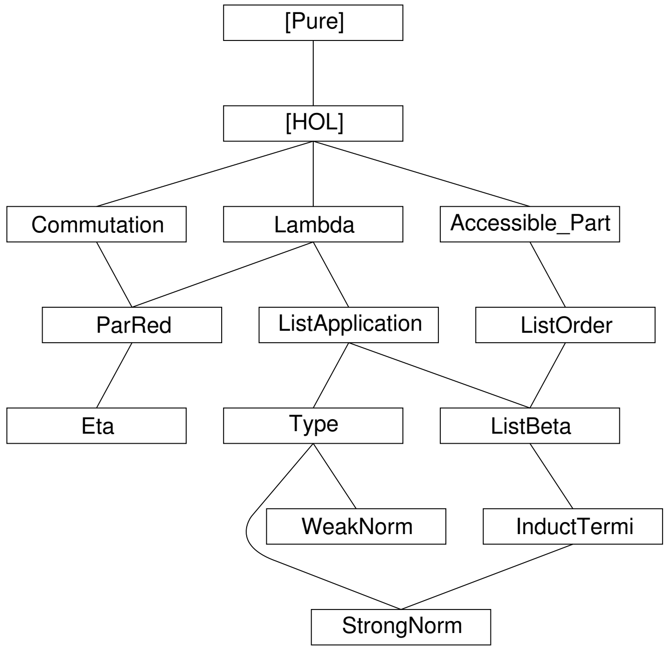
Fundamental Properties of Lambda-calculus

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* ° *t*) *k* = *lift s k* ° *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* ° *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*.

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* ° *t*) *k* = *liftn n s k* ° *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* ° *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

consts
  beta :: (*dB* × *dB*) *set*

syntax

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

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

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

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*!]:
$s \rightarrow_\beta{}^* s'$ ==> $Abs\ s \rightarrow_\beta{}^* Abs\ s'$
apply (*erule rtrancl-induct*)
apply (*blast intro*: *rtrancl-into-rtrancl*)+
done

lemma *rtrancl-beta-AppL*:
$s \rightarrow_\beta{}^* s'$ ==> $s \circ t \rightarrow_\beta{}^* s' \circ t$
apply (*erule rtrancl-induct*)
apply (*blast intro*: *rtrancl-into-rtrancl*)+
done

lemma *rtrancl-beta-AppR*:
$t \rightarrow_\beta{}^* t'$ ==> $s \circ t \rightarrow_\beta{}^* s \circ t'$
apply (*erule rtrancl-induct*)
apply (*blast intro*: *rtrancl-into-rtrancl*)+
done

lemma *rtrancl-beta-App* [*intro*]:
[| $s \rightarrow_\beta{}^* s'$; $t \rightarrow_\beta{}^* t'$ |] ==> $s \circ t \rightarrow_\beta{}^* s' \circ t'$
apply (*blast intro*!: *rtrancl-beta-AppL rtrancl-beta-AppR*
intro: *rtrancl-trans*)
done

1.4 Substitution-lemmas

```
lemma subst-eq [simp]: (Var k)[u/k] = u
  apply (simp add: subst-Var)
  done

lemma subst-gt [simp]: i < j ==> (Var j)[u/i] = Var (j - 1)
  apply (simp add: subst-Var)
  done

lemma subst-lt [simp]: j < i ==> (Var j)[u/i] = Var j
  apply (simp add: subst-Var)
  done

lemma lift-lift [rule-format]:
    ∀ i k. i < k + 1 --> lift (lift t i) (Suc k) = lift (lift t k) i
  apply (induct-tac t)
    apply auto
  done

lemma lift-subst [simp]:
    ∀ i j s. j < i + 1 --> lift (t[s/j]) i = (lift t (i + 1)) [lift s i / j]
  apply (induct-tac t)
    apply (simp-all add: diff-Suc subst-Var lift-lift split: nat.split)
  done

lemma lift-subst-lt:
    ∀ i j s. i < j + 1 --> lift (t[s/j]) i = (lift t i) [lift s i / j + 1]
  apply (induct-tac t)
    apply (simp-all add: subst-Var lift-lift)
  done

lemma subst-lift [simp]:
    ∀ k s. (lift t k)[s/k] = t
  apply (induct-tac t)
    apply simp-all
  done

lemma subst-subst [rule-format]:
    ∀ i j u v. i < j + 1 --> t[lift v i / Suc j][u[v/j]/i] = t[u/i][v/j]
  apply (induct-tac t)
    apply (simp-all
      add: diff-Suc subst-Var lift-lift [symmetric] lift-subst-lt
      split: nat.split)
  done
```

1.5 Equivalence proof for optimized substitution

```
lemma liftn-0 [simp]: ∀ k. liftn 0 t k = t
  apply (induct-tac t)
```

apply (*simp-all add*: *subst-Var*)
done

lemma *liftn-lift* [*simp*]:
$\forall k.\ liftn\ (Suc\ n)\ t\ k = lift\ (liftn\ n\ t\ k)\ k$
apply (*induct-tac t*)
apply (*simp-all add*: *subst-Var*)
done

lemma *substn-subst-n* [*simp*]:
$\forall n.\ substn\ t\ s\ n = t[liftn\ n\ s\ 0\ /\ n]$
apply (*induct-tac t*)
apply (*simp-all add*: *subst-Var*)
done

theorem *substn-subst-0*: $substn\ t\ s\ 0 = t[s/0]$
apply *simp*
done

1.6 Preservation theorems

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

theorem *subst-preserves-beta* [*simp*]:
$r \to_\beta s \implies (\bigwedge t\ i.\ r[t/i] \to_\beta s[t/i])$
apply (*induct set*: *beta*)
apply (*simp-all add*: *subst-subst* [*symmetric*])
done

theorem *subst-preserves-beta'*: $r \to_\beta^* s \implies r[t/i] \to_\beta^* s[t/i]$
apply (*erule rtrancl.induct*)
apply (*rule rtrancl-refl*)
apply (*erule rtrancl-into-rtrancl*)
apply (*erule subst-preserves-beta*)
done

theorem *lift-preserves-beta* [*simp*]:
$r \to_\beta s \implies (\bigwedge i.\ lift\ r\ i \to_\beta lift\ s\ i)$
by (*induct set*: *beta*) *auto*

theorem *lift-preserves-beta'*: $r \to_\beta^* s \implies lift\ r\ i \to_\beta^* lift\ s\ i$
apply (*erule rtrancl.induct*)
apply (*rule rtrancl-refl*)
apply (*erule rtrancl-into-rtrancl*)
apply (*erule lift-preserves-beta*)
done

theorem *subst-preserves-beta2* [*simp*]:
$\bigwedge r\ s\ i.\ r \to_\beta s \implies t[r/i] \to_\beta^* t[s/i]$

```
  apply (induct t)
    apply (simp add: subst-Var r-into-rtrancl)
   apply (simp add: rtrancl-beta-App)
  apply (simp add: rtrancl-beta-Abs)
  done
```

theorem *subst-preserves-beta2'*: $r \to_\beta{}^* \ s \implies t[r/i] \to_\beta{}^* \ t[s/i]$

```
  apply (erule rtrancl.induct)
  apply (rule rtrancl-refl)
  apply (erule rtrancl-trans)
  apply (erule subst-preserves-beta2)
  done
```

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 ==
$\forall x\ y.\ (x, y) \in R \longrightarrow (\forall z.\ (x, z) \in S \longrightarrow (\exists u.\ (y, u) \in T \land (z, u) \in U))$

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 ==
$\forall x\ y.\ (x, y) \in (R \cup R\hat{}{-1})\hat{}{*} \longrightarrow (\exists z.\ (x, z) \in R\hat{}{*} \land (y, z) \in R\hat{}{*})$

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* ==> *square S R U T*

```
  apply (unfold square-def)
  apply blast
```

done

lemma *square-subset*:
  [| *square* R S T U; $T \subseteq T'$ |] ==> *square* R S T' U
apply (*unfold square-def*)
apply *blast*
done

lemma *square-reflcl*:
  [| *square* R S T (Rˆ=); $S \subseteq T$ |] ==> *square* (Rˆ=) S T (Rˆ=)
apply (*unfold square-def*)
apply *blast*
done

lemma *square-rtrancl*:
  square R S S T ==> *square* (Rˆ$*$) S S (Tˆ$*$)
apply (*unfold square-def*)
apply (*intro strip*)
apply (*erule rtrancl-induct*)
 apply *blast*
apply (*blast intro*: *rtrancl-into-rtrancl*)
done

lemma *square-rtrancl-reflcl-commute*:
  square R S (Sˆ$*$) (Rˆ=) ==> *commute* (Rˆ$*$) (Sˆ$*$)
apply (*unfold commute-def*)
apply (*fastsimp dest*: *square-reflcl square-sym* [*THEN square-rtrancl*]
  elim: *r-into-rtrancl*)
done

commute

lemma *commute-sym*: *commute* R S ==> *commute* S R
apply (*unfold commute-def*)
apply (*blast intro*: *square-sym*)
done

lemma *commute-rtrancl*: *commute* R S ==> *commute* (Rˆ$*$) (Sˆ$*$)
apply (*unfold commute-def*)
apply (*blast intro*: *square-rtrancl square-sym*)
done

lemma *commute-Un*:
  [| *commute* R T; *commute* S T |] ==> *commute* ($R \cup S$) T
apply (*unfold commute-def square-def*)
apply *blast*
done

diamond, confluence, and union

lemma *diamond-Un*:
  [| *diamond R*; *diamond S*; *commute R S* |] ==> *diamond* (*R* ∪ *S*)
 apply (*unfold diamond-def*)
 apply (*assumption* | *rule commute-Un commute-sym*)+
 done

lemma *diamond-confluent*: *diamond R* ==> *confluent R*
 apply (*unfold diamond-def*)
 apply (*erule commute-rtrancl*)
 done

lemma *square-reflcl-confluent*:
  square R R (*R*ˆ=) (*R*ˆ=) ==> *confluent R*
 apply (*unfold diamond-def*)
 apply (*fast intro*: *square-rtrancl-reflcl-commute r-into-rtrancl*
  elim: *square-subset*)
 done

lemma *confluent-Un*:
  [| *confluent R*; *confluent S*; *commute* (*R*ˆ∗) (*S*ˆ∗) |] ==> *confluent* (*R* ∪ *S*)
 apply (*rule rtrancl-Un-rtrancl* [*THEN subst*])
 apply (*blast dest*: *diamond-Un intro*: *diamond-confluent*)
 done

lemma *diamond-to-confluence*:
  [| *diamond R*; *T* ⊆ *R*; *R* ⊆ *T*ˆ∗ |] ==> *confluent T*
 apply (*force intro*: *diamond-confluent*
  dest: *rtrancl-subset* [*symmetric*])
 done

2.3 Church-Rosser

lemma *Church-Rosser-confluent*: *Church-Rosser R* = *confluent R*
 apply (*unfold square-def commute-def diamond-def Church-Rosser-def*)
 apply (*tactic* ⟪ *safe-tac HOL-cs* ⟫)
  apply (*tactic* ⟪
   blast-tac (*HOL-cs addIs*
    [*Un-upper2 RS rtrancl-mono RS subsetD RS rtrancl-trans*,
    rtrancl-converseI, *converseI*, *Un-upper1 RS rtrancl-mono RS subsetD*]) *1* ⟫)
 apply (*erule rtrancl-induct*)
  apply *blast*
 apply (*blast del*: *rtrancl-refl intro*: *rtrancl-trans*)
 done

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 $\bigwedge b\ c.\ (a,\ b) \in R^* \Longrightarrow (a,\ c) \in R^* \Longrightarrow$
    $\exists\, d.\ (b,\ d) \in R^* \wedge (c,\ d) \in R^*$
  using *wf*
proof *induct*
  case (*less x b c*)
  have *xc*: $(x,\ c) \in R^*$ **.**
  have *xb*: $(x,\ b) \in R^*$ **. thus** *?case*
  proof (*rule converse-rtranclE*)
    assume $x = b$
    with *xc* **have** $(b,\ c) \in R^*$ **by** *simp*
    thus *?thesis* **by** *iprover*
  next
    fix *y*
    assume *xy*: $(x,\ y) \in R$
    assume *yb*: $(y,\ b) \in R^*$
    from *xc* **show** *?thesis*
    proof (*rule converse-rtranclE*)
      assume $x = c$
      with *xb* **have** $(c,\ b) \in R^*$ **by** *simp*
      thus *?thesis* **by** *iprover*
    next
      fix y'
      assume $y'c$: $(y',\ c) \in R^*$
      assume xy': $(x,\ y') \in R$
      with *xy* **have** $\exists\, u.\ (y,\ u) \in R^* \wedge (y',\ u) \in R^*$ **by** (*rule lc*)
      then obtain *u* **where** *yu*: $(y,\ u) \in R^*$ **and** $y'u$: $(y',\ u) \in R^*$ **by** *iprover*
      from *xy* **have** $(y,\ x) \in R^{-1}$ **..**
      from *this* **and** *yb yu* **have** $\exists\, d.\ (b,\ d) \in R^* \wedge (u,\ d) \in R^*$ **by** (*rule less*)
      then obtain *v* **where** *bv*: $(b,\ v) \in R^*$ **and** *uv*: $(u,\ v) \in R^*$ **by** *iprover*
      from xy' **have** $(y',\ x) \in R^{-1}$ **..**
      moreover from $y'u$ **and** *uv* **have** $(y',\ v) \in R^*$ **by** (*rule rtrancl-trans*)
      moreover note $y'c$
      ultimately have $\exists\, d.\ (v,\ d) \in R^* \wedge (c,\ d) \in R^*$ **by** (*rule less*)
      then obtain *w* **where** *vw*: $(v,\ w) \in R^*$ **and** *cw*: $(c,\ w) \in R^*$ **by** *iprover*
      from *bv vw* **have** $(b,\ w) \in R^*$ **by** (*rule rtrancl-trans*)
      with *cw* **show** *?thesis* **by** *iprover*
    qed
  qed
qed

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: ⋀a b c. (a, b) ∈ R ⟹ (a, c) ∈ R ⟹
    ∃ d. (b, d) ∈ R* ∧ (c, d) ∈ R*
  shows ⋀b c. (a, b) ∈ R* ⟹ (a, c) ∈ R* ⟹
             ∃ d. (b, d) ∈ R* ∧ (c, d) ∈ R*
using wf
proof induct
  case (less x b c)
  have IH: ⋀y b c. ⟦(y,x) ∈ R⁻¹; (y,b) ∈ R*; (y,c) ∈ R*⟧
                  ⟹ ∃ d. (b,d) ∈ R* ∧ (c,d) ∈ R* by(rule less)
  have xc: (x, c) ∈ R* .
  have xb: (x, b) ∈ R* .
  thus ?case
  proof (rule converse-rtranclE)
    assume x = b
    with xc have (b, c) ∈ R* by simp
    thus ?thesis by iprover
  next
    fix y
    assume xy: (x, y) ∈ R
    assume yb: (y, b) ∈ R*
    from xc show ?thesis
    proof (rule converse-rtranclE)
      assume x = c
      with xb have (c, b) ∈ R* by simp
      thus ?thesis by iprover
    next
      fix y'
      assume y'c: (y', c) ∈ R*
      assume xy': (x, y') ∈ R
      with xy obtain u where u: (y, u) ∈ R* (y', u) ∈ R*
        by (blast dest:lc)
      from yb u y'c show ?thesis
        by(blast del: rtrancl-refl
              intro:rtrancl-trans
              dest:IH[OF xy[symmetric]] IH[OF xy'[symmetric]])
    qed
  qed
qed

end
```

3 Parallel reduction and a complete developments

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

3.1 Parallel reduction

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

syntax
  par-beta :: [*dB*, *dB*] => *bool* (**infixl** => *50*)
translations
  s => *t* == (*s*, *t*) ∈ *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* ° *t* => *s*′ ° *t*′
    beta [*simp*, *intro*!]: [| *s* => *s*′; *t* => *t*′ |] ==> (*Abs s*) ° *t* => *s*′[*t*′/*0*]

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

3.2 Inclusions

beta ⊆ *par-beta* ⊆ *beta*ˆ∗

lemma *par-beta-varL* [*simp*]:
    (*Var n* => *t*) = (*t* = *Var n*)
  apply *blast*
  done

lemma *par-beta-refl* [*simp*]: *t* => *t*
  apply (*induct-tac t*)
    apply *simp-all*
  done

lemma *beta-subset-par-beta*: *beta* <= *par-beta*
  apply (*rule subsetI*)
  apply *clarify*
  apply (*erule beta.induct*)
     apply (*blast intro*!: *par-beta-refl*)+
  done

lemma *par-beta-subset-beta*: *par-beta* <= *beta*ˆ∗
  apply (*rule subsetI*)
  apply *clarify*
  apply (*erule par-beta.induct*)
     apply *blast*

```
  apply (blast del: rtrancl-refl intro: rtrancl-into-rtrancl)+
    — rtrancl-refl complicates the proof by increasing the branching factor
  done
```

3.3 Misc properties of par-beta

```
lemma par-beta-lift [rule-format, simp]:
    ∀ t' n. t => t' --> lift t n => lift t' n
  apply (induct-tac t)
    apply fastsimp+
  done

lemma par-beta-subst [rule-format]:
    ∀ s s' t' n. s => s' --> t => t' --> t[s/n] => t'[s'/n]
  apply (induct-tac t)
    apply (simp add: subst-Var)
   apply (intro strip)
   apply (erule par-beta-cases)
    apply simp
   apply (simp add: subst-subst [symmetric])
   apply (fastsimp intro!: par-beta-lift)
  apply fastsimp
  done
```

3.4 Confluence (directly)

```
lemma diamond-par-beta: diamond par-beta
  apply (unfold diamond-def commute-def square-def)
  apply (rule impI [THEN allI [THEN allI]])
  apply (erule par-beta.induct)
     apply (blast intro!: par-beta-subst)+
  done
```

3.5 Complete developments

```
consts
  cd :: dB => dB
recdef cd measure size
  cd (Var n) = Var n
  cd (Var n ° t) = Var n ° cd t
  cd ((s1 ° s2) ° t) = cd (s1 ° s2) ° cd t
  cd (Abs u ° t) = (cd u)[cd t/0]
  cd (Abs s) = Abs (cd s)

lemma par-beta-cd [rule-format]:
    ∀ t. s => t --> t => cd s
  apply (induct-tac s rule: cd.induct)
      apply auto
  apply (fast intro!: par-beta-subst)
  done
```

3.6 Confluence (via complete developments)

lemma *diamond-par-beta2*: *diamond par-beta*
apply (*unfold diamond-def commute-def square-def*)
apply (*blast intro*: *par-beta-cd*)
done

theorem *beta-confluent*: *confluent beta*
apply (*rule diamond-par-beta2 diamond-to-confluence*
par-beta-subset-beta beta-subset-par-beta)+
done

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* ∨ *free t i*)
free (*Abs s*) *i* = *free s* (*i* + *1*)

consts
eta :: (*dB* × *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*) ∈ *eta*
s −*e*>> *t* == (*s*, *t*) ∈ *eta*ˆ∗
s −*e*>= *t* == (*s*, *t*) ∈ *eta*ˆ=

inductive *eta*
intros
eta [*simp*, *intro*]: ¬ *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]:
    ∀ i t u. ¬ free s i −−> s[t/i] = s[u/i]
  apply (induct-tac s)
    apply (simp-all add: subst-Var)
  done

lemma free-lift [simp]:
    ∀ i k. free (lift t k) i =
      (i < k ∧ free t i ∨ k < i ∧ free t (i − 1))
  apply (induct-tac t)
    apply (auto cong: conj-cong)
  apply arith
  done

lemma free-subst [simp]:
    ∀ i k t. free (s[t/k]) i =
      (free s k ∧ free t i ∨ free s (if i < k then i else i + 1))
  apply (induct-tac s)
    prefer 2
    apply simp
    apply blast
   prefer 2
   apply simp
  apply (simp add: diff-Suc subst-Var split: nat.split)
  done

lemma free-eta [rule-format]:
    s −e> t ==> ∀ i. free t i = free s i
  apply (erule eta.induct)
     apply (simp-all cong: conj-cong)
  done

lemma not-free-eta:
    [| s −e> t; ¬ free s i |] ==> ¬ free t i
  apply (simp add: free-eta)
  done

lemma eta-subst [rule-format, simp]:
    s −e> t ==> ∀ u i. s[u/i] −e> t[u/i]
  apply (erule eta.induct)
  apply (simp-all add: subst-subst [symmetric])
  done
```

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

by (*induct s*) *simp-all*

4.3 Confluence of eta

lemma *square-eta*: *square eta eta* (*eta*ˆ=) (*eta*ˆ=)
  apply (*unfold square-def id-def*)
  apply (*rule impI* [*THEN allI* [*THEN allI*]])
  apply *simp*
  apply (*erule eta.induct*)
    apply (*slowsimp intro*: *subst-not-free eta-subst free-eta* [*THEN iffD1*])
   apply *safe*
     prefer *5*
     apply (*blast intro*!: *eta-subst intro*: *free-eta* [*THEN iffD1*])
    apply *blast*+
  done

theorem *eta-confluent*: *confluent eta*
  apply (*rule square-eta* [*THEN square-reflcl-confluent*])
  done

4.4 Congruence rules for eta*

lemma *rtrancl-eta-Abs*: $s\ -e{>}{>}\ s' ==> Abs\ s\ -e{>}{>}\ Abs\ s'$
  apply (*erule rtrancl-induct*)
   apply (*blast intro*: *rtrancl-refl rtrancl-into-rtrancl*)+
  done

lemma *rtrancl-eta-AppL*: $s\ -e{>}{>}\ s' ==> s \circ t\ -e{>}{>}\ s' \circ t$
  apply (*erule rtrancl-induct*)
   apply (*blast intro*: *rtrancl-refl rtrancl-into-rtrancl*)+
  done

lemma *rtrancl-eta-AppR*: $t\ -e{>}{>}\ t' ==> s \circ t\ -e{>}{>}\ s \circ t'$
  apply (*erule rtrancl-induct*)
   apply (*blast intro*: *rtrancl-refl rtrancl-into-rtrancl*)+
  done

lemma *rtrancl-eta-App*:
    $[\!|\ s\ -e{>}{>}\ s';\ t\ -e{>}{>}\ t'\ |\!] ==> s \circ t\ -e{>}{>}\ s' \circ t'$
  apply (*blast intro*!: *rtrancl-eta-AppL rtrancl-eta-AppR intro*: *rtrancl-trans*)
  done

4.5 Commutation of beta and eta

lemma *free-beta* [*rule-format*]:
    $s\ -{>}\ t ==> \forall i.\ free\ t\ i\ --{>}\ free\ s\ i$
  apply (*erule beta.induct*)
    apply *simp-all*
  done

lemma *beta-subst* [*rule-format*, *intro*]:
  s −> *t* ==> ∀ *u i*. *s*[*u*/*i*] −> *t*[*u*/*i*]
 apply (*erule beta.induct*)
   apply (*simp-all add*: *subst-subst* [*symmetric*])
 done

lemma *subst-Var-Suc* [*simp*]: ∀ *i*. *t*[*Var i*/*i*] = *t*[*Var*(*i*)/*i* + *1*]
 apply (*induct-tac t*)
 apply (*auto elim*!: *linorder-neqE simp*: *subst-Var*)
 done

lemma *eta-lift* [*rule-format*, *simp*]:
  s −*e*> *t* ==> ∀ *i*. *lift s i* −*e*> *lift t i*
 apply (*erule eta.induct*)
   apply *simp-all*
 done

lemma *rtrancl-eta-subst* [*rule-format*]:
  ∀ *s t i*. *s* −*e*> *t* −−> *u*[*s*/*i*] −*e*>> *u*[*t*/*i*]
 apply (*induct-tac u*)
   apply (*simp-all add*: *subst-Var*)
   apply (*blast*)
  apply (*blast intro*: *rtrancl-eta-App*)
 apply (*blast intro*!: *rtrancl-eta-Abs eta-lift*)
 done

lemma *square-beta-eta*: *square beta eta* (*eta*ˆ∗) (*beta*ˆ=)
 apply (*unfold square-def*)
 apply (*rule impI* [*THEN allI* [*THEN allI*]])
 apply (*erule beta.induct*)
   apply (*slowsimp intro*: *rtrancl-eta-subst eta-subst*)
   apply (*blast intro*: *rtrancl-eta-AppL*)
  apply (*blast intro*: *rtrancl-eta-AppR*)
 apply *simp*
 apply (*slowsimp intro*: *rtrancl-eta-Abs free-beta*
  iff del: *dB.distinct simp*: *dB.distinct*)
 done

lemma *confluent-beta-eta*: *confluent* (*beta* ∪ *eta*)
 apply (*assumption* |
  rule square-rtrancl-reflcl-commute confluent-Un
   beta-confluent eta-confluent square-beta-eta)+
 done

4.6 Implicit definition of eta

Abs (*lift s 0* ° *Var 0*) −*e*> *s*

lemma *not-free-iff-lifted* [*rule-format*]:
  ∀ *i*. (¬ *free s i*) = (∃ *t*. *s* = *lift t i*)

```
apply (induct-tac s)
  apply simp
  apply clarify
  apply (rule iffI)
   apply (erule linorder-neqE)
    apply (rule-tac x = Var nat in exI)
    apply simp
   apply (rule-tac x = Var (nat - 1) in exI)
   apply simp
  apply clarify
  apply (rule notE)
   prefer 2
   apply assumption
  apply (erule thin-rl)
  apply (case-tac t)
    apply simp
   apply simp
  apply simp
 apply simp
 apply (erule thin-rl)
 apply (erule thin-rl)
 apply (rule allI)
 apply (rule iffI)
  apply (elim conjE exE)
  apply (rename-tac u1 u2)
  apply (rule-tac x = u1 ° u2 in exI)
  apply simp
 apply (erule exE)
 apply (erule rev-mp)
 apply (case-tac t)
   apply simp
  apply simp
  apply blast
 apply simp
apply simp
apply (erule thin-rl)
apply (rule allI)
apply (rule iffI)
 apply (erule exE)
 apply (rule-tac x = Abs t in exI)
 apply simp
apply (erule exE)
apply (erule rev-mp)
apply (case-tac t)
  apply simp
 apply simp
apply simp
apply blast
done
```

theorem *explicit-is-implicit*:
  $(\forall s\ u.\ (\neg\ free\ s\ 0)\ {-}{-}{>}\ R\ (Abs\ (s\ \circ\ Var\ 0))\ (s[u/0])) =$
    $(\forall s.\ R\ (Abs\ (lift\ s\ 0\ \circ\ Var\ 0))\ s)$
  apply (*auto simp add*: *not-free-iff-lifted*)
  done

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*]: $\neg\ free\ s\ 0 \Longrightarrow s \Rightarrow_\eta s' \Longrightarrow Abs\ (s\ \circ\ Var\ 0) \Rightarrow_\eta s'[dummy/0]$
  app [*simp*, *intro*]: $s \Rightarrow_\eta s' \Longrightarrow t \Rightarrow_\eta t' \Longrightarrow s\ \circ\ t \Rightarrow_\eta s'\ \circ\ t'$
  abs [*simp*, *intro*]: $s \Rightarrow_\eta t \Longrightarrow Abs\ s \Rightarrow_\eta Abs\ t$

lemma *free-par-eta* [*simp*]: **assumes** *eta*: $s \Rightarrow_\eta t$
  shows $\bigwedge i.\ free\ t\ i = free\ s\ i$ **using** *eta*
  by *induct* (*simp-all cong*: *conj-cong*)

lemma *par-eta-refl* [*simp*]: $t \Rightarrow_\eta t$
  by (*induct t*) *simp-all*

lemma *par-eta-lift* [*simp*]:
  assumes *eta*: $s \Rightarrow_\eta t$
  shows $\bigwedge i.\ lift\ s\ i \Rightarrow_\eta lift\ t\ i$ **using** *eta*
  by *induct simp-all*

lemma *par-eta-subst* [*simp*]:
  assumes *eta*: $s \Rightarrow_\eta t$
  shows $\bigwedge u\ u'\ i.\ u \Rightarrow_\eta u' \Longrightarrow s[u/i] \Rightarrow_\eta t[u'/i]$ **using** *eta*
  by *induct* (*simp-all add*: *subst-subst* [*symmetric*] *subst-Var*)

theorem *eta-subset-par-eta*: *eta* ⊆ *par-eta*
  apply (*rule subsetI*)
  apply *clarify*
  apply (*erule eta.induct*)
  apply (*blast intro*!: *par-eta-refl*)+

done

theorem *par-eta-subset-eta*: *par-eta* $\subseteq$ *eta**
apply (*rule subsetI*)
apply *clarify*
apply (*erule par-eta.induct*)
apply *blast*
apply (*rule rtrancl-into-rtrancl*)
apply (*rule rtrancl-eta-Abs*)
apply (*rule rtrancl-eta-AppL*)
apply *assumption*
apply (*rule eta.eta*)
apply *simp*
apply (*rule rtrancl-eta-App*)
apply *assumption+*
apply (*rule rtrancl-eta-Abs*)
apply *assumption*
done

4.8 n-ary eta-expansion

consts *eta-expand* :: *nat* $\Rightarrow$ *dB* $\Rightarrow$ *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* $\circ$ *Var 0*)

lemma *eta-expand-Suc*$'$:
$\bigwedge$*t*. *eta-expand* (*Suc n*) *t* = *eta-expand n* (*Abs* (*lift t 0* $\circ$ *Var 0*))
by (*induct n*) *simp-all*

theorem *lift-eta-expand*: *lift* (*eta-expand k t*) *i* = *eta-expand k* (*lift t i*)
by (*induct k*) (*simp-all add*: *lift-lift*)

theorem *eta-expand-beta*:
assumes *u*: *u* => *u*$'$
shows $\bigwedge$*t t*$'$. *t* => *t*$'$ $\Longrightarrow$ *eta-expand k* (*Abs t*) $\circ$ *u* => *t*$'$[*u*$'$/*0*]
proof (*induct k*)
case *0* **with** *u* **show** *?case* **by** *simp*
next
case (*Suc k*)
hence *Abs* (*lift t* (*Suc 0*)) $\circ$ *Var 0* => *lift t*$'$ (*Suc 0*)[*Var 0*/*0*]
by (*blast intro*: *par-beta-lift*)
with *Suc* **show** *?case* **by** (*simp del*: *eta-expand-Suc add*: *eta-expand-Suc*$'$)
qed

theorem *eta-expand-red*:
assumes *t*: *t* => *t*$'$
shows *eta-expand k t* => *eta-expand k t*$'$
by (*induct k*) (*simp-all add*: *t*)

theorem *eta-expand-eta*: $\bigwedge t\ t'.\ t \Rightarrow_\eta t' \Longrightarrow$ *eta-expand k t* $\Rightarrow_\eta t'$
proof (*induct k*)
  case *0*
  show *?case* **by** *simp*
next
  case (*Suc k*)
  have *Abs* (*lift* (*eta-expand k t*) *0* $\circ$ *Var 0*) $\Rightarrow_\eta$ *lift* t' *0*[*arbitrary/0*]
    by (*fastsimp intro*: *par-eta-lift Suc*)
  thus *?case* **by** *simp*
qed

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* $(u1 \circ u2) \wedge u1 \Rightarrow_\eta u1' \wedge u2 \Rightarrow_\eta u2'$ **using** *eta*
proof *induct*
  case (*app s s' t t'*)
  have $s \circ t =$ *eta-expand 0* $(s \circ t)$ **by** *simp*
  with *app* **show** *?case* **by** *blast*
next
  case (*eta dummy s s'*)
  then obtain $u1''\ u2''$ **where** s': $s' = u1'' \circ u2''$
   by (*cases s'*) (*auto simp add*: *subst-Var free-par-eta* [*symmetric*] *split*: *split-if-asm*)
  then have $\exists\, u1\ u2\ k.\ s =$ *eta-expand k* $(u1 \circ u2) \wedge u1 \Rightarrow_\eta u1'' \wedge u2 \Rightarrow_\eta u2''$
by (*rule eta*)
  then obtain *u1 u2 k* **where** *s*: $s =$ *eta-expand k* $(u1 \circ u2)$
    and *u1*: $u1 \Rightarrow_\eta u1''$ **and** *u2*: $u2 \Rightarrow_\eta u2''$ **by** *iprover*
  from *u1 u2 eta s'* **have** $\neg$ *free u1 0* **and** $\neg$ *free u2 0*
    by (*simp-all del*: *free-par-eta add*: *free-par-eta* [*symmetric*])
  with *s* **have** *Abs* $(s \circ$ *Var 0*$) =$ *eta-expand* (*Suc k*) (*u1*[*dummy/0*] $\circ$ *u2*[*dummy/0*])
    by (*simp del*: *lift-subst add*: *lift-subst-dummy lift-eta-expand*)
  moreover from *u1 par-eta-refl* **have** *u1*[*dummy/0*] $\Rightarrow_\eta$ $u1''$[*dummy/0*]
    by (*rule par-eta-subst*)
  moreover from *u2 par-eta-refl* **have** *u2*[*dummy/0*] $\Rightarrow_\eta$ $u2''$[*dummy/0*]
    by (*rule par-eta-subst*)
  ultimately show *?case* **using** *eta s'*
    by (*simp only*: *subst.simps dB.simps*) *blast*
next
  case *var* **thus** *?case* **by** *simp*
next
  case *abs* **thus** *?case* **by** *simp*
qed

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*) $\wedge\ u \Rightarrow_\eta u'$ **using** *eta*
proof *induct*

case $(abs\ s\ t)$
have $Abs\ s = eta\text{-}expand\ 0\ (Abs\ s)$ **by** *simp*
with *abs* **show** *?case* **by** *blast*
next
case $(eta\ dummy\ s\ s')$
then obtain u'' **where** s': $s' = Abs\ u''$
by $(cases\ s')$ $(auto\ simp\ add$: *subst-Var free-par-eta* [*symmetric*] *split*: *split-if-asm*)
then have $\exists\, u\ k.\ s = eta\text{-}expand\ k\ (Abs\ u) \wedge u \Rightarrow_\eta u''$ **by** (*rule eta*)
then obtain $u\ k$ **where** s: $s = eta\text{-}expand\ k\ (Abs\ u)$ **and** u: $u \Rightarrow_\eta u''$ **by** *iprover*
from $eta\ u\ s'$ **have** $\neg\ free\ u\ (Suc\ 0)$
by (*simp del*: *free-par-eta add*: *free-par-eta* [*symmetric*])
with s **have** $Abs\ (s \circ Var\ 0) = eta\text{-}expand\ (Suc\ k)\ (Abs\ (u[lift\ dummy\ 0/Suc\ 0]))$
by (*simp del*: *lift-subst add*: *lift-eta-expand lift-subst-dummy*)
moreover from *u par-eta-refl* **have** $u[lift\ dummy\ 0/Suc\ 0] \Rightarrow_\eta u''[lift\ dummy\ 0/Suc\ 0]$
by (*rule par-eta-subst*)
ultimately show *?case* **using** $eta\ s'$ **by** *fastsimp*
next
case *var* **thus** *?case* **by** *simp*
next
case *app* **thus** *?case* **by** *simp*
qed

4.10 Eta-postponement theorem

Based on a proof by Masako Takahashi [2].

theorem *par-eta-beta*: $\bigwedge s\ u.\ s \Rightarrow_\eta t \Longrightarrow t \Rightarrow u \Longrightarrow \exists\, t'.\ s \Rightarrow t' \wedge t' \Rightarrow_\eta u$
proof (*induct t rule*: *measure-induct* [*of size*, *rule-format*])
case $(1\ t)$
from $1(3)$
show *?case*
proof *cases*
case $(var\ n)$
with *1* **show** *?thesis*
by (*auto intro*: *par-beta-refl*)
next
case $(abs\ r'\ r'')$
with *1* **have** $s \Rightarrow_\eta Abs\ r'$ **by** *simp*
then obtain $r\ k$ **where** s: $s = eta\text{-}expand\ k\ (Abs\ r)$ **and** rr: $r \Rightarrow_\eta r'$
by (*blast dest*: *par-eta-elim-abs*)
from *abs* **have** $size\ r' < size\ t$ **by** *simp*
with $abs(2)$ *rr* **obtain** t' **where** rt: $r \Rightarrow t'$ **and** t': $t' \Rightarrow_\eta r''$
by (*blast dest*: *1*)
with *s abs* **show** *?thesis*
by (*auto intro*: *eta-expand-red eta-expand-eta*)
next
case $(app\ q'\ q''\ r'\ r'')$
with *1* **have** $s \Rightarrow_\eta q' \circ r'$ **by** *simp*

then **obtain** $q\ r\ k$ **where** s: $s = \mathit{eta\text{-}expand}\ k\ (q \circ r)$
and qq: $q \Rightarrow_\eta q'$ **and** rr: $r \Rightarrow_\eta r'$
by (*blast dest*: *par-eta-elim-app* [*OF - refl*])
from *app* **have** *size* q' < *size* t **and** *size* r' < *size* t **by** *simp-all*
with *app*(*2*,*3*) *qq rr* **obtain** $t'\ t''$ **where** q => t' **and**
$t' \Rightarrow_\eta q''$ **and** r => t'' **and** $t'' \Rightarrow_\eta r''$
by (*blast dest*: *1*)
with *s app* **show** *?thesis*
by (*fastsimp intro*: *eta-expand-red eta-expand-eta*)
next
case (*beta* $q'\ q''\ r'\ r''$)
with *1* **have** $s \Rightarrow_\eta$ *Abs* $q' \circ r'$ **by** *simp*
then obtain $q\ r\ k\ k'$ **where** s: s = *eta-expand* k (*eta-expand* k' (*Abs* q) $\circ$ r)
and qq: $q \Rightarrow_\eta q'$ **and** rr: $r \Rightarrow_\eta r'$
by (*blast dest*: *par-eta-elim-app par-eta-elim-abs*)
from *beta* **have** *size* q' < *size* t **and** *size* r' < *size* t **by** *simp-all*
with *beta*(*2*,*3*) *qq rr* **obtain** $t'\ t''$ **where** q => t' **and**
$t' \Rightarrow_\eta q''$ **and** r => t'' **and** $t'' \Rightarrow_\eta r''$
by (*blast dest*: *1*)
with *s beta* **show** *?thesis*
by (*auto intro*: *eta-expand-red eta-expand-beta eta-expand-eta par-eta-subst*)
qed
qed

theorem *eta-postponement'*: **assumes** *eta*: s −*e*>> t
shows $\bigwedge u.\ t$ => $u \Longrightarrow \exists t'.\ s$ => $t' \wedge t'$ −*e*>> u
using *eta* [*simplified rtrancl-subset*
[*OF eta-subset-par-eta par-eta-subset-eta*, *symmetric*]]
proof *induct*
case *1*
thus *?case* **by** *blast*
next
case (*2* $s'\ s''\ s'''$)
from *2* **obtain** t' **where** s': s' => t' **and** t': $t' \Rightarrow_\eta s'''$
by (*auto dest*: *par-eta-beta*)
from s' **obtain** t'' **where** s: s => t'' **and** t'': t'' −*e*>> t'
by (*blast dest*: *2*)
from *par-eta-subset-eta* t' **have** t' −*e*>> s''' **..**
with t'' **have** t'' −*e*>> s''' **by** (*rule rtrancl-trans*)
with s **show** *?case* **by** *iprover*
qed

theorem *eta-postponement*:
assumes *st*: $(s,\ t) \in (\mathit{beta} \cup \mathit{eta})^*$
shows $(s,\ t) \in \mathit{eta}^*\ O\ \mathit{beta}^*$ **using** *st*
proof *induct*
case *1*
show *?case* **by** *blast*
next

case ($2\ s'\ s''$)
from $2(3)$ **obtain** t' **where** s: $s \rightarrow_\beta{}^* t'$ **and** t': $t' -e{>}{>}\ s'$ **by** *blast*
from $2(2)$ **show** *?case*
proof
assume $s' -{>}\ s''$
with *beta-subset-par-beta* **have** $s' ={>}\ s''$ **..**
with t' **obtain** t'' **where** *st*: $t' ={>}\ t''$ **and** *tu*: $t'' -e{>}{>}\ s''$
by (*auto dest*: *eta-postponement'*)
from *par-beta-subset-beta st* **have** $t' \rightarrow_\beta{}^* t''$ **..**
with s **have** $s \rightarrow_\beta{}^* t''$ **by** (*rule rtrancl-trans*)
thus *?thesis* **using** *tu* **..**
next
assume $s' -e{>}\ s''$
with t' **have** $t' -e{>}{>}\ s''$ **..**
with s **show** *?thesis* **..**
qed
qed

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)
apply (*induct-tac ts rule*: *rev-induct*)
apply *auto*
done

lemma *Var-eq-apps-conv* [*iff*]:
$\bigwedge s.$ (*Var* m = s °° ss) = (*Var* $m = s \wedge ss = []$)
apply (*induct ss*)
apply *auto*
done

lemma *Var-apps-eq-Var-apps-conv* [*iff*]:
$\bigwedge ss.$ (*Var* m °° rs = *Var* n °° ss) = ($m = n \wedge rs = ss$)
apply (*induct rs rule*: *rev-induct*)
apply *simp*
apply *blast*
apply (*induct-tac ss rule*: *rev-induct*)
apply *auto*
done

lemma *App-eq-foldl-conv*:
$(r \circ s = t \circ\circ ts) =$
$(if\ ts = []\ then\ r \circ s = t$
$else\ (\exists\, ss.\ ts = ss\ @\ [s] \wedge r = t \circ\circ ss))$
apply (*rule-tac xs = ts* **in** *rev-exhaust*)
apply *auto*
done

lemma *Abs-eq-apps-conv* [*iff*]:
$(Abs\ r = s \circ\circ ss) = (Abs\ r = s \wedge ss = [])$
apply (*induct-tac ss rule*: *rev-induct*)
apply *auto*
done

lemma *apps-eq-Abs-conv* [*iff*]: $(s \circ\circ ss = Abs\ r) = (s = Abs\ r \wedge ss = [])$
apply (*induct-tac ss rule*: *rev-induct*)
apply *auto*
done

lemma *Abs-apps-eq-Abs-apps-conv* [*iff*]:
$\bigwedge ss.\ (Abs\ r \circ\circ rs = Abs\ s \circ\circ ss) = (r = s \wedge rs = ss)$
apply (*induct rs rule*: *rev-induct*)
apply *simp*
apply *blast*
apply (*induct-tac ss rule*: *rev-induct*)
apply *auto*
done

lemma *Abs-App-neq-Var-apps* [*iff*]:
$\forall\, s\ t.\ Abs\ s \circ t\ {}^\sim\!= Var\ n \circ\circ ss$
apply (*induct-tac ss rule*: *rev-induct*)
apply *auto*
done

lemma *Var-apps-neq-Abs-apps* [*iff*]:
$\bigwedge ts.\ Var\ n \circ\circ ts\ {}^\sim\!= Abs\ r \circ\circ ss$
apply (*induct ss rule*: *rev-induct*)
apply *simp*
apply (*induct-tac ts rule*: *rev-induct*)
apply *auto*
done

lemma *ex-head-tail*:
$\exists\, ts\ h.\ t = h \circ\circ ts \wedge ((\exists\, n.\ h = Var\ n) \vee (\exists\, u.\ h = Abs\ u))$
apply (*induct-tac t*)
apply (*rule-tac x = []* **in** *exI*)
apply *simp*
apply *clarify*

apply (*rename-tac ts1 ts2 h1 h2*)
apply (*rule-tac* $x = ts1$ @ [$h2$ $^{\circ\circ}$ $ts2$] **in** *exI*)
apply *simp*
apply *simp*
done

lemma *size-apps* [*simp*]:
size (r $^{\circ\circ}$ rs) = *size* r + *foldl* (*op* +) *0* (*map size rs*) + *length rs*
apply (*induct-tac rs rule*: *rev-induct*)
apply *auto*
done

lemma *lem0*: [| (*0*::*nat*) < k; m <= n |] ==> $m < n + k$
apply *simp*
done

lemma *lift-map* [*simp*]:
$\bigwedge t$. *lift* (t $^{\circ\circ}$ ts) i = *lift* t i $^{\circ\circ}$ *map* (λt. *lift* t i) ts
by (*induct ts*) *simp-all*

lemma *subst-map* [*simp*]:
$\bigwedge t$. *subst* (t $^{\circ\circ}$ ts) u i = *subst* t u i $^{\circ\circ}$ *map* (λt. *subst* t u i) ts
by (*induct ts*) *simp-all*

lemma *app-last*: (t $^{\circ\circ}$ ts) $^{\circ}$ u = t $^{\circ\circ}$ (ts @ [u])
by *simp*

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

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 −
case *rule-context*
show *?thesis*
apply (*induct-tac n rule*: *nat-less-induct*)
apply (*rule allI*)
apply (*cut-tac* $t = t$ **in** *ex-head-tail*)
apply *clarify*
apply (*erule disjE*)
apply *clarify*
apply (*rule prems*)
apply *clarify*
apply (*erule allE*, *erule impE*)
prefer *2*
apply (*erule allE*, *erule mp*, *rule refl*)
apply *simp*
apply (*rule lem0*)
apply *force*

apply (*rule elem-le-sum*)
apply *force*
apply *clarify*
apply (*rule prems*)
apply (*erule allE*, *erule impE*)
prefer *2*
apply (*erule allE*, *erule mp*, *rule refl*)
apply *simp*
apply *clarify*
apply (*erule allE*, *erule impE*)
prefer *2*
apply (*erule allE*, *erule mp*, *rule refl*)
apply *simp*
apply (*rule le-imp-less-Suc*)
apply (*rule trans-le-add1*)
apply (*rule trans-le-add2*)
apply (*rule elem-le-sum*)
apply *force*
done
qed

theorem *Apps-dB-induct*:
[| !!*n ts*. ∀ *t* ∈ *set ts*. *P t* ==> *P* (*Var n* °° *ts*);
!!*u ts*. [| *P u*; ∀ *t* ∈ *set ts*. *P t* |] ==> *P* (*Abs u* °° *ts*)
|] ==> *P t*
proof −
case *rule-context*
show *?thesis*
apply (*rule-tac t* = *t* **in** *lem*)
prefer *3*
apply (*rule refl*)
apply (*assumption* | *rule prems*)+
done
qed

end

6 Simply-typed lambda terms

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

6.1 Environments

constdefs
shift :: (*nat* ⇒ '*a*) ⇒ *nat* ⇒ '*a* ⇒ *nat* ⇒ '*a* (-<-:-> [*90*, *0*, *0*] *91*)
e<*i*:*a*> ≡ λ*j*. *if j* < *i then e j else if j* = *i then a else e* (*j* − *1*)
syntax (*xsymbols*)
shift :: (*nat* ⇒ '*a*) ⇒ *nat* ⇒ '*a* ⇒ *nat* ⇒ '*a* (-⟨-:-⟩ [*90*, *0*, *0*] *91*)

```
syntax (HTML output)
  shift :: (nat ⇒ 'a) ⇒ nat ⇒ 'a ⇒ nat ⇒ 'a    (-⟨-:-⟩ [90, 0, 0] 91)

lemma shift-eq [simp]: i = j ⟹ (e⟨i:T⟩) j = T
  by (simp add: shift-def)

lemma shift-gt [simp]: j < i ⟹ (e⟨i:T⟩) j = e j
  by (simp add: shift-def)

lemma shift-lt [simp]: i < j ⟹ (e⟨i:T⟩) j = e (j − 1)
  by (simp add: shift-def)

lemma shift-commute [simp]: e⟨i:U⟩⟨0:T⟩ = e⟨0:T⟩⟨Suc i:U⟩
  apply (rule ext)
  apply (case-tac x)
   apply simp
  apply (case-tac nat)
   apply (simp-all add: shift-def)
  done
```

6.2 Types and typing rules

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

consts
  typing :: ((nat ⇒ type) × dB × type) set
  typings :: (nat ⇒ type) ⇒ dB list ⇒ type list ⇒ bool

syntax
  -funs :: type list ⇒ type ⇒ type    (infixr =>> 200)
  -typing :: (nat ⇒ type) ⇒ dB ⇒ type ⇒ bool    (- |− - : - [50, 50, 50] 50)
  -typings :: (nat ⇒ type) ⇒ dB list ⇒ type list ⇒ bool
    (- ||− - : - [50, 50, 50] 50)
syntax (xsymbols)
  -typing :: (nat ⇒ type) ⇒ dB ⇒ type ⇒ bool    (- ⊢ - : - [50, 50, 50] 50)
syntax (latex)
  -funs :: type list ⇒ type ⇒ type    (infixr ⇛ 200)
  -typings :: (nat ⇒ type) ⇒ dB list ⇒ type list ⇒ bool
    (- ⊩ - : - [50, 50, 50] 50)
translations
  Ts ⇛ T ⇌ foldr Fun Ts T
  env ⊢ t : T ⇌ (env, t, T) ∈ typing
  env ⊩ ts : Ts ⇌ typings env ts Ts

inductive typing
  intros
    Var [intro!]: env x = T ⟹ env ⊢ 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
$(e \Vdash [] : Ts) = (Ts = [])$
$(e \Vdash (t \# ts) : Ts) =$
$(case\ Ts\ of$
$[] \Rightarrow False$
$|\ T \# Ts \Rightarrow e \vdash t : T \wedge e \Vdash ts : Ts)$

6.3 Some examples

lemma $e \vdash Abs\ (Abs\ (Abs\ (Var\ 1 \circ (Var\ 2 \circ Var\ 1 \circ Var\ 0)))) : ?T$
by *force*

lemma $e \vdash Abs\ (Abs\ (Abs\ (Var\ 2 \circ Var\ 0 \circ (Var\ 1 \circ Var\ 0)))) : ?T$
by *force*

6.4 Lists of types

lemma *lists-typings*:
$\bigwedge Ts.\ e \Vdash ts : Ts \Longrightarrow ts \in lists\ \{t.\ \exists T.\ e \vdash t : T\}$
apply (*induct ts*)
apply (*case-tac Ts*)
apply *simp*
apply (*rule lists.Nil*)
apply *simp*
apply (*case-tac Ts*)
apply *simp*
apply *simp*
apply (*rule lists.Cons*)
apply *blast*
apply *blast*
done

lemma *types-snoc*: $\bigwedge Ts.\ e \Vdash ts : Ts \Longrightarrow e \vdash t : T \Longrightarrow e \Vdash ts\ @\ [t] : Ts\ @\ [T]$
apply (*induct ts*)
apply *simp*
apply (*case-tac Ts*)
apply *simp+*
done

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

apply (*case-tac Ts*)
apply *simp+*
apply (*case-tac Ts*)
apply (*case-tac ts* @ [*t*])
apply *simp+*
done

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
apply (*subgoal-tac* $\forall ys.\ xs = rev\ ys \longrightarrow P$)
apply (*erule-tac x=rev xs* **in** *allE*)
apply *simp*
apply (*rule allI*)
apply (*rule impI*)
apply (*case-tac ys*)
apply *simp*
apply *simp*
apply *atomize*
apply (*erule allE*)+
apply (*erule mp*, *rule conjI*)
apply (*rule refl*)+
done

lemma *types-snocE*: $e \Vdash ts\ @\ [t] : Ts \Longrightarrow$
$(\bigwedge Us\ U.\ Ts = Us\ @\ [U] \Longrightarrow e \Vdash ts : Us \Longrightarrow e \vdash t : U \Longrightarrow P) \Longrightarrow P$
apply (*cases Ts rule*: *rev-exhaust2*)
apply *simp*
apply (*case-tac ts* @ [*t*])
apply (*simp add*: *types-snoc-eq*)+
apply *iprover*
done

6.5 n-ary function types

lemma *list-app-typeD*:
$\bigwedge t\ T.\ e \vdash t\ {}^{\circ\circ}\ ts : T \Longrightarrow \exists Ts.\ e \vdash t : Ts \Rrightarrow T \wedge e \Vdash ts : Ts$
apply (*induct ts*)
apply *simp*
apply *atomize*
apply *simp*
apply (*erule-tac* $x = t \circ a$ **in** *allE*)
apply (*erule-tac* $x = T$ **in** *allE*)
apply (*erule impE*)
apply *assumption*
apply (*elim exE conjE*)
apply (*ind-cases* $e \vdash t \circ u : T$)
apply (*rule-tac* $x = Ta\ \#\ Ts$ **in** *exI*)
apply *simp*

done

lemma *list-app-typeE*:
$e \vdash t \circ\circ ts : T \Longrightarrow (\bigwedge Ts.\ e \vdash t : Ts \Rrightarrow T \Longrightarrow e \Vdash ts : Ts \Longrightarrow C) \Longrightarrow C$
by (*insert list-app-typeD*) *fast*

lemma *list-app-typeI*:
$\bigwedge t\ T\ Ts.\ e \vdash t : Ts \Rrightarrow T \Longrightarrow e \Vdash ts : Ts \Longrightarrow e \vdash t \circ\circ ts : T$
apply (*induct ts*)
 apply *simp*
apply *atomize*
apply (*case-tac Ts*)
 apply *simp*
apply *simp*
apply (*erule-tac* $x = t \circ a$ **in** *allE*)
apply (*erule-tac* $x = T$ **in** *allE*)
apply (*erule-tac* $x = list$ **in** *allE*)
apply (*erule impE*)
 apply (*erule conjE*)
 apply (*erule typing.App*)
 apply *assumption*
apply *blast*
done

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*:
$\bigwedge T\ U.\ e \vdash Var\ i \circ\circ ts : T \Longrightarrow e \vdash Var\ i \circ\circ ts : U \Longrightarrow T = U$
apply (*induct ts rule*: *rev-induct*)
apply *simp*
apply (*ind-cases* $e \vdash Var\ i : T$)
apply (*ind-cases* $e \vdash Var\ i : T$)
apply *simp*
apply *simp*
apply (*ind-cases* $e \vdash t \circ u : T$)
apply (*ind-cases* $e \vdash t \circ u : T$)
apply *atomize*
apply (*erule-tac* $x{=}Ta \Rightarrow T$ **in** *allE*)
apply (*erule-tac* $x{=}Tb \Rightarrow U$ **in** *allE*)
apply (*erule impE*)
apply *assumption*
apply (*erule impE*)
apply *assumption*
apply *simp*
done

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

apply (*induct us*)
apply *simp*
apply (*erule var-app-type-eq*)
apply *assumption*
apply *simp*
apply *atomize*
apply (*case-tac U*)
apply (*rule FalseE*)
apply *simp*
apply (*erule list-app-typeE*)
apply (*ind-cases* $e \vdash t \circ u : T$)
apply (*drule-tac* T=*Atom nat* **and** U=*Ta* $\Rightarrow$ *Tsa* $\Rrightarrow$ T **in** *var-app-type-eq*)
apply *assumption*
apply *simp*
apply (*erule-tac* x=*ts* @ [*a*] **in** *allE*)
apply (*erule-tac* x=*Ts* @ [*type1*] **in** *allE*)
apply (*erule-tac* x=*type2* **in** *allE*)
apply *simp*
apply (*erule impE*)
apply (*rule types-snoc*)
apply *assumption*
apply (*erule list-app-typeE*)
apply (*ind-cases* $e \vdash t \circ u : T$)
apply (*drule-tac* T=*type1* $\Rightarrow$ *type2* **and** U=*Ta* $\Rightarrow$ *Tsa* $\Rrightarrow$ T **in** *var-app-type-eq*)
apply *assumption*
apply *simp*
apply (*erule impE*)
apply (*rule typing.App*)
apply *assumption*
apply (*erule list-app-typeE*)
apply (*ind-cases* $e \vdash t \circ u : T$)
apply (*frule-tac* T=*type1* $\Rightarrow$ *type2* **and** U=*Ta* $\Rightarrow$ *Tsa* $\Rrightarrow$ T **in** *var-app-type-eq*)
apply *assumption*
apply *simp*
apply (*erule exE*)
apply (*rule-tac* x=*type1* # *Us* **in** *exI*)
apply *simp*
apply (*erule list-app-typeE*)
apply (*ind-cases* $e \vdash t \circ u : T$)
apply (*frule-tac* T=*type1* $\Rightarrow$ *Us* $\Rrightarrow$ T **and** U=*Ta* $\Rightarrow$ *Tsa* $\Rrightarrow$ T **in** *var-app-type-eq*)
apply *assumption*
apply *simp*
done

lemma *var-app-typesE*: $e \vdash Var\ i \circ\circ ts : T \Longrightarrow$
$(\bigwedge Ts.\ e \vdash Var\ i : Ts \Rrightarrow T \Longrightarrow e \Vdash ts : Ts \Longrightarrow P) \Longrightarrow P$
apply (*drule var-app-types* [*of* - - [], *simplified*])
apply (*iprover intro*: *typing.Var*)+
done

lemma *abs-typeE*: $e \vdash Abs\ t : T \Longrightarrow (\bigwedge U\ V.\ e\langle 0{:}U\rangle \vdash t : V \Longrightarrow P) \Longrightarrow P$
  apply (*cases T*)
  apply (*rule FalseE*)
  apply (*erule typing.elims*)
  apply *simp-all*
  apply *atomize*
  apply (*erule-tac x=type1* **in** *allE*)
  apply (*erule-tac x=type2* **in** *allE*)
  apply (*erule mp*)
  apply (*erule typing.elims*)
  apply *simp-all*
  done

6.6 Lifting preserves well-typedness

lemma *lift-type* [*intro*!]: $e \vdash t : T \Longrightarrow (\bigwedge i\ U.\ e\langle i{:}U\rangle \vdash lift\ t\ i : T)$
  by (*induct set*: *typing*) *auto*

lemma *lift-types*:
  $\bigwedge Ts.\ e \Vdash ts : Ts \Longrightarrow e\langle i{:}U\rangle \Vdash (map\ (\lambda t.\ lift\ t\ i)\ ts) : Ts$
  apply (*induct ts*)
   apply *simp*
  apply (*case-tac Ts*)
   apply *auto*
  done

6.7 Substitution lemmas

lemma *subst-lemma*:
    $e \vdash t : T \Longrightarrow (\bigwedge e'\ i\ U\ u.\ e' \vdash u : U \Longrightarrow e = e'\langle i{:}U\rangle \Longrightarrow e' \vdash t[u/i] : T)$
  apply (*induct set*: *typing*)
    apply (*rule-tac* $x = x$ **and** $y = i$ **in** *linorder-cases*)
      apply *auto*
  apply *blast*
  done

lemma *substs-lemma*:
  $\bigwedge Ts.\ e \vdash u : T \Longrightarrow e\langle i{:}T\rangle \Vdash ts : Ts \Longrightarrow$
    $e \Vdash (map\ (\lambda t.\ t[u/i])\ ts) : Ts$
  apply (*induct ts*)
   apply (*case-tac Ts*)
    apply *simp*
   apply *simp*
  apply *atomize*
  apply (*case-tac Ts*)
   apply *simp*
  apply *simp*
  apply (*erule conjE*)
  apply (*erule (1) subst-lemma*)

apply (*rule refl*)
done

6.8 Subject reduction

lemma *subject-reduction*: $e \vdash t : T \Longrightarrow (\bigwedge t'. \ t \ {-}{>} \ t' \Longrightarrow e \vdash t' : T)$
apply (*induct set*: *typing*)
apply *blast*
apply *blast*
apply *atomize*
apply (*ind-cases* $s \circ t \ {-}{>} \ t'$)
apply *hypsubst*
apply (*ind-cases* $env \vdash Abs\ t : T \Rightarrow U$)
apply (*rule subst-lemma*)
apply *assumption*
apply *assumption*
apply (*rule ext*)
apply (*case-tac x*)
apply *auto*
done

theorem *subject-reduction'*: $t \rightarrow_\beta{}^* \ t' \Longrightarrow e \vdash t : T \Longrightarrow e \vdash t' : T$
by (*induct set*: *rtrancl*) (*iprover intro*: *subject-reduction*)+

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 −
case *rule-context*
show *?thesis*
proof (*induct T*)
case *Atom*
show *?case* **by** (*rule rule-context*) *simp-all*
next
case *Fun*
show *?case* **by** (*rule rule-context*) (*insert Fun*, *simp-all*)
qed
qed

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* $r)\hat{}-1$
  apply (*unfold step1-def*)
  apply *blast*
  done

lemma *in-step1-converse* [*iff*]: $(p \in$ *step1* $(r\hat{}-1)) = (p \in ($*step1* $r)\hat{}-1)$
  apply *auto*
  done

lemma *not-Nil-step1* [*iff*]: $([], xs) \notin$ *step1 r*
  apply (*unfold step1-def*)
  apply *blast*
  done

lemma *not-step1-Nil* [*iff*]: $(xs, []) \notin$ *step1 r*
  apply (*unfold step1-def*)
  apply *blast*
  done

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*)
  apply (*unfold step1-def*)
  apply *simp*
  apply (*rule iffI*)
   apply (*erule exE*)
   apply (*rename-tac ts*)
   apply (*case-tac ts*)
    apply *fastsimp*
   apply *force*
  apply (*erule disjE*)
   apply *blast*
  apply (*blast intro*: *Cons-eq-appendI*)
  done

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*
  apply (*unfold step1-def*)
  apply *auto*
   apply *blast*
  apply (*blast intro*: *append-eq-appendI*)

```
  done

lemma Cons-step1E [rule-format, elim!]:
  [| (ys, x # xs) ∈ step1 r;
    ∀ y. ys = y # xs --> (y, x) ∈ r --> R;
    ∀ zs. ys = x # zs --> (zs, xs) ∈ step1 r --> R
   |] ==> R
  apply (case-tac ys)
   apply (simp add: step1-def)
  apply blast
  done

lemma Snoc-step1-SnocD:
  (ys @ [y], xs @ [x]) ∈ step1 r
    ==> ((ys, xs) ∈ step1 r ∧ y = x ∨ ys = xs ∧ (y, x) ∈ r)
  apply (unfold step1-def)
  apply simp
  apply (clarify del: disjCI)
  apply (rename-tac vs)
  apply (rule-tac xs = vs in rev-exhaust)
   apply force
  apply simp
  apply blast
  done

lemma Cons-acc-step1I [rule-format, intro!]:
    x ∈ acc r ==> ∀ xs. xs ∈ acc (step1 r) --> x # xs ∈ acc (step1 r)
  apply (erule acc-induct)
  apply (erule thin-rl)
  apply clarify
  apply (erule acc-induct)
  apply (rule accI)
  apply blast
  done

lemma lists-accD: xs ∈ lists (acc r) ==> xs ∈ acc (step1 r)
  apply (erule lists.induct)
   apply (rule accI)
   apply simp
  apply (rule accI)
  apply (fast dest: acc-downward)
  done

lemma ex-step1I:
  [| x ∈ set xs; (y, x) ∈ r |]
    ==> ∃ ys. (ys, xs) ∈ step1 r ∧ y ∈ set ys
  apply (unfold step1-def)
  apply (drule in-set-conv-decomp [THEN iffD1])
  apply force
```

```
  done

lemma lists-accI: xs ∈ acc (step1 r) ==> xs ∈ lists (acc r)
  apply (erule acc-induct)
  apply clarify
  apply (rule accI)
  apply (drule ex-step1I, assumption)
  apply blast
  done

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)
  apply (erule beta.induct)
     apply simp
    apply (rule allI)
    apply (rule-tac xs = rs in rev-exhaust)
     apply simp
    apply (force intro: append-step1I)
   apply (rule allI)
   apply (rule-tac xs = rs in rev-exhaust)
    apply simp
    apply (auto 0 3 intro: disjI2 [THEN append-step1I])
  done

lemma head-Var-reduction:
  Var n °° rs -> v ==> (∃ ss. rs => ss ∧ v = Var n °° ss)
  apply (drule head-Var-reduction-aux)
  apply blast
  done

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))
  apply (erule beta.induct)
```

```
    apply (clarify del: disjCI)
    apply (case-tac r)
      apply simp
     apply (simp add: App-eq-foldl-conv)
     apply (split split-if-asm)
      apply simp
      apply blast
     apply simp
    apply (simp add: App-eq-foldl-conv)
    apply (split split-if-asm)
     apply simp
    apply simp
   apply (clarify del: disjCI)
   apply (drule App-eq-foldl-conv [THEN iffD1])
   apply (split split-if-asm)
    apply simp
    apply blast
   apply (force intro!: disjI1 [THEN append-step1I])
  apply (clarify del: disjCI)
  apply (drule App-eq-foldl-conv [THEN iffD1])
  apply (split split-if-asm)
   apply simp
   apply blast
  apply (clarify, auto 0 3 intro!: exI intro: append-step1I)
 done

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 -
  assume major: r °° rs -> s
  case rule-context
  show ?thesis
    apply (cut-tac major [THEN apps-betasE-aux, THEN spec, THEN spec])
    apply (assumption | rule refl | erule prems exE conjE impE disjE)+
    done
qed

lemma apps-preserves-beta [simp]:
    r -> s ==> r °° ss -> s °° ss
  apply (induct-tac ss rule: rev-induct)
  apply auto
  done

lemma apps-preserves-beta2 [simp]:
    r ->> s ==> r °° ss ->> s °° ss
  apply (erule rtrancl-induct)
```

apply *blast*
apply (*blast intro*: *apps-preserves-beta rtrancl-into-rtrancl*)
done

lemma *apps-preserves-betas* [*rule-format*, *simp*]:
$\forall ss.\ rs => ss --> r\ {}^{\circ\circ}\ rs -> r\ {}^{\circ\circ}\ ss$
apply (*induct-tac rs rule*: *rev-induct*)
apply *simp*
apply *simp*
apply *clarify*
apply (*rule-tac xs* = *ss* **in** *rev-exhaust*)
apply *simp*
apply *simp*
apply (*drule Snoc-step1-SnocD*)
apply *blast*
done

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*]:
$s : termi\ beta ==> \forall t.\ t : termi\ beta -->$
$(\forall r\ ss.\ t = r[s/0]\ {}^{\circ\circ}\ ss --> Abs\ r\ {}^{\circ}\ s\ {}^{\circ\circ}\ ss : termi\ beta)$
apply (*erule acc-induct*)
apply (*erule thin-rl*)
apply (*rule allI*)
apply (*rule impI*)
apply (*erule acc-induct*)
apply *clarify*
apply (*rule accI*)
apply (*safe elim*!: *apps-betasE*)

apply *assumption*
apply (*blast intro*: *subst-preserves-beta apps-preserves-beta*)
apply (*blast intro*: *apps-preserves-beta2 subst-preserves-beta2 rtrancl-converseI*
dest: *acc-downwards*)
apply (*blast dest*: *apps-preserves-betas*)
done

lemma *IT-implies-termi*: $t : IT ==> t : termi\ beta$
apply (*erule IT.induct*)
apply (*drule rev-subsetD*)
apply (*rule lists-mono*)
apply (*rule Int-lower2*)
apply *simp*
apply (*drule lists-accD*)
apply (*erule acc-induct*)
apply (*rule accI*)
apply (*blast dest*: *head-Var-reduction*)
apply (*erule acc-induct*)
apply (*rule accI*)
apply *blast*
apply (*blast intro*: *double-induction-lemma*)
done

9.3 Every terminating term is in IT

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

lemma [*simp*, *THEN not-sym*, *simp*]: $Var\ n \circ\circ ss \neq Abs\ r \circ s \circ\circ ts$
apply (*simp add*: *foldl-Cons* [*symmetric*] *del*: *foldl-Cons*)
done

lemma [*simp*]:
$(Abs\ r \circ s \circ\circ ss = Abs\ r' \circ s' \circ\circ ss') = (r = r' \wedge s = s' \wedge ss = ss')$
apply (*simp add*: *foldl-Cons* [*symmetric*] *del*: *foldl-Cons*)
done

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

theorem *termi-implies-IT*: $r : termi\ beta ==> r : IT$
apply (*erule acc-induct*)
apply (*rename-tac r*)
apply (*erule thin-rl*)
apply (*erule rev-mp*)
apply *simp*
apply (*rule-tac* $t = r$ **in** *Apps-dB-induct*)
apply *clarify*

apply (*rule IT.intros*)
apply *clarify*
apply (*drule bspec*, *assumption*)
apply (*erule mp*)
apply *clarify*
apply (*drule converseI*)
apply (*drule ex-step1I*, *assumption*)
apply *clarify*
apply (*rename-tac us*)
apply (*erule-tac* $x = Var\ n \circ\circ us$ **in** *allE*)
apply *force*
apply (*rename-tac u ts*)
apply (*case-tac ts*)
 apply *simp*
 apply *blast*
apply (*rename-tac s ss*)
apply *simp*
apply *clarify*
apply (*rule IT.intros*)
 apply (*blast intro*: *apps-preserves-beta*)
apply (*erule mp*)
apply *clarify*
apply (*rename-tac t*)
apply (*erule-tac* $x = Abs\ u \circ t \circ\circ ss$ **in** *allE*)
apply *force*
done

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*!]: $t \in IT \Longrightarrow (\bigwedge i.\ lift\ t\ i \in IT)$
 apply (*induct set*: *IT*)
  apply (*simp* (*no-asm*))
  apply (*rule conjI*)
   apply
   (*rule impI*,
    rule IT.Var,
    erule lists.induct,

```
      simp (no-asm),
      rule lists.Nil,
      simp (no-asm),
      erule IntE,
      rule lists.Cons,
      blast,
      assumption)+
    apply auto
  done

lemma lifts-IT: ts ∈ lists IT ⟹ map (λt. lift t 0) ts ∈ lists IT
  by (induct ts) auto

lemma subst-Var-IT: r ∈ IT ⟹ (⋀i j. r[Var i/j] ∈ IT)
  apply (induct set: IT)
```

Case *Var*:

```
    apply (simp (no-asm) add: subst-Var)
    apply
    ((rule conjI impI)+,
      rule IT.Var,
      erule lists.induct,
      simp (no-asm),
      rule lists.Nil,
      simp (no-asm),
      erule IntE,
      erule CollectE,
      rule lists.Cons,
      fast,
      assumption)+
```

Case *Lambda*:

```
   apply atomize
   apply simp
   apply (rule IT.Lambda)
   apply fast
```

Case *Beta*:

```
  apply atomize
  apply (simp (no-asm-use) add: subst-subst [symmetric])
  apply (rule IT.Beta)
   apply auto
  done

lemma Var-IT: Var n ∈ IT
  apply (subgoal-tac Var n °° [] ∈ IT)
   apply simp
  apply (rule IT.Var)
  apply (rule lists.Nil)
```

```
  done

lemma app-Var-IT: t ∈ IT ⟹ t ° Var i ∈ IT
  apply (induct set: IT)
    apply (subst app-last)
    apply (rule IT.Var)
    apply simp
    apply (rule lists.Cons)
     apply (rule Var-IT)
    apply (rule lists.Nil)
   apply (rule IT.Beta [where ?ss = [], unfolded foldl-Nil [THEN eq-reflection]])
    apply (erule subst-Var-IT)
   apply (rule Var-IT)
  apply (subst app-last)
  apply (rule IT.Beta)
   apply (subst app-last [symmetric])
   apply assumption
  apply assumption
  done
```

10.2 Well-typed substitution preserves termination

```
lemma subst-type-IT:
  ⋀t e T u i. t ∈ IT ⟹ e⟨i:U⟩ ⊢ t : T ⟹
    u ∈ IT ⟹ e ⊢ u : U ⟹ t[u/i] ∈ IT
  (is PROP ?P U is ⋀t e T u i. - ⟹ PROP ?Q t e T u i U)
proof (induct U)
  fix T t
  assume MI1: ⋀T1 T2. T = T1 ⇒ T2 ⟹ PROP ?P T1
  assume MI2: ⋀T1 T2. T = T1 ⇒ T2 ⟹ PROP ?P T2
  assume t ∈ IT
  thus ⋀e T' u i. PROP ?Q t e T' u i T
  proof induct
    fix e T' u i
    assume uIT: u ∈ IT
    assume uT: e ⊢ u : T
    {
      case (Var n rs e- T'- u- i-)
      assume nT: e⟨i:T⟩ ⊢ Var n °° rs : T'
      let ?ty = {t. ∃T'. e⟨i:T⟩ ⊢ t : T'}
      let ?R = λt. ∀e T' u i.
        e⟨i:T⟩ ⊢ t : T' ⟶ u ∈ IT ⟶ e ⊢ u : T ⟶ t[u/i] ∈ IT
      show (Var n °° rs)[u/i] ∈ IT
      proof (cases n = i)
        case True
        show ?thesis
        proof (cases rs)
          case Nil
          with uIT True show ?thesis by simp
```

next
  case (*Cons a as*)
  with *nT* **have** $e\langle i{:}T\rangle \vdash \mathit{Var}\ n \circ a \circ\circ \mathit{as} : T'$ **by** *simp*
  then obtain *Ts*
    where *headT*: $e\langle i{:}T\rangle \vdash \mathit{Var}\ n \circ a : \mathit{Ts} \Rrightarrow T'$
    and *argsT*: $e\langle i{:}T\rangle \Vdash \mathit{as} : \mathit{Ts}$
   by (*rule list-app-typeE*)
  from *headT* **obtain** T''
    where *varT*: $e\langle i{:}T\rangle \vdash \mathit{Var}\ n : T'' \Rightarrow \mathit{Ts} \Rrightarrow T'$
    and *argT*: $e\langle i{:}T\rangle \vdash a : T''$
   by *cases simp-all*
  from *varT True* **have** *T*: $T = T'' \Rightarrow \mathit{Ts} \Rrightarrow T'$
   by *cases auto*
  with *uT* **have** *uT'*: $e \vdash u : T'' \Rightarrow \mathit{Ts} \Rrightarrow T'$ **by** *simp*
  from *T* **have** $(\mathit{Var}\ 0 \circ\circ \mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ 0)$
   $(\mathit{map}\ (\lambda t.\ t[u/i])\ \mathit{as}))[(u \circ a[u/i])/0] \in \mathit{IT}$
  proof (*rule MI2*)
   from *T* **have** $(\mathit{lift}\ u\ 0 \circ \mathit{Var}\ 0)[a[u/i]/0] \in \mathit{IT}$
   proof (*rule MI1*)
    have $\mathit{lift}\ u\ 0 \in \mathit{IT}$ **by** (*rule lift-IT*)
    thus $\mathit{lift}\ u\ 0 \circ \mathit{Var}\ 0 \in \mathit{IT}$ **by** (*rule app-Var-IT*)
    show $e\langle 0{:}T''\rangle \vdash \mathit{lift}\ u\ 0 \circ \mathit{Var}\ 0 : \mathit{Ts} \Rrightarrow T'$
    proof (*rule typing.App*)
     show $e\langle 0{:}T''\rangle \vdash \mathit{lift}\ u\ 0 : T'' \Rightarrow \mathit{Ts} \Rrightarrow T'$
      by (*rule lift-type*) (*rule uT'*)
     show $e\langle 0{:}T''\rangle \vdash \mathit{Var}\ 0 : T''$
      by (*rule typing.Var*) *simp*
    qed
    from *Var* **have** *?R a* **by** *cases* (*simp-all add*: *Cons*)
    with *argT uIT uT* **show** $a[u/i] \in \mathit{IT}$ **by** *simp*
    from *argT uT* **show** $e \vdash a[u/i] : T''$
     by (*rule subst-lemma*) *simp*
   qed
   thus $u \circ a[u/i] \in \mathit{IT}$ **by** *simp*
   from *Var* **have** $\mathit{as} \in \mathit{lists}\ \{t.\ ?R\ t\}$
    by *cases* (*simp-all add*: *Cons*)
   moreover from *argsT* **have** $\mathit{as} \in \mathit{lists}\ ?ty$
    by (*rule lists-typings*)
   ultimately have $\mathit{as} \in \mathit{lists}\ (\{t.\ ?R\ t\} \cap ?ty)$
    by (*rule lists-IntI*)
   hence $\mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ 0)\ (\mathit{map}\ (\lambda t.\ t[u/i])\ \mathit{as}) \in \mathit{lists}\ \mathit{IT}$
    (**is** (*?ls as*) ∈ -)
   proof *induct*
    case *Nil*
    show *?case* **by** *fastsimp*
   next
    case (*Cons b bs*)
    hence *I*: *?R b* **by** *simp*
    from *Cons* **obtain** *U* **where** $e\langle i{:}T\rangle \vdash b : U$ **by** *fast*

with *uT uIT I* **have** $b[u/i] \in IT$ **by** *simp*
hence *lift* $(b[u/i])$ *0* $\in IT$ **by** (*rule lift-IT*)
hence *lift* $(b[u/i])$ *0* # *?ls bs* $\in$ *lists IT*
by (*rule lists.Cons*) (*rule Cons*)
thus *?case* **by** *simp*
qed
thus *Var 0* °° *?ls as* $\in IT$ **by** (*rule IT.Var*)
have $e\langle 0{:}Ts \Rrightarrow T'\rangle \vdash Var\ 0 : Ts \Rrightarrow T'$
by (*rule typing.Var*) *simp*
moreover from *uT argsT* **have** $e \Vdash map\ (\lambda t.\ t[u/i])\ as : Ts$
by (*rule substs-lemma*)
hence $e\langle 0{:}Ts \Rrightarrow T'\rangle \Vdash\ ?ls\ as : Ts$
by (*rule lift-types*)
ultimately show $e\langle 0{:}Ts \Rrightarrow T'\rangle \vdash Var\ 0\ {}^{\circ\circ}\ ?ls\ as : T'$
by (*rule list-app-typeI*)
from *argT uT* **have** $e \vdash a[u/i] : T''$
by (*rule subst-lemma*) (*rule refl*)
with *uT'* **show** $e \vdash u\ {}^\circ\ a[u/i] : Ts \Rrightarrow T'$
by (*rule typing.App*)
qed
with *Cons True* **show** *?thesis*
by (*simp add*: *map-compose* [*symmetric*] *o-def*)
qed
next
case *False*
from *Var* **have** $rs \in lists\ \{t.\ ?R\ t\}$ **by** *simp*
moreover from *nT* **obtain** *Ts* **where** $e\langle i{:}T\rangle \Vdash rs : Ts$
by (*rule list-app-typeE*)
hence $rs \in lists\ ?ty$ **by** (*rule lists-typings*)
ultimately have $rs \in lists\ (\{t.\ ?R\ t\} \cap ?ty)$
by (*rule lists-IntI*)
hence $map\ (\lambda x.\ x[u/i])\ rs \in lists\ IT$
proof *induct*
case *Nil*
show *?case* **by** *fastsimp*
next
case (*Cons a as*)
hence *I*: *?R a* **by** *simp*
from *Cons* **obtain** *U* **where** $e\langle i{:}T\rangle \vdash a : U$ **by** *fast*
with *uT uIT I* **have** $a[u/i] \in IT$ **by** *simp*
hence $(a[u/i]\ \#\ map\ (\lambda t.\ t[u/i])\ as) \in lists\ IT$
by (*rule lists.Cons*) (*rule Cons*)
thus *?case* **by** *simp*
qed
with *False* **show** *?thesis* **by** (*auto simp add*: *subst-Var*)
qed
next
case (*Lambda r e- T'- u- i-*)
assume $e\langle i{:}T\rangle \vdash Abs\ r : T'$

```
      and ⋀e T′ u i. PROP ?Q r e T′ u i T
    with uIT uT show Abs r[u/i] ∈ IT
      by fastsimp
  next
    case (Beta r a as e- T′- u- i-)
    assume T: e⟨i:T⟩ ⊢ Abs r ° a °° as : T′
    assume SI1: ⋀e T′ u i. PROP ?Q (r[a/0] °° as) e T′ u i T
    assume SI2: ⋀e T′ u i. PROP ?Q a e T′ u i T
    have Abs (r[lift u 0/Suc i]) ° a[u/i] °° map (λt. t[u/i]) as ∈ IT
    proof (rule IT.Beta)
      have Abs r ° a °° as →β r[a/0] °° as
        by (rule apps-preserves-beta) (rule beta.beta)
      with T have e⟨i:T⟩ ⊢ r[a/0] °° as : T′
        by (rule subject-reduction)
      hence (r[a/0] °° as)[u/i] ∈ IT
        by (rule SI1)
      thus r[lift u 0/Suc i][a[u/i]/0] °° map (λt. t[u/i]) as ∈ IT
        by (simp del: subst-map add: subst-subst subst-map [symmetric])
      from T obtain U where e⟨i:T⟩ ⊢ Abs r ° a : U
        by (rule list-app-typeE) fast
      then obtain T′′ where e⟨i:T⟩ ⊢ a : T′′ by cases simp-all
      thus a[u/i] ∈ IT by (rule SI2)
    qed
    thus (Abs r ° a °° as)[u/i] ∈ IT by simp
  }
 qed
qed
```

10.3 Well-typed terms are strongly normalizing

```
lemma type-implies-IT: e ⊢ t : T ⟹ t ∈ IT
proof −
  assume e ⊢ t : T
  thus ?thesis
  proof induct
    case Var
    show ?case by (rule Var-IT)
  next
    case Abs
    show ?case by (rule IT.Lambda)
  next
    case (App T U e s t)
    have (Var 0 ° lift t 0)[s/0] ∈ IT
    proof (rule subst-type-IT)
      have lift t 0 ∈ IT by (rule lift-IT)
      hence [lift t 0] ∈ lists IT by (rule lists.Cons) (rule lists.Nil)
      hence Var 0 °° [lift t 0] ∈ IT by (rule IT.Var)
      also have Var 0 °° [lift t 0] = Var 0 ° lift t 0 by simp
      finally show ... ∈ IT .
```

have $e\langle 0{:}T \Rightarrow U\rangle \vdash \mathit{Var}\ 0 : T \Rightarrow U$
by (*rule typing.Var*) *simp*
moreover have $e\langle 0{:}T \Rightarrow U\rangle \vdash \mathit{lift}\ t\ 0 : T$
by (*rule lift-type*)
ultimately show $e\langle 0{:}T \Rightarrow U\rangle \vdash \mathit{Var}\ 0 \circ \mathit{lift}\ t\ 0 : U$
by (*rule typing.App*)
qed
thus *?case* **by** *simp*
qed
qed

theorem *type-implies-termi*: $e \vdash t : T \Longrightarrow t \in \mathit{termi\ beta}$
proof −
assume $e \vdash t : T$
hence $t \in \mathit{IT}$ **by** (*rule type-implies-IT*)
thus *?thesis* **by** (*rule IT-implies-termi*)
qed

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

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

theorem *listall-nil*: *listall P* []
by (*simp add*: *listall-def*)

theorem *listall-nil-eq* [*simp*]: *listall P* [] = *True*
by (*iprover intro*: *listall-nil*)

theorem *listall-cons*: $P\ x \Longrightarrow \mathit{listall}\ P\ \mathit{xs} \Longrightarrow \mathit{listall}\ P\ (x\ \#\ \mathit{xs})$
apply (*simp add*: *listall-def*)
apply (*rule allI impI*)+
apply (*case-tac i*)
apply *simp*+

done

theorem *listall-cons-eq* [*simp*]: *listall P* (*x* # *xs*) = (*P x* ∧ *listall P xs*)
apply (*rule iffI*)
prefer *2*
apply (*erule conjE*)
apply (*erule listall-cons*)
apply *assumption*
apply (*unfold listall-def*)
apply (*rule conjI*)
apply (*erule-tac x=0* **in** *allE*)
apply *simp*
apply *simp*
apply (*rule allI*)
apply (*erule-tac x=Suc i* **in** *allE*)
apply *simp*
done

lemma *listall-conj1*: *listall* (λ*x*. *P x* ∧ *Q x*) *xs* ⟹ *listall P xs*
by (*induct xs*) *simp+*

lemma *listall-conj2*: *listall* (λ*x*. *P x* ∧ *Q x*) *xs* ⟹ *listall Q xs*
by (*induct xs*) *simp+*

lemma *listall-app*: *listall P* (*xs* @ *ys*) = (*listall P xs* ∧ *listall P ys*)
apply (*induct xs*)
apply (*rule iffI*, *simp*, *simp*)
apply (*rule iffI*, *simp*, *simp*)
done

lemma *listall-snoc* [*simp*]: *listall P* (*xs* @ [*x*]) = (*listall P xs* ∧ *P x*)
apply (*rule iffI*)
apply (*simp add*: *listall-app*)+
done

lemma *listall-cong* [*cong*, *extraction-expand*]:
xs = *ys* ⟹ *listall P xs* = *listall P ys*
— Currently needed for strange technical reasons
by (*unfold listall-def*) *simp*

consts *NF* :: *dB set*
inductive *NF*
intros
App: *listall* (λ*t*. *t* ∈ *NF*) *ts* ⟹ *Var x* °° *ts* ∈ *NF*
Abs: *t* ∈ *NF* ⟹ *Abs t* ∈ *NF*
monos *listall-def*

lemma *nat-eq-dec*: ⋀*n*::*nat*. *m* = *n* ∨ *m* ≠ *n*
apply (*induct m*)

apply (*case-tac n*)
apply (*case-tac* [*3*] *n*)
apply (*simp only*: *nat.simps*, *iprover?*)+
done

lemma *nat-le-dec*: $\bigwedge n::nat.\ m < n \lor \neg (m < n)$
apply (*induct m*)
apply (*case-tac n*)
apply (*case-tac* [*3*] *n*)
apply (*simp del*: *simp-thms*, *iprover?*)+
done

lemma *App-NF-D*: **assumes** *NF*: *Var n* °° *ts* ∈ *NF*
shows *listall* (λ*t*. *t* ∈ *NF*) *ts* **using** *NF*
by *cases simp-all*

11.2 Properties of *NF*

lemma *Var-NF*: *Var n* ∈ *NF*
apply (*subgoal-tac Var n* °° [] ∈ *NF*)
apply *simp*
apply (*rule NF.App*)
apply *simp*
done

lemma *subst-terms-NF*: *listall* (λ*t*. *t* ∈ *NF*) *ts* ⟹
listall (λ*t*. ∀ *i j*. *t*[*Var i*/*j*] ∈ *NF*) *ts* ⟹
listall (λ*t*. *t* ∈ *NF*) (*map* (λ*t*. *t*[*Var i*/*j*]) *ts*)
by (*induct ts*) *simp*+

lemma *subst-Var-NF*: *t* ∈ *NF* ⟹ (⋀*i j*. *t*[*Var i*/*j*] ∈ *NF*)
apply (*induct set*: *NF*)
apply *simp*
apply (*frule listall-conj1*)
apply (*drule listall-conj2*)
apply (*drule-tac i*=*i* **and** *j*=*j* **in** *subst-terms-NF*)
apply *assumption*
apply (*rule-tac m*=*x* **and** *n*=*j* **in** *nat-eq-dec* [*THEN disjE*, *standard*])
apply *simp*
apply (*erule NF.App*)
apply (*rule-tac m*=*j* **and** *n*=*x* **in** *nat-le-dec* [*THEN disjE*, *standard*])
apply *simp*
apply (*iprover intro*: *NF.App*)
apply *simp*
apply (*iprover intro*: *NF.App*)
apply *simp*
apply (*iprover intro*: *NF.Abs*)
done

lemma *app-Var-NF*: $t \in NF \Longrightarrow \exists\, t'.\ t \circ\ Var\ i \rightarrow_{\beta}{}^{*}\ t' \wedge t' \in NF$
  apply (*induct set*: *NF*)
  apply (*simplesubst app-last*) — Using *subst* makes extraction fail
  apply (*rule exI*)
  apply (*rule conjI*)
  apply (*rule rtrancl-refl*)
  apply (*rule NF.App*)
  apply (*drule listall-conj1*)
  apply (*simp add*: *listall-app*)
  apply (*rule Var-NF*)
  apply (*rule exI*)
  apply (*rule conjI*)
  apply (*rule rtrancl-into-rtrancl*)
  apply (*rule rtrancl-refl*)
  apply (*rule beta*)
  apply (*erule subst-Var-NF*)
  done

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 $(\lambda t.\ t \in NF)$ $(map\ (\lambda t.\ lift\ t\ i)\ ts)$
  by (*induct ts*) *simp+*

lemma *lift-NF*: $t \in NF \Longrightarrow (\bigwedge i.\ lift\ t\ i \in NF)$
  apply (*induct set*: *NF*)
  apply (*frule listall-conj1*)
  apply (*drule listall-conj2*)
  apply (*drule-tac i=i* **in** *lift-terms-NF*)
  apply *assumption*
  apply (*rule-tac m=x* **and** *n=i* **in** *nat-le-dec* [*THEN disjE*, *standard*])
  apply *simp*
  apply (*rule NF.App*)
  apply *assumption*
  apply *simp*
  apply (*rule NF.App*)
  apply *assumption*
  apply *simp*
  apply (*rule NF.Abs*)
  apply *simp*
  done

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** $\bigwedge t\ e\ T\ u\ i.$ - $\Longrightarrow$ *PROP ?Q t e T u i U*)
proof (*induct U*)
  fix *T t*

let $?R = \lambda t.\ \forall\, e\ T'\ u\ i.$
$e\langle i{:}T\rangle \vdash t : T' \longrightarrow u \in NF \longrightarrow e \vdash u : T \longrightarrow (\exists\, t'.\ t[u/i] \rightarrow_{\beta}{}^{*}\ t' \wedge t' \in NF)$
assume $MI1$: $\bigwedge T1\ T2.\ T = T1 \Rightarrow T2 \Longrightarrow PROP\ ?P\ T1$
assume $MI2$: $\bigwedge T1\ T2.\ T = T1 \Rightarrow T2 \Longrightarrow PROP\ ?P\ T2$
assume $t \in NF$
thus $\bigwedge e\ T'\ u\ i.\ PROP\ ?Q\ t\ e\ T'\ u\ i\ T$
proof *induct*
fix $e\ T'\ u\ i$ **assume** uNF: $u \in NF$ **and** uT: $e \vdash u : T$
{
case $(App\ ts\ x\ e\text{-}\ T'\text{-}\ u\text{-}\ i\text{-})$
assume $appT$: $e\langle i{:}T\rangle \vdash Var\ x\ {}^{\circ\circ}\ ts : T'$
from *nat-eq-dec* **show** $\exists\, t'.\ (Var\ x\ {}^{\circ\circ}\ ts)[u/i] \rightarrow_{\beta}{}^{*}\ t' \wedge t' \in NF$
proof
assume eq: $x = i$
show *?thesis*
proof $(cases\ ts)$
case *Nil*
with *eq* **have** $(Var\ x\ {}^{\circ\circ}\ [])[u/i] \rightarrow_{\beta}{}^{*}\ u$ **by** *simp*
with *Nil* **and** *uNF* **show** *?thesis* **by** *simp iprover*
next
case $(Cons\ a\ as)$
with *appT* **have** $e\langle i{:}T\rangle \vdash Var\ x\ {}^{\circ\circ}\ (a\ \#\ as) : T'$ **by** *simp*
then obtain *Us*
where $varT'$: $e\langle i{:}T\rangle \vdash Var\ x : Us \Rrightarrow T'$
and $argsT'$: $e\langle i{:}T\rangle \Vdash a\ \#\ as : Us$
by $(rule\ var\text{-}app\text{-}typesE)$
from $argsT'$ **obtain** $T''\ Ts$ **where** Us: $Us = T''\ \#\ Ts$
by $(cases\ Us)\ (rule\ FalseE,\ simp+)$
from $varT'$ **and** Us **have** $varT$: $e\langle i{:}T\rangle \vdash Var\ x : T'' \Rightarrow Ts \Rrightarrow T'$
by *simp*
from $varT\ eq$ **have** T: $T = T'' \Rightarrow Ts \Rrightarrow T'$ **by** *cases auto*
with uT **have** uT': $e \vdash u : T'' \Rightarrow Ts \Rrightarrow T'$ **by** *simp*
from $argsT'$ **and** Us **have** $argsT$: $e\langle i{:}T\rangle \Vdash as : Ts$ **by** *simp*
from $argsT'$ **and** Us **have** $argT$: $e\langle i{:}T\rangle \vdash a : T''$ **by** *simp*
from $argT\ uT\ refl$ **have** aT: $e \vdash a[u/i] : T''$ **by** $(rule\ subst\text{-}lemma)$
have as: $\bigwedge Us.\ e\langle i{:}T\rangle \Vdash as : Us \Longrightarrow listall\ ?R\ as \Longrightarrow$
$\exists\, as'.\ Var\ 0\ {}^{\circ\circ}\ map\ (\lambda t.\ lift\ (t[u/i])\ 0)\ as \rightarrow_{\beta}{}^{*}$
$Var\ 0\ {}^{\circ\circ}\ map\ (\lambda t.\ lift\ t\ 0)\ as' \wedge$
$Var\ 0\ {}^{\circ\circ}\ map\ (\lambda t.\ lift\ t\ 0)\ as' \in NF$
(**is** $\bigwedge Us.\ \text{-} \Longrightarrow \text{-} \Longrightarrow \exists\, as'.\ ?ex\ Us\ as\ as'$)
proof $(induct\ as\ rule{:}\ rev\text{-}induct)$
case $(Nil\ Us)$
with *Var-NF* **have** $?ex\ Us\ []\ []$ **by** *simp*
thus *?case* **..**
next
case $(snoc\ b\ bs\ Us)$
have $e\langle i{:}T\rangle \Vdash bs\ @\ [b] : Us$ **.**
then obtain $Vs\ W$ **where** Us: $Us = Vs\ @\ [W]$
and bs: $e\langle i{:}T\rangle \Vdash bs : Vs$ **and** bT: $e\langle i{:}T\rangle \vdash b : W$ **by** $(rule\ types\text{-}snocE)$

from *snoc* **have** *listall ?R bs* **by** *simp*
with *bs* **have** $\exists\, bs'.$ *?ex Vs bs bs'* **by** (*rule snoc*)
then obtain *bs'* **where**
bsred: $\mathit{Var}\ 0\ {}^{\circ\circ}\ \mathit{map}\ (\lambda t.\ \mathit{lift}\ (t[u/i])\ 0)\ \mathit{bs} \rightarrow_\beta{}^*$
$\mathit{Var}\ 0\ {}^{\circ\circ}\ \mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ 0)\ \mathit{bs}'$
and *bsNF*: $\mathit{Var}\ 0\ {}^{\circ\circ}\ \mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ 0)\ \mathit{bs}' \in \mathit{NF}$ **by** *iprover*
from *snoc* **have** *?R b* **by** *simp*
with *bT* **and** *uNF* **and** *uT* **have** $\exists\, b'.\ b[u/i] \rightarrow_\beta{}^*\ b' \wedge b' \in \mathit{NF}$ **by** *iprover*
then obtain *b'* **where** *bred*: $b[u/i] \rightarrow_\beta{}^*\ b'$ **and** *bNF*: $b' \in \mathit{NF}$ **by** *iprover*
from *bsNF* **have** $\mathit{listall}\ (\lambda t.\ t \in \mathit{NF})\ (\mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ 0)\ \mathit{bs}')$
by (*rule App-NF-D*)
moreover have $\mathit{lift}\ b'\ 0 \in \mathit{NF}$ **by** (*rule lift-NF*)
ultimately have $\mathit{listall}\ (\lambda t.\ t \in \mathit{NF})\ (\mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ 0)\ (\mathit{bs}'\ @\ [b']))$
by *simp*
hence $\mathit{Var}\ 0\ {}^{\circ\circ}\ \mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ 0)\ (\mathit{bs}'\ @\ [b']) \in \mathit{NF}$ **by** (*rule NF.App*)
moreover from *bred* **have** $\mathit{lift}\ (b[u/i])\ 0 \rightarrow_\beta{}^*\ \mathit{lift}\ b'\ 0$
by (*rule lift-preserves-beta'*)
with *bsred* **have**
$(\mathit{Var}\ 0\ {}^{\circ\circ}\ \mathit{map}\ (\lambda t.\ \mathit{lift}\ (t[u/i])\ 0)\ \mathit{bs})\ {}^{\circ}\ \mathit{lift}\ (b[u/i])\ 0 \rightarrow_\beta{}^*$
$(\mathit{Var}\ 0\ {}^{\circ\circ}\ \mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ 0)\ \mathit{bs}')\ {}^{\circ}\ \mathit{lift}\ b'\ 0$ **by** (*rule rtrancl-beta-App*)
ultimately have $?\mathit{ex}\ \mathit{Us}\ (\mathit{bs}\ @\ [b])\ (\mathit{bs}'\ @\ [b'])$ **by** *simp*
thus *?case* **..**
qed
from *App* **and** *Cons* **have** *listall ?R as* **by** *simp* (*iprover dest*: *listall-conj2*)
with *argsT* **have** $\exists\, as'.$ *?ex Ts as as'* **by** (*rule as*)
then obtain *as'* **where**
asred: $\mathit{Var}\ 0\ {}^{\circ\circ}\ \mathit{map}\ (\lambda t.\ \mathit{lift}\ (t[u/i])\ 0)\ \mathit{as} \rightarrow_\beta{}^*$
$\mathit{Var}\ 0\ {}^{\circ\circ}\ \mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ 0)\ \mathit{as}'$
and *asNF*: $\mathit{Var}\ 0\ {}^{\circ\circ}\ \mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ 0)\ \mathit{as}' \in \mathit{NF}$ **by** *iprover*
from *App* **and** *Cons* **have** *?R a* **by** *simp*
with *argT* **and** *uNF* **and** *uT* **have** $\exists\, a'.\ a[u/i] \rightarrow_\beta{}^*\ a' \wedge a' \in \mathit{NF}$
by *iprover*
then obtain *a'* **where** *ared*: $a[u/i] \rightarrow_\beta{}^*\ a'$ **and** *aNF*: $a' \in \mathit{NF}$ **by** *iprover*
from *uNF* **have** $\mathit{lift}\ u\ 0 \in \mathit{NF}$ **by** (*rule lift-NF*)
hence $\exists\, u'.\ \mathit{lift}\ u\ 0\ {}^{\circ}\ \mathit{Var}\ 0 \rightarrow_\beta{}^*\ u' \wedge u' \in \mathit{NF}$ **by** (*rule app-Var-NF*)
then obtain *u'* **where** *ured*: $\mathit{lift}\ u\ 0\ {}^{\circ}\ \mathit{Var}\ 0 \rightarrow_\beta{}^*\ u'$ **and** *u'NF*: $u' \in \mathit{NF}$
by *iprover*
from *T* **and** *u'NF* **have** $\exists\, \mathit{ua}.\ u'[a'/0] \rightarrow_\beta{}^*\ \mathit{ua} \wedge \mathit{ua} \in \mathit{NF}$
proof (*rule MI1*)
have $e\langle 0{:}T''\rangle \vdash \mathit{lift}\ u\ 0\ {}^{\circ}\ \mathit{Var}\ 0 : \mathit{Ts} \Rrightarrow T'$
proof (*rule typing.App*)
from *uT'* **show** $e\langle 0{:}T''\rangle \vdash \mathit{lift}\ u\ 0 : T'' \Rightarrow \mathit{Ts} \Rrightarrow T'$ **by** (*rule lift-type*)
show $e\langle 0{:}T''\rangle \vdash \mathit{Var}\ 0 : T''$ **by** (*rule typing.Var*) *simp*
qed
with *ured* **show** $e\langle 0{:}T''\rangle \vdash u' : \mathit{Ts} \Rrightarrow T'$ **by** (*rule subject-reduction'*)
from *ared aT* **show** $e \vdash a' : T''$ **by** (*rule subject-reduction'*)
qed
then obtain *ua* **where** *uared*: $u'[a'/0] \rightarrow_\beta{}^*\ \mathit{ua}$ **and** *uaNF*: $\mathit{ua} \in \mathit{NF}$

by *iprover*
from *ared* **have** $(\mathit{lift}\ u\ 0 \circ \mathit{Var}\ 0)[a[u/i]/0] \rightarrow_\beta^* (\mathit{lift}\ u\ 0 \circ \mathit{Var}\ 0)[a'/0]$
by (*rule subst-preserves-beta2'*)
also from *ured* **have** $(\mathit{lift}\ u\ 0 \circ \mathit{Var}\ 0)[a'/0] \rightarrow_\beta^* u'[a'/0]$
by (*rule subst-preserves-beta'*)
also note *uared*
finally have $(\mathit{lift}\ u\ 0 \circ \mathit{Var}\ 0)[a[u/i]/0] \rightarrow_\beta^* ua$ **.**
hence *uared'*: $u \circ a[u/i] \rightarrow_\beta^* ua$ **by** *simp*
from *T* **have** $\exists r.\ (\mathit{Var}\ 0 \circ\circ \mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ 0)\ \mathit{as}')[ua/0] \rightarrow_\beta^* r \wedge r \in NF$
proof (*rule MI2*)
have $e\langle 0{:}Ts \Rrightarrow T'\rangle \vdash \mathit{Var}\ 0 \circ\circ \mathit{map}\ (\lambda t.\ \mathit{lift}\ (t[u/i])\ 0)\ \mathit{as} : T'$
proof (*rule list-app-typeI*)
show $e\langle 0{:}Ts \Rrightarrow T'\rangle \vdash \mathit{Var}\ 0 : Ts \Rrightarrow T'$ **by** (*rule typing.Var*) *simp*
from *uT argsT* **have** $e \Vdash \mathit{map}\ (\lambda t.\ t[u/i])\ \mathit{as} : Ts$
by (*rule substs-lemma*)
hence $e\langle 0{:}Ts \Rrightarrow T'\rangle \Vdash \mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ 0)\ (\mathit{map}\ (\lambda t.\ t[u/i])\ \mathit{as}) : Ts$
by (*rule lift-types*)
thus $e\langle 0{:}Ts \Rrightarrow T'\rangle \Vdash \mathit{map}\ (\lambda t.\ \mathit{lift}\ (t[u/i])\ 0)\ \mathit{as} : Ts$
by (*simp-all add*: *map-compose* [*symmetric*] *o-def*)
qed
with *asred* **show** $e\langle 0{:}Ts \Rrightarrow T'\rangle \vdash \mathit{Var}\ 0 \circ\circ \mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ 0)\ \mathit{as}' : T'$
by (*rule subject-reduction'*)
from *argT uT refl* **have** $e \vdash a[u/i] : T''$ **by** (*rule subst-lemma*)
with *uT'* **have** $e \vdash u \circ a[u/i] : Ts \Rrightarrow T'$ **by** (*rule typing.App*)
with *uared'* **show** $e \vdash ua : Ts \Rrightarrow T'$ **by** (*rule subject-reduction'*)
qed
then obtain *r* **where** *rred*: $(\mathit{Var}\ 0 \circ\circ \mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ 0)\ \mathit{as}')[ua/0] \rightarrow_\beta^* r$
and *rnf*: $r \in NF$ **by** *iprover*
from *asred* **have**
$(\mathit{Var}\ 0 \circ\circ \mathit{map}\ (\lambda t.\ \mathit{lift}\ (t[u/i])\ 0)\ \mathit{as})[u \circ a[u/i]/0] \rightarrow_\beta^*$
$(\mathit{Var}\ 0 \circ\circ \mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ 0)\ \mathit{as}')[u \circ a[u/i]/0]$
by (*rule subst-preserves-beta'*)
also from *uared'* **have** $(\mathit{Var}\ 0 \circ\circ \mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ 0)\ \mathit{as}')[u \circ a[u/i]/0] \rightarrow_\beta^*$
$(\mathit{Var}\ 0 \circ\circ \mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ 0)\ \mathit{as}')[ua/0]$ **by** (*rule subst-preserves-beta2'*)
also note *rred*
finally have $(\mathit{Var}\ 0 \circ\circ \mathit{map}\ (\lambda t.\ \mathit{lift}\ (t[u/i])\ 0)\ \mathit{as})[u \circ a[u/i]/0] \rightarrow_\beta^* r$ **.**
with *rnf Cons eq* **show** *?thesis*
by (*simp add*: *map-compose* [*symmetric*] *o-def*) *iprover*
qed
next
assume *neq*: $x \neq i$
show *?thesis*
proof −
from *appT* **obtain** *Us*
where *varT*: $e\langle i{:}T\rangle \vdash \mathit{Var}\ x : Us \Rrightarrow T'$
and *argsT*: $e\langle i{:}T\rangle \Vdash ts : Us$
by (*rule var-app-typesE*)
have *ts*: $\bigwedge Us.\ e\langle i{:}T\rangle \Vdash ts : Us \Longrightarrow \mathit{listall}\ ?R\ ts \Longrightarrow$
$\exists ts'.\ \forall x'.\ \mathit{Var}\ x' \circ\circ \mathit{map}\ (\lambda t.\ t[u/i])\ ts \rightarrow_\beta^* \mathit{Var}\ x' \circ\circ ts' \wedge$

```
            Var x' °° ts' ∈ NF
          (is ⋀Us. - ⟹ - ⟹ ∃ts'. ?ex Us ts ts')
        proof (induct ts rule: rev-induct)
          case (Nil Us)
          with Var-NF have ?ex Us [] [] by simp
          thus ?case ..
        next
          case (snoc b bs Us)
          have e⟨i:T⟩ ⊩ bs  @ [b] : Us .
          then obtain Vs W where Us: Us = Vs @ [W]
          and bs: e⟨i:T⟩ ⊩ bs : Vs and bT: e⟨i:T⟩ ⊢ b : W by (rule types-snocE)
          from snoc have listall ?R bs by simp
          with bs have ∃bs'. ?ex Vs bs bs' by (rule snoc)
          then obtain bs' where
            bsred: ⋀x'. Var x' °° map (λt. t[u/i]) bs →β* Var x' °° bs'
            and bsNF: ⋀x'. Var x' °° bs' ∈ NF by iprover
          from snoc have ?R b by simp
            with bT and uNF and uT have ∃b'. b[u/i] →β* b' ∧ b' ∈ NF by
iprover
        then obtain b' where bred: b[u/i] →β* b' and bNF: b' ∈ NF by iprover
         from bsred bred have ⋀x'. (Var x' °° map (λt. t[u/i]) bs) ° b[u/i] →β*
            (Var x' °° bs') ° b' by (rule rtrancl-beta-App)
          moreover from bsNF [of 0] have listall (λt. t ∈ NF) bs'
            by (rule App-NF-D)
          with bNF have listall (λt. t ∈ NF) (bs' @ [b']) by simp
          hence ⋀x'. Var x' °° (bs' @ [b']) ∈ NF by (rule NF.App)
          ultimately have ?ex Us (bs @ [b]) (bs' @ [b']) by simp
          thus ?case ..
        qed
        from App have listall ?R ts by (iprover dest: listall-conj2)
        with argsT have ∃ts'. ?ex Ts ts ts' by (rule ts)
        then obtain ts' where NF: ?ex Ts ts ts' ..
        from nat-le-dec show ?thesis
        proof
          assume i < x
          with NF show ?thesis by simp iprover
        next
          assume ¬ (i < x)
          with NF neq show ?thesis by (simp add: subst-Var) iprover
        qed
      qed
    qed
  next
    case (Abs r e- T'- u- i-)
    assume absT: e⟨i:T⟩ ⊢ Abs r : T'
   then obtain R S where e⟨0:R⟩⟨Suc i:T⟩  ⊢ r : S by (rule abs-typeE) simp
    moreover have lift u 0 ∈ NF by (rule lift-NF)
    moreover have e⟨0:R⟩ ⊢ lift u 0 : T by (rule lift-type)
    ultimately have ∃t'. r[lift u 0/Suc i] →β* t' ∧ t' ∈ NF by (rule Abs)
```

thus $\exists\, t'.\ Abs\ r[u/i] \rightarrow_\beta{}^* t' \wedge t' \in NF$
by *simp* (*iprover intro*: *rtrancl-beta-Abs NF.Abs*)
}
qed
qed

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

syntax
-rtyping :: (*nat* ⇒ *type*) ⇒ *dB* ⇒ *type* ⇒ *bool* (- $|-_R$ - : - [*50*, *50*, *50*] *50*)
syntax (*xsymbols*)
-rtyping :: (*nat* ⇒ *type*) ⇒ *dB* ⇒ *type* ⇒ *bool* (- $\vdash_R$ - : - [*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*: $e \vdash_R t : T \Longrightarrow e \vdash t : T$
apply (*induct set*: *rtyping*)
apply (*erule typing.Var*)
apply (*erule typing.Abs*)
apply (*erule typing.App*)
apply *assumption*
done

theorem *type-NF*: **assumes** *T*: $e \vdash_R t : T$
shows $\exists\, t'.\ t \rightarrow_\beta{}^* t' \wedge t' \in NF$ **using** *T*
proof *induct*
case *Var*
show *?case* **by** (*iprover intro*: *Var-NF*)
next
case *Abs*
thus *?case* **by** (*iprover intro*: *rtrancl-beta-Abs NF.Abs*)
next
case (*App T U e s t*)
from *App* **obtain** $s'\ t'$ **where**
sred: $s \rightarrow_\beta{}^* s'$ **and** *sNF*: $s' \in NF$
and *tred*: $t \rightarrow_\beta{}^* t'$ **and** *tNF*: $t' \in NF$ **by** *iprover*
have $\exists\, u.\ (Var\ 0\ {}^\circ\ lift\ t'\ 0)[s'/0] \rightarrow_\beta{}^* u \wedge u \in NF$
proof (*rule subst-type-NF*)
have $lift\ t'\ 0 \in NF$ **by** (*rule lift-NF*)
hence $listall\ (\lambda t.\ t \in NF)\ [lift\ t'\ 0]$ **by** (*rule listall-cons*) (*rule listall-nil*)

```
    hence Var 0 °° [lift t′ 0] ∈ NF by (rule NF.App)
    thus Var 0 ° lift t′ 0 ∈ NF by simp
    show e⟨0:T ⇒ U⟩ ⊢ Var 0 ° lift t′ 0 : U
    proof (rule typing.App)
      show e⟨0:T ⇒ U⟩ ⊢ Var 0 : T ⇒ U
        by (rule typing.Var) simp
      from tred have e ⊢ t′ : T
        by (rule subject-reduction′) (rule rtyping-imp-typing)
      thus e⟨0:T ⇒ U⟩ ⊢ lift t′ 0 : T
        by (rule lift-type)
    qed
    from sred show e ⊢ s′ : T ⇒ U
      by (rule subject-reduction′) (rule rtyping-imp-typing)
  qed
  then obtain u where ured: s′ ° t′ →β* u and unf: u ∈ NF by simp iprover
  from sred tred have s ° t →β* s′ ° t′ by (rule rtrancl-beta-App)
  hence s ° t →β* u using ured by (rule rtrancl-trans)
  with unf show ?case by iprover
qed
```

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) ∈ rtranclR r) = ((a, b) ∈ rtrancl (Collect r))
  apply (rule iffI)
  apply (erule rtranclR.induct)
  apply (rule rtrancl-refl)
  apply (erule rtrancl-into-rtrancl)
  apply (erule CollectI)
  apply (erule rtrancl.induct)
  apply (rule rtranclR.rtrancl-refl)
  apply (erule rtranclR.rtrancl-into-rtrancl)
  apply (erule CollectD)
  done

lemma NFR-imp-NF: (nf, t) ∈ NFR ⟹ t ∈ NF
  apply (erule NFR.induct)
  apply (rule NF.intros)
  apply (simp add: listall-def)
  apply (erule NF.intros)
  done
```

The program corresponding to the proof of the central lemma, which performs substitution and normalization, is shown in Figure 1. The correctness

```
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

theorem corresponding to the program *subst-type-NF* is

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

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 ⟪
```
fun nat-of-int 0 = Norm.id-0
  | nat-of-int n = Norm.Suc (nat-of-int (n−1));

fun int-of-nat Norm.id-0 = 0
  | int-of-nat (Norm.Suc n) = 1 + int-of-nat n;

fun dBtype-of-typ (Type (fun, [T, U])) =
      Norm.Fun (dBtype-of-typ T, dBtype-of-typ U)
  | dBtype-of-typ (TFree (s, -)) = (case explode s of
        [', a] => Norm.Atom (nat-of-int (ord a − 97))
```

```
      | _ => error dBtype-of-typ: variable name)
  | dBtype-of-typ _ = error dBtype-of-typ: bad type;

fun dB-of-term (Bound i) = Norm.dB-Var (nat-of-int i)
  | dB-of-term (t $ u) = Norm.App (dB-of-term t, dB-of-term u)
  | dB-of-term (Abs (_, _, t)) = Norm.Abs (dB-of-term t)
  | dB-of-term _ = error dB-of-term: bad term;

fun term-of-dB Ts (Type (fun, [T, U])) (Norm.Abs dBt) =
      Abs (x, T, term-of-dB (T :: Ts) U dBt)
  | term-of-dB Ts _ dBt = term-of-dB' Ts dBt
and term-of-dB' Ts (Norm.dB-Var n) = Bound (int-of-nat n)
  | term-of-dB' Ts (Norm.App (dBt, dBu)) =
      let val t = term-of-dB' Ts dBt
      in case fastype-of1 (Ts, t) of
          Type (fun, [T, U]) => t $ term-of-dB Ts T dBu
        | _ => error term-of-dB: function type expected
      end
  | term-of-dB' _ _ = error term-of-dB: term not in normal form;

fun typing-of-term Ts e (Bound i) =
      Norm.Var (e, nat-of-int i, dBtype-of-typ (List.nth (Ts, i)))
  | typing-of-term Ts e (t $ u) = (case fastype-of1 (Ts, t) of
        Type (fun, [T, U]) => Norm.rtypingT-App (e, dB-of-term t,
          dBtype-of-typ T, dBtype-of-typ U, dB-of-term u,
          typing-of-term Ts e t, typing-of-term Ts e u)
      | _ => error typing-of-term: function type expected)
  | typing-of-term Ts e (Abs (s, T, t)) =
      let val dBT = dBtype-of-typ T
      in Norm.rtypingT-Abs (e, dBT, dB-of-term t,
        dBtype-of-typ (fastype-of1 (T :: Ts, t)),
        typing-of-term (T :: Ts) (Norm.shift e Norm.id-0 dBT) t)
      end
  | typing-of-term _ _ _ = error typing-of-term: bad term;

fun dummyf _ = error dummy;
⟫
```

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

ML ⟪
```
val sg = sign-of (the-context());
fun rd s = read-cterm sg (s, TypeInfer.logicT);

val ct1 = rd %f. ((%f x. f (f (f x))) ((%f x. f (f (f (f x)))) f));
val (dB1, _) = Norm.type-NF (typing-of-term [] dummyf (term-of ct1));
val ct1' = cterm-of sg (term-of-dB [] (#T (rep-cterm ct1)) dB1);

val ct2 = rd
  %f x. (%x. f x x) ((%x. f x x) ((%x. f x x) ((%x. f x x) ((%x. f x x) ((%x. f x x)
```

x)))));
val (*dB2*, -) = *Norm.type-NF* (*typing-of-term* [] *dummyf* (*term-of ct2*));
val ct2' = *cterm-of sg* (*term-of-dB* [] (#*T* (*rep-cterm ct2*)) *dB2*);
⟫

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.