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

Alexander Bentkamp

May 26, 2024

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*
using *opp_def* **by** *auto*

lemma *shallower_pos*: *position_of t* (*p* @ *q* @ [*dq*]) ⟹ *position_of t* (*p* @ [*dp*])
proof (*induct p arbitrary*:*t*)
case *Nil*
then show *?case*
apply (*cases t*)
apply (*metis position_of_Hd Nil_is_append_conv list.exhaust*)
by (*metis* (*full_types*) *position_of_Nil position_of_left*
position_of_right dir.exhaust self_append_conv2)
next
case (*Cons a p*)
then show *?case*
apply (*cases a*)
apply (*metis position_of_Hd position_of_left append_Cons args.cases*)
by (*metis position_of_Hd position_of_right append_Cons args.cases*)
qed

lemma *no_position_replicate_num_args*: ¬ *position_of t* (*replicate* (*num_args t*) *Left* @ [*d*])
proof (*induct num_args t arbitrary*:*t*)
case *0*
have *is_Hd t*
using *0.hyps args_Nil_iff_is_Hd* **by** *auto*
then show *?case*
by (*metis position_of.elims*(*2*) *snoc_eq_iff_butlast tm.discI*(*2*))
next
case (*Suc n*)
have *is_App t*
using *Suc.hyps*(*2*) *args_Nil_iff_is_Hd* **by** *force*
then have ¬ *position_of* (*fun t*) (*replicate* (*num_args* (*fun t*)) *dir.Left* @ [*d*]) **using** *Suc.hyps*(*1*)[*of fun t*]
by (*metis Suc Suc_inject args.elims length_0_conv length_append_singleton nat.distinct*(*1*) *tm.sel*(*4*))
have ⋀*ds*. *replicate* (*num_args t*) *dir.Left* @ [*d*] ≠ *dir.Right* # *ds*
by (*metis* ‹*is_App t*› *args_Nil_iff_is_Hd dir.distinct*(*1*) *hd_append hd_replicate length_0_conv list.sel*(*1*)
replicate_empty)
then show *?case* **using** *position_of.elims*(*2*)[*of t* (*replicate* (*num_args t*) *dir.Left* @ [*d*])]
by (*metis position_of_left* ‹¬ *position_of* (*fun t*) (*replicate* (*num_args* (*fun t*)) *dir.Left* @ [*d*])›
‹*is_App t*› *append_Cons args.simps*(*2*) *length_append_singleton replicate_Suc tm.collapse*(*2*))
qed

lemma *shorten_position*:*position_of t* (*p* @ *q*) ⟹ *position_of t p*
proof (*induct q rule*: *rev_induct*)
case *Nil*

then show *?case*
by *simp*
next
case (*snoc x xs*)
then show *?case*
by (*metis position_of_Nil append_assoc append_butlast_last_id shallower_pos*)
qed

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*
by (*metis emb_step_at.simps*(*5*) *is_Hd_def*)

Definition of an embedding step without using positions.

inductive *emb_step* (**infix** →*emb* *50*) **where**
left: (*App t1 t2*) →*emb* *t1* |
right: (*App t1 t2*) →*emb* *t2* |
context_left: *t* →*emb* *s* ⟹ (*App t u*) →*emb* (*App s u*) |
context_right: *t* →*emb* *s* ⟹ (*App u t*) →*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
show *t* →*emb* *s* ⟹ (∃ *p d*. *emb_step_at p d t* = *s*) ∧ *t* ≠ *s*
proof (*induct t s rule*:*emb_step.induct*)
case *left*
then show *?case*
by (*metis add_Suc_right emb_step_at.simps*(*1*) *leD le_add_same_cancel1 le_imp_less_Suc tm.size*(*4*) *zero_order*(*1*))
next
case *right*
then show *?case*
by (*metis add.right_neutral add_le_cancel_left antisym emb_step_at.simps*(*2*) *le_add_same_cancel2 le_imp_less_Suc length_greater_0_conv list.size*(*3*) *tm.size*(*4*) *zero_order*(*1*))
next
case (*context_left t s*)
obtain *p d* **where** *emb_step_at p d t* = *s*
using *context_left.hyps*(*2*) **by** *blast*
then have ⋀*u*. *emb_step_at* (*Left* # *p*) *d* (*App t u*) = *App s u*
by (*metis emb_step_at.simps*(*3*))
then show *?case*
using *context_left.hyps*(*2*) **by** *blast*
next
case (*context_right t s*)
obtain *p d* **where** *emb_step_at p d t* = *s*
using *context_right.hyps*(*2*) **by** *blast*
then have ⋀*u*. *emb_step_at* (*Right* # *p*) *d* (*App u t*) = *App u s*
by (*metis emb_step_at.simps*(*4*))
then show *?case*
using *context_right.hyps*(*2*) **by** *blast*

```
  qed
  show (∃ p d. emb_step_at p d t = s) ∧ t ≠ s ⟹ t →emb s
  proof −
    assume (∃ p d. emb_step_at p d t = s) ∧ t ≠ s
    then obtain p d where emb_step_at p d t = s t ≠ s by blast
    then show ?thesis
    proof (induct arbitrary:t rule:emb_step_at_induct)
      case (left u1 u2)
      show ?case using left(1)
        apply (rule emb_step_at.elims)
        using emb_step.left by blast+
    next
      case (right u1 u2)
      show ?case  using right(1)
        apply (rule emb_step_at.elims)
        using emb_step.right by blast+
    next
      case (left_context u1 u2 t' s')
      show ?case
      by (metis emb_step_at_left_context emb_step.simps emb_step_at_is_App left_context.hyps left_context.prems(1)
left_context.prems(2)
          tm.collapse(2) tm.inject(2))
    next
      case (right_context u1 u2 t' s')
      show ?case
      by (metis emb_step_at_right_context emb_step.simps emb_step_at_is_App right_context.hyps right_context.prems(1)
right_context.prems(2) tm.collapse(2) tm.inject(2))
    next
      case (head uu uv h)
      show ?case using head(1)
        apply (rule emb_step_at.elims)
        using emb_step.left apply blast
        using emb_step.right apply blast
        apply simp_all
        using head.prems(2) by blast
    qed
  qed
qed

lemma emb_step_fun: is_App t ⟹ t →emb (fun t)
  by (metis emb_step.intros(1) tm.collapse(2))

lemma emb_step_arg: is_App t ⟹ t →emb (arg t)
  by (metis emb_step.intros(2) tm.collapse(2))

lemma emb_step_hsize: t →emb s ⟹ hsize t > hsize s
  by (induction rule: emb_step.induct; simp_all)

lemma emb_step_vars: t →emb s ⟹ vars s ⊆ vars t
  by (induction rule: emb_step.induct, auto)

lemma emb_step_equiv': emb_step t s ⟷ (∃ p. p ≠ [] ∧ emb_step_at' p t = s) ∧ t ≠ s
  using butlast_snoc emb_step_equiv last_snoc
  by (metis snoc_eq_iff_butlast)

lemma position_if_emb_step_at: emb_step_at p d t = u ⟹ t ≠ u ⟹ position_of t (p @ [d])
proof (induct p arbitrary:t u)
  case Nil
 then show ?case by (metis emb_step_at_head position_of_Nil append_self_conv2 list.sel(3) position_of.elims(3))
next
  case (Cons a p)
  then show ?case
  proof (cases t)
```

```
    case (Hd x)
    then show ?thesis using Cons by simp
  next
    case (App t1 t2)
    then show ?thesis using Cons  apply (cases a) apply simp
      apply blast
      by auto
  qed
qed

lemma emb_step_at_if_position:
  assumes
    position_of t (p @ [d])
  shows t →emb emb_step_at p d t
  using assms proof (induct p arbitrary:t)
  case Nil
  then show ?case
    by (cases d;cases t;simp add: emb_step.left emb_step.right)
next
  case (Cons a p)
  then show ?case
    apply (cases a)
     apply (cases t)
      apply (simp_all add: context_left context_right)
    apply (cases t)
    by (simp_all add: context_left context_right)
qed
```

2.3 Embedding relation

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

```
fun emb_at :: (dir list × dir) list ⇒ ('s, 'v) tm  ⇒ ('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 ⊵emb 50) where
  refl: t ⊵emb t |
  step: t ⊵emb u ⟹ u →emb s ⟹ t ⊵emb s

abbreviation emb_neq (infix ⊳emb 50) where emb_neq t s ≡ t ⊵emb s ∧ t ≠ s
```

The two definitions coincide:

```
lemma emb_equiv: (t ⊵emb s) = (∃ps. emb_at ps t = s)
proof
  show ∃ps. emb_at ps t = s ⟹ t ⊵emb s
  proof −
    assume ∃ps. emb_at ps t = s
    then obtain ps where emb_at ps t = s
      by blast
    then show ?thesis
    proof (induct ps arbitrary:s)
      case Nil
      then show ?case by (simp add: emb.refl)
    next
      case (Cons a ps)
      show ?case
        using Cons(2) apply (rule emb_at.elims)
       apply simp
       by (metis Cons.hyps emb.simps emb_step_equiv list.inject)
    qed
  qed
```

next
  show $t \trianglerighteq_{emb} s \Longrightarrow \exists ps.\ emb\_at\ ps\ t = s$
  proof (*induct rule*: *emb.induct*)
    case (*refl t*)
    then show *?case*
      using *emb_at_Nil* **by** *blast*
  next
    case (*step t u s*)
    then show *?case*
      by (*metis emb_at_Cons emb_step_equiv*)
  qed
qed

lemma *emb_at_trans*: $emb\_at\ ps\ t = u \Longrightarrow emb\_at\ qs\ u = s \Longrightarrow emb\_at\ (qs\ @\ ps)\ t = s$
  by (*induct qs arbitrary*:*s*; *auto*)

lemma *emb_trans*: $t \trianglerighteq_{emb} u \Longrightarrow u \trianglerighteq_{emb} s \Longrightarrow t \trianglerighteq_{emb} s$
  by (*metis emb_at_trans emb_equiv*)

lemma *emb_step_is_emb*: $t \rightarrow_{emb} s \Longrightarrow t \trianglerighteq_{emb} s$
  by (*meson emb.simps*)

lemma *emb_hsize*: $t \trianglerighteq_{emb} s \Longrightarrow hsize\ t \geq hsize\ s$
  apply (*induction rule*: *emb.induct*)
  apply *simp*
  using *emb_step_hsize* **by** *fastforce*

lemma *emb_prepend_step*: $t \rightarrow_{emb} u \Longrightarrow u \trianglerighteq_{emb} s \Longrightarrow t \trianglerighteq_{emb} s$
  using *emb_step_is_emb emb_trans* **by** *blast*

lemma *sub_emb*: $sub\ s\ t \Longrightarrow t \trianglerighteq_{emb} s$
proof (*induction rule*:*sub.induct*)
  case (*sub_refl s*)
  then show *?case*
    by (*simp add*: *emb.refl*)
next
  case (*sub_fun s t u*)
  then show *?case*
    using *emb_prepend_step right* **by** *blast*
next
  case (*sub_arg s t u*)
  then show *?case*
    using *emb_prepend_step left* **by** *blast*
qed

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
  show $t \trianglerighteq_{emb} s \Longrightarrow \exists us.\ us \neq [] \wedge hd\ us = t \wedge last\ us = s \wedge (\forall i.\ Suc\ i < length\ us \longrightarrow us\ !\ i \rightarrow_{emb} us\ !\ Suc\ i)$
  proof (*induct rule*:*emb.induct*)
    case (*refl t*)
    then show *?case*
      using *Suc_less_eq add.right_neutral add_Suc_right append_Nil last_snoc list.sel*(*1*) *list.size*(*3*) *list.size*(*4*) *not_less_zero*
      by *auto*
  next
    case (*step t u s*)
    then obtain *us* **where**
      $us \neq []$
      $hd\ us = t$
      $last\ us = u$
      $\forall i.\ Suc\ i < length\ us \longrightarrow us\ !\ i \rightarrow_{emb} us\ !\ Suc\ i$ **by** *blast*
    then have $hd\ (us\ @\ [s]) = t\ last\ (us\ @\ [s]) = s$ **by** *simp_all*

```
    moreover have  (∀ i. Suc i < length (us @ [s]) ⟶ (us @ [s]) ! i →emb (us @ [s]) ! Suc i)
      by (metis (mono_tags, lifting) Suc_less_eq ‹∀ i. Suc i < length us ⟶ us ! i →emb us ! Suc i› ‹last us = u›
‹us ≠ []›
        append_butlast_last_id length_append_singleton less_antisym nth_append nth_append_length step.hyps(3))
    ultimately show ?case by blast
  qed
  show ∃ us. us ≠ [] ∧ hd us = t ∧ last us = s ∧ (∀ i. Suc i < length us ⟶ us ! i →emb us ! Suc i) ⟹ t ⊵emb s
  proof −
    assume ∃ us. us ≠ [] ∧ hd us = t ∧ last us = s ∧ (∀ i. Suc i < length us ⟶ us ! i →emb us ! Suc i)
    then obtain us where us ≠ [] hd us = t last us = s ∀ i. Suc i < length us ⟶ us ! i →emb us ! Suc i by blast
    then show ?thesis
    proof (induct us arbitrary:t)
      case Nil
      then show ?case by simp
    next
      case (Cons a us)
      then show ?case
        using emb.refl emb_step_is_emb emb_trans hd_conv_nth by fastforce
    qed
  qed
qed

lemma emb_induct_reverse [consumes 1, case_names refl step]:
  assumes
emb: t ⊵emb s and
refl: ⋀t. P t t and
step: ⋀t u s. t →emb u ⟹ u ⊵emb s ⟹ P u s ⟹ P t s
shows
P t s
proof −
  obtain us where us≠[] hd us = t last us = s ∀ i. Suc i < length us ⟶ us ! i →emb us ! Suc i
    using emb sequence_emb_steps by blast
  define us′ where us′ == tl us
  then have last ([t] @ us′) = s ∀ i. i < length us′ ⟶ ([t] @ us′) ! i →emb ([t] @ us′) ! Suc i
    using ‹hd us = t› ‹last us = s› ‹us ≠ []› apply force
    using ‹∀ i. Suc i < length us ⟶ us ! i →emb us ! Suc i› ‹hd us = t› ‹us ≠ []› us′_def by force
  then show ?thesis
  proof (induct us′ arbitrary:t)
    case Nil
  then show ?case
    by (simp add: local.refl)
  next
    case (Cons a us′)
    then have P a s
      by (metis Suc_mono append_Cons append_Nil last.simps length_Cons list.discI nth_Cons_Suc)
    have t →emb a
      using Cons.prems(2) by auto
    have a ⊵emb s unfolding sequence_emb_steps
    by (metis append_Cons append_Nil last_append list.sel(1) list.simps(3) local.Cons(2) local.Cons(3) nth_Cons_Suc)
    show ?case using step[OF ‹t →emb a› ‹a ⊵emb s› ‹P a s›] .
  qed
qed

lemma emb_cases_reverse [consumes 1, case_names refl step]:
  t ⊵emb s ⟹ (⋀t′. t = t′ ⟹ s = t′ ⟹ P) ⟹ (⋀t′ u s′. t = t′ ⟹ s = s′ ⟹ t′ →emb u ⟹ u ⊵emb s′ ⟹
P) ⟹ P
  by (induct rule:emb_induct_reverse; blast+)

lemma emb_vars: t ⊵emb s ⟹ vars s ⊆ vars t
  apply (induct rule: emb.induct)
   apply simp
  using emb_step_vars by auto
```

lemma *ground_emb*: $t \trianglerighteq_{emb} s \Longrightarrow$ *ground* $t \Longrightarrow$ *ground* s
using *emb_vars* **by** *blast*

lemma *arg_emb*: $s \in$ *set* (*args* t) $\Longrightarrow t \trianglerighteq_{emb} s$
by (*simp add*: *sub_args sub_emb*)

lemma *emb_step_at_subst*:
assumes
position_of t (p @ [d])
shows
emb_step_at p d (*subst* ϱ t) = *subst* ϱ (*emb_step_at* p d t)
using *assms* **proof** (*induction p arbitrary*:t)
case *Nil*
then obtain *t1 t2* **where** t = *App t1 t2*
using *position_of_Hd append_Nil tm.collapse*(*1*)
by (*metis tm.collapse*(*2*))
then show *?case*
by (*cases d*; *simp add*: ‹t = *App t1 t2*›)
next
case (*Cons a p*)
then obtain *t1 t2* **where** t = *App t1 t2*
using *position_of.elims*(*2*) **by** *blast*
then show *?case*
using *Cons.IH Cons.prems* **by** (*cases a*;*auto*)
qed

lemma *emb_step_subst*: $t \rightarrow_{emb} s \Longrightarrow$ *subst* ϱ $t \rightarrow_{emb}$ *subst* ϱ s
by (*induct rule*:*emb_step.induct*;
simp_all add: *emb_step.left emb_step.right context_left context_right*)

lemma *emb_subst*: $t \trianglerighteq_{emb} s \Longrightarrow$ *subst* ϱ $t \trianglerighteq_{emb}$ *subst* ϱ s
apply (*induct rule*:*emb.induct*)
apply (*simp add*: *emb.refl*)
using *emb.step emb_step_subst* **by** *blast*

lemma *emb_hsize_neq*:
assumes
$t \trianglerighteq_{emb} s$ $t \neq s$
shows
hsize $t >$ *hsize* s
by (*metis assms*(*1*) *assms*(*2*) *emb_cases_reverse emb_hsize emb_step_hsize leD le_imp_less_or_eq*)

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*])
using *assms*
proof (*induct length p* + *length q arbitrary*:*t p q rule*:*less_induct*)
case *less*
then show *?case*
proof (*cases is_Hd t* ∨ p = [] ∨ q = [])
case *True*
then show *?thesis*
by (*metis emb_step_at_is_App less.prems*(*1*) *less.prems*(*2*) *list.size*(*3*) *take_eq_Nil*)
next
case *False*

```
    then obtain t1 t2 p0 p' q0 q' where t = App t1 t2 p = p0 # p' q = q0 # q'
      by (metis list.collapse tm.collapse(2))
    then show ?thesis
      apply (cases p0; cases q0)
      using less.hyps less.prems(1) less.prems(2) by auto
  qed
qed
```

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]) ⟷ position_of (emb_step_at (p @ [opp d1] @ q) d2 t) (p@[d1])
proof (induct p arbitrary:t)
  case Nil
  then show ?case by (cases t, simp, cases d1, simp_all)
next
  case (Cons a p)
  then show ?case
    apply (cases a)
    apply (metis emb_step_at_left_context position_of_left append_Cons emb_step_at_is_App tm.collapse(2))
    by (metis emb_step_at_right_context position_of_right append_Cons emb_step_at_is_App tm.collapse(2))
qed
```

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]) ⟷ position_of t (p @ [d1])
proof (induct p arbitrary:t)
  case Nil
  then show ?case by (cases t, simp, cases d1, simp_all)
next
  case (Cons a p)
  then show ?case
    apply (cases a;cases t)
    using Cons.hyps Cons.prems by auto
qed
```

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 ≤ length q ⟹ take (length p) q ≠ p length q ≤ length p ⟹take (length q) p ≠ q
  shows
    emb_step_at q d2 (emb_step_at p d1 t) = emb_step_at p d1 (emb_step_at q d2 t)
  using assms
proof (induct length p + length q arbitrary:t p q rule:less_induct)
  case less
  then show ?case
  proof (cases is_Hd t ∨ p = [] ∨ q = [])
    case True
    then show ?thesis
      using emb_step_at_is_App less.prems(1) less.prems(2) list.size(3) take_eq_Nil
      by (metis le_zero_eq nat_le_linear)
  next
    case False
    then obtain t1 t2 p0 p' q0 q' where t = App t1 t2 p = p0 # p' q = q0 # q'
      by (metis list.collapse tm.collapse(2))
    then show ?thesis
      apply (cases p0; cases q0)
      using less.hyps less.prems(1) less.prems(2) by auto
  qed
```

qed

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
proof (*induct p arbitrary:t*)
case *Nil*
then show *?case* **by** (*cases t*, *simp*, *cases d1*, *simp_all*)
next
case (*Cons a p*)
then show *?case*
apply (*cases a*)
apply (*metis* (*full_types*) *emb_step_at_left_context append_Cons emb_step_at_is_App tm.collapse*(*2*))
by (*metis* (*full_types*) *emb_step_at_right_context append_Cons emb_step_at_is_App tm.collapse*(*2*))
qed

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)
proof (*induct p arbitrary:t*)
case *Nil*
then show *?case* **by** (*cases t*, *simp*, *cases d1*, *simp_all*)
next
case (*Cons a p*)
then show *?case*
apply (*cases a*;*cases t*)
using *Cons.hyps Cons.prems* **by** *auto*
qed

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* ⇒ *nat*) ⇒ ('*s*,'*v*) *tm* ⇒ ('*s*,'*v*) *tm* ⇒ *dir list* **where**
prio_emb_pos prio t s = (*ARG_MIN prio p*. *p* ≠ [] ∧ *position_of t p* ∧ *emb_step_at*′ *p t* ⊵$_{emb}$ *s*)

definition *prio_emb_step* :: (*dir list* ⇒ *nat*) ⇒ ('*s*,'*v*) *tm* ⇒ ('*s*,'*v*) *tm* ⇒ ('*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 ⊵$_{emb}$ *s* ⟹ *t* ≠ *s* ⟹ *prio_emb_pos prio t s* ≠ [] ∧ *position_of t* (*prio_emb_pos prio t s*) ∧ *emb_step_at*′ (*prio_emb_pos prio t s*) *t* ⊵$_{emb}$ *s*
proof (*induct rule*:*emb_induct_reverse*)
case (*refl t*)
then show *?case* **by** *simp*
next
case (*step t u s*)
obtain *p* **where** *0*:*p* ≠ [] ∧ *position_of t p* ∧ *emb_step_at*′ *p t* ⊵$_{emb}$ *s*
using *emb_step_equiv*′ *step.hyps*(*1*) *append_butlast_last_id position_if_emb_step_at*
by (*metis step.hyps*(*2*))
then have *prio_emb_pos prio t s* ≠ [] ∧ *position_of t* (*prio_emb_pos prio t s*) ∧ *emb_step_at*′ (*prio_emb_pos prio t s*) *t* ⊵$_{emb}$ *s*
using *prio_emb_pos_def step.hyps*(*2*) *arg_min_natI* **by** (*metis* (*mono_tags*, *lifting*))
then show *?case* **by** *blast*
qed

lemma *prio_emb_pos_le*:
assumes *p* ≠ [] *position_of t p emb_step_at*′ *p t* ⊵$_{emb}$ *s*

shows *prio* (*prio_emb_pos prio t s*) ≤ *prio p*
by (*simp add*: *arg_min_nat_le assms*(*1*) *assms*(*2*) *assms*(*3*) *prio_emb_pos_def*)

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 ⊵$_{emb}$ *s t* ≠ *s prio_emb_step prio t s* ≠ *s* **and**
valid_prio: ⋀*p q dp dq*. *prio* (*p* @ [*dp*]) > *prio* (*q* @ [*dq*]) ⟹ *take* (*length p*) *q* ≠ *p*
shows
prio (*prio_emb_pos prio t s*) ≤ *prio* (*prio_emb_pos prio* (*prio_emb_step prio t s*) *s*)
(**is** *prio ?p1* ≤ *prio ?p2*)
proof (*rule ccontr*)
assume *contr*: ¬ *prio ?p1* ≤ *prio ?p2*
have *take* (*length* (*butlast ?p1*)) (*butlast ?p2*) ≠ (*butlast ?p1*)
using *contr valid_prio*
by (*metis append_butlast_last_id assms*(*1*) *assms*(*2*) *assms*(*3*) *not_le_imp_less prio_emb_posI prio_emb_step_def*)
then have *butlast_neq*: *butlast ?p2* ≠ *butlast ?p1*
by *auto*

define *u* **where** *u* = *prio_emb_step prio* (*prio_emb_step prio t s*) *s*
then have *u* ⊵$_{emb}$ *s*
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *prio_emb_posI prio_emb_step_def*)

define *p* **where** *p* = *butlast ?p2*
define *q* **where** *q* = *drop* (*Suc* (*length* (*butlast ?p2*))) (*butlast ?p1*)
define *dp* **where** *dp* = *last ?p2*
define *dq* **where** *dq* = *last ?p1*
define *f* **where** *f* = *butlast ?p1* ! *length* (*butlast ?p2*)

have *position*: *position_of* (*prio_emb_step prio t s*) *?p2*
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *prio_emb_posI prio_emb_step_def*)

show *False*
proof (*cases take* (*length p*) (*butlast ?p1*) = *p*)
case *True*
have *length* (*butlast ?p1*) ≠ *length p* **using** *butlast_neq*
using *True p_def* **by** *auto*
then have *length* (*butlast ?p1*) > *length p*
using *nat_le_linear True nat_neq_iff* **by** *auto*
then have *0*:*p* @ *f* # *q* = *butlast ?p1*
by (*metis True f_def id_take_nth_drop p_def q_def*)
have *u1*: *u* = *emb_step_at p dp* (*emb_step_at* (*p* @ *f* # *q*) *dq t*)
unfolding *0* **unfolding** *p_def u_def dp_def dq_def prio_emb_step_def* **by** *simp*
show *False*
proof (*cases f* = *dp*)
case *True*
then have *u* = *emb_step_at* (*p* @ *q*) *dq* (*emb_step_at p dp t*)
using *swap_nested_emb_step_at*[*of p q dq dp t*] *u1* **by** *simp*
then have *emb_step_at p dp t* →$_{emb}$ *u* ∨ *emb_step_at p dp t* = *u*
using *emb_step_equiv*[*of emb_step_at p dp t u*] **by** *blast*
then have *emb_step_at p dp t* ⊵$_{emb}$ *s*
using ‹*u* ⊵$_{emb}$ *s*› *emb_prepend_step* **by** *blast*
then have *position_of t ?p2*
by (*metis position 0 position_of_Nil True append_Cons append_Nil2 append_butlast_last_id*
append_eq_append_conv_if append_eq_conv_conj dp_def dq_def p_def pos_emb_step_at_nested prio_emb_step_def)
then show *?thesis*
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *contr dp_def p_def prio_emb_posI*
prio_emb_pos_le prio_emb_step_def
by (*metis* ‹*emb_step_at p dp t* ⊵$_{emb}$ *s*›)
next
case *False*
then have *opp dp* = *f*

```
      by (metis dir.exhaust opp_def)
    then have u = emb_step_at p dp t
      by (subst merge_emb_step_at[of p dp q dq t, symmetric], simp add: u1)
    then have position_of t ?p2
      by (metis Cons_nth_drop_Suc position_of_Nil True dp_def
          ‹length p < length (butlast ?p1)› ‹opp dp = f› append.assoc append_butlast_last_id
       append_take_drop_id f_def list.sel(1) p_def pos_emb_step_at_opp position take_hd_drop prio_emb_step_def)
    then show ?thesis
      by (metis dp_def p_def ‹u = emb_step_at p dp t› ‹u ⊵emb s› assms(1) assms(2) assms(3) contr
       prio_emb_posI prio_emb_pos_le prio_emb_step_def)
  qed
 next
  case False
  define p' where p' = butlast ?p1
  have take (length p') p ≠ p'
    using ‹take (length (butlast ?p1)) (butlast ?p2) ≠ butlast ?p1› p'_def p_def by blast
  have take (length p) p' ≠ p using False p'_def by metis
  have u = emb_step_at p dp (emb_step_at p' dq t)
    unfolding u_def prio_emb_step_def
    by (simp add: dp_def dq_def p'_def p_def prio_emb_step_def)
  then have u = emb_step_at p' dq (emb_step_at p dp t)
    using swap_disjunct_emb_step_at[of p p' dq dp t]
    by (simp add: ‹take (length p') p ≠ p'› ‹take (length p) p' ≠ p›)
  then have emb_step_at p dp t ⊵emb s
    by (metis ‹u ⊵emb s› emb_prepend_step emb_step_equiv)
  have position_of t (p @ [dp]) using pos_emb_step_at_disjunct
    by (metis ‹take (length p') p ≠ p'› ‹take (length p) p' ≠ p› append_butlast_last_id butlast.simps(1)
    dp_def list.size(3) p'_def p_def position take_eq_Nil prio_emb_step_def)
  then show ?thesis
    by (metis False ‹emb_step_at p dp t ⊵emb s› append_butlast_last_id butlast.simps(1) contr
    dp_def list.size(3) p_def take_eq_Nil prio_emb_pos_le)
 qed
qed

lemma sequence_prio_emb_steps:
 assumes
  t ⊵emb s
 shows
  ∃ us. us≠[] ∧ hd us = t ∧ last us = s ∧
  (∀ i. Suc i < length us ⟶ (prio_emb_step prio (us ! i) s = us ! Suc i ∧ us ! i →emb us ! Suc i))
 using assms
proof (induct t rule:measure_induct_rule[of hsize])
 case (less t)
 show ?case
 proof (cases t = s)
  case True
  then show ?thesis by auto
 next
  case False
  then have prio_emb_pos prio t s ≠ [] emb_step_at' (prio_emb_pos prio t s) t ⊵emb s
    using prio_emb_posI less by blast+
  have emb_step1:t →emb emb_step_at' (prio_emb_pos prio t s) t
    by (simp add: False emb_step_at_if_position less.prems prio_emb_posI)
  have hsize (emb_step_at' (prio_emb_pos prio t s) t) < hsize t
    by (simp add: False emb_step_at_if_position emb_step_hsize less.prems(1) prio_emb_posI)
  then obtain us where us_def:
    us ≠ []
    hd us = emb_step_at' (prio_emb_pos prio t s) t
    last us = s
    ∀ i. Suc i < length us ⟶ prio_emb_step prio (us ! i) s = us ! Suc i ∧ us ! i →emb us ! Suc i
    using less(1)[of emb_step_at' (prio_emb_pos prio t s) t] emb_step_hsize
     ‹emb_step_at' (prio_emb_pos prio t s) t ⊵emb s› by blast
```

```
    have t # us ≠ []  hd (t # us) = t last (t # us) = s using us_def by simp_all
    have
      ∀ i. Suc i < length (t # us) ⟶ (prio_emb_step prio ((t # us) ! i) s = (t # us) ! Suc i ∧ (t # us) ! i →emb
(t # us) ! Suc i)
    proof (rule allI, rule impI)
      fix i assume  Suc i < length (t # us)
      show prio_emb_step prio ((t # us) ! i) s = (t # us) ! Suc i ∧ (t # us) ! i →emb (t # us) ! Suc i
      proof (cases i)
        case 0
        show ?thesis
          using 0 emb_step1 hd_conv_nth prio_emb_step_def us_def(1) us_def(2) by fastforce
      next
        case (Suc nat)
        then show ?thesis
          using ‹Suc i < length (t # us)› us_def(4) by auto
      qed
    qed
    then show ?thesis
      using ‹hd (t # us) = t› ‹last (t # us) = s› by blast
  qed
qed
```

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
    ¬ list_all (λx. x = Left) p
  shows
    head (emb_step_at p d t) = head t
  using assms  proof(induction p arbitrary:t)
  case Nil
  then show ?case
    by simp
next
  case (Cons a p)
  then show ?case
  proof (cases t)
    case (Hd x)
    then show ?thesis
      by (metis emb_step_at_head)
  next
    case (App t1 t2)
    then show ?thesis
    proof (cases a)
      case Left
      then show ?thesis using Cons
        by (simp add: App)
    next
      case Right
      then show ?thesis
        by (metis App emb_step_at_right_context head_App)
    qed
  qed
qed

lemma emb_step_under_args_num_args:
  assumes
    ¬ list_all (λx. x = Left) p
  shows
```

```
    num_args (emb_step_at p d t) = num_args t
  using assms
proof(induction p arbitrary:t)
  case Nil
  then show ?case
    using list_all_simps(2) by blast
next
  case (Cons a p)
  then show ?case
  proof (cases t)
    case (Hd x1)
    then show ?thesis
      by (metis emb_step_at_head)
  next
    case (App t1 t2)
    then show ?thesis unfolding App apply (cases a)
      using App Cons.IH Cons.prems(1) by auto
  qed
qed

lemma emb_step_under_args_emb_step:
  assumes
   ¬ list_all (λx. x = Left) p
   position_of t (p @ [d])
 obtains i where
   i < num_args t
   args t ! i →emb args (emb_step_at p d t) ! i and
   ⋀j. j < num_args t ⟹ i ≠ j ⟹ args t ! j = args (emb_step_at p d t) ! j
  using assms
proof(induction p arbitrary:t thesis)
  case Nil
  then show ?case
    by simp
next
  case (Cons a p)
  obtain t1 t2 where App:t = App t1 t2
    by (metis Cons.prems(3) emb_step_at_if_position emb_step_at_is_App emb_step_equiv' tm.collapse(2))
  then show ?case
  proof (cases a)
    case Left
    have IH_cond1: ¬ list_all (λx. x = dir.Left) p
      using Cons.prems(2) Left by auto
    have IH_cond2: position_of t1 (p @ [d])
      using App Cons.prems(3) Left by auto
    obtain i' where i'_def:
      i' < num_args t1
      args t1 ! i' →emb args (emb_step_at p d t1) ! i'
      ⋀j. j < num_args t1 ⟹ i' ≠ j ⟹ args t1 ! j = args (emb_step_at p d t1) ! j
      using Cons.IH[OF _ IH_cond1 IH_cond2] by blast
    have num_args:num_args (emb_step_at p d t1) = num_args t1
      using App Cons.prems(1) Left emb_step_under_args_num_args
      by (simp add: emb_step_under_args_num_args IH_cond1)
    have 1:i' < num_args t
      by (simp add: App i'_def(1) less_SucI)
    have 2:args t ! i' →emb args (emb_step_at (a # p) d t) ! i'
      by (simp add: App Left i'_def(1) i'_def(2) nth_append num_args)
    have 3:⋀j. j < num_args t ⟹ i' ≠ j ⟹ args t ! j = args (emb_step_at (a # p) d t) ! j
      by (simp add: App Left i'_def(3) nth_append num_args)
    show ?thesis using Cons.prems(1)[OF 1 2 3] by blast
  next
    case Right
    have 0:⋀j. j < num_args t ⟹ num_args t − 1 ≠ j ⟹ args t ! j = args (emb_step_at (a # p) d t) ! j
      by (metis (no_types, lifting) App Cons.prems(2) emb_step_at_right_context Nitpick.size_list_simp(2) Right
```

args.simps(2) butlast_snoc emb_step_under_args_num_args length_butlast length_tl less_antisym nth_butlast)
show *?thesis*
using *Cons.prems(1)*[*of num_args t − 1*]
using *App Cons.prems(3) Right emb_step_at_if_position 0* **by** *auto*
qed
qed

lemma *emb_step_under_args_emb*:
assumes ¬ *list_all* (λ*x*. *x* = *Left*) *p*
position_of t (*p* @ [*d*])
shows
∀ *i*. *i* < *num_args t* ⟶ *args t* ! *i* ⊵$_{emb}$ *args* (*emb_step_at p d t*) ! *i*
by (*metis assms emb.simps emb_step_under_args_emb_step*)

lemma *position_Left_only_subst*:
assumes *list_all* (λ*x*. *x* = *Left*) *p*
and *position_of* (*subst* ϱ *w*) (*p* @ [*d*])
and *num_args* (*subst* ϱ *w*) = *num_args w*
shows *position_of w* (*p* @ [*d*])
using *assms* **proof** (*induct p arbitrary*:*w*)
case *Nil*
then show *?case*
by (*metis position_of_Hd position_of_Nil Hd_head_id append_self_conv2 args.simps(1) length_0_conv list.sel(3) position_of.elims(3)*)
next
case (*Cons a p*)
then show *?case*
proof (*cases w*)
case (*Hd x*)
then show *?thesis*
by (*metis Cons.prems(2) Cons.prems(3) position_of_Hd Hd_head_id append_Cons args.simps(1) list.size(3)*)
next
case (*App w1 w2*)
have *num_args* (*subst* ϱ *w1*) = *num_args w1*
using *App Cons.prems(3)* **by** *auto*
then show *?thesis*
by (*metis* (*full_types*) *App Cons.hyps Cons.prems(1) Cons.prems(2) position_of_left append_Cons list.pred_inject(2) subst.simps(2)*)
qed
qed

2.8 Rearranging embedding steps: first above, then below arguments

lemma *perform_emb_above_vars0*:
assumes
subst ϱ *s* ⊵$_{emb}$ *u*
obtains *w* **where**
s ⊵$_{emb}$ *w*
subst ϱ *w* ⊵$_{emb}$ *u*
∀ *w*′. *w* →$_{emb}$ *w*′ ⟶ ¬ *subst* ϱ *w*′ ⊵$_{emb}$ *u*
using *assms*
proof (*induct s rule*:*measure_induct_rule*[*of hsize*])
case (*less s*)
show *?case*
proof (*cases* ∀ *w*′. *s* →$_{emb}$ *w*′ ⟶ ¬ *subst* ϱ *w*′ ⊵$_{emb}$ *u*)
case *True*
then show *?thesis*
using *emb.refl less.prems(1)*
using *less.prems(2)* **by** *blast*
next
case *False*
then show *?thesis*
by (*meson emb_step_is_emb emb_step_hsize emb_trans less.hyps less.prems(1)*)
qed

qed

lemma *emb_only_below_vars*:
assumes
subst ϱ *s* $\trianglerighteq_{emb}$ *u*
s $\trianglerighteq_{emb}$ *w*
is_Sym (*head w*)
subst ϱ *w* $\trianglerighteq_{emb}$ *u*
$\forall$ *w'*. *w* $\rightarrow_{emb}$ *w'* $\longrightarrow$ $\neg$ *subst* ϱ *w'* $\trianglerighteq_{emb}$ *u*
obtains *ws* **where**
ws $\neq$ []
hd ws = *subst* ϱ *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* (λ*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* ϱ *w*) ! *k* $\trianglerighteq_{emb}$ *args* (*ws* ! *i*) ! *k*)
proof −
define *prio* :: *dir list* $\Rightarrow$ *nat* **where** *prio* = (λ*p*. *if list_all* (λ*x*. *x* = *Left*) (*butlast p*) *then 0 else 1*)
obtain *ws* **where** *ws_def*:
ws $\neq$ []
hd ws = *subst* ϱ *w*
last ws = *u*
$\forall$ *i*. *Suc i* < *length ws* $\longrightarrow$ *prio_emb_step prio* (*ws* ! *i*) *u* = *ws* ! *Suc i* $\wedge$ *ws* ! *i* $\rightarrow_{emb}$ *ws* ! *Suc i*
using ‹*subst* ϱ *w* $\trianglerighteq_{emb}$ *u*› *sequence_prio_emb_steps* **by** *blast*

have *ws_emb_u*:$\forall$ *i*. *Suc i* < *length ws* $\longrightarrow$ *ws* ! *i* $\neq$ *u* $\wedge$ *ws* ! *i* $\trianglerighteq_{emb}$ *u*
proof (*rule allI*)
{
fix *j* **assume** *Suc j* < *length ws*
then have *ws* ! (*length ws* − (*Suc* (*Suc j*))) $\neq$ *u* $\wedge$ *ws* ! (*length ws* − (*Suc* (*Suc j*))) $\trianglerighteq_{emb}$ *u*
proof (*induction j*)
case *0*
have *prio_emb_step prio* (*ws* ! (*length ws* − *Suc* (*Suc 0*))) *u* = *ws* ! (*length ws* − (*Suc 0*))
using *0.prems Suc_diff_Suc ws_def*(*4*) **by** *auto*
then show *?case*
using *0.prems One_nat_def Suc_diff_Suc Suc_lessD diff_Suc_less emb_step_equiv last_conv_nth ws_def*(*1*) *ws_def*(*3*) *ws_def*(*4*)
by (*metis emb_step_is_emb*)
next
case (*Suc j*)
then have *ws* ! (*length ws* − (*Suc* (*Suc* (*Suc j*)))) $\rightarrow_{emb}$ *ws* ! (*length ws* − *Suc* (*Suc j*))
by (*metis Suc_diff_Suc diff_Suc_less length_greater_0_conv ws_def*(*1*) *ws_def*(*4*))
then show *?case* **using** *emb_hsize_neq*
by (*metis Suc.IH Suc.prems Suc_lessD emb_hsize emb_step_is_emb emb_trans leD*)
qed
}
note *0* = *this*
fix *i*
show *Suc i* < *length ws* $\longrightarrow$ *ws* ! *i* $\neq$ *u* $\wedge$ *ws* ! *i* $\trianglerighteq_{emb}$ *u* **using** *0*[*of length ws* − *Suc* (*Suc i*)]
using *Suc_diff_Suc ws_def*(*1*) **by** *auto*
qed

{
fix *i*
assume *i* < *length ws*
then have *head* (*ws* ! *i*) = *head w*
$\wedge$ *num_args* (*ws* ! *i*) = *num_args w*
$\wedge$ ($\forall$ *k*. *k* < *num_args w* $\longrightarrow$ *args* (*subst* ϱ *w*) ! *k* $\trianglerighteq_{emb}$ *args* (*ws* ! *i*) ! *k*)
$\wedge$ (*Suc i* < *length ws* $\longrightarrow$ *prio* (*prio_emb_pos prio* (*ws* ! *i*) *u*) = *1*)
proof (*induct i*)
case *0*
have *1*:*head* (*ws* ! *0*) = *head w*

by (*metis assms(3) ground_imp_subst_iden hd.collapse(2) hd.simps(18) hd_conv_nth head_subst tm.sel(1) tm.simps(17) ws_def(1) ws_def(2)*)
have *2*:*num_args* (*ws* ! *0*) = *num_args w*
by (*metis* (*no_types*, *lifting*) *append_self_conv2 args_subst assms(3) hd.case_eq_if hd_conv_nth length_map ws_def(1) ws_def(2)*)
have *3*:∀ *k*. *k* < *num_args w* ⟶ *args* (*subst* ϱ *w*) ! *k* ⊵$_{emb}$ *args* (*ws* ! *0*) ! *k*
using *emb.refl hd_conv_nth ws_def(1) ws_def(2)* **by** *fastforce*
have *4*:(*Suc 0* < *length ws* ⟶ *prio* (*prio_emb_pos prio* (*ws* ! *0*) *u*) = *1*)
by (*metis* ‹*num_args* (*ws* ! *0*) = *num_args w*› *prio_def append_butlast_last_id assms(5) emb_step_at_if_position*

emb_step_at_subst hd_conv_nth position_Left_only_subst prio_emb_posI ws_def(1) ws_def(2) ws_emb_u)
show *?case* **using** *1 2 3 4* **by** *blast*
next
case (*Suc i*)
then have *ih*:
prio (*prio_emb_pos prio* (*ws* ! *i*) *u*) = *1*
head (*ws* ! *i*) = *head w*
num_args (*ws* ! *i*) = *num_args w* **by** *simp_all*
have *1*:*head* (*ws* ! *Suc i*) = *head w*
by (*metis Suc.prems prio_def emb_step_under_args_head ih(1) ih(2) prio_emb_step_def ws_def(4) zero_neq_one*)
have *2*:*num_args* (*ws* ! *Suc i*) = *num_args w*
by (*metis Suc.prems emb_step_under_args_num_args ih(1) ih(3) prio_def prio_emb_step_def ws_def(4) zero_neq_one*)
have ∀ *k*<*num_args* (*ws* ! *i*). *args* (*ws* ! *i*) ! *k* ⊵$_{emb}$ *args* (*ws* ! *Suc i*) ! *k*
using *Suc.prems emb_step_under_args_emb ih(1) prio_def prio_emb_step_def ws_def(4) zero_neq_one*
by (*metis* (*no_types*, *lifting*) *append_butlast_last_id prio_emb_posI ws_emb_u*)
then have *3*:∀ *k*. *k* < *num_args w* ⟶ *args* (*subst* ϱ *w*) ! *k* ⊵$_{emb}$ *args* (*ws* ! *Suc i*) ! *k*
using *Suc.hyps Suc.prems emb_trans* **by** *force*
have ⋀*p q dp dq*. *prio* (*q* @ [*dq*]) < *prio* (*p* @ [*dp*]) ⟹ *take* (*length p*) *q* ≠ *p* **unfolding** *prio_def*
using *append_take_drop_id list_all_append*
by (*metis* (*mono_tags*, *lifting*) *butlast_snoc gr_implies_not0 less_not_refl2*)
then have *4*:*Suc* (*Suc i*) < *length ws* ⟶ *prio* (*prio_emb_pos prio* (*ws* ! *Suc i*) *u*) = *1*
by (*metis Suc.prems ih(1) not_one_le_zero prio_def prio_emb_pos_increase ws_def(4) ws_emb_u*)
then show *?case* **using** *1 2 3 4* **by** *auto*
qed
}
note *1* = *this*

have ∀ *i*. *Suc i* < *length ws* ⟶
(∃ *p d*. *emb_step_at p d* (*ws* ! *i*) = *ws* ! *Suc i* ∧ ¬ *list_all* (λ*x*. *x* = *Left*) *p*)
by (*metis 1 Suc_lessD prio_def prio_emb_step_def ws_def(4) zero_neq_one*)

then show *?thesis*
using *that ws_def(1) ws_def(2) ws_def(3)*
using *1* **by** *blast*
qed

lemma *perform_emb_above_vars*:
assumes
subst ϱ *s* ⊵$_{emb}$ *u*
obtains *w* **where**
s ⊵$_{emb}$ *w*
subst ϱ *w* ⊵$_{emb}$ *u*
is_Sym (*head w*) ⟹ *head w* = *head u* ∧ *num_args w* = *num_args u* ∧ (∀ *k*. *k*<*num_args w* ⟶ *args* (*subst* ϱ *w*) ! *k* ⊵$_{emb}$ *args u* ! *k*)
proof −
obtain *w* **where** *w_def*:
s ⊵$_{emb}$ *w*
subst ϱ *w* ⊵$_{emb}$ *u*
∀ *w'*. *w* →$_{emb}$ *w'* ⟶ ¬ *subst* ϱ *w'* ⊵$_{emb}$ *u*
using *perform_emb_above_vars0*[*OF* ‹*subst* ϱ *s* ⊵$_{emb}$ *u*›] **by** *metis*
{

assume *is_Sym (head w)*
obtain *ws* **where** *ws_def*:
ws ≠ []
hd ws = *subst ϱ w*
last ws = *u*
∀ *i*. *Suc i* < *length ws* ⟶ (∃ *p d*. *emb_step_at p d* (*ws* ! *i*) = *ws* ! *Suc i* ∧ ¬ *list_all* (λ*x*. *x* = *Left*) *p*)
∀ *i*<*length ws*. *head* (*ws* ! *i*) = *head w* ∧ *num_args* (*ws* ! *i*) = *num_args w*
∀ *i*<*length ws*. ∀ *k*<*num_args w*. *args* (*subst ϱ w*) ! *k* ⊵$_{emb}$ *args* (*ws* ! *i*) ! *k*
using *emb_only_below_vars*[*OF* ‹*subst ϱ s* ⊵$_{emb}$ *u*› ‹*s* ⊵$_{emb}$ *w*› ‹*is_Sym* (*head w*)› ‹*subst ϱ w* ⊵$_{emb}$ *u*› *w_def*(*3*)] **by** *metis*
have *head w* = *head u* ∧ *num_args w* = *num_args u* ∧ (∀ *k*. *k*<*num_args w* ⟶ *args* (*subst ϱ w*) ! *k* ⊵$_{emb}$ *args u* ! *k*)
by (*metis One_nat_def append_butlast_last_id diff_Suc_1 diff_Suc_less length_append_singleton length_greater_0_conv nth_append_length*)

}
then show *?thesis*
using *that w_def*(*1*) *w_def*(*2*) **by** *blast*
qed

end

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

theory *Chop*
imports *Embeddings*
begin

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

3.1 Basic properties

lemma *chop_App_Hd*: *is_Hd s* ⟹ *chop* (*App s t*) = *t*
unfolding *chop_def args.simps*
using *args_Nil_iff_is_Hd* **by** *force*

lemma *chop_apps*: *is_App t* ⟹ *chop* (*apps t ts*) = *apps* (*chop t*) *ts*
unfolding *chop_def* **by** (*simp add*: *args_Nil_iff_is_Hd*)

lemma *vars_chop*: *is_App t* ⟹ *vars* (*chop t*) ∪ *vars_hd* (*head t*) = *vars t*
by (*induct rule*:*tm_induct_apps*; *metis* (*no_types*, *lifting*) *chop_def UN_insert Un_commute list.exhaust_sel list.simps*(*15*)
args_Nil_iff_is_Hd tm.simps(*17*) *tm_exhaust_apps_sel vars_apps*)

lemma *ground_chop*: *is_App t* ⟹ *ground t* ⟹ *ground* (*chop t*)
using *vars_chop* **by** *auto*

lemma *hsize_chop*: *is_App t* ⟹ (*Suc* (*hsize* (*chop t*))) = *hsize t*
unfolding *hsize_args*[*of t*, *symmetric*] *chop_def hsize_apps*
by (*metis Nil_is_map_conv args_Nil_iff_is_Hd list.exhaust_sel list.map_sel*(*1*) *map_tl plus_1_eq_Suc sum_list.Cons*)

lemma *hsize_chop_lt*: *is_App t* ⟹ *hsize* (*chop t*) < *hsize t*
by (*simp add*: *Suc_le_lessD less_or_eq_imp_le hsize_chop*)

lemma *chop_fun*:
assumes *is_App t is_App* (*fun t*)
shows *App* (*chop* (*fun t*)) (*arg t*) = *chop t*
proof −
have *args* (*fun t*) ≠ []
using *assms*(*2*) *args_Nil_iff_is_Hd* **by** *blast*
then show *?thesis*

unfolding *chop_def*
using *assms*(*1*) **by** (*metis* (*no_types*) *App_apps args.simps*(*2*) *hd_append2 tl_append2 tm.collapse*(*2*))
qed

3.2 Chop and the Embedding Relation

lemma *emb_step_chop*: *is_App t* $\Longrightarrow$ *t* $\rightarrow_{emb}$ *chop t*
proof (*induct num_args t − 1 arbitrary*:*t*)
case *0*
have *nil*: *num_args t* = *0* $\Longrightarrow$ *t* $\rightarrow_{emb}$ *chop t* **unfolding** *chop_def* **using** *0 args_Nil_iff_is_Hd* **by** *force*
have *single*: $\bigwedge$*a*. *args t* = [*a*] $\Longrightarrow$ *t* $\rightarrow_{emb}$ *chop t* **unfolding** *chop_def*
by (*metis 0.prems apps.simps*(*1*) *args.elims args_Nil_iff_is_Hd emb_step_arg last.simps last_snoc list.sel*(*1*) *list.sel*(*3*) *tm.sel*(*6*))
then show *?case* **using** *nil single*
by (*metis 0.hyps length_0_conv length_tl list.exhaust_sel*)
next
case (*Suc x*)
have *1*:*apps* (*Hd* (*head t*)) (*butlast* (*args t*)) $\rightarrow_{emb}$ *chop* (*apps* (*Hd* (*head t*)) (*butlast* (*args t*)))
using *Suc*(*1*)[*of apps* (*Hd* (*head t*)) (*butlast* (*args t*))]
by (*metis Suc.hyps*(*2*) *Suc_eq_plus1 add_diff_cancel_right*′ *args_Nil_iff_is_Hd length_butlast list.size*(*3*) *nat.distinct*(*1*) *tm_exhaust_apps_sel tm_inject_apps*)
have *2*:*App* (*apps* (*Hd* (*head t*)) (*butlast* (*args t*))) (*last* (*args t*)) = *t*
by (*simp add*: *App_apps Suc.prems args_Nil_iff_is_Hd*)
have *3*:*App* (*chop* (*apps* (*Hd* (*head t*)) (*butlast* (*args t*)))) (*last* (*args t*)) = *chop t*
proof −
have *f1*: *hd* (*args t*) = *hd* (*butlast* (*args t*))
by (*metis Suc.hyps*(*2*) *Suc.prems append_butlast_last_id args_Nil_iff_is_Hd hd_append2 length_0_conv length_butlast nat.simps*(*3*))
have *tl* (*args t*) = *tl* (*butlast* (*args t*)) @ [*last* (*args t*)]
by (*metis* (*no_types*) *Suc.hyps*(*2*) *Suc.prems append_butlast_last_id args_Nil_iff_is_Hd length_0_conv length_butlast nat.simps*(*3*) *tl_append2*)
then show *?thesis*
using *f1 chop_def*
by (*metis App_apps append_Nil args.simps*(*1*) *args_apps*)
qed
then show *?case* **using** *1 2 3* **by** (*metis context_left*)
qed

lemma *chop_emb_step_at*:
assumes *is_App t*
shows *chop t* = *emb_step_at* (*replicate* (*num_args* (*fun t*)) *Left*) *Right t*
using *assms* **proof** (*induct num_args* (*fun t*) *arbitrary*: *t*)
case *0*
then have *rep_Nil*:*replicate* (*num_args* (*fun t*)) *dir.Left* = []
by *simp*
then show *?case* **unfolding** *rep_Nil*
by (*metis 0.hyps 0.prems emb_step_at_right append_Nil apps.simps*(*1*) *args.simps*(*2*) *chop_def length_0_conv list.sel*(*1*) *list.sel*(*3*) *tm.collapse*(*2*))
next
case (*Suc n*)
then show *?case* **using** *Suc.hyps*(*1*)[*of fun t*]
using *emb_step_at_left_context args.elims args_Nil_iff_is_Hd chop_fun butlast_snoc diff_Suc_1 length_0_conv length_butlast nat.distinct*(*1*) *replicate_Suc tm.collapse*(*2*) *tm.sel*(*4*)
by *metis*
qed

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* (λ*x*. *x* = *Left*) *p*
shows *chop t* = *s* ∨ *chop t* $\rightarrow_{emb}$ *s*
proof −
have *is_App t*
by (*metis emb_step_at_if_position emb_step_at_is_App emb_step_equiv pos*)

```
  have p_replicate: replicate (length p) Left = p using replicate_length_same[of p Left]
    by (simp add: Ball_set all_Left)
  show ?thesis
  proof (cases Suc (length p) = num_args t)
    case True
    then have p = replicate (num_args (fun t)) Left using p_replicate
      by (metis Suc_inject ‹is_App t› args.elims args_Nil_iff_is_Hd length_append_singleton tm.sel(4))
    then have chop t = s unfolding chop_emb_step_at[OF ‹is_App t›]
      using pos emb_step_at by blast
    then show ?thesis by blast
  next
    case False
    then have Suc (length p) < num_args t
      using pos emb_step_at ‹is_App t› ‹list_all (λx. x = dir.Left) p›
    proof (induct p arbitrary:t s)
      case Nil
      then show ?case
        by (metis Suc_lessI args_Nil_iff_is_Hd length_greater_0_conv list.size(3))
    next
      case (Cons a p)
      have a = Left
        using Cons.prems(5) by auto
      have 1:Suc (length p) ≠ num_args (fun t)
        by (metis (no_types, lifting) Cons.prems(1) Cons.prems(4) args.elims args_Nil_iff_is_Hd length_Cons
length_append_singleton tm.sel(4))
      have 2:position_of (fun t) (p @ [Right])  using ‹position_of t ((a # p) @ [Right])› ‹is_App t›
        by (metis (full_types) Cons.prems(5) position_of_left append_Cons list_all_simps(1) tm.collapse(2))
      have 3: emb_step_at p dir.Right (fun t) = emb_step_at p dir.Right (fun t)
        using emb_step_at_left_context[of p Right fun t arg t] by blast
      have Suc (length p) < num_args (fun t) using Cons.hyps[OF 1 2 3]
        by (metis 2 Cons.prems(5) Nil_is_append_conv list_all_simps(1) not_Cons_self2 position_of.elims(2)
tm.discI(2))
      then show ?case
        by (metis Cons.prems(4) Suc_less_eq2 args.simps(2) length_Cons length_append_singleton tm.collapse(2))
    qed
    define q where q = replicate (num_args (fun t) − Suc (length p)) dir.Left
    have chop t = emb_step_at (p @ [Left] @ q) dir.Right t
    proof −
      have length p + (num_args (fun t) − length p) = num_args (fun t)
        using ‹Suc (length p) < num_args t›
        by (metis Suc_less_eq2 ‹is_App t› args.simps(2) diff_Suc_1 leD length_butlast nat_le_linear
            ordered_cancel_comm_monoid_diff_class.add_diff_inverse snoc_eq_iff_butlast tm.collapse(2))
      then have 1:replicate (num_args (fun t)) dir.Left = p @ replicate (num_args (fun t) − length p) dir.Left
        by (metis p_replicate replicate_add)
      have 0 < num_args (fun t) − length p
        by (metis (no_types) False ‹is_App t› ‹length p + (num_args (fun t) − length p) = num_args (fun t)›
args.simps(2) length_append_singleton less_Suc_eq less_add_Suc1 tm.collapse(2) zero_less_diff)
      then have replicate (num_args (fun t) − length p) dir.Left = [Left] @ q unfolding q_def
        by (metis (no_types) Cons_replicate_eq Nat.diff_cancel Suc_eq_plus1 ‹length p + (num_args (fun t) −
length p) = num_args (fun t)› append_Cons self_append_conv2)
      then show ?thesis using chop_emb_step_at
        using ‹is_App t› 1
        by (simp add: chop_emb_step_at)
    qed
    then have chop t →emb s
      using pos merge_emb_step_at[of p Right q Right t]
      by (metis emb_step_at_if_position opp_simps(1) emb_step_at pos_emb_step_at_opp)
    then show ?thesis by blast
  qed
qed

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 (λx. x = Left) p
  shows let i = num_args t − Suc (length p) in
  head t = head s ∧ i < num_args t ∧ args s = take i (args t) @ drop (Suc i) (args t)
proof −
  have is_App t
    by (metis emb_step_at_if_position emb_step_at_is_App emb_step_equiv pos)
  have C1: head t = head s
    using all_Left emb_step_at pos proof (induct p arbitrary:s t)
    case Nil
    then have s = emb_step_at [] dir.Left (App (fun t) (arg t))
      by (metis position_of.elims(2) snoc_eq_iff_butlast tm.collapse(2) tm.discI(2))
    then have s = fun t by simp
    then show ?case by simp
  next
    case (Cons a p)
    then have a = Left by simp
    then have head (emb_step_at p Left (fun t)) = head t
      by (metis Cons.hyps Cons.prems(1) head_fun list.pred_inject(2) position_if_emb_step_at)
    then show ?case using emb_step_at_left_context[of p a fun t arg t]
      by (metis Cons.prems(2) ‹a = Left› emb_step_at_is_App head_App tm.collapse(2))
  qed

  let ?i = num_args t − Suc (length p)

  have C2:?i < num_args t
    by (simp add: ‹is_App t› args_Nil_iff_is_Hd)

  have C3:args s = take ?i (args t) @ drop (Suc ?i) (args t)
    using all_Left pos emb_step_at ‹is_App t› proof (induct p arbitrary:s t)
    case Nil
    then show ?case using emb_step_at_left[of fun t arg t]
      by (simp, metis One_nat_def args.simps(2) butlast_conv_take butlast_snoc tm.collapse(2))
  next
    case (Cons a p)
    have position_of (fun t) (p @ [Left])
      by (metis (full_types) Cons.prems(1) Cons.prems(2) Cons.prems(4) position_of_left
          append_Cons list.pred_inject(2) tm.collapse(2))
    then have 0:args (emb_step_at p Left (fun t))
                = take (num_args (fun t) − Suc (length p)) (args (fun t))
                @ drop (Suc (num_args (fun t) − Suc (length p))) (args (fun t))
      using Cons.hyps[of fun t] by (metis Cons.prems(1) append_Nil args_Nil_iff_is_Hd drop_Nil
          emb_step_at_is_App list.size(3) list_all_simps(1) take_0 zero_diff)
    have 1:s = App (emb_step_at p Left (fun t)) (arg t) using emb_step_at_left_context[of p Left fun t arg t]
      using Cons.prems by auto
    define k where k_def:k = (num_args (fun t) − Suc (length p))
    have 2:take k (args (fun t)) = take (num_args t − Suc (length (a # p))) (args t)
      by (metis (no_types, lifting) Cons.prems(4) args.elims args_Nil_iff_is_Hd butlast_snoc
          diff_Suc_eq_diff_pred diff_less k_def length_Cons length_butlast length_greater_0_conv
          take_butlast tm.sel(4) zero_less_Suc)
    have k_def': k = num_args t − Suc (Suc (length p)) using k_def
      by (metis args.simps(2) diff_Suc_Suc length_append_singleton local.Cons(5) tm.collapse(2))
    have 3:args (fun t) @ [arg t] = args t
      by (metis Cons.prems(4) args.simps(2) tm.collapse(2))
    have num_args t > 1 using ‹position_of t ((a # p) @ [Left])›
    by (metis 3 ‹position_of (fun t) (p @ [dir.Left])› args_Nil_iff_is_Hd butlast_snoc emb_step.simps emb_step_at_if_position
length_butlast length_greater_0_conv tm.discI(2) zero_less_diff)
    then have Suc k<num_args t unfolding k_def'
      using ‹1 < num_args t› by linarith
    have ∀k< num_args t. drop k (args (fun t)) @ [arg t] = drop k (args t)
       by (metis (no_types, lifting) ‹args (fun t) @ [arg t] = args t› drop_butlast drop_eq_Nil last_drop leD
snoc_eq_iff_butlast)
    then show ?case using 0 1 2 3 k_def'
```

```
    using ‹Suc k < num_args t› k_def by auto
  qed
  show ?thesis using C1 C2 C3 by simp
qed

lemma emb_step_cases [consumes 1, case_names chop extended_chop remove_arg under_arg]:
  assumes emb:t →emb s
    and chop:chop t = s ⟹ P
    and extended_chop:chop t →emb s ⟹ P
    and remove_arg:⋀i. head t = head s ⟹ i<num_args t ⟹ args s = take i (args t) @ drop (Suc i) (args t) ⟹
P
    and under_arg:⋀i. head t = head s ⟹ num_args t = num_args s ⟹ args t ! i →emb args s ! i ⟹
        (⋀j. j<num_args t ⟹ i ≠ j ⟹ args t ! j = args s ! j) ⟹ P
  shows P
proof −
  obtain p d where pd_def:emb_step_at p d t = s position_of t (p @ [d])
    using emb emb_step_equiv′ position_if_emb_step_at by metis
  have is_App t
    by (metis emb emb_step_at_is_App emb_step_equiv)
  show ?thesis
  proof (cases list_all (λx. x = Left) p)
    case True
    show ?thesis
    proof (cases d)
      case Left
      then show P using emb_step_at_remove_arg
        by (metis True pd_def(1) pd_def(2) remove_arg)
    next
      case Right
      then show P
        using True chop emb_step_at_chop extended_chop pd_def(1) pd_def(2) by blast
    qed
  next
    case False
    have 1:num_args t = num_args s using emb_step_under_args_num_args
      by (metis False pd_def(1))
    have 2:head t = head s using emb_step_under_args_head
      by (metis False pd_def(1))
    show ?thesis using 1 2 under_arg emb_step_under_args_emb_step
      by (metis False pd_def(1) pd_def(2))
  qed
qed

lemma chop_position_of:
  assumes is_App s
  shows position_of s (replicate (num_args (fun s)) dir.Left @ [Right])
 by (metis Suc_n_not_le_n assms chop_emb_step_at lessI less_imp_le_nat position_if_emb_step_at hsize_chop)
```

3.3 Chop and Substitutions

```
lemma Suc_num_args: is_App t ⟹ Suc (num_args (fun t)) = num_args t
  by (metis args.simps(2) length_append_singleton tm.collapse(2))


lemma fun_subst: is_App s ⟹ subst ϱ (fun s) = fun (subst ϱ s)
  by (metis subst.simps(2) tm.collapse(2) tm.sel(4))


lemma args_subst_Hd:
  assumes is_Hd (subst ϱ (Hd (head s)))
  shows  args (subst ϱ s) = map (subst ϱ) (args s)
  using assms
  by (metis append_Nil args_Nil_iff_is_Hd args_apps subst_apps tm_exhaust_apps_sel)
```

lemma *chop_subst_emb0*:
assumes *is_App s*
assumes *chop (subst ϱ s) ≠ subst ϱ (chop s)*
shows *emb_step_at (replicate (num_args (fun s)) Left) Right (chop (subst ϱ s)) = subst ϱ (chop s)*
proof −
have *is_App (subst ϱ s)*
by (*metis assms(1) subst.simps(2) tm.collapse(2) tm.disc(2)*)
have *1*:*subst ϱ (chop s) = emb_step_at (replicate (num_args (fun s)) dir.Left) dir.Right (subst ϱ s)*
using *chop_emb_step_at*[*OF assms(1)*] **using** *emb_step_at_subst chop_position_of*[*OF ‹is_App s›*]
by (*metis*)
have *num_args (fun s) ≤ num_args (fun (subst ϱ s))*
using *fun_subst*[*OF ‹is_App s›*]
by (*metis args_subst leI length_append length_map not_add_less2*)
then have *num_args (fun s) < num_args (fun (subst ϱ s))*
using *assms(2) 1 ‹is_App (subst ϱ s)› chop_emb_step_at le_imp_less_or_eq*
by *fastforce*
then have *num_args s ≤ num_args (fun (subst ϱ s))*
using *Suc_num_args*[*OF ‹is_App s›*] **by** *linarith*
then have *replicate (num_args (fun s)) dir.Left @*
[*opp dir.Right*] *@ replicate (num_args (fun (subst ϱ s))) − num_args s) dir.Left =*
replicate (num_args (fun (subst ϱ s))) dir.Left
unfolding *append.simps opp_simps replicate_Suc*[*symmetric*] *replicate_add*[*symmetric*]
using *Suc_num_args*[*OF ‹is_App s›*]
by (*metis add_Suc_shift ordered_cancel_comm_monoid_diff_class.add_diff_inverse*)
then show *?thesis* **unfolding** *1*
unfolding *chop_emb_step_at*[*OF ‹is_App (subst ϱ s)›*]
by (*metis merge_emb_step_at*)
qed

lemma *chop_subst_emb*:
assumes *is_App s*
shows *chop (subst ϱ s)* $\trianglerighteq_{emb}$ *subst ϱ (chop s)*
using *chop_subst_emb0*
by (*metis assms emb.refl emb_step_equiv emb_step_is_emb*)

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 −
have *is_App (subst ϱ s)*
by (*metis assms(1) subst.simps(2) tm.collapse(2) tm.disc(2)*)
have *num_args (fun s) = num_args (fun (subst ϱ s))* **unfolding** *fun_subst*[*OF ‹is_App s›,symmetric*]
using *args_subst_Hd* **using** *assms(2)* **by** *auto*
then show *?thesis*
unfolding *chop_emb_step_at*[*OF assms(1)*] *chop_emb_step_at*[*OF ‹is_App (subst ϱ s)›*]
using *emb_step_at_subst*[*OF chop_position_of*[*OF ‹is_App s›*]]
by *simp*
qed

lemma *chop_subst_Sym*:
assumes *is_App s*
assumes *is_Sym (head s)*
shows *chop (subst ϱ s) = subst ϱ (chop s)*
by (*metis assms(1) assms(2) chop_subst_Hd ground_imp_subst_iden hd.collapse(2) hd.simps(18) tm.disc(1) tm.simps(17)*)

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*)
by (*rule ext_trans_before_irrefl.axioms*(*1*)[*OF extf_ext_trans_before_irrefl*])

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

lemmas *extf_mono_strong* = *ext.mono_strong*[*OF extf_ext*]
lemmas *extf_mono* = *ext.mono*[*OF extf_ext*, *mono*]
lemmas *extf_map* = *ext.map*[*OF extf_ext*]
lemmas *extf_trans* = *ext_trans.trans*[*OF extf_ext_trans*]
lemmas *extf_irrefl_from_trans* =
ext_trans_before_irrefl.irrefl_from_trans[*OF extf_ext_trans_before_irrefl*]
lemmas *extf_compat_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 (*induction xs*)
case *Nil*
then show *?case*
using *extf_min_empty* **by** *force*
next
case (*Cons a xs*)
then show *?case*
by (*simp add*: *extf_compat_cons*)
qed

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* $\vee$ *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* ⟷
(*if is_Var* (*head t*)
then is_App t ∧ *chkchop gt* (*chop t*) *s*
else chkchop gt t s)

lemma *chkchop_mono*[*mono*]: *gt* ≤ *gt′* ⟹ *chkchop gt* ≤ *chkchop gt′*
using *chkchop_def* **by** *blast*

lemma *chkchop_same_mono*[*mono*]: *gt* ≤ *gt′* ⟹ *chkchop_same gt* ≤ *chkchop_same gt′*
using *chkchop_same_def* **by** *fastforce*

inductive *gt* :: ('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm* ⇒ *bool* (**infix** >$_t$ *50*) **where**
gt_chop: *is_App t* ⟹ *chop t* >$_t$ *s* ∨ *chop t* = *s* ⟹ *t* >$_t$ *s*
| *gt_diff*: *head t* >$_{hd}$ *head s* ⟹ *is_Sym* (*head s*) ⟹ *chkchop* (>$_t$) *t s* ⟹ *t* >$_t$ *s*
| *gt_same*: *head t* = *head s* ⟹ *chkchop_same* (>$_t$) *t s* ⟹
(∀*f* ∈ *ground_heads* (*head t*). *extf f* (>$_t$) (*args t*) (*args s*)) ⟹ *t* >$_t$ *s*

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

inductive *gt_chop* :: ('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm* ⇒ *bool* **where**
gt_chopI: *is_App t* ⟹ *chop t* ≥$_t$ *s* ⟹ *gt_chop t s*

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

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

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

4.3 Transitivity

lemma *t_gt_chop_t*: *is_App t* ⟹ *t* >$_t$ *chop t*
by (*simp add*: *gt_chop*)

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

show *u* >$_t$ *s*
using *u_gt_t*
proof *cases*
case *gt_chop*
then have *chop u* >$_t$ *s* **using** *ih*[*of chop u t s*]
using *add_mset_lt_left_lt hsize_chop_lt t_gt_s* **by** *blast*
show *?thesis*
using *local.gt_chop*(*1*) *local.gt_chop*(*2*) ‹*chop u* >$_t$ *s*› *gt.gt_chop* **by** *presburger*
next
case *gt_diff_u_t*: *gt_diff*
show *?thesis*
using *t_gt_s*
proof *cases*
case *gt_chop*

```
    then have u >_t chop t
      using chkchop_def gt_diff_u_t(3) by presburger
    then show ?thesis using ih[of u chop t s]
      using hsize_chop_lt local.gt_chop(1) local.gt_chop(2) by fastforce
  next
    case gt_diff_t_s: gt_diff
    have head u >_hd head s
      using gt_diff_u_t(1) gt_diff_t_s(1) by (auto intro: gt_hd_trans)
    thus ?thesis using ih[of u t chop s]
      by (metis add_mset_lt_left_lt add_mset_lt_right_lt chkchop_def gt_diff gt_diff_t_s(2)
        gt_diff_t_s(3) hsize_chop_lt u_gt_t)
  next
    case gt_same_t_s: gt_same
    have head u >_hd head s
      using gt_diff_u_t(1) gt_same_t_s(1) by auto
    thus ?thesis using ih[of u t chop s]
      by (metis add_mset_lt_left_lt add_mset_lt_right_lt chkchop_def chkchop_same_def gt_diff
        gt_diff_u_t(2) gt_same_t_s(1) gt_same_t_s(2) hsize_chop_lt u_gt_t)
  qed
next
  case gt_same_u_t: gt_same
  show ?thesis
    using t_gt_s
  proof cases
    case gt_chop
    then show ?thesis
    proof (cases is_Var (head u))
      case True
      then show ?thesis  using ih[of chop u chop t s]
        by (metis add_mset_lt_left_lt add_mset_lt_lt_lt chkchop_def chkchop_same_def gt.gt_chop
          gt_same_u_t(2) hsize_chop_lt local.gt_chop(1) local.gt_chop(2))
    next
      case False
      then show ?thesis using ih[of u chop t s]
        by (metis add_mset_lt_left_lt add_mset_lt_right_lt chkchop_def chkchop_same_def
          gt_same_u_t(2) hsize_chop_lt local.gt_chop(1) local.gt_chop(2))
    qed
  next
    case gt_diff_t_s: gt_diff
    have head u >_hd head s
      by (simp add: gt_diff_t_s(1) gt_same_u_t(1))
    thus ?thesis using ih[of u t chop s]
      by (metis add_mset_lt_left_lt add_mset_lt_right_lt chkchop_def gt_diff gt_diff_t_s(1)
        gt_diff_t_s(2) gt_diff_t_s(3) gt_same_u_t(1) hsize_chop_lt  u_gt_t)
  next
    case gt_same_t_s: gt_same
    have hd_u_s: head u = head s
      using gt_same_u_t(1) gt_same_t_s(1) by simp

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

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

```
    have ∀f ∈ ground_heads (head u). extf f (>t) (args u) (args s)
    proof (clarify, rule extf_trans[OF _ _ _ gt_trans_args])
      fix f
      assume f_in_grounds: f ∈ ground_heads (head u)
      show extf f (>t) (args u) (args t)
        using f_in_grounds gt_same_u_t(3) by blast
      show extf f (>t) (args t) (args s)
        using f_in_grounds gt_same_t_s(3) unfolding gt_same_u_t(1) by blast
    qed auto
    have chkchop_same (>t) u s
    proof (cases head u)
      case (Var x)
      then show ?thesis
      proof (cases u)
        case (Hd _)
        then show ?thesis
          using Var
          using gt_same_u_t(2) by force
      next
        case (App u1 u2)
        then have chop u >t chop t
          by (metis Var chkchop_def chkchop_same_def gt_same_t_s(2) gt_same_u_t(1)
            gt_same_u_t(2) hd.disc(1))
        then show ?thesis
        proof (cases t)
          case (Hd _)
          then show ?thesis
            using Var gt_same_t_s(2) gt_same_u_t(1) by force
        next
          case t_App: (App t1 t2)
          then have is_App s ⟹ chop t >t chop s
           using gt_same_t_s unfolding chkchop_same_def unfolding chkchop_def using Var hd_u_s by auto
          then have chkchop (>t) (chop u) s
            unfolding chkchop_def using ih[of chop u chop t chop s]
         by (metis App ‹chop u >t chop t› t_App add_mset_lt_lt_le less_imp_le mset_lt_single_iff hsize_chop_lt
tm.disc(2))
          then show ?thesis unfolding chkchop_same_def
            using Var by (simp add: App)
        qed
      qed
    next
      case (Sym f)
      have chkchop (>t) u s
       proof (cases s)
         case (Hd _)
         then show ?thesis
           by simp
       next
         case (App s1 s2)
         then have t >t chop s
           using Sym gt_same_t_s(1) gt_same_t_s(2) hd_u_s by auto
         then have u >t chop s using ih[of u t chop s]
           by (metis App add_mset_lt_right_lt mset_lt_single_iff hsize_chop_lt tm.disc(2) u_gt_t)
         then show ?thesis unfolding chkchop_def
           by blast
       qed
      then show ?thesis
        by (simp add: Sym)
    qed
    thus ?thesis
      using ‹∀f∈local.ground_heads (head u). extf f (>t) (args u) (args s)› gt_same hd_u_s by blast
  qed
 qed
```

qed

4.4 Irreflexivity

theorem *gt_irrefl*: $\neg\ s >_t s$
proof (*standard*, *induct s rule*: *measure_induct_rule*[*of hsize*])
  case (*less s*)
  note *ih* = *this*(*1*) **and** *s_gt_s* = *this*(*2*)

  show *False*
    using *s_gt_s*
  proof *cases*
    case *gt_chop*
    then show *False* **using** *ih*[*of chop s*]
      by (*metis gt.gt_chop gt_trans hsize_chop_lt*)
  next
    case *gt_diff*
    thus *False*
      by (*cases head s*) (*auto simp*: *gt_hd_irrefl*)
  next
    case *gt_same*
    note *in_grounds* = *this*(*3*)

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

lemma *gt_antisym*: $t >_t s \Longrightarrow \neg\ s >_t t$
  using *gt_irrefl gt_trans* **by** *blast*

4.5 Subterm Property

lemma *gt_emb_fun*: *App s t* $>_t$ *s*
proof (*induction s rule*:*measure_induct_rule*[*of hsize*])
  case (*less s*)
  have *extf*: ∀ *f* ∈ *ground_heads* (*head s*). *extf f* ($>_t$) (*args* (*App s t*)) (*args s*)
    using *extf_snoc* **by** *force*
  have *head* (*App s t*) = *head s*
    by *simp*
  have *chkchop_same* ($>_t$) (*App s t*) *s*
  proof (*cases is_Hd s*)
    case *True*
    then show *?thesis*
      by *simp*
  next
    case *False*
    have *chop_gt_chop*: (*chop* (*App s t*)) $>_t$ *chop s* **using** *less*[*of chop s*]
      using *False hsize_chop_lt* **by** (*metis App_apps apps.simps*(*1*) *chop_apps*)
    then show *?thesis*
    proof (*cases is_Var* (*head s*))
      case *True*
      then show *?thesis*
        by (*simp add*: *True chop_gt_chop*)
    next
      case *False*
      then show *?thesis* **using** *less*[*of chop s*]
        by (*simp add*: *chop_gt_chop gt_chop*)
    qed

```
  qed
  then show ?case
    using extf gt_same by auto
qed

lemma gt_emb_arg: App s t >t t
proof (induction s rule:measure_induct_rule[of hsize])
  case (less s)
  then show ?case
  proof (cases is_Hd s)
    case True
    then show ?thesis
      by (metis chop_App_Hd t_gt_chop_t tm.disc(2))
  next
    case False
    have chop (App s t) >t t using less[of chop s]
      by (metis App_apps False apps.simps(1) chop_apps hsize_chop_lt)
    then show ?thesis
      using gt_chop tm.disc(2) by blast
  qed
qed
```

4.6 Compatibility with Contexts

```
lemma gt_compat_fun:
  assumes t' >t t
  shows App s t' >t App s t
using assms  apply (simp only:atomize_imp)
proof (induction rule:measure_induct_rule[of λ(t, s). hsize t + hsize s λ(t, s). t' >t t ⟶ App s t' >t App s t
(t,s),
  simplified prod.case],
  simp only: split_paired_all prod.case atomize_imp[symmetric])

  fix t s ::('s, 'v) tm
  assume ih:⋀ta sa. hsize ta + hsize sa < hsize t + hsize s ⟹ t' >t ta ⟹ App sa t' >t App sa ta
  and t'_gt_t:t' >t t

  have t'_ne_t: t' ≠ t
    using gt_antisym t'_gt_t by blast
  have extf_args_single: ∀f ∈ ground_heads (head s). extf f (>t) (args s @ [t']) (args s @ [t])
    by (simp add: extf_compat_list t'_gt_t t'_ne_t)

  show  App s t' >t App s t
  proof (rule gt_same)
    show head (App s t') = head (App s t) by simp
    show ∀f∈local.ground_heads (head (App s t')). extf f (>t) (args (App s t')) (args (App s t))
      by (simp add: extf_args_single)
    have 0: chop (App s t') >t chop (App s t)
    proof (cases s)
      case (Hd _)
      then show ?thesis using chop_App_Hd
        by (simp add: chop_App_Hd t'_gt_t)
    next
      case (App s1 s2)
      then show ?thesis using ih[of t chop s] chop_fun
        by (metis nat_add_left_cancel_less hsize_chop_lt t'_gt_t tm.disc(2) tm.sel(4) tm.sel(6))
    qed
    show chkchop_same (>t) (App s t') (App s t)
    proof (cases is_Var (head (App s t')))
      case True
      then show ?thesis unfolding chkchop_same_def chkchop_def
        using True 0 by auto
    next
      case False
```

```
    have App s t' >t chop (App s t) using 0 by (simp add: gt_chop)
    then show ?thesis  unfolding chkchop_same_def chkchop_def
      using False by auto
  qed
 qed
qed

theorem gt_compat_arg:
 shows s' >t s ⟹ t' ≥t t ⟹ App s' t' >t App s t
proof (simp only:atomize_imp,induction rule:measure_induct[of λ(s',s,t). hsize s' + hsize s + hsize t λ(s',s,t). s'
>t s ⟶ t' ≥t t ⟶ App s' t' >t App s t (s',s,t), simplified prod.case],
   simp only: split_paired_all prod.case atomize_imp[symmetric] atomize_all[symmetric])
 fix s' s t
 assume ih:⋀ab ac ba. hsize ab + hsize ac + hsize ba < hsize s' + hsize s + hsize t ⟹ ab >t ac ⟹ t' ≥t ba ⟹
App ab t' >t App ac ba
   and s' >t s and t' ≥t t

 {
  fix s''::('s,'v) tm assume hsize_s'':hsize s'' ≤ hsize s'
  assume chkchop_s'_s: chkchop (>t) s'' s
  then have chkchop (>t) (App s'' t') (App s t)
  proof (cases is_Hd s)
    case True
    then show ?thesis using chkchop_s'_s unfolding chkchop_def
      by (metis ‹t' ≥t t› chop_App_Hd gt_emb_arg gt_trans)
  next
    case False
    then show ?thesis using chkchop_s'_s unfolding chkchop_def
      using ih[of s'' chop s t] hsize_s''
        by (metis ‹t' ≥t t› add_less_mono add_mono_thms_linordered_field(1) chop_fun le_eq_less_or_eq
nat_add_left_cancel_less hsize_chop_lt tm.sel(4) tm.sel(6))
  qed
 }
 note chkchop_compat_arg = this

 show App s' t' >t App s t  using ‹s' >t s›
 proof (cases rule:gt.cases)
  case gt_chop
  then show ?thesis
  proof (cases t = t')
    case True
    show ?thesis using True gt.gt_chop[of App s' t' App s t] gt_chop chkchop_compat_arg[of chop s']
      by (metis add_strict_right_mono chop_fun ih hsize_chop_lt tm.disc(2) tm.sel(4) tm.sel(6))
  next
    case False
    then have t' >t t
      using ‹t' ≥t t› by blast
    have App s' t' >t App (chop s') t'
      by (metis chop_fun local.gt_chop(1) t_gt_chop_t tm.disc(2) tm.sel(4) tm.sel(6))
    moreover have ... >t App s t  using ih[of chop s' s t]
      using ‹t' >t t› gt_compat_fun local.gt_chop(1) local.gt_chop(2) hsize_chop_lt by fastforce
    ultimately show ?thesis using gt_trans by blast
  qed
 next
  case gt_diff
  then show ?thesis
    using chkchop_compat_arg gt.gt_diff by auto
 next
  case gt_same
  have hd_s'_eq_s: head s' = head s
    by (simp add: local.gt_same(1))
  {
    fix f assume f_gh: f ∈ ground_heads (head s)
```

```
    have f_s_args:
      extf f (>t) (args s') (args s)
      using local.gt_same(3) f_gh by (simp add: hd_s'_eq_s)
    have f_compat_snoc:
      ⋀xs ys x. extf f (>t) ys xs ⟹ extf f (>t) (ys @ [x]) (xs @ [x])
      by (simp add: extf_compat_append_right)

    have f_st_args2:
      extf f (>t) (args (App s' t)) (args (App s t))
      by (simp add: f_compat_snoc f_s_args)
    have 0:∀ z∈UNIV. ∀ y∈UNIV. ∀ x∈UNIV. z >t y ⟶ y >t x ⟶ z >t x
      using gt_trans by blast
    then have f_trans:⋀xs ys zs. extf f (>t) zs ys ⟹ extf f (>t) ys xs ⟹ extf f (>t) zs xs
      using  extf_trans[of _ UNIV, unfolded lists_UNIV, OF UNIV_I UNIV_I UNIV_I 0] by metis
    have extf f (>t) (args (App s' t')) (args (App s t))
    proof (cases t' = t)
      case True
      then show ?thesis using f_st_args2 by metis
    next
      case False
      have f_st_args1:
        extf f (>t) (args (App s' t')) (args (App s' t))
        using extf_compat_list ‹t' ≥t t› False by simp
      then show ?thesis using f_trans f_st_args1 f_st_args2 by metis
    qed
  }
  note extf_cond = this
  have chkchop_same (>t) (App s' t') (App s t) unfolding chkchop_same_def
    using chop_fun chkchop_compat_arg[of chop s', unfolded le_eq_less_or_eq]
    chkchop_compat_arg[of s'] chkchop_def chkchop_same_def
    hsize_chop_lt  epo.extf_min_empty[OF epo_axioms] gt.gt_same gt_antisym hd_s'_eq_s head_App
    leI less_irrefl_nat local.gt_same(2) local.gt_same(3) tm.sel(4) tm.sel(6) tm.disc(2)
    by metis
  then show ?thesis
    using extf_cond gt.gt_same hd_s'_eq_s by auto
 qed
qed

theorem gt_compat_fun_strong:
 assumes t'_gt_t: t' >t t
 shows apps s (t' # us) >t apps s (t # us)
proof (induct us rule: rev_induct)
 case Nil
 then show ?case
 by (simp add: gt_compat_fun t'_gt_t)
next
 case (snoc x xs)
 then show ?case unfolding App_apps[symmetric] append_Cons[symmetric]
  using gt_compat_arg by blast
qed

theorem gt_or_eq_compat_App: s' ≥t s ⟹ t' ≥t t ⟹ App s' t' ≥t App s t
 using gt_compat_fun gt_compat_arg by blast

theorem gt_compat_App:
 shows s' ≥t s ⟹ t' >t t ⟹ App s' t' >t App s t
 using gt_compat_fun gt_compat_arg by blast
```

4.7 Compatibility with Embedding Relation

```
lemma gt_embedding_step_property:
 assumes t →emb s
 shows t >t s
using assms proof (induct)
```

case (*left t1 t2*)
then show *?case*
using *gt_emb_fun* **by** *blast*
next
case (*right t1 t2*)
then show *?case*
using *gt_emb_arg* **by** *blast*
next
case (*context_left t s u*)
then show *?case*
using *gt_compat_arg* **by** *blast*
next
case (*context_right t s u*)
then show *?case*
using *gt_compat_fun* **by** *auto*
qed

lemma *gt_embedding_property*:
assumes $t \trianglerighteq_{emb} s$ $t \neq s$
shows $t >_t s$
using *assms*
proof (*induction*)
case (*refl t*)
then show *?case* **by** *simp*
next
case (*step t u s*)
then show *?case* **using** *gt_embedding_step_property gt_trans* **by** *blast*
qed

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*)
apply (*rule extf_map*[*of set ys* ∪ *set xs ys xs* $(>_t)$ *h f*])
apply *simp*
apply (*simp add*: *in_listsI*)
apply (*simp add*: *in_listsI*)
using *gt_antisym* **apply** *blast*
using *gt_trans* **apply** *blast*
by (*simp add*: *assms*)+

theorem *gt_sus*:
assumes *ϱ_wary*: *wary_subst ϱ*
assumes *ghd*: $\bigwedge x.$ *ground_heads* (*Var x*) = *UNIV*
shows $t >_t s \Longrightarrow$ *subst ϱ t* $>_t$ *subst ϱ s*
proof (*simp only*:*atomize_imp*,*induction rule*:*measure_induct*[*of* $\lambda(t,s).$ {# *hsize t*, *hsize s* #} $\lambda(t,s).$ $t >_t s \longrightarrow$ *subst ϱ t* $>_t$ *subst ϱ s* (*t*,*s*), *simplified prod.case*],
simp only: *split_paired_all prod.case atomize_imp*[*symmetric*] *atomize_all*[*symmetric*])
fix *t s*
assume *ih*:$\bigwedge$*tt ss.*
{# *hsize tt*, *hsize ss* #} < {# *hsize t*, *hsize s* #} $\Longrightarrow$
$tt >_t ss \Longrightarrow$ *subst ϱ tt* $>_t$ *subst ϱ ss*
and $t >_t s$
show *subst ϱ t* $>_t$ *subst ϱ s* **using** ‹$t >_t s$›
proof (*cases*)
case *t_gt_s_chop*: *gt_chop*
then show *?thesis*
using *emb_step_subst emb_step_chop*[*OF t_gt_s_chop*(*1*)] *gt_embedding_step_property*
emb_step_hsize gt_trans ih[*of chop t s*] **by** (*metis add_mset_lt_left_lt*)
next

```
  case t_gt_s_diff: gt_diff
  have gt_diff1: head (subst ϱ t) >_hd head (subst ϱ s)
    by (meson assms gt_hd_def subsetCE t_gt_s_diff(1) wary_subst_ground_heads)
  have gt_diff2: is_Sym (head (subst ϱ s))
   by (metis ground_imp_subst_iden hd.collapse(2) hd.simps(18) head_subst t_gt_s_diff(2) tm.sel(1) tm.simps(17))
  have gt_diff3: chkchop (>_t) (subst ϱ t) (subst ϱ s)
  proof (cases s)
    case (Hd _)
    then show ?thesis
      using t_gt_s_diff unfolding chkchop_def
      by (metis ground_imp_subst_iden hd.collapse(2) hd.simps(18) tm.disc(1) tm.sel(1) tm.simps(17))
  next
    case s_App: (App s1 s2)
    then show ?thesis using t_gt_s_diff unfolding chkchop_def
      using ih[of t chop s] chop_subst_Sym hsize_chop_lt tm.disc(2)
      by (metis add_mset_lt_left_lt add_mset_lt_right_lt)
  qed
  show ?thesis
    using gt_diff gt_diff1 gt_diff2 gt_diff3 by blast
next
  case t_gt_s_same: gt_same
  have gt_same1: head (subst ϱ t) = head (subst ϱ s)
    by (simp add: t_gt_s_same(1))

  have extf_map_ts:∀ f∈ground_heads (head t). extf f (>_t) (map (subst ϱ) (args t)) (map (subst ϱ) (args s))
  proof −
   have ih_args: ∀ y∈set (args t) ∪ set (args s). ∀ x∈set (args t) ∪ set (args s). y >_t x ⟶ subst ϱ y >_t subst ϱ x
     by (metis Un_iff less_multiset_doubletons hsize_in_args ih)
   have ∀ f∈ground_heads (head t). extf f (>_t) (args t) (args s)
     using ghd t_gt_s_same(3) by metis
   then show ?thesis
     using extf_map[of set (args t) ∪ set (args s) args t args s gt subst ϱ]
     using gt_irrefl gt_trans ih_args by blast
  qed

  show ?thesis
  proof (cases head t)
    case (Var x1)
    then have is_Var (head t) by simp
    {
      fix u :: ('s, 'v) tm
      assume ground_heads (head u) ⊆ ground_heads (head t) hsize u ≤ hsize (subst ϱ (Hd (head t)))
      then have apps u (map (subst ϱ) (args t)) >_t apps u (map (subst ϱ) (args s))
      proof (induct hsize u arbitrary:u rule:less_induct)
        case less
        then show ?case
        proof (cases u)
          case u_Hd: (Hd _)
          then have args u = []
            by simp
          then show ?thesis
          proof (cases s)
            case s_Hd: (Hd _)
            show ?thesis
              by (smt (verit, best) Nil_is_map_conv UNIV_I Var ‹args u = []› ‹is_Var (head t)›
                  append_self_conv2 args.simps(1) args_Nil_iff_is_Hd args_apps chkchop_def
                  chkchop_same_def extf_map_ts ghd gt_same head_apps s_Hd t_gt_s_same(2))
          next
            case s_App: (App _ _)
            then have is_App t
              using ‹is_Var (head t)› chkchop_same_def t_gt_s_same(2) by presburger

            have chop t >_t chop s
```

```
        using ‹is_App t› ‹is_Var (head t)› s_App t_gt_s_same(2) by auto
      then have subst ϱ (chop t) >t subst ϱ (chop s) using ih
        by (metis ‹is_App t› less_multiset_doubletons s_App hsize_chop_lt tm.disc(2))

      define ut where ut = apps u (map (subst ϱ) (args t))
      define us where us = apps u (map (subst ϱ) (args s))
      have 0:⋀ss. args (apps u ss) = ss
        using ‹args u = []› by simp
      have chop_us: chop us = subst ϱ (chop s)
        unfolding chop_def subst_apps us_def 0 using hd_map
        by (metis (no_types, lifting) args_Nil_iff_is_Hd map_tl s_App tm.disc(2))
      have chop_ut: chop ut = subst ϱ (chop t)
        unfolding chop_def subst_apps ut_def 0 using ‹is_App t›
        by (simp add: args_Nil_iff_is_Hd hd_map map_tl)

      have head ut = head us
        by (simp add: us_def ut_def)
      moreover have chkchop_same (>t) ut us unfolding chkchop_def chkchop_same_def
        by (metis 0 Nil_is_map_conv ‹is_App t› ‹subst ϱ (chop t) >t subst ϱ (chop s)›
          args_Nil_iff_is_Hd chop_us chop_ut gt_chop ut_def)
      moreover have ∀f∈local.ground_heads (head ut). extf f (>t) (args ut) (args us)
        using extf_map_ts less us_def ut_def using 0 by auto
      ultimately show ut >t us
        using gt_same by blast
    qed
   next
    case u_app: (App _ _)
    let ?ut = apps u (map (subst ϱ) (args t))
    let ?us = apps u (map (subst ϱ) (args s))
    have 1:head ?ut = head ?ut
      by simp
    have apps (chop u) (map (subst ϱ) (args t)) >t apps (chop u) (map (subst ϱ) (args s))
      using less.hyps[of chop u] hsize_chop_lt
     by (metis Var dual_order.trans ghd less.prems(2) less_or_eq_imp_le subset_UNIV tm.disc(2) u_app)
    then have chop ?ut >t chop ?us
      by (simp add: chop_apps u_app)
    then have 2:chkchop_same (>t) ?ut ?us unfolding chkchop_same_def chkchop_def
      by (metis (no_types, lifting) Nil_is_map_conv ‹is_Var (head t)› append_is_Nil_conv
        args_Nil_iff_is_Hd args_apps chkchop_same_def gt.simps t_gt_s_same(2))
    have 3:∀f∈local.ground_heads (head ?ut). extf f (>t) (args ?ut) (args ?us)
      using extf_compat_append_left extf_map_ts less.prems(1) by auto
    show ?thesis using gt_same 1 2 3 by simp
   qed
  qed
 }
 note inner_induction = this
 show ?thesis using inner_induction[of subst ϱ (Hd (head t)), unfolded subst_apps[symmetric]]
  by (metis Var ghd order_refl subset_UNIV t_gt_s_same(1) tm_collapse_apps)
next
 case (Sym _)
 then have is_Sym (head (subst ϱ t)) head (subst ϱ t) = head t
  by simp_all
 then have chkchop_same (>t) t s
  using t_gt_s_same unfolding chkchop_same_def chkchop_def
  using Sym by metis
 then have gt_same2: chkchop_same (>t) (subst ϱ t) (subst ϱ s) unfolding chkchop_same_def chkchop_def
   using ih[of t chop s]
   by (metis (no_types, lifting) Sym ‹head (subst ϱ t) = head t› ‹is_Sym (head (subst ϱ t))›
     add_mset_commute add_mset_lt_left_lt chop_subst_Sym ground_imp_subst_iden hd.simps(18)
     hsize_chop_lt t_gt_s_same(1) tm.collapse(1) tm.simps(17))
 have gt_same3: ∀f∈local.ground_heads (head (subst ϱ t)). extf f (>t) (args (subst ϱ t)) (args (subst ϱ s))
  using ‹head (subst ϱ t) = head t› extf_compat_append_left extf_map_ts t_gt_s_same(1) by auto
 show ?thesis using gt_same gt_same1 gt_same2 gt_same3 by blast
```

```
    qed
  qed
qed
```

4.9 Totality on Ground Terms

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

  let ?case = t >_t s ∨ s >_t t ∨ t = s

  have chkchop (>_t) t s ∨ s >_t t
    unfolding chkchop_def tm.case_eq_if using ih[of t chop s]
   by (metis (no_types, lifting) add_mset_commute add_mset_lt_left_lt gr_s gr_t ground_chop gt_chop hsize_chop_lt)
  moreover have chkchop (>_t) s t ∨ t >_t s
    unfolding chkchop_def tm.case_eq_if using ih[of chop t s]
    by (metis add_mset_lt_left_lt gr_s gr_t ground_chop gt_chop.intros gt_iff_chop_diff_same hsize_chop_lt)
  moreover
  {
    assume
      chkembs_t_s: chkchop (>_t) t s and
      chkembs_s_t: chkchop (>_t) s t

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

    {
      assume g_gt_f: g >_s f
      have t >_t s
        using chkembs_t_s f g g_gt_f gt_diff gt_sym_imp_hd by auto
    }
    moreover
    {
      assume f_gt_g: f >_s g
      have s >_t t
        using chkembs_s_t f f_gt_g g gt_diff gt_sym_imp_hd by auto
    }
    moreover
    {
      assume g_eq_f: g = f
      hence hd_t: head t = head s
        using g f by auto

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

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

      {
```

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

4.10 Well-foundedness

```
lemma gt_imp_vars: t >t s ⟹ vars t ⊇ vars s
proof (simp only: atomize_imp,
   rule measure_induct_rule[of λ(t, s). hsize t + hsize s
     λ(t, s). t >t s ⟶ vars t ⊇ vars s (t, s), simplified prod.case],
   simp only: split_paired_all prod.case atomize_imp[symmetric])
 fix t s
 assume
   ih: ⋀ta sa. hsize ta + hsize sa < hsize t + hsize s ⟹ ta >t sa ⟹ vars ta ⊇ vars sa and
   t_gt_s: t >t s
 show vars t ⊇ vars s
   using t_gt_s
 proof cases
   case (gt_chop)
   thus ?thesis
     using ih
     by (metis add_mono_thms_linordered_field(1) le_supI1 order_refl hsize_chop_lt vars_chop)
 next
   case gt_diff
   show ?thesis
   proof (cases s)
     case Hd
     thus ?thesis
       using gt_diff(2)
       by (metis empty_iff hd.collapse(2) hd.simps(18) subsetI tm.sel(1) tm.simps(17))
   next
     case (App s1 s2)
     have vars (chop s) ⊆ vars t using ih
       using App chkchop_def local.gt_diff(3) nat_add_left_cancel_less hsize_chop_lt tm.disc(2) by blast
     thus ?thesis
       using  App  le_sup_iff local.gt_diff(2) tm.disc(2) vars_chop
       by (metis empty_iff hd.collapse(2) hd.simps(18) subsetI)
   qed
 next
   case gt_same
```

```
    thus ?thesis
    proof (cases head t)
      case (Var x)
      then show ?thesis
      proof (cases t)
        case (Hd _)
        then show ?thesis using Var local.gt_same(2) by force
      next
        case (App t1 t2)
        then show ?thesis
        proof (cases s)
          case (Hd _)
          then show ?thesis
            using local.gt_same(1) vars_head_subseteq by fastforce
        next
          case (App s1 s2)
          then have chop t >t chop s
            using Var local.gt_same(2) by force
          then have vars (chop s) ⊆ vars (chop t) using ih[OF _ ‹chop t >t chop s›]
            by (metis App Var add_less_mono chkchop_same_def hd.disc(1) hsize_chop_lt
             local.gt_same(2) tm.disc(2))
          then show ?thesis using gt_same(1) vars_chop[of t] vars_chop[of s]
            by (metis App Var chkchop_same_def hd.disc(1) le_sup_iff local.gt_same(2)
              sup.coboundedI1 tm.disc(2) vars_head_subseteq)
        qed
      qed
    next
      case (Sym f)
      then have chkchop (>t) t s using gt_same chkchop_same_def by auto
      then show ?thesis
      proof (cases s)
        case (Hd _)
        then show ?thesis using local.gt_same(1) vars_head_subseteq by force
      next
        case (App s1 s2)
        then show ?thesis unfolding chkchop_def using vars_chop ih[of t chop s]
          by (metis ‹chkchop (>t) t s› chkchop_def le_sup_iff local.gt_same(1)
              nat_add_left_cancel_less hsize_chop_lt tm.disc(2) vars_head_subseteq)
      qed
    qed
  qed
qed

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

theorem gt_wf:
  assumes ghd_UNIV: ⋀x. ground_heads_var x = UNIV
  assumes extf_wf: ⋀f. ext_wf (extf f)
  shows wfP (λs t. t >t s)
proof −
  have ground_wfP: wfP (λs t. t >tg s)
    unfolding wfP_iff_no_inf_chain
  proof
    assume ∃f. inf_chain (>tg) f
    then obtain t where t_bad: bad (>tg) t
      unfolding inf_chain_def bad_def by blast

    let ?ff = worst_chain (>tg) (λt s. hsize t > hsize s)
    let ?U_of = λi. {u. (?ff i) ▷emb u}

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

```
define U where U = (⋃i. ?U_of i)

have gr: ⋀i. ground (?ff i)
  using worst_chain_bad[OF wf_sz t_bad, unfolded inf_chain_def] by fast
have gr_u: ⋀u. u ∈ U ⟹ ground u unfolding U_def
  using gr ground_emb by fastforce

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

  assume u_bad: bad (>tg) u
  have sz_u: hsize u < hsize ?ti
    using emb_hsize_neq u_in by blast

  show False
  proof (cases i)
    case 0
    thus False
      using sz_u min_worst_chain_0[OF wf_sz u_bad] by simp
  next
    case Suc
    hence ?ff (i − 1) >t ?ff i
      using worst_chain_pred[OF wf_sz t_bad] by simp
    moreover have ?ff i >t u
      using gt_embedding_property u_in by blast
    ultimately have ?ff (i − 1) >t u
      by (rule gt_trans)
    thus False
      using Suc sz_u min_worst_chain_Suc[OF wf_sz u_bad] gr by fastforce
  qed
qed
hence u_good: ⋀u. u ∈ U ⟹ ¬ bad (>tg) u
  unfolding U_def by blast

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

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

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

  have ¬ gt_chop (?ff i) (?ff (Suc i))
  proof
    assume a: gt_chop (?ff i) (?ff (Suc i))
    then have chop (?ff i) ∈ ?U_of i
   by (metis (mono_tags, lifting) emb_step_chop emb_step_is_emb gt_chop gt_chop.cases gt_irrefl mem_Collect_eq)
    then have  uij_in:chop (?ff i) ∈ U unfolding U_def by fast

    have ⋀n. ?ff n >t ?ff (Suc n)
      by (rule worst_chain_pred[OF wf_sz t_bad, THEN conjunct2])
    hence uij_gt_i_plus_3: chop (?ff i) >t ?ff (Suc (Suc i))
      using gt_trans by (metis (mono_tags, lifting) a gt_chop.cases)

    have inf_chain (>tg) (λj. if j = 0 then chop (?ff i) else ?ff (Suc (i + j)))
      unfolding inf_chain_def
      by (auto intro!: gr gr_u[OF uij_in] uij_gt_i_plus_3 worst_chain_pred[OF wf_sz t_bad])
    hence bad (>tg) (chop (?ff i))
      unfolding bad_def by fastforce
```

```
      thus False
        using u_good[OF uij_in] by sat
    qed
    thus gt_diff (?ff i) (?ff (Suc i)) ∨ gt_same (?ff i) (?ff (Suc i))
      using gt unfolding gt_iff_chop_diff_same by sat
  qed

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

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

  have diff_O_same: {(s, t). ground t ∧ gt_diff t s} O {(s, t). ground t ∧ gt_same t s}
    ⊆ {(s, t). ground t ∧ gt_diff t s}
    unfolding gt_diff.simps gt_same.simps
    by clarsimp (metis chkchop_def chkchop_same_def gt_same gt_trans)

  have diff_same_as_union: {(s, t). ground t ∧ (gt_diff t s ∨ gt_same t s)} =
    {(s, t). ground t ∧ gt_diff t s} ∪ {(s, t). ground t ∧ gt_same t s}
    by auto

  obtain k where bad_same: inf_chain (λt s. ground t ∧ gt_same t s) (λi. ?ff (i + k))
    using wf_infinite_down_chain_compatible[OF wf_diff _ diff_O_same, of ?ff] bad_diff_same
    unfolding inf_chain_def diff_same_as_union[symmetric] by auto
  hence hd_sym: ⋀i. is_Sym (head (?ff (i + k)))
    unfolding inf_chain_def by (simp add: ground_head)

  define f where f = sym (head (?ff k))

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

  let ?gtu = λt s. t ∈ U ∧ t >t s
  thm UnionI CollectI
  have t ∈ set (args (?ff i)) ⟹ t ∈ U for t i
    unfolding U_def apply (rule UnionI[of ?U_of i])
    using arg_emb CollectI arg_emb hsize_in_args by fast+
  moreover have ⋀i. extf f (>tg) (args (?ff (i + k))) (args (?ff (Suc i + k)))
    using bad_same hd_eq_f unfolding inf_chain_def gt_same.simps f_def hd.collapse(2)[OF ground_head,
OF gr]
    using extf_mono_strong[of _ _ (>t) (λt s. ground t ∧ t >t s) ] ground_hd_in_ground_heads
    by (metis (no_types, lifting) ground_args)
  ultimately have ⋀i. extf f ?gtu (args (?ff (i + k))) (args (?ff (Suc i + k)))
```

```
    using extf_mono_strong[of _ _ (λt s. ground t ∧ t >t s) λt s. t ∈ U ∧ t >t s] unfolding U_def by blast
  hence inf_chain (extf f ?gtu) (λi. args (?ff (i + k)))
    unfolding inf_chain_def by blast
  hence nwf_ext: ¬ wfP (λxs ys. extf f ?gtu ys xs)
    unfolding wfP_iff_no_inf_chain by fast

  have gtu_le_gtg: ?gtu ≤ (>tg)
    by (auto intro!: gr_u)

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

    have f 0 ∈ U
      using bad_f unfolding inf_chain_def by blast
    hence good_f0: ¬ bad ?gtu (f 0)
      using u_good bad_f inf_chain_bad inf_chain_subset[OF _ gtu_le_gtg] by blast

    show False
      using bad_f0 good_f0 by sat
  qed
  hence wf_ext: wfP (λxs ys. extf f ?gtu ys xs)
    by (rule ext_wf.wf[OF extf_wf, rule_format])

  show False
    using nwf_ext wf_ext by blast
qed

let ?subst = subst grounding_ϱ

have wfP (λs t. ?subst t >tg ?subst s)
  by (rule wfP_app[OF ground_wfP])
hence wfP (λs t. ?subst t >t ?subst s)
  by (simp add: ground_grounding_ϱ)
thus ?thesis
  using gt_sus ghd_UNIV ground_heads.simps(1) wary_grounding_ϱ wfP_eq_minimal
  by (metis (no_types, lifting))
qed

end

end
```