

Fundamental Properties of Lambda-calculus

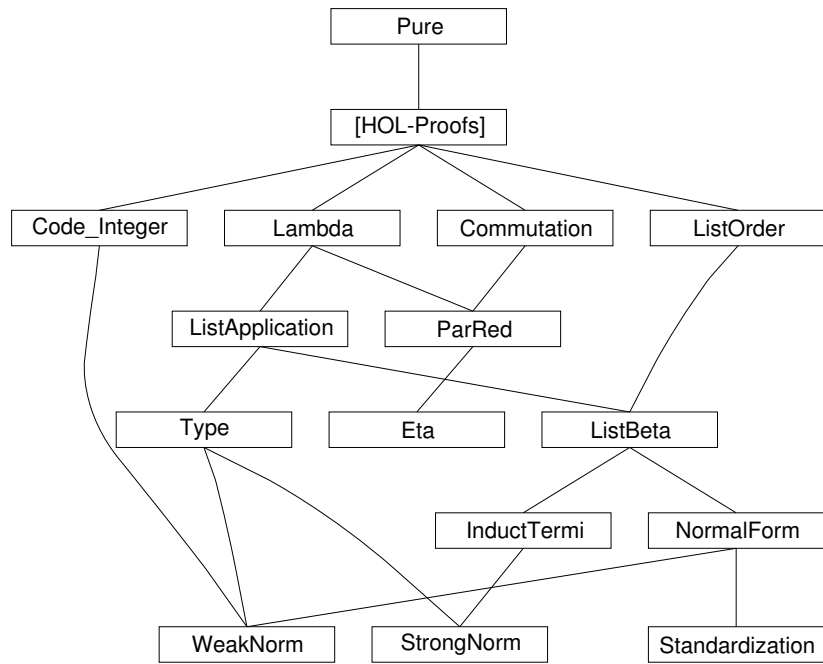
Tobias Nipkow
Stefan Berghofer

June 21, 2010

Contents

1	Basic definitions of Lambda-calculus	4
1.1	Lambda-terms in de Bruijn notation and substitution	4
1.2	Beta-reduction	4
1.3	Congruence rules	5
1.4	Substitution-lemmas	5
1.5	Equivalence proof for optimized substitution	6
1.6	Preservation theorems	6
2	Abstract commutation and confluence notions	7
2.1	Basic definitions	7
2.2	Basic lemmas	7
2.3	Church-Rosser	8
2.4	Newman's lemma	9
3	Parallel reduction and a complete developments	9
3.1	Parallel reduction	9
3.2	Inclusions	10
3.3	Misc properties of <i>par-beta</i>	10
3.4	Confluence (directly)	10
3.5	Complete developments	10
3.6	Confluence (via complete developments)	11
4	Eta-reduction	11
4.1	Definition of eta-reduction and relatives	11
4.2	Properties of <i>eta</i> , <i>subst</i> and <i>free</i>	12
4.3	Confluence of <i>eta</i>	12
4.4	Congruence rules for <i>eta</i> [*]	12
4.5	Commutation of <i>beta</i> and <i>eta</i>	13
4.6	Implicit definition of <i>eta</i>	13
4.7	Eta-postponement theorem	14

5	Application of a term to a list of terms	14
6	Simply-typed lambda terms	16
6.1	Environments	16
6.2	Types and typing rules	16
6.3	Some examples	17
6.4	Lists of types	17
6.5	n-ary function types	18
6.6	Lifting preserves well-typedness	19
6.7	Substitution lemmas	19
6.8	Subject reduction	19
6.9	Alternative induction rule for types	19
7	Lifting an order to lists of elements	19
8	Lifting beta-reduction to lists	21
9	Inductive characterization of terminating lambda terms	21
9.1	Terminating lambda terms	22
9.2	Every term in IT terminates	22
9.3	Every terminating term is in IT	22
10	Strong normalization for simply-typed lambda calculus	22
10.1	Properties of IT	23
10.2	Well-typed substitution preserves termination	23
10.3	Well-typed terms are strongly normalizing	23
11	Inductive characterization of lambda terms in normal form	23
11.1	Terms in normal form	23
11.2	Properties of NF	25
12	Standardization	25
12.1	Standard reduction relation	26
12.2	Leftmost reduction and weakly normalizing terms	27
13	Weak normalization for simply-typed lambda calculus	28
13.1	Main theorems	28
13.2	Extracting the program	29
13.3	Generating executable code	32



1 Basic definitions of Lambda-calculus

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

1.1 Lambda-terms in de Bruijn notation and substitution

datatype *dB* =

Var nat
| *App dB dB* (**infixl** \circ 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* \circ *t*) *k* = *lift s k* \circ *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* \circ *u*)[*s/k*] = *t[s/k]* \circ *u[s/k]*
| *subst-Abs*: (*Abs t*)[*s/k*] = *Abs (t[lift s 0 / k+1])*

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

Optimized versions of *subst* and *lift*.

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* \circ *t*) *k* = *liftn n s k* \circ *liftn n t k*
| *liftn n* (*Abs s*) *k* = *Abs (liftn n s (k + 1))*

primrec

substn :: [*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* \circ *u*) *s k* = *substn t s k* \circ *substn u s k*
| *substn* (*Abs t*) *s k* = *Abs (substn t s (k + 1))*

1.2 Beta-reduction

inductive *beta* :: [*dB*, *dB*] => *bool* (**infixl** \rightarrow_β 50)

where

beta [*simp*, *intro!*]: *Abs s* \circ *t* \rightarrow_β *s[t/0]*
| *appL* [*simp*, *intro!*]: *s* \rightarrow_β *t* \implies *s* \circ *u* \rightarrow_β *t* \circ *u*

| *appR* [*simp*, *intro!*]: $s \rightarrow_{\beta} t \implies u \circ s \rightarrow_{\beta} u \circ t$
| *abs* [*simp*, *intro!*]: $s \rightarrow_{\beta} t \implies \text{Abs } s \rightarrow_{\beta} \text{Abs } t$

abbreviation

beta-reds :: [*dB*, *dB*] => *bool* (**infixl** ->> 50) **where**
s ->> *t* == *beta* ^* *s* *t*

notation (*latex*)

beta-reds (**infixl** \rightarrow_{β}^* 50)

inductive-cases *beta-cases* [*elim!*]:

Var *i* \rightarrow_{β} *t*
Abs *r* \rightarrow_{β} *s*
s \circ *t* \rightarrow_{β} *u*

declare *if-not-P* [*simp*] *not-less-eq* [*simp*]
— don't add *r-into-rtranc1*[*intro!*]

1.3 Congruence rules

lemma *rtranc1-beta-Abs* [*intro!*]:

$s \rightarrow_{\beta}^* s' \implies \text{Abs } s \rightarrow_{\beta}^* \text{Abs } s'$
<proof>

lemma *rtranc1-beta-AppL*:

$s \rightarrow_{\beta}^* s' \implies s \circ t \rightarrow_{\beta}^* s' \circ t$
<proof>

lemma *rtranc1-beta-AppR*:

$t \rightarrow_{\beta}^* t' \implies s \circ t \rightarrow_{\beta}^* s \circ t'$
<proof>

lemma *rtranc1-beta-App* [*intro*]:

$[[s \rightarrow_{\beta}^* s'; t \rightarrow_{\beta}^* t']] \implies s \circ t \rightarrow_{\beta}^* s' \circ t'$
<proof>

1.4 Substitution-lemmas

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

<proof>

lemma *subst-gt* [*simp*]: $i < j \implies (\text{Var } j)[u/i] = \text{Var } (j - 1)$

<proof>

lemma *subst-lt* [*simp*]: $j < i \implies (\text{Var } j)[u/i] = \text{Var } j$

<proof>

lemma *lift-lift*:

$i < k + 1 \implies \text{lift } (\text{lift } t \ i) \ (\text{Suc } k) = \text{lift } (\text{lift } t \ k) \ i$
<proof>

lemma *lift-subst [simp]*:

$$j < i + 1 \implies \text{lift } (t[s/j]) \ i = (\text{lift } t \ (i + 1)) \ [\text{lift } s \ i \ / \ j]$$

<proof>

lemma *lift-subst-lt*:

$$i < j + 1 \implies \text{lift } (t[s/j]) \ i = (\text{lift } t \ i) \ [\text{lift } s \ i \ / \ j + 1]$$

<proof>

lemma *subst-lift [simp]*:

$$(\text{lift } t \ k)[s/k] = t$$

<proof>

lemma *subst-subst*:

$$i < j + 1 \implies t[\text{lift } v \ i \ / \ \text{Suc } j][u[v/j]/i] = t[u/i][v/j]$$

<proof>

1.5 Equivalence proof for optimized substitution

lemma *liftn-0 [simp]*: $\text{liftn } 0 \ t \ k = t$

<proof>

lemma *liftn-lift [simp]*: $\text{liftn } (\text{Suc } n) \ t \ k = \text{lift } (\text{liftn } n \ t \ k) \ k$

<proof>

lemma *substn-subst-n [simp]*: $\text{substn } t \ s \ n = t[\text{liftn } n \ s \ 0 \ / \ n]$

<proof>

theorem *substn-subst-0*: $\text{substn } t \ s \ 0 = t[s/0]$

<proof>

1.6 Preservation theorems

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

theorem *subst-preserves-beta [simp]*:

$$r \rightarrow_{\beta} s \implies r[t/i] \rightarrow_{\beta} s[t/i]$$

<proof>

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

<proof>

theorem *lift-preserves-beta [simp]*:

$$r \rightarrow_{\beta} s \implies \text{lift } r \ i \rightarrow_{\beta} \text{lift } s \ i$$

<proof>

theorem *lift-preserves-beta'*: $r \rightarrow_{\beta}^* s \implies \text{lift } r \ i \rightarrow_{\beta}^* \text{lift } s \ i$

<proof>

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

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

end

2 Abstract commutation and confluence notions

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

2.1 Basic definitions

definition

square :: [$'a \Rightarrow 'a \Rightarrow \text{bool}$, $'a \Rightarrow 'a \Rightarrow \text{bool}$, $'a \Rightarrow 'a \Rightarrow \text{bool}$, $'a \Rightarrow 'a \Rightarrow \text{bool}$] $\Rightarrow \text{bool}$ **where**
square $R\ S\ T\ U =$
 $(\forall x\ y. R\ x\ y \longrightarrow (\forall z. S\ x\ z \longrightarrow (\exists u. T\ y\ u \wedge U\ z\ u)))$

definition

commute :: [$'a \Rightarrow 'a \Rightarrow \text{bool}$, $'a \Rightarrow 'a \Rightarrow \text{bool}$] $\Rightarrow \text{bool}$ **where**
commute $R\ S = \text{square}\ R\ S\ S\ R$

definition

diamond :: ($'a \Rightarrow 'a \Rightarrow \text{bool}$) $\Rightarrow \text{bool}$ **where**
diamond $R = \text{commute}\ R\ R$

definition

Church-Rosser :: ($'a \Rightarrow 'a \Rightarrow \text{bool}$) $\Rightarrow \text{bool}$ **where**
Church-Rosser $R =$
 $(\forall x\ y. (\sup R\ (R^{\hat{\ } - - 1}))^{\hat{\ } **} x\ y \longrightarrow (\exists z. R^{\hat{\ } **} x\ z \wedge R^{\hat{\ } **} y\ z))$

abbreviation

confluent :: ($'a \Rightarrow 'a \Rightarrow \text{bool}$) $\Rightarrow \text{bool}$ **where**
confluent $R == \text{diamond}\ (R^{\hat{\ } **})$

2.2 Basic lemmas

square

lemma *square-sym*: $\text{square}\ R\ S\ T\ U \implies \text{square}\ S\ R\ U\ T$
 ⟨*proof*⟩

lemma *square-subset*:

$[[\text{square}\ R\ S\ T\ U; T \leq T']] \implies \text{square}\ R\ S\ T'\ U$
 ⟨*proof*⟩

lemma *square-reflcl*:

$[| \text{square } R \ S \ T \ (R \hat{=}) ; S \leq T \ |] \implies \text{square } (R \hat{=}) \ S \ T \ (R \hat{=})$
 $\langle \text{proof} \rangle$

lemma *square-rtrancl*:

$\text{square } R \ S \ S \ T \implies \text{square } (R^{**}) \ S \ S \ (T^{**})$
 $\langle \text{proof} \rangle$

lemma *square-rtrancl-reflcl-commute*:

$\text{square } R \ S \ (S^{**}) \ (R \hat{=}) \implies \text{commute } (R^{**}) \ (S^{**})$
 $\langle \text{proof} \rangle$

commute

lemma *commute-sym*: $\text{commute } R \ S \implies \text{commute } S \ R$

$\langle \text{proof} \rangle$

lemma *commute-rtrancl*: $\text{commute } R \ S \implies \text{commute } (R^{**}) \ (S^{**})$

$\langle \text{proof} \rangle$

lemma *commute-Un*:

$[| \text{commute } R \ T ; \text{commute } S \ T \ |] \implies \text{commute } (\sup R \ S) \ T$
 $\langle \text{proof} \rangle$

diamond, confluence, and union

lemma *diamond-Un*:

$[| \text{diamond } R ; \text{diamond } S ; \text{commute } R \ S \ |] \implies \text{diamond } (\sup R \ S)$
 $\langle \text{proof} \rangle$

lemma *diamond-confluent*: $\text{diamond } R \implies \text{confluent } R$

$\langle \text{proof} \rangle$

lemma *square-reflcl-confluent*:

$\text{square } R \ R \ (R \hat{=}) \ (R \hat{=}) \implies \text{confluent } R$
 $\langle \text{proof} \rangle$

lemma *confluent-Un*:

$[| \text{confluent } R ; \text{confluent } S ; \text{commute } (R^{**}) \ (S^{**}) \ |] \implies \text{confluent } (\sup R \ S)$
 $\langle \text{proof} \rangle$

lemma *diamond-to-confluence*:

$[| \text{diamond } R ; T \leq R ; R \leq T^{**} \ |] \implies \text{confluent } T$
 $\langle \text{proof} \rangle$

2.3 Church-Rosser

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

$\langle \text{proof} \rangle$

2.4 Newman's lemma

Proof by Stefan Berghofer

theorem *newman*:

assumes *wf*: $wfP\ (R^{-1-1})$

and *lc*: $\bigwedge a\ b\ c.\ R\ a\ b \implies R\ a\ c \implies$

$\exists d.\ R^{**}\ b\ d \wedge R^{**}\ c\ d$

shows $\bigwedge b\ c.\ R^{**}\ a\ b \implies R^{**}\ a\ c \implies$

$\exists d.\ R^{**}\ b\ d \wedge R^{**}\ c\ d$

$\langle proof \rangle$

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

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

theorem *newman'*:

assumes *wf*: $wfP\ (R^{-1-1})$

and *lc*: $\bigwedge a\ b\ c.\ R\ a\ b \implies R\ a\ c \implies$

$\exists d.\ R^{**}\ b\ d \wedge R^{**}\ c\ d$

shows $\bigwedge b\ c.\ R^{**}\ a\ b \implies R^{**}\ a\ c \implies$

$\exists d.\ R^{**}\ b\ d \wedge R^{**}\ c\ d$

$\langle proof \rangle$

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

lemma *eq-imp-rtranclp*: $x = y \implies r^{**}\ x\ y$

$\langle proof \rangle$

theorem *newman''*:

assumes *wf*: $wfP\ (R^{-1-1})$

and *lc*: $\bigwedge a\ b\ c.\ R\ a\ b \implies R\ a\ c \implies$

$\exists d.\ R^{**}\ b\ d \wedge R^{**}\ c\ d$

shows $\bigwedge b\ c.\ R^{**}\ a\ b \implies R^{**}\ a\ c \implies$

$\exists d.\ R^{**}\ b\ d \wedge R^{**}\ c\ d$

$\langle proof \rangle$

end

3 Parallel reduction and a complete developments

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

3.1 Parallel reduction

inductive *par-beta* :: $[dB, dB] \implies bool$ (**infixl** $\implies 50$)

where

$var\ [simp, intro!]: Var\ n \implies Var\ n$

$| abs\ [simp, intro!]: s \implies t \implies Abs\ s \implies Abs\ t$

| *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*)
⟨*proof*⟩

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

⟨*proof*⟩

lemma *beta-subset-par-beta*: *beta* <= *par-beta*

⟨*proof*⟩

lemma *par-beta-subset-beta*: *par-beta* <= *beta* ^**

⟨*proof*⟩

3.3 Misc properties of *par-beta*

lemma *par-beta-lift* [*simp*]:

t => *t'* ==> *lift* *t* *n* => *lift* *t'* *n*
⟨*proof*⟩

lemma *par-beta-subst*:

s => *s'* ==> *t* => *t'* ==> *t*[*s/n*] => *t'*[*s'/n*]
⟨*proof*⟩

3.4 Confluence (directly)

lemma *diamond-par-beta*: *diamond* *par-beta*

⟨*proof*⟩

3.5 Complete developments

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*

| $cd (Abs\ u \circ t) = (cd\ u)[cd\ t/0]$
| $cd (Abs\ s) = Abs\ (cd\ s)$

lemma *par-beta-cd*: $s \Rightarrow t \implies t \Rightarrow cd\ s$
 $\langle proof \rangle$

3.6 Confluence (via complete developments)

lemma *diamond-par-beta2*: *diamond par-beta*
 $\langle proof \rangle$

theorem *beta-confluent*: *confluent beta*
 $\langle proof \rangle$

end

4 Eta-reduction

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

4.1 Definition of eta-reduction and relatives

primrec

$free :: dB \Rightarrow nat \Rightarrow bool$

where

$free\ (Var\ j)\ i = (j = i)$
| $free\ (s \circ t)\ i = (free\ s\ i \vee free\ t\ i)$
| $free\ (Abs\ s)\ i = free\ s\ (i + 1)$

inductive

$eta :: [dB, dB] \Rightarrow bool\ \text{(infixl } \rightarrow_\eta\ 50)$

where

$eta\ [simp, intro]: \neg free\ s\ 0 \implies Abs\ (s \circ Var\ 0) \rightarrow_\eta s[dummy/0]$
| $appL\ [simp, intro]: s \rightarrow_\eta t \implies s \circ u \rightarrow_\eta t \circ u$
| $appR\ [simp, intro]: s \rightarrow_\eta t \implies u \circ s \rightarrow_\eta u \circ t$
| $abs\ [simp, intro]: s \rightarrow_\eta t \implies Abs\ s \rightarrow_\eta Abs\ t$

abbreviation

$eta-reds :: [dB, dB] \Rightarrow bool\ \text{(infixl } -e>>\ 50)$ **where**
 $s -e>> t == eta^{**}\ s\ t$

abbreviation

$eta-red0 :: [dB, dB] \Rightarrow bool\ \text{(infixl } -e>= 50)$ **where**
 $s -e>= t == eta^{==}\ s\ t$

notation (*xsymbols*)

$eta-reds\ \text{(infixl } \rightarrow_\eta^*\ 50)$ **and**
 $eta-red0\ \text{(infixl } \rightarrow_\eta^= 50)$

inductive-cases *eta-cases* [elim!]:

$Abs\ s \rightarrow_{\eta} z$
 $s \circ t \rightarrow_{\eta} u$
 $Var\ i \rightarrow_{\eta} t$

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

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

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

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

lemma *free-eta*: $s \rightarrow_{\eta} t \implies free\ t\ i = free\ s\ i$
 ⟨proof⟩

lemma *not-free-eta*:
 $[| s \rightarrow_{\eta} t; \neg free\ s\ i |] \implies \neg free\ t\ i$
 ⟨proof⟩

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

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

4.3 Confluence of *eta*

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

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

4.4 Congruence rules for *eta**

lemma *rtrancl-eta-Abs*: $s \rightarrow_{\eta}^* s' \implies Abs\ s \rightarrow_{\eta}^* Abs\ s'$
 ⟨proof⟩

lemma *rtrancl-eta-AppL*: $s \rightarrow_{\eta}^* s' \implies s \circ t \rightarrow_{\eta}^* s' \circ t$
 ⟨proof⟩

lemma *rtranc1-eta-AppR*: $t \rightarrow_{\eta}^* t' \implies s \circ t \rightarrow_{\eta}^* s \circ t'$
 $\langle \text{proof} \rangle$

lemma *rtranc1-eta-App*:
 $[| s \rightarrow_{\eta}^* s'; t \rightarrow_{\eta}^* t' |] \implies s \circ t \rightarrow_{\eta}^* s' \circ t'$
 $\langle \text{proof} \rangle$

4.5 Commutation of *beta* and *eta*

lemma *free-beta*:
 $s \rightarrow_{\beta} t \implies \text{free } t \ i \implies \text{free } s \ i$
 $\langle \text{proof} \rangle$

lemma *beta-subst [intro]*: $s \rightarrow_{\beta} t \implies s[u/i] \rightarrow_{\beta} t[u/i]$
 $\langle \text{proof} \rangle$

lemma *subst-Var-Suc [simp]*: $t[\text{Var } i/i] = t[\text{Var}(i)/i + 1]$
 $\langle \text{proof} \rangle$

lemma *eta-lift [simp]*: $s \rightarrow_{\eta} t \implies \text{lift } s \ i \rightarrow_{\eta} \text{lift } t \ i$
 $\langle \text{proof} \rangle$

lemma *rtranc1-eta-subst*: $s \rightarrow_{\eta} t \implies u[s/i] \rightarrow_{\eta}^* u[t/i]$
 $\langle \text{proof} \rangle$

lemma *rtranc1-eta-subst'*:
fixes $s \ t :: dB$
assumes $\text{eta}: s \rightarrow_{\eta}^* t$
shows $s[u/i] \rightarrow_{\eta}^* t[u/i]$ $\langle \text{proof} \rangle$

lemma *rtranc1-eta-subst''*:
fixes $s \ t :: dB$
assumes $\text{eta}: s \rightarrow_{\eta}^* t$
shows $u[s/i] \rightarrow_{\eta}^* u[t/i]$ $\langle \text{proof} \rangle$

lemma *square-beta-eta*: $\text{square } \text{beta } \text{eta} \ (\text{eta}^{**}) \ (\text{beta}^{\hat{}} \implies)$
 $\langle \text{proof} \rangle$

lemma *confluent-beta-eta*: $\text{confluent } (\text{sup } \text{beta } \text{eta})$
 $\langle \text{proof} \rangle$

4.6 Implicit definition of *eta*

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

lemma *not-free-iff-lifted*:
 $(\neg \text{free } s \ i) = (\exists t. s = \text{lift } t \ i)$
 $\langle \text{proof} \rangle$

theorem *explicit-is-implicit*:

$(\forall s\ u. (\neg \text{free } s\ 0) \dashv\dashv > R\ (\text{Abs } (s \circ \text{Var } 0))\ (s[u/0])) =$
 $(\forall s. R\ (\text{Abs } (\text{lift } s\ 0 \circ \text{Var } 0))\ s)$
 $\langle \text{proof} \rangle$

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 $\text{free} : \neg \text{free } s\ 0$
and $s : s[\text{dummy}/0] => u$
shows $\exists t'. \text{Abs } (s \circ \text{Var } 0) => t' \wedge t' \rightarrow_{\eta}^* u$
 $\langle \text{proof} \rangle$

theorem *eta-par-beta*:

assumes $st : s \rightarrow_{\eta} t$
and $tu : t => u$
shows $\exists t'. s => t' \wedge t' \rightarrow_{\eta}^* u \langle \text{proof} \rangle$

theorem *eta-postponement'*:

assumes $\text{eta} : s \rightarrow_{\eta}^* t$ **and** $\text{beta} : t => u$
shows $\exists t'. s => t' \wedge t' \rightarrow_{\eta}^* u \langle \text{proof} \rangle$

theorem *eta-postponement*:

assumes $(\text{sup } \text{beta } \text{eta})^{**} s\ t$
shows $(\text{beta}^{**} \text{OO } \text{eta}^{**}) s\ t \langle \text{proof} \rangle$

end

5 Application of a term to a list of terms

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

abbreviation

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

lemma *apps-eq-tail-conv* [iff]: $(r \circ^{\circ} ts = s \circ^{\circ} ts) = (r = s)$
 $\langle \text{proof} \rangle$

lemma *Var-eq-apps-conv* [iff]: $(\text{Var } m = s \circ^{\circ} ss) = (\text{Var } m = s \wedge ss = [])$
 $\langle \text{proof} \rangle$

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

$\langle proof \rangle$

lemma *App-eq-foldl-conv*:

$(r \circ s = t \circ\circ ts) =$
 (if $ts = []$ *then* $r \circ s = t$
 else $(\exists ss. ts = ss @ [s] \wedge r = t \circ\circ ss))$
 $\langle proof \rangle$

lemma *Abs-eq-apps-conv* [iff]:

$(Abs\ r = s \circ\circ ss) = (Abs\ r = s \wedge ss = [])$
 $\langle proof \rangle$

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

$\langle proof \rangle$

lemma *Abs-apps-eq-Abs-apps-conv* [iff]:

$(Abs\ r \circ\circ rs = Abs\ s \circ\circ ss) = (r = s \wedge rs = ss)$
 $\langle proof \rangle$

lemma *Abs-App-neq-Var-apps* [iff]:

$Abs\ s \circ t \neq Var\ n \circ\circ ss$
 $\langle proof \rangle$

lemma *Var-apps-neq-Abs-apps* [iff]:

$Var\ n \circ\circ ts \neq Abs\ r \circ\circ ss$
 $\langle proof \rangle$

lemma *ex-head-tail*:

$\exists ts\ h. t = h \circ\circ ts \wedge ((\exists n. h = Var\ n) \vee (\exists u. h = Abs\ u))$
 $\langle proof \rangle$

lemma *size-apps* [simp]:

$size\ (r \circ\circ rs) = size\ r + foldl\ (op\ +)\ 0\ (map\ size\ rs) + length\ rs$
 $\langle proof \rangle$

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

$\langle proof \rangle$

lemma *lift-map* [simp]:

$lift\ (t \circ\circ ts)\ i = lift\ t\ i \circ\circ map\ (\lambda t. lift\ t\ i)\ ts$
 $\langle proof \rangle$

lemma *subst-map* [simp]:

$subst\ (t \circ\circ ts)\ u\ i = subst\ t\ u\ i \circ\circ map\ (\lambda t. subst\ t\ u\ i)\ ts$
 $\langle proof \rangle$

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

$\langle proof \rangle$

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

lemma *lem*:

assumes $!!n\ ts. \forall t \in \text{set } ts. P\ t \implies P\ (Var\ n\ ^{\circ\circ}\ ts)$
and $!!u\ ts. [\![\ P\ u; \forall t \in \text{set } ts. P\ t\]\!] \implies P\ (Abs\ u\ ^{\circ\circ}\ ts)$
shows $\text{size } t = n \implies P\ t$
 $\langle \text{proof} \rangle$

theorem *Apps-dB-induct*:

assumes $!!n\ ts. \forall t \in \text{set } ts. P\ t \implies P\ (Var\ n\ ^{\circ\circ}\ ts)$
and $!!u\ ts. [\![\ P\ u; \forall t \in \text{set } ts. P\ t\]\!] \implies P\ (Abs\ u\ ^{\circ\circ}\ ts)$
shows $P\ t$
 $\langle \text{proof} \rangle$

end

6 Simply-typed lambda terms

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

6.1 Environments

definition

$\text{shift} :: (\text{nat} \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow \text{nat} \Rightarrow 'a\ (-\langle -: \rangle [90, 0, 0]\ 91)$ **where**
 $e\langle i:a \rangle = (\lambda j. \text{if } j < i \text{ then } e\ j \text{ else if } j = i \text{ then } a \text{ else } e\ (j - 1))$

notation (*xsymbols*)

$\text{shift}\ (-\langle -: \rangle [90, 0, 0]\ 91)$

notation (*HTML output*)

$\text{shift}\ (-\langle -: \rangle [90, 0, 0]\ 91)$

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

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

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

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

6.2 Types and typing rules

datatype *type* =

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

inductive *typing* :: (nat \Rightarrow type) \Rightarrow dB \Rightarrow type \Rightarrow bool (- \vdash - : - [50, 50, 50] 50)

where

Var [intro!]: env $x = T \implies \text{env} \vdash \text{Var } x : T$
 | Abs [intro!]: env $\langle 0:T \rangle \vdash t : U \implies \text{env} \vdash \text{Abs } t : (T \Rightarrow U)$
 | App [intro!]: env $\vdash s : T \Rightarrow U \implies \text{env} \vdash t : T \implies \text{env} \vdash (s \circ t) : U$

inductive-cases *typing-elim* [elim!]:

$e \vdash \text{Var } i : T$
 $e \vdash t \circ u : T$
 $e \vdash \text{Abs } t : T$

primrec

typings :: (nat \Rightarrow type) \Rightarrow dB list \Rightarrow type list \Rightarrow bool

where

typings e [] $Ts = (Ts = [])$
 | *typings* e ($t \# ts$) $Ts =$
 (case Ts of
 [] $\Rightarrow \text{False}$
 | $T \# Ts \Rightarrow e \vdash t : T \wedge \text{typings } e \text{ } ts \text{ } Ts)$

abbreviation

typings-rel :: (nat \Rightarrow type) \Rightarrow dB list \Rightarrow type list \Rightarrow bool
 (- || - : - [50, 50, 50] 50) **where**
 env || - $ts : Ts == \text{typings } \text{env } ts \text{ } Ts$

notation (*latex*)

typings-rel (- \Vdash - : - [50, 50, 50] 50)

abbreviation

*fun*s :: type list \Rightarrow type \Rightarrow type (**infixr** $\Rightarrow \Rightarrow$ 200) **where**
 $Ts \Rightarrow \Rightarrow T == \text{foldr } \text{Fun } Ts \text{ } T$

notation (*latex*)

*fun*s (**infixr** \Rightarrow 200)

6.3 Some examples

schematic-lemma $e \vdash \text{Abs } (\text{Abs } (\text{Abs } (\text{Var } 1 \circ (\text{Var } 2 \circ \text{Var } 1 \circ \text{Var } 0)))) : ?T$
 $\langle \text{proof} \rangle$

schematic-lemma $e \vdash \text{Abs } (\text{Abs } (\text{Abs } (\text{Var } 2 \circ \text{Var } 0 \circ (\text{Var } 1 \circ \text{Var } 0)))) : ?T$
 $\langle \text{proof} \rangle$

6.4 Lists of types

lemma *lists-typings*:

$e \Vdash ts : Ts \implies \text{listsp } (\lambda t. \exists T. e \vdash t : T) \text{ } ts$
 $\langle \text{proof} \rangle$

lemma *types-snoc*: $e \Vdash ts : Ts \implies e \vdash t : T \implies e \Vdash ts @ [t] : Ts @ [T]$
 $\langle proof \rangle$

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

lemma *rev-exhaust2* [*extraction-expand*]:
obtains $(Nil) \ xs = [] \mid (snoc) \ ys \ y \textbf{ where } xs = ys @ [y]$
 — Cannot use *rev-exhaust* from the *List* theory, since it is not constructive
 $\langle proof \rangle$

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

6.5 n-ary function types

lemma *list-app-typeD*:
 $e \vdash t \circ\circ ts : T \implies \exists Ts. e \vdash t : Ts \Rightarrow T \wedge e \Vdash ts : Ts$
 $\langle proof \rangle$

lemma *list-app-typeE*:
 $e \vdash t \circ\circ ts : T \implies (\bigwedge Ts. e \vdash t : Ts \Rightarrow T \implies e \Vdash ts : Ts \implies C) \implies C$
 $\langle proof \rangle$

lemma *list-app-typeI*:
 $e \vdash t : Ts \Rightarrow T \implies e \Vdash ts : Ts \implies e \vdash t \circ\circ ts : T$
 $\langle proof \rangle$

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*:
 $e \vdash Var \ i \circ\circ ts : T \implies e \vdash Var \ i \circ\circ ts : U \implies T = U$
 $\langle proof \rangle$

lemma *var-app-types*: $e \vdash Var \ i \circ\circ ts \circ\circ us : T \implies e \Vdash ts : Ts \implies$
 $e \vdash Var \ i \circ\circ ts : U \implies \exists Us. U = Us \Rightarrow T \wedge e \Vdash us : Us$
 $\langle proof \rangle$

lemma *var-app-typesE*: $e \vdash Var \ i \circ\circ ts : T \implies$
 $(\bigwedge Ts. e \vdash Var \ i : Ts \Rightarrow T \implies e \Vdash ts : Ts \implies P) \implies P$
 $\langle proof \rangle$

lemma *abs-typeE*: $e \vdash Abs \ t : T \implies (\bigwedge U \ V. e \langle \emptyset : U \rangle \vdash t : V \implies P) \implies P$
 $\langle proof \rangle$

6.6 Lifting preserves well-typedness

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

lemma *lift-types*:

$e \Vdash ts : Ts \implies e\langle i:U \rangle \Vdash (\text{map } (\lambda t. \text{lift } t \ i) \ ts) : Ts$
 $\langle \text{proof} \rangle$

6.7 Substitution lemmas

lemma *subst-lemma*:

$e \vdash t : T \implies e' \vdash u : U \implies e = e'\langle i:U \rangle \implies e' \vdash t[u/i] : T$
 $\langle \text{proof} \rangle$

lemma *substs-lemma*:

$e \vdash u : T \implies e\langle i:T \rangle \Vdash ts : Ts \implies$
 $e \Vdash (\text{map } (\lambda t. t[u/i]) \ ts) : Ts$
 $\langle \text{proof} \rangle$

6.8 Subject reduction

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

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

6.9 Alternative induction rule for types

lemma *type-induct* [*induct type*]:

assumes

$(\bigwedge T. (\bigwedge T1 \ T2. T = T1 \implies T2 \implies P \ T1) \implies$
 $(\bigwedge T1 \ T2. T = T1 \implies T2 \implies P \ T2) \implies P \ T)$

shows $P \ T$

$\langle \text{proof} \rangle$

end

7 Lifting an order to lists of elements

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

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

definition

step1 :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$ **where**

step1 $r =$

$(\lambda ys \ xs. \exists us \ z \ z' \ vs. xs = us @ z \# vs \wedge r \ z' \ z \wedge ys =$

$us @ z' \# vs)$

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

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

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

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

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

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

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

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

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

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

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

lemma *lists-accI*: $\text{accp } (\text{step1 } r) \text{ } xs \implies \text{listsp } (\text{accp } r) \text{ } xs$
 $\langle \text{proof} \rangle$

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 :: $\text{dB list} \Rightarrow \text{dB list} \Rightarrow \text{bool}$ (**infixl** \Rightarrow 50) **where**
 $rs \Rightarrow ss == \text{step1 beta } rs \text{ } ss$

lemma *head-Var-reduction*:

$\text{Var } n \circ\circ rs \rightarrow_\beta v \implies \exists ss. rs \Rightarrow ss \wedge v = \text{Var } n \circ\circ ss$
 $\langle \text{proof} \rangle$

lemma *apps-betasE* [*elim!*]:

assumes *major*: $r \circ\circ rs \rightarrow_\beta s$
and cases: $!!r'. [| r \rightarrow_\beta r'; s = r' \circ\circ rs |] \implies R$
 $!!rs'. [| rs \Rightarrow rs'; s = r \circ\circ rs' |] \implies R$
 $!!t \text{ } u \text{ } us. [| r = \text{Abs } t; rs = u \# us; s = t[u/\theta] \circ\circ us |] \implies R$
shows R

$\langle \text{proof} \rangle$

lemma *apps-preserves-beta* [*simp*]:

$r \rightarrow_\beta s \implies r \circ\circ ss \rightarrow_\beta s \circ\circ ss$
 $\langle \text{proof} \rangle$

lemma *apps-preserves-beta2* [*simp*]:

$r ->> s \implies r \circ\circ ss ->> s \circ\circ ss$
 $\langle \text{proof} \rangle$

lemma *apps-preserves-betas* [*simp*]:

$rs \Rightarrow ss \implies r \circ\circ rs \rightarrow_\beta r \circ\circ ss$
 $\langle \text{proof} \rangle$

end

9 Inductive characterization of terminating lambda terms

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

9.1 Terminating lambda terms

inductive $IT :: dB \Rightarrow bool$

where

$Var [intro]: listsp\ IT\ rs \Rightarrow IT\ (Var\ n\ \circ\circ\ rs)$
 $| Lambda [intro]: IT\ r \Rightarrow IT\ (Abs\ r)$
 $| Beta [intro]: IT\ ((r[s/0])\ \circ\circ\ ss) \Rightarrow IT\ s \Rightarrow IT\ ((Abs\ r\ \circ\ s)\ \circ\circ\ ss)$

9.2 Every term in IT terminates

lemma *double-induction-lemma* [rule-format]:

$termip\ beta\ s \Rightarrow \forall t. termip\ beta\ t \longrightarrow$
 $(\forall r\ ss. t = r[s/0]\ \circ\circ\ ss \longrightarrow termip\ beta\ (Abs\ r\ \circ\ s\ \circ\circ\ ss))$
 $\langle proof \rangle$

lemma *IT-implies-termi*: $IT\ t \Rightarrow termip\ beta\ t$

$\langle proof \rangle$

9.3 Every terminating term is in IT

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

lemma [simp, THEN not-sym, simp]: $Var\ n\ \circ\circ\ ss \neq Abs\ r\ \circ\ s\ \circ\circ\ ts$
 $\langle proof \rangle$

lemma [simp]:

$(Abs\ r\ \circ\ s\ \circ\circ\ ss = Abs\ r'\ \circ\ s'\ \circ\circ\ ss') = (r = r' \wedge s = s' \wedge ss = ss')$
 $\langle proof \rangle$

inductive-cases [elim!]:

$IT\ (Var\ n\ \circ\circ\ ss)$
 $IT\ (Abs\ t)$
 $IT\ (Abs\ r\ \circ\ s\ \circ\circ\ ts)$

theorem *termi-implies-IT*: $termip\ beta\ r \Rightarrow IT\ r$

$\langle proof \rangle$

end

10 Strong normalization for simply-typed lambda calculus

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

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

10.1 Properties of IT

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

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

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

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

lemma *app-Var-IT*: $IT\ t \implies IT\ (t \circ Var\ i)$
⟨proof⟩

10.2 Well-typed substitution preserves termination

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

10.3 Well-typed terms are strongly normalizing

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

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

end

11 Inductive characterization of lambda terms in normal form

theory *NormalForm*
imports *ListBeta*
begin

11.1 Terms in normal form

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

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

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

theorem *listall-nil*: $listall\ P\ []$
 $\langle proof \rangle$

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

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

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

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

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

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

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

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

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

lemma *listall-listsp-eq*: $listall\ P\ xs = listsp\ P\ xs$
 $\langle proof \rangle$

inductive *NF* :: *dB* \Rightarrow *bool*

where

App: $listall\ NF\ ts \Longrightarrow NF\ (Var\ x\ {}^{\circ\circ}\ ts)$

| *Abs*: $NF\ t \Longrightarrow NF\ (Abs\ t)$

monos *listall-def*

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

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

lemma *App-NF-D*: **assumes** $NF: NF \ (Var \ n \ \circ\circ \ ts)$
shows $listall \ NF \ ts \ \langle proof \rangle$

11.2 Properties of NF

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

lemma *Abs-NF*:
assumes $NF: NF \ (Abs \ t \ \circ\circ \ ts)$
shows $ts = [] \ \langle proof \rangle$

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

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

lemma *app-Var-NF*: $NF \ t \implies \exists t'. \ t \ \circ \ Var \ i \rightarrow_{\beta}^* t' \wedge NF \ t'$
 $\langle proof \rangle$

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

lemma *lift-NF*: $NF \ t \implies NF \ (lift \ t \ i)$
 $\langle proof \rangle$

NF characterizes exactly the terms that are in normal form.

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

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* \Rightarrow *dB* \Rightarrow *bool* (**infixl** \rightarrow_s 50)

and *sredlist* :: *dB list* \Rightarrow *dB list* \Rightarrow *bool* (**infixl** $[\rightarrow_s]$ 50)

where

$s [\rightarrow_s] t \equiv \text{listrelp } op \rightarrow_s s t$

| *Var*: $rs [\rightarrow_s] rs' \Longrightarrow Var\ x \circ\!\circ\ rs \rightarrow_s Var\ x \circ\!\circ\ rs'$

| *Abs*: $r \rightarrow_s r' \Longrightarrow ss [\rightarrow_s] ss' \Longrightarrow Abs\ r \circ\!\circ\ ss \rightarrow_s Abs\ r' \circ\!\circ\ ss'$

| *Beta*: $r[s/0] \circ\!\circ\ ss \rightarrow_s t \Longrightarrow Abs\ r \circ\ s \circ\!\circ\ ss \rightarrow_s t$

lemma *refl-listrelp*: $\forall x \in set\ xs. R\ x\ x \Longrightarrow \text{listrelp } R\ xs\ xs$

<proof>

lemma *refl-sred*: $t \rightarrow_s t$

<proof>

lemma *refl-sreds*: $ts [\rightarrow_s] ts$

<proof>

lemma *listrelp-conj1*: $\text{listrelp } (\lambda x\ y. R\ x\ y \wedge S\ x\ y)\ x\ y \Longrightarrow \text{listrelp } R\ x\ y$

<proof>

lemma *listrelp-conj2*: $\text{listrelp } (\lambda x\ y. R\ x\ y \wedge S\ x\ y)\ x\ y \Longrightarrow \text{listrelp } S\ x\ y$

<proof>

lemma *listrelp-app*:

assumes *xsys*: $\text{listrelp } R\ xs\ ys$

shows $\text{listrelp } R\ xs'\ ys' \Longrightarrow \text{listrelp } R\ (xs\ @\ xs')\ (ys\ @\ ys')\ \langle proof \rangle$

lemma *lemma1*:

assumes *r*: $r \rightarrow_s r'$ **and** *s*: $s \rightarrow_s s'$

shows $r \circ\ s \rightarrow_s r' \circ\ s'\ \langle proof \rangle$

lemma *lemma1'*:

assumes *ts*: $ts [\rightarrow_s] ts'$

shows $r \rightarrow_s r' \Longrightarrow r \circ\!\circ\ ts \rightarrow_s r' \circ\!\circ\ ts'\ \langle proof \rangle$

lemma *lemma2-1*:

assumes *beta*: $t \rightarrow_\beta u$

shows $t \rightarrow_s u\ \langle proof \rangle$

lemma *listrelp-betas*:

assumes *ts*: $\text{listrelp } op \rightarrow_\beta^* ts\ ts'$

shows $\bigwedge t\ t'. t \rightarrow_\beta^* t' \Longrightarrow t \circ\!\circ\ ts \rightarrow_\beta^* t' \circ\!\circ\ ts'\ \langle proof \rangle$

lemma *lemma2-2*:

assumes *t*: $t \rightarrow_s u$

shows $t \rightarrow_{\beta}^* u \langle proof \rangle$

lemma *sred-lift*:

assumes $s: s \rightarrow_s t$

shows $lift\ s\ i \rightarrow_s lift\ t\ i \langle proof \rangle$

lemma *lemma3*:

assumes $r: r \rightarrow_s r'$

shows $s \rightarrow_s s' \implies r[s/x] \rightarrow_s r'[s'/x] \langle proof \rangle$

lemma *lemma4-aux*:

assumes $rs: listrelp\ (\lambda t\ u. t \rightarrow_s u \wedge (\forall r. u \rightarrow_{\beta} r \longrightarrow t \rightarrow_s r))\ rs\ rs'$

shows $rs' \Rightarrow ss \implies rs \rightarrow_s ss \langle proof \rangle$

lemma *lemma4*:

assumes $r: r \rightarrow_s r'$

shows $r' \rightarrow_{\beta} r'' \implies r \rightarrow_s r'' \langle proof \rangle$

lemma *rtrancl-beta-sred*:

assumes $r: r \rightarrow_{\beta}^* r'$

shows $r \rightarrow_s r' \langle proof \rangle$

12.2 Leftmost reduction and weakly normalizing terms

inductive

$lred :: dB \Rightarrow dB \Rightarrow bool$ (**infixl** \rightarrow_l 50)

and $lredlist :: dB\ list \Rightarrow dB\ list \Rightarrow bool$ (**infixl** $[\rightarrow_l]$ 50)

where

$s [\rightarrow_l] t \equiv listrelp\ op\ \rightarrow_l\ s\ t$

| $Var: rs [\rightarrow_l] rs' \implies Var\ x\ ^{\circ\circ} rs \rightarrow_l Var\ x\ ^{\circ\circ} rs'$

| $Abs: r \rightarrow_l r' \implies Abs\ r \rightarrow_l Abs\ r'$

| $Beta: r[s/0] ^{\circ\circ} ss \rightarrow_l t \implies Abs\ r\ ^{\circ} s\ ^{\circ\circ} ss \rightarrow_l t$

lemma *lred-imp-sred*:

assumes $lred: s \rightarrow_l t$

shows $s \rightarrow_s t \langle proof \rangle$

inductive $WN :: dB \Rightarrow bool$

where

$Var: listsp\ WN\ rs \implies WN\ (Var\ n\ ^{\circ\circ} rs)$

| $Lambda: WN\ r \implies WN\ (Abs\ r)$

| $Beta: WN\ ((r[s/0]) ^{\circ\circ} ss) \implies WN\ ((Abs\ r\ ^{\circ} s) ^{\circ\circ} ss)$

lemma *listrelp-imp-listsp1*:

assumes $H: listrelp\ (\lambda x\ y. P\ x)\ xs\ ys$

shows $listsp\ P\ xs \langle proof \rangle$

lemma *listrelp-imp-listsp2*:

assumes $H: listrelp\ (\lambda x\ y. P\ y)\ xs\ ys$

```

shows listsp P ys ⟨proof⟩

lemma lemma5:
  assumes lred:  $r \rightarrow_l r'$ 
  shows  $WN\ r$  and  $NF\ r' \langle proof \rangle$ 

lemma lemma6:
  assumes wn:  $WN\ r$ 
  shows  $\exists r'.\ r \rightarrow_l r' \langle proof \rangle$ 

lemma lemma7:
  assumes r:  $r \rightarrow_s r'$ 
  shows  $NF\ r' \implies r \rightarrow_l r' \langle proof \rangle$ 

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

end

```

13 Weak normalization for simply-typed lambda calculus

```

theory WeakNorm
imports Type NormalForm Code-Integer
begin

```

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

13.1 Main theorems

```

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

```

```

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

```

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

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

13.2 Extracting the program

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

extract *type-NF*

lemma *rtrancR-rtrancL-eq*: $rtrancpR\ r\ a\ b = r^{**}\ a\ b$
 $\langle proof \rangle$

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

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

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

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

```

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

```

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

```

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

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

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

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

```

Figure 2: Program extracted from lemmas and main theorem

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

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

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

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

$$\begin{aligned} e \ x = T &\implies \text{rtypingR } (\text{rtypingT.Var } e \ x \ T) \ e \ (dB.Var \ x) \ T \\ \text{rtypingR } ty \ (e \langle 0:T \rangle) \ t \ U &\implies \text{rtypingR } (\text{rtypingT.Abs } e \ T \ t \ U \ ty) \ e \ (dB.Abs \ t) \\ &\quad (T \Rightarrow U) \\ \text{rtypingR } ty \ e \ s \ (T \Rightarrow U) &\implies \\ \text{rtypingR } ty' \ e \ t \ T &\implies \text{rtypingR } (\text{rtypingT.App } e \ s \ T \ U \ t \ ty \ ty') \ e \ (s \circ t) \ U \end{aligned}$$

13.3 Generating executable code

```

instantiation NFT :: default
begin

definition default = Dummy ()

instance <proof>

end

instantiation dB :: default
begin

definition default = dB.Var 0

instance <proof>

end

instantiation * :: (default, default) default
begin

definition default = (default, default)

instance <proof>

end

```



```

instantiation list :: (type) default
begin

definition default = []

instance  $\langle proof \rangle$ 

end

instantiation fun :: (type, default) default
begin

definition default = ( $\lambda x.$  default)

instance  $\langle proof \rangle$ 

end

definition int-of-nat :: nat  $\Rightarrow$  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.

$\langle ML \rangle$

The same story again for the SML code generator.

```

consts-code
  default ((error default))
  default :: 'a  $\Rightarrow$  'b::default ((fn ' - => error default))

```

```

code-module Norm
contains
  test = type-NF

```

$\langle ML \rangle$

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

$\langle ML \rangle$

```

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.