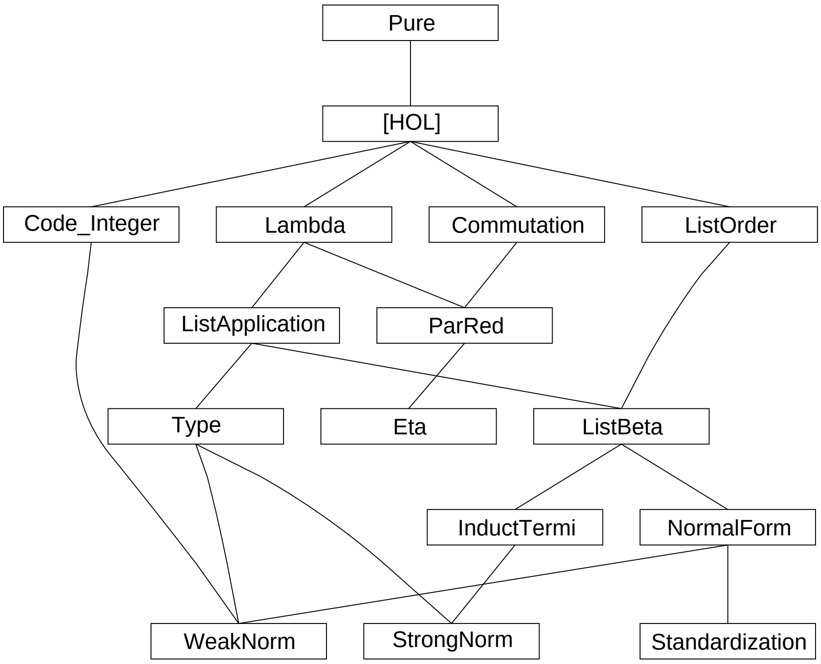
Fundamental Properties of Lambda-calculus

Tobias Nipkow
Stefan Berghofer

April 19, 2009

Contents


Pure
[HOL]
Code_Integer
Lambda
Commutation
ListOrder
ListApplication
ParRed
Type
Eta
ListBeta
InductTermi
NormalForm
WeakNorm
StrongNorm
Standardization

1 Basic definitions of Lambda-calculus

theory *Lambda* **imports** *Main* **begin**

1.1 Lambda-terms in de Bruijn notation and substitution

datatype *dB* =
  Var nat
| *App dB dB* (**infixl** ° *200*)
| *Abs dB*

primrec
lift :: [*dB*, *nat*] => *dB*
where
  lift (*Var i*) *k* = (*if i* < *k then Var i else Var* (*i* + *1*))
| *lift* (*s* ° *t*) *k* = *lift s k* ° *lift t k*
| *lift* (*Abs s*) *k* = *Abs* (*lift s* (*k* + *1*))

primrec
subst :: [*dB*, *dB*, *nat*] => *dB* (-[-'/-] [*300*, *0*, *0*] *300*)
where
  subst-Var: (*Var i*)[*s*/*k*] =
    (*if k* < *i then Var* (*i* − *1*) *else if i* = *k then s else Var i*)
| *subst-App*: (*t* ° *u*)[*s*/*k*] = *t*[*s*/*k*] ° *u*[*s*/*k*]
| *subst-Abs*: (*Abs t*)[*s*/*k*] = *Abs* (*t*[*lift s 0* / *k*+*1*])

declare *subst-Var* [*simp del*]

Optimized versions of *subst* and *lift*.

primrec
liftn :: [*nat*, *dB*, *nat*] => *dB*
where
  liftn n (*Var i*) *k* = (*if i* < *k then Var i else Var* (*i* + *n*))
| *liftn n* (*s* ° *t*) *k* = *liftn n s k* ° *liftn n t k*
| *liftn n* (*Abs s*) *k* = *Abs* (*liftn n s* (*k* + *1*))

primrec
substn :: [*dB*, *dB*, *nat*] => *dB*
where
  substn (*Var i*) *s k* =
    (*if k* < *i then Var* (*i* − *1*) *else if i* = *k then liftn k s 0 else Var i*)
| *substn* (*t* ° *u*) *s k* = *substn t s k* ° *substn u s k*
| *substn* (*Abs t*) *s k* = *Abs* (*substn t s* (*k* + *1*))

1.2 Beta-reduction

inductive *beta* :: [*dB*, *dB*] => *bool* (**infixl** $\rightarrow_\beta$ *50*)
where
  beta [*simp*, *intro*!]: *Abs s* ° *t* $\rightarrow_\beta$ *s*[*t*/*0*]
| *appL* [*simp*, *intro*!]: *s* $\rightarrow_\beta$ *t* ==> *s* ° *u* $\rightarrow_\beta$ *t* ° *u*

| *appR* [*simp*, *intro*!]: $s \to_\beta t ==> u \circ s \to_\beta u \circ t$
| *abs* [*simp*, *intro*!]: $s \to_\beta t ==> Abs\ s \to_\beta Abs\ t$

abbreviation
beta-reds :: [*dB*, *dB*] => *bool* (**infixl** −>> *50*) **where**
$s -\!>> t == beta\hat{}** \ s\ t$

notation (*latex*)
beta-reds (**infixl** $\to_\beta{}^*$ *50*)

inductive-cases *beta-cases* [*elim*!]:
$Var\ i \to_\beta t$
$Abs\ r \to_\beta s$
$s \circ t \to_\beta u$

declare *if-not-P* [*simp*] *not-less-eq* [*simp*]
— don't add *r-into-rtrancl*[*intro*!]

1.3 Congruence rules

lemma *rtrancl-beta-Abs* [*intro*!]:
$s \to_\beta{}^* s' ==> Abs\ s \to_\beta{}^* Abs\ s'$
⟨*proof*⟩

lemma *rtrancl-beta-AppL*:
$s \to_\beta{}^* s' ==> s \circ t \to_\beta{}^* s' \circ t$
⟨*proof*⟩

lemma *rtrancl-beta-AppR*:
$t \to_\beta{}^* t' ==> s \circ t \to_\beta{}^* s \circ t'$
⟨*proof*⟩

lemma *rtrancl-beta-App* [*intro*]:
$[|\ s \to_\beta{}^* s';\ t \to_\beta{}^* t'\ |] ==> s \circ t \to_\beta{}^* s' \circ t'$
⟨*proof*⟩

1.4 Substitution-lemmas

lemma *subst-eq* [*simp*]: $(Var\ k)[u/k] = u$
⟨*proof*⟩

lemma *subst-gt* [*simp*]: $i < j ==> (Var\ j)[u/i] = Var\ (j - 1)$
⟨*proof*⟩

lemma *subst-lt* [*simp*]: $j < i ==> (Var\ j)[u/i] = Var\ j$
⟨*proof*⟩

lemma *lift-lift*:
$i < k + 1 \Longrightarrow lift\ (lift\ t\ i)\ (Suc\ k) = lift\ (lift\ t\ k)\ i$
⟨*proof*⟩

lemma *lift-subst* [*simp*]:
$j < i + 1 \Longrightarrow lift\ (t[s/j])\ i = (lift\ t\ (i + 1))\ [lift\ s\ i\ /\ j]$
⟨*proof*⟩

lemma *lift-subst-lt*:
$i < j + 1 \Longrightarrow lift\ (t[s/j])\ i = (lift\ t\ i)\ [lift\ s\ i\ /\ j + 1]$
⟨*proof*⟩

lemma *subst-lift* [*simp*]:
$(lift\ t\ k)[s/k] = t$
⟨*proof*⟩

lemma *subst-subst*:
$i < j + 1 \Longrightarrow t[lift\ v\ i\ /\ Suc\ j][u[v/j]/i] = t[u/i][v/j]$
⟨*proof*⟩

1.5 Equivalence proof for optimized substitution

lemma *liftn-0* [*simp*]: $liftn\ 0\ t\ k = t$
⟨*proof*⟩

lemma *liftn-lift* [*simp*]: $liftn\ (Suc\ n)\ t\ k = lift\ (liftn\ n\ t\ k)\ k$
⟨*proof*⟩

lemma *substn-subst-n* [*simp*]: $substn\ t\ s\ n = t[liftn\ n\ s\ 0\ /\ n]$
⟨*proof*⟩

theorem *substn-subst-0*: $substn\ t\ s\ 0 = t[s/0]$
⟨*proof*⟩

1.6 Preservation theorems

Not used in Church-Rosser proof, but in Strong Normalization.

theorem *subst-preserves-beta* [*simp*]:
$r \rightarrow_\beta s ==> r[t/i] \rightarrow_\beta s[t/i]$
⟨*proof*⟩

theorem *subst-preserves-beta'*: $r \rightarrow_\beta{}^* s ==> r[t/i] \rightarrow_\beta{}^* s[t/i]$
⟨*proof*⟩

theorem *lift-preserves-beta* [*simp*]:
$r \rightarrow_\beta s ==> lift\ r\ i \rightarrow_\beta lift\ s\ i$
⟨*proof*⟩

theorem *lift-preserves-beta'*: $r \rightarrow_\beta{}^* s ==> lift\ r\ i \rightarrow_\beta{}^* lift\ s\ i$
⟨*proof*⟩

theorem *subst-preserves-beta2* [*simp*]: $r \to_\beta s$ ==> $t[r/i] \to_\beta{}^* t[s/i]$
⟨*proof*⟩

theorem *subst-preserves-beta2'*: $r \to_\beta{}^* s$ ==> $t[r/i] \to_\beta{}^* t[s/i]$
⟨*proof*⟩

end

2 Abstract commutation and confluence notions

theory *Commutation* **imports** *Main* **begin**

2.1 Basic definitions

definition
square :: ['a => 'a => bool, 'a => 'a => bool, 'a => 'a => bool, 'a => 'a => bool] => bool **where**
square R S T U =
($\forall x\ y.\ R\ x\ y$ --> ($\forall z.\ S\ x\ z$ --> ($\exists u.\ T\ y\ u \land U\ z\ u$)))

definition
commute :: ['a => 'a => bool, 'a => 'a => bool] => bool **where**
commute R S = *square R S S R*

definition
diamond :: ('a => 'a => bool) => bool **where**
diamond R = *commute R R*

definition
Church-Rosser :: ('a => 'a => bool) => bool **where**
Church-Rosser R =
($\forall x\ y.$ (*sup R* (R^--1))^** x y --> ($\exists z.$ R^** x z $\land$ R^** y z))

abbreviation
confluent :: ('a => 'a => bool) => bool **where**
confluent R == *diamond* (R^**)

2.2 Basic lemmas

square

lemma *square-sym*: *square R S T U* ==> *square S R U T*
⟨*proof*⟩

lemma *square-subset*:
[| *square R S T U*; $T \le T'$ |] ==> *square R S T' U*
⟨*proof*⟩

lemma *square-reflcl*:
  [| *square R S T* (*R*ˆ==); *S* ≤ *T* |] ==> *square* (*R*ˆ==) *S T* (*R*ˆ==)
⟨*proof*⟩

lemma *square-rtrancl*:
  square R S S T ==> *square* (*R*ˆ∗∗) *S S* (*T*ˆ∗∗)
⟨*proof*⟩

lemma *square-rtrancl-reflcl-commute*:
  square R S (*S*ˆ∗∗) (*R*ˆ==) ==> *commute* (*R*ˆ∗∗) (*S*ˆ∗∗)
⟨*proof*⟩

commute

lemma *commute-sym*: *commute R S* ==> *commute S R*
⟨*proof*⟩

lemma *commute-rtrancl*: *commute R S* ==> *commute* (*R*ˆ∗∗) (*S*ˆ∗∗)
⟨*proof*⟩

lemma *commute-Un*:
  [| *commute R T*; *commute S T* |] ==> *commute* (*sup R S*) *T*
⟨*proof*⟩

diamond, *confluence*, **and** *union*

lemma *diamond-Un*:
  [| *diamond R*; *diamond S*; *commute R S* |] ==> *diamond* (*sup R S*)
⟨*proof*⟩

lemma *diamond-confluent*: *diamond R* ==> *confluent R*
⟨*proof*⟩

lemma *square-reflcl-confluent*:
  square R R (*R*ˆ==) (*R*ˆ==) ==> *confluent R*
⟨*proof*⟩

lemma *confluent-Un*:
  [| *confluent R*; *confluent S*; *commute* (*R*ˆ∗∗) (*S*ˆ∗∗) |] ==> *confluent* (*sup R S*)
⟨*proof*⟩

lemma *diamond-to-confluence*:
  [| *diamond R*; *T* ≤ *R*; *R* ≤ *T*ˆ∗∗ |] ==> *confluent T*
⟨*proof*⟩

2.3 Church-Rosser

lemma *Church-Rosser-confluent*: *Church-Rosser R* = *confluent R*
⟨*proof*⟩

2.4 Newman's lemma

Proof by Stefan Berghofer

theorem *newman*:
  assumes *wf*: $wfP\ (R^{-1-1})$
  and *lc*: $\bigwedge a\ b\ c.\ R\ a\ b \Longrightarrow R\ a\ c \Longrightarrow$
    $\exists\, d.\ R^{**}\ b\ d \wedge R^{**}\ c\ d$
  shows $\bigwedge b\ c.\ R^{**}\ a\ b \Longrightarrow R^{**}\ a\ c \Longrightarrow$
    $\exists\, d.\ R^{**}\ b\ d \wedge R^{**}\ c\ d$
  ⟨*proof*⟩

Alternative version. Partly automated by Tobias Nipkow. Takes 2 minutes (2002).

This is the maximal amount of automation possible using *blast*.

theorem *newman'*:
  assumes *wf*: $wfP\ (R^{-1-1})$
  and *lc*: $\bigwedge a\ b\ c.\ R\ a\ b \Longrightarrow R\ a\ c \Longrightarrow$
    $\exists\, d.\ R^{**}\ b\ d \wedge R^{**}\ c\ d$
  shows $\bigwedge b\ c.\ R^{**}\ a\ b \Longrightarrow R^{**}\ a\ c \Longrightarrow$
    $\exists\, d.\ R^{**}\ b\ d \wedge R^{**}\ c\ d$
  ⟨*proof*⟩

Using the coherent logic prover, the proof of the induction step is completely automatic.

lemma *eq-imp-rtranclp*: $x = y \Longrightarrow r^{**}\ x\ y$
  ⟨*proof*⟩

theorem *newman''*:
  assumes *wf*: $wfP\ (R^{-1-1})$
  and *lc*: $\bigwedge a\ b\ c.\ R\ a\ b \Longrightarrow R\ a\ c \Longrightarrow$
    $\exists\, d.\ R^{**}\ b\ d \wedge R^{**}\ c\ d$
  shows $\bigwedge b\ c.\ R^{**}\ a\ b \Longrightarrow R^{**}\ a\ c \Longrightarrow$
    $\exists\, d.\ R^{**}\ b\ d \wedge R^{**}\ c\ d$
  ⟨*proof*⟩

end

3 Parallel reduction and a complete developments

theory *ParRed* **imports** *Lambda Commutation* **begin**

3.1 Parallel reduction

inductive *par-beta* :: [*dB*, *dB*] => *bool* (**infixl** => *50*)
  where
    var [*simp*, *intro*!]: *Var n* => *Var n*
  | *abs* [*simp*, *intro*!]: *s* => *t* ==> *Abs s* => *Abs t*

| *app* [*simp*, *intro*!]: [| s => s'; t => t' |] ==> $s \circ t$ => $s' \circ t'$
| *beta* [*simp*, *intro*!]: [| s => s'; t => t' |] ==> (*Abs* s) $\circ$ t => $s'[t'/0]$

inductive-cases *par-beta-cases* [*elim*!]:
Var n => *t*
Abs s => *Abs t*
(*Abs s*) $\circ$ *t* => *u*
s $\circ$ *t* => *u*
Abs s => *t*

3.2 Inclusions

beta $\subseteq$ *par-beta* $\subseteq$ *beta*^*

lemma *par-beta-varL* [*simp*]:
(*Var n* => *t*) = (*t* = *Var n*)
⟨*proof*⟩

lemma *par-beta-refl* [*simp*]: *t* => *t*
⟨*proof*⟩

lemma *beta-subset-par-beta*: *beta* <= *par-beta*
⟨*proof*⟩

lemma *par-beta-subset-beta*: *par-beta* <= *beta*^**
⟨*proof*⟩

3.3 Misc properties of *par-beta*

lemma *par-beta-lift* [*simp*]:
t => t' $\Longrightarrow$ *lift* t n => *lift* t' n
⟨*proof*⟩

lemma *par-beta-subst*:
s => s' $\Longrightarrow$ t => t' $\Longrightarrow$ $t[s/n]$ => $t'[s'/n]$
⟨*proof*⟩

3.4 Confluence (directly)

lemma *diamond-par-beta*: *diamond par-beta*
⟨*proof*⟩

3.5 Complete developments

consts
cd :: *dB* => *dB*
recdef *cd measure size*
cd (*Var n*) = *Var n*
cd (*Var n* $\circ$ *t*) = *Var n* $\circ$ *cd t*
cd ((*s1* $\circ$ *s2*) $\circ$ *t*) = *cd* (*s1* $\circ$ *s2*) $\circ$ *cd t*

cd (*Abs u* ° *t*) = (*cd u*)[*cd t*/*0*]
cd (*Abs s*) = *Abs* (*cd s*)

lemma *par-beta-cd*: $s \Rightarrow t \Longrightarrow t \Rightarrow cd\ s$
⟨*proof*⟩

3.6 Confluence (via complete developments)

lemma *diamond-par-beta2*: *diamond par-beta*
⟨*proof*⟩

theorem *beta-confluent*: *confluent beta*
⟨*proof*⟩

end

4 Eta-reduction

theory *Eta* **imports** *ParRed* **begin**

4.1 Definition of eta-reduction and relatives

primrec
free :: *dB* => *nat* => *bool*
where
free (*Var j*) *i* = (*j* = *i*)
| *free* (*s* ° *t*) *i* = (*free s i* ∨ *free t i*)
| *free* (*Abs s*) *i* = *free s* (*i* + *1*)

inductive
eta :: [*dB*, *dB*] => *bool* (**infixl** $\to_\eta$ *50*)
where
eta [*simp*, *intro*]: ¬ *free s 0* ==> *Abs* (*s* ° *Var 0*) $\to_\eta$ *s*[*dummy*/*0*]
| *appL* [*simp*, *intro*]: *s* $\to_\eta$ *t* ==> *s* ° *u* $\to_\eta$ *t* ° *u*
| *appR* [*simp*, *intro*]: *s* $\to_\eta$ *t* ==> *u* ° *s* $\to_\eta$ *u* ° *t*
| *abs* [*simp*, *intro*]: *s* $\to_\eta$ *t* ==> *Abs s* $\to_\eta$ *Abs t*

abbreviation
eta-reds :: [*dB*, *dB*] => *bool* (**infixl** −*e*>> *50*) **where**
s −*e*>> *t* == *eta*ˆ∗∗ *s t*

abbreviation
eta-red0 :: [*dB*, *dB*] => *bool* (**infixl** −*e*>= *50*) **where**
s −*e*>= *t* == *eta*ˆ== *s t*

notation (*xsymbols*)
eta-reds (**infixl** ${\to_\eta}^*$ *50*) **and**
eta-red0 (**infixl** ${\to_\eta}^=$ *50*)

inductive-cases *eta-cases* [*elim*!]:
$Abs\ s \to_\eta z$
$s \circ t \to_\eta u$
$Var\ i \to_\eta t$

4.2 Properties of *eta*, *subst* and *free*

lemma *subst-not-free* [*simp*]: $\neg$ *free s i* $\Longrightarrow s[t/i] = s[u/i]$
⟨*proof*⟩

lemma *free-lift* [*simp*]:
$free\ (lift\ t\ k)\ i = (i < k \wedge free\ t\ i \vee k < i \wedge free\ t\ (i - 1))$
⟨*proof*⟩

lemma *free-subst* [*simp*]:
$free\ (s[t/k])\ i =$
$(free\ s\ k \wedge free\ t\ i \vee free\ s\ (if\ i < k\ then\ i\ else\ i + 1))$
⟨*proof*⟩

lemma *free-eta*: $s \to_\eta t ==> free\ t\ i = free\ s\ i$
⟨*proof*⟩

lemma *not-free-eta*:
$[|\ s \to_\eta t;\ \neg\ free\ s\ i\ |] ==> \neg\ free\ t\ i$
⟨*proof*⟩

lemma *eta-subst* [*simp*]:
$s \to_\eta t ==> s[u/i] \to_\eta t[u/i]$
⟨*proof*⟩

theorem *lift-subst-dummy*: $\neg\ free\ s\ i \Longrightarrow lift\ (s[dummy/i])\ i = s$
⟨*proof*⟩

4.3 Confluence of *eta*

lemma *square-eta*: *square eta eta* (*eta*ˆ==) (*eta*ˆ==)
⟨*proof*⟩

theorem *eta-confluent*: *confluent eta*
⟨*proof*⟩

4.4 Congruence rules for eta^*

lemma *rtrancl-eta-Abs*: $s \to_\eta{}^* s' ==> Abs\ s \to_\eta{}^* Abs\ s'$
⟨*proof*⟩

lemma *rtrancl-eta-AppL*: $s \to_\eta{}^* s' ==> s \circ t \to_\eta{}^* s' \circ t$
⟨*proof*⟩

lemma *rtrancl-eta-AppR*: $t \rightarrow_\eta{}^* t' ==> s \circ t \rightarrow_\eta{}^* s \circ t'$
⟨*proof*⟩

lemma *rtrancl-eta-App*:
$[| s \rightarrow_\eta{}^* s'; t \rightarrow_\eta{}^* t' |] ==> s \circ t \rightarrow_\eta{}^* s' \circ t'$
⟨*proof*⟩

4.5 Commutation of *beta* and *eta*

lemma *free-beta*:
$s \rightarrow_\beta t ==> \mathit{free}\ t\ i \Longrightarrow \mathit{free}\ s\ i$
⟨*proof*⟩

lemma *beta-subst* [*intro*]: $s \rightarrow_\beta t ==> s[u/i] \rightarrow_\beta t[u/i]$
⟨*proof*⟩

lemma *subst-Var-Suc* [*simp*]: $t[\mathit{Var}\ i/i] = t[\mathit{Var}(i)/i + 1]$
⟨*proof*⟩

lemma *eta-lift* [*simp*]: $s \rightarrow_\eta t ==> \mathit{lift}\ s\ i \rightarrow_\eta \mathit{lift}\ t\ i$
⟨*proof*⟩

lemma *rtrancl-eta-subst*: $s \rightarrow_\eta t \Longrightarrow u[s/i] \rightarrow_\eta{}^* u[t/i]$
⟨*proof*⟩

lemma *rtrancl-eta-subst′*:
fixes $s\ t :: \mathit{dB}$
assumes *eta*: $s \rightarrow_\eta{}^* t$
shows $s[u/i] \rightarrow_\eta{}^* t[u/i]$ ⟨*proof*⟩

lemma *rtrancl-eta-subst″*:
fixes $s\ t :: \mathit{dB}$
assumes *eta*: $s \rightarrow_\eta{}^* t$
shows $u[s/i] \rightarrow_\eta{}^* u[t/i]$ ⟨*proof*⟩

lemma *square-beta-eta*: *square beta eta* (*eta*^**) (*beta*^==)
⟨*proof*⟩

lemma *confluent-beta-eta*: *confluent* (*sup beta eta*)
⟨*proof*⟩

4.6 Implicit definition of *eta*

$\mathit{Abs}\ (\mathit{lift}\ s\ 0 \circ \mathit{Var}\ 0) \rightarrow_\eta s$

lemma *not-free-iff-lifted*:
$(\neg\ \mathit{free}\ s\ i) = (\exists\, t.\ s = \mathit{lift}\ t\ i)$
⟨*proof*⟩

theorem *explicit-is-implicit*:

$(\forall\, s\ u.\ (\neg\ free\ s\ 0) \dashrightarrow R\ (Abs\ (s \circ\ Var\ 0))\ (s[u/0])) =$
  $(\forall\, s.\ R\ (Abs\ (lift\ s\ 0 \circ\ Var\ 0))\ s)$
  ⟨*proof*⟩

4.7 Eta-postponement theorem

Based on a paper proof due to Andreas Abel. Unlike the proof by Masako Takahashi [4], it does not use parallel eta reduction, which only seems to complicate matters unnecessarily.

theorem *eta-case*:
  fixes $s :: dB$
  assumes *free*: $\neg\ free\ s\ 0$
  and *s*: $s[dummy/0] => u$
  shows $\exists\, t'.\ Abs\ (s \circ\ Var\ 0) => t' \wedge t' \rightarrow_{\eta}{}^{*}\ u$
⟨*proof*⟩

theorem *eta-par-beta*:
  assumes *st*: $s \rightarrow_{\eta}\ t$
  and *tu*: $t => u$
  shows $\exists\, t'.\ s => t' \wedge t' \rightarrow_{\eta}{}^{*}\ u$ ⟨*proof*⟩

theorem *eta-postponement'*:
  assumes *eta*: $s \rightarrow_{\eta}{}^{*}\ t$ **and** *beta*: $t => u$
  shows $\exists\, t'.\ s => t' \wedge t' \rightarrow_{\eta}{}^{*}\ u$ ⟨*proof*⟩

theorem *eta-postponement*:
  assumes $(sup\ beta\ eta)^{**}\ s\ t$
  shows $(eta^{**}\ OO\ beta^{**})\ s\ t$ ⟨*proof*⟩

end

5 Application of a term to a list of terms

theory *ListApplication* **imports** *Lambda* **begin**

abbreviation
  $list\text{-}application :: dB => dB\ list => dB$ (**infixl** $\circ\circ$ *150*) **where**
  $t \circ\circ ts == foldl\ (op\ \circ)\ t\ ts$

lemma *apps-eq-tail-conv* [*iff*]: $(r \circ\circ ts = s \circ\circ ts) = (r = s)$
  ⟨*proof*⟩

lemma *Var-eq-apps-conv* [*iff*]: $(Var\ m = s \circ\circ ss) = (Var\ m = s \wedge ss = [])$
  ⟨*proof*⟩

lemma *Var-apps-eq-Var-apps-conv* [*iff*]:
    $(Var\ m \circ\circ rs = Var\ n \circ\circ ss) = (m = n \wedge rs = ss)$

$\langle proof \rangle$

lemma *App-eq-foldl-conv*:
$(r \circ s = t \circ\circ ts) =$
$(\textit{if } ts = [] \textit{ then } r \circ s = t$
$\textit{else } (\exists ss.\ ts = ss \mathbin{@} [s] \wedge r = t \circ\circ ss))$
$\langle proof \rangle$

lemma *Abs-eq-apps-conv* [*iff*]:
$(Abs\ r = s \circ\circ ss) = (Abs\ r = s \wedge ss = [])$
$\langle proof \rangle$

lemma *apps-eq-Abs-conv* [*iff*]: $(s \circ\circ ss = Abs\ r) = (s = Abs\ r \wedge ss = [])$
$\langle proof \rangle$

lemma *Abs-apps-eq-Abs-apps-conv* [*iff*]:
$(Abs\ r \circ\circ rs = Abs\ s \circ\circ ss) = (r = s \wedge rs = ss)$
$\langle proof \rangle$

lemma *Abs-App-neq-Var-apps* [*iff*]:
$Abs\ s \circ t \neq Var\ n \circ\circ ss$
$\langle proof \rangle$

lemma *Var-apps-neq-Abs-apps* [*iff*]:
$Var\ n \circ\circ ts \neq Abs\ r \circ\circ ss$
$\langle proof \rangle$

lemma *ex-head-tail*:
$\exists ts\ h.\ t = h \circ\circ ts \wedge ((\exists n.\ h = Var\ n) \vee (\exists u.\ h = Abs\ u))$
$\langle proof \rangle$

lemma *size-apps* [*simp*]:
$size\ (r \circ\circ rs) = size\ r + foldl\ (op\ +)\ 0\ (map\ size\ rs) + length\ rs$
$\langle proof \rangle$

lemma *lem0*: $[|\ (0::nat) < k;\ m <= n\ |] ==> m < n + k$
$\langle proof \rangle$

lemma *lift-map* [*simp*]:
$lift\ (t \circ\circ ts)\ i = lift\ t\ i \circ\circ map\ (\lambda t.\ lift\ t\ i)\ ts$
$\langle proof \rangle$

lemma *subst-map* [*simp*]:
$subst\ (t \circ\circ ts)\ u\ i = subst\ t\ u\ i \circ\circ map\ (\lambda t.\ subst\ t\ u\ i)\ ts$
$\langle proof \rangle$

lemma *app-last*: $(t \circ\circ ts) \circ u = t \circ\circ (ts \mathbin{@} [u])$
$\langle proof \rangle$

A customized induction schema for $^{\circ\circ}$.

lemma *lem*:
  assumes !!*n ts*. $\forall\, t \in$ *set ts*. *P t* ==> *P* (*Var n* $^{\circ\circ}$ *ts*)
    and !!*u ts*. [| *P u*; $\forall\, t \in$ *set ts*. *P t* |] ==> *P* (*Abs u* $^{\circ\circ}$ *ts*)
  shows *size t* = *n* $\Longrightarrow$ *P t*
  ⟨*proof*⟩

theorem *Apps-dB-induct*:
  assumes !!*n ts*. $\forall\, t \in$ *set ts*. *P t* ==> *P* (*Var n* $^{\circ\circ}$ *ts*)
    and !!*u ts*. [| *P u*; $\forall\, t \in$ *set ts*. *P t* |] ==> *P* (*Abs u* $^{\circ\circ}$ *ts*)
  shows *P t*
  ⟨*proof*⟩

end

6 Simply-typed lambda terms

theory *Type* **imports** *ListApplication* **begin**

6.1 Environments

definition
  shift :: (*nat* ⇒ *'a*) ⇒ *nat* ⇒ *'a* ⇒ *nat* ⇒ *'a* (-<-:-> [*90*, *0*, *0*] *91*) **where**
  e<*i*:*a*> = (λ*j*. *if j* < *i then e j else if j* = *i then a else e* (*j* − *1*))

notation (*xsymbols*)
  shift (-⟨-:-⟩ [*90*, *0*, *0*] *91*)

notation (*HTML* **output**)
  shift (-⟨-:-⟩ [*90*, *0*, *0*] *91*)

lemma *shift-eq* [*simp*]: *i* = *j* $\Longrightarrow$ (*e*⟨*i*:*T*⟩) *j* = *T*
  ⟨*proof*⟩

lemma *shift-gt* [*simp*]: *j* < *i* $\Longrightarrow$ (*e*⟨*i*:*T*⟩) *j* = *e j*
  ⟨*proof*⟩

lemma *shift-lt* [*simp*]: *i* < *j* $\Longrightarrow$ (*e*⟨*i*:*T*⟩) *j* = *e* (*j* − *1*)
  ⟨*proof*⟩

lemma *shift-commute* [*simp*]: *e*⟨*i*:*U*⟩⟨*0*:*T*⟩ = *e*⟨*0*:*T*⟩⟨*Suc i*:*U*⟩
  ⟨*proof*⟩

6.2 Types and typing rules

datatype *type* =
    Atom nat
  | *Fun type type*   (**infixr** ⇒ *200*)

inductive *typing* :: (*nat* ⇒ *type*) ⇒ *dB* ⇒ *type* ⇒ *bool* (- ⊢ - : - [*50*, *50*, *50*] *50*)
where
Var [*intro*!]: *env x* = *T* ⟹ *env* ⊢ *Var x* : *T*
| *Abs* [*intro*!]: *env*⟨*0*:*T*⟩ ⊢ *t* : *U* ⟹ *env* ⊢ *Abs t* : (*T* ⇒ *U*)
| *App* [*intro*!]: *env* ⊢ *s* : *T* ⇒ *U* ⟹ *env* ⊢ *t* : *T* ⟹ *env* ⊢ (*s* ° *t*) : *U*

inductive-cases *typing-elims* [*elim*!]:
e ⊢ *Var i* : *T*
e ⊢ *t* ° *u* : *T*
e ⊢ *Abs t* : *T*

primrec
typings :: (*nat* ⇒ *type*) ⇒ *dB list* ⇒ *type list* ⇒ *bool*
where
typings e [] *Ts* = (*Ts* = [])
| *typings e* (*t* # *ts*) *Ts* =
(*case Ts of*
[] ⇒ *False*
| *T* # *Ts* ⇒ *e* ⊢ *t* : *T* ∧ *typings e ts Ts*)

abbreviation
typings-rel :: (*nat* ⇒ *type*) ⇒ *dB list* ⇒ *type list* ⇒ *bool*
(- ||− - : - [*50*, *50*, *50*] *50*) **where**
env ||− *ts* : *Ts* == *typings env ts Ts*

notation (*latex*)
typings-rel (- ⊩ - : - [*50*, *50*, *50*] *50*)

abbreviation
funs :: *type list* ⇒ *type* ⇒ *type* (**infixr** =>> *200*) **where**
Ts =>> *T* == *foldr Fun Ts T*

notation (*latex*)
funs (**infixr** ⇛ *200*)

6.3 Some examples

lemma *e* ⊢ *Abs* (*Abs* (*Abs* (*Var 1* ° (*Var 2* ° *Var 1* ° *Var 0*)))) : *?T*
⟨*proof*⟩

lemma *e* ⊢ *Abs* (*Abs* (*Abs* (*Var 2* ° *Var 0* ° (*Var 1* ° *Var 0*)))) : *?T*
⟨*proof*⟩

6.4 Lists of types

lemma *lists-typings*:
e ⊩ *ts* : *Ts* ⟹ *listsp* (λ*t*. ∃ *T*. *e* ⊢ *t* : *T*) *ts*
⟨*proof*⟩

lemma *types-snoc*: $e \Vdash ts : Ts \Longrightarrow e \vdash t : T \Longrightarrow e \Vdash ts @ [t] : Ts @ [T]$
⟨*proof*⟩

lemma *types-snoc-eq*: $e \Vdash ts @ [t] : Ts @ [T] =$
$(e \Vdash ts : Ts \wedge e \vdash t : T)$
⟨*proof*⟩

lemma *rev-exhaust2* [*extraction-expand*]:
obtains (Nil) $xs = []$ | $(snoc)$ ys y **where** $xs = ys @ [y]$
— Cannot use *rev-exhaust* from the *List* theory, since it is not constructive
⟨*proof*⟩

lemma *types-snocE*: $e \Vdash ts @ [t] : Ts \Longrightarrow$
$(\bigwedge Us\ U.\ Ts = Us @ [U] \Longrightarrow e \Vdash ts : Us \Longrightarrow e \vdash t : U \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

6.5 n-ary function types

lemma *list-app-typeD*:
$e \vdash t \circ\circ ts : T \Longrightarrow \exists Ts.\ e \vdash t : Ts \Rrightarrow T \wedge e \Vdash ts : Ts$
⟨*proof*⟩

lemma *list-app-typeE*:
$e \vdash t \circ\circ ts : T \Longrightarrow (\bigwedge Ts.\ e \vdash t : Ts \Rrightarrow T \Longrightarrow e \Vdash ts : Ts \Longrightarrow C) \Longrightarrow C$
⟨*proof*⟩

lemma *list-app-typeI*:
$e \vdash t : Ts \Rrightarrow T \Longrightarrow e \Vdash ts : Ts \Longrightarrow e \vdash t \circ\circ ts : T$
⟨*proof*⟩

For the specific case where the head of the term is a variable, the following theorems allow to infer the types of the arguments without analyzing the typing derivation. This is crucial for program extraction.

theorem *var-app-type-eq*:
$e \vdash Var\ i \circ\circ ts : T \Longrightarrow e \vdash Var\ i \circ\circ ts : U \Longrightarrow T = U$
⟨*proof*⟩

lemma *var-app-types*: $e \vdash Var\ i \circ\circ ts \circ\circ us : T \Longrightarrow e \Vdash ts : Ts \Longrightarrow$
$e \vdash Var\ i \circ\circ ts : U \Longrightarrow \exists Us.\ U = Us \Rrightarrow T \wedge e \Vdash us : Us$
⟨*proof*⟩

lemma *var-app-typesE*: $e \vdash Var\ i \circ\circ ts : T \Longrightarrow$
$(\bigwedge Ts.\ e \vdash Var\ i : Ts \Rrightarrow T \Longrightarrow e \Vdash ts : Ts \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *abs-typeE*: $e \vdash Abs\ t : T \Longrightarrow (\bigwedge U\ V.\ e\langle 0{:}U\rangle \vdash t : V \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

6.6 Lifting preserves well-typedness

lemma *lift-type* [*intro*!]: $e \vdash t : T \Longrightarrow e\langle i{:}U\rangle \vdash \mathit{lift}\ t\ i : T$
$\langle \mathit{proof} \rangle$

lemma *lift-types*:
$e \Vdash ts : Ts \Longrightarrow e\langle i{:}U\rangle \Vdash (\mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ i)\ ts) : Ts$
$\langle \mathit{proof} \rangle$

6.7 Substitution lemmas

lemma *subst-lemma*:
$e \vdash t : T \Longrightarrow e' \vdash u : U \Longrightarrow e = e'\langle i{:}U\rangle \Longrightarrow e' \vdash t[u/i] : T$
$\langle \mathit{proof} \rangle$

lemma *substs-lemma*:
$e \vdash u : T \Longrightarrow e\langle i{:}T\rangle \Vdash ts : Ts \Longrightarrow$
$e \Vdash (\mathit{map}\ (\lambda t.\ t[u/i])\ ts) : Ts$
$\langle \mathit{proof} \rangle$

6.8 Subject reduction

lemma *subject-reduction*: $e \vdash t : T \Longrightarrow t \rightarrow_\beta t' \Longrightarrow e \vdash t' : T$
$\langle \mathit{proof} \rangle$

theorem *subject-reduction'*: $t \rightarrow_\beta{}^* t' \Longrightarrow e \vdash t : T \Longrightarrow e \vdash t' : T$
$\langle \mathit{proof} \rangle$

6.9 Alternative induction rule for types

lemma *type-induct* [*induct type*]:
assumes
$(\bigwedge T.\ (\bigwedge T1\ T2.\ T = T1 \Rightarrow T2 \Longrightarrow P\ T1) \Longrightarrow$
$(\bigwedge T1\ T2.\ T = T1 \Rightarrow T2 \Longrightarrow P\ T2) \Longrightarrow P\ T)$
shows $P\ T$
$\langle \mathit{proof} \rangle$

end

7 Lifting an order to lists of elements

theory *ListOrder* **imports** *Main* **begin**

Lifting an order to lists of elements, relating exactly one element.

definition
$\mathit{step1} :: ('a \Rightarrow 'a \Rightarrow \mathit{bool}) \Rightarrow 'a\ \mathit{list} \Rightarrow 'a\ \mathit{list} \Rightarrow \mathit{bool}$ **where**
$\mathit{step1}\ r =$
$(\lambda ys\ xs.\ \exists us\ z\ z'\ vs.\ xs = us\ @\ z\ \#\ vs \wedge r\ z'\ z \wedge ys =$

$us\ @\ z'\ \#\ vs)$

lemma *step1-converse* [*simp*]: $step1\ (r\hat{}{-}{-}1) = (step1\ r)\hat{}{-}{-}1$
⟨*proof*⟩

lemma *in-step1-converse* [*iff*]: $(step1\ (r\hat{}{-}{-}1)\ x\ y) = ((step1\ r)\hat{}{-}{-}1\ x\ y)$
⟨*proof*⟩

lemma *not-Nil-step1* [*iff*]: $\neg\ step1\ r\ []\ xs$
⟨*proof*⟩

lemma *not-step1-Nil* [*iff*]: $\neg\ step1\ r\ xs\ []$
⟨*proof*⟩

lemma *Cons-step1-Cons* [*iff*]:
$(step1\ r\ (y\ \#\ ys)\ (x\ \#\ xs)) =$
$(r\ y\ x \wedge xs = ys \vee x = y \wedge step1\ r\ ys\ xs)$
⟨*proof*⟩

lemma *append-step1I*:
$step1\ r\ ys\ xs \wedge vs = us \vee ys = xs \wedge step1\ r\ vs\ us$
$==>\ step1\ r\ (ys\ @\ vs)\ (xs\ @\ us)$
⟨*proof*⟩

lemma *Cons-step1E* [*elim*!]:
assumes $step1\ r\ ys\ (x\ \#\ xs)$
and $!!y.\ ys = y\ \#\ xs \Longrightarrow r\ y\ x \Longrightarrow R$
and $!!zs.\ ys = x\ \#\ zs \Longrightarrow step1\ r\ zs\ xs \Longrightarrow R$
shows R
⟨*proof*⟩

lemma *Snoc-step1-SnocD*:
$step1\ r\ (ys\ @\ [y])\ (xs\ @\ [x])$
$==>\ (step1\ r\ ys\ xs \wedge y = x \vee ys = xs \wedge r\ y\ x)$
⟨*proof*⟩

lemma *Cons-acc-step1I* [*intro*!]:
$accp\ r\ x\ ==>\ accp\ (step1\ r)\ xs \Longrightarrow accp\ (step1\ r)\ (x\ \#\ xs)$
⟨*proof*⟩

lemma *lists-accD*: $listsp\ (accp\ r)\ xs\ ==>\ accp\ (step1\ r)\ xs$
⟨*proof*⟩

lemma *ex-step1I*:
$[|\ x \in set\ xs;\ r\ y\ x\ |]$
$==>\ \exists ys.\ step1\ r\ ys\ xs \wedge y \in set\ ys$
⟨*proof*⟩

lemma *lists-accI*: *accp* (*step1 r*) *xs* ==> *listsp* (*accp r*) *xs*
⟨*proof*⟩

end

8 Lifting beta-reduction to lists

theory *ListBeta* **imports** *ListApplication ListOrder* **begin**

Lifting beta-reduction to lists of terms, reducing exactly one element.

abbreviation
list-beta :: *dB list* => *dB list* => *bool* (**infixl** => *50*) **where**
rs => *ss* == *step1 beta rs ss*

lemma *head-Var-reduction*:
$Var\ n\ {}^{\circ\circ}\ rs \to_\beta v \Longrightarrow \exists ss.\ rs => ss \wedge v = Var\ n\ {}^{\circ\circ}\ ss$
⟨*proof*⟩

lemma *apps-betasE* [*elim*!]:
assumes *major*: $r\ {}^{\circ\circ}\ rs \to_\beta s$
and *cases*: $!!r'.\ [|\ r \to_\beta r';\ s = r'\ {}^{\circ\circ}\ rs\ |] ==> R$
$!!rs'.\ [|\ rs => rs';\ s = r\ {}^{\circ\circ}\ rs'\ |] ==> R$
$!!t\ u\ us.\ [|\ r = Abs\ t;\ rs = u\ \#\ us;\ s = t[u/0]\ {}^{\circ\circ}\ us\ |] ==> R$
shows *R*
⟨*proof*⟩

lemma *apps-preserves-beta* [*simp*]:
$r \to_\beta s ==> r\ {}^{\circ\circ}\ ss \to_\beta s\ {}^{\circ\circ}\ ss$
⟨*proof*⟩

lemma *apps-preserves-beta2* [*simp*]:
$r\ {-}{>}{>}\ s ==> r\ {}^{\circ\circ}\ ss\ {-}{>}{>}\ s\ {}^{\circ\circ}\ ss$
⟨*proof*⟩

lemma *apps-preserves-betas* [*simp*]:
$rs => ss \Longrightarrow r\ {}^{\circ\circ}\ rs \to_\beta r\ {}^{\circ\circ}\ ss$
⟨*proof*⟩

end

9 Inductive characterization of terminating lambda terms

theory *InductTermi* **imports** *ListBeta* **begin**

9.1 Terminating lambda terms

inductive *IT* :: *dB* => *bool*
  where
    Var [*intro*]: *listsp IT rs* ==> *IT* (*Var n* $^{\circ\circ}$ *rs*)
  | *Lambda* [*intro*]: *IT r* ==> *IT* (*Abs r*)
  | *Beta* [*intro*]: *IT* ((*r*[*s*/*0*]) $^{\circ\circ}$ *ss*) ==> *IT s* ==> *IT* ((*Abs r* $^{\circ}$ *s*) $^{\circ\circ}$ *ss*)

9.2 Every term in *IT* terminates

lemma *double-induction-lemma* [*rule-format*]:
  termip beta s ==> $\forall$ *t*. *termip beta t* −−>
    ($\forall$ *r ss*. *t* = *r*[*s*/*0*] $^{\circ\circ}$ *ss* −−> *termip beta* (*Abs r* $^{\circ}$ *s* $^{\circ\circ}$ *ss*))
  ⟨*proof*⟩

lemma *IT-implies-termi*: *IT t* ==> *termip beta t*
  ⟨*proof*⟩

9.3 Every terminating term is in *IT*

declare *Var-apps-neq-Abs-apps* [*symmetric*, *simp*]

lemma [*simp*, *THEN not-sym*, *simp*]: *Var n* $^{\circ\circ}$ *ss* $\neq$ *Abs r* $^{\circ}$ *s* $^{\circ\circ}$ *ts*
  ⟨*proof*⟩

lemma [*simp*]:
  (*Abs r* $^{\circ}$ *s* $^{\circ\circ}$ *ss* = *Abs r′* $^{\circ}$ *s′* $^{\circ\circ}$ *ss′*) = (*r* = *r′* $\wedge$ *s* = *s′* $\wedge$ *ss* = *ss′*)
  ⟨*proof*⟩

inductive-cases [*elim*!]:
  IT (*Var n* $^{\circ\circ}$ *ss*)
  IT (*Abs t*)
  IT (*Abs r* $^{\circ}$ *s* $^{\circ\circ}$ *ts*)

theorem *termi-implies-IT*: *termip beta r* ==> *IT r*
  ⟨*proof*⟩

end

10 Strong normalization for simply-typed lambda calculus

theory *StrongNorm* **imports** *Type InductTermi* **begin**

Formalization by Stefan Berghofer. Partly based on a paper proof by Felix Joachimski and Ralph Matthes [1].

10.1 Properties of IT

lemma *lift-IT* [*intro*!]: $IT\ t \Longrightarrow IT\ (lift\ t\ i)$
⟨*proof*⟩

lemma *lifts-IT*: $listsp\ IT\ ts \Longrightarrow listsp\ IT\ (map\ (\lambda t.\ lift\ t\ 0)\ ts)$
⟨*proof*⟩

lemma *subst-Var-IT*: $IT\ r \Longrightarrow IT\ (r[Var\ i/j])$
⟨*proof*⟩

lemma *Var-IT*: $IT\ (Var\ n)$
⟨*proof*⟩

lemma *app-Var-IT*: $IT\ t \Longrightarrow IT\ (t\ {}^\circ\ Var\ i)$
⟨*proof*⟩

10.2 Well-typed substitution preserves termination

lemma *subst-type-IT*:
$\bigwedge t\ e\ T\ u\ i.\ IT\ t \Longrightarrow e\langle i{:}U\rangle \vdash t : T \Longrightarrow$
$IT\ u \Longrightarrow e \vdash u : U \Longrightarrow IT\ (t[u/i])$
(**is** $PROP\ ?P\ U$ **is** $\bigwedge t\ e\ T\ u\ i.\ \text{-} \Longrightarrow PROP\ ?Q\ t\ e\ T\ u\ i\ U$)
⟨*proof*⟩

10.3 Well-typed terms are strongly normalizing

lemma *type-implies-IT*:
assumes $e \vdash t : T$
shows $IT\ t$
⟨*proof*⟩

theorem *type-implies-termi*: $e \vdash t : T \Longrightarrow termip\ beta\ t$
⟨*proof*⟩

end

11 Inductive characterization of lambda terms in normal form

theory *NormalForm*
imports *ListBeta*
begin

11.1 Terms in normal form

definition
listall :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow bool$ **where**

$listall\ P\ xs \equiv (\forall\, i.\ i < length\ xs \longrightarrow P\ (xs\ !\ i))$

declare *listall-def* [*extraction-expand*]

theorem *listall-nil*: $listall\ P\ []$
⟨*proof*⟩

theorem *listall-nil-eq* [*simp*]: $listall\ P\ [] = True$
⟨*proof*⟩

theorem *listall-cons*: $P\ x \Longrightarrow listall\ P\ xs \Longrightarrow listall\ P\ (x\ \#\ xs)$
⟨*proof*⟩

theorem *listall-cons-eq* [*simp*]: $listall\ P\ (x\ \#\ xs) = (P\ x \wedge listall\ P\ xs)$
⟨*proof*⟩

lemma *listall-conj1*: $listall\ (\lambda x.\ P\ x \wedge Q\ x)\ xs \Longrightarrow listall\ P\ xs$
⟨*proof*⟩

lemma *listall-conj2*: $listall\ (\lambda x.\ P\ x \wedge Q\ x)\ xs \Longrightarrow listall\ Q\ xs$
⟨*proof*⟩

lemma *listall-app*: $listall\ P\ (xs\ @\ ys) = (listall\ P\ xs \wedge listall\ P\ ys)$
⟨*proof*⟩

lemma *listall-snoc* [*simp*]: $listall\ P\ (xs\ @\ [x]) = (listall\ P\ xs \wedge P\ x)$
⟨*proof*⟩

lemma *listall-cong* [*cong*, *extraction-expand*]:
$xs = ys \Longrightarrow listall\ P\ xs = listall\ P\ ys$
— Currently needed for strange technical reasons
⟨*proof*⟩

listsp is equivalent to *listall*, but cannot be used for program extraction.

lemma *listall-listsp-eq*: $listall\ P\ xs = listsp\ P\ xs$
⟨*proof*⟩

inductive $NF :: dB \Rightarrow bool$
where
App: $listall\ NF\ ts \Longrightarrow NF\ (Var\ x\ {}^{\circ\circ}\ ts)$
| *Abs*: $NF\ t \Longrightarrow NF\ (Abs\ t)$
monos *listall-def*

lemma *nat-eq-dec*: $\bigwedge n{::}nat.\ m = n \vee m \neq n$
⟨*proof*⟩

lemma *nat-le-dec*: $\bigwedge n{::}nat.\ m < n \vee \neg\ (m < n)$
⟨*proof*⟩

lemma *App-NF-D*: **assumes** *NF*: $NF\ (Var\ n\ {}^{\circ\circ}\ ts)$
shows *listall NF ts* ⟨*proof*⟩

11.2 Properties of *NF*

lemma *Var-NF*: $NF\ (Var\ n)$
⟨*proof*⟩

lemma *Abs-NF*:
assumes *NF*: $NF\ (Abs\ t\ {}^{\circ\circ}\ ts)$
shows $ts = []$ ⟨*proof*⟩

lemma *subst-terms-NF*: $listall\ NF\ ts \Longrightarrow$
$listall\ (\lambda t.\ \forall\, i\ j.\ NF\ (t[Var\ i/j]))\ ts \Longrightarrow$
$listall\ NF\ (map\ (\lambda t.\ t[Var\ i/j])\ ts)$
⟨*proof*⟩

lemma *subst-Var-NF*: $NF\ t \Longrightarrow NF\ (t[Var\ i/j])$
⟨*proof*⟩

lemma *app-Var-NF*: $NF\ t \Longrightarrow \exists\, t'.\ t\ {}^{\circ}\ Var\ i \rightarrow_{\beta}{}^{*}\ t' \wedge NF\ t'$
⟨*proof*⟩

lemma *lift-terms-NF*: $listall\ NF\ ts \Longrightarrow$
$listall\ (\lambda t.\ \forall\, i.\ NF\ (lift\ t\ i))\ ts \Longrightarrow$
$listall\ NF\ (map\ (\lambda t.\ lift\ t\ i)\ ts)$
⟨*proof*⟩

lemma *lift-NF*: $NF\ t \Longrightarrow NF\ (lift\ t\ i)$
⟨*proof*⟩

NF characterizes exactly the terms that are in normal form.

lemma *NF-eq*: $NF\ t = (\forall\, t'.\ \neg\ t \rightarrow_{\beta}\ t')$
⟨*proof*⟩

end

12 Standardization

theory *Standardization*
imports *NormalForm*
begin

Based on lecture notes by Ralph Matthes [3], original proof idea due to Ralph Loader [2].

12.1 Standard reduction relation

declare *listrel-mono* [*mono-set*]

inductive
  sred :: *dB* ⇒ *dB* ⇒ *bool* (**infixl** $\to_s$ *50*)
  and *sredlist* :: *dB list* ⇒ *dB list* ⇒ *bool* (**infixl** $[\to_s]$ *50*)
where
  $s\ [\to_s]\ t \equiv listrelp\ op \to_s s\ t$
| *Var*: $rs\ [\to_s]\ rs' \Longrightarrow Var\ x\ ^{\circ\circ}\ rs \to_s Var\ x\ ^{\circ\circ}\ rs'$
| *Abs*: $r \to_s r' \Longrightarrow ss\ [\to_s]\ ss' \Longrightarrow Abs\ r\ ^{\circ\circ}\ ss \to_s Abs\ r'\ ^{\circ\circ}\ ss'$
| *Beta*: $r[s/0]\ ^{\circ\circ}\ ss \to_s t \Longrightarrow Abs\ r\ ^{\circ}\ s\ ^{\circ\circ}\ ss \to_s t$

lemma *refl-listrelp*: $\forall x \in set\ xs.\ R\ x\ x \Longrightarrow listrelp\ R\ xs\ xs$
  ⟨*proof*⟩

lemma *refl-sred*: $t \to_s t$
  ⟨*proof*⟩

lemma *refl-sreds*: $ts\ [\to_s]\ ts$
  ⟨*proof*⟩

lemma *listrelp-conj1*: $listrelp\ (\lambda x\ y.\ R\ x\ y \wedge S\ x\ y)\ x\ y \Longrightarrow listrelp\ R\ x\ y$
  ⟨*proof*⟩

lemma *listrelp-conj2*: $listrelp\ (\lambda x\ y.\ R\ x\ y \wedge S\ x\ y)\ x\ y \Longrightarrow listrelp\ S\ x\ y$
  ⟨*proof*⟩

lemma *listrelp-app*:
  assumes *xsys*: $listrelp\ R\ xs\ ys$
  shows $listrelp\ R\ xs'\ ys' \Longrightarrow listrelp\ R\ (xs\ @\ xs')\ (ys\ @\ ys')$ ⟨*proof*⟩

lemma *lemma1*:
  assumes *r*: $r \to_s r'$ **and** *s*: $s \to_s s'$
  shows $r\ ^{\circ}\ s \to_s r'\ ^{\circ}\ s'$ ⟨*proof*⟩

lemma *lemma1'*:
  assumes *ts*: $ts\ [\to_s]\ ts'$
  shows $r \to_s r' \Longrightarrow r\ ^{\circ\circ}\ ts \to_s r'\ ^{\circ\circ}\ ts'$ ⟨*proof*⟩

lemma *lemma2-1*:
  assumes *beta*: $t \to_\beta u$
  shows $t \to_s u$ ⟨*proof*⟩

lemma *listrelp-betas*:
  assumes *ts*: $listrelp\ op \to_\beta{}^*\ ts\ ts'$
  shows $\bigwedge t\ t'.\ t \to_\beta{}^*\ t' \Longrightarrow t\ ^{\circ\circ}\ ts \to_\beta{}^*\ t'\ ^{\circ\circ}\ ts'$ ⟨*proof*⟩

lemma *lemma2-2*:
  assumes *t*: $t \to_s u$

shows $t \rightarrow_\beta{}^* u$ ⟨*proof*⟩

lemma *sred-lift*:
assumes *s*: $s \rightarrow_s t$
shows $\mathit{lift}\ s\ i \rightarrow_s \mathit{lift}\ t\ i$ ⟨*proof*⟩

lemma *lemma3*:
assumes *r*: $r \rightarrow_s r'$
shows $s \rightarrow_s s' \Longrightarrow r[s/x] \rightarrow_s r'[s'/x]$ ⟨*proof*⟩

lemma *lemma4-aux*:
assumes *rs*: $\mathit{listrelp}\ (\lambda t\ u.\ t \rightarrow_s u \wedge (\forall r.\ u \rightarrow_\beta r \longrightarrow t \rightarrow_s r))\ \mathit{rs}\ \mathit{rs}'$
shows $\mathit{rs}' \Rightarrow \mathit{ss} \Longrightarrow \mathit{rs}\ [\rightarrow_s]\ \mathit{ss}$ ⟨*proof*⟩

lemma *lemma4*:
assumes *r*: $r \rightarrow_s r'$
shows $r' \rightarrow_\beta r'' \Longrightarrow r \rightarrow_s r''$ ⟨*proof*⟩

lemma *rtrancl-beta-sred*:
assumes *r*: $r \rightarrow_\beta{}^* r'$
shows $r \rightarrow_s r'$ ⟨*proof*⟩

12.2 Leftmost reduction and weakly normalizing terms

inductive
lred :: $dB \Rightarrow dB \Rightarrow bool$ (**infixl** $\rightarrow_l$ *50*)
and *lredlist* :: $dB\ list \Rightarrow dB\ list \Rightarrow bool$ (**infixl** $[\rightarrow_l]$ *50*)
where
$s\ [\rightarrow_l]\ t \equiv \mathit{listrelp}\ \mathit{op} \rightarrow_l s\ t$
| *Var*: $\mathit{rs}\ [\rightarrow_l]\ \mathit{rs}' \Longrightarrow \mathit{Var}\ x\ {}^{\circ\circ}\ \mathit{rs} \rightarrow_l \mathit{Var}\ x\ {}^{\circ\circ}\ \mathit{rs}'$
| *Abs*: $r \rightarrow_l r' \Longrightarrow \mathit{Abs}\ r \rightarrow_l \mathit{Abs}\ r'$
| *Beta*: $r[s/0]\ {}^{\circ\circ}\ \mathit{ss} \rightarrow_l t \Longrightarrow \mathit{Abs}\ r\ {}^{\circ}\ s\ {}^{\circ\circ}\ \mathit{ss} \rightarrow_l t$

lemma *lred-imp-sred*:
assumes *lred*: $s \rightarrow_l t$
shows $s \rightarrow_s t$ ⟨*proof*⟩

inductive *WN* :: $dB \Rightarrow bool$
where
Var: $\mathit{listsp}\ \mathit{WN}\ \mathit{rs} \Longrightarrow \mathit{WN}\ (\mathit{Var}\ n\ {}^{\circ\circ}\ \mathit{rs})$
| *Lambda*: $\mathit{WN}\ r \Longrightarrow \mathit{WN}\ (\mathit{Abs}\ r)$
| *Beta*: $\mathit{WN}\ ((r[s/0])\ {}^{\circ\circ}\ \mathit{ss}) \Longrightarrow \mathit{WN}\ ((\mathit{Abs}\ r\ {}^{\circ}\ s)\ {}^{\circ\circ}\ \mathit{ss})$

lemma *listrelp-imp-listsp1*:
assumes *H*: $\mathit{listrelp}\ (\lambda x\ y.\ P\ x)\ \mathit{xs}\ \mathit{ys}$
shows $\mathit{listsp}\ P\ \mathit{xs}$ ⟨*proof*⟩

lemma *listrelp-imp-listsp2*:
assumes *H*: $\mathit{listrelp}\ (\lambda x\ y.\ P\ y)\ \mathit{xs}\ \mathit{ys}$

shows *listsp P ys* ⟨*proof*⟩

lemma *lemma5*:
assumes *lred*: $r \rightarrow_l r'$
shows *WN* r **and** *NF* r' ⟨*proof*⟩

lemma *lemma6*:
assumes *wn*: *WN* r
shows $\exists\, r'.\ r \rightarrow_l r'$ ⟨*proof*⟩

lemma *lemma7*:
assumes *r*: $r \rightarrow_s r'$
shows *NF* $r' \Longrightarrow r \rightarrow_l r'$ ⟨*proof*⟩

lemma *WN-eq*: $\mathit{WN}\ t = (\exists\, t'.\ t \rightarrow_\beta{}^* t' \wedge \mathit{NF}\ t')$
⟨*proof*⟩

end

13 Weak normalization for simply-typed lambda calculus

theory *WeakNorm*
imports *Type NormalForm Code-Integer*
begin

Formalization by Stefan Berghofer. Partly based on a paper proof by Felix Joachimski and Ralph Matthes [1].

13.1 Main theorems

lemma *norm-list*:
assumes *f-compat*: $\bigwedge t\ t'.\ t \rightarrow_\beta{}^* t' \Longrightarrow f\ t \rightarrow_\beta{}^* f\ t'$
and *f-NF*: $\bigwedge t.\ \mathit{NF}\ t \Longrightarrow \mathit{NF}\ (f\ t)$
and *uNF*: $\mathit{NF}\ u$ **and** *uT*: $e \vdash u : T$
shows $\bigwedge \mathit{Us}.\ e\langle i{:}T\rangle \Vdash \mathit{as} : \mathit{Us} \Longrightarrow$
$\mathit{listall}\ (\lambda t.\ \forall\, e\ T'\ u\ i.\ e\langle i{:}T\rangle \vdash t : T' \longrightarrow$
$\mathit{NF}\ u \longrightarrow e \vdash u : T \longrightarrow (\exists\, t'.\ t[u/i] \rightarrow_\beta{}^* t' \wedge \mathit{NF}\ t'))\ \mathit{as} \Longrightarrow$
$\exists\, \mathit{as}'.\ \forall\, j.\ \mathit{Var}\ j \circ\circ \mathit{map}\ (\lambda t.\ f\ (t[u/i]))\ \mathit{as} \rightarrow_\beta{}^*$
$\mathit{Var}\ j \circ\circ \mathit{map}\ f\ \mathit{as}' \wedge \mathit{NF}\ (\mathit{Var}\ j \circ\circ \mathit{map}\ f\ \mathit{as}')$
(**is** $\bigwedge \mathit{Us}.\ \text{-} \Longrightarrow \mathit{listall}\ ?R\ \mathit{as} \Longrightarrow \exists\, \mathit{as}'.\ ?\mathit{ex}\ \mathit{Us}\ \mathit{as}\ \mathit{as}'$)
⟨*proof*⟩

lemma *subst-type-NF*:
$\bigwedge t\ e\ T\ u\ i.\ \mathit{NF}\ t \Longrightarrow e\langle i{:}U\rangle \vdash t : T \Longrightarrow \mathit{NF}\ u \Longrightarrow e \vdash u : U \Longrightarrow \exists\, t'.\ t[u/i]$
$\rightarrow_\beta{}^* t' \wedge \mathit{NF}\ t'$
(**is** $\mathit{PROP}\ ?P\ U$ **is** $\bigwedge t\ e\ T\ u\ i.\ \text{-} \Longrightarrow \mathit{PROP}\ ?Q\ t\ e\ T\ u\ i\ U$)

$\langle proof \rangle$
inductive *rtyping* :: $(nat \Rightarrow type) \Rightarrow dB \Rightarrow type \Rightarrow bool$ $(- \vdash_R - : - [50, 50, 50]$ $50)$
where
Var: $e\ x = T \Longrightarrow e \vdash_R Var\ x : T$
$|$ Abs: $e\langle 0{:}T\rangle \vdash_R t : U \Longrightarrow e \vdash_R Abs\ t : (T \Rightarrow U)$
$|$ App: $e \vdash_R s : T \Rightarrow U \Longrightarrow e \vdash_R t : T \Longrightarrow e \vdash_R (s \circ t) : U$

lemma *rtyping-imp-typing*: $e \vdash_R t : T \Longrightarrow e \vdash t : T$
$\langle proof \rangle$

theorem *type-NF*:
assumes $e \vdash_R t : T$
shows $\exists\, t'.\ t \rightarrow_\beta{}^* t' \wedge NF\ t'$ $\langle proof \rangle$

13.2 Extracting the program

declare *NF.induct* [*ind-realizer*]
declare *rtranclp.induct* [*ind-realizer irrelevant*]
declare *rtyping.induct* [*ind-realizer*]
lemmas [*extraction-expand*] = *conj-assoc listall-cons-eq*

extract *type-NF*
lemma *rtranclR-rtrancl-eq*: $rtranclpR\ r\ a\ b = r^{**}\ a\ b$
$\langle proof \rangle$

lemma *NFR-imp-NF*: $NFR\ nf\ t \Longrightarrow NF\ t$
$\langle proof \rangle$

The program corresponding to the proof of the central lemma, which performs substitution and normalization, is shown in Figure 1. The correctness theorem corresponding to the program *subst-type-NF* is

$$
\begin{aligned}
&\textstyle\bigwedge x.\ NFR\ x\ t \Longrightarrow \\
&\quad e\langle i{:}U\rangle \vdash t : T \Longrightarrow \\
&\quad (\textstyle\bigwedge xa.\ NFR\ xa\ u \Longrightarrow \\
&\qquad e \vdash u : U \Longrightarrow \\
&\qquad t[u/i] \rightarrow_\beta{}^* fst\ (subst\text{-}type\text{-}NF\ t\ e\ i\ U\ T\ u\ x\ xa) \wedge \\
&\qquad NFR\ (snd\ (subst\text{-}type\text{-}NF\ t\ e\ i\ U\ T\ u\ x\ xa))\ (fst\ (subst\text{-}type\text{-}NF\ t\ e\ i\ U \\
&T\ u\ x\ xa)))
\end{aligned}
$$

where *NFR* is the realizability predicate corresponding to the datatype *NFT*, which is inductively defined by the rules

```
subst-type-NF ≡
λx xa xb xc xd xe H Ha.
  type-induct-P xc
   (λx H2 H2a xa xb xc xd xe H.
      NFT-rec default
       (λts xa xaa r xb xc xd xe H.
          var-app-typesE-P (xb⟨xe:x⟩) xa ts
           (λUs--. case nat-eq-dec xa xe of
                  Left ⇒ case ts of [] ⇒ (xd, H)
                         | a # list ⇒
                            case Us-- of [] ⇒ default
                            | T''-- # Ts-- ⇒
                               let (x, y) =
                                     norm-list (λt. lift t 0) xd xb xe list Ts--
                                      (λt. lift-NF 0) H
                                      (listall-conj2-P-Q list (λi. (xaa (Suc i), r (Suc i))));
                                   (xa, ya) = snd (xaa 0, r 0) xb T''-- xd xe H;
                                   (xd, yb) = app-Var-NF 0 (lift-NF 0 H);
                                   (xa, ya) =
                                     H2 T''-- (Ts-- ⇛ xc) xd xb (Ts-- ⇛ xc) xa 0 yb ya;
                                   (x, y) =
                                     H2a T''-- (Ts-- ⇛ xc) (dB.Var 0 °° map (λt. lift t 0) x)
                                      xb xc xa 0 (y 0) ya
                               in (x, y)
                  | Right ⇒
                     let (x, y) =
                           let (x, y) =
                                 norm-list (λt. t) xd xb xe ts Us-- (λx H. H) H
                                  (listall-conj2-P-Q ts (λz. (xaa z, r z)))
                           in (x, λx. y x)
                     in case nat-le-dec xe xa of
                        Left ⇒ (dB.Var (xa − Suc 0) °° x, y (xa − Suc 0))
                        | Right ⇒ (dB.Var xa °° x, y xa)))
       (λt x r xa xb xc xd H.
          abs-typeE-P xb
           (λU V. let (x, y) =
                        let (x, y) = r (λa. (xa⟨0:U⟩) a) V (lift xc 0) (Suc xd) (lift-NF 0 H)
                        in (dB.Abs x, NFT.Abs x y)
                  in (x, y)))
       H (λa. xb a) xc xd xe)
   x xa xd xe xb H Ha
```

Figure 1: Program extracted from *subst-type-NF*

```
subst-Var-NF ≡
λx xa H.
  NFT-rec default
   (λts x xa r xb xc.
       case nat-eq-dec x xc of
       Left ⇒ NFT.App (map (λt. t[dB.Var xb/xc]) ts) xb
                (subst-terms-NF ts xb xc (listall-conj1-P-Q ts (λz. (xa z, r z)))
                  (listall-conj2-P-Q ts (λz. (xa z, r z))))
       | Right ⇒
           case nat-le-dec xc x of
           Left ⇒ NFT.App (map (λt. t[dB.Var xb/xc]) ts) (x − Suc 0)
                    (subst-terms-NF ts xb xc (listall-conj1-P-Q ts (λz. (xa z, r z)))
                      (listall-conj2-P-Q ts (λz. (xa z, r z))))
           | Right ⇒
               NFT.App (map (λt. t[dB.Var xb/xc]) ts) x
                (subst-terms-NF ts xb xc (listall-conj1-P-Q ts (λz. (xa z, r z)))
                  (listall-conj2-P-Q ts (λz. (xa z, r z)))))
   (λt x r xa xb. NFT.Abs (t[dB.Var (Suc xa)/Suc xb]) (r (Suc xa) (Suc xb))) H x xa

app-Var-NF ≡
λx. NFT-rec default
     (λts xa xaa r.
         (dB.Var xa °° (ts @ [dB.Var x]),
          NFT.App (ts @ [dB.Var x]) xa
           (snd (listall-app-P ts)
             (listall-conj1-P-Q ts (λz. (xaa z, r z)),
              listall-cons-P (Var-NF x) listall-nil-eq-P))))
     (λt xa r. (t[dB.Var x/0], subst-Var-NF x 0 xa))

lift-NF ≡
λx H. NFT-rec default
        (λts x xa r xb.
            case nat-le-dec x xb of
            Left ⇒ NFT.App (map (λt. lift t xb) ts) x
                     (lift-terms-NF ts xb (listall-conj1-P-Q ts (λz. (xa z, r z)))
                       (listall-conj2-P-Q ts (λz. (xa z, r z))))
            | Right ⇒
                NFT.App (map (λt. lift t xb) ts) (Suc x)
                 (lift-terms-NF ts xb (listall-conj1-P-Q ts (λz. (xa z, r z)))
                   (listall-conj2-P-Q ts (λz. (xa z, r z)))))
        (λt x r xa. NFT.Abs (lift t (Suc xa)) (r (Suc xa))) H x

type-NF ≡
λH. rtypingT-rec (λe x T. (dB.Var x, Var-NF x))
      (λe T t U x r. let (x, y) = r in (dB.Abs x, NFT.Abs x y))
      (λe s T U t x xa r ra.
          let (x, y) = r; (xa, ya) = ra;
              (x, y) =
                let (x, y) =
                      subst-type-NF (dB.App (dB.Var 0) (lift xa 0)) e 0 (T ⇒ U) U x
                       (NFT.App [lift xa 0] 0 (listall-cons-P (lift-NF 0 ya) listall-nil-P)) y
                in (x, y)
          in (x, y))
      H
```

Figure 2: Program extracted from lemmas and main theorem

$\forall\, i{<}length\ ts.\ NFR\ (nfs\ i)\ (ts\ !\ i) \Longrightarrow NFR\ (NFT.App\ ts\ x\ nfs)\ (dB.Var\ x\ {}^{\circ\circ}\ ts)$
$NFR\ nf\ t \Longrightarrow NFR\ (NFT.Abs\ t\ nf)\ (dB.Abs\ t)$

The programs corresponding to the main theorem *type-NF*, as well as to some lemmas, are shown in Figure 2. The correctness statement for the main function *type-NF* is

$\bigwedge x.\ rtypingR\ x\ e\ t\ T \Longrightarrow t \rightarrow_\beta{}^*\ fst\ (type\text{-}NF\ x) \wedge NFR\ (snd\ (type\text{-}NF\ x))\ (fst\ (type\text{-}NF\ x))$

where the realizability predicate *rtypingR* corresponding to the computationally relevant version of the typing judgement is inductively defined by the rules

$e\ x = T \Longrightarrow rtypingR\ (rtypingT.Var\ e\ x\ T)\ e\ (dB.Var\ x)\ T$
$rtypingR\ ty\ (e\langle 0{:}T\rangle)\ t\ U \Longrightarrow rtypingR\ (rtypingT.Abs\ e\ T\ t\ U\ ty)\ e\ (dB.Abs\ t)\ (T \Rightarrow U)$
$rtypingR\ ty\ e\ s\ (T \Rightarrow U) \Longrightarrow$
$rtypingR\ ty'\ e\ t\ T \Longrightarrow rtypingR\ (rtypingT.App\ e\ s\ T\ U\ t\ ty\ ty')\ e\ (dB.App\ s\ t)\ U$

13.3 Generating executable code

instantiation *NFT* :: *default*
begin

definition *default* = *Dummy* ()

instance ⟨*proof*⟩

end

instantiation *dB* :: *default*
begin

definition *default* = *dB.Var 0*

instance ⟨*proof*⟩

end

instantiation $*$ :: (*default*, *default*) *default*
begin

definition *default* = (*default*, *default*)

instance ⟨*proof*⟩

end

instantiation *list* :: (*type*) *default*
begin

definition *default* = []

instance ⟨*proof*⟩

end

instantiation *fun* :: (*type*, *default*) *default*
begin

definition *default* = (λx. *default*)

instance ⟨*proof*⟩

end

definition *int-of-nat* :: *nat* ⇒ *int* **where**
  int-of-nat = *of-nat*

The following functions convert between Isabelle's built-in `term` datatype and the generated `dB` datatype. This allows to generate example terms using Isabelle's parser and inspect normalized terms using Isabelle's pretty printer.

⟨*ML*⟩

The same story again for the SML code generator.

consts-code
  default ((*error default*))
  default :: ′*a* ⇒ ′*b*::*default* ((*fn* ′- => *error default*))

code-module *Norm*
contains
  test = *type-NF*

⟨*ML*⟩

We now try out the extracted program *type-NF* on some example terms.

⟨*ML*⟩

end

References

[1] F. Joachimski and R. Matthes. Short proofs of normalization for the simply-typed λ-calculus, permutative conversions and Gödel's T. *Archive for Mathematical Logic*, 42(1):59–87, 2003.

[2] R. Loader. Notes on Simply Typed Lambda Calculus. Technical Report ECS-LFCS-98-381, Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh, 1998.

[3] R. Matthes. Lambda Calculus: A Case for Inductive Definitions. In *Lecture notes of the 12th European Summer School in Logic, Language and Information (ESSLLI 2000)*. School of Computer Science, University of Birmingham, August 2000.

[4] M. Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118(1):120–127, April 1995.