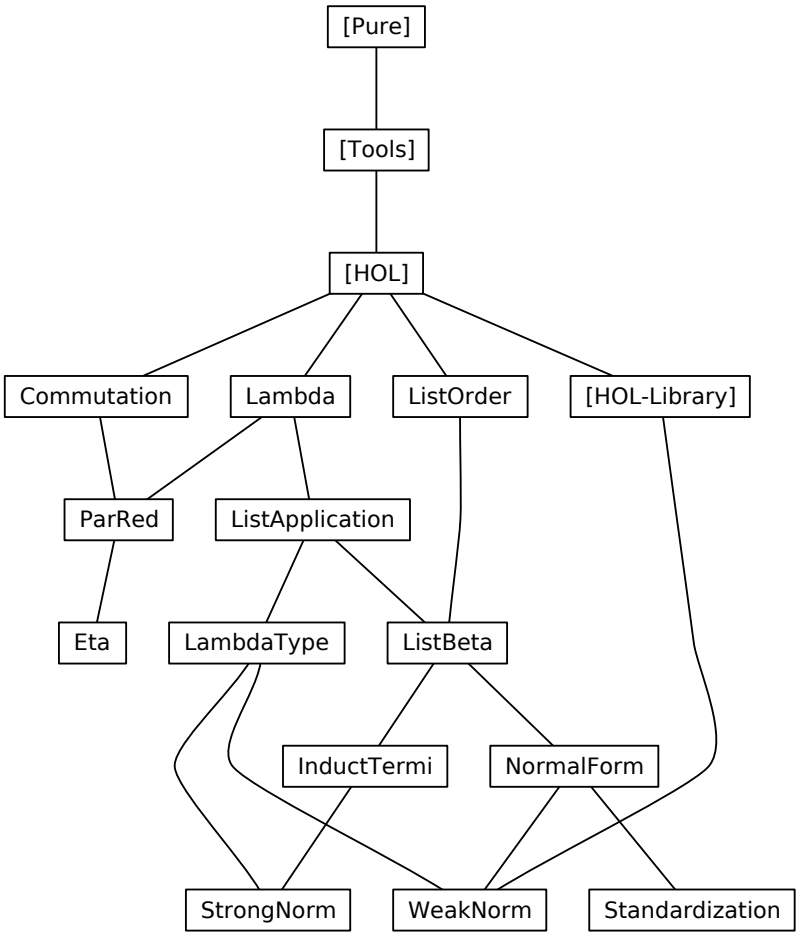
Fundamental Properties of Lambda-calculus

Tobias Nipkow
Stefan Berghofer

March 13, 2025

Contents


[Pure]
[Tools]
[HOL]
Commutation
Lambda
ListOrder
[HOL-Library]
ParRed
ListApplication
Eta
LambdaType
ListBeta
InductTermi
NormalForm
StrongNorm
WeakNorm
Standardization

1 Basic definitions of Lambda-calculus

theory *Lambda*
imports *Main*
begin

declare [[*syntax-ambiguity-warning* = *false*]]

1.1 Lambda-terms in de Bruijn notation and substitution

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

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

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

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

Optimized versions of *subst* and *lift*.

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

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

1.2 Beta-reduction

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

abbreviation
beta-reds :: [*dB*, *dB*] => *bool* (**infixl** ‹$\to_\beta{}^*$› *50*) **where**
s $\to_\beta{}^*$ *t* == *beta*** *s t*

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

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

1.3 Congruence rules

lemma *rtrancl-beta-Abs* [*intro*!]:
s $\to_\beta{}^*$ *s'* ⟹ *Abs s* $\to_\beta{}^*$ *Abs s'*
by (*induct set*: *rtranclp*) (*blast intro*: *rtranclp.rtrancl-into-rtrancl*)+

lemma *rtrancl-beta-AppL*:
s $\to_\beta{}^*$ *s'* ⟹ *s* ° *t* $\to_\beta{}^*$ *s'* ° *t*
by (*induct set*: *rtranclp*) (*blast intro*: *rtranclp.rtrancl-into-rtrancl*)+

lemma *rtrancl-beta-AppR*:
t $\to_\beta{}^*$ *t'* ⟹ *s* ° *t* $\to_\beta{}^*$ *s* ° *t'*
by (*induct set*: *rtranclp*) (*blast intro*: *rtranclp.rtrancl-into-rtrancl*)+

lemma *rtrancl-beta-App* [*intro*]:
⟦*s* $\to_\beta{}^*$ *s'*; *t* $\to_\beta{}^*$ *t'*⟧ ⟹ *s* ° *t* $\to_\beta{}^*$ *s'* ° *t'*
by (*blast intro*!: *rtrancl-beta-AppL rtrancl-beta-AppR intro*: *rtranclp-trans*)

1.4 Substitution-lemmas

lemma *subst-eq* [*simp*]: (*Var k*)[*u*/*k*] = *u*
by (*simp add*: *subst-Var*)

lemma *subst-gt* [*simp*]: *i* < *j* ⟹ (*Var j*)[*u*/*i*] = *Var* (*j* − *1*)
by (*simp add*: *subst-Var*)

lemma *subst-lt* [*simp*]: *j* < *i* ⟹ (*Var j*)[*u*/*i*] = *Var j*
by (*simp add*: *subst-Var*)

lemma *lift-lift*:
  $i < k + 1 \Longrightarrow lift\ (lift\ t\ i)\ (Suc\ k) = lift\ (lift\ t\ k)\ i$
 by (*induct t arbitrary*: *i k*) *auto*

lemma *lift-subst* [*simp*]:
  $j < i + 1 \Longrightarrow lift\ (t[s/j])\ i = (lift\ t\ (i + 1))\ [lift\ s\ i\ /\ j]$
 by (*induct t arbitrary*: *i j s*)
  (*simp-all add*: *diff-Suc subst-Var lift-lift split*: *nat.split*)

lemma *lift-subst-lt*:
  $i < j + 1 \Longrightarrow lift\ (t[s/j])\ i = (lift\ t\ i)\ [lift\ s\ i\ /\ j + 1]$
 by (*induct t arbitrary*: *i j s*) (*simp-all add*: *subst-Var lift-lift*)

lemma *subst-lift* [*simp*]:
  $(lift\ t\ k)[s/k] = t$
 by (*induct t arbitrary*: *k s*) *simp-all*

lemma *subst-subst*:
  $i < j + 1 \Longrightarrow t[lift\ v\ i\ /\ Suc\ j][u[v/j]/i] = t[u/i][v/j]$
 by (*induct t arbitrary*: *i j u v*)
  (*simp-all add*: *diff-Suc subst-Var lift-lift* [*symmetric*] *lift-subst-lt*
    split: *nat.split*)

1.5 Equivalence proof for optimized substitution

lemma *liftn-0* [*simp*]: $liftn\ 0\ t\ k = t$
 by (*induct t arbitrary*: *k*) (*simp-all add*: *subst-Var*)

lemma *liftn-lift* [*simp*]: $liftn\ (Suc\ n)\ t\ k = lift\ (liftn\ n\ t\ k)\ k$
 by (*induct t arbitrary*: *k*) (*simp-all add*: *subst-Var*)

lemma *substn-subst-n* [*simp*]: $substn\ t\ s\ n = t[liftn\ n\ s\ 0\ /\ n]$
 by (*induct t arbitrary*: *n*) (*simp-all add*: *subst-Var*)

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

1.6 Preservation theorems

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

theorem *subst-preserves-beta* [*simp*]:
  $r \rightarrow_\beta s \Longrightarrow r[t/i] \rightarrow_\beta s[t/i]$
 by (*induct arbitrary*: *t i set*: *beta*) (*simp-all add*: *subst-subst* [*symmetric*])

theorem *subst-preserves-beta'*: $r \rightarrow_\beta{}^* s \Longrightarrow r[t/i] \rightarrow_\beta{}^* s[t/i]$
proof (*induct set*: *rtranclp*)
 case *base*
 then show *?case*

by (*iprover intro*: *rtrancl-refl*)
next
case (*step y z*)
then show *?case*
by (*iprover intro*: *rtranclp.simps subst-preserves-beta*)
qed

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

theorem *lift-preserves-beta'*: $r \to_\beta^* s \Longrightarrow lift\ r\ i \to_\beta^* lift\ s\ i$
proof (*induct set*: *rtranclp*)
case *base*
then show *?case*
by (*iprover intro*: *rtrancl-refl*)
next
case (*step y z*)
then show *?case*
by (*iprover intro*: *lift-preserves-beta rtranclp.simps*)
qed

theorem *subst-preserves-beta2* [*simp*]: $r \to_\beta s \Longrightarrow t[r/i] \to_\beta^* t[s/i]$
proof (*induct t arbitrary*: *r s i*)
case (*Var x*)
then show *?case*
by (*simp add*: *subst-Var r-into-rtranclp*)
next
case (*App t1 t2*)
then show *?case*
by (*simp add*: *rtrancl-beta-App*)
next
case (*Abs t*)
then show *?case* **by** (*simp add*: *rtrancl-beta-Abs*)
qed

theorem *subst-preserves-beta2'*: $r \to_\beta^* s \Longrightarrow t[r/i] \to_\beta^* t[s/i]$
proof (*induct set*: *rtranclp*)
case *base*
then show *?case* **by** (*iprover intro*: *rtrancl-refl*)
next
case (*step y z*)
then show *?case*
by (*iprover intro*: *rtranclp-trans subst-preserves-beta2*)
qed

end

2 Abstract commutation and confluence notions

theory *Commutation*
imports *Main*
begin

declare [[*syntax-ambiguity-warning* = *false*]]

2.1 Basic definitions

definition
square :: ['*a* => '*a* => *bool*, '*a* => '*a* => *bool*, '*a* => '*a* => *bool*, '*a* => '*a* => *bool*] => *bool* **where**
square R S T U =
(∀ *x y*. *R x y* −−> (∀ *z*. *S x z* −−> (∃ *u*. *T y u* ∧ *U z u*)))

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

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

definition
Church-Rosser :: ('*a* => '*a* => *bool*) => *bool* **where**
Church-Rosser R =
(∀ *x y*. (*sup R* (R^{-1-1}))** *x y* ⟶ (∃ *z*. *R*** *x z* ∧ *R*** *y z*))

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

2.2 Basic lemmas

square

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

lemma *square-subset*:
[| *square R S T U*; *T* ≤ *T'* |] ==> *square R S T' U*
apply (*unfold square-def*)
apply (*blast dest*: *predicate2D*)
done

lemma *square-reflcl*:
[| *square R S T* ($R^{==}$); *S* ≤ *T* |] ==> *square* ($R^{==}$) *S T* ($R^{==}$)

apply (*unfold square-def*)
apply (*blast dest*: *predicate2D*)
done

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

lemma *square-rtrancl-reflcl-commute*:
square R S (*S***) (*R*$^{==}$) ==> *commute* (*R***) (*S***)
apply (*unfold commute-def*)
apply (*fastforce dest*: *square-reflcl square-sym* [*THEN square-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* (*R***) (*S***)
apply (*unfold commute-def*)
apply (*blast intro*: *square-rtrancl square-sym*)
done

lemma *commute-Un*:
[| *commute R T*; *commute S T* |] ==> *commute* (*sup R 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* (*sup R S*)
apply (*unfold diamond-def*)
apply (*blast intro*: *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 $(R^{==})$ $(R^{==})$ ==> *confluent R*
 apply (*unfold diamond-def*)
 apply (*fast intro*: *square-rtrancl-reflcl-commute elim*: *square-subset*)
 done

lemma *confluent-Un*:
  [| *confluent R*; *confluent S*; *commute* (R^{**}) (S^{**}) |] ==> *confluent* (*sup R S*)
 apply (*rule rtranclp-sup-rtranclp* [*THEN subst*])
 apply (*blast dest*: *diamond-Un intro*: *diamond-confluent*)
 done

lemma *diamond-to-confluence*:
  [| *diamond R*; $T \leq R$; $R \leq T^{**}$ |] ==> *confluent T*
 apply (*force intro*: *diamond-confluent*
  dest: *rtranclp-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* (*put-claset HOL-cs* ***context***)›)
  apply (*tactic* ‹
   blast-tac (*put-claset HOL-cs* ***context*** *addIs*
     [@{*thm sup-ge2*} *RS* @{*thm rtranclp-mono*} *RS* @{*thm predicate2D*} *RS*
@{*thm rtranclp-trans*},
     @{*thm rtranclp-converseI*}, @{*thm conversepI*},
     @{*thm sup-ge1*} *RS* @{*thm rtranclp-mono*} *RS* @{*thm predicate2D*}]) *1*›)
 apply (*erule rtranclp-induct*)
  apply *blast*
 apply (*blast del*: *rtranclp.rtrancl-refl intro*: *rtranclp-trans*)
 done

2.4 Newman's lemma

Proof by Stefan Berghofer

theorem *newman*:
 assumes *wf*: *wfP* (R^{-1-1})
 and *lc*: $\bigwedge a\ b\ c.\ R\ a\ b \Longrightarrow R\ a\ c \Longrightarrow$
   $\exists d.\ R^{**}\ b\ d \wedge R^{**}\ c\ d$
 shows $\bigwedge b\ c.\ R^{**}\ a\ b \Longrightarrow R^{**}\ a\ c \Longrightarrow$
   $\exists d.\ R^{**}\ b\ d \wedge R^{**}\ c\ d$
 using *wf*
proof *induct*
 case (*less x b c*)
 have *xc*: $R^{**}\ x\ c$ **by** *fact*
 have *xb*: $R^{**}\ x\ b$ **by** *fact* **thus** *?case*
 proof (*rule converse-rtranclpE*)

```
    assume x = b
    with xc have R** b c by simp
    thus ?thesis by iprover
  next
    fix y
    assume xy: R x y
    assume yb: R** y b
    from xc show ?thesis
    proof (rule converse-rtranclpE)
      assume x = c
      with xb have R** c b by simp
      thus ?thesis by iprover
    next
      fix y'
      assume y'c: R** y' c
      assume xy': R x y'
      with xy have ∃u. R** y u ∧ R** y' u by (rule lc)
      then obtain u where yu: R** y u and y'u: R** y' u by iprover
      from xy have R^-1-1 y x ..
      from this and yb yu have ∃d. R** b d ∧ R** u d by (rule less)
      then obtain v where bv: R** b v and uv: R** u v by iprover
      from xy' have R^-1-1 y' x ..
      moreover from y'u and uv have R** y' v by (rule rtranclp-trans)
      moreover note y'c
      ultimately have ∃d. R** v d ∧ R** c d by (rule less)
      then obtain w where vw: R** v w and cw: R** c w by iprover
      from bv vw have R** b w by (rule rtranclp-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 using *blast*.

```
theorem newman':
  assumes wf: wfP (R^-1-1)
  and lc: ⋀a b c. R a b ⟹ R a c ⟹
    ∃d. R** b d ∧ R** c d
  shows ⋀b c. R** a b ⟹ R** a c ⟹
    ∃d. R** b d ∧ R** c d
  using wf
proof induct
  case (less x b c)
  note IH = ‹⋀y b c. ⟦R^-1-1 y x; R** y b; R** y c⟧
                ⟹ ∃d. R** b d ∧ R** c d›
  have xc: R** x c by fact
  have xb: R** x b by fact
  thus ?case
```

```
  proof (rule converse-rtranclpE)
    assume x = b
    with xc have R** b c by simp
    thus ?thesis by iprover
  next
    fix y
    assume xy: R x y
    assume yb: R** y b
    from xc show ?thesis
    proof (rule converse-rtranclpE)
      assume x = c
      with xb have R** c b by simp
      thus ?thesis by iprover
    next
      fix y'
      assume y'c: R** y' c
      assume xy': R x y'
      with xy obtain u where u: R** y u R** y' u
        by (blast dest: lc)
      from yb u y'c show ?thesis
        by (blast del: rtranclp.rtrancl-refl
            intro: rtranclp-trans
            dest: IH [OF conversepI, OF xy] IH [OF conversepI, OF xy'])
    qed
  qed
qed
```

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

```
lemma eq-imp-rtranclp: x = y ⟹ r** x y
  by simp

theorem newman'':
  assumes wf: wfP (R^-1-1)
  and lc: ⋀a b c. R a b ⟹ R a c ⟹
    ∃d. R** b d ∧ R** c d
  shows ⋀b c. R** a b ⟹ R** a c ⟹
    ∃d. R** b d ∧ R** c d
  using wf
proof induct
  case (less x b c)
  note IH = ‹⋀y b c. ⟦R^-1-1 y x; R** y b; R** y c⟧
                ⟹ ∃d. R** b d ∧ R** c d›
  show ?case
    by (coherent
      ‹R** x c› ‹R** x b›
      refl [where 'a='a] sym
      eq-imp-rtranclp
      r-into-rtranclp [of R]
```

rtranclp-trans
lc IH [*OF conversepI*]
converse-rtranclpE)
qed

end

3 Parallel reduction and a complete developments

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

3.1 Parallel reduction

inductive *par-beta* :: [*dB*, *dB*] => *bool* (**infixl** ‹=>› *50*)
where
var [*simp*, *intro*!]: *Var n* => *Var n*
| *abs* [*simp*, *intro*!]: *s* => *t* ==> *Abs s* => *Abs t*
| *app* [*simp*, *intro*!]: [| *s* => *s'*; *t* => *t'* |] ==> *s* ° *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**

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

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

lemma *beta-subset-par-beta*: *beta* <= *par-beta*
apply (*rule predicate2I*)
apply (*erule beta.induct*)
apply (*blast intro*!: *par-beta-refl*)+
done

lemma *par-beta-subset-beta*: *par-beta* ≤ *beta***
apply (*rule predicate2I*)
apply (*erule par-beta.induct*)
apply *blast*
apply (*blast del*: *rtranclp.rtrancl-refl intro*: *rtranclp.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* [*simp*]:
t => t' ⟹ *lift* t n => *lift* t' n
by (*induct* t *arbitrary*: t' n) *fastforce*+

lemma *par-beta-subst*:
s => s' ⟹ t => t' ⟹ $t[s/n]$ => $t'[s'/n]$
apply (*induct* t *arbitrary*: s s' t' n)
apply (*simp add*: *subst-Var*)
apply (*erule par-beta-cases*)
apply *simp*
apply (*simp add*: *subst-subst* [*symmetric*])
apply (*fastforce intro*!: *par-beta-lift*)
apply *fastforce*
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

fun
cd :: *dB* => *dB*
where
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*: s => t ⟹ t => *cd* s
apply (*induct* s *arbitrary*: t *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*
unfolding *diamond-def commute-def square-def*

by (*blast intro*: *par-beta-cd*)

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

end

4 Eta-reduction

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

4.1 Definition of eta-reduction and relatives

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

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

abbreviation
eta-reds :: [*dB*, *dB*] => *bool* (**infixl** ‹→$_\eta^*$› *50*) **where**
s →$_\eta^*$ *t* == *eta*** *s t*

abbreviation
eta-red0 :: [*dB*, *dB*] => *bool* (**infixl** ‹→$_\eta^=$› *50*) **where**
s →$_\eta^=$ *t* == *eta*$^{==}$ *s t*

inductive-cases *eta-cases* [*elim*!]:
Abs s →$_\eta$ *z*
s ° *t* →$_\eta$ *u*
Var i →$_\eta$ *t*

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

lemma *subst-not-free* [*simp*]: ¬ *free s i* ⟹ *s*[*t*/*i*] = *s*[*u*/*i*]
by (*induct s arbitrary*: *i t u*) (*simp-all add*: *subst-Var*)

lemma *free-lift* [*simp*]:
free (*lift t k*) *i* = (*i* < *k* ∧ *free t i* ∨ *k* < *i* ∧ *free t* (*i* − *1*))

apply (*induct t arbitrary*: *i k*)
apply (*auto cong*: *conj-cong*)
done

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

lemma *free-eta*: *s* →$_\eta$ *t* ==> *free t i* = *free s i*
by (*induct arbitrary*: *i set*: *eta*) (*simp-all cong*: *conj-cong*)

lemma *not-free-eta*:
[| *s* →$_\eta$ *t*; ¬ *free s i* |] ==> ¬ *free t i*
by (*simp add*: *free-eta*)

lemma *eta-subst* [*simp*]:
s →$_\eta$ *t* ==> *s*[*u*/*i*] →$_\eta$ *t*[*u*/*i*]
by (*induct arbitrary*: *u i set*: *eta*) (*simp-all add*: *subst-subst* [*symmetric*])

theorem *lift-subst-dummy*: ¬ *free s i* ⟹ *lift* (*s*[*dummy*/*i*]) *i* = *s*
by (*induct s arbitrary*: *i dummy*) *simp-all*

4.3 Confluence of *eta*

lemma *square-eta*: *square eta eta* (*eta*$^{==}$) (*eta*$^{==}$)
apply (*unfold square-def id-def*)
apply (*rule impI* [*THEN allI* [*THEN allI*]])
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* →$_\eta^*$ *s′* ==> *Abs s* →$_\eta^*$ *Abs s′*

by (*induct set*: *rtranclp*)
(*blast intro*: *rtranclp.rtrancl-into-rtrancl*)+

lemma *rtrancl-eta-AppL*: $s \to_\eta^* s' \Longrightarrow s \circ t \to_\eta^* s' \circ t$
by (*induct set*: *rtranclp*)
(*blast intro*: *rtranclp.rtrancl-into-rtrancl*)+

lemma *rtrancl-eta-AppR*: $t \to_\eta^* t' \Longrightarrow s \circ t \to_\eta^* s \circ t'$
by (*induct set*: *rtranclp*) (*blast intro*: *rtranclp.rtrancl-into-rtrancl*)+

lemma *rtrancl-eta-App*:
$[\![s \to_\eta^* s'; t \to_\eta^* t']\!] \Longrightarrow s \circ t \to_\eta^* s' \circ t'$
by (*blast intro*!: *rtrancl-eta-AppL rtrancl-eta-AppR intro*: *rtranclp-trans*)

4.5 Commutation of *beta* and *eta*

lemma *free-beta*:
$s \to_\beta t \Longrightarrow free\ t\ i \Longrightarrow free\ s\ i$
by (*induct arbitrary*: *i set*: *beta*) *auto*

lemma *beta-subst* [*intro*]: $s \to_\beta t \Longrightarrow s[u/i] \to_\beta t[u/i]$
by (*induct arbitrary*: *u i set*: *beta*) (*simp-all add*: *subst-subst* [*symmetric*])

lemma *subst-Var-Suc* [*simp*]: $t[Var\ i/i] = t[Var(i)/i + 1]$
by (*induct t arbitrary*: *i*) (*auto elim*!: *linorder-neqE simp*: *subst-Var*)

lemma *eta-lift* [*simp*]: $s \to_\eta t \Longrightarrow lift\ s\ i \to_\eta lift\ t\ i$
by (*induct arbitrary*: *i set*: *eta*) *simp-all*

lemma *rtrancl-eta-subst*: $s \to_\eta t \Longrightarrow u[s/i] \to_\eta^* u[t/i]$
apply (*induct u arbitrary*: *s t i*)
apply (*simp-all add*: *subst-Var*)
apply *blast*
apply (*blast intro*: *rtrancl-eta-App*)
apply (*blast intro*!: *rtrancl-eta-Abs eta-lift*)
done

lemma *rtrancl-eta-subst'*:
fixes *s t* :: *dB*
assumes *eta*: $s \to_\eta^* t$
shows $s[u/i] \to_\eta^* t[u/i]$ **using** *eta*
by *induct* (*iprover intro*: *eta-subst*)+

lemma *rtrancl-eta-subst''*:
fixes *s t* :: *dB*
assumes *eta*: $s \to_\eta^* t$
shows $u[s/i] \to_\eta^* u[t/i]$ **using** *eta*
by *induct* (*iprover intro*: *rtrancl-eta-subst rtranclp-trans*)+

lemma *square-beta-eta*: *square beta eta* (*eta***) (*beta*$^{==}$)
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* (*sup 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*) $\rightarrow_\eta$ *s*

lemma *not-free-iff-lifted*:
(¬ *free s i*) = (∃ *t*. *s* = *lift t i*)
apply (*induct s arbitrary*: *i*)
apply *simp*
apply (*rule iffI*)
apply (*erule linorder-neqE*)
apply (*rename-tac nat a*, *rule-tac x* = *Var nat* **in** *exI*)
apply *simp*
apply (*rename-tac nat a*, *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 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 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:
  (∀ s u. (¬ free s 0) −−> R (Abs (s ° Var 0)) (s[u/0])) =
    (∀ s. R (Abs (lift s 0 ° Var 0)) s)
  by (auto simp add: not-free-iff-lifted)
```

4.7 Eta-postponement theorem

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

```
theorem eta-case:
  fixes s :: dB
  assumes free: ¬ free s 0
  and s: s[dummy/0] => u
  shows ∃ t'. Abs (s ° Var 0) => t' ∧ t' →η* u
proof −
  from s have lift (s[dummy/0]) 0 => lift u 0 by (simp del: lift-subst)
  with free have s => lift u 0 by (simp add: lift-subst-dummy del: lift-subst)
  hence Abs (s ° Var 0) => Abs (lift u 0 ° Var 0) by simp
  moreover have ¬ free (lift u 0) 0 by simp
  hence Abs (lift u 0 ° Var 0) →η lift u 0[dummy/0]
    by (rule eta.eta)
  hence Abs (lift u 0 ° Var 0) →η* u by simp
  ultimately show ?thesis by iprover
qed

theorem eta-par-beta:
  assumes st: s →η t
```

```
  and tu: t => u
  shows ∃t'. s => t' ∧ t' →η* u using tu st
proof (induct arbitrary: s)
  case (var n)
  thus ?case by (iprover intro: par-beta-refl)
next
  case (abs s' t)
  note abs' = this
  from ‹s →η Abs s'› show ?case
  proof cases
    case (eta s'' dummy)
    from abs have Abs s' => Abs t by simp
    with eta have s''[dummy/0] => Abs t by simp
    with ‹¬ free s'' 0› have ∃t'. Abs (s'' ° Var 0) => t' ∧ t' →η* Abs t
      by (rule eta-case)
    with eta show ?thesis by simp
  next
    case (abs r)
    from ‹r →η s'›
    obtain t' where r: r => t' and t': t' →η* t by (iprover dest: abs')
    from r have Abs r => Abs t' ..
    moreover from t' have Abs t' →η* Abs t by (rule rtrancl-eta-Abs)
    ultimately show ?thesis using abs by simp iprover
  qed
next
  case (app u u' t t')
  from ‹s →η u ° t› show ?case
  proof cases
    case (eta s' dummy)
    from app have u ° t => u' ° t' by simp
    with eta have s'[dummy/0] => u' ° t' by simp
    with ‹¬ free s' 0› have ∃r. Abs (s' ° Var 0) => r ∧ r →η* u' ° t'
      by (rule eta-case)
    with eta show ?thesis by simp
  next
    case (appL s')
    from ‹s' →η u›
    obtain r where s': s' => r and r: r →η* u' by (iprover dest: app)
    from s' and app have s' ° t => r ° t' by simp
    moreover from r have r ° t' →η* u' ° t' by (simp add: rtrancl-eta-AppL)
    ultimately show ?thesis using appL by simp iprover
  next
    case (appR s')
    from ‹s' →η t›
    obtain r where s': s' => r and r: r →η* t' by (iprover dest: app)
    from s' and app have u ° s' => u' ° r by simp
    moreover from r have u' ° r →η* u' ° t' by (simp add: rtrancl-eta-AppR)
    ultimately show ?thesis using appR by simp iprover
  qed
```

```
next
  case (beta u u' t t')
  from ‹s →η Abs u ° t› show ?case
  proof cases
    case (eta s' dummy)
    from beta have Abs u ° t => u'[t'/0] by simp
    with eta have s'[dummy/0] => u'[t'/0] by simp
    with ‹¬ free s' 0› have ∃ r. Abs (s' ° Var 0) => r ∧ r →η* u'[t'/0]
      by (rule eta-case)
    with eta show ?thesis by simp
  next
    case (appL s')
    from ‹s' →η Abs u› show ?thesis
    proof cases
      case (eta s'' dummy)
      have Abs (lift u 1) = lift (Abs u) 0 by simp
      also from eta have ... = s'' by (simp add: lift-subst-dummy del: lift-subst)
      finally have s: s = Abs (Abs (lift u 1) ° Var 0) ° t using appL and eta by
simp
      from beta have lift u 1 => lift u' 1 by simp
      hence Abs (lift u 1) ° Var 0 => lift u' 1[Var 0/0]
        using par-beta.var ..
      hence Abs (Abs (lift u 1) ° Var 0) ° t => lift u' 1[Var 0/0][t'/0]
        using ‹t => t'› ..
      with s have s => u'[t'/0] by simp
      thus ?thesis by iprover
    next
      case (abs r)
      from ‹r →η u›
      obtain r'' where r: r => r'' and r'': r'' →η* u' by (iprover dest: beta)
      from r and beta have Abs r ° t => r''[t'/0] by simp
      moreover from r'' have r''[t'/0] →η* u'[t'/0]
        by (rule rtrancl-eta-subst')
      ultimately show ?thesis using abs and appL by simp iprover
    qed
  next
    case (appR s')
    from ‹s' →η t›
    obtain r where s': s' => r and r: r →η* t' by (iprover dest: beta)
    from s' and beta have Abs u ° s' => u'[r/0] by simp
    moreover from r have u'[r/0] →η* u'[t'/0]
      by (rule rtrancl-eta-subst'')
    ultimately show ?thesis using appR by simp iprover
  qed
qed

theorem eta-postponement':
  assumes eta: s →η* t and beta: t => u
  shows ∃ t'. s => t' ∧ t' →η* u using eta beta
```

proof (*induct arbitrary*: *u*)
case *base*
thus *?case* **by** *blast*
next
case (*step* s' s'' s''')
then obtain t' **where** s': $s' => t'$ **and** t': $t' \to_\eta^* s'''$
by (*auto dest*: *eta-par-beta*)
from s' **obtain** t'' **where** *s*: $s => t''$ **and** t'': $t'' \to_\eta^* t'$ **using** *step*
by *blast*
from t'' **and** t' **have** $t'' \to_\eta^* s'''$ **by** (*rule rtranclp-trans*)
with *s* **show** *?case* **by** *iprover*
qed

theorem *eta-postponement*:
assumes (*sup beta eta*)** *s t*
shows (*beta*** *OO eta***) *s t* **using** *assms*
proof *induct*
case *base*
show *?case* **by** *blast*
next
case (*step* s' s'')
from *step*(*3*) **obtain** t' **where** *s*: $s \to_\beta^* t'$ **and** t': $t' \to_\eta^* s'$ **by** *blast*
from *step*(*2*) **show** *?case*
proof
assume $s' \to_\beta s''$
with *beta-subset-par-beta* **have** $s' => s''$ **..**
with t' **obtain** t'' **where** *st*: $t' => t''$ **and** *tu*: $t'' \to_\eta^* s''$
by (*auto dest*: *eta-postponement'*)
from *par-beta-subset-beta st* **have** $t' \to_\beta^* t''$ **..**
with *s* **have** $s \to_\beta^* t''$ **by** (*rule rtranclp-trans*)
thus *?thesis* **using** *tu* **..**
next
assume $s' \to_\eta s''$
with t' **have** $t' \to_\eta^* s''$ **..**
with *s* **show** *?thesis* **..**
qed
qed

end

5 Application of a term to a list of terms

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

abbreviation
list-application :: *dB => dB list => dB* (**infixl** ‹°°› *150*) **where**
t °° *ts* == *foldl* (°) *t ts*

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

by (*induct ts rule*: *rev-induct*) *auto*

lemma *Var-eq-apps-conv* [*iff*]: (*Var m* = *s* °° *ss*) = (*Var m* = *s* ∧ *ss* = []) **by** (*induct ss arbitrary*: *s*) *auto*

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

lemma *App-eq-foldl-conv*:
(*r* ° *s* = *t* °° *ts*) =
(*if ts* = [] *then r* ° *s* = *t*
else (∃ *ss*. *ts* = *ss* @ [*s*] ∧ *r* = *t* °° *ss*))
apply (*rule-tac xs* = *ts* **in** *rev-exhaust*)
apply *auto*
done

lemma *Abs-eq-apps-conv* [*iff*]:
(*Abs r* = *s* °° *ss*) = (*Abs r* = *s* ∧ *ss* = [])
by (*induct ss rule*: *rev-induct*) *auto*

lemma *apps-eq-Abs-conv* [*iff*]: (*s* °° *ss* = *Abs r*) = (*s* = *Abs r* ∧ *ss* = [])
by (*induct ss rule*: *rev-induct*) *auto*

lemma *Abs-apps-eq-Abs-apps-conv* [*iff*]:
(*Abs r* °° *rs* = *Abs s* °° *ss*) = (*r* = *s* ∧ *rs* = *ss*)
apply (*induct rs arbitrary*: *ss rule*: *rev-induct*)
apply *simp*
apply *blast*
apply (*induct-tac ss rule*: *rev-induct*)
apply *auto*
done

lemma *Abs-App-neq-Var-apps* [*iff*]:
Abs s ° *t* ≠ *Var n* °° *ss*
by (*induct ss arbitrary*: *s t rule*: *rev-induct*) *auto*

lemma *Var-apps-neq-Abs-apps* [*iff*]:
Var n °° *ts* ≠ *Abs r* °° *ss*
apply (*induct ss arbitrary*: *ts 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 t*)
  apply (*rule-tac x* = [] **in** *exI*)
  apply *simp*
 apply *clarify*
 apply (*rename-tac ts1 ts2 h1 h2*)
 apply (*rule-tac x* = *ts1* @ [*h2* °° *ts2*] **in** *exI*)
 apply *simp*
apply *simp*
done

lemma *size-apps* [*simp*]:
 size (*r* °° *rs*) = *size r* + *foldl* (+) *0* (*map size rs*) + *length rs*
 by (*induct rs rule*: *rev-induct*) *auto*

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

lemma *lift-map* [*simp*]:
  lift (*t* °° *ts*) *i* = *lift t i* °° *map* (λ*t*. *lift t i*) *ts*
 by (*induct ts arbitrary*: *t*) *simp-all*

lemma *subst-map* [*simp*]:
  subst (*t* °° *ts*) *u i* = *subst t u i* °° *map* (λ*t*. *subst t u i*) *ts*
 by (*induct ts arbitrary*: *t*) *simp-all*

lemma *app-last*: (*t* °° *ts*) ° *u* = *t* °° (*ts* @ [*u*])
 by *simp*

A customized induction schema for °°.

lemma *lem*:
 assumes !!*n ts*. ∀ *t* ∈ *set ts*. *P t* ==> *P* (*Var n* °° *ts*)
  and !!*u ts*. [| *P u*; ∀ *t* ∈ *set ts*. *P t* |] ==> *P* (*Abs u* °° *ts*)
 shows *size t* = *n* ⟹ *P t*
 apply (*induct n arbitrary*: *t rule*: *nat-less-induct*)
 apply (*cut-tac t* = *t* **in** *ex-head-tail*)
 apply *clarify*
 apply (*erule disjE*)
  apply *clarify*
  apply (*rule assms*)
  apply *clarify*
  apply (*erule allE*, *erule impE*)
   prefer *2*
   apply (*erule allE*, *erule mp*, *rule refl*)
  apply *simp*
  apply (*simp only*: *foldl-conv-fold add.commute fold-plus-sum-list-rev*)
  apply (*fastforce simp add*: *sum-list-map-remove1*)

apply *clarify*
apply (*rule assms*)
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 (*simp only*: *foldl-conv-fold add.commute fold-plus-sum-list-rev*)
apply (*simp add*: *member-le-sum-list*)
done

theorem *Apps-dB-induct*:
assumes !!*n ts*. ∀ *t* ∈ *set ts*. *P t* ==> *P* (*Var n* °° *ts*)
and !!*u ts*. [| *P u*; ∀ *t* ∈ *set ts*. *P t* |] ==> *P* (*Abs u* °° *ts*)
shows *P t*
apply (*rule-tac t* = *t* **in** *lem*)
prefer *3*
apply (*rule refl*)
using *assms* **apply** *iprover*+
done

end

6 Simply-typed lambda terms

theory *LambdaType* **imports** *ListApplication* **begin**

6.1 Environments

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

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*]: $e\langle i{:}U\rangle\langle 0{:}T\rangle = e\langle 0{:}T\rangle\langle Suc\ i{:}U\rangle$
by (*rule ext*) (*simp-all add*: *shift-def split*: *nat.split*)

6.2 Types and typing rules

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

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

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

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

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

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

6.3 Some examples

schematic-goal *e* ⊢ *Abs* (*Abs* (*Abs* (*Var 1* ∘ (*Var 2* ∘ *Var 1* ∘ *Var 0*)))) : *?T*
by *force*

schematic-goal *e* ⊢ *Abs* (*Abs* (*Abs* (*Var 2* ∘ *Var 0* ∘ (*Var 1* ∘ *Var 0*)))) : *?T*
by *force*

6.4 Lists of types

lemma *lists-typings*:

```
    e ⊩ ts : Ts ⟹ listsp (λt. ∃ T. e ⊢ t : T) ts
proof (induct ts arbitrary: Ts)
  case Nil
  then show ?case
    by simp
next
  case c: (Cons a ts)
  show ?case
  proof (cases Ts)
    case Nil
    with c show ?thesis
      by simp
  next
    case (Cons T list)
    with c show ?thesis  by force
  qed
qed

lemma types-snoc: e ⊩ ts : Ts ⟹ e ⊢ t : T ⟹ e ⊩ ts @ [t] : Ts @ [T]
  by (induct ts arbitrary: Ts) (auto split: list.split-asm)

lemma types-snoc-eq: e ⊩ ts @ [t] : Ts @ [T] =
  (e ⊩ ts : Ts ∧ e ⊢ t : T)
proof (induct ts arbitrary: Ts)
  case Nil
  then show ?case
    by (auto split: list.split)
next
  case (Cons a ts)
  have ¬ e ⊩ ts @ [t] : []
  by (cases ts @ [t]; simp)
  with Cons show ?case
    by (auto split: list.split)
qed
```

Cannot use *rev-exhaust* from the *List* theory, since it is not constructive

```
lemma rev-exhaust2 [extraction-expand]:
  obtains (Nil) xs = []  |  (snoc) ys y where xs = ys @ [y]
proof −
  have §: xs = rev ys ⟹ thesis for ys
    by (cases ys) (simp-all add: local.Nil snoc)
  show thesis
    using § [of rev xs] by simp
qed

lemma types-snocE:
  assumes ‹e ⊩ ts @ [t] : Ts›
  obtains Us and U where ‹Ts = Us @ [U]› ‹e ⊩ ts : Us› ‹e ⊢ t : U›
proof (cases Ts rule: rev-exhaust2)
```

case *Nil*
with *assms* **show** *?thesis*
by (*cases ts* @ [*t*]) *simp-all*
next
case (*snoc Us U*)
with *assms* **have** $e \Vdash ts$ @ [*t*] : *Us* @ [*U*] **by** *simp*
then have $e \Vdash ts$: *Us* $e \vdash t$: *U* **by** (*simp-all add*: *types-snoc-eq*)
with *snoc* **show** *?thesis* **by** (*rule that*)
qed

6.5 n-ary function types

lemma *list-app-typeD*:
$e \vdash t \circ\circ ts : T \Longrightarrow \exists Ts.\ e \vdash t : Ts \Rrightarrow T \wedge e \Vdash ts : Ts$
proof (*induct ts arbitrary*: *t T*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a b t T*)
then show *?case*
by (*auto simp*: *split*: *list.split*)
qed

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

lemma *list-app-typeI*:
$e \vdash t : Ts \Rrightarrow T \Longrightarrow e \Vdash ts : Ts \Longrightarrow e \vdash t \circ\circ ts : T$
by (*induct ts arbitrary*: *t Ts*) (*auto simp add*: *split*: *list.split-asm*)

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

theorem *var-app-type-eq*:
$e \vdash Var\ i \circ\circ ts : T \Longrightarrow e \vdash Var\ i \circ\circ ts : U \Longrightarrow T = U$
by (*induct ts arbitrary*: *T U rule*: *rev-induct*) *auto*

lemma *var-app-types*: $e \vdash Var\ i \circ\circ ts \circ\circ us : T \Longrightarrow e \Vdash ts : Ts \Longrightarrow$
$e \vdash Var\ i \circ\circ ts : U \Longrightarrow \exists Us.\ U = Us \Rrightarrow T \wedge e \Vdash us : Us$
proof (*induct us arbitrary*: *ts Ts U*)
case *Nil*
then show *?case*
by (*simp add*: *var-app-type-eq*)
next
case (*Cons a b ts Ts U*)
then show *?case*
apply *atomize*
apply (*case-tac U*)

```
    apply (rule FalseE)
    apply simp
    apply (erule list-app-typeE)
    apply (ind-cases e ⊢ t ° u : T for t u T)
    apply (rename-tac nat Ts' T')
    apply (drule-tac T=Atom nat and U=T' ⇒ Ts' ⇛ T in var-app-type-eq)
     apply assumption
    apply simp
   apply (rename-tac type1 type2)
   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 ⊢ t ° u : T for t u T)
   using var-app-type-eq apply fastforce
   apply (erule impE)
    apply (rule typing.App)
     apply assumption
    apply (erule list-app-typeE)
    apply (ind-cases e ⊢ t ° u : T for t u T)
   using var-app-type-eq apply fastforce
   apply (erule exE)
   apply (rule-tac x=type1 # Us in exI)
   apply simp
   apply (erule list-app-typeE)
   apply (ind-cases e ⊢ t ° u : T for t u T)
   using var-app-type-eq by fastforce
qed

lemma var-app-typesE: e ⊢ Var i °° ts : T ⟹
  (⋀Ts. e ⊢ Var i : Ts ⇛ T ⟹ e ⊩ ts : Ts ⟹ P) ⟹ P
  by (iprover intro: typing.Var dest: var-app-types [of - - [], simplified])

lemma abs-typeE:
  assumes e ⊢ Abs t : T ⋀U V. e⟨0:U⟩ ⊢ t : V ⟹ P
  shows P
proof (cases T)
  case (Atom x1)
  with assms(1) show ?thesis
    apply−
    apply (rule FalseE)
    apply (erule typing.cases)
     apply simp-all
    done
next
```

case (*Fun type1 type2*)
with *assms* **show** *?thesis*
apply *atomize*
apply (*erule-tac x=type1* **in** *allE*)
apply (*erule-tac x=type2* **in** *allE*)
apply (*erule mp*)
apply (*erule typing.cases*)
apply *simp-all*
done
qed

6.6 Lifting preserves well-typedness

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

lemma *lift-types*:
$e \Vdash \mathit{ts} : \mathit{Ts} \Longrightarrow e\langle i{:}U\rangle \Vdash (\mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ i)\ \mathit{ts}) : \mathit{Ts}$
by (*induct ts arbitrary*: *Ts*) (*auto split*: *list.split*)

6.7 Substitution lemmas

lemma *subst-lemma*:
$e \vdash t : T \Longrightarrow e' \vdash u : U \Longrightarrow e = e'\langle i{:}U\rangle \Longrightarrow e' \vdash t[u/i] : T$
proof (*induct arbitrary*: e' *i U u set*: *typing*)
case (*Var env x T*)
then show *?case*
by (*force simp add*: *shift-def*)
next
case (*Abs env T t U*)
then show *?case* **by** *force*
qed *auto*

lemma *substs-lemma*:
$e \vdash u : T \Longrightarrow e\langle i{:}T\rangle \Vdash \mathit{ts} : \mathit{Ts} \Longrightarrow$
$e \Vdash (\mathit{map}\ (\lambda t.\ t[u/i])\ \mathit{ts}) : \mathit{Ts}$
proof (*induct ts arbitrary*: *Ts*)
case *Nil*
then show *?case*
by *auto*
next
case (*Cons a ts*)
with *subst-lemma* **show** *?case*
by (*auto split*: *list.split*)
qed

6.8 Subject reduction

lemma *subject-reduction*: $e \vdash t : T \Longrightarrow t \to_\beta t' \Longrightarrow e \vdash t' : T$
proof (*induct arbitrary*: t' *set*: *typing*)

case (*App env s T U t*)
with *subst-lemma* **show** *?case*
by *auto*
qed *auto*

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

6.9 Alternative induction rule for types

lemma *type-induct* [*induct type*]:
assumes
$(\bigwedge T.\ (\bigwedge T1\ T2.\ T = T1 \Rightarrow T2 \Longrightarrow P\ T1) \Longrightarrow$
$(\bigwedge T1\ T2.\ T = T1 \Rightarrow T2 \Longrightarrow P\ T2) \Longrightarrow P\ T)$
shows *P T*
proof (*induct T*)
case *Atom*
show *?case* **by** (*rule assms*) *simp-all*
next
case *Fun*
show *?case* **by** (*rule assms*) (*insert Fun*, *simp-all*)
qed

end

7 Lifting an order to lists of elements

theory *ListOrder*
imports *Main*
begin

declare [[*syntax-ambiguity-warning* = *false*]]

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

definition
step1 :: ('*a* => '*a* => *bool*) => '*a list* => '*a list* => *bool* **where**
step1 r =
(λ*ys xs*. ∃ *us z z' vs*. *xs* = *us* @ *z* # *vs* ∧ *r z' z* ∧ *ys* =
us @ *z'* # *vs*)

lemma *step1-converse* [*simp*]: *step1* $(r^{-1-1}) = (step1\ r)^{-1-1}$
apply (*unfold step1-def*)
apply (*blast intro*!: *order-antisym*)
done

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

lemma *not-Nil-step1* [*iff*]: ¬ *step1 r* [] *xs*
apply (*unfold step1-def*)
apply *blast*
done

lemma *not-step1-Nil* [*iff*]: ¬ *step1 r xs* []
apply (*unfold step1-def*)
apply *blast*
done

lemma *Cons-step1-Cons* [*iff*]:
(*step1 r* (*y* # *ys*) (*x* # *xs*)) =
(*r y x* ∧ *xs* = *ys* ∨ *x* = *y* ∧ *step1 r ys xs*)
apply (*unfold step1-def*)
apply (*rule iffI*)
apply (*erule exE*)
apply (*rename-tac ts*)
apply (*case-tac ts*)
apply *fastforce*
apply *force*
apply (*erule disjE*)
apply *blast*
apply (*blast intro*: *Cons-eq-appendI*)
done

lemma *append-step1I*:
step1 r ys xs ∧ *vs* = *us* ∨ *ys* = *xs* ∧ *step1 r vs us*
==> *step1 r* (*ys* @ *vs*) (*xs* @ *us*)
apply (*unfold step1-def*)
apply *auto*
apply *blast*
apply (*blast intro*: *append-eq-appendI*)
done

lemma *Cons-step1E* [*elim*!]:
assumes *step1 r ys* (*x* # *xs*)
and !!*y*. *ys* = *y* # *xs* ⟹ *r y x* ⟹ *R*
and !!*zs*. *ys* = *x* # *zs* ⟹ *step1 r zs xs* ⟹ *R*
shows *R*
using *assms*
apply (*cases ys*)
apply (*simp add*: *step1-def*)
apply *blast*
done

lemma *Snoc-step1-SnocD*:
step1 r (*ys* @ [*y*]) (*xs* @ [*x*])
==> (*step1 r ys xs* ∧ *y* = *x* ∨ *ys* = *xs* ∧ *r y x*)

apply (*unfold step1-def*)
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* [*intro*!]:
Wellfounded.accp r x ==> *Wellfounded.accp* (*step1 r*) *xs* ⟹ *Wellfounded.accp* (*step1 r*) (*x* # *xs*)
apply (*induct arbitrary*: *xs set*: *Wellfounded.accp*)
apply (*erule thin-rl*)
apply (*erule accp-induct*)
apply (*rule accp.accI*)
apply *blast*
done

lemma *lists-accD*: *listsp* (*Wellfounded.accp r*) *xs* ==> *Wellfounded.accp* (*step1 r*) *xs*
apply (*induct set*: *listsp*)
apply (*rule accp.accI*)
apply *simp*
apply (*rule accp.accI*)
apply (*fast dest*: *accp-downward*)
done

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

lemma *lists-accI*: *Wellfounded.accp* (*step1 r*) *xs* ==> *listsp* (*Wellfounded.accp r*) *xs*
apply (*induct set*: *Wellfounded.accp*)
apply *clarify*
apply (*rule accp.accI*)
apply (*drule-tac r*=*r* **in** *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.

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

lemma *head-Var-reduction*:
Var n °° *rs* →$_\beta$ *v* ⟹ ∃ *ss*. *rs* => *ss* ∧ *v* = *Var n* °° *ss*
apply (*induct u* == *Var n* °° *rs v arbitrary*: *rs set*: *beta*)
apply *simp*
apply (*rule-tac xs* = *rs* **in** *rev-exhaust*)
apply *simp*
apply (*atomize*, *force intro*: *append-step1I*)
apply (*rule-tac xs* = *rs* **in** *rev-exhaust*)
apply *simp*
apply (*auto 0 3 intro*: *disjI2* [*THEN append-step1I*])
done

lemma *apps-betasE* [*elim*!]:
assumes *major*: *r* °° *rs* →$_\beta$ *s*
and *cases*: !!*r'*. [| *r* →$_\beta$ *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*
shows *R*
proof −
from *major* **have**
(∃ *r'*. *r* →$_\beta$ *r'* ∧ *s* = *r'* °° *rs*) ∨
(∃ *rs'*. *rs* => *rs'* ∧ *s* = *r* °° *rs'*) ∨
(∃ *t u us*. *r* = *Abs t* ∧ *rs* = *u* # *us* ∧ *s* = *t*[*u*/*0*] °° *us*)
apply (*induct u* == *r* °° *rs s arbitrary*: *r rs set*: *beta*)
apply (*case-tac r*)
apply *simp*
apply (*simp add*: *App-eq-foldl-conv*)
apply (*split if-split-asm*)
apply *simp*
apply *blast*
apply *simp*
apply (*simp add*: *App-eq-foldl-conv*)
apply (*split if-split-asm*)
apply *simp*
apply *simp*
apply (*drule App-eq-foldl-conv* [*THEN iffD1*])
apply (*split if-split-asm*)
apply *simp*
apply *blast*
apply (*force intro*!: *disjI1* [*THEN append-step1I*])

```
    apply (drule App-eq-foldl-conv [THEN iffD1])
    apply (split if-split-asm)
     apply simp
     apply blast
    apply (clarify, auto 0 3 intro!: exI intro: append-step1I)
   done
 with cases show ?thesis by blast
qed

lemma apps-preserves-beta [simp]:
   r →β s ==> r °° ss →β s °° ss
 by (induct ss rule: rev-induct) auto

lemma apps-preserves-beta2 [simp]:
   r →β* s ==> r °° ss →β* s °° ss
 apply (induct set: rtranclp)
  apply blast
 apply (blast intro: apps-preserves-beta rtranclp.rtrancl-into-rtrancl)
 done

lemma apps-preserves-betas [simp]:
   rs => ss ⟹ r °° rs →β r °° ss
 apply (induct rs arbitrary: ss rule: rev-induct)
  apply simp
 apply simp
 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

```
inductive IT :: dB => bool
 where
   Var [intro]: listsp IT rs ==> IT (Var n °° rs)
 | Lambda [intro]: IT r ==> IT (Abs r)
 | Beta [intro]: IT ((r[s/0]) °° ss) ==> IT s ==> IT ((Abs r ° s) °° ss)
```

9.2 Every term in IT terminates

lemma *double-induction-lemma* [*rule-format*]:
termip beta s ==> ∀ *t. termip beta t* −−>
(∀ *r ss. t* = *r*[*s*/*0*] °° *ss* −−> *termip beta* (*Abs r* ° *s* °° *ss*))
apply (*erule accp-induct*)
apply (*rule allI*)
apply (*rule impI*)
apply (*erule thin-rl*)
apply (*erule accp-induct*)
apply *clarify*
apply (*rule accp.accI*)
apply (*safe elim*!: *apps-betasE*)
apply (*blast intro*: *subst-preserves-beta apps-preserves-beta*)
apply (*blast intro*: *apps-preserves-beta2 subst-preserves-beta2 rtranclp-converseI*
dest: *accp-downwards*)
apply (*blast dest*: *apps-preserves-betas*)
done

lemma *IT-implies-termi*: *IT t* ==> *termip beta t*
apply (*induct set*: *IT*)
apply (*drule rev-predicate1D* [*OF - listsp-mono* [**where** *B*=*termip beta*]])
apply (*fast intro*!: *predicate1I*)
apply (*drule lists-accD*)
apply (*erule accp-induct*)
apply (*rule accp.accI*)
apply (*blast dest*: *head-Var-reduction*)
apply (*erule accp-induct*)
apply (*rule accp.accI*)
apply *blast*
apply (*blast intro*: *double-induction-lemma*)
done

9.3 Every terminating term is in IT

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

lemma [*simp*, *THEN not-sym*, *simp*]: *Var n* °° *ss* ≠ *Abs r* ° *s* °° *ts*
by (*simp add*: *foldl-Cons* [*symmetric*] *del*: *foldl-Cons*)

lemma [*simp*]:
(*Abs r* ° *s* °° *ss* = *Abs r'* ° *s'* °° *ss'*) = (*r* = *r'* ∧ *s* = *s'* ∧ *ss* = *ss'*)
by (*simp add*: *foldl-Cons* [*symmetric*] *del*: *foldl-Cons*)

inductive-cases [*elim*!]:
IT (*Var n* °° *ss*)
IT (*Abs t*)
IT (*Abs r* ° *s* °° *ts*)

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

apply (*erule accp-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-tac r=beta* **in** *conversepI*)
apply (*drule-tac r=beta*$^{-1-1}$ **in** *ex-step1I*, *assumption*)
apply *clarify*
apply (*rename-tac us*)
apply (*erule-tac x = Var n* °° *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* ° *t* °° *ss* **in** *allE*)
apply *force*
done

end

10 Strong normalization for simply-typed lambda calculus

theory *StrongNorm* **imports** *LambdaType InductTermi* **begin**

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

10.1 Properties of *IT*

lemma *lift-IT* [*intro*!]: *IT t* ⟹ *IT* (*lift t i*)
apply (*induct arbitrary*: *i set*: *IT*)

apply (*simp* (*no-asm*))
apply (*rule conjI*)
apply
(*rule impI*,
rule IT.Var,
erule listsp.induct,
simp (*no-asm*),
simp (*no-asm*),
rule listsp.Cons,
blast,
assumption)+
apply *auto*
done

lemma *lifts-IT*: *listsp IT ts* $\Longrightarrow$ *listsp IT* (*map* ($\lambda t.$ *lift t 0*) *ts*)
by (*induct ts*) *auto*

lemma *subst-Var-IT*: *IT r* $\Longrightarrow$ *IT* (*r*[*Var i*/*j*])
apply (*induct arbitrary*: *i j set*: *IT*)

Case *Var*:

apply (*simp* (*no-asm*) *add*: *subst-Var*)
apply
((*rule conjI impI*)+,
rule IT.Var,
erule listsp.induct,
simp (*no-asm*),
simp (*no-asm*),
rule listsp.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*: *IT* (*Var n*)
apply (*subgoal-tac IT* (*Var n* °° []))
apply *simp*
apply (*rule IT.Var*)

```
  apply (rule listsp.Nil)
  done

lemma app-Var-IT: IT t ⟹ IT (t ° Var i)
  apply (induct set: IT)
   apply (subst app-last)
   apply (rule IT.Var)
   apply simp
   apply (rule listsp.Cons)
    apply (rule Var-IT)
   apply (rule listsp.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. IT t ⟹ e⟨i:U⟩ ⊢ t : T ⟹
    IT u ⟹ e ⊢ u : U ⟹ IT (t[u/i])
  (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 IT t
  thus ⋀e T' u i. PROP ?Q t e T' u i T
  proof induct
    fix e T' u i
    assume uIT: IT u
    assume uT: e ⊢ u : T
    {
      case (Var rs n e1 T'1 u1 i1)
      assume nT: e⟨i:T⟩ ⊢ Var n °° rs : T'
      let ?ty = λt. ∃ T'. e⟨i:T⟩ ⊢ t : T'
      let ?R = λt. ∀ e T' u i.
        e⟨i:T⟩ ⊢ t : T' ⟶ IT u ⟶ e ⊢ u : T ⟶ IT (t[u/i])
      show IT ((Var n °° rs)[u/i])
      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⟨i:T⟩ ⊢ Var n ° a °° as : T' by simp
  then obtain Ts
      where headT: e⟨i:T⟩ ⊢ Var n ° a : Ts ⇛ T'
      and argsT: e⟨i:T⟩ ⊩ as : Ts
    by (rule list-app-typeE)
  from headT obtain T''
      where varT: e⟨i:T⟩ ⊢ Var n : T'' ⇒ Ts ⇛ T'
      and argT: e⟨i:T⟩ ⊢ a : T''
    by cases simp-all
  from varT True have T: T = T'' ⇒ Ts ⇛ T'
    by cases auto
  with uT have uT': e ⊢ u : T'' ⇒ Ts ⇛ T' by simp
  from T have IT ((Var 0 °° map (λt. lift t 0)
    (map (λt. t[u/i]) as))[(u ° a[u/i])/0])
  proof (rule MI2)
    from T have IT ((lift u 0 ° Var 0)[a[u/i]/0])
    proof (rule MI1)
      have IT (lift u 0) by (rule lift-IT [OF uIT])
      thus IT (lift u 0 ° Var 0) by (rule app-Var-IT)
      show e⟨0:T''⟩ ⊢ lift u 0 ° Var 0 : Ts ⇛ T'
      proof (rule typing.App)
        show e⟨0:T''⟩ ⊢ lift u 0 : T'' ⇒ Ts ⇛ T'
          by (rule lift-type) (rule uT')
        show e⟨0:T''⟩ ⊢ 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 IT (a[u/i]) by simp
      from argT uT show e ⊢ a[u/i] : T''
        by (rule subst-lemma) simp
    qed
    thus IT (u ° a[u/i]) by simp
    from Var have listsp ?R as
      by cases (simp-all add: Cons)
    moreover from argsT have listsp ?ty as
      by (rule lists-typings)
    ultimately have listsp (λt. ?R t ∧ ?ty t) as
      by simp
    hence listsp IT (map (λt. lift t 0) (map (λt. t[u/i]) as))
      (is listsp IT (?ls as))
    proof induct
      case Nil
      show ?case by fastforce
    next
      case (Cons b bs)
      hence I: ?R b by simp
```

```
      from Cons obtain U where e⟨i:T⟩ ⊢ b : U by fast
      with uT uIT I have IT (b[u/i]) by simp
      hence IT (lift (b[u/i]) 0) by (rule lift-IT)
      hence listsp IT (lift (b[u/i]) 0 # ?ls bs)
        by (rule listsp.Cons) (rule Cons)
      thus ?case by simp
    qed
    thus IT (Var 0 °° ?ls as) by (rule IT.Var)
    have e⟨0:Ts ⇛ T'⟩ ⊢ Var 0 : Ts ⇛ T'
      by (rule typing.Var) simp
    moreover from uT argsT have e ⊨ map (λt. t[u/i]) as : Ts
      by (rule substs-lemma)
    hence e⟨0:Ts ⇛ T'⟩ ⊨ ?ls as : Ts
      by (rule lift-types)
    ultimately show e⟨0:Ts ⇛ T'⟩ ⊢ Var 0 °° ?ls as : T'
      by (rule list-app-typeI)
    from argT uT have e ⊢ a[u/i] : T''
      by (rule subst-lemma) (rule refl)
    with uT' show e ⊢ u ° a[u/i] : Ts ⇛ T'
      by (rule typing.App)
  qed
  with Cons True show ?thesis
    by (simp add: comp-def)
 qed
next
 case False
 from Var have listsp ?R rs by simp
 moreover from nT obtain Ts where e⟨i:T⟩ ⊨ rs : Ts
   by (rule list-app-typeE)
 hence listsp ?ty rs by (rule lists-typings)
 ultimately have listsp (λt. ?R t ∧ ?ty t) rs
   by simp
 hence listsp IT (map (λx. x[u/i]) rs)
 proof induct
   case Nil
   show ?case by fastforce
 next
   case (Cons a as)
   hence I: ?R a by simp
   from Cons obtain U where e⟨i:T⟩ ⊢ a : U by fast
   with uT uIT I have IT (a[u/i]) by simp
   hence listsp IT (a[u/i] # map (λt. t[u/i]) as)
     by (rule listsp.Cons) (rule Cons)
   thus ?case by simp
 qed
 with False show ?thesis by (auto simp add: subst-Var)
 qed
next
 case (Lambda r e1 T'1 u1 i1)
```

```
    assume e⟨i:T⟩ ⊢ Abs r : T'
      and ⋀e T' u i. PROP ?Q r e T' u i T
    with uIT uT show IT (Abs r[u/i])
      by fastforce
  next
    case (Beta r a as e1 T'1 u1 i1)
    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 IT (Abs (r[lift u 0/Suc i]) ∘ a[u/i] °° map (λt. t[u/i]) as)
    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 IT ((r[a/0] °° as)[u/i])
        using uIT uT by (rule SI1)
      thus IT (r[lift u 0/Suc i][a[u/i]/0] °° map (λt. t[u/i]) as)
        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 IT (a[u/i]) using uIT uT by (rule SI2)
    qed
    thus IT ((Abs r ∘ a °° as)[u/i]) by simp
  }
 qed
qed
```

10.3 Well-typed terms are strongly normalizing

```
lemma type-implies-IT:
  assumes e ⊢ t : T
  shows IT t
  using assms
proof induct
  case Var
  show ?case by (rule Var-IT)
next
  case Abs
  show ?case by (rule IT.Lambda) (rule Abs)
next
  case (App e s T U t)
  have IT ((Var 0 ∘ lift t 0)[s/0])
  proof (rule subst-type-IT)
    have IT (lift t 0) using ‹IT t› by (rule lift-IT)
    hence listsp IT [lift t 0] by (rule listsp.Cons) (rule listsp.Nil)
    hence IT (Var 0 °° [lift t 0]) 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*) (*rule App.hyps*)
ultimately show $e\langle 0{:}T \Rightarrow U\rangle \vdash \mathit{Var}\ 0 \circ \mathit{lift}\ t\ 0 : U$
by (*rule typing.App*)
show *IT s* **by** *fact*
show $e \vdash s : T \Rightarrow U$ **by** *fact*
qed
thus *?case* **by** *simp*
qed

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

end

11 Inductive characterization of lambda terms in normal form

theory *NormalForm*
imports *ListBeta*
begin

11.1 Terms in normal form

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

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

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*: $P\ x \Longrightarrow \mathit{listall}\ P\ xs \Longrightarrow \mathit{listall}\ P\ (x\ \#\ xs)$
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-all*

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

lemma *listall-app*: *listall P* (*xs* @ *ys*) = (*listall P xs* ∧ *listall P ys*)
by (*induct xs*; *intro iffI*; *simp*)

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

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*

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

lemma *listall-listsp-eq*: *listall P xs* = *listsp P xs*
by (*induct xs*) (*auto intro*: *listsp.intros*)

inductive *NF* :: *dB* ⇒ *bool*
where
App: *listall NF ts* ⟹ *NF* (*Var x* °° *ts*)
| *Abs*: *NF t* ⟹ *NF* (*Abs t*)
monos *listall-def*

lemma *nat-eq-dec*: ⋀*n*::*nat*. *m* = *n* ∨ *m* ≠ *n*
proof (*induction m*)
case *0*
then show *?case*

by (*cases n*; *simp only*: *nat.simps*; *iprover*)
next
case (*Suc m*)
then show *?case*
by (*cases n*; *simp only*: *nat.simps*; *iprover*)
qed

lemma *nat-le-dec*: $\bigwedge n::nat.\ m < n \vee \neg\ (m < n)$
proof (*induction m*)
case *0*
then show *?case*
by (*cases n*; *simp only*: *order.irrefl zero-less-Suc*; *iprover*)
next
case (*Suc m*)
then show *?case*
by (*cases n*; *simp only*: *not-less-zero Suc-less-eq*; *iprover*)
qed

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

11.2 Properties of *NF*

lemma *Var-NF*: *NF* (*Var n*)
proof −
have *NF* (*Var n* °° [])
by (*rule NF.App*) *simp*
then show *?thesis* **by** *simp*
qed

lemma *Abs-NF*:
assumes *NF*: *NF* (*Abs t* °° *ts*)
shows *ts* = [] **using** *NF*
proof *cases*
case (*App us i*)
thus *?thesis* **by** (*simp add*: *Var-apps-neq-Abs-apps* [*THEN not-sym*])
next
case (*Abs u*)
thus *?thesis* **by** *simp*
qed

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

lemma *subst-Var-NF*: *NF t* ⟹ *NF* (*t*[*Var i*/*j*])
apply (*induct arbitrary*: *i j 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 m1=x and n1=j in nat-eq-dec [THEN disjE])
   apply simp
   apply (erule NF.App)
  apply (rule-tac m1=j and n1=x in nat-le-dec [THEN disjE])
   apply (simp-all add: NF.App NF.Abs)
 done

lemma app-Var-NF: NF t ⟹ ∃ t'. t ∘ Var i →β* t' ∧ NF t'
 apply (induct set: NF)
  apply (simplesubst app-last)  — Using subst makes extraction fail
  apply (rule exI)
  apply (rule conjI)
   apply (rule rtranclp.rtrancl-refl)
  apply (rule NF.App)
  apply (drule listall-conj1)
  apply (simp add: listall-app)
  apply (rule Var-NF)
 apply (iprover intro: rtranclp.rtrancl-into-rtrancl rtranclp.rtrancl-refl beta subst-Var-NF)
 done

lemma lift-terms-NF: listall NF ts ⟹
    listall (λt. ∀ i. NF (lift t i)) ts ⟹
    listall NF (map (λt. lift t i) ts)
 by (induct ts) simp-all

lemma lift-NF: NF t ⟹ NF (lift t i)
 apply (induct arbitrary: i set: NF)
  apply (frule listall-conj1)
  apply (drule listall-conj2)
  apply (drule-tac i=i in lift-terms-NF)
   apply assumption
  apply (rule-tac m1=x and n1=i in nat-le-dec [THEN disjE])
   apply (simp-all add: NF.App NF.Abs)
 done
```

NF characterizes exactly the terms that are in normal form.

```
lemma NF-eq: NF t = (∀ t'. ¬ t →β t')
proof
 assume NF t
 then have ⋀t'. ¬ t →β t'
 proof induct
   case (App ts t)
   show ?case
   proof
```

```
    assume Var t °° ts →β t'
    then obtain rs where ts => rs
      by (iprover dest: head-Var-reduction)
    with App show False
      by (induct rs arbitrary: ts) auto
  qed
 next
  case (Abs t)
  show ?case
  proof
    assume Abs t →β t'
    then show False using Abs by cases simp-all
  qed
 qed
 then show ∀ t'. ¬ t →β t' ..
next
 assume H: ∀ t'. ¬ t →β t'
 then show NF t
 proof (induct t rule: Apps-dB-induct)
  case (1 n ts)
  then have ∀ ts'. ¬ ts => ts'
    by (iprover intro: apps-preserves-betas)
  with 1(1) have listall NF ts
    by (induct ts) auto
  then show ?case by (rule NF.App)
 next
  case (2 u ts)
  show ?case
  proof (cases ts)
    case Nil
    from 2 have ∀ u'. ¬ u →β u'
      by (auto intro: apps-preserves-beta)
    then have NF u by (rule 2)
    then have NF (Abs u) by (rule NF.Abs)
    with Nil show ?thesis by simp
  next
    case (Cons r rs)
    have Abs u ° r →β u[r/0] ..
    then have Abs u ° r °° rs →β u[r/0] °° rs
      by (rule apps-preserves-beta)
    with Cons have Abs u °° ts →β u[r/0] °° rs
      by simp
    with 2 show ?thesis by iprover
  qed
 qed
qed

end
```

12 Standardization

theory *Standardization*
imports *NormalForm*
begin

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

12.1 Standard reduction relation

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

inductive
  sred :: *dB* ⇒ *dB* ⇒ *bool* (**infixl** ‹→$_s$› *50*)
  and *sredlist* :: *dB list* ⇒ *dB list* ⇒ *bool* (**infixl** ‹[→$_s$]› *50*)
where
  s [→$_s$] *t* ≡ *listrelp* (→$_s$) *s t*
| *Var*: *rs* [→$_s$] *rs′* ⟹ *Var x* °° *rs* →$_s$ *Var x* °° *rs′*
| *Abs*: *r* →$_s$ *r′* ⟹ *ss* [→$_s$] *ss′* ⟹ *Abs r* °° *ss* →$_s$ *Abs r′* °° *ss′*
| *Beta*: *r*[*s*/*0*] °° *ss* →$_s$ *t* ⟹ *Abs r* ° *s* °° *ss* →$_s$ *t*

lemma *refl-listrelp*: ∀ *x*∈*set xs*. *R x x* ⟹ *listrelp R xs xs*
  by (*induct xs*) (*auto intro*: *listrelp.intros*)

lemma *refl-sred*: *t* →$_s$ *t*
  by (*induct t rule*: *Apps-dB-induct*) (*auto intro*: *refl-listrelp sred.intros*)

lemma *refl-sreds*: *ts* [→$_s$] *ts*
  by (*simp add*: *refl-sred refl-listrelp*)

lemma *listrelp-conj1*: *listrelp* (λ*x y*. *R x y* ∧ *S x y*) *x y* ⟹ *listrelp R x y*
  by (*erule listrelp.induct*) (*auto intro*: *listrelp.intros*)

lemma *listrelp-conj2*: *listrelp* (λ*x y*. *R x y* ∧ *S x y*) *x y* ⟹ *listrelp S x y*
  by (*erule listrelp.induct*) (*auto intro*: *listrelp.intros*)

lemma *listrelp-app*:
  assumes *xsys*: *listrelp R xs ys*
  shows *listrelp R xs′ ys′* ⟹ *listrelp R* (*xs* @ *xs′*) (*ys* @ *ys′*) **using** *xsys*
  by (*induct arbitrary*: *xs′ ys′*) (*auto intro*: *listrelp.intros*)

lemma *lemma1*:
  assumes *r*: *r* →$_s$ *r′* **and** *s*: *s* →$_s$ *s′*
  shows *r* ° *s* →$_s$ *r′* ° *s′* **using** *r*
proof *induct*
  case (*Var rs rs′ x*)
  then have *rs* [→$_s$] *rs′* **by** (*rule listrelp-conj1*)
  moreover have [*s*] [→$_s$] [*s′*] **by** (*iprover intro*: *s listrelp.intros*)

ultimately have rs @ $[s]$ $[\rightarrow_s]$ rs' @ $[s']$ **by** (*rule listrelp-app*)
hence $Var\ x \circ\circ (rs @ [s]) \rightarrow_s Var\ x \circ\circ (rs' @ [s'])$ **by** (*rule sred.Var*)
thus *?case* **by** (*simp only*: *app-last*)
next
case (*Abs r r' ss ss'*)
from *Abs*(*3*) **have** ss $[\rightarrow_s]$ ss' **by** (*rule listrelp-conj1*)
moreover have $[s]$ $[\rightarrow_s]$ $[s']$ **by** (*iprover intro*: *s listrelp.intros*)
ultimately have $ss @ [s]$ $[\rightarrow_s]$ $ss' @ [s']$ **by** (*rule listrelp-app*)
with $\langle r \rightarrow_s r' \rangle$ **have** $Abs\ r \circ\circ (ss @ [s]) \rightarrow_s Abs\ r' \circ\circ (ss' @ [s'])$
by (*rule sred.Abs*)
thus *?case* **by** (*simp only*: *app-last*)
next
case (*Beta r u ss t*)
hence $r[u/0] \circ\circ (ss @ [s]) \rightarrow_s t \circ s'$ **by** (*simp only*: *app-last*)
hence $Abs\ r \circ u \circ\circ (ss @ [s]) \rightarrow_s t \circ s'$ **by** (*rule sred.Beta*)
thus *?case* **by** (*simp only*: *app-last*)
qed

lemma *lemma1'*:
assumes *ts*: ts $[\rightarrow_s]$ ts'
shows $r \rightarrow_s r' \Longrightarrow r \circ\circ ts \rightarrow_s r' \circ\circ ts'$ **using** *ts*
by (*induct arbitrary*: *r r'*) (*auto intro*: *lemma1*)

lemma *lemma2-1*:
assumes *beta*: $t \rightarrow_\beta u$
shows $t \rightarrow_s u$ **using** *beta*
proof *induct*
case (*beta s t*)
have $Abs\ s \circ t \circ\circ [] \rightarrow_s s[t/0] \circ\circ []$ **by** (*iprover intro*: *sred.Beta refl-sred*)
thus *?case* **by** *simp*
next
case (*appL s t u*)
thus *?case* **by** (*iprover intro*: *lemma1 refl-sred*)
next
case (*appR s t u*)
thus *?case* **by** (*iprover intro*: *lemma1 refl-sred*)
next
case (*abs s t*)
hence $Abs\ s \circ\circ [] \rightarrow_s Abs\ t \circ\circ []$ **by** (*iprover intro*: *sred.Abs listrelp.Nil*)
thus *?case* **by** *simp*
qed

lemma *listrelp-betas*:
assumes *ts*: *listrelp* $(\rightarrow_\beta{}^*)$ *ts ts'*
shows $\bigwedge t\ t'.\ t \rightarrow_\beta{}^*\ t' \Longrightarrow t \circ\circ ts \rightarrow_\beta{}^*\ t' \circ\circ ts'$ **using** *ts*
by *induct auto*

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

```
  shows t →β* u using t
  by induct (auto dest: listrelp-conj2
    intro: listrelp-betas apps-preserves-beta converse-rtranclp-into-rtranclp)

lemma sred-lift:
  assumes s: s →s t
  shows lift s i →s lift t i using s
proof (induct arbitrary: i)
  case (Var rs rs' x)
  hence map (λt. lift t i) rs [→s] map (λt. lift t i) rs'
    by induct (auto intro: listrelp.intros)
  thus ?case by (cases x < i) (auto intro: sred.Var)
next
  case (Abs r r' ss ss')
  from Abs(3) have map (λt. lift t i) ss [→s] map (λt. lift t i) ss'
    by induct (auto intro: listrelp.intros)
  thus ?case by (auto intro: sred.Abs Abs)
next
  case (Beta r s ss t)
  thus ?case by (auto intro: sred.Beta)
qed

lemma lemma3:
  assumes r: r →s r'
  shows s →s s' ⟹ r[s/x] →s r'[s'/x] using r
proof (induct arbitrary: s s' x)
  case (Var rs rs' y)
  hence map (λt. t[s/x]) rs [→s] map (λt. t[s'/x]) rs'
    by induct (auto intro: listrelp.intros Var)
  moreover have Var y[s/x] →s Var y[s'/x]
  proof (cases y < x)
    case True thus ?thesis by simp (rule refl-sred)
  next
    case False
    thus ?thesis
      by (cases y = x) (auto simp add: Var intro: refl-sred)
  qed
  ultimately show ?case by simp (rule lemma1')
next
  case (Abs r r' ss ss')
  from Abs(4) have lift s 0 →s lift s' 0 by (rule sred-lift)
  hence r[lift s 0/Suc x] →s r'[lift s' 0/Suc x] by (fast intro: Abs.hyps)
  moreover from Abs(3) have map (λt. t[s/x]) ss [→s] map (λt. t[s'/x]) ss'
    by induct (auto intro: listrelp.intros Abs)
  ultimately show ?case by simp (rule sred.Abs)
next
  case (Beta r u ss t)
  thus ?case by (auto simp add: subst-subst intro: sred.Beta)
qed
```

lemma *lemma4-aux*:
assumes *rs*: *listrelp* $(\lambda t\ u.\ t \to_s u \wedge (\forall r.\ u \to_\beta r \longrightarrow t \to_s r))\ rs\ rs'$
shows $rs' \Rightarrow ss \Longrightarrow rs\ [\to_s]\ ss$ **using** *rs*
proof (*induct arbitrary*: *ss*)
case *Nil*
thus *?case* **by** *cases* (*auto intro*: *listrelp.Nil*)
next
case (*Cons x y xs ys*)
note $Cons' = Cons$
show *?case*
proof (*cases ss*)
case *Nil* **with** *Cons* **show** *?thesis* **by** *simp*
next
case (*Cons y' ys'*)
hence *ss*: $ss = y' \# ys'$ **by** *simp*
from *Cons Cons'* **have** $y \to_\beta y' \wedge ys' = ys \vee y' = y \wedge ys \Rightarrow ys'$ **by** *simp*
hence $x \# xs\ [\to_s]\ y' \# ys'$
proof
assume *H*: $y \to_\beta y' \wedge ys' = ys$
with *Cons'* **have** $x \to_s y'$ **by** *blast*
moreover from *Cons'* **have** $xs\ [\to_s]\ ys$ **by** (*iprover dest*: *listrelp-conj1*)
ultimately have $x \# xs\ [\to_s]\ y' \# ys$ **by** (*rule listrelp.Cons*)
with *H* **show** *?thesis* **by** *simp*
next
assume *H*: $y' = y \wedge ys \Rightarrow ys'$
with *Cons'* **have** $x \to_s y'$ **by** *blast*
moreover from *H* **have** $xs\ [\to_s]\ ys'$ **by** (*blast intro*: *Cons'*)
ultimately show *?thesis* **by** (*rule listrelp.Cons*)
qed
with *ss* **show** *?thesis* **by** *simp*
qed
qed

lemma *lemma4*:
assumes *r*: $r \to_s r'$
shows $r' \to_\beta r'' \Longrightarrow r \to_s r''$ **using** *r*
proof (*induct arbitrary*: r'')
case (*Var rs rs' x*)
then obtain *ss* **where** *rs*: $rs' \Rightarrow ss$ **and** r'': $r'' = Var\ x \circ\circ ss$
by (*blast dest*: *head-Var-reduction*)
from *Var*(*1*) *rs* **have** *rs* $[\to_s]$ *ss* **by** (*rule lemma4-aux*)
hence $Var\ x \circ\circ rs \to_s Var\ x \circ\circ ss$ **by** (*rule sred.Var*)
with r'' **show** *?case* **by** *simp*
next
case (*Abs r r' ss ss'*)
from ‹$Abs\ r' \circ\circ ss' \to_\beta r''$› **show** *?case*
proof
fix *s*

```
    assume r'': r'' = s °° ss'
    assume Abs r' →β s
    then obtain r''' where s: s = Abs r''' and r''': r' →β r''' by cases auto
    from r''' have r →s r''' by (blast intro: Abs)
    moreover from Abs have ss [→s] ss' by (iprover dest: listrelp-conj1)
    ultimately have Abs r °° ss →s Abs r''' °° ss' by (rule sred.Abs)
    with r'' s show Abs r °° ss →s r'' by simp
  next
    fix rs'
    assume ss' => rs'
    with Abs(3) have ss [→s] rs' by (rule lemma4-aux)
    with ‹r →s r'› have Abs r °° ss →s Abs r' °° rs' by (rule sred.Abs)
    moreover assume r'' = Abs r' °° rs'
    ultimately show Abs r °° ss →s r'' by simp
  next
    fix t u' us'
    assume ss' = u' # us'
    with Abs(3) obtain u us where
      ss: ss = u # us and u: u →s u' and us: us [→s] us'
      by cases (auto dest!: listrelp-conj1)
    have r[u/0] →s r'[u'/0] using Abs(1) and u by (rule lemma3)
    with us have r[u/0] °° us →s r'[u'/0] °° us' by (rule lemma1')
    hence Abs r ° u °° us →s r'[u'/0] °° us' by (rule sred.Beta)
    moreover assume Abs r' = Abs t and r'' = t[u'/0] °° us'
    ultimately show Abs r °° ss →s r'' using ss by simp
  qed
next
  case (Beta r s ss t)
  show ?case
    by (rule sred.Beta) (rule Beta)+
qed

lemma rtrancl-beta-sred:
  assumes r: r →β* r'
  shows r →s r' using r
  by induct (iprover intro: refl-sred lemma4)+
```

12.2 Leftmost reduction and weakly normalizing terms

```
inductive
  lred :: dB ⇒ dB ⇒ bool  (infixl ‹→l› 50)
  and lredlist :: dB list ⇒ dB list ⇒ bool  (infixl ‹[→l]› 50)
where
  s [→l] t ≡ listrelp (→l) s t
| Var: rs [→l] rs' ⟹ Var x °° rs →l Var x °° rs'
| Abs: r →l r' ⟹ Abs r →l Abs r'
| Beta: r[s/0] °° ss →l t ⟹ Abs r ° s °° ss →l t

lemma lred-imp-sred:
```

```
  assumes lred: s →l t
  shows s →s t using lred
proof induct
  case (Var rs rs' x)
  then have rs [→s] rs'
    by induct (iprover intro: listrelp.intros)+
  then show ?case by (rule sred.Var)
next
  case (Abs r r')
  from ‹r →s r'›
  have Abs r °° [] →s Abs r' °° [] using listrelp.Nil
    by (rule sred.Abs)
  then show ?case by simp
next
  case (Beta r s ss t)
  from ‹r[s/0] °° ss →s t›
  show ?case by (rule sred.Beta)
qed

inductive WN :: dB => bool
  where
    Var: listsp WN rs ⟹ WN (Var n °° rs)
  | Lambda: WN r ⟹ WN (Abs r)
  | Beta: WN ((r[s/0]) °° ss) ⟹ WN ((Abs r ° s) °° ss)

lemma listrelp-imp-listsp1:
  assumes H: listrelp (λx y. P x) xs ys
  shows listsp P xs using H
  by induct auto

lemma listrelp-imp-listsp2:
  assumes H: listrelp (λx y. P y) xs ys
  shows listsp P ys using H
  by induct auto

lemma lemma5:
  assumes lred: r →l r'
  shows WN r and NF r' using lred
  by induct
    (iprover dest: listrelp-conj1 listrelp-conj2
     listrelp-imp-listsp1 listrelp-imp-listsp2 intro: WN.intros
     NF.intros [simplified listall-listsp-eq])+

lemma lemma6:
  assumes wn: WN r
  shows ∃r'. r →l r' using wn
proof induct
  case (Var rs n)
  then have ∃rs'. rs [→l] rs'
```

```
    by induct (iprover intro: listrelp.intros)+
  then show ?case by (iprover intro: lred.Var)
qed (iprover intro: lred.intros)+

lemma lemma7:
  assumes r: r →s r'
  shows NF r' ⟹ r →l r' using r
proof induct
  case (Var rs rs' x)
  from ‹NF (Var x °° rs')› have listall NF rs'
    by cases simp-all
  with Var(1) have rs [→l] rs'
  proof induct
    case Nil
    show ?case by (rule listrelp.Nil)
  next
    case (Cons x y xs ys)
    hence x →l y and xs [→l] ys by simp-all
    thus ?case by (rule listrelp.Cons)
  qed
  thus ?case by (rule lred.Var)
next
  case (Abs r r' ss ss')
  from ‹NF (Abs r' °° ss')›
  have ss': ss' = [] by (rule Abs-NF)
  from Abs(3) have ss: ss = [] using ss'
    by cases simp-all
  from ss' Abs have NF (Abs r') by simp
  hence NF r' by cases simp-all
  with Abs have r →l r' by simp
  hence Abs r →l Abs r' by (rule lred.Abs)
  with ss ss' show ?case by simp
next
  case (Beta r s ss t)
  hence r[s/0] °° ss →l t by simp
  thus ?case by (rule lred.Beta)
qed

lemma WN-eq: WN t = (∃ t'. t →β* t' ∧ NF t')
proof
  assume WN t
  then have ∃ t'. t →l t' by (rule lemma6)
  then obtain t' where t': t →l t' ..
  then have NF: NF t' by (rule lemma5)
  from t' have t →s t' by (rule lred-imp-sred)
  then have t →β* t' by (rule lemma2-2)
  with NF show ∃ t'. t →β* t' ∧ NF t' by iprover
next
  assume ∃ t'. t →β* t' ∧ NF t'
```

then obtain t' **where** t': $t \to_\beta^* t'$ **and** NF: $NF\ t'$
by *iprover*
from t' **have** $t \to_s t'$ **by** (*rule rtrancl-beta-sred*)
then have $t \to_l t'$ **using** NF **by** (*rule lemma7*)
then show $WN\ t$ **by** (*rule lemma5*)
qed

end

13 Weak normalization for simply-typed lambda calculus

theory *WeakNorm*
imports *LambdaType NormalForm HOL−Library.Realizers HOL−Library.Code-Target-Int*
begin

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

13.1 Main theorems

lemma *norm-list*:
assumes *f-compat*: $\bigwedge t\ t'.\ t \to_\beta^* t' \Longrightarrow f\ t \to_\beta^* f\ t'$
and *f-NF*: $\bigwedge t.\ NF\ t \Longrightarrow NF\ (f\ t)$
and *uNF*: $NF\ u$ **and** *uT*: $e \vdash u : T$
shows $\bigwedge Us.\ e\langle i{:}T\rangle \Vdash as : Us \Longrightarrow$
$listall\ (\lambda t.\ \forall e\ T'\ u\ i.\ e\langle i{:}T\rangle \vdash t : T' \longrightarrow$
$NF\ u \longrightarrow e \vdash u : T \longrightarrow (\exists t'.\ t[u/i] \to_\beta^* t' \wedge NF\ t'))\ as \Longrightarrow$
$\exists as'.\ \forall j.\ Var\ j \circ\circ map\ (\lambda t.\ f\ (t[u/i]))\ as \to_\beta^*$
$Var\ j \circ\circ map\ f\ as' \wedge NF\ (Var\ j \circ\circ map\ f\ as')$
(**is** $\bigwedge Us.\ \text{-} \Longrightarrow listall\ ?R\ as \Longrightarrow \exists as'.\ ?ex\ Us\ as\ as'$)
proof (*induct as rule*: *rev-induct*)
case (*Nil Us*)
with *Var-NF* **have** *?ex Us* [] [] **by** *simp*
thus *?case* **..**
next
case (*snoc b bs Us*)
have $e\langle i{:}T\rangle \Vdash bs\ @\ [b] : Us$ **by** *fact*
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-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*: $Var\ j \circ\circ map\ (\lambda t.\ f\ (t[u/i]))\ bs \to_\beta^* Var\ j \circ\circ map\ f\ bs'$
and *bsNF*: $NF\ (Var\ j \circ\circ map\ f\ bs')$ **for** j
by *iprover*
from *snoc* **have** *?R b* **by** *simp*

with *bT* **and** *uNF* **and** *uT* **have** $\exists\, b'.\ b[u/i] \to_\beta{}^*\ b' \wedge NF\ b'$
by *iprover*
then obtain *b'* **where** *bred*: $b[u/i] \to_\beta{}^*\ b'$ **and** *bNF*: $NF\ b'$
by *iprover*
from *bsNF* [*of 0*] **have** *listall NF* (*map f bs'*)
by (*rule App-NF-D*)
moreover have *NF* (*f b'*) **using** *bNF* **by** (*rule f-NF*)
ultimately have *listall NF* (*map f* (*bs'* @ [*b'*]))
by *simp*
hence $\bigwedge j.$ *NF* (*Var j* °° *map f* (*bs'* @ [*b'*])) **by** (*rule NF.App*)
moreover from *bred* **have** *f* (*b*[*u*/*i*]) $\to_\beta{}^*$ *f b'*
by (*rule f-compat*)
with *bsred* **have**
$\bigwedge j.$ (*Var j* °° *map* (λ*t*. *f* (*t*[*u*/*i*])) *bs*) ∘ *f* (*b*[*u*/*i*]) $\to_\beta{}^*$
(*Var j* °° *map f bs'*) ∘ *f b'* **by** (*rule rtrancl-beta-App*)
ultimately have *?ex Us* (*bs* @ [*b*]) (*bs'* @ [*b'*]) **by** *simp*
thus *?case* **..**
qed

lemma *subst-type-NF*:
$\bigwedge t\ e\ T\ u\ i.$ *NF t* $\Longrightarrow$ $e\langle i{:}U\rangle \vdash t : T \Longrightarrow$ *NF u* $\Longrightarrow$ $e \vdash u : U \Longrightarrow \exists\, t'.\ t[u/i] \to_\beta{}^*\ t' \wedge NF\ t'$
(**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* = λ*t*. ∀ *e T' u i*.
$e\langle i{:}T\rangle \vdash t : T' \longrightarrow$ *NF u* $\longrightarrow$ $e \vdash u : T \longrightarrow (\exists\, t'.\ t[u/i] \to_\beta{}^*\ t' \wedge NF\ t')$
assume *MI1*: $\bigwedge$*T1 T2*. *T* = *T1* ⇒ *T2* $\Longrightarrow$ *PROP ?P T1*
assume *MI2*: $\bigwedge$*T1 T2*. *T* = *T1* ⇒ *T2* $\Longrightarrow$ *PROP ?P T2*
assume *NF t*
thus $\bigwedge e\ T'\ u\ i.$ *PROP ?Q t e T' u i T*
proof *induct*
fix *e T' u i* **assume** *uNF*: *NF u* **and** *uT*: $e \vdash u : T$
{
case (*App ts x e1 T'1 u1 i1*)
assume $e\langle i{:}T\rangle \vdash$ *Var x* °° *ts* : *T'*
then obtain *Us*
where *varT*: $e\langle i{:}T\rangle \vdash$ *Var x* : *Us* ⇛ *T'*
and *argsT*: $e\langle i{:}T\rangle \Vdash$ *ts* : *Us*
by (*rule var-app-typesE*)
from *nat-eq-dec* **show** $\exists\, t'.$ (*Var x* °° *ts*)[*u*/*i*] $\to_\beta{}^*\ t' \wedge NF\ t'$
proof
assume *eq*: *x* = *i*
show *?thesis*
proof (*cases ts*)
case *Nil*
with *eq* **have** (*Var x* °° [])[*u*/*i*] $\to_\beta{}^*$ *u* **by** *simp*
with *Nil* **and** *uNF* **show** *?thesis* **by** *simp iprover*
next

case (*Cons a as*)
with *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 Us Cons* **have** *argsT'*: $e\langle i{:}T\rangle \Vdash as : Ts$ **by** *simp*
from *argsT Us Cons* **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-lemma*)
from *App* **and** *Cons* **have** *listall ?R as* **by** *simp* (*iprover dest*: *listall-conj2*)
with *lift-preserves-beta' lift-NF uNF uT argsT'*
have $\exists as'.\ \forall j.\ Var\ j \circ\circ map\ (\lambda t.\ lift\ (t[u/i])\ 0)\ as \rightarrow_\beta^*$
$Var\ j \circ\circ map\ (\lambda t.\ lift\ t\ 0)\ as' \wedge$
$NF\ (Var\ j \circ\circ map\ (\lambda t.\ lift\ t\ 0)\ as')$ **by** (*rule norm-list*)
then obtain *as'* **where**
asred: $Var\ 0 \circ\circ map\ (\lambda t.\ lift\ (t[u/i])\ 0)\ as \rightarrow_\beta^*$
$Var\ 0 \circ\circ map\ (\lambda t.\ lift\ t\ 0)\ as'$
and *asNF*: $NF\ (Var\ 0 \circ\circ map\ (\lambda t.\ lift\ t\ 0)\ as')$ **by** *iprover*
from *App* **and** *Cons* **have** *?R a* **by** *simp*
with *argT* **and** *uNF* **and** *uT* **have** $\exists a'.\ a[u/i] \rightarrow_\beta^*\ a' \wedge NF\ a'$
by *iprover*
then obtain a' **where** *ared*: $a[u/i] \rightarrow_\beta^*\ a'$ **and** *aNF*: $NF\ a'$ **by** *iprover*
from *uNF* **have** $NF\ (lift\ u\ 0)$ **by** (*rule lift-NF*)
hence $\exists u'.\ lift\ u\ 0 \circ Var\ 0 \rightarrow_\beta^*\ u' \wedge NF\ u'$ **by** (*rule app-Var-NF*)
then obtain u' **where** *ured*: $lift\ u\ 0 \circ Var\ 0 \rightarrow_\beta^*\ u'$ **and** *u'NF*: $NF\ u'$
by *iprover*
from *T* **and** *u'NF* **have** $\exists ua.\ u'[a'/0] \rightarrow_\beta^*\ ua \wedge NF\ ua$
proof (*rule MI1*)
have $e\langle 0{:}T''\rangle \vdash lift\ u\ 0 \circ Var\ 0 : Ts \Rrightarrow T'$
proof (*rule typing.App*)
from *uT'* **show** $e\langle 0{:}T''\rangle \vdash lift\ u\ 0 : T'' \Rightarrow Ts \Rrightarrow T'$ **by** (*rule lift-type*)
show $e\langle 0{:}T''\rangle \vdash Var\ 0 : T''$ **by** (*rule typing.Var*) *simp*
qed
with *ured* **show** $e\langle 0{:}T''\rangle \vdash u' : Ts \Rrightarrow T'$ **by** (*rule subject-reduction'*)
from *ared aT* **show** $e \vdash a' : T''$ **by** (*rule subject-reduction'*)
show $NF\ a'$ **by** *fact*
qed
then obtain *ua* **where** *uared*: $u'[a'/0] \rightarrow_\beta^*\ ua$ **and** *uaNF*: $NF\ ua$
by *iprover*
from *ared* **have** $(lift\ u\ 0 \circ Var\ 0)[a[u/i]/0] \rightarrow_\beta^*\ (lift\ u\ 0 \circ Var\ 0)[a'/0]$
by (*rule subst-preserves-beta2'*)
also from *ured* **have** $(lift\ u\ 0 \circ Var\ 0)[a'/0] \rightarrow_\beta^*\ u'[a'/0]$
by (*rule subst-preserves-beta'*)
also note *uared*
finally have $(lift\ u\ 0 \circ Var\ 0)[a[u/i]/0] \rightarrow_\beta^*\ ua$ **.**
hence *uared'*: $u \circ a[u/i] \rightarrow_\beta^*\ ua$ **by** *simp*
from *T asNF - uaNF* **have** $\exists r.\ (Var\ 0 \circ\circ map\ (\lambda t.\ lift\ t\ 0)\ as')[ua/0] \rightarrow_\beta^*\ r \wedge NF\ r$

```
    proof (rule MI2)
      have e⟨0:Ts ⇛ T'⟩ ⊢ Var 0 °° map (λt. lift (t[u/i]) 0) as : T'
      proof (rule list-app-typeI)
        show e⟨0:Ts ⇛ T'⟩ ⊢ Var 0 : Ts ⇛ T' by (rule typing.Var) simp
        from uT argsT' have e ⊩ map (λt. t[u/i]) as : Ts
          by (rule substs-lemma)
        hence e⟨0:Ts ⇛ T'⟩ ⊩ map (λt. lift t 0) (map (λt. t[u/i]) as) : Ts
          by (rule lift-types)
        thus e⟨0:Ts ⇛ T'⟩ ⊩ map (λt. lift (t[u/i]) 0) as : Ts
          by (simp-all add: o-def)
      qed
      with asred show e⟨0:Ts ⇛ T'⟩ ⊢ Var 0 °° map (λt. lift t 0) as' : T'
        by (rule subject-reduction')
      from argT uT refl have e ⊢ a[u/i] : T'' by (rule subst-lemma)
      with uT' have e ⊢ u ° a[u/i] : Ts ⇛ T' by (rule typing.App)
      with uared' show e ⊢ ua : Ts ⇛ T' by (rule subject-reduction')
    qed
    then obtain r where rred: (Var 0 °° map (λt. lift t 0) as')[ua/0] →β* r
      and rnf: NF r by iprover
    from asred have
      (Var 0 °° map (λt. lift (t[u/i]) 0) as)[u ° a[u/i]/0] →β*
      (Var 0 °° map (λt. lift t 0) as')[u ° a[u/i]/0]
      by (rule subst-preserves-beta')
    also from uared' have (Var 0 °° map (λt. lift t 0) as')[u ° a[u/i]/0] →β*
      (Var 0 °° map (λt. lift t 0) as')[ua/0] by (rule subst-preserves-beta2')
    also note rred
    finally have (Var 0 °° map (λt. lift (t[u/i]) 0) as)[u ° a[u/i]/0] →β* r .
    with rnf Cons eq show ?thesis
      by (simp add: o-def) iprover
  qed
next
  assume neq: x ≠ i
  from App have listall ?R ts by (iprover dest: listall-conj2)
  with uNF uT argsT
  have ∃ts'. ∀j. Var j °° map (λt. t[u/i]) ts →β* Var j °° ts' ∧
    NF (Var j °° ts') (is ∃ts'. ?ex ts')
    by (rule norm-list [of λt. t, simplified])
  then obtain ts' where NF: ?ex 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
next
  case (Abs r e1 T'1 u1 i1)
```

```
    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 NF (lift u 0) using ‹NF u› by (rule lift-NF)
    moreover have e⟨0:R⟩ ⊢ lift u 0 : T using uT by (rule lift-type)
    ultimately have ∃ t'. r[lift u 0/Suc i] →β* t' ∧ NF t' by (rule Abs)
    thus ∃ t'. Abs r[u/i] →β* t' ∧ NF t'
      by simp (iprover intro: rtrancl-beta-Abs NF.Abs)
  }
 qed
qed


— A computationally relevant copy of e ⊢ t : T
inductive rtyping :: (nat ⇒ type) ⇒ dB ⇒ type ⇒ bool  (‹- ⊢R - : -› [50, 50, 50]
50)
  where
    Var: e x = T ⟹ e ⊢R Var x : T
  | Abs: e⟨0:T⟩ ⊢R t : U ⟹ e ⊢R Abs t : (T ⇒ U)
  | App: e ⊢R s : T ⇒ U ⟹ e ⊢R t : T ⟹ e ⊢R (s ∘ t) : U

lemma rtyping-imp-typing: e ⊢R t : T ⟹ e ⊢ 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 e ⊢R t : T
  shows ∃ t'. t →β* t' ∧ NF t' using assms
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 e s T U t)
  from App obtain s' t' where
    sred: s →β* s' and NF s'
    and tred: t →β* t' and tNF: NF t' by iprover
  have ∃ u. (Var 0 ∘ lift t' 0)[s'/0] →β* u ∧ NF u
  proof (rule subst-type-NF)
    have NF (lift t' 0) using tNF by (rule lift-NF)
    hence listall NF [lift t' 0] by (rule listall-cons) (rule listall-nil)
    hence NF (Var 0 °° [lift t' 0]) by (rule NF.App)
    thus NF (Var 0 ∘ lift t' 0) 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, rule App.hyps)
      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, rule App.hyps)
    show NF s' by fact
  qed
  then obtain u where ured: s' ∘ t' →β* u and unf: NF u 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 rtranclp-trans)
  with unf show ?case by iprover
qed
```

13.2 Extracting the program

```
declare NF.induct [ind-realizer]
declare rtranclp.induct [ind-realizer irrelevant]
declare rtyping.induct [ind-realizer]
lemmas [extraction-expand] = conj-assoc listall-cons-eq subst-all equal-allI

extract type-NF

lemma rtranclR-rtrancl-eq: rtranclpR r a b = r** a b
proof
  show rtranclpR r a b ⟹ r** a b
    apply (erule rtranclpR.induct)
     apply (rule rtranclp.rtrancl-refl)
    apply (metis rtranclp.rtrancl-into-rtrancl)
    done
  show r** a b ⟹ rtranclpR r a b
    apply (erule rtranclp.induct)
     apply (rule rtranclpR.rtrancl-refl)
    apply (metis rtranclpR.rtrancl-into-rtrancl)
    done
qed

lemma NFR-imp-NF: NFR nf t ⟹ NF t
  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 per-

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

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

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

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

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

type-NF ≡
λH. rec-rtypingT (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.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

forms substitution and normalization, is shown in Figure 1. The correctness theorem corresponding to the program *subst-type-NF* is

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

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

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

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

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

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

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

13.3 Generating executable code

instantiation *NFT* :: *default*
begin

definition *default* = *Dummy* ()

instance ..

end

instantiation *dB* :: *default*
begin

definition *default* = *dB.Var 0*

instance ..

end

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

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

instance ..

end

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

definition *default* = []

instance ..

end

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

definition *default* = (λ*x*. *default*)

instance ..

end

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

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

ML ‹
```
val nat-of-integer = @{code nat} o @{code int-of-integer};

fun dBtype-of-typ (Type (fun, [T, U])) =
      @{code Fun} (dBtype-of-typ T, dBtype-of-typ U)
  | dBtype-of-typ (TFree (s, -)) = (case raw-explode s of
        [', a] => @{code Atom} (nat-of-integer (ord a - 97))
      | - => error dBtype-of-typ: variable name)
  | dBtype-of-typ - = error dBtype-of-typ: bad type;

fun dB-of-term (Bound i) = @{code dB.Var} (nat-of-integer i)
  | dB-of-term (t $ u) = @{code dB.App} (dB-of-term t, dB-of-term u)
  | dB-of-term (Abs (-, -, t)) = @{code dB.Abs} (dB-of-term t)
  | dB-of-term - = error dB-of-term: bad term;

fun term-of-dB Ts (Type (fun, [T, U])) (@{code dB.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 (@{code dB.Var} n) = Bound (@{code integer-of-nat} n)
  | term-of-dB' Ts (@{code dB.App} (dBt, dBu)) =
      let val t = term-of-dB' Ts dBt
      in case fastype-of1 (Ts, t) of
          Type (fun, [T, -]) => 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) =
      @{code Var} (e, nat-of-integer i, dBtype-of-typ (nth Ts i))
  | typing-of-term Ts e (t $ u) = (case fastype-of1 (Ts, t) of
        Type (fun, [T, U]) => @{code 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 (-, T, t)) =
      let val dBT = dBtype-of-typ T
      in @{code Abs} (e, dBT, dB-of-term t,
        dBtype-of-typ (fastype-of1 (T :: Ts, t)),
        typing-of-term (T :: Ts) (@{code shift} e @{code 0::nat} dBT) t)
      end
  | typing-of-term - - - = error typing-of-term: bad term;

fun dummyf - = error dummy;

val ct1 = @{cterm %f. ((%f x. f (f (f x))) ((%f x. f (f (f (f x)))) f))};
val (dB1, -) = @{code type-NF} (typing-of-term [] dummyf (Thm.term-of ct1));
val ct1' = Thm.cterm-of @{context} (term-of-dB [] (Thm.typ-of-cterm ct1) dB1);

val ct2 = @{cterm %f x. (%x. f x x) ((%x. f x x) ((%x. f x x) ((%x. f
x x) ((%x. f x x) x)))))};
val (dB2, -) = @{code type-NF} (typing-of-term [] dummyf (Thm.term-of ct2));
val ct2' = Thm.cterm-of @{context} (term-of-dB [] (Thm.typ-of-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] R. Loader. Notes on Simply Typed Lambda Calculus. Technical Report ECS-LFCS-98-381, Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh, 1998.

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

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