

The $\lambda\mu$ -calculus

Cristina Matache Victor B. F. Gomes Dominic P. Mulligan

May 4, 2022

Contents

1	The $\lambda\mu$-calculus	1
1.1	Syntax	2
1.2	Types	2
1.3	De Bruijn indices	4
1.4	Logical and structural substitution	6
1.5	Reduction relation	8
1.6	Contextual typing	9
1.7	Type preservation	10
1.8	Progress	13
1.9	Peirce	14

Abstract

The propositions-as-types correspondence is ordinarily presented as linking the metatheory of typed λ -calculi and the proof theory of intuitionistic logic. Griffin [2] observed that this correspondence could be extended to classical logic through the use of control operators. This observation set off a flurry of further research, leading to the development of Parigots $\lambda\mu$ -calculus [4]. In this work, we formalise $\lambda\mu$ -calculus in Isabelle/HOL and prove several metatheoretical properties such as type preservation and progress.

1 The $\lambda\mu$ -calculus

More examples, as well as a call-by-value programming language built on top of our formalisation, can be found in an associated Bitbucket repository [3].

```
theory Syntax  
  imports Main  
begin
```

1.1 Syntax

```
datatype type =  
  Iota  
  | Fun type type (infixr  $\rightarrow$  200)
```

To deal with α -equivalence, we use De Bruijn's nameless representation wherein each bound variable is represented by a natural number, its index, that denotes the number of binders that must be traversed to arrive at the one that binds the given variable. Each free variable has an index that points into the top-level context, not enclosed in any abstractions.

```
datatype trm =  
  LVar nat ('- [100] 100)  
  | Lbd type trm ( $\lambda$ -:- [0, 60] 60)  
  | App trm trm (infix  $\circ$  60)  
  | Mu type cmd ( $\mu$ -:- [0, 60] 60)  
and cmd =  
  MVar nat trm ( $\langle$ ->- [0, 60] 60)
```

```
datatype ctxt =  
  ContextEmpty ( $\diamond$ ) |  
  ContextApp ctxt trm (infixl  $\bullet$  75)
```

```
primrec ctxt-app :: ctxt  $\Rightarrow$  ctxt  $\Rightarrow$  ctxt (infix  $.$  60) where  
   $\diamond . F = F$  |  
   $(E \bullet t) . F = (E . F) \bullet t$ 
```

```
fun is-val :: trm  $\Rightarrow$  bool where  
  is-val ( $\lambda T : v$ ) = True |  
  is-val - = False
```

end

1.2 Types

```
theory Types  
  imports Syntax  
begin
```

We implement typing environments as (total) functions from natural numbers to types, following the approach of Stefan Berghofer in his formalisation of the simply typed λ -calculus in the Isabelle/HOL library. An empty typing environment may be represented by an arbitrary function of the correct type as it will never be queried when a typing judgement is valid. We split typing environments, dedicating one environment to λ -variables

and another to μ -variables, and use Γ and Δ to range over the former and latter, respectively.

From src/HOL/Proofs/LambdaType.thy

definition

shift :: (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a \Rightarrow nat \Rightarrow 'a (-\(\cdot\)) [90, 0, 0] 91)

where

$e\langle i:a \rangle = (\lambda j. \text{if } j < i \text{ then } e\ j \text{ else if } j = i \text{ then } a \text{ else } e\ (j-1))$

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

by (simp add: shift-def)

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

by (simp add: shift-def)

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

by (simp add: shift-def)

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

by (rule ext) (simp-all add: shift-def split: nat.split, force)

inductive *typing-trm* :: (nat \Rightarrow type) \Rightarrow (nat \Rightarrow type) \Rightarrow trm \Rightarrow type
 \Rightarrow bool

(-, - \vdash_T - : - [50, 50, 50, 50] 50)

and *typing-cmd* :: (nat \Rightarrow type) \Rightarrow (nat \Rightarrow type) \Rightarrow cmd \Rightarrow bool

(-, - \vdash_C - [50, 50, 50] 50)

where

var [intro!]: $\llbracket \Gamma\ x = T \rrbracket \Longrightarrow \Gamma, \Delta \vdash_T 'x : T \mid$

app [intro!]: $\llbracket \Gamma, \Delta \vdash_T t : (T1 \rightarrow T2); \Gamma, \Delta \vdash_T s : T1 \rrbracket$

$\Longrightarrow \Gamma, \Delta \vdash_T (t^\circ s) : T2 \mid$

lambda [intro!]: $\llbracket \Gamma\langle 0:T1 \rangle, \Delta \vdash_T t : T2 \rrbracket$

$\Longrightarrow \Gamma, \Delta \vdash_T (\lambda\ T1 : t) : (T1 \rightarrow T2) \mid$

activate [intro!]: $\llbracket \Gamma, \Delta\langle 0:T \rangle \vdash_C c \rrbracket \Longrightarrow \Gamma, \Delta \vdash_T (\mu\ T : c) : T \mid$

passivate [intro!]: $\llbracket \Gamma, \Delta \vdash_T t : T; \Delta\ x = T \rrbracket \Longrightarrow \Gamma, \Delta \vdash_C \langle x \rangle t$

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

$\Gamma, \Delta \vdash_T 'x : T$

$\Gamma, \Delta \vdash_T t^\circ s : T$

$\Gamma, \Delta \vdash_T \lambda\ T1 : t : T$

$\Gamma, \Delta \vdash_T \mu\ T1 : t : T$

$\Gamma, \Delta \vdash_C \langle x \rangle t$

inductive-cases *type-arrow-elim*:

$\Gamma, \Delta \vdash_T t : T1 \rightarrow T2$

lemma *uniqueness*:

$\Gamma, \Delta \vdash_T t : T1 \Longrightarrow \Gamma, \Delta \vdash_T t : T2 \Longrightarrow T1 = T2$

$\Gamma, \Delta \vdash_C c \Longrightarrow \Gamma, \Delta \vdash_C c$

apply(induct arbitrary: T2 rule: typing-trm-typing-cmd.inducts)

```

using typing-cmd.cases apply blast+
done

```

```

end
theory DeBruijn
  imports Syntax
begin

```

1.3 De Bruijn indices

Functions to find the free λ and μ variables in an expression.

```

primrec flv-trm :: trm  $\Rightarrow$  nat  $\Rightarrow$  nat set
  and flv-cmd :: cmd  $\Rightarrow$  nat  $\Rightarrow$  nat set

```

where

```

  flv-trm ('i) k = (if  $i \geq k$  then {i-k} else {})
  | flv-trm ( $\lambda$  T : t) k = flv-trm t (k+1)
  | flv-trm (sot) k = (flv-trm s k)  $\cup$  (flv-trm t k)
  | flv-trm ( $\mu$  T : c) k = flv-cmd c k
  | flv-cmd (<i> t) k = flv-trm t k

```

```

primrec fmv-trm :: trm  $\Rightarrow$  nat  $\Rightarrow$  nat set
  and fmv-cmd :: cmd  $\Rightarrow$  nat  $\Rightarrow$  nat set

```

where

```

  fmv-trm ('i) k = {}
  | fmv-trm ( $\lambda$  T : t) k = fmv-trm t k
  | fmv-trm (sot) k = (fmv-trm s k)  $\cup$  (fmv-trm t k)
  | fmv-trm ( $\mu$  T : c) k = fmv-cmd c (k+1)
  | fmv-cmd (<i> t) k = (if  $i \geq k$  then {i-k}  $\cup$  (fmv-trm t k) else
(fmv-trm t k))

```

abbreviation *lambda-closed* :: - **where**

```

lambda-closed t  $\equiv$  flv-trm t 0 = {}

```

abbreviation *lambda-closedC* :: - **where**

```

lambda-closedC c  $\equiv$  flv-cmd c 0 = {}

```

Free variables in a context.

```

primrec fmv-ctxt :: ctxt  $\Rightarrow$  nat  $\Rightarrow$  nat set where

```

```

  fmv-ctxt  $\diamond$  k = {}
  | fmv-ctxt (E • t) k = (fmv-ctxt E k)  $\cup$  (fmv-trm t k)

```

Shift free λ and μ variables in terms and commands to make substitution capture avoiding.

primrec

```

liftL-trm :: [trm, nat]  $\Rightarrow$  trm and

```

```

liftL-cmd :: [cmd, nat]  $\Rightarrow$  cmd

```

where

```

liftL-trm ('i) k = (if  $i < k$  then 'i else '(i+1)) |

```

$$\begin{aligned}
\text{liftL-trm } (\lambda T : t) k &= \lambda T : (\text{liftL-trm } t (k+1)) \mid \\
\text{liftL-trm } (s \circ t) k &= \text{liftL-trm } s k \circ \text{liftL-trm } t k \mid \\
\text{liftL-trm } (\mu T : c) k &= \mu T : (\text{liftL-cmd } c k) \mid \\
\text{liftL-cmd } (\langle i \rangle t) k &= \langle i \rangle (\text{liftL-trm } t k)
\end{aligned}$$

primrec

$$\begin{aligned}
\text{liftM-trm} &:: [\text{trm}, \text{nat}] \Rightarrow \text{trm} \text{ and} \\
\text{liftM-cmd} &:: [\text{cmd}, \text{nat}] \Rightarrow \text{cmd}
\end{aligned}$$

where

$$\begin{aligned}
\text{liftM-trm } (\text{'i}) k &= \text{'i} \mid \\
\text{liftM-trm } (\lambda T : t) k &= \lambda T : (\text{liftM-trm } t k) \mid \\
\text{liftM-trm } (s \circ t) k &= \text{liftM-trm } s k \circ \text{liftM-trm } t k \mid \\
\text{liftM-trm } (\mu T : c) k &= \mu T : (\text{liftM-cmd } c (k+1)) \mid \\
\text{liftM-cmd } (\langle i \rangle t) k &= \\
&\quad (\text{if } i < k \text{ then } (\langle i \rangle (\text{liftM-trm } t k)) \text{ else } (\langle i+1 \rangle (\text{liftM-trm } t k)))
\end{aligned}$$

Shift free λ and μ variables in contexts to make structural substitution capture avoiding.

primrec $\text{liftL-ctxt} :: \text{ctxt} \Rightarrow \text{nat} \Rightarrow \text{ctxt}$ **where**

$$\begin{aligned}
\text{liftL-ctxt } \diamond n &= \diamond \mid \\
\text{liftL-ctxt } (E \bullet t) n &= (\text{liftL-ctxt } E n) \bullet (\text{liftL-trm } t n)
\end{aligned}$$

primrec $\text{liftM-ctxt} :: \text{ctxt} \Rightarrow \text{nat} \Rightarrow \text{ctxt}$ **where**

$$\begin{aligned}
\text{liftM-ctxt } \diamond n &= \diamond \mid \\
\text{liftM-ctxt } (E \bullet t) n &= (\text{liftM-ctxt } E n) \bullet (\text{liftM-trm } t n)
\end{aligned}$$

A function to decrement the indices of free μ -variables when a μ surrounding the expression disappears as a result of a reduction

primrec

$$\begin{aligned}
\text{dropM-trm} &:: [\text{trm}, \text{nat}] \Rightarrow \text{trm} \text{ and} \\
\text{dropM-cmd} &:: [\text{cmd}, \text{nat}] \Rightarrow \text{cmd}
\end{aligned}$$

where

$$\begin{aligned}
\text{dropM-trm } (\text{'i}) k &= \text{'i} \\
\mid \text{dropM-trm } (\lambda T : t) k &= \lambda T : (\text{dropM-trm } t k) \\
\mid \text{dropM-trm } (s \circ t) k &= (\text{dropM-trm } s k) \circ (\text{dropM-trm } t k) \\
\mid \text{dropM-trm } (\mu T : c) k &= \mu T : (\text{dropM-cmd } c (k+1)) \\
\mid \text{dropM-cmd } (\langle i \rangle t) k &= \\
&\quad (\text{if } i > k \text{ then } (\langle i-1 \rangle (\text{dropM-trm } t k)) \text{ else } (\langle i \rangle (\text{dropM-trm } t k)))
\end{aligned}$$

lemma fmv-liftL :

$$\begin{aligned}
\beta \notin \text{fmv-trm } t n &\implies \beta \notin \text{fmv-trm } (\text{liftL-trm } t m) n \\
\beta \notin \text{fmv-cmd } c n &\implies \beta \notin \text{fmv-cmd } (\text{liftL-cmd } c m) n \\
\text{by}(\text{induct } t \text{ and } c \text{ arbitrary: } m n \text{ and } m n) \text{ auto}
\end{aligned}$$

lemma fmv-liftL-ctxt :

$$\begin{aligned}
\beta \notin \text{fmv-ctxt } E m &\implies \beta \notin \text{fmv-ctxt } (\text{liftL-ctxt } E n) m \\
\text{by}(\text{induct } E) (\text{fastforce simp add: fmv-liftL})+
\end{aligned}$$

lemma *fmv-suc*:

$\beta \notin \text{fmv-cmd } c \text{ (Suc } n) \implies (\text{Suc } \beta) \notin \text{fmv-cmd } c \ n$

$\beta \notin \text{fmv-trm } t \text{ (Suc } n) \implies (\text{Suc } \beta) \notin \text{fmv-trm } t \ n$

proof (*induct c and t arbitrary: n and n*)

case (*MVar x1 x2*)

then show *?case*

by *clarsimp (metis UnI1 UnI2 diff-Suc-1 diff-Suc-eq-diff-pred diff-commute diff-is-0-eq nat.simps(3) not-less-eq-eq singletonI)*

qed (*force*)**+**

lemma *flv-drop*:

$\text{flv-trm } t \ k = \{\} \longrightarrow \text{flv-trm } (\text{dropM-trm } t \ j) \ k = \{\}$

$\text{flv-cmd } c \ k = \{\} \longrightarrow \text{flv-cmd } (\text{dropM-cmd } c \ j) \ k = \{\}$

by (*induct t and c arbitrary: j k and j k*) *clarsimp+*

end

1.4 Logical and structural substitution

theory *Substitution*

imports *DeBruijn*

begin

primrec

subst-trm :: [*trm, trm, nat*] \Rightarrow *trm* ($-\text{[-'/-]}^T$ [*300, 0, 0*] *300*) **and**

subst-cmd :: [*cmd, trm, nat*] \Rightarrow *cmd* ($-\text{[-'/-]}^C$ [*300, 0, 0*] *300*)

where

subst-LVar: ($'i$) $[s/k]^T$ =

(*if* $k < i$ *then* $'(i-1)$ *else if* $k = i$ *then* s *else* ($'i$))

| *subst-Lbd*: ($\lambda T : t$) $[s/k]^T$ = $\lambda T : (t[(\text{liftL-trm } s \ 0) / k+1]^T)$

| *subst-App*: ($t \circ u$) $[s/k]^T$ = $t[s/k]^T \circ u[s/k]^T$

| *subst-Mu*: ($\mu T : c$) $[s/k]^T$ = $\mu T : (c[(\text{liftM-trm } s \ 0) / k]^C)$

| *subst-MVar*: ($\langle i \rangle t$) $[s/k]^C$ = $\langle i \rangle (t[s/k]^T)$

Substituting a term for the hole in a context.

primrec *ctxt-subst* :: *ctxt* \Rightarrow *trm* \Rightarrow *trm* **where**

ctxt-subst $\diamond s = s$ |

ctxt-subst ($E \bullet t$) $s = (\text{ctxt-subst } E \ s)^\circ t$

lemma *ctxt-app-subst*:

shows *ctxt-subst* E (*ctxt-subst* F t) = *ctxt-subst* ($E \cdot F$) t

by (*induction E, auto*)

The structural substitution is based on Geuvers and al. [1].

primrec

struct-subst-trm :: [*trm, nat, nat, ctxt*] \Rightarrow *trm* ($-\text{[-=- -]}^T$ [*300, 0, 0*] *300*) **and**

struct-subst-cmd :: [*cmd, nat, nat, ctxt*] \Rightarrow *cmd* ($-\text{[-=- -]}^C$ [*300, 0, 0*] *300*)

where

$$\begin{aligned}
\text{struct-LVar: } ('i)[j=k E]^T &= ('i) \mid \\
\text{struct-Lbd: } (\lambda T : t)[j=k E]^T &= (\lambda T : (t[j=k (\text{liftL-ctxt } E \ 0)]^T)) \mid \\
\text{struct-App: } (t^\circ s)[j=k E]^T &= (t[j=k E]^T)^\circ (s[j=k E]^T) \mid \\
\text{struct-Mu: } (\mu T : c)[j=k E]^T &= \mu T : (c[(j+1)=(k+1) (\text{liftM-ctxt } \\
&E \ 0)]^C) \mid \\
\text{struct-MVar: } (\langle i \rangle t)[j=k E]^C &= \\
&(\text{if } i=j \text{ then } (\langle k \rangle (\text{ctxt-subst } E \ (t[j=k E]^T))) \\
&\text{else } (\text{if } j < i \wedge i \leq k \text{ then } (\langle i-1 \rangle (t[j=k E]^T)) \\
&\text{else } (\text{if } k \leq i \wedge i < j \text{ then } (\langle i+1 \rangle (t[j=k E]^T)) \\
&\text{else } (\langle i \rangle (t[j=k E]^T))))))
\end{aligned}$$

Lifting of lambda and mu variables commute with each other

lemma *liftLM-comm*:

$$\begin{aligned}
\text{liftL-trm } (\text{liftM-trm } t \ n) \ m &= \text{liftM-trm } (\text{liftL-trm } t \ m) \ n \\
\text{liftL-cmd } (\text{liftM-cmd } c \ n) \ m &= \text{liftM-cmd } (\text{liftL-cmd } c \ m) \ n \\
\text{by}(\text{induct } t \ \mathbf{and} \ c \ \text{arbitrary: } n \ m \ \mathbf{and} \ n \ m) \ \text{auto}
\end{aligned}$$

lemma *liftLM-comm-ctxt*:

$$\begin{aligned}
\text{liftL-ctxt } (\text{liftM-ctxt } E \ n) \ m &= \text{liftM-ctxt } (\text{liftL-ctxt } E \ m) \ n \\
\text{by}(\text{induct } E \ \text{arbitrary: } n \ m, \ \text{auto simp add: liftLM-comm})
\end{aligned}$$

Lifting of μ -variables (almost) commutes.

lemma *liftMM-comm*:

$$\begin{aligned}
n \geq m &\implies \text{liftM-trm } (\text{liftM-trm } t \ n) \ m = \text{liftM-trm } (\text{liftM-trm } t \ m) \\
&(\text{Suc } n) \\
n \geq m &\implies \text{liftM-cmd } (\text{liftM-cmd } c \ n) \ m = \text{liftM-cmd } (\text{liftM-cmd } c \ m) \\
&(\text{Suc } n) \\
\text{by}(\text{induct } t \ \mathbf{and} \ c \ \text{arbitrary: } n \ m \ \mathbf{and} \ n \ m) \ \text{auto}
\end{aligned}$$

lemma *liftMM-comm-ctxt*:

$$\begin{aligned}
\text{liftM-ctxt } (\text{liftM-ctxt } E \ n) \ 0 &= \text{liftM-ctxt } (\text{liftM-ctxt } E \ 0) \ (n+1) \\
\text{by}(\text{induct } E \ \text{arbitrary: } n, \ \text{auto simp add: liftMM-comm})
\end{aligned}$$

If a μ variable i doesn't occur in a term or a context, then these remain the same after structural substitution of variable i .

lemma *liftM-struct-subst*:

$$\begin{aligned}
\text{liftM-trm } t \ i[i=i F]^T &= \text{liftM-trm } t \ i \\
\text{liftM-cmd } c \ i[i=i F]^C &= \text{liftM-cmd } c \ i \\
\text{by}(\text{induct } t \ \mathbf{and} \ c \ \text{arbitrary: } i \ F \ \mathbf{and} \ i \ F) \ \text{auto}
\end{aligned}$$

lemma *liftM-ctxt-struct-subst*:

$$\begin{aligned}
(\text{ctxt-subst } (\text{liftM-ctxt } E \ i) \ t)[i=i F]^T &= \text{ctxt-subst } (\text{liftM-ctxt } E \ i) \\
&(t[i=i F]^T) \\
\text{by}(\text{induct } E \ \text{arbitrary: } i \ t \ F; \ \text{force simp add: liftM-struct-subst})
\end{aligned}$$

end

1.5 Reduction relation

theory *Reduction*

imports *Substitution*

begin

inductive *red-term* :: [*trm*, *trm*] \Rightarrow *bool* (**infixl** \longrightarrow 50)
and *red-cmd* :: [*cmd*, *cmd*] \Rightarrow *bool* (**infixl** $_C \longrightarrow$ 50)

where

beta [*intro*]: $(\lambda T : t)^\circ r \longrightarrow t[r/0]^T$ |
struct [*intro*]: $(\mu (T1 \rightarrow T2) : c)^\circ s \longrightarrow \mu T2 : (c[0 = 0 \ (\diamond \bullet \text{liftM-trm } s \ 0)])^C$ |
rename [*intro*]: $\llbracket 0 \notin (\text{fmv-trm } t \ 0) \rrbracket \Longrightarrow (\mu T : (\langle 0 \rangle t)) \longrightarrow \text{dropM-trm } t \ 0$ |
mueta [*intro*]: $\langle i \rangle (\mu T : c) \ _C \longrightarrow (\text{dropM-cmd } (c[0 = i \ \diamond]^C) \ i)$ |

lambda [*intro*]: $\llbracket s \longrightarrow t \rrbracket \Longrightarrow (\lambda T : s) \longrightarrow (\lambda T : t)$ |
appL [*intro*]: $\llbracket s \longrightarrow u \rrbracket \Longrightarrow (s^\circ t) \longrightarrow (u^\circ t)$ |
appR [*intro*]: $\llbracket t \longrightarrow u \rrbracket \Longrightarrow (s^\circ t) \longrightarrow (s^\circ u)$ |
mu [*intro*]: $\llbracket c \ _C \longrightarrow d \rrbracket \Longrightarrow (\mu T : c) \longrightarrow (\mu T : d)$ |
cmd [*intro*]: $\llbracket t \longrightarrow s \rrbracket \Longrightarrow (\langle i \rangle t) \ _C \longrightarrow (\langle i \rangle s)$

inductive-cases *redE* [*elim*]:

$i \longrightarrow s$
 $(\lambda T : t) \longrightarrow s$
 $s^\circ t \longrightarrow u$
 $(\mu T : c) \longrightarrow t$
 $\langle i \rangle t \ _C \longrightarrow c$

Reflexive transitive closure

inductive *beta-rtc-term* :: [*trm*, *trm*] \Rightarrow *bool* (**infixl** \longrightarrow^* 50)

where

refl-term [*iff*]: $s \longrightarrow^* s$ |
step-term: $\llbracket s \longrightarrow t; t \longrightarrow^* u \rrbracket \Longrightarrow s \longrightarrow^* u$

lemma *step-term2*: $\llbracket s \longrightarrow^* t; t \longrightarrow u \rrbracket \Longrightarrow s \longrightarrow^* u$

by (*induct rule: beta-rtc-term.induct*) (*auto intro: step-term*)

inductive *beta-rtc-command* :: [*cmd*, *cmd*] \Rightarrow *bool* (**infixl** $_C \longrightarrow^*$ 50) **where**

refl-command [*iff*]: $c \ _C \longrightarrow^* c$ |
step-command: $c \ _C \longrightarrow d \Longrightarrow d \ _C \longrightarrow^* e \Longrightarrow c \ _C \longrightarrow^* e$

The beta reduction relation is included in the reflexive transitive closure.

lemma *rtc-term-incl* [*intro*]: $s \longrightarrow t \Longrightarrow s \longrightarrow^* t$

by (*meson beta-rtc-term.simps*)

lemma *[intro]*: $c _C \longrightarrow d \implies c _C \longrightarrow^* d$
by(*auto intro: step-command*)

Proof that the reflexive transitive closure as defined above is transitive.

lemma *rtc-term-trans [intro]*: $s \longrightarrow^* t \implies t \longrightarrow^* u \implies s \longrightarrow^* u$
by(*induction rule: beta-rtc-term.induct*) (*auto simp add: step-term*)

lemma *rtc-command-trans*[*intro*]: $c _C \longrightarrow^* d \implies d _C \longrightarrow^* e \implies c _C \longrightarrow^* e$
by(*induction rule: beta-rtc-command.induct*) (*auto simp add: step-command*)

Congruence rules for the reflexive transitive closure.

lemma *rtc-lambda*: $s \longrightarrow^* t \implies (\lambda T : s) \longrightarrow^* (\lambda T : t)$
apply(*induction rule: beta-rtc-term.induct*)
by force (*meson red-term-red-cmd.lambda step-term*)

lemma *rtc-appL*: $s \longrightarrow^* u \implies (s^\circ t) \longrightarrow^* (u^\circ t)$
apply(*induction rule: beta-rtc-term.induct*)
by force (*meson appL step-term*)

end

1.6 Contextual typing

theory *ContextFacts*

imports

Reduction

Types

begin

Naturally, we may wonder when instantiating the hole in a context is type-preserving. To assess this, we define a typing judgement for contexts.

inductive *typing-ctxt* :: $(\text{nat} \Rightarrow \text{type}) \Rightarrow (\text{nat} \Rightarrow \text{type}) \Rightarrow \text{ctxt} \Rightarrow \text{type} \Rightarrow \text{type} \Rightarrow \text{bool}$
 $(-, - \vdash_{\text{ctxt}} - : - \Leftarrow - [50, 50, 50, 50, 50] 50)$

where

type-ctxtEmpty [*intro!*]: $\Gamma, \Delta \vdash_{\text{ctxt}} \diamond : T \Leftarrow T$ |
type-ctxtApp [*intro!*]: $\llbracket \Gamma, \Delta \vdash_{\text{ctxt}} E : (T1 \rightarrow T2) \Leftarrow U; \Gamma, \Delta \vdash_T t : T1 \rrbracket \implies \Gamma, \Delta \vdash_{\text{ctxt}} (E \bullet t) : T2 \Leftarrow U$

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

$\Gamma, \Delta \vdash_{\text{ctxt}} \diamond : T \Leftarrow T$
 $\Gamma, \Delta \vdash_{\text{ctxt}} (E \bullet t) : T \Leftarrow U$

lemma *typing-ctxt-correct1*:

shows $\Gamma, \Delta \vdash_T (\text{ctxt-subst } E \ r) : T \implies \exists U. (\Gamma, \Delta \vdash_T r : U \wedge \Gamma, \Delta \vdash_{\text{ctxt}} E : T \Leftarrow U)$

by(*induction E arbitrary: $\Gamma \ \Delta \ T \ r$; force*)

lemma *typing-ctxt-correct2*:

shows $\Gamma, \Delta \vdash_{\text{ctxt}} E : T \Leftarrow U \implies \Gamma, \Delta \vdash_T r : U \implies \Gamma, \Delta \vdash_T (\text{ctxt-subst } E \ r) : T$

by(*induction arbitrary: r rule: typing-ctxt.induct*) *auto*

lemma *ctxt-subst-basecase*:

$\forall n. c[n = n \ \diamond]^C = c$

$\forall n. t[n = n \ \diamond]^T = t$

by(*induct c and t*) (*auto*)

lemma *ctxt-subst-caseApp*:

$\forall n \ E \ s. (c[n=n \ (\text{liftM-ctxt } E \ n)]^C)[n=n \ (\diamond \bullet (\text{liftM-trm } s \ n))]^C = c[n=n \ ((\text{liftM-ctxt } E \ n) \bullet (\text{liftM-trm } s \ n))]^C$

$\forall n \ E \ s. (t[n=n \ (\text{liftM-ctxt } E \ n)]^T)[n=n \ (\diamond \bullet (\text{liftM-trm } s \ n))]^T = t[n=n \ ((\text{liftM-ctxt } E \ n) \bullet (\text{liftM-trm } s \ n))]^T$

by (*induction c and t*) (*auto simp add: liftLM-comm-ctxt liftLM-comm liftMM-comm-ctxt liftMM-comm liftM-ctxt-struct-subst*)

lemma *ctxt-subst*:

assumes $\Gamma, \Delta \vdash_{\text{ctxt}} E : U \Leftarrow T$

shows $(\text{ctxt-subst } E \ (\mu \ T : c)) \longrightarrow^* \mu \ U : (c[0 = 0 \ (\text{liftM-ctxt } E \ 0)]^C)$

using *assms proof*(*induct E arbitrary: U T $\Gamma \ \Delta \ c$*)

case *ContextEmpty*

have *ctxtEmpty-inv*: $\Gamma, \Delta \vdash_{\text{ctxt}} \diamond : U \Leftarrow T \implies U = T$

by(*cases $\Gamma \ \Delta \ \diamond$ rule: typing-ctxt.cases, fastforce, simp*)

show *?case*

using *ContextEmpty by* (*clarsimp dest!: ctxtEmpty-inv simp: ctxt-subst-basecase*)

next

case (*ContextApp E x*)

then show *?case*

by *clarsimp* (*rule rtc-term-trans, rule rtc-appL, assumption, rule step-term, force,clarsimp simp add: ctxt-subst-caseApp(1)*)

qed

end

1.7 Type preservation

theory *TypePreservation*

imports

ContextFacts

begin

Shifting lambda variables preserves well-typedness.

lemma *liftL-type*:

$\Gamma, \Delta \vdash_T t : T \implies \forall k. \Gamma\langle k:U \rangle, \Delta \vdash_T (\text{liftL-trm } t \ k) : T$

$\Gamma, \Delta \vdash_C c \implies \forall k. \Gamma\langle k:U \rangle, \Delta \vdash_C (\text{liftL-cmd } c \ k)$

by (*induct rule: typing-trm-typing-cmd.inducts*) *auto*

Shifting mu variables preserves well-typedness.

lemma *liftM-type*:

$\Gamma, \Delta \vdash_T t : T \implies \forall k. \Gamma, \Delta\langle k:U \rangle \vdash_T (\text{liftM-trm } t \ k) : T$

$\Gamma, \Delta \vdash_C c \implies \forall k. \Gamma, \Delta\langle k:U \rangle \vdash_C (\text{liftM-cmd } c \ k)$

by(*induct rule: typing-trm-typing-cmd.inducts*) *auto*

lemma *dropM-type*:

$\Gamma, \Delta 1 \vdash_T t : T \implies k \notin \text{fmv-trm } t \ 0 \implies (\forall x. x < k \longrightarrow \Delta 1 \ x = \Delta x)$

$\implies (\forall x. x > k \longrightarrow \Delta 1 \ x = \Delta (x-1)) \implies \Gamma, \Delta \vdash_T \text{dropM-trm } t \ k : T$

$\Gamma, \Delta 1 \vdash_C c \implies k \notin \text{fmv-cmd } c \ 0 \implies (\forall x. x < k \longrightarrow \Delta 1 \ x = \Delta x)$

$\implies (\forall x. x > k \longrightarrow \Delta 1 \ x = \Delta (x-1)) \implies \Gamma, \Delta \vdash_C \text{dropM-cmd } c \ k$

proof (*induct arbitrary: k Δ and k Δ rule: typing-trm-typing-cmd.inducts*)

case (*activate* $\Gamma \ \Delta \ T \ c$)

then show *?case*

apply(*erule-tac* $x = \text{Suc } k$ **in** *meta-allE*)

apply(*insert* *fmv-suc*(1))

apply(*clarsimp simp add: shift-def*)

done

qed *fastforce+*

Lifting λ and μ -variables in contexts preserves contextual typing judgements.

lemma *liftL-ctxt-type*:

assumes $\Gamma, \Delta \vdash_{\text{ctxt}} E : T \Leftarrow U$

shows $\forall k. \Gamma\langle k:T1 \rangle, \Delta \vdash_{\text{ctxt}} (\text{liftL-ctxt } E \ k) : T \Leftarrow U$

using *assms* **by** (*induct rule: typing-ctxt.induct*) (*auto simp add: liftL-type*)

lemma *liftM-ctxt-type*:

assumes $\Gamma, \Delta \vdash_{\text{ctxt}} E : T \Leftarrow U$

shows $\Gamma, \Delta\langle k:T1 \rangle \vdash_{\text{ctxt}} (\text{liftM-ctxt } E \ k) : T \Leftarrow U$

using *assms* **by**(*induct rule: typing-ctxt.induct*) (*auto simp add: liftM-type*)

Substitution lemma for logical substitution.

theorem *subst-type*:

$\Gamma 1, \Delta \vdash_T t : T \implies \Gamma, \Delta \vdash_T r : T1 \implies \Gamma 1 = \Gamma\langle k:T1 \rangle \implies \Gamma, \Delta \vdash_T t[r/k]^T : T$

$\Gamma 1, \Delta \vdash_C c \implies \Gamma, \Delta \vdash_T r : T1 \implies \Gamma 1 = \Gamma\langle k:T1 \rangle \implies \Gamma, \Delta \vdash_C c[r/k]^C$

proof(*induct arbitrary: Γ k T1 r and Γ k r T1 rule: typing-trm-typing-cmd.inducts*)

case (*lambda* $\Gamma \ T1 \ \Delta \ t \ T2$)

then show *?case*

by $-(\text{rotate-tac } 2, \text{drule liftL-type}(1), \text{fastforce})$
 next
 case $(\text{activate } \Gamma \Delta T c)$
 then show $?case$
 by $-(\text{rotate-tac } 2, \text{drule liftM-type}(1), \text{fastforce})$
 qed force+

Substitution lemma for structural substitution. The proof is by induction on the first typing judgement.

lemma *struct-subst-command*:

assumes $\Gamma, \Delta \vdash_T t : T \Delta x = T \Gamma, \Delta' \vdash_{\text{ctxt}} E : U \Leftarrow T1 \Delta = \Delta' \langle \alpha : T1 \rangle$
 $(\Gamma, \Delta' \vdash_{\text{ctxt}} E : U \Leftarrow T1 \implies \Delta = \Delta' \langle \alpha : T1 \rangle \implies \Gamma, \Delta' \langle \beta : U \rangle \vdash_T t[\alpha=\beta (\text{liftM-ctxt } E \beta)]^T : T)$
 shows $\Gamma, (\Delta' \langle \beta : U \rangle) \vdash_C (\langle x \rangle t)[\alpha=\beta (\text{liftM-ctxt } E \beta)]^C$
 using *assms* apply *clarsimp*
 apply $(\text{rule conjI}, \text{force}, \text{clarsimp})+$
 apply (rule conjI)
 apply $(\text{metis liftM-ctxt-type passivate shift-eq typing-ctxt-correct2})$
 apply *force*
 done

theorem *struct-subst-type*:

$\Gamma, \Delta 1 \vdash_T t : T \implies \Gamma, \Delta \vdash_{\text{ctxt}} E : U \Leftarrow T1 \implies \Delta 1 = \Delta \langle \alpha : T1 \rangle \implies \Gamma, \Delta \langle \beta : U \rangle \vdash_T t[\alpha=\beta (\text{liftM-ctxt } E \beta)]^T : T$
 $\Gamma, \Delta 1 \vdash_C c \implies \Gamma, \Delta \vdash_{\text{ctxt}} E : U \Leftarrow T1 \implies \Delta 1 = \Delta \langle \alpha : T1 \rangle \implies \Gamma, \Delta \langle \beta : U \rangle \vdash_C c[\alpha=\beta (\text{liftM-ctxt } E \beta)]^C$

proof *(induct arbitrary: $\Delta T1 E U \beta \alpha$ and $\Delta T1 E U \beta \alpha$ rule: typing-trm-typing-cmd.inducts)*

case $(\text{lambda } \Gamma T1 \Delta t T2)$
 then show $?case$
 by $-(\text{drule liftL-ctxt-type}, \text{fastforce simp: liftLM-comm-ctxt})$
 next
 case $(\text{activate } \Gamma \Delta T c)$
 then show $?case$
 by $-(\text{drule liftM-ctxt-type}, \text{fastforce simp: liftMM-comm-ctxt})$
 next
 case $(\text{passivate } \Gamma \Delta t T x)$
 then show $?case$
 using *struct-subst-command* by *force*
 qed fastforce+

lemma *struct-subst-type-command*: $\Gamma, \Delta 1 \vdash_C c \implies \Gamma, \Delta \vdash_{\text{ctxt}} E : U \Leftarrow T1$

$\implies \Delta 1 = \Delta \langle \alpha : T1 \rangle$
 $\implies \Gamma, \Delta \langle \beta : U \rangle \vdash_C c[\alpha=\beta (\text{liftM-ctxt } E \beta)]^C$
 using *struct-subst-type* by *force*

lemma *dropM-env*:
 $\Gamma, \Delta 1 \vdash_T t[k=x \diamond]^T : T \implies \Delta 1 = \Delta \langle x : (\Delta x) \rangle \implies \Gamma, \Delta \vdash_T$
dropM-trm $(t[k=x \diamond]^T) x : T$
 $\Gamma, \Delta 1 \vdash_C c[k=x \diamond]^C \implies \Delta 1 = \Delta \langle x : (\Delta x) \rangle \implies \Gamma, \Delta \vdash_C$ *dropM-cmd*
 $(c[k=x \diamond]^C) x$
by (*induct t and c arbitrary*: $\Gamma T \Delta 1 x k \Delta$ **and** $\Gamma T \Delta 1 x k \Delta$)
fastforce+

theorem *type-preservation*:
 $\Gamma, \Delta \vdash_T t : T \implies t \longrightarrow s \implies \Gamma, \Delta \vdash_T s : T$
 $\Gamma, \Delta \vdash_C c \implies c \xrightarrow{C} d \implies \Gamma, \Delta \vdash_C d$
proof(*induct arbitrary*: s **and** d *rule*: *typing-trm-typing-cmd.inducts*)
case (*app* $\Gamma \Delta t T1 T2 s s'$)
then show *?case*
apply –
apply (*erule redE*; *clarsimp simp*: *subst-type(1)*)
apply (*subgoal-tac* $\Gamma, \Delta \vdash_T$ *ctxt-subst* $(\diamond \bullet s) (\mu(T1 \rightarrow T2) : c)$
 $: T2)$
apply (*drule typing-ctxt-correct1*, *clarsimp*)
apply (*subgoal-tac* $\Gamma, \Delta \langle 0 : T2 \rangle \vdash_C c[0=0$ (*liftM-ctxt* $(\diamond \bullet s)$
 $0)]^C)$
apply (*clarsimp*, *rule struct-subst-type-command*)
apply *force+*
done
next
case (*passivate* $\Gamma \Delta t T x$)
then show *?case*
apply *clarsimp*
apply (*erule redE*, *clarsimp*)
apply (*drule struct-subst-type-command* [**where** $\beta = x$])
apply (*fastforce simp*: *dropM-env(2)*)
done
qed (*force simp*: *dropM-type*)
end

1.8 Progress

theory *Progress*
imports *TypePreservation*
begin

We say that a term t is in *weak-head-normal* form when one of the following conditions are met:

1. t is a value,
2. there exists α and v such that $t = \mu : \tau.[\alpha] v$ with $\alpha \in \text{fcv}(v)$ whenever $\alpha = 0$.

fun (*sequential*) *is-nf* :: *trm* \Rightarrow *bool* **where**

$is\text{-}nf (\mu U: \langle \beta \rangle v) = (is\text{-}val v \wedge (\beta = 0 \longrightarrow 0 \in fmv\text{-}trm v 0)) \mid$
 $is\text{-}nf v = is\text{-}val v$

lemma *progress'*:

$\Gamma, \Delta \vdash_T t : T \implies lambda\text{-}closed t \implies (\forall s. \neg(t \longrightarrow s)) \implies$
 $is\text{-}nf t$

$\Gamma, \Delta \vdash_C c \implies lambda\text{-}closedC c \implies (\forall \beta t. c = \langle \beta \rangle t \longrightarrow (\forall$
 $d. \neg(t \longrightarrow d))) \longrightarrow is\text{-}nf t$

proof (*induct rule: typing-trm-typing-cmd.inducts*)

case (*app* $\Gamma \Delta t T1 T2 s$)

then show *?case*

by $\neg(erule\ type\text{-}arrow\text{-}elim; force)$

next

case (*activate* $\Gamma \Delta T c$)

then show *?case*

proof(*cases c*)

case (*MVar* αt)

then show *?thesis*

using activate by (*case-tac t; force*)

qed

qed *force+*

theorem *progress*:

assumes $\Gamma, \Delta \vdash_T t : T\ lambda\text{-}closed t$

shows $is\text{-}nf t \vee (\exists s. t \longrightarrow s)$

using *assms by* (*fastforce intro: progress'*)

end

1.9 Peirce

theory *Peirce*

imports *Types*

begin

As an example of our $\lambda\mu$ formalisation, we show show a $\lambda\mu$ -term inhabiting Peirce's Law. The example is due to Parigot [4].

Peirce's law: $((A \rightarrow B) \rightarrow A) \rightarrow A$.

lemma $\Gamma, \Delta \vdash_T \lambda (A \rightarrow B) \rightarrow A: (\mu A: \langle 0 \rangle ((\lambda A: (\mu B: \langle 1 \rangle$
 $\langle 0 \rangle))))))$

$: ((A \rightarrow B) \rightarrow A) \rightarrow A$

by *fastforce*

end

References

- [1] H. Geuvers, R. Krebbers, and J. McKinna. The $\lambda\mu^T$ -calculus. *Annals of Pure and Applied Logic*, 164(6), 2013.
- [2] T. Griffin. A formulae-as-types notion of control. In *POPL*, 1990.
- [3] C. Matache, V. B. F. Gomes, and D. P. Mulligan. $\lambda\mu$ -calculus and muML public Bitbucket repository: https://bitbucket.org/Cristina_Matache/prog-classical-types/, 2017.
- [4] M. Parigot. Lambda-Mu-Calculus: An algorithmic interpretation of Classical Natural Deduction. In *LPAR*, 1992.