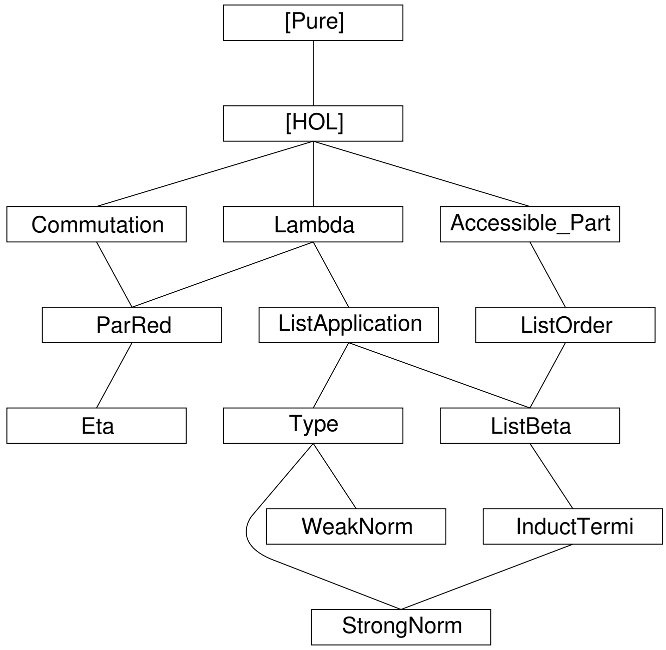
Fundamental Properties of Lambda-calculus

Tobias Nipkow
Stefan Berghofer

1st October 2005

Contents


[Pure]
[HOL]
Commutation
Lambda
Accessible_Part
ParRed
ListApplication
ListOrder
Eta
Type
ListBeta
WeakNorm
InductTermi
StrongNorm

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*

consts
 subst :: [*dB*, *dB*, *nat*] => *dB* (-[-'/-] [*300*, *0*, *0*] *300*)
 lift :: [*dB*, *nat*] => *dB*

primrec
 $lift\ (Var\ i)\ k = (if\ i < k\ then\ Var\ i\ else\ Var\ (i + 1))$
 $lift\ (s \circ t)\ k = lift\ s\ k \circ lift\ t\ k$
 $lift\ (Abs\ s)\ k = Abs\ (lift\ s\ (k + 1))$

primrec
 subst-Var: $(Var\ i)[s/k] =$
  $(if\ k < i\ then\ Var\ (i - 1)\ else\ if\ i = k\ then\ s\ else\ Var\ i)$
 subst-App: $(t \circ u)[s/k] = t[s/k] \circ u[s/k]$
 subst-Abs: $(Abs\ t)[s/k] = Abs\ (t[lift\ s\ 0\ /\ k+1])$

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

Optimized versions of *subst* and *lift*.

consts
 substn :: [*dB*, *dB*, *nat*] => *dB*
 liftn :: [*nat*, *dB*, *nat*] => *dB*

primrec
 $liftn\ n\ (Var\ i)\ k = (if\ i < k\ then\ Var\ i\ else\ Var\ (i + n))$
 $liftn\ n\ (s \circ t)\ k = liftn\ n\ s\ k \circ liftn\ n\ t\ k$
 $liftn\ n\ (Abs\ s)\ k = Abs\ (liftn\ n\ s\ (k + 1))$

primrec
 $substn\ (Var\ i)\ s\ k =$
  $(if\ k < i\ then\ Var\ (i - 1)\ else\ if\ i = k\ then\ liftn\ k\ s\ 0\ else\ Var\ i)$
 $substn\ (t \circ u)\ s\ k = substn\ t\ s\ k \circ substn\ u\ s\ k$
 $substn\ (Abs\ t)\ s\ k = Abs\ (substn\ t\ s\ (k + 1))$

1.2 Beta-reduction

consts
 $beta :: (dB \times dB)\ set$

syntax

-beta :: [*dB*, *dB*] => *bool* (**infixl** -> *50*)
-beta-rtrancl :: [*dB*, *dB*] => *bool* (**infixl** ->> *50*)
syntax (*latex*)
-beta :: [*dB*, *dB*] => *bool* (**infixl** $\to_\beta$ *50*)
-beta-rtrancl :: [*dB*, *dB*] => *bool* (**infixl** $\to_\beta{}^*$ *50*)
translations
$s \to_\beta t == (s, t) \in beta$
$s \to_\beta{}^* t == (s, t) \in beta\hat{}*$

inductive *beta*
intros
beta [*simp*, *intro*!]: $Abs\ s \circ t \to_\beta s[t/0]$
appL [*simp*, *intro*!]: $s \to_\beta t ==> s \circ u \to_\beta t \circ u$
appR [*simp*, *intro*!]: $s \to_\beta t ==> u \circ s \to_\beta u \circ t$
abs [*simp*, *intro*!]: $s \to_\beta t ==> Abs\ s \to_\beta Abs\ t$

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*]: $j < i \Longrightarrow (Var\ j)[u/i] = Var\ j$
⟨*proof*⟩

lemma *lift-lift* [*rule-format*]:
$\forall\, i\ k.\ i < k + 1 \longrightarrow lift\ (lift\ t\ i)\ (Suc\ k) = lift\ (lift\ t\ k)\ i$
⟨*proof*⟩

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

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

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

lemma *subst-subst* [*rule-format*]:
$\forall\, i\ j\ u\ v.\ i < j + 1 \longrightarrow t[lift\ v\ i\ /\ Suc\ j][u[v/j]/i] = t[u/i][v/j]$
⟨*proof*⟩

1.5 Equivalence proof for optimized substitution

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

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

lemma *substn-subst-n* [*simp*]:
$\forall\, n.\ substn\ t\ s\ n = t[liftn\ n\ s\ 0\ /\ n]$
⟨*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*]:
$r \to_\beta s \Longrightarrow (\bigwedge t\ i.\ r[t/i] \to_\beta s[t/i])$
⟨*proof*⟩

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

theorem *lift-preserves-beta* [*simp*]:
$r \rightarrow_\beta s \Longrightarrow (\bigwedge i.\ lift\ r\ i \rightarrow_\beta lift\ s\ i)$
⟨*proof*⟩

theorem *lift-preserves-beta′*: $r \rightarrow_\beta{}^* s \Longrightarrow lift\ r\ i \rightarrow_\beta{}^* lift\ s\ i$
⟨*proof*⟩

theorem *subst-preserves-beta2* [*simp*]:
$\bigwedge r\ s\ i.\ r \rightarrow_\beta s \Longrightarrow t[r/i] \rightarrow_\beta{}^* t[s/i]$
⟨*proof*⟩

theorem *subst-preserves-beta2′*: $r \rightarrow_\beta{}^* s \Longrightarrow t[r/i] \rightarrow_\beta{}^* t[s/i]$
⟨*proof*⟩

end

2 Abstract commutation and confluence notions

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

2.1 Basic definitions

constdefs
square :: $[('a \times 'a)\ set, ('a \times 'a)\ set, ('a \times 'a)\ set, ('a \times 'a)\ set] \Rightarrow bool$
square $R\ S\ T\ U$ ==
$\forall x\ y.\ (x, y) \in R \longrightarrow (\forall z.\ (x, z) \in S \longrightarrow (\exists u.\ (y, u) \in T \wedge (z, u) \in U))$

commute :: $[('a \times 'a)\ set, ('a \times 'a)\ set] \Rightarrow bool$
commute $R\ S$ == *square* $R\ S\ S\ R$

diamond :: $('a \times 'a)\ set \Rightarrow bool$
diamond R == *commute* $R\ R$

Church-Rosser :: $('a \times 'a)\ set \Rightarrow bool$
Church-Rosser R ==
$\forall x\ y.\ (x, y) \in (R \cup R\hat{}{-1})\hat{}{*} \longrightarrow (\exists z.\ (x, z) \in R\hat{}{*} \wedge (y, z) \in R\hat{}{*})$

syntax
confluent :: $('a \times 'a)\ set \Rightarrow bool$
translations
confluent R == *diamond* $(R\hat{}{*})$

2.2 Basic lemmas

square

lemma *square-sym*: *square* $R\ S\ T\ U \Longrightarrow$ *square* $S\ R\ U\ T$

⟨*proof*⟩

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

lemma *square-reflcl*:
$[| \ square\ R\ S\ T\ (R\hat{}=);\ S \subseteq T \ |] ==> square\ (R\hat{}=)\ S\ T\ (R\hat{}=)$
⟨*proof*⟩

lemma *square-rtrancl*:
$square\ R\ S\ S\ T ==> square\ (R\hat{}*)\ S\ S\ (T\hat{}*)$
⟨*proof*⟩

lemma *square-rtrancl-reflcl-commute*:
$square\ R\ S\ (S\hat{}*)\ (R\hat{}=) ==> commute\ (R\hat{}*)\ (S\hat{}*)$
⟨*proof*⟩

commute

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

lemma *commute-rtrancl*: $commute\ R\ S ==> commute\ (R\hat{}*)\ (S\hat{}*)$
⟨*proof*⟩

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

diamond, confluence, and union

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

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

lemma *square-reflcl-confluent*:
$square\ R\ R\ (R\hat{}=)\ (R\hat{}=) ==> confluent\ R$
⟨*proof*⟩

lemma *confluent-Un*:
$[| \ confluent\ R;\ confluent\ S;\ commute\ (R\hat{}*)\ (S\hat{}*) \ |] ==> confluent\ (R \cup S)$
⟨*proof*⟩

lemma *diamond-to-confluence*:
$[| \ diamond\ R;\ T \subseteq R;\ R \subseteq T\hat{}* \ |] ==> confluent\ T$
⟨*proof*⟩

2.3 Church-Rosser

lemma *Church-Rosser-confluent*: $Church\text{-}Rosser\ R = confluent\ R$
⟨*proof*⟩

2.4 Newman's lemma

Proof by Stefan Berghofer

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

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

This is the maximal amount of automation possible at the moment.

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

end

3 Parallel reduction and a complete developments

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

3.1 Parallel reduction

consts
par-beta :: $(dB \times dB)\ set$

syntax
par-beta :: $[dB,\ dB]$ => *bool* (**infixl** => *50*)
translations
s => *t* == $(s,\ t) \in$ *par-beta*

inductive *par-beta*
intros
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 ⊆ *par-beta* ⊆ *beta*ˆ∗

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*ˆ∗
⟨*proof*⟩

3.3 Misc properties of par-beta

lemma *par-beta-lift* [*rule-format*, *simp*]:
$\forall t'\ n.\ t => t' --> lift\ t\ n => lift\ t'\ n$
⟨*proof*⟩

lemma *par-beta-subst* [*rule-format*]:
$\forall s\ s'\ t'\ n.\ s => s' --> t => t' --> t[s/n] => t'[s'/n]$
⟨*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* [*rule-format*]:
$\forall$ *t. s => t --> t => 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

consts
free :: *dB => nat => bool*
primrec
free (Var j) i = (j = i)
free (s ° t) i = (free s i $\vee$ *free t i)*
free (Abs s) i = free s (i + 1)

consts
eta :: *(dB* $\times$ *dB) set*

syntax
-eta :: *[dB, dB] => bool* (**infixl** *-e> 50*)
-eta-rtrancl :: *[dB, dB] => bool* (**infixl** *-e>> 50*)
-eta-reflcl :: *[dB, dB] => bool* (**infixl** *-e>= 50*)
translations
s -e> t == (s, t) $\in$ *eta*
s -e>> t == (s, t) $\in$ *eta^**
s -e>= t == (s, t) $\in$ *eta^=*

inductive *eta*
intros
eta [*simp*, *intro*]: $\neg$ *free s 0 ==> Abs (s ° Var 0) -e> s[dummy/0]*
appL [*simp*, *intro*]: *s -e> t ==> s ° u -e> t ° u*
appR [*simp*, *intro*]: *s -e> t ==> u ° s -e> u ° t*
abs [*simp*, *intro*]: *s -e> t ==> Abs s -e> Abs t*

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

4.2 Properties of eta, subst and free

lemma *subst-not-free* [*rule-format*, *simp*]:
$\forall\, i\ t\ u.\ \neg\ \mathit{free}\ s\ i\ {-}{-}{>}\ s[t/i] = s[u/i]$
⟨*proof*⟩

lemma *free-lift* [*simp*]:
$\forall\, i\ k.\ \mathit{free}\ (\mathit{lift}\ t\ k)\ i =$
$(i < k \land \mathit{free}\ t\ i \lor k < i \land \mathit{free}\ t\ (i - 1))$
⟨*proof*⟩

lemma *free-subst* [*simp*]:
$\forall\, i\ k\ t.\ \mathit{free}\ (s[t/k])\ i =$
$(\mathit{free}\ s\ k \land \mathit{free}\ t\ i \lor \mathit{free}\ s\ (\mathit{if}\ i < k\ \mathit{then}\ i\ \mathit{else}\ i + 1))$
⟨*proof*⟩

lemma *free-eta* [*rule-format*]:
$s\ {-}e{>}\ t\ {=}{=}{>}\ \forall\, i.\ \mathit{free}\ t\ i = \mathit{free}\ s\ i$
⟨*proof*⟩

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

lemma *eta-subst* [*rule-format*, *simp*]:
$s\ {-}e{>}\ t\ {=}{=}{>}\ \forall\, u\ i.\ s[u/i]\ {-}e{>}\ t[u/i]$
⟨*proof*⟩

theorem *lift-subst-dummy*: $\bigwedge i\ \mathit{dummy}.\ \neg\ \mathit{free}\ s\ i \Longrightarrow \mathit{lift}\ (s[\mathit{dummy}/i])\ i = s$
⟨*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*: $s\ {-}e{>}{>}\ s'\ {=}{=}{>}\ \mathit{Abs}\ s\ {-}e{>}{>}\ \mathit{Abs}\ s'$
⟨*proof*⟩

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

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

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

4.5 Commutation of beta and eta

lemma *free-beta* [*rule-format*]:
$s\ -{>}\ t\ ==>\ \forall\, i.\ free\ t\ i\ --{>}\ free\ s\ i$
⟨*proof*⟩

lemma *beta-subst* [*rule-format*, *intro*]:
$s\ -{>}\ t\ ==>\ \forall\, u\ i.\ s[u/i]\ -{>}\ t[u/i]$
⟨*proof*⟩

lemma *subst-Var-Suc* [*simp*]: $\forall\, i.\ t[Var\ i/i] = t[Var(i)/i + 1]$
⟨*proof*⟩

lemma *eta-lift* [*rule-format*, *simp*]:
$s\ -e{>}\ t\ ==>\ \forall\, i.\ lift\ s\ i\ -e{>}\ lift\ t\ i$
⟨*proof*⟩

lemma *rtrancl-eta-subst* [*rule-format*]:
$\forall\, s\ t\ i.\ s\ -e{>}\ t\ --{>}\ u[s/i]\ -e{>}{>}\ u[t/i]$
⟨*proof*⟩

lemma *square-beta-eta*: *square beta eta* $(eta\hat{}{*})\ (beta\hat{}{=})$
⟨*proof*⟩

lemma *confluent-beta-eta*: *confluent* $(beta \cup eta)$
⟨*proof*⟩

4.6 Implicit definition of eta

$Abs\ (lift\ s\ 0\ \circ\ Var\ 0)\ -e{>}\ s$

lemma *not-free-iff-lifted* [*rule-format*]:
$\forall\, i.\ (\neg\ free\ s\ i) = (\exists\, t.\ s = lift\ t\ i)$
⟨*proof*⟩

theorem *explicit-is-implicit*:
$(\forall\, s\ u.\ (\neg\ free\ s\ 0)\ --{>}\ R\ (Abs\ (s\ \circ\ Var\ 0))\ (s[u/0])) =$
$(\forall\, s.\ R\ (Abs\ (lift\ s\ 0\ \circ\ Var\ 0))\ s)$
⟨*proof*⟩

4.7 Parallel eta-reduction

consts
  par-eta :: (*dB* × *dB*) *set*

syntax
  -par-eta :: [*dB*, *dB*] => *bool* (**infixl** =*e*> *50*)
translations
  s =*e*> *t* == (*s*, *t*) ∈ *par-eta*

syntax (*xsymbols*)
  -par-eta :: [*dB*, *dB*] => *bool* (**infixl** $\Rightarrow_\eta$ *50*)

inductive *par-eta*
intros
  var [*simp*, *intro*]: *Var x* $\Rightarrow_\eta$ *Var x*
  eta [*simp*, *intro*]: ¬ *free s 0* ⟹ *s* $\Rightarrow_\eta$ *s*′⟹ *Abs* (*s* ° *Var 0*) $\Rightarrow_\eta$ *s*′[*dummy*/*0*]
  app [*simp*, *intro*]: *s* $\Rightarrow_\eta$ *s*′ ⟹ *t* $\Rightarrow_\eta$ *t*′ ⟹ *s* ° *t* $\Rightarrow_\eta$ *s*′ ° *t*′
  abs [*simp*, *intro*]: *s* $\Rightarrow_\eta$ *t* ⟹ *Abs s* $\Rightarrow_\eta$ *Abs t*

lemma *free-par-eta* [*simp*]: **assumes** *eta*: *s* $\Rightarrow_\eta$ *t*
  shows ⋀*i*. *free t i* = *free s i* ⟨*proof*⟩

lemma *par-eta-refl* [*simp*]: *t* $\Rightarrow_\eta$ *t*
  ⟨*proof*⟩

lemma *par-eta-lift* [*simp*]:
  assumes *eta*: *s* $\Rightarrow_\eta$ *t*
  shows ⋀*i*. *lift s i* $\Rightarrow_\eta$ *lift t i* ⟨*proof*⟩

lemma *par-eta-subst* [*simp*]:
  assumes *eta*: *s* $\Rightarrow_\eta$ *t*
  shows ⋀*u u*′ *i*. *u* $\Rightarrow_\eta$ *u*′ ⟹ *s*[*u*/*i*] $\Rightarrow_\eta$ *t*[*u*′/*i*] ⟨*proof*⟩

theorem *eta-subset-par-eta*: *eta* ⊆ *par-eta*
  ⟨*proof*⟩

theorem *par-eta-subset-eta*: *par-eta* ⊆ *eta**
  ⟨*proof*⟩

4.8 n-ary eta-expansion

consts *eta-expand* :: *nat* ⇒ *dB* ⇒ *dB*
primrec
  eta-expand-0: *eta-expand 0 t* = *t*
  eta-expand-Suc: *eta-expand* (*Suc n*) *t* = *Abs* (*lift* (*eta-expand n t*) *0* ° *Var 0*)

lemma *eta-expand-Suc*′:
  ⋀*t*. *eta-expand* (*Suc n*) *t* = *eta-expand n* (*Abs* (*lift t 0* ° *Var 0*))
  ⟨*proof*⟩

theorem *lift-eta-expand*: *lift* (*eta-expand* k t) i = *eta-expand* k (*lift* t i)
⟨*proof*⟩

theorem *eta-expand-beta*:
assumes u: u => u'
shows $\bigwedge t\ t'.\ t$ => $t' \Longrightarrow$ *eta-expand* k (*Abs* t) ° u => $t'[u'/0]$
⟨*proof*⟩

theorem *eta-expand-red*:
assumes t: t => t'
shows *eta-expand* k t => *eta-expand* k t'
⟨*proof*⟩

theorem *eta-expand-eta*: $\bigwedge t\ t'.\ t \Rightarrow_\eta t' \Longrightarrow$ *eta-expand* k $t \Rightarrow_\eta t'$
⟨*proof*⟩

4.9 Elimination rules for parallel eta reduction

theorem *par-eta-elim-app*: **assumes** *eta*: $t \Rightarrow_\eta u$
shows $\bigwedge u1'\ u2'.\ u = u1' \circ u2' \Longrightarrow$
$\exists\, u1\ u2\ k.\ t =$ *eta-expand* k $(u1 \circ u2) \land u1 \Rightarrow_\eta u1' \land u2 \Rightarrow_\eta u2'$ ⟨*proof*⟩

theorem *par-eta-elim-abs*: **assumes** *eta*: $t \Rightarrow_\eta t'$
shows $\bigwedge u'.\ t' =$ *Abs* $u' \Longrightarrow$
$\exists\, u\ k.\ t =$ *eta-expand* k (*Abs* u) $\land\ u \Rightarrow_\eta u'$ ⟨*proof*⟩

4.10 Eta-postponement theorem

Based on a proof by Masako Takahashi [2].

theorem *par-eta-beta*: $\bigwedge s\ u.\ s \Rightarrow_\eta t \Longrightarrow t$ => $u \Longrightarrow \exists\, t'.\ s$ => $t' \land t' \Rightarrow_\eta u$
⟨*proof*⟩

theorem *eta-postponement'*: **assumes** *eta*: s −e>> t
shows $\bigwedge u.\ t$ => $u \Longrightarrow \exists\, t'.\ s$ => $t' \land t'$ −e>> u
⟨*proof*⟩

theorem *eta-postponement*:
assumes *st*: $(s, t) \in (beta \cup eta)^*$
shows $(s, t) \in eta^*\ O\ beta^*$ ⟨*proof*⟩

end

5 Application of a term to a list of terms

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

syntax
  -list-application :: dB => dB *list* => dB (**infixl** °° *150*)
translations
  t °° ts == *foldl* (*op* °) t ts

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

lemma *Var-eq-apps-conv* [*iff*]:
  $\bigwedge s.$ (*Var* m = s °° ss) = (*Var* m = s ∧ ss = [])
  ⟨*proof*⟩

lemma *Var-apps-eq-Var-apps-conv* [*iff*]:
  $\bigwedge ss.$ (*Var* m °° rs = *Var* n °° ss) = (m = n ∧ rs = ss)
  ⟨*proof*⟩

lemma *App-eq-foldl-conv*:
  (r ° s = t °° ts) =
    (*if* ts = [] *then* r ° s = t
    else (∃ ss. ts = ss @ [s] ∧ r = t °° ss))
  ⟨*proof*⟩

lemma *Abs-eq-apps-conv* [*iff*]:
  (*Abs* r = s °° ss) = (*Abs* r = s ∧ ss = [])
  ⟨*proof*⟩

lemma *apps-eq-Abs-conv* [*iff*]: (s °° ss = *Abs* r) = (s = *Abs* r ∧ ss = [])
  ⟨*proof*⟩

lemma *Abs-apps-eq-Abs-apps-conv* [*iff*]:
  $\bigwedge ss.$ (*Abs* r °° rs = *Abs* s °° ss) = (r = s ∧ rs = ss)
  ⟨*proof*⟩

lemma *Abs-App-neq-Var-apps* [*iff*]:
  ∀ s t. *Abs* s ° t ~= *Var* n °° ss
  ⟨*proof*⟩

lemma *Var-apps-neq-Abs-apps* [*iff*]:
  $\bigwedge ts.$ *Var* n °° ts ~= *Abs* r °° ss
  ⟨*proof*⟩

lemma *ex-head-tail*:
  ∃ ts h. t = h °° ts ∧ ((∃ n. h = *Var* n) ∨ (∃ u. h = *Abs* u))
  ⟨*proof*⟩

lemma *size-apps* [*simp*]:
  size (r °° rs) = *size* r + *foldl* (*op* +) *0* (*map size* rs) + *length* rs
  ⟨*proof*⟩

lemma *lem0*: [| (*0*::*nat*) < *k*; *m* <= *n* |] ==> *m* < *n* + *k*
⟨*proof*⟩

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

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

lemma *app-last*: $(t \circ\circ\ ts) \circ\ u = t \circ\circ\ (ts\ @\ [u])$
⟨*proof*⟩

A customized induction schema for °°.

lemma *lem* [*rule-format* (*no-asm*)]:
[| !!$n\ ts.\ \forall t \in set\ ts.\ P\ t$ ==> $P\ (Var\ n \circ\circ\ ts)$;
!!$u\ ts.$ [| $P\ u$; $\forall t \in set\ ts.\ P\ t$ |] ==> $P\ (Abs\ u \circ\circ\ ts)$
|] ==> $\forall t.\ size\ t = n$ −−> $P\ t$
⟨*proof*⟩

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

end

6 Simply-typed lambda terms

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

6.1 Environments

constdefs
shift :: $(nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a \Rightarrow nat \Rightarrow 'a$ (-<-:-> [*90*, *0*, *0*] *91*)
e<*i*:*a*> ≡ $\lambda j.$ *if* $j < i$ *then* $e\ j$ *else if* $j = i$ *then* a *else* $e\ (j - 1)$
syntax (*xsymbols*)
shift :: $(nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a \Rightarrow nat \Rightarrow 'a$ (-⟨-:-⟩ [*90*, *0*, *0*] *91*)
syntax (*HTML* **output**)
shift :: $(nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a \Rightarrow nat \Rightarrow 'a$ (-⟨-:-⟩ [*90*, *0*, *0*] *91*)

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

lemma *shift-gt* [*simp*]: $j < i \Longrightarrow (e\langle i{:}T\rangle)\ j = e\ j$

$\langle proof \rangle$

lemma *shift-lt* [*simp*]: $i < j \Longrightarrow (e\langle i{:}T\rangle)\ j = e\ (j - 1)$
$\langle proof \rangle$

lemma *shift-commute* [*simp*]: $e\langle i{:}U\rangle\langle 0{:}T\rangle = e\langle 0{:}T\rangle\langle Suc\ i{:}U\rangle$
$\langle proof \rangle$

6.2 Types and typing rules

datatype *type* =
  Atom nat
| *Fun type type* (**infixr** $\Rightarrow$ *200*)

consts
  $typing :: ((nat \Rightarrow type) \times dB \times type)\ set$
  $typings :: (nat \Rightarrow type) \Rightarrow dB\ list \Rightarrow type\ list \Rightarrow bool$

syntax
  $\text{-}funs :: type\ list \Rightarrow type \Rightarrow type$ (**infixr** =>> *200*)
  $\text{-}typing :: (nat \Rightarrow type) \Rightarrow dB \Rightarrow type \Rightarrow bool$ (- |− - : - [*50*, *50*, *50*] *50*)
  $\text{-}typings :: (nat \Rightarrow type) \Rightarrow dB\ list \Rightarrow type\ list \Rightarrow bool$
  (- ||− - : - [*50*, *50*, *50*] *50*)
syntax (*xsymbols*)
  $\text{-}typing :: (nat \Rightarrow type) \Rightarrow dB \Rightarrow type \Rightarrow bool$ (- $\vdash$ - : - [*50*, *50*, *50*] *50*)
syntax (*latex*)
  $\text{-}funs :: type\ list \Rightarrow type \Rightarrow type$ (**infixr** $\Rrightarrow$ *200*)
  $\text{-}typings :: (nat \Rightarrow type) \Rightarrow dB\ list \Rightarrow type\ list \Rightarrow bool$
  (- $\Vdash$ - : - [*50*, *50*, *50*] *50*)
translations
  $Ts \Rrightarrow T \rightleftharpoons foldr\ Fun\ Ts\ T$
  $env \vdash t : T \rightleftharpoons (env, t, T) \in typing$
  $env \Vdash ts : Ts \rightleftharpoons typings\ env\ ts\ Ts$

inductive *typing*
 intros
  Var [*intro*!]: $env\ x = T \Longrightarrow env \vdash Var\ x : T$
  Abs [*intro*!]: $env\langle 0{:}T\rangle \vdash t : U \Longrightarrow env \vdash Abs\ t : (T \Rightarrow U)$
  App [*intro*!]: $env \vdash s : T \Rightarrow U \Longrightarrow env \vdash t : T \Longrightarrow env \vdash (s \circ t) : U$

inductive-cases *typing-elims* [*elim*!]:
  $e \vdash Var\ i : T$
  $e \vdash t \circ u : T$
  $e \vdash Abs\ t : T$

primrec
  $(e \Vdash [] : Ts) = (Ts = [])$
  $(e \Vdash (t \# ts) : Ts) =$
  $(case\ Ts\ of$

$[] \Rightarrow False$
$\mid T \# Ts \Rightarrow e \vdash t : T \wedge e \Vdash ts : Ts)$

6.3 Some examples

lemma $e \vdash Abs\ (Abs\ (Abs\ (Var\ 1 \circ (Var\ 2 \circ Var\ 1 \circ Var\ 0)))) : ?T$
⟨*proof*⟩

lemma $e \vdash Abs\ (Abs\ (Abs\ (Var\ 2 \circ Var\ 0 \circ (Var\ 1 \circ Var\ 0)))) : ?T$
⟨*proof*⟩

6.4 Lists of types

lemma *lists-typings*:
$\bigwedge Ts.\ e \Vdash ts : Ts \Longrightarrow ts \in lists\ \{t.\ \exists T.\ e \vdash t : T\}$
⟨*proof*⟩

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

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

lemma *rev-exhaust2* [*case-names Nil snoc*, *extraction-expand*]:
$(xs = [] \Longrightarrow P) \Longrightarrow (\bigwedge ys\ y.\ xs = ys\ @\ [y] \Longrightarrow P) \Longrightarrow P$
— 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*:
$\bigwedge t\ T.\ e \vdash t \circ\circ ts : T \Longrightarrow \exists Ts.\ e \vdash t : Ts \Rrightarrow T \wedge e \Vdash ts : Ts$
⟨*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*:
$\bigwedge t\ T\ Ts.\ e \vdash t : Ts \Rrightarrow T \Longrightarrow e \Vdash ts : Ts \Longrightarrow e \vdash t \circ\circ ts : T$
⟨*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*:
$\bigwedge T\ U.\ e \vdash Var\ i\ {}^{\circ\circ}\ ts : T \Longrightarrow e \vdash Var\ i\ {}^{\circ\circ}\ ts : U \Longrightarrow T = U$
⟨*proof*⟩

lemma *var-app-types*: $\bigwedge ts\ Ts\ U.\ e \vdash Var\ i\ {}^{\circ\circ}\ ts\ {}^{\circ\circ}\ us : T \Longrightarrow e \Vdash ts : Ts \Longrightarrow$
$e \vdash Var\ i\ {}^{\circ\circ}\ ts : U \Longrightarrow \exists\, Us.\ U = Us \Rrightarrow T \wedge e \Vdash us : Us$
⟨*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*!]: $e \vdash t : T \Longrightarrow (\bigwedge i\ U.\ e\langle i{:}U\rangle \vdash lift\ t\ i : T)$
⟨*proof*⟩

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

6.7 Substitution lemmas

lemma *subst-lemma*:
$e \vdash t : T \Longrightarrow (\bigwedge e'\ i\ U\ u.\ e' \vdash u : U \Longrightarrow e = e'\langle i{:}U\rangle \Longrightarrow e' \vdash t[u/i] : T)$
⟨*proof*⟩

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

6.8 Subject reduction

lemma *subject-reduction*: $e \vdash t : T \Longrightarrow (\bigwedge t'.\ t\ {-}{>}\ t' \Longrightarrow e \vdash t' : T)$
⟨*proof*⟩

theorem *subject-reduction'*: $t \to_\beta{}^*\ t' \Longrightarrow e \vdash t : T \Longrightarrow e \vdash t' : T$
⟨*proof*⟩

6.9 Alternative induction rule for types

lemma *type-induct* [*induct type*]:
$(\bigwedge T.\ (\bigwedge T1\ T2.\ T = T1 \Rightarrow T2 \Longrightarrow P\ T1) \Longrightarrow$
$(\bigwedge T1\ T2.\ T = T1 \Rightarrow T2 \Longrightarrow P\ T2) \Longrightarrow P\ T) \Longrightarrow P\ T$
⟨*proof*⟩

end

7 Lifting an order to lists of elements

theory *ListOrder* **imports** *Accessible-Part* **begin**

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

constdefs
$step1 :: ('a \times 'a)\ set => ('a\ list \times 'a\ list)\ set$
$step1\ r ==$
$\{(ys, xs).\ \exists us\ z\ z'\ vs.\ xs = us\ @\ z\ \#\ vs \wedge (z', z) \in r \wedge ys =$
$us\ @\ z'\ \#\ vs\}$

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

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

lemma *not-Nil-step1* [*iff*]: $([], xs) \notin step1\ r$
⟨*proof*⟩

lemma *not-step1-Nil* [*iff*]: $(xs, []) \notin step1\ r$
⟨*proof*⟩

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

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

lemma *Cons-step1E* [*rule-format*, *elim*!]:
$[|\ (ys, x\ \#\ xs) \in step1\ r;$
$\forall y.\ ys = y\ \#\ xs\ {-}{-}{>}\ (y, x) \in r\ {-}{-}{>}\ R;$
$\forall zs.\ ys = x\ \#\ zs\ {-}{-}{>}\ (zs, xs) \in step1\ r\ {-}{-}{>}\ R$
$|] ==> R$
⟨*proof*⟩

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

lemma *Cons-acc-step1I* [*rule-format*, *intro*!]:
  x ∈ acc r ==> ∀xs. xs ∈ acc (step1 r) −−> x # xs ∈ acc (step1 r)
⟨*proof*⟩

lemma *lists-accD*: xs ∈ lists (acc r) ==> xs ∈ acc (step1 r)
⟨*proof*⟩

lemma *ex-step1I*:
[| x ∈ set xs; (y, x) ∈ r |]
  ==> ∃ys. (ys, xs) ∈ step1 r ∧ y ∈ set ys
⟨*proof*⟩

lemma *lists-accI*: xs ∈ acc (step1 r) ==> xs ∈ lists (acc r)
⟨*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.

syntax
-list-beta :: dB => dB => bool (**infixl** => *50*)
translations
rs => ss == (rs, ss) : step1 beta

lemma *head-Var-reduction-aux*:
v −> v′ ==> ∀rs. v = Var n °° rs −−> (∃ss. rs => ss ∧ v′ = Var n °° ss)
⟨*proof*⟩

lemma *head-Var-reduction*:
Var n °° rs −> v ==> (∃ss. rs => ss ∧ v = Var n °° ss)
⟨*proof*⟩

lemma *apps-betasE-aux*:
u −> u′ ==> ∀r rs. u = r °° rs −−>
  ((∃r′. r −> r′ ∧ u′ = r′ °° rs) ∨
   (∃rs′. rs => rs′ ∧ u′ = r °° rs′) ∨
   (∃s t ts. r = Abs s ∧ rs = t # ts ∧ u′ = s[t/0] °° ts))
⟨*proof*⟩

lemma *apps-betasE* [*elim*!]:
[| r °° rs −> s; !!r′. [| r −> r′; s = r′ °° rs |] ==> R;
  !!rs′. [| rs => rs′; s = r °° rs′ |] ==> R;
  !!t u us. [| r = Abs t; rs = u # us; s = t[u/0] °° us |] ==> R |]
==> R
⟨*proof*⟩

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

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

lemma *apps-preserves-betas* [*rule-format*, *simp*]:
$\forall\, ss.\ rs \; {=}{>} \; ss \; {-}{-}{>} \; r \circ\circ \; rs \; {-}{>} \; r \circ\circ \; ss$
⟨*proof*⟩

end

9 Inductive characterization of terminating lambda terms

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

9.1 Terminating lambda terms

consts
IT :: *dB set*

inductive *IT*
intros
Var [*intro*]: $rs : lists\ IT \; {=}{=}{>} \; Var\ n \circ\circ \; rs : IT$
Lambda [*intro*]: $r : IT \; {=}{=}{>} \; Abs\ r : IT$
Beta [*intro*]: $(r[s/0]) \circ\circ \; ss : IT \; {=}{=}{>} \; s : IT \; {=}{=}{>} \; (Abs\ r \circ s) \circ\circ \; ss : IT$

9.2 Every term in IT terminates

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

lemma *IT-implies-termi*: $t : IT \; {=}{=}{>} \; t : termi\ beta$
⟨*proof*⟩

9.3 Every terminating term is in IT

declare *Var-apps-neq-Abs-apps* [*THEN not-sym*, *simp*]

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

lemma [*simp*]:
$(Abs\ r\ ^\circ\ s\ ^{\circ\circ}\ ss = Abs\ r'\ ^\circ\ s'\ ^{\circ\circ}\ ss') = (r = r' \wedge s = s' \wedge ss = ss')$
$\langle proof \rangle$

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

theorem *termi-implies-IT*: $r : termi\ beta ==> r : IT$
$\langle proof \rangle$

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*!]: $t \in IT \Longrightarrow (\bigwedge i.\ lift\ t\ i \in IT)$
$\langle proof \rangle$

lemma *lifts-IT*: $ts \in lists\ IT \Longrightarrow map\ (\lambda t.\ lift\ t\ 0)\ ts \in lists\ IT$
$\langle proof \rangle$

lemma *subst-Var-IT*: $r \in IT \Longrightarrow (\bigwedge i\ j.\ r[Var\ i/j] \in IT)$
$\langle proof \rangle$

lemma *Var-IT*: $Var\ n \in IT$
$\langle proof \rangle$

lemma *app-Var-IT*: $t \in IT \Longrightarrow t\ ^\circ\ Var\ i \in IT$
$\langle proof \rangle$

10.2 Well-typed substitution preserves termination

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

10.3 Well-typed terms are strongly normalizing

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

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

end

11 Weak normalization for simply-typed lambda calculus

theory *WeakNorm* **imports** *Type* **begin**

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

11.1 Terms in normal form

constdefs
$listall :: ('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow bool$
$listall\ P\ xs \equiv (\forall i.\ i < length\ xs \longrightarrow P\ (xs\ !\ i))$

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*]: $listall\ P\ (x\ \#\ xs) = (P\ x \land listall\ P\ xs)$
⟨*proof*⟩

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

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

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

lemma *listall-snoc* [*simp*]: $listall\ P\ (xs\ @\ [x]) = (listall\ P\ xs \land P\ x)$

$\langle proof \rangle$

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

consts *NF* :: *dB set*
inductive *NF*
intros
App: $listall\ (\lambda t.\ t \in NF)\ ts \Longrightarrow Var\ x\ {}^{\circ\circ}\ ts \in NF$
Abs: $t \in NF \Longrightarrow Abs\ t \in NF$
monos *listall-def*

lemma *nat-eq-dec*: $\bigwedge n{::}nat.\ m = n \lor m \neq n$
$\langle proof \rangle$

lemma *nat-le-dec*: $\bigwedge n{::}nat.\ m < n \lor \neg\ (m < n)$
$\langle proof \rangle$

lemma *App-NF-D*: **assumes** *NF*: $Var\ n\ {}^{\circ\circ}\ ts \in NF$
shows $listall\ (\lambda t.\ t \in NF)\ ts$ $\langle proof \rangle$

11.2 Properties of *NF*

lemma *Var-NF*: $Var\ n \in NF$
$\langle proof \rangle$

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

lemma *subst-Var-NF*: $t \in NF \Longrightarrow (\bigwedge i\ j.\ t[Var\ i/j] \in NF)$
$\langle proof \rangle$

lemma *app-Var-NF*: $t \in NF \Longrightarrow \exists\, t'.\ t\ {}^{\circ}\ Var\ i \rightarrow_{\beta}{}^{*}\ t' \land t' \in NF$
$\langle proof \rangle$

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

lemma *lift-NF*: $t \in NF \Longrightarrow (\bigwedge i.\ lift\ t\ i \in NF)$
$\langle proof \rangle$

11.3 Main theorems

lemma *subst-type-NF*:

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

consts — A computationally relevant copy of $e \vdash t : T$
$rtyping :: ((nat \Rightarrow type) \times dB \times type)\ set$

syntax
$\text{-}rtyping :: (nat \Rightarrow type) \Rightarrow dB \Rightarrow type \Rightarrow bool \quad (\text{-} \mid\!-_R \text{-} : \text{-} [50, 50, 50]\ 50)$
syntax ($xsymbols$)
$\text{-}rtyping :: (nat \Rightarrow type) \Rightarrow dB \Rightarrow type \Rightarrow bool \quad (\text{-} \vdash_R \text{-} : \text{-} [50, 50, 50]\ 50)$
translations
$e \vdash_R t : T \rightleftharpoons (e, t, T) \in rtyping$

inductive $rtyping$
intros
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** T: $e \vdash_R t : T$
shows $\exists t'.\ t \rightarrow_\beta{}^* t' \wedge t' \in NF$ $\langle proof\rangle$

11.4 Extracting the program

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

extract *type-NF*

lemma *rtranclR-rtrancl-eq*: $((a, b) \in rtranclR\ r) = ((a, b) \in rtrancl\ (Collect\ r))$
$\langle proof\rangle$

lemma *NFR-imp-NF*: $(nf, t) \in NFR \Longrightarrow t \in NF$
$\langle proof\rangle$

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

$\bigwedge x.\ (x, t) \in NFR \Longrightarrow$
$e\langle i{:}U\rangle \vdash t : T \Longrightarrow$

```
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 arbitrary
       (λts xa xaa r xb xc xd xe H.
          case nat-eq-dec xa xe of
          Left ⇒ case ts of [] ⇒ (xd, H)
                 | a # list ⇒
                    var-app-typesE-P (xb⟨xe:x⟩) xa (a # list)
                     (λUs. case Us of [] ⇒ arbitrary
                           | T'' # Ts ⇒
                              let (x, y) =
                                    rev-induct-P list (λx H. ([], Var-NF 0))
                                     (λx xa H2 xc Ha.
                                         types-snocE-P xa x xc
                                          (λVs W.
 let (x, y) = H2 Vs (fst (fst (listall-snoc-P xa) Ha));
    (xa, ya) = snd (fst (listall-snoc-P xa) Ha) xb W xd xe H
 in (x @ [xa],
     NFT.App (map (λt. lift t 0) (x @ [xa])) 0
      (λxa. snd (listall-snoc-P (map (λt. lift t 0) x)) (App-NF-D y, lift-NF 0 ya) xa))))
                                     Ts (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)
                                    (foldl dB.App (dB.Var 0) (map (λt. lift t 0) x)) xb xc xa
                                    0 y ya
                              in (x, y))
          | Right ⇒
              var-app-typesE-P (xb⟨xe:x⟩) xa ts
               (λUs. let (x, y) =
                           rev-induct-P ts (λx H. ([], λx. Var-NF x))
                            (λx xa H2 xc Ha.
                                types-snocE-P xa x xc
                                 (λVs W. let (x, y) = H2 Vs (fst (fst (listall-snoc-P xa) Ha));
                                             (xa, ya) =
snd (fst (listall-snoc-P xa) Ha) xb W xd xe H
                                         in (x @ [xa],
                                             λxb.
 NFT.App (x @ [xa]) xb (snd (listall-snoc-P x) (App-NF-D (y 0), ya)))))
                            Us (listall-conj2-P-Q ts (λz. (xaa z, r z)))
                     in case nat-le-dec xe xa of
                        Left ⇒ (foldl (λu ua. dB.App u ua) (dB.Var (xa − Suc 0)) x,
                                y (xa − Suc 0))
                        | Right ⇒ (foldl (λu ua. dB.App u ua) (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 (λu. (xa⟨0:U⟩) u) V (lift xc 0) (Suc xd) (lift-NF 0 H)
                        in (dB.Abs x, NFT.Abs x y)
                  in (x, y)))
       H (λu. xb u) 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 arbitrary
   (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 arbitrary
     (λts xa xaa r.
         (foldl dB.App (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 arbitrary
        (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

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

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

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

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

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

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

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

11.5 Generating executable code

consts-code
  arbitrary :: *'a*   ((*error arbitrary*))
  arbitrary :: *'a* ⇒ *'b* ((*fn '- => error arbitrary*))

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

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*⟩

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