
imports *Embeddings*
begin

definition $chop :: ('s, 'v)\ tm \Rightarrow ('s, 'v)\ tm$ **where**
$chop\ t = apps\ (hd\ (args\ t))\ (tl\ (args\ t))$

3.1 Basic properties

lemma *chop_App_Hd*: $is\_Hd\ s \Longrightarrow chop\ (App\ s\ t) = t$
⟨*proof*⟩

lemma *chop_apps*: $is\_App\ t \Longrightarrow chop\ (apps\ t\ ts) = apps\ (chop\ t)\ ts$
⟨*proof*⟩

lemma *vars_chop*: $is\_App\ t \Longrightarrow vars\ (chop\ t) \cup vars\_hd\ (head\ t) = vars\ t$
⟨*proof*⟩

lemma *ground_chop*: $is\_App\ t \Longrightarrow ground\ t \Longrightarrow ground\ (chop\ t)$
⟨*proof*⟩

lemma *hsize_chop*: $is\_App\ t \Longrightarrow (Suc\ (hsize\ (chop\ t))) = hsize\ t$
⟨*proof*⟩

lemma *hsize_chop_lt*: $is\_App\ t \Longrightarrow hsize\ (chop\ t) < hsize\ t$
⟨*proof*⟩

lemma *chop_fun*:
assumes $is\_App\ t\ is\_App\ (fun\ t)$
shows $App\ (chop\ (fun\ t))\ (arg\ t) = chop\ t$
⟨*proof*⟩

3.2 Chop and the Embedding Relation

lemma *emb_step_chop*: $is\_App\ t \Longrightarrow t \rightarrow_{emb} chop\ t$
⟨*proof*⟩

lemma *chop_emb_step_at*:
assumes $is\_App\ t$
shows $chop\ t = emb\_step\_at\ (replicate\ (num\_args\ (fun\ t))\ Left)\ Right\ t$
⟨*proof*⟩

lemma *emb_step_at_chop*:
assumes *emb_step_at*: $emb\_step\_at\ p\ Right\ t = s$
and *pos*:$position\_of\ t\ (p\ @\ [Right])$
and *all_Left*: $list\_all\ (\lambda x.\ x = Left)\ p$
shows $chop\ t = s \vee chop\ t \rightarrow_{emb} s$
⟨*proof*⟩

lemma *emb_step_at_remove_arg*:
assumes *emb_step_at*: $emb\_step\_at\ p\ Left\ t = s$
and *pos*:$position\_of\ t\ (p\ @\ [Left])$
and *all_Left*: $list\_all\ (\lambda x.\ x = Left)\ p$
shows $let\ i = num\_args\ t - Suc\ (length\ p)\ in$
$head\ t = head\ s \wedge i < num\_args\ t \wedge args\ s = take\ i\ (args\ t)\ @\ drop\ (Suc\ i)\ (args\ t)$
⟨*proof*⟩

lemma *emb_step_cases* [*consumes 1*, *case_names chop extended_chop remove_arg under_arg*]:
assumes *emb*:$t \rightarrow_{emb} s$
and *chop*:$chop\ t = s \Longrightarrow P$
and *extended_chop*:$chop\ t \rightarrow_{emb} s \Longrightarrow P$
and *remove_arg*:$\bigwedge i.\ head\ t = head\ s \Longrightarrow i<num\_args\ t \Longrightarrow args\ s = take\ i\ (args\ t)\ @\ drop\ (Suc\ i)\ (args\ t) \Longrightarrow$
P
and *under_arg*:$\bigwedge i.\ head\ t = head\ s \Longrightarrow num\_args\ t = num\_args\ s \Longrightarrow args\ t\ !\ i \rightarrow_{emb} args\ s\ !\ i \Longrightarrow$
$(\bigwedge j.\ j<num\_args\ t \Longrightarrow i \neq j \Longrightarrow args\ t\ !\ j = args\ s\ !\ j) \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *chop_position_of*:
assumes $is\_App\ s$
shows $position\_of\ s\ (replicate\ (num\_args\ (fun\ s))\ dir.Left\ @\ [Right])$
⟨*proof*⟩

3.3 Chop and Substitutions

lemma *Suc_num_args*: $is\_App\ t \Longrightarrow Suc\ (num\_args\ (fun\ t)) = num\_args\ t$
⟨*proof*⟩

lemma *fun_subst*: $is\_App\ s \Longrightarrow subst\ \varrho\ (fun\ s) = fun\ (subst\ \varrho\ s)$
⟨*proof*⟩

lemma *args_subst_Hd*:
assumes $is\_Hd\ (subst\ \varrho\ (Hd\ (head\ s)))$
shows $args\ (subst\ \varrho\ s) = map\ (subst\ \varrho)\ (args\ s)$
⟨*proof*⟩

lemma *chop_subst_emb0*:
assumes $is\_App\ s$
assumes $chop\ (subst\ \varrho\ s) \neq subst\ \varrho\ (chop\ s)$
shows $emb\_step\_at\ (replicate\ (num\_args\ (fun\ s))\ Left)\ Right\ (chop\ (subst\ \varrho\ s)) = subst\ \varrho\ (chop\ s)$
⟨*proof*⟩

lemma *chop_subst_emb*:
assumes $is\_App\ s$

shows *chop* (*subst* ϱ *s*) $\trianglerighteq_{emb}$ *subst* ϱ (*chop* *s*)
⟨*proof*⟩

lemma *chop_subst_Hd*:
assumes *is_App s*
assumes *is_Hd* (*subst* ϱ (*Hd* (*head s*)))
shows *chop* (*subst* ϱ *s*) = *subst* ϱ (*chop s*)
⟨*proof*⟩

lemma *chop_subst_Sym*:
assumes *is_App s*
assumes *is_Sym* (*head s*)
shows *chop* (*subst* ϱ *s*) = *subst* ϱ (*chop s*)
⟨*proof*⟩

end

4 The Embedding Path Order for Lambda-Free Higher-Order Terms

theory *Lambda_Free_EPO*
imports *Chop Nested_Multisets_Ordinals.Multiset_More*
abbrevs $>t$ = $>_t$
and $\geq t$ = $\geq_t$
begin

This theory defines the embedding path order for λ-free higher-order terms.

4.1 Setup

locale *epo* = *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_list: *ext_compat_list* (*extf f*)
assumes *extf_ext_compat_snoc*: *ext_compat_snoc* (*extf f*)
assumes *extf_ext_compat_cons*: *ext_compat_cons* (*extf f*)
assumes *extf_min_empty*: *extf f gt* [*a*] []
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_list* = *ext_compat_list.compat_list*[*OF extf_ext_compat_list*]

lemmas *extf_compat_cons* = *ext_compat_cons.compat_cons*[*OF extf_ext_compat_cons*]
lemmas *extf_compat_snoc* = *ext_compat_snoc.compat_snoc*[*OF extf_ext_compat_snoc*]

lemmas *extf_compat_append_right* = *ext_compat_snoc.compat_append_right*[*OF extf_ext_compat_snoc*]
lemmas *extf_compat_append_left* = *ext_compat_cons.compat_append_left*[*OF extf_ext_compat_cons*]

lemma *extf_snoc*: $extf\ f\ gt\ (xs\ @\ [z])\ xs$
⟨*proof*⟩

4.2 Inductive Definitions

definition
  $chkchop :: (('s,\ 'v)\ tm \Rightarrow ('s,\ 'v)\ tm \Rightarrow bool) \Rightarrow ('s,\ 'v)\ tm \Rightarrow ('s,\ 'v)\ tm \Rightarrow bool$
where
  [*simp*]: $chkchop\ gt\ t\ s \longleftrightarrow is\_Hd\ s \lor gt\ t\ (chop\ s)$

definition
  $chkchop\_same :: (('s,\ 'v)\ tm \Rightarrow ('s,\ 'v)\ tm \Rightarrow bool) \Rightarrow ('s,\ 'v)\ tm \Rightarrow ('s,\ 'v)\ tm \Rightarrow bool$
where
  [*simp*]: $chkchop\_same\ gt\ t\ s \longleftrightarrow$
    $(if\ is\_Var\ (head\ t)$
    $then\ is\_App\ t \land chkchop\ gt\ (chop\ t)\ s$
    $else\ chkchop\ gt\ t\ s)$

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

lemma *chkchop_same_mono*[*mono*]: $gt \le gt' \Longrightarrow chkchop\_same\ gt \le chkchop\_same\ gt'$
  ⟨*proof*⟩

inductive $gt :: ('s,\ 'v)\ tm \Rightarrow ('s,\ 'v)\ tm \Rightarrow bool$ (**infix** ‹$>_t$› *50*) **where**
  gt_chop: $is\_App\ t \Longrightarrow chop\ t >_t s \lor chop\ t = s \Longrightarrow t >_t s$
| gt_diff: $head\ t >_{hd} head\ s \Longrightarrow is\_Sym\ (head\ s) \Longrightarrow chkchop\ (>_t)\ t\ s \Longrightarrow t >_t s$
| gt_same: $head\ t = head\ s \Longrightarrow chkchop\_same\ (>_t)\ t\ s \Longrightarrow$
    $(\forall f \in ground\_heads\ (head\ t).\ extf\ f\ (>_t)\ (args\ t)\ (args\ s)) \Longrightarrow t >_t s$

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

inductive $gt\_chop :: ('s,\ 'v)\ tm \Rightarrow ('s,\ 'v)\ tm \Rightarrow bool$ **where**
  gt_chopI: $is\_App\ t \Longrightarrow chop\ t \ge_t s \Longrightarrow gt\_chop\ t\ s$

inductive $gt\_diff :: ('s,\ 'v)\ tm \Rightarrow ('s,\ 'v)\ tm \Rightarrow bool$ **where**
  gt_diffI: $head\ t >_{hd} head\ s \Longrightarrow is\_Sym\ (head\ s) \Longrightarrow chkchop\ (>_t)\ t\ s \Longrightarrow gt\_diff\ t\ s$

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

lemma *gt_iff_chop_diff_same*: $t >_t s \longleftrightarrow gt\_chop\ t\ s \lor gt\_diff\ t\ s \lor gt\_same\ t\ s$
  ⟨*proof*⟩

4.3 Transitivity

lemma *t_gt_chop_t*: $is\_App\ t \Longrightarrow t >_t chop\ t$
  ⟨*proof*⟩

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

4.4 Irreflexivity

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

lemma *gt_antisym*: $t >_t s \Longrightarrow \neg\ s >_t t$
  ⟨*proof*⟩

4.5 Subterm Property

lemma *gt_emb_fun*: $App\ s\ t >_t s$
⟨*proof*⟩

lemma *gt_emb_arg*: $App\ s\ t >_t t$
⟨*proof*⟩

4.6 Compatibility with Contexts

lemma *gt_compat_fun*:
assumes $t' >_t t$
shows $App\ s\ t' >_t App\ s\ t$
⟨*proof*⟩

theorem *gt_compat_arg*:
shows $s' >_t s \Longrightarrow t' \geq_t t \Longrightarrow App\ s'\ t' >_t App\ s\ t$
⟨*proof*⟩

theorem *gt_compat_fun_strong*:
assumes *t′_gt_t*: $t' >_t t$
shows $apps\ s\ (t' \# us) >_t apps\ s\ (t \# us)$
⟨*proof*⟩

theorem *gt_or_eq_compat_App*: $s' \geq_t s \Longrightarrow t' \geq_t t \Longrightarrow App\ s'\ t' \geq_t App\ s\ t$
⟨*proof*⟩

theorem *gt_compat_App*:
shows $s' \geq_t s \Longrightarrow t' >_t t \Longrightarrow App\ s'\ t' >_t App\ s\ t$
⟨*proof*⟩

4.7 Compatibility with Embedding Relation

lemma *gt_embedding_step_property*:
assumes $t \rightarrow_{emb} s$
shows $t >_t s$
⟨*proof*⟩

lemma *gt_embedding_property*:
assumes $t \trianglerighteq_{emb} s\ t \neq s$
shows $t >_t s$
⟨*proof*⟩

4.8 Stability under Substitutions

lemma *extf_map2*:
assumes
$\forall y \in set\ ys \cup set\ xs.\ \forall x \in set\ ys \cup set\ xs.\ y >_t x \longrightarrow (h\ y) >_t (h\ x)$
$extf\ f\ (>_t)\ ys\ xs$
shows
$extf\ f\ (>_t)\ (map\ h\ ys)\ (map\ h\ xs)$
⟨*proof*⟩

theorem *gt_sus*:
assumes *ϱ_wary*: $wary\_subst\ \varrho$
assumes *ghd*: $\bigwedge x.\ ground\_heads\ (Var\ x) = UNIV$
shows $t >_t s \Longrightarrow subst\ \varrho\ t >_t subst\ \varrho\ s$
⟨*proof*⟩

4.9 Totality on Ground Terms

theorem *gt_total_ground*:
assumes *extf_total*: $\bigwedge f.\ ext\_total\ (extf\ f)$
shows $ground\ t \Longrightarrow ground\ s \Longrightarrow t >_t s \vee s >_t t \vee t = s$

⟨*proof*⟩

4.10 Well-foundedness

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

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

theorem *gt_wf*:
assumes *ghd_UNIV*: $\bigwedge x.\ ground\_heads\_var\ x = UNIV$
assumes *extf_wf*: $\bigwedge f.\ ext\_wf\ (extf\ f)$
shows $wfP\ (\lambda s\ t.\ t >_t s)$
⟨*proof*⟩

end

end