

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

Alexander Bentkamp

October 21, 2018

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	1
2	The Embedding Relation for Lambda-Free Higher-Order Terms	2
2.1	Positions of terms	2
2.2	Embedding step	3
2.3	Embedding relation	5
2.4	How are positions preserved under embedding steps?	8
2.5	Swapping embedding steps	9
2.6	Performing embedding steps in order of a given priority	10
2.7	Embedding steps under arguments	13
2.8	Rearranging embedding steps: first above, then below arguments	15
3	The Chop Operation on Lambda-Free Higher-Order Terms	18
3.1	Basic properties	18
3.2	Chop and the Embedding Relation	19
3.3	Chop and Substitutions	22
4	The Embedding Path Order for Lambda-Free Higher-Order Terms	24
4.1	Setup	24
4.2	Inductive Definitions	24
4.3	Transitivity	25
4.4	Irreflexivity	29
4.5	Compatibility with Embedding Relation	29
4.6	Subterm Property	34
4.7	Compatibility with Contexts	34
4.8	Stability under Substitutions	36
4.9	Totality on Ground Terms	39
4.10	Well-foundedness	41

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* \Rightarrow *dir list* \Rightarrow *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* \Rightarrow *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]*) \Longrightarrow *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*: \neg *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 \neg *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 \bigwedge *ds*. *replicate* (*num_args* *t*) *dir.Left @ [d]* \neq *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* (\neg *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*) \Longrightarrow *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  $\Rightarrow$  dir  $\Rightarrow$  ('s, 'v) tm  $\Rightarrow$  ('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 $emb_step_at_is_App: emb_step_at p d u \neq u \implies is_App u$
by (metis emb_step_at.simps(5) is_Hd_def)

Definition of an embedding step without using positions.

```

inductive emb_step (infix  $\rightarrow_{emb}$  50) where
  left: (App t1 t2)  $\rightarrow_{emb}$  t1 |
  right: (App t1 t2)  $\rightarrow_{emb}$  t2 |
  context_left:  $t \rightarrow_{emb} s \implies (App t u) \rightarrow_{emb} (App s u)$  |
  context_right:  $t \rightarrow_{emb} s \implies (App u t) \rightarrow_{emb} (App u s)$ 

```

The two definitions of an embedding step are equivalent:

lemma $emb_step_equiv: emb_step t s \iff (\exists p d. emb_step_at p d t = s) \wedge t \neq s$

proof

show $t \rightarrow_{emb} s \implies (\exists p d. emb_step_at p d t = s) \wedge t \neq s$

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 $\bigwedge u. emb_step_at (Left \# p) d (App t u) = App s u$

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 $\bigwedge u. emb_step_at (Right \# p) d (App u t) = App u s$

by (metis emb_step_at.simps(4))

then show ?case

using context_right.hyps(2) **by** blast

```

qed
show ( $\exists p d. \text{emb\_step\_at } p d t = s$ )  $\wedge t \neq s \implies t \rightarrow_{\text{emb}} s$ 
proof -
  assume ( $\exists p d. \text{emb\_step\_at } p d t = s$ )  $\wedge t \neq s$ 
  then obtain  $p d$  where  $\text{emb\_step\_at } p d t = s$   $t \neq s$  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.prem(1)
left_context.prem(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.prem(1)
right_context.prem(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.prem(2) by blast
  qed
qed
qed
qed

lemma emb_step_fun:  $\text{is\_App } t \implies t \rightarrow_{\text{emb}} (\text{fun } t)$ 
by (metis emb_step.intros(1) tm.collapse(2))

lemma emb_step_arg:  $\text{is\_App } t \implies t \rightarrow_{\text{emb}} (\text{arg } t)$ 
by (metis emb_step.intros(2) tm.collapse(2))

lemma emb_step_size:  $t \rightarrow_{\text{emb}} s \implies \text{size } t > \text{size } s$ 
by (induction rule: emb_step.induct; simp_all)

lemma emb_step_vars:  $t \rightarrow_{\text{emb}} s \implies \text{vars } s \subseteq \text{vars } t$ 
by (induction rule: emb_step.induct, auto)

lemma emb_step_equiv':  $\text{emb\_step } t s \iff (\exists p. p \neq [] \wedge \text{emb\_step\_at } p t = s) \wedge t \neq s$ 
using butlast_snoc emb_step_equiv last_snoc
by (metis snoc_eq_iff_butlast)

lemma position_if_emb_step_at:  $\text{emb\_step\_at } p d t = u \implies t \neq u \implies \text{position\_of } t (p @ [d])$ 
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  $t \rightarrow_{emb} emb\_step\_at\ p\ d\ t$ 
  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  $\times$  dir) list  $\Rightarrow$  ('s, 'v) tm  $\Rightarrow$  ('s, 'v) tm where
  emb_at_Nil: emb_at [] t = t |
  emb_at_Cons: emb_at ((p,d) # ps) t = emb_step_at p d (emb_at ps t)

```

Definition of an embedding without using positions:

```

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

```

abbreviation emb_neq (infix \triangleright_{emb} 50) where $emb_neq\ t\ s \equiv t \triangleright_{emb} s \wedge t \neq s$

The two definitions coincide:

```

lemma emb_equiv:  $(t \triangleright_{emb} s) = (\exists ps. emb\_at\ ps\ t = s)$ 

```

```

proof
  show  $\exists ps. emb\_at\ ps\ t = s \Longrightarrow t \triangleright_{emb} s$ 
  proof -
    assume  $\exists ps. emb\_at\ ps\ t = s$ 
    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
qed

```

```

next
  show  $t \triangleright_{emb} s \implies \exists ps. emb\_at\ ps\ t = s$ 
  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:  $emb\_at\ ps\ t = u \implies emb\_at\ qs\ u = s \implies emb\_at\ (qs\ @\ ps)\ t = s$ 
  by (induct qs arbitrary:s; auto)

lemma emb_trans:  $t \triangleright_{emb} u \implies u \triangleright_{emb} s \implies t \triangleright_{emb} s$ 
  by (metis emb_at_trans emb_equiv)

lemma emb_step_is_emb:  $t \rightarrow_{emb} s \implies t \triangleright_{emb} s$ 
  by (meson emb.simps)

lemma emb_size:  $t \triangleright_{emb} s \implies size\ t \geq size\ s$ 
  apply (induction rule: emb.induct)
  apply simp
  using emb_step_size by fastforce

lemma emb_prepend_step:  $t \rightarrow_{emb} u \implies u \triangleright_{emb} s \implies t \triangleright_{emb} s$ 
  using emb_step_is_emb emb_trans by blast

lemma sub_emb:  $sub\ s\ t \implies t \triangleright_{emb} s$ 
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:  $t \triangleright_{emb} s \iff (\exists us. us \neq [] \wedge hd\ us = t \wedge last\ us = s \wedge (\forall i. Suc\ i < length\ us \longrightarrow us\ !\ i \rightarrow_{emb} us\ !\ Suc\ i))$ 
proof
  show  $t \triangleright_{emb} s \implies \exists us. us \neq [] \wedge hd\ us = t \wedge last\ us = s \wedge (\forall i. Suc\ i < length\ us \longrightarrow us\ !\ i \rightarrow_{emb} us\ !\ Suc\ i)$ 
  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 $(\forall i. \text{Suc } i < \text{length } (us @ [s]) \longrightarrow (us @ [s]) ! i \rightarrow_{emb} (us @ [s]) ! \text{Suc } i)$
by (*metis* *mono_tags*, *lifting*) *Suc_less_eq* $\langle \forall i. \text{Suc } i < \text{length } us \longrightarrow us ! i \rightarrow_{emb} us ! \text{Suc } i \rangle$ $\langle \text{last } us = u \rangle$
 $\langle us \neq [] \rangle$
append_butlast_last_id *length_append_singleton* *less_antisym* *nth_append_nth_append_length* *step.hyps*(3)
ultimately show *?case* **by** *blast*
qed
show $\exists us. us \neq [] \wedge \text{hd } us = t \wedge \text{last } us = s \wedge (\forall i. \text{Suc } i < \text{length } us \longrightarrow us ! i \rightarrow_{emb} us ! \text{Suc } i) \Longrightarrow t \succeq_{emb} s$
proof –
assume $\exists us. us \neq [] \wedge \text{hd } us = t \wedge \text{last } us = s \wedge (\forall i. \text{Suc } i < \text{length } us \longrightarrow us ! i \rightarrow_{emb} us ! \text{Suc } i)$
then obtain *us* **where** $us \neq [] \wedge \text{hd } us = t \wedge \text{last } us = s \wedge (\forall i. \text{Suc } i < \text{length } us \longrightarrow us ! i \rightarrow_{emb} us ! \text{Suc } i)$ **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: $t \succeq_{emb} s$ **and**
refl: $\bigwedge t. P t t$ **and**
step: $\bigwedge t u s. t \rightarrow_{emb} u \Longrightarrow u \succeq_{emb} s \Longrightarrow P u s \Longrightarrow P t s$
shows
 $P t s$
proof –
obtain *us* **where** $us \neq [] \wedge \text{hd } us = t \wedge \text{last } us = s \wedge (\forall i. \text{Suc } i < \text{length } us \longrightarrow us ! i \rightarrow_{emb} us ! \text{Suc } i)$
using *emb_sequence_emb_steps* **by** *blast*
define *us'* **where** $us' = tl\ us$
then have $\text{last } ([t] @ us') = s \wedge (\forall i. i < \text{length } us' \longrightarrow ([t] @ us') ! i \rightarrow_{emb} ([t] @ us') ! \text{Suc } i)$
using $\langle \text{hd } us = t \rangle$ $\langle \text{last } us = s \rangle$ $\langle us \neq [] \rangle$ **apply** *force*
using $\langle \forall i. \text{Suc } i < \text{length } us \longrightarrow us ! i \rightarrow_{emb} us ! \text{Suc } i \rangle$ $\langle \text{hd } us = t \rangle$ $\langle us \neq [] \rangle$ *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 $t \rightarrow_{emb} a$
using *Cons.premis*(2) **by** *auto*
have $a \succeq_{emb} s$ **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* $\langle t \rightarrow_{emb} a \rangle$ $\langle a \succeq_{emb} s \rangle$ $\langle P a s \rangle$ *]*.
qed
qed

lemma *emb_cases_reverse* [*consumes 1*, *case_names refl step*]:

$t \succeq_{emb} s \Longrightarrow (\bigwedge t'. t = t' \Longrightarrow s = t' \Longrightarrow P) \Longrightarrow (\bigwedge t' u s'. t = t' \Longrightarrow s = s' \Longrightarrow t' \rightarrow_{emb} u \Longrightarrow u \succeq_{emb} s' \Longrightarrow P) \Longrightarrow P$
by (*induct rule:emb_induct_reverse; blast+*)

lemma *emb_vars*: $t \succeq_{emb} s \Longrightarrow \text{vars } s \subseteq \text{vars } t$

apply (*induct rule: emb.induct*)
apply *simp*
using *emb_step_vars* **by** *auto*

lemma *ground_emb*: $t \triangleright_{emb} s \implies \text{ground } t \implies \text{ground } s$
using *emb_vars* **by** *blast*

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

lemma *emb_step_at_subst*:
assumes
 position_of t (p @ $[d]$)
shows
 $\text{emb_step_at } p \ d \ (\text{subst } \varrho \ t) = \text{subst } \varrho \ (\text{emb_step_at } p \ d \ t)$
using *assms* **proof** (*induction p arbitrary:t*)
case *Nil*
then obtain $t1 \ t2$ **where** $t = \text{App } t1 \ t2$
 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** $t = \text{App } t1 \ t2$
 using *position_of.elims(2)* **by** *blast*
then show *?case*
 using *Cons.IH Cons.prem* **by** (*cases a; auto*)
qed

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

lemma *emb_subst*: $t \triangleright_{emb} s \implies \text{subst } \varrho \ t \triangleright_{emb} \text{subst } \varrho \ s$
apply (*induct rule:emb.induct*)
apply (*simp add: emb.refl*)
using *emb.step emb_step_subst* **by** *blast*

lemma *emb_size_neq*:
assumes
 $t \triangleright_{emb} s \ t \neq s$
shows
 $\text{size } t > \text{size } s$
by (*metis assms(1) assms(2) emb_cases_reverse emb_size emb_step_size 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
 $\text{position_of } t \ (p \ @ \ [d1]) \longleftrightarrow \text{position_of } (\text{emb_step_at } q \ d2 \ t) \ (p \ @ \ [d1])$
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 \vee p = [] \vee q = []*)
 case *True*
 then show *?thesis*
 by (*metis emb_step_at_is_App less.prem(1) less.prem(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.prem(1) less.prem(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])  $\longleftrightarrow$  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])  $\longleftrightarrow$  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.prem 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  $\leq$  length q  $\implies$  take (length p) q  $\neq$  p length q  $\leq$  length p  $\implies$  take (length q) p  $\neq$  q
  shows
    emb_step_at q d2 (emb_step_at p d1 t) = emb_step_at p d1 (emb_step_at q d2 t)
  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  $\vee$  p = []  $\vee$  q = [])
  case True
  then show ?thesis
  using emb_step_at_is_App less.prem(1) less.prem(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.prem(1) less.prem(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 $f (g b) \rightarrow \text{emb } f b \rightarrow \text{emb } f$ can be achieved using a single step $f (g b) \rightarrow \text{emb } f$.

lemma *merge_emb_step_at*:

$\text{emb_step_at } p \ d1 \ (\text{emb_step_at } (p \ @ \ [\text{opp } d1] \ @ \ q) \ d2 \ t) = \text{emb_step_at } p \ d1 \ t$

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*:

$\text{emb_step_at } (p \ @ \ q) \ d2 \ (\text{emb_step_at } p \ d1 \ t) = \text{emb_step_at } p \ d1 \ (\text{emb_step_at } (p \ @ \ [d1] \ @ \ q) \ d2 \ t)$

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.prem* **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* \Rightarrow *nat*) \Rightarrow (*'s, 'v*) *tm* \Rightarrow (*'s, 'v*) *tm* \Rightarrow *dir list* **where**

$\text{prio_emb_pos } \text{prio } t \ s = (\text{ARG_MIN } \text{prio } p. \ p \neq [] \wedge \text{position_of } t \ p \wedge \text{emb_step_at}' \ p \ t \ \succeq_{\text{emb}} \ s)$

definition *prio_emb_step* :: (*dir list* \Rightarrow *nat*) \Rightarrow (*'s, 'v*) *tm* \Rightarrow (*'s, 'v*) *tm* \Rightarrow (*'s, 'v*) *tm* **where**

$\text{prio_emb_step } \text{prio } t \ s = \text{emb_step_at}' \ (\text{prio_emb_pos } \text{prio } t \ s) \ t$

lemma *prio_emb_posI*:

$t \ \succeq_{\text{emb}} \ s \ \Longrightarrow \ t \neq s \ \Longrightarrow \ \text{prio_emb_pos } \text{prio } t \ s \neq [] \wedge \text{position_of } t \ (\text{prio_emb_pos } \text{prio } t \ s) \wedge \text{emb_step_at}' \ (\text{prio_emb_pos } \text{prio } t \ s) \ t \ \succeq_{\text{emb}} \ s$

proof (*induct rule:emb_induct_reverse*)

case (*refl t*)

then show *?case* **by** *simp*

next

case (*step t u s*)

obtain *p* **where** $0 : p \neq [] \wedge \text{position_of } t \ p \wedge \text{emb_step_at}' \ p \ t \ \succeq_{\text{emb}} \ s$

using *emb_step_equiv' step.hyps(1) append_butlast_last_id position_if_emb_step_at*

by (*metis step.hyps(2)*)

then have $\text{prio_emb_pos } \text{prio } t \ s \neq [] \wedge \text{position_of } t \ (\text{prio_emb_pos } \text{prio } t \ s) \wedge \text{emb_step_at}' \ (\text{prio_emb_pos } \text{prio } t \ s) \ t \ \succeq_{\text{emb}} \ s$

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 $p \neq [] \wedge \text{position_of } t \ p \wedge \text{emb_step_at}' \ p \ t \ \succeq_{\text{emb}} \ s$

shows $\text{prio} (\text{prio_emb_pos} \text{ prio } t \ s) \leq \text{prio } p$
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

$t \succeq_{\text{emb}} s \ t \neq s \ \text{prio_emb_step} \ \text{prio} \ t \ s \neq s$ **and**

valid_prio: $\bigwedge p \ q \ dp \ dq. \ \text{prio} (p \ @ \ [dp]) > \text{prio} (q \ @ \ [dq]) \implies \text{take} (\text{length } p) \ q \neq p$

shows

$\text{prio} (\text{prio_emb_pos} \ \text{prio} \ t \ s) \leq \text{prio} (\text{prio_emb_pos} \ \text{prio} (\text{prio_emb_step} \ \text{prio} \ t \ s) \ s)$

(**is** $\text{prio} \ ?p1 \leq \text{prio} \ ?p2$)

proof (*rule ccontr*)

assume *contr*: $\neg \text{prio} \ ?p1 \leq \text{prio} \ ?p2$

have $\text{take} (\text{length} (\text{butlast } ?p1)) (\text{butlast } ?p2) \neq (\text{butlast } ?p1)$

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*: $\text{butlast } ?p2 \neq \text{butlast } ?p1$

by *auto*

define *u* **where** $u = \text{prio_emb_step} \ \text{prio} (\text{prio_emb_step} \ \text{prio} \ t \ s) \ s$

then have $u \succeq_{\text{emb}} s$

by (*metis assms(1) assms(2) assms(3) prio_emb_posI prio_emb_step_def*)

define *p* **where** $p = \text{butlast } ?p2$

define *q* **where** $q = \text{drop} (\text{Suc} (\text{length} (\text{butlast } ?p2))) (\text{butlast } ?p1)$

define *dp* **where** $dp = \text{last } ?p2$

define *dq* **where** $dq = \text{last } ?p1$

define *f* **where** $f = \text{butlast } ?p1 \ ! \ \text{length} (\text{butlast } ?p2)$

have *position*: $\text{position_of} (\text{prio_emb_step} \ \text{prio} \ t \ s) \ ?p2$

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 $\text{length} (\text{butlast } ?p1) \neq \text{length } p$ **using** *butlast_neq*

using *True p_def by auto*

then have $\text{length} (\text{butlast } ?p1) > \text{length } p$

using *nat_le_linear True nat_neq_iff by auto*

then have $0 : p \ @ \ f \ \# \ q = \text{butlast } ?p1$

by (*metis True f_def id_take_nth_drop p_def q_def*)

have *u1*: $u = \text{emb_step_at } p \ dp \ (\text{emb_step_at } (p \ @ \ f \ \# \ q) \ dq \ t)$

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 $u = \text{emb_step_at } (p \ @ \ q) \ dq \ (\text{emb_step_at } p \ dp \ t)$

using *swap_nested_emb_step_at[of p q dq dp t] u1 by simp*

then have $\text{emb_step_at } p \ dp \ t \rightarrow_{\text{emb}} u \ \vee \ \text{emb_step_at } p \ dp \ t = u$

using *emb_step_equiv[of emb_step_at p dp t u] by blast*

then have $\text{emb_step_at } p \ dp \ t \succeq_{\text{emb}} s$

using $\langle u \succeq_{\text{emb}} s \rangle \ \text{emb_prepend_step}$ **by** *blast*

then have $\text{position_of } t \ ?p2$

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 succemb s*)

next

case *False*

then have $\text{opp } dp = f$

```

    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  $\succeq_{emb}$  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  $\neq$  p'
  using ‹take (length (butlast ?p1)) (butlast ?p2)  $\neq$  butlast ?p1› p'_def p_def by blast
have take (length p) p'  $\neq$  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  $\neq$  p'› ‹take (length p) p'  $\neq$  p›)
then have emb_step_at p dp t  $\succeq_{emb}$  s
  by (metis ‹u  $\succeq_{emb}$  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  $\neq$  p'› ‹take (length p) p'  $\neq$  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  $\succeq_{emb}$  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  $\succeq_{emb}$  s
  shows
     $\exists us. us \neq [] \wedge hd\ us = t \wedge last\ us = s \wedge$ 
     $(\forall i. Suc\ i < length\ us \longrightarrow (prio\_emb\_step\ prio\ (us\ !\ i)\ s = us\ !\ Suc\ i \wedge us\ !\ i \rightarrow_{emb}\ us\ !\ Suc\ i))$ 
  using assms
proof (induct t rule:measure_induct_rule[of size])
  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  $\neq []$  emb_step_at' (prio_emb_pos prio t s) t  $\succeq_{emb}$  s
    using prio_emb_posI less by blast+
  have emb_step1: t  $\rightarrow_{emb}$  emb_step_at' (prio_emb_pos prio t s) t
    by (simp add: False emb_step_at_if_position less.premis prio_emb_posI)
  have size (emb_step_at' (prio_emb_pos prio t s) t) < size t
    by (simp add: False emb_step_at_if_position emb_step_size less.premis(1) prio_emb_posI)
  then obtain us where us_def:
    us  $\neq []$ 
    hd us = emb_step_at' (prio_emb_pos prio t s) t
    last us = s
     $\forall i. Suc\ i < length\ us \longrightarrow prio\_emb\_step\ prio\ (us\ !\ i)\ s = us\ !\ Suc\ i \wedge us\ !\ i \rightarrow_{emb}\ us\ !\ Suc\ i$ 
  using less(1)[of emb_step_at' (prio_emb_pos prio t s) t] emb_step_size
    ‹emb_step_at' (prio_emb_pos prio t s) t  $\succeq_{emb}$  s› by blast

```

```

have  $t \# us \neq []$   $hd (t \# us) = t$   $last (t \# us) = s$  using  $us\_def$  by  $simp\_all$ 
have
   $\forall i. Suc\ i < length\ (t \# us) \longrightarrow (prio\_emb\_step\ prio\ ((t \# us)!\ i)\ s = (t \# us)!\ Suc\ i \wedge (t \# us)!\ i \rightarrow_{emb}$ 
 $(t \# us)!\ Suc\ i)$ 
  proof ( $rule\ allI$ ,  $rule\ impI$ )
    fix  $i$  assume  $Suc\ i < length\ (t \# us)$ 
    show  $prio\_emb\_step\ prio\ ((t \# us)!\ i)\ s = (t \# us)!\ Suc\ i \wedge (t \# us)!\ i \rightarrow_{emb}\ (t \# us)!\ Suc\ i$ 
    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  $\langle Suc\ i < length\ (t \# us) \rangle\ us\_def(4)$  by  $auto$ 
      qed
    qed
  then show  $?thesis$ 
  using  $\langle hd\ (t \# us) = t \rangle\ \langle last\ (t \# us) = s \rangle$  by  $blast$ 
qed
qed

```

2.7 Embedding steps under arguments

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

lemma $emb_step_under_args_head$:

```

assumes
   $\neg list\_all\ (\lambda x. x = Left)\ p$ 
shows
   $head\ (emb\_step\_at\ p\ d\ t) = head\ t$ 
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
   $\neg list\_all\ (\lambda x. x = Left)\ p$ 
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.prem(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.prem(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.prem(2) Left by auto
    have IH_cond2: position_of t1 (p @ [d])
      using App Cons.prem(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.prem(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.prem(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.prem(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.premis(1)[of num_args t - 1]
    using App Cons.premis(3) Right emb_step_at_if_position 0 by auto
  qed
qed

lemma emb_step_under_args_emb:
  assumes  $\neg$  list_all ( $\lambda x. x = \text{Left}$ ) p
    position_of t (p @ [d])
  shows
     $\forall i. i < \text{num\_args } t \longrightarrow \text{args } t ! i \succeq_{\text{emb}} \text{args } (\text{emb\_step\_at } p \ d \ t) ! i$ 
  by (metis assms emb.simps emb_step_under_args_emb_step)

lemma position_Left_only_subst:
  assumes list_all ( $\lambda x. x = \text{Left}$ ) p
    and position_of (subst  $\varrho$  w) (p @ [d])
    and num_args (subst  $\varrho$  w) = num_args w
  shows position_of w (p @ [d])
  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.premis(2) Cons.premis(3) position_of_Hd Hd_head_id append_Cons args.simps(1) list.size(3))
  next
  case (App w1 w2)
  have num_args (subst  $\varrho$  w1) = num_args w1
  using App Cons.premis(3) by auto
  then show ?thesis
  by (metis (full_types) App Cons.hyps Cons.premis(1) Cons.premis(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  $\varrho$  s  $\succeq_{\text{emb}}$  u
  obtains w where
    s  $\succeq_{\text{emb}}$  w
    subst  $\varrho$  w  $\succeq_{\text{emb}}$  u
     $\forall w'. w \rightarrow_{\text{emb}} w' \longrightarrow \neg \text{subst } \varrho \ w' \succeq_{\text{emb}} u$ 
  using assms
proof (induct s rule:measure_induct_rule[of size])
  case (less s)
  show ?case
  proof (cases  $\forall w'. s \rightarrow_{\text{emb}} w' \longrightarrow \neg \text{subst } \varrho \ w' \succeq_{\text{emb}} u$ )
  case True
  then show ?thesis
  using emb.refl less.premis(1)
  using less.premis(2) by blast
  next
  case False
  then show ?thesis
  by (meson emb_step_is_emb emb_step_size emb_trans less.hyps less.premis(1))
  qed
qed

```

qed

lemma *emb_only_below_vars*:

assumes

$subst\ \varrho\ s\ \succeq_{emb}\ u$
 $s\ \succeq_{emb}\ w$
 $is_Sym\ (head\ w)$
 $subst\ \varrho\ w\ \succeq_{emb}\ u$
 $\forall w'. w \rightarrow_{emb} w' \longrightarrow \neg subst\ \varrho\ w' \succeq_{emb}\ u$

obtains *ws* **where**

$ws \neq []$
 $hd\ ws = subst\ \varrho\ w$
 $last\ ws = u$
 $\forall i. Suc\ i < length\ ws \longrightarrow$
 $(\exists p\ d. emb_step_at\ p\ d\ (ws\ !\ i) = ws\ !\ Suc\ i \wedge \neg list_all\ (\lambda x. x = Left)\ p)$
 $\forall i. i < length\ ws \longrightarrow head\ (ws\ !\ i) = head\ w \wedge num_args\ (ws\ !\ i) = num_args\ w$
 $\forall i. i < length\ ws \longrightarrow (\forall k. k < num_args\ w \longrightarrow args\ (subst\ \varrho\ w)\ !\ k \succeq_{emb}\ args\ (ws\ !\ i)\ !\ k)$

proof –

define *prio* :: *dir list* \Rightarrow *nat* **where** *prio* = $(\lambda p. if\ list_all\ (\lambda x. x = Left)\ (butlast\ p)\ then\ 0\ else\ 1)$

obtain *ws* **where** *ws_def*:

$ws \neq []$
 $hd\ ws = subst\ \varrho\ w$
 $last\ ws = u$
 $\forall i. Suc\ i < length\ ws \longrightarrow prio_emb_step\ prio\ (ws\ !\ i)\ u = ws\ !\ Suc\ i \wedge ws\ !\ i \rightarrow_{emb}\ ws\ !\ Suc\ i$
using $\langle subst\ \varrho\ w \succeq_{emb}\ u \rangle$ *sequence_prio_emb_steps* **by** *blast*

have *ws_emb_u*: $\forall i. Suc\ i < length\ ws \longrightarrow ws\ !\ i \neq u \wedge ws\ !\ i \succeq_{emb}\ u$

proof (*rule allI*)

{

fix *j* **assume** $Suc\ j < length\ ws$

then have $ws\ !\ (length\ ws - (Suc\ (Suc\ j))) \neq u \wedge ws\ !\ (length\ ws - (Suc\ (Suc\ j))) \succeq_{emb}\ u$

proof (*induction j*)

case 0

have $prio_emb_step\ prio\ (ws\ !\ (length\ ws - Suc\ (Suc\ 0)))\ u = ws\ !\ (length\ ws - (Suc\ 0))$

using $0.prem\ Suc_diff_Suc\ ws_def(4)$ **by** *auto*

then show *?case*

using $0.prem\ One_nat_def\ Suc_diff_Suc\ Suc_lessD\ diff_Suc_less\ emb_step_equiv\ last_conv_nth$
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_size_neq*

by (*metis Suc.IH Suc.prem\ Suc_lessD\ emb_size\ emb_step_is_emb\ emb_trans\ leD*)

qed

}

note 0 = *this*

fix *i*

show $Suc\ i < length\ ws \longrightarrow ws\ !\ i \neq u \wedge ws\ !\ i \succeq_{emb}\ u$ **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 $head\ (ws\ !\ i) = head\ w$

$\wedge num_args\ (ws\ !\ i) = num_args\ w$

$\wedge (\forall k. k < num_args\ w \longrightarrow args\ (subst\ \varrho\ w)\ !\ k \succeq_{emb}\ args\ (ws\ !\ i)\ !\ k)$

$\wedge (Suc\ i < length\ ws \longrightarrow prio\ (prio_emb_pos\ prio\ (ws\ !\ i)\ u) = 1)$

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: $\forall k. k < \text{num\_args } w \longrightarrow \text{args } (\text{subst } \varrho w) ! k \succeq_{\text{emb}} \text{args } (ws ! 0) ! k$ 
  using emb.refl hd_conv_nth ws_def(1) ws_def(2) by fastforce
  have 4:(Suc 0 < length ws  $\longrightarrow$  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.prem1 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.prem1 emb_step_under_args_num_args ih(1) ih(3) prio_def prio_emb_step_def ws_def(4)
zero_neq_one)
  have  $\forall k < \text{num\_args } (ws ! i). \text{args } (ws ! i) ! k \succeq_{\text{emb}} \text{args } (ws ! \text{Suc } i) ! k$ 
  using Suc.prem1 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: $\forall k. k < \text{num\_args } w \longrightarrow \text{args } (\text{subst } \varrho w) ! k \succeq_{\text{emb}} \text{args } (ws ! \text{Suc } i) ! k$ 
  using Suc.hyps Suc.prem1 emb_trans by force
  have  $\bigwedge p q dp dq. \text{prio } (q @ [dq]) < \text{prio } (p @ [dp]) \implies \text{take } (\text{length } p) q \neq p$  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  $\longrightarrow$  prio (prio_emb_pos prio (ws ! Suc i) u) = 1
  by (metis Suc.prem1 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  $\forall i. \text{Suc } i < \text{length } ws \longrightarrow$ 
  ( $\exists p d. \text{emb\_step\_at } p d (ws ! i) = ws ! \text{Suc } i \wedge \neg \text{list\_all } (\lambda x. x = \text{Left}) p$ )
  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  $\varrho s \succeq_{\text{emb}} u$ 
  obtains w where
     $s \succeq_{\text{emb}} w$ 
    subst  $\varrho w \succeq_{\text{emb}} u$ 
    is_Sym (head w)  $\implies \text{head } w = \text{head } u \wedge \text{num\_args } w = \text{num\_args } u \wedge (\forall k. k < \text{num\_args } w \longrightarrow \text{args } (\text{subst } \varrho w) ! k \succeq_{\text{emb}} \text{args } u ! k)$ 
  proof -
  obtain w where w_def:
     $s \succeq_{\text{emb}} w$ 
    subst  $\varrho w \succeq_{\text{emb}} u$ 
     $\forall w'. w \rightarrow_{\text{emb}} w' \longrightarrow \neg \text{subst } \varrho w' \succeq_{\text{emb}} u$ 
  using perform_emb_above_vars0[OF (subst  $\varrho s \succeq_{\text{emb}} u$ )] 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 (λx. 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
  then 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 size_apps: size (apps t ts) = size t + sum_list (map size ts) + length ts
  by (induct ts arbitrary:t; simp)

```

```

lemma size_args_plus_num_args: 1 + sum_list (map size (args t)) + num_args t = size t
  by (metis One_nat_def size_apps tm.size(3) tm_collapse_apps)

```

```

lemma size_chop: is_App t ⟹ Suc (Suc (size (chop t))) = size t
  unfolding size_args_plus_num_args[of t, symmetric] chop_def size_apps
  by (metis Nitpick.size_list_simp(1) ab_semigroup_add_class.add_ac(1) args_Nil_iff_is_Hd plus_1_eq_Suc
  size_list_conv_sum_list)

```

```

lemma size_chop_lt: is_App t ⟹ size (chop t) < size t

```

by (simp add: Suc_le_lessD less_or_eq_imp_le size_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 \implies t \rightarrow_{emb} chop t

proof (induct num_args t - 1 arbitrary:t)

case 0

have nil: num_args t = 0 \implies t \rightarrow_{emb} chop t unfolding chop_def using 0 args_Nil_iff_is_Hd by force

have single: $\bigwedge a. \text{args } t = [a] \implies t \rightarrow_{emb} \text{chop } t$ unfolding chop_def

by (metis 0.prem1 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.prem1 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.prem1 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.prem1 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.prem1 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 (λx. x = Left) p
  shows chop t = s ∨ chop t →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.prem(5) by auto
    have 1: Suc (length p) ≠ num_args (fun t)
    by (metis (no_types, lifting) Cons.prem(1) Cons.prem(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.prem(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.prem(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.prem(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_eq_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  $\rightarrow_{emb}$  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 ( $\lambda x. x = \text{Left}$ ) p
shows let i = num_args t - Suc (length p) in
  head t = head s  $\wedge$  i < num_args t  $\wedge$  args s = take i (args t) @ drop (Suc i) (args t)

```

proof –

```

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.prem(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.prem(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.prem(1) Cons.prem(2) Cons.prem(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.prem(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.prem 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 (smt k_def Cons.prem(4) args.elims args_Nil_iff_is_Hd butlast_snoc diff_Suc_eq_diff_pred
    diff_Suc_less length_Cons length_butlast length_greater_0_conv take_butlast tm.sel(4))
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.prem(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 size_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*) \neq *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* \langle *is_App* *s* \rangle]
by (*metis*)
have *num_args* (*fun* *s*) \leq *num_args* (*fun* (*subst* ρ *s*))
using *fun_subst*[*OF* \langle *is_App* *s* \rangle]
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 \langle *is_App* (*subst* ρ *s*) \rangle *chop_emb_step_at* *le_imp_less_or_eq*
by *fastforce*
then have *num_args* *s* \leq *num_args* (*fun* (*subst* ρ *s*))
using *Suc_num_args*[*OF* \langle *is_App* *s* \rangle] **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* \langle *is_App* *s* \rangle]
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* \langle *is_App* (*subst* ρ *s*) \rangle]
by (*metis* *merge_emb_step_at*)

qed

lemma *chop_subst_emb*:
assumes *is_App* *s*
shows *chop* (*subst* ρ *s*) \triangleright_{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* \langle *is_App* *s* \rangle ,*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* \langle *is_App* (*subst* ρ *s*) \rangle]
using *emb_step_at_subst*[*OF* *chop_position_of*[*OF* \langle *is_App* *s* \rangle]]
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
abbrevs >t = >t
and ≥t = ≥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 ⇒ 's ⇒ bool (infix >s 50) and
  arity_sym :: 's ⇒ enat and
  arity_var :: 'v ⇒ enat +
fixes
  extf :: 's ⇒ (('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool) ⇒ ('s, 'v) tm list ⇒ ('s, 'v) tm list ⇒ bool
assumes
  extf_ext_trans_before_irrefl: ext_trans_before_irrefl (extf 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_ext_snoc: ext_snoc (extf f)
assumes extf_min_empty: ¬ extf f gt [] ss
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_snoc = ext_snoc.snoc[OF extf_ext_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_ext_insert_arg: extf f gt (xs @ z # ys) (xs @ ys)
using extf_compat_append_left extf_compat_append_right extf_snoc[of f gt Nil z]
by fastforce
```

4.2 Inductive Definitions

```
definition
  chkchop :: (('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool) ⇒ ('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool
where
  [simp]: chkchop gt t s ⟷ is_Hd s ∨ 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]: $chkchop_same\ gt\ t\ s \iff$
 (if $is_V\bar{a}r\ (head\ t)$
 then $is_Hd\ t \vee chkchop\ gt\ (chop\ t)\ s$
 else $chkchop\ gt\ t\ s$)

lemma $chkchop_mono[mono]: gt \leq gt' \implies chkchop\ gt \leq chkchop\ gt'$

using $chkchop_def$ by blast

lemma $chkchop_same_mono[mono]: gt \leq gt' \implies chkchop_same\ gt \leq chkchop_same\ gt'$

using $chkchop_same_def$ by fastforce

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

$gt_chop: is_App\ t \implies chop\ t >_t\ s \vee chop\ t = s \implies t >_t\ s$
 $| gt_diff: head\ t >_{hd}\ head\ s \implies is_Sym\ (head\ s) \implies chkchop\ (>_t)\ t\ s \implies t >_t\ s$
 $| gt_same: head\ t = head\ s \implies chkchop_same\ (>_t)\ t\ s \implies$
 $(\forall f \in ground_heads\ (head\ t). extf\ f\ (>_t)\ (args\ t)\ (args\ s)) \implies t >_t\ s$

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

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

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

$gt_chopI: is_App\ t \implies chop\ t \geq_t\ s \implies gt_chop\ t\ s$

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

$gt_diffI: head\ t >_{hd}\ head\ s \implies is_Sym\ (head\ s) \implies chkchop\ (>_t)\ t\ s \implies gt_diff\ t\ s$

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

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

lemma $gt_iff_chop_diff_same: t >_t\ s \iff gt_chop\ t\ s \vee gt_diff\ t\ s \vee gt_same\ t\ s$

by (subst $gt.simps$) (auto simp: $gt_chop.simps\ gt_diff.simps\ gt_same.simps$)

4.3 Transitivity

lemma $t_gt_chop_t: is_App\ t \implies t >_t\ chop\ t$

by (simp add: gt_chop)

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

proof (simp only: $atomize_imp$,

$rule\ measure_induct_rule$ [of $\lambda(t, s). size\ t + size\ s$
 $\lambda(t, s). t >_t\ s \longrightarrow vars\ t \supseteq vars\ s\ (t, s)$, $simplified\ prod.case$],
 $simp\ only: split_paired_all\ prod.case\ atomize_imp[symmetric]$)

fix $t\ s$

assume

$ih: \bigwedge ta\ sa. size\ ta + size\ sa < size\ t + size\ s \implies ta >_t\ sa \implies vars\ ta \supseteq vars\ sa$ and

$t_gt_s: t >_t\ s$

show $vars\ t \supseteq vars\ s$

using t_gt_s

proof cases

case (gt_chop)

thus ?thesis

using ih

by (metis $add_mono_thms_linordered_field(1)\ le_supI1\ order_refl\ size_chop_lt\ vars_chop$)

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)  $\subseteq$  vars t using ih
    using App chkchop_def local.gt_diff(3) nat_add_left_cancel_less size_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 gt_same extf_min_empty[of _ (>t) args s]
    by simp
  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
    by (metis Var args.simps(1) chkchop_def chkchop_same_def epo.extf_min_empty
      epo_axioms gt_hd_def gt_hd_irrefl hd.disc(1) local.gt_same(2) local.gt_same(3) tm.collapse(1)
      tm.disc(2))
  then have vars (chop s)  $\subseteq$  vars (chop t) using ih[OF _ (chop t >t chop s)]
    by (metis App add_mono_thms_linordered_field(5) args_Nil_iff_is_Hd extf_min_empty gt_hd_def
      gt_hd_irrefl local.gt_same(3) size_chop_lt tm.disc(2))
  then show ?thesis using gt_same(1) vars_chop[of t] vars_chop[of s]
    by (metis App args_Nil_iff_is_Hd extf_min_empty gt_hd_def gt_hd_irrefl le_sup_iff local.gt_same(3)
      order_refl sup.coboundedI1 tm.disc(2))
  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 size_chop_lt tm.disc(2) vars_head_subseteq)
qed
qed
qed
qed

lemma gt_trans: u >t t  $\implies$  t >t s  $\implies$  u >t s
proof (simp only: atomize_imp,
  rule measure_induct_rule[of  $\lambda(u, t, s). \{\#size\ u, size\ t, size\ s\}$ 
   $\lambda(u, t, s). u >_t t \longrightarrow t >_t s \longrightarrow u >_t s$  (u, t, s),
  simplified prod.case],
  simp only: split_paired_all prod.case atomize_imp[symmetric])
fix u t s

```

```

assume
  ih:  $\bigwedge u \ ta \ sa. \{\#size \ ua, \ size \ ta, \ size \ sa\# \} < \{\#size \ u, \ size \ t, \ size \ s\# \} \implies$ 
     $ua >_t ta \implies ta >_t sa \implies ua >_t sa$  and
     $u_{gt} \ t: u >_t t$  and  $t_{gt} \ s: t >_t s$ 

have  $u_{gt} \ s \text{ if } ui: chop \ u \geq_t t \implies u >_t s$  if  $ui \ in: is\_App \ u$ 
  using  $ih[of \ chop \ u \ t \ s, \ simplified] \ t_{gt} \ s \ gt\_chop \ size\_chop\_lt \ ui \ in$  by blast

show  $u >_t s$ 
  using  $t_{gt} \ s$ 
proof cases
  case  $gt\_chop$ 
  have  $u_{gt} \ s \text{ if } chk\_u \ t: ?thesis$  if  $chk\_u \ t: chkchop \ (>_t) \ u \ t$ 
  using  $ih[of \ u \ chop \ u \ s] \ gt\_chop \ chk\_u \ t$ 
  by (metis  $add\_mset\_lt\_left\_lt \ add\_mset\_lt\_right\_lt \ chkchop\_def \ ih \ size\_chop\_lt$ )
  show ?thesis
  by (metis  $args\_Nil\_iff\_is\_Hd \ chkchop\_def \ chkchop\_same\_def$ 
     $epo.extf\_min\_empty \ epo\_axioms \ gt.simps \ gt\_hd\_def \ gt\_hd\_irrefl$ 
     $u_{gt} \ s \text{ if } chk\_u \ t \ u_{gt} \ s \text{ if } ui \ u_{gt} \ t$ )
next
  case  $gt\_diff \ t \ s: gt\_diff$ 
  show ?thesis
  using  $u_{gt} \ t$ 
  proof cases
  case  $gt\_chop$ 
  then show ?thesis
  using  $u_{gt} \ s \text{ if } ui$  by blast
next
  case  $gt\_diff \ u \ t: gt\_diff$ 
  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  $add\_mset\_lt\_left\_lt \ add\_mset\_lt\_right\_lt \ chkchop\_def \ gt\_diff \ gt\_diff \ t \ s(3) \ ih \ size\_chop\_lt$ 
 $u_{gt} \ t$ 
  by (metis  $gt\_diff \ t \ s(2)$ )
next
  case  $gt\_same \ u \ t: gt\_same$ 
  have  $head \ u >_{hd} head \ s$ 
  using  $gt\_diff \ t \ s(1) \ gt\_same \ u \ t(1)$  by auto
  thus ?thesis
  using  $add\_mset\_lt\_left\_lt \ add\_mset\_lt\_right\_lt \ chkchop\_def \ gt\_diff \ gt\_diff \ t \ s(3) \ ih \ size\_chop\_lt \ u_{gt} \ t$ 
  by (metis  $gt\_diff \ t \ s(2)$ )
  qed
next
  case  $gt\_same \ t \ s: gt\_same$ 
  show ?thesis
  using  $u_{gt} \ t$ 
  proof cases
  case  $gt\_chop$ 
  then show ?thesis
  using  $u_{gt} \ s \text{ if } ui$  by linarith
next
  case  $gt\_diff \ u \ t: gt\_diff$ 
  have  $head \ u >_{hd} head \ s$ 
  using  $gt\_diff \ u \ t(1) \ gt\_same \ t \ s(1)$  by simp
  thus ?thesis
  using  $add\_mset\_lt\_left\_lt \ add\_mset\_lt\_right\_lt \ chkchop\_def \ gt\_diff \ gt\_same \ t \ s$  ih  $size\_chop\_lt \ u_{gt} \ t$ 
  by (metis  $chkchop\_same\_def \ gt\_diff \ u \ t(2)$ )
next
  case  $gt\_same \ u \ t: gt\_same$ 
  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
      size_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 by auto
  next
  case (App u1 u2)
  then have chop u >t chop t
  by (metis Var args.simps(1) chkchop_def chkchop_same_def epo.extf_min_empty epo_axioms gt_hd_def
    gt_hd_irrefl gt_same_t_s(3) gt_same_u_t(2) hd.disc(1) tm.collapse(1) tm.disc(2))
  then show ?thesis
  proof (cases t)
    case (Hd _)
    then show ?thesis
    using extf_min_empty gt_same_t_s(3) by auto
  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_le less_imp_le mset_lt_single_iff size_chop_lt
    tm.disc(2))
  then show ?thesis unfolding chkchop_same_def
  using Var by auto
  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_size_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  $(\forall f \in local\_ground\_heads\ (head\ u).\ extf\ f\ (>_t)\ (args\ u)\ (args\ s))\ gt\_same\ hd\_u\_s$  by blast
  qed
  qed
  qed

```

4.4 Irreflexivity

```

theorem gt_irrefl:  $\neg s >_t s$ 
proof (standard, induct s rule: measure_induct_rule[of size])
  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 size_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  $si\_in\_args: si \in set\ (args\ s)$  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  $size\ si < size\ s$ 
      by (rule  $size\_in\_args[OF\ si\_in\_args]$ )
    thus False
      by (rule  $ih[OF\ _\ si\_gt\_si]$ )
  qed
  qed

```

```

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

```

4.5 Compatibility with Embedding Relation

```

lemma nth_drop_lemma:
  assumes  $length\ xs = length\ ys$ 
  and  $k \leq length\ xs$ 
  and  $\bigwedge i. i < length\ xs \longrightarrow i \geq k \longrightarrow xs\ !\ i = ys\ !\ i$ 
  shows  $drop\ k\ xs = drop\ k\ ys$ 
  using assms proof (induct arbitrary:k rule: list_induct2)
  case Nil
  then show ?case by simp
next
  case (Cons x xs y ys)
  then show ?case proof (cases k)
  case 0

```

```

then have  $x \# xs = y \# ys$ 
  by (metis Cons.hyps(1) Cons.prem(2) leI length_Cons not_less_zero nth_equalityI)
then show ?thesis
  by blast
next
  case (Suc m)
  then have  $drop\ m\ xs = drop\ m\ ys$ 
  by (metis Cons.hyps(2) Cons.prem(1) Cons.prem(2) Suc_le_mono Suc_mono length_Cons nth_Cons_Suc)
  then show ?thesis
  by (simp add: Suc)
qed
qed

```

lemma gt_embedding_step_property:

```

assumes  $t \rightarrow_{emb} s$ 
shows  $t >_t s$ 
using assms
apply(simp only: atomize_imp)
apply (rule measure_induct_rule[of  $\lambda(t, s). size\ t + size\ s$ 
   $\lambda(t, s). t \rightarrow_{emb} s \longrightarrow t >_t s (t, s)$ , simplified prod.case])
proof(simp only: split_paired_all prod.case atomize_imp[symmetric])
  fix  $s\ t :: ('s, 'v)\ tm$ 
  assume  $t \rightarrow_{emb} s$ 
  and ih:  $\bigwedge tt\ ss. size\ tt + size\ ss < size\ t + size\ s \implies tt \rightarrow_{emb} ss \implies tt >_t s$ 
  have is_App t
  by (metis  $\langle t \rightarrow_{emb} s \rangle emb\_step\_at\_is\_App\ emb\_step\_equiv$ )

obtain  $p\ d$  where  $emb\_step\_at\ p\ d\ t = s\ position\_of\ t\ (p\ @\ [d])$ 
  using  $\langle t \rightarrow_{emb} s \rangle emb\_step\_equiv$  by (metis position_if_emb_step_at)
define  $q$  where  $q\_rep\_t: q = replicate\ (num\_args\ (fun\ t))\ Left$ 

```

```

show  $t >_t s$ 
proof (cases list_all ( $\lambda x. x = Left$ ) p)
  case True
  show ?thesis
  proof (cases d)

```

Embedding removes an argument i

```

  case Left
  define  $i$  where  $i = num\_args\ t - Suc\ (length\ p)$ 
  then have  $head\ t = head\ s\ i < num\_args\ t\ args\ s = take\ i\ (args\ t)\ @\ drop\ (Suc\ i)\ (args\ t)$ 
  using  $emb\_step\_at\_remove\_arg\ Left\ True\ \langle emb\_step\_at\ p\ d\ t = s \rangle\ \langle position\_of\ t\ (p\ @\ [d]) \rangle$ 
  by metis+
  have  $is\_App\ s \implies chop\ t \rightarrow_{emb} chop\ s$ 
  proof (cases  $p = q$ )
  case True
  assume  $is\_App\ s$ 
  show ?thesis
  proof -
  have  $Suc\ (num\_args\ (fun\ s)) = num\_args\ (fun\ t)$ 
  by (metis One_nat_def Suc_num_args True  $\langle args\ s = take\ i\ (args\ t)\ @\ drop\ (Suc\ i)\ (args\ t) \rangle\ \langle is\_App\ s \rangle$ 
   $\langle is\_App\ t \rangle\ append\_self\_conv2\ cancel\_comm\_monoid\_add\_class.diff\_cancel\ diff\_Suc\_1\ i\_def\ length\_drop\ length\_replicate$ 
   $q\_rep\_t\ take\_eq\_Nil$ )
  have  $emb\_step\_at\ (replicate\ (num\_args\ (fun\ s))\ Left)\ Right\ (chop\ t) =$ 
   $emb\_step\_at\ (replicate\ (num\_args\ (fun\ s))\ Left)\ Right\ t$ 
  using  $merge\_emb\_step\_at$ [of  $replicate\ (num\_args\ (fun\ s))\ Left\ Right\ Nil\ Right\ t$ , unfolded  $append\_Nil2$ 
   $opp\_simps(1)\ replicate\_append\_same$ ]
  by (metis  $\langle Suc\ (num\_args\ (fun\ s)) = num\_args\ (fun\ t) \rangle\ \langle is\_App\ t \rangle\ chop\_emb\_step\_at\ replicate\_Suc$ )
  then have  $emb\_step\_at\ (replicate\ (num\_args\ (fun\ s))\ dir.Left)\ dir.Right\ (chop\ t) = chop\ s$  unfolding
   $chop\_emb\_step\_at$ [OF  $\langle is\_App\ s \rangle$ ]
  using  $merge\_emb\_step\_at$ [of  $replicate\ (num\_args\ (fun\ s))\ Left\ Right\ Nil\ Left\ t$ , unfolded  $append\_Nil2$ 
   $opp\_simps(1)\ replicate\_append\_same$ ]
  by (metis Left True  $\langle Suc\ (num\_args\ (fun\ s)) = num\_args\ (fun\ t) \rangle\ \langle emb\_step\_at\ p\ d\ t = s \rangle\ q\_rep\_t$ 

```

```

replicate_Suc)
  then show chop t →emb chop s
    by (metis ‹is_App s› ‹is_App t› ‹t →emb s› emb_step_equiv emb_step_size nat_neq_iff size_chop)
  qed
next
case False
assume is_App s
have p_rep: p = replicate (length p) Left
  by (metis (full_types) ‹list_all (λx. x = Left) p› list_all_iff replicate_length_same)
have length_p: length p < num_args t using no_position_replicate_num_args ‹position_of t (p @ [d])›
  replicate_add[of num_args t length p - num_args t Left] p_rep q_rep_t
  by (metis Left add_diff_inverse_nat replicate_app_Cons_same replicate_append_same shallower_pos)
then have length p ≤ length q
  using Suc_num_args ‹is_App t› q_rep_t by fastforce
then have length p < length q
  using False le_neq_implies_less p_rep q_rep_t by fastforce
then have take (Suc (length p)) q = p @ [Left]
  by (metis (no_types, lifting) ‹length p ≤ length q› length_replicate min.orderE nth_replicate p_rep q_rep_t
take_Suc_conv_app_nth take_replicate)
then obtain q' where q = p @ [Left] @ q'
  by (metis append.assoc append_take_drop_id)
have Suc (num_args (fun s)) = num_args (fun t)
  by (metis (no_types, lifting) Cons_nth_drop_Suc Suc_num_args ‹args s = take i (args t) @ drop (Suc i)
(args t)› ‹i < num_args t› ‹is_App s› ‹is_App t› add_Suc_right append_take_drop_id diff_Suc_1 length_Cons
length_append)
then have chop s = emb_step_at p dir.Left (chop t)
  using swap_nested_emb_step_at[of p q' Right Left t] chop_emb_step_at[OF ‹is_App s›]
  chop_emb_step_at[OF ‹is_App t›]
  by (metis (no_types, lifting) Cons_replicate_eq Left ‹emb_step_at p d t = s› ‹q = p @ [dir.Left] @ q'›
append.assoc append_Cons diff_Suc_1 p_rep q_rep_t replicate_append_same)
then show chop t →emb chop s
  by (metis Left ‹is_App t› ‹position_of t (p @ [d])› ‹q = p @ [dir.Left] @ q'› chop_emb_step_at
emb_step_at_if_position pos_emb_step_at_nested q_rep_t)
qed
then have chkchop_same (>t) t s
proof (cases is_Var (head t))
case True
then show ?thesis unfolding chkchop_same_def chkchop_def using ih[of chop t chop s]
  add_less_mono size_chop_lt ‹is_App s ⇒ chop t →emb chop s› by metis
next
case False
then show ?thesis unfolding chkchop_same_def chkchop_def
  using ‹is_App t› add_less_mono gt_chop ih size_chop_lt
  ‹is_App s ⇒ chop t →emb chop s› by metis
qed
have ∀f∈local.ground_heads (head t). extf f (>t) (args t) (args s)
  using extf_ext_insert_arg[of _ _ take i (args t) args t ! i drop (Suc i) (args t)]
  using ‹args s = take i (args t) @ drop (Suc i) (args t)› ‹i < num_args t› id_take_nth_drop by fastforce
then show ?thesis using gt_same ‹head t = head s› ‹chkchop_same (>t) t s› by blast
next

```

Embedding chops and might remove arguments from the left

```

case Right
show ?thesis using emb_step_at_chop
  by (metis Right True ‹emb_step_at p d t = s› ‹is_App t› ‹position_of t (p @ [d])› add_Suc gt_chop ih
less_Suc_eq size_chop)
qed
next

```

Embedding operates under one of the arguments

```

case False
have num_args t = num_args s using emb_step_under_args_num_args[OF False]
  by (metis (no_types) ‹∧ t d. num_args (emb_step_at p d t) = num_args t› ‹emb_step_at p d t = s›)

```

```

then have is_App s using ⟨num_args t = num_args s⟩
  by (metis args_Nil_iff_is_Hd length_0_conv ⟨is_App t⟩)

have q_rep_s: q = replicate (num_args (fun s)) Left
by (metis q_rep_t ⟨is_App s⟩ ⟨is_App t⟩ ⟨num_args t = num_args s⟩ args.simps(2) butlast_snoc length_butlast
tm.collapse(2))

have chop t  $\rightarrow_{emb}$  chop s
proof (cases take (num_args (fun t)) p = q)
  case True
    have num_args (fun t) < length p
    proof (rule ccontr)
      assume  $\neg$  num_args (fun t) < length p
      then have num_args (fun t) = length p q = p
        using True q_rep_t by auto
      then show False using False
        using list_all_length q_rep_t by fastforce
    qed

    have p ! (num_args (fun t)) = Right
    proof (rule ccontr)
      assume p ! num_args (fun t)  $\neq$  Right
      then have p ! num_args (fun t) = Left using dir.exhaust by blast
      then have take (num_args t) p = replicate (num_args t) Left using True Suc_num_args[OF ⟨is_App t⟩
q_rep_t
        take_Suc_conv_app_nth[of num_args (fun t) p]
        by (metis ⟨num_args (fun t) < length p⟩ replicate_Suc replicate_append_same)
      then show False
        by (metis ⟨num_args t = num_args s⟩ ⟨num_args t = num_args s⟩ ⟨position_of t (p @ [d])⟩
          append.assoc append_eq_Cons_conv append_take_drop_id no_position_replicate_num_args shal-
lower_pos)
    qed

then obtain q' where p = q @ [Right] @ q'
by (metis Cons_nth_drop_Suc True ⟨num_args (fun t) < length p⟩ append_Cons append_Nil append_eq_conv_conj
length_replicate q_rep_t)
have emb_step_at (q @ q') d (chop t) = chop s
  unfolding chop_emb_step_at[OF ⟨is_App t⟩ chop_emb_step_at[OF ⟨is_App s⟩]
  using swap_nested_emb_step_at[of q q' d Right t, unfolded] ⟨emb_step_at p d t = s⟩ ⟨p = q @ [Right] @
q'
  q_rep_t q_rep_s by auto
moreover have chop t  $\neq$  chop s
  by (metis ⟨is_App s⟩ ⟨is_App t⟩ t  $\rightarrow_{emb}$  s) emb_step_size nat_less_le size_chop)
ultimately show chop t  $\rightarrow_{emb}$  chop s
  using emb_step_equiv by blast
next
  case False
  then have takepq: take (length q) p  $\neq$  q
    using q_rep_t by auto
  have takeqp: length p  $\leq$  length q  $\implies$  take (length p) q  $\neq$  p
    using ( $\neg$  list_all ( $\lambda x. x = Left$ ) p) [unfolded list_all_length]
    using diff_diff_cancel take_replicate length_replicate nth_replicate q_rep_s by metis
  have chop s = emb_step_at p d (chop t)
    using swap_disjunct_emb_step_at[of p q Right d t, OF takeqp takepq, unfolded] ⟨emb_step_at p d t = s⟩
    using ⟨is_App s⟩ ⟨is_App t⟩ chop_emb_step_at q_rep_s q_rep_t
    by (simp add: chop_emb_step_at)
  then show ?thesis
    by (metis False ⟨is_App t⟩ ⟨position_of t (p @ [d])⟩ chop_emb_step_at emb_step_at_if_position length_replicate
nat_le_linear pos_emb_step_at_disjunct q_rep_t take_all takeqp)
  qed
then have gt1:chkchop_same ( $>$ ) t s
proof (cases head t)
  case (Var _)

```



```

then have chop t >t chop s using ih[of chop t chop s]
  by (meson ⟨chop t →emb chop s⟩ ⟨is_App s⟩ ⟨is_App t⟩ add_strict_mono size_chop_lt)
then show ?thesis unfolding chkchop_same_def
  by (simp add: Var)
next
case (Sym _)
then show ?thesis unfolding chkchop_same_def using ⟨chop t →emb chop s⟩
  ⟨is_App t⟩ add_Suc add_Suc_shift chkchop_def gt_trans ih
  less_Suc_eq size_chop_t gt_chop_t
  by (metis (no_types, lifting))
qed
have gt2:head t = head s
  by (metis emb_step_under_args_head False ⟨emb_step_at p d t = s⟩)
have gt3:∀ f∈local.ground_heads (head t). extf f (>t) (args t) (args s)
proof (rule ballI)
  fix f assume f∈local.ground_heads (head t)
  obtain i where i_def:
    i < num_args t
    args t ! i →emb args (emb_step_at p d t) ! i
    ∧j. j < num_args t ⇒ i ≠ j ⇒ args t ! j = args (emb_step_at p d t) ! j
  using emb_step_under_args_emb_step[of p t d]
  using False ⟨position_of t (p @ [d])⟩ by blast
  have compat_list1: args t ! i >t args s ! i
    by (metis ⟨num_args t = num_args s⟩ ⟨emb_step_at p d t = s⟩ add_less_mono i_def(1) i_def(2) ih
nth_mem size_in_args)
  have compat_list2: args t ! i ≠ args s ! i
    using emb_step_equiv i_def(2) ⟨emb_step_at p d t = s⟩ by blast
  have argst:args t = take i (args t) @ args t ! i # drop (Suc i) (args t)
    by (simp add: Cons_nth_drop_Suc i_def(1))
  have argss:args s = take i (args t) @ args (emb_step_at p d t) ! i # drop (Suc i) (args t)
proof -
  have take i (args t) = take i (args s)
    apply (rule nth_take_lemma)
  using ⟨num_args t = num_args s⟩ i_def(1) i_def(3)[unfolded ⟨emb_step_at p d t = s⟩] by auto
  moreover have drop (Suc i) (args t) = drop (Suc i) (args s)
    apply (rule nth_drop_lemma)
  apply (simp add: ⟨num_args t = num_args s⟩)
  apply (simp add: Suc_le_eq i_def(1))
  using Suc_n_not_le_n ⟨emb_step_at p d t = s⟩ i_def(3) by blast
  ultimately show ?thesis
    using ⟨emb_step_at p d t = s⟩ ⟨num_args t = num_args s⟩ i_def(1) id_take_nth_drop by auto
qed
show extf f (>t) (args t) (args s)
  using extf_compat_list[of args t ! i args (emb_step_at p d t) ! i gt f take i (args t) drop (Suc i) (args t)]
  using ⟨emb_step_at p d t = s⟩ argss argst compat_list1 compat_list2 by force
qed
show ?thesis using gt_same using gt1 gt2 gt3 by blast
qed
qed

```

lemma gt_embedding_property:

```

assumes t ≥emb s t ≠ 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

```

4.6 Subterm Property

theorem *gt_proper_sub*: $\text{proper_sub } s \ t \implies t >_t s$
using *gt_embedding_property sub_emb* **by** *blast*

lemma

gt_emb_fun: $\text{App } s \ t >_t s$ **and**
gt_emb_arg: $\text{App } s \ t >_t t$
by (*simp_all add: gt_embedding_step_property left right*)

4.7 Compatibility with Contexts

lemma *gt_fun_imp*: $\text{fun } t >_t s \implies t >_t s$
by (*metis emb_step_fun gt_embedding_step_property gt_trans tm.disc(2) tm.exhaust_sel tm.sel(3)*)

lemma *gt_arg_imp*: $\text{arg } t >_t s \implies t >_t s$
by (*metis emb_step_arg gt_embedding_step_property gt_trans tm.disc(2) tm.exhaust_sel tm.sel(5)*)

lemma *gt_compat_fun*:

assumes $t' >_t t$
shows $\text{App } s \ t' >_t \text{App } s \ t$

using *assms apply (simp only: atomize_imp)*

proof (*induction rule: measure_induct_rule*[of $\lambda(t, s). \text{size } t + \text{size } s$]. $t' >_t t \longrightarrow \text{App } s \ t' >_t \text{App } s \ t (t, s)$,

simplified prod.case],

simp only: split_paired_all prod.case atomize_imp[*symmetric*])

fix $t \ s :: ('s, 'v) \text{tm}$

assume *ih*: $\bigwedge ta \ sa. \text{size } ta + \text{size } sa < \text{size } t + \text{size } s \implies t' >_t ta \implies \text{App } sa \ t' >_t \text{App } sa \ t$
and $t' \text{_gt_} t: t' >_t t$

have $t' \text{_ne_} t: t' \neq t$

using *gt_antisym t'_gt_t* **by** *blast*

have *extf_args_single*: $\forall f \in \text{ground_heads } (\text{head } s). \text{extf } f (>_t) (\text{args } s @ [t']) (\text{args } s @ [t])$

by (*simp add: extf_compat_list t'_gt_t t'_ne_t*)

show $\text{App } s \ t' >_t \text{App } s \ t$

proof (*rule gt_same*)

show $\text{head } (\text{App } s \ t') = \text{head } (\text{App } s \ t)$ **by** *simp*

show $\forall f \in \text{local.ground_heads } (\text{head } (\text{App } s \ t')). \text{extf } f (>_t) (\text{args } (\text{App } s \ t')) (\text{args } (\text{App } s \ t))$

by (*simp add: extf_args_single*)

have $0: \text{chop } (\text{App } s \ t') >_t \text{chop } (\text{App } s \ t)$

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 size_chop_lt t'_gt_t tm.disc(2) tm.sel(4) tm.sel(6)*)

qed

show *chkchop_same* $(>_t) (\text{App } s \ t') (\text{App } s \ t)$

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 $\text{App } s \ t' >_t \text{chop } (\text{App } s \ t)$ **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  $s' >_t s \implies t' \geq_t t \implies \text{App } s' t' >_t \text{App } s t$ 
proof (simp only: atomize_imp, induction rule: measure_induct[of  $\lambda(s',s,t). \text{size } s' + \text{size } s + \text{size } t \lambda(s',s,t)$ ].  $s' >_t s \longrightarrow t' \geq_t t \longrightarrow \text{App } s' t' >_t \text{App } s t (s',s,t)$ , simplified prod.case),
  simp only: split_paired_all prod.case atomize_imp[symmetric] atomize_all[symmetric])
fix  $s' s t$ 
assume  $ih: \bigwedge ab ac ba. \text{size } ab + \text{size } ac + \text{size } ba < \text{size } s' + \text{size } s + \text{size } t \implies ab >_t ac \implies t' \geq_t ba \implies \text{App } ab t' >_t \text{App } ac ba$ 
  and  $s' >_t s$  and  $t' \geq_t t$ 

{
  fix  $s''::('s,'v) \text{tm}$  assume  $\text{size } s'': \text{size } s'' \leq \text{size } s'$ 
  assume  $\text{chkchop } s' s: \text{chkchop } (>_t) s'' s$ 
  then have  $\text{chkchop } (>_t) (\text{App } s'' t') (\text{App } s t)$ 
  proof (cases is_Hd s)
    case True
    then show ?thesis using  $\text{chkchop } s' s$  unfolding  $\text{chkchop\_def}$ 
      by (metis  $\langle t' \geq_t t \rangle \text{ chop\_App\_Hd } gt\_arg\_imp \text{ gt\_emb\_arg } tm.sel(6)$ )
    next
    case False
    then show ?thesis using  $\text{chkchop } s' s$  unfolding  $\text{chkchop\_def}$ 
      using  $ih[\text{of } s'' \text{ chop } s t] \text{ size } s''$ 
      by (metis  $\langle t' \geq_t t \rangle \text{ add\_less\_mono } \text{add\_mono\_thms\_linordered\_field}(1) \text{ chop\_fun } le\_eq\_less\_or\_eq \text{ nat\_add\_left\_cancel\_less } \text{size\_chop\_lt } tm.sel(4) \text{ tm.sel}(6)$ )
  qed
}
note  $\text{chkchop\_compat\_arg} = \text{this}$ 

show  $\text{App } s' t' >_t \text{App } s t$  using  $\langle s' >_t s \rangle$ 
proof (cases rule:gt.cases)
  case gt_chop
  then show ?thesis
  proof (cases  $t = t'$ )
    case True
    show ?thesis using True  $gt.gt\_chop[\text{of } \text{App } s' t' \text{App } s t] \text{ gt\_chop } \text{chkchop\_compat\_arg}[\text{of } \text{chop } s']$ 
      by (metis  $\text{add\_strict\_right\_mono } \text{chop\_fun } ih \text{ size\_chop\_lt } tm.disc(2) \text{ tm.sel}(4) \text{ tm.sel}(6)$ )
    next
    case False
    then have  $t' >_t t$ 
      using  $\langle t' \geq_t t \rangle$  by blast
    have  $\text{App } s' t' >_t \text{App } (\text{chop } s') t'$ 
      by (simp add: context_left emb_step_chop gt_embedding_step_property local.gt_chop(1))
    moreover have  $\dots >_t \text{App } s t$  using  $ih[\text{of } \text{chop } s' s t]$ 
      using  $\langle t' >_t t \rangle \text{ gt\_compat\_fun } local.gt\_chop(1) \text{ local.gt\_chop}(2) \text{ size\_chop\_lt}$  by fastforce
    ultimately show ?thesis using  $gt\_trans$  by blast
  qed
next
  case gt_diff
  then show ?thesis
    using  $\text{chkchop\_compat\_arg } gt.gt\_diff$  by auto
next
  case gt_same
  have  $hd\_s'\_eq\_s: \text{head } s' = \text{head } s$ 
    by (simp add: local.gt_same(1))
  {
    fix  $f$  assume  $f\_gh: f \in \text{ground\_heads } (\text{head } s)$ 
    have  $f\_s\_args:$ 
      extf  $f (>_t) (\text{args } s') (\text{args } s)$ 
      using  $local.gt\_same(3) f\_gh$  by (simp add:  $hd\_s'\_eq\_s$ )
    have  $f\_compat\_snoc:$ 
       $\bigwedge xs \ ys \ x. \text{extf } f (>_t) \ ys \ xs \implies \text{extf } f (>_t) (\ys @ [x]) (xs @ [x])$ 
      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 args.simps(1) chop_fun chkchop_compat_arg[of chop s', unfolded le_eq_less_or_eq]
  chkchop_compat_arg[of s'] chkchop_def chkchop_same_def
  size_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.collapse(1) tm.sel(4) tm.sel(6)
  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

```

4.8 Stability under Substitutions

```

lemma extf_map2:
  assumes
    ∀ y ∈ set ys ∪ set xs. ∀ x ∈ set ys ∪ set xs. y >t x → (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)+

```

lemma *less_multiset_doubletons*:

```

assumes
  y < t ∨ y < s
  x < t ∨ x < s
shows
  {# y, x#} < {# t, s#}
unfolding less_multiset_DM
proof (intro exI)
  let ?X = {# t, s#}
  let ?Y = {# y, x#}
  show ?X ≠ {#} ∧ ?X ⊆# {# t, s#} ∧ {# y, x#} = {# t, s#} - ?X + ?Y ∧ (∀ k. k ∈# ?Y → (∃ a. a ∈#
  ?X ∧ k < a))
  using add_eq_conv_diff assms(1) assms(2) by auto
qed

```

theorem *gt_sus*:

```

assumes ρ_wary: wary_subst ρ
assumes ghd: ∧x. ground_heads (Var x) = UNIV
shows t >t s ⇒ subst ρ t >t subst ρ s
proof (simp only: atomize_imp, induction rule: measure_induct[of λ(t,s). {# size t, size s #} λ(t,s). t >t s →
  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: ∧tt ss.
  {# size tt, size ss #} < {# size t, size s #} ⇒
  tt >t ss ⇒ 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_size 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 size_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:  $\forall f \in \text{ground\_heads } (\text{head } t). \text{extf } f \ (\>_t) \ (\text{map } (\text{subst } \varrho) \ (\text{args } t)) \ (\text{map } (\text{subst } \varrho) \ (\text{args } s))$ 
proof -
have ih_args:  $\forall y \in \text{set } (\text{args } t) \cup \text{set } (\text{args } s). \forall x \in \text{set } (\text{args } t) \cup \text{set } (\text{args } s). y \>_t x \longrightarrow \text{subst } \varrho \ y \>_t \text{subst } \varrho \ x$ 
  by (metis Un_iff less_multiset_doubletons size_in_args ih)
have  $\forall f \in \text{ground\_heads } (\text{head } t). \text{extf } f \ (\>_t) \ (\text{args } t) \ (\text{args } s)$ 
  using ghd_t_gt_s_same(3) by metis
then show ?thesis
  using extf_map[of set (args t)  $\cup$  set (args s) args t args s gt subst  $\varrho$ ]
  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)  $\subseteq$  ground_heads (head t) size u  $\leq$  size (subst  $\varrho$  (Hd (head t)))
    then have apps u (map (subst  $\varrho$ ) (args t))  $\>_t$  apps u (map (subst  $\varrho$ ) (args s))
    proof (induct size 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
          apply (rule gt_same)
          using extf_map_ts args_Nil_iff_is_Hd s_Hd u_Hd  $\langle$ args u = [] $\rangle$  less by fastforce+
        next
          case s_App: (App _ _)
          then have is_App t
            by (metis args_Nil_iff_is_Hd extf_min_empty gt_hd_def gt_hd_irrefl t_gt_s_same(3))
          have chop t  $\>_t$  chop s
            using  $\langle$ is_App t $\rangle$   $\langle$ is_Var (head t) $\rangle$  s_App t_gt_s_same(2) by auto
          then have subst  $\varrho$  (chop t)  $\>_t$  subst  $\varrho$  (chop s) using ih
            by (metis  $\langle$ is_App t $\rangle$  less_multiset_doubletons s_App size_chop_lt tm.disc(2))

          define ut where ut = apps u (map (subst  $\varrho$ ) (args t))
          define us where us = apps u (map (subst  $\varrho$ ) (args s))
          have 0:  $\bigwedge ss. \text{args } (\text{apps } u \ ss) = ss$ 
            using  $\langle$ args u = [] $\rangle$  by simp
          have chop_us: chop us = subst  $\varrho$  (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  $\varrho$  (chop t)
            unfolding chop_def subst_apps ut_def 0 using  $\langle$ is_App t $\rangle$ 
            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  $\langle$ > $\rangle$  ut us unfolding chkchop_def chkchop_same_def
            by (metis 0 UNIV_witness  $\langle$ is_Var (head t) $\rangle$   $\langle$ subst  $\varrho$  (chop t) $\rangle$   $\>_t$  subst  $\varrho$  (chop s))
            args.simps(1) chop_us chop_ut extf_map_ts extf_min_empty
            ghd_gt_chop is_Var_def tm.collapse(1) ut_def)
          moreover have  $\forall f \in \text{local.ground\_heads } (\text{head } ut). \text{extf } f \ (\>_t) \ (\text{args } ut) \ (\text{args } us)$ 
            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] size_chop_lt
  by (metis Var dual_order.trans ghd less.prem(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 UNIV_I ⟨is_Var (head t)⟩ args_Nil_iff_is_Hd args_apps extf_compat_append_left
    extf_map_ts extf_min_empty ghd gt_chop is_Var_def)
have 3:∀f∈local.ground_heads (head ?ut). extf f (>t) (args ?ut) (args ?us)
  using extf_compat_append_left extf_map_ts less.prem(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)
    size_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*: $\bigwedge f. \text{ext_total } (extf\ f)$

shows $ground\ t \implies ground\ s \implies t >_t s \vee s >_t t \vee t = s$

proof (*simp only*: *atomize_imp*,

rule measure_induct_rule[of $\lambda(t, s). \text{size } t + \text{size } s$

$\lambda(t, s). \text{ground } t \longrightarrow \text{ground } s \longrightarrow t >_t s \vee s >_t t \vee t = s (t, s)$, *simplified prod.case*],

simp only: *split_paired_all prod.case atomize_imp*[*symmetric*])

fix $t\ s :: ('s, 'v)\ tm$

assume

ih: $\bigwedge ta\ sa. \text{size } ta + \text{size } sa < \text{size } t + \text{size } s \implies \text{ground } ta \implies \text{ground } sa \implies$

$ta >_t sa \vee sa >_t ta \vee ta = sa$ **and**

gr_t: *ground t* **and** *gr_s*: *ground s*

let $?case = t >_t s \vee s >_t t \vee t = s$

have *chkchop* (>_t) $t\ s \vee s >_t t$

unfolding *chkchop_def* *tm.case_eq_if* **using** *ih*[of *t fun s*] *ih*[of *t arg s*]

by (*metis gr_s gr_t gt_chop gt_imp_vars ih nat_add_left_cancel_less size_chop_lt subset_empty*)

moreover **have** *chkchop* (>_t) $s\ t \vee t >_t s$

```

unfolding chkchop_def tm.case_eq_if using ih[of fun t s] ih[of arg t s]
by (metis add_less_cancel_right gr_s gr_t gt_chop gt_imp_vars ih size_chop_lt subset_empty)
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:  $\forall ta \in \text{set } ?ts. \text{ground } ta$ 
      using ground_args[OF gr_t] by blast
    have gr_ss:  $\forall sa \in \text{set } ?ss. \text{ground } sa$ 
      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]
        by (metis add_strict_mono in_listsI size_in_args)
  }
  ultimately have ?case
    using gt_sym_total by blast
}

```


ultimately show ?case
 by fast
 qed

4.10 Well-foundedness

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

theorem gt_wf :
 assumes $ghd_UNIV: \bigwedge x. ground_heads_var\ x = UNIV$
 assumes $extf_wf: \bigwedge f. ext_wf\ (extf\ f)$
 shows $wfP\ (\lambda s\ t. t >_t s)$

proof -

have $ground_wfP: wfP\ (\lambda s\ t. t >_{tg}\ s)$
 unfolding $wfP_iff_no_inf_chain$

proof

assume $\exists f. inf_chain\ (>_{tg})\ f$
 then obtain t where $t_bad: bad\ (>_{tg})\ t$
 unfolding $inf_chain_def\ bad_def$ by blast

let $?ff = worst_chain\ (>_{tg})\ (\lambda t\ s. size\ t > size\ s)$
 let $?U_of = \lambda i. \{u. (?ff\ i) \triangleright_{emb}\ u\}$

note $wf_sz = wf_app[OF\ wellorder_class.wf, of\ size, simplified]$

define U where $U = (\bigcup i. ?U_of\ i)$

have $gr: \bigwedge i. ground\ (?ff\ i)$
 using $worst_chain_bad[OF\ wf_sz\ t_bad, unfolded\ inf_chain_def]$ by fast
 have $gr_u: \bigwedge u. u \in U \implies ground\ u$ unfolding U_def
 using $gr\ ground_emb$ by fastforce

have $\neg bad\ (>_{tg})\ u$ if $u_in: u \in ?U_of\ i$ for $u\ i$

proof

let $?ti = ?ff\ i$

assume $u_bad: bad\ (>_{tg})\ u$
 have $sz_u: size\ u < size\ ?ti$
 using $emb_size_neq\ u_in$ 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 $u_good: \bigwedge u. u \in U \implies \neg bad\ (>_{tg})\ u$

unfolding U_def by blast

have $bad_diff_same: inf_chain\ (\lambda t\ s. ground\ t \wedge (gt_diff\ t\ s \vee gt_same\ t\ s))\ ?ff$

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 =  $\lambda t s. t \in U \wedge t >_t s$ 
thm UnionI CollectI
have  $t \in \text{set } (\text{args } (?ff\ i)) \implies t \in U$  for t i
  unfolding U_def apply (rule UnionI[of ?U_of i])
  using arg_emb CollectI arg_emb size_in_args by fast+
moreover have  $\bigwedge i. \text{extf } f (>_{t_g}) (\text{args } (?ff\ (i + k))) (\text{args } (?ff\ (Suc\ i + k)))$ 
  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) ( $\lambda t s. \text{ground } t \wedge t >_t s$ )] ground_hd_in_ground_heads
  by (metis (no_types, lifting) ground_args)
ultimately have  $\bigwedge i. \text{extf } f ?gtu (\text{args } (?ff\ (i + k))) (\text{args } (?ff\ (Suc\ i + k)))$ 
  using extf_mono_strong[of _ _ ( $\lambda t s. \text{ground } t \wedge t >_t s$ )  $\lambda t s. t \in U \wedge t >_t s]$  unfolding U_def by blast
hence inf_chain (extf f ?gtu) ( $\lambda i. \text{args } (?ff\ (i + k))$ )
  unfolding inf_chain_def by blast
hence nwf_ext:  $\neg \text{wfP } (\lambda xs\ ys. \text{extf } f ?gtu\ ys\ xs)$ 
  unfolding wfP_iff_no_inf_chain by fast

have gtu_le_gtg: ?gtu  $\leq$  ( $>_{t_g}$ )
  by (auto intro!: gr_u)

have wfP ( $\lambda s\ t. ?gtu\ t\ s$ )
  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  $\in U$ 
  using bad_f unfolding inf_chain_def by blast
hence good_f0:  $\neg \text{bad } ?gtu\ (f\ 0)$ 
  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 ( $\lambda xs\ ys. \text{extf } f ?gtu\ ys\ xs$ )
  by (rule ext_wf.wf[OF extf_wf, rule_format])

show False
  using nwf_ext wf_ext by blast
qed

let ?subst = subst grounding_0

have wfP ( $\lambda s\ t. ?subst\ t >_{t_g}\ ?subst\ s$ )
  by (rule wfP_app[OF ground_wfP])
hence wfP ( $\lambda s\ t. ?subst\ t >_t\ ?subst\ s$ )

```

```
  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
```