The $\lambda\mu$-calculus

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

October 26, 2026

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 . . . 7
1.6 Contextual typing . . . 9
1.7 Type preservation . . . 10
1.8 Progress . . . 12
1.9 Peirce . . . 12

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** ‹→› *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* (‹λ-:-› [*0*, *60*] *60*)
| *App trm trm* (**infix** ‹°› *60*)
| *Mu type cmd* (‹μ-:-› [*0*, *60*] *60*)
and *cmd* =
    MVar nat trm (‹<->-› [*0*, *60*] *60*)

datatype *ctxt* =
  ContextEmpty (‹◊›) |
  ContextApp ctxt trm (**infixl** ‹•› *75*)

primrec *ctxt-app* :: *ctxt* ⇒ *ctxt* ⇒ *ctxt* (**infix** ‹.› *60*) **where**
  ◊ . *F* = *F* |
  (*E* • *t*) . *F* = (*E* . *F*) • *t*

fun *is-val* :: *trm* ⇒ *bool* **where**
  is-val (λ *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 :: $(nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a \Rightarrow nat \Rightarrow 'a$ $(\langle\text{-}\langle\text{-:-}\rangle\rangle\ [90,\ 0,\ 0]\ 91)$
where
$e\langle i{:}a\rangle = (\lambda j.\ \textit{if}\ j < i\ \textit{then}\ e\ j\ \textit{else if}\ j = i\ \textit{then}\ a\ \textit{else}\ e\ (j-1))$

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

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

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

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

inductive *typing-trm* :: $(nat \Rightarrow type) \Rightarrow (nat \Rightarrow type) \Rightarrow trm \Rightarrow type \Rightarrow bool$
$(\langle\text{-}\ ,\ \text{-}\vdash_T\ \text{-}:\ \text{-}\rangle\ [50,\ 50,\ 50,\ 50]\ 50)$
and *typing-cmd* :: $(nat \Rightarrow type) \Rightarrow (nat \Rightarrow type) \Rightarrow cmd \Rightarrow bool$
$(\langle\text{-}\ ,\ \text{-}\vdash_C\ \text{-}\rangle\ [50,\ 50,\ 50]\ 50)$
where
var [*intro*!]: $[\![\ \Gamma\ x = T\ ]\!] \Longrightarrow \Gamma, \Delta \vdash_T\ `x : T\ |$
app [*intro*!]: $[\![\ \Gamma, \Delta \vdash_T t : (T1 \to T2); \Gamma, \Delta \vdash_T s : T1\ ]\!] \Longrightarrow \Gamma, \Delta \vdash_T (t\circ s) : T2\ |$
lambda [*intro*!]: $[\![\ \Gamma\langle 0{:}T1\rangle, \Delta \vdash_T t : T2\ ]\!] \Longrightarrow \Gamma, \Delta \vdash_T (\lambda\ T1 : t) : (T1 \to T2)\ |$
activate [*intro*!]: $[\![\ \Gamma, \Delta\langle 0{:}T\rangle \vdash_C c\ ]\!] \Longrightarrow \Gamma, \Delta \vdash_T (\mu\ T : c) : T\ |$
passivate [*intro*!]: $[\![\ \Gamma, \Delta \vdash_T t : T; \Delta\ x = T]\!] \Longrightarrow \Gamma, \Delta \vdash_C (<x>\ t)$

inductive-cases *typing-elims* [*elim*!]:
$\Gamma, \Delta \vdash_T\ `x : T$
$\Gamma, \Delta \vdash_T t\circ s : T$
$\Gamma, \Delta \vdash_T \lambda\ T1 : t : T$
$\Gamma, \Delta \vdash_T \mu\ T1 : t : T$
$\Gamma, \Delta \vdash_C <x>\ t$

inductive-cases *type-arrow-elim*:
$\Gamma\ , \Delta \vdash_T t : T1 \to T2$

lemma *uniqueness*:
$\Gamma, \Delta \vdash_T t : T1 \Longrightarrow \Gamma, \Delta \vdash_T t : T2 \Longrightarrow T1 = T2$
$\Gamma, \Delta \vdash_C c \Longrightarrow \Gamma, \Delta \vdash_C c$
⟨*proof*⟩

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* ⇒ *nat* ⇒ *nat set*
and *flv-cmd* :: *cmd* ⇒ *nat* ⇒ *nat set*
where
flv-trm (‘*i*) *k* = (*if i*≥*k then* {*i*−*k*} *else* {})
| *flv-trm* (λ *T* : *t*) *k* = *flv-trm t* (*k*+*1*)
| *flv-trm* (*s*°*t*) *k* = (*flv-trm s k*) ∪ (*flv-trm t k*)
| *flv-trm* (μ *T* : *c*) *k* = *flv-cmd c k*
| *flv-cmd* (<*i*> *t*) *k* = *flv-trm t k*

primrec *fmv-trm* :: *trm* ⇒ *nat* ⇒ *nat set*
and *fmv-cmd* :: *cmd* ⇒ *nat* ⇒ *nat set*
where
fmv-trm (‘*i*) *k* = {}
| *fmv-trm* (λ *T* : *t*) *k* = *fmv-trm t k*
| *fmv-trm* (*s*°*t*) *k* = (*fmv-trm s k*) ∪ (*fmv-trm t k*)
| *fmv-trm* (μ *T* : *c*) *k* = *fmv-cmd c* (*k*+*1*)
| *fmv-cmd* (<*i*> *t*) *k* = (*if i*≥*k then* {*i*−*k*} ∪ (*fmv-trm t k*) *else* (*fmv-trm t k*))

abbreviation *lambda-closed* :: - **where**
lambda-closed t ≡ *flv-trm t 0* = {}

abbreviation *lambda-closedC* :: - **where**
lambda-closedC c ≡ *flv-cmd c 0* = {}

Free variables in a context.

primrec *fmv-ctxt* :: *ctxt* ⇒ *nat* ⇒ *nat set* **where**
fmv-ctxt ◊ *k* = {}
| *fmv-ctxt* (*E* • *t*) *k* = (*fmv-ctxt E k*) ∪ (*fmv-trm t k*)

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

primrec
liftL-trm :: [*trm*, *nat*] ⇒ *trm* **and**
liftL-cmd :: [*cmd*, *nat*] ⇒ *cmd*
where
liftL-trm (‘*i*) *k* = (*if i*<*k then* ‘*i else* ‘(*i*+*1*)) |
liftL-trm (λ *T* : *t*) *k* = λ *T* : (*liftL-trm t* (*k*+*1*)) |
liftL-trm (*s* ° *t*) *k* = *liftL-trm s k* ° *liftL-trm t k* |

liftL-trm (μ T : c) k = μ T : (*liftL-cmd* c k) |
liftL-cmd (<i> t) k = <i> (*liftL-trm* t k)

primrec
liftM-trm :: [*trm*, *nat*] ⇒ *trm* **and**
liftM-cmd :: [*cmd*, *nat*] ⇒ *cmd*
where
liftM-trm (‘i) k = ‘i |
liftM-trm (λ T : t) k = λ T : (*liftM-trm* t k) |
liftM-trm (s ∘ t) k = *liftM-trm* s k ∘ *liftM-trm* t k |
liftM-trm (μ T : c) k = μ T : (*liftM-cmd* c (k+*1*)) |
liftM-cmd (<i> t) k =
(*if* i<k *then* (<i> (*liftM-trm* t k)) *else* (<i+*1*> (*liftM-trm* t k)))

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

primrec *liftL-ctxt* :: *ctxt* ⇒ *nat* ⇒ *ctxt* **where**
liftL-ctxt ◊ n = ◊ |
liftL-ctxt (E • t) n = (*liftL-ctxt* E n) • (*liftL-trm* t n)

primrec *liftM-ctxt* :: *ctxt* ⇒ *nat* ⇒ *ctxt* **where**
liftM-ctxt ◊ n = ◊ |
liftM-ctxt (E • t) n = (*liftM-ctxt* E n) • (*liftM-trm* t n)

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

primrec
dropM-trm :: [*trm*, *nat*] ⇒ *trm* **and**
dropM-cmd :: [*cmd*, *nat*] ⇒ *cmd*
where
dropM-trm (‘i) k = ‘i
| *dropM-trm* (λ T : t) k = λ T : (*dropM-trm* t k)
| *dropM-trm* (s ∘ t) k = (*dropM-trm* s k) ∘ (*dropM-trm* t k)
| *dropM-trm* (μ T : c) k = μ T : (*dropM-cmd* c (k+*1*))
| *dropM-cmd* (<i> t) k =
(*if* i>k *then* (<i−*1*> (*dropM-trm* t k)) *else* (<i> (*dropM-trm* t k)))

lemma *fmv-liftL*:
$\beta \notin$ *fmv-trm* t n ⟹ $\beta \notin$ *fmv-trm* (*liftL-trm* t m) n
$\beta \notin$ *fmv-cmd* c n ⟹ $\beta \notin$ *fmv-cmd* (*liftL-cmd* c m) n
⟨*proof*⟩

lemma *fmv-liftL-ctxt*:
$\beta \notin$ *fmv-ctxt* E m ⟹ $\beta \notin$ *fmv-ctxt* (*liftL-ctxt* E n) m
⟨*proof*⟩

lemma *fmv-suc*:
$\beta \notin$ *fmv-cmd* c (*Suc* n) ⟹ (*Suc* β) $\notin$ *fmv-cmd* c n

$\beta \notin$ *fmv-trm t* (*Suc n*) $\Longrightarrow$ (*Suc* β) $\notin$ *fmv-trm t n*
⟨*proof*⟩

lemma *flv-drop*:
flv-trm t k = {} $\longrightarrow$ *flv-trm* (*dropM-trm t j*) *k* = {}
flv-cmd c k = {} $\longrightarrow$ *flv-cmd* (*dropM-cmd c j*) *k* = {}
⟨*proof*⟩

end

1.4 Logical and structural substitution

theory *Substitution*
imports *DeBruijn*
begin

primrec
subst-trm :: [*trm*, *trm*, *nat*] ⇒ *trm* (‹-[-'/-]^T› [*300*, *0*, *0*] *300*) **and**
subst-cmd :: [*cmd*, *trm*, *nat*] ⇒ *cmd* (‹-[-'/-]^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*: (λ *T* : *t*)[*s*/*k*]^T = λ *T* : (*t*[(*liftL-trm s 0*) / *k*+*1*]^T)
| *subst-App*: (*t* ∘ *u*)[*s*/*k*]^T = *t*[*s*/*k*]^T ∘ *u*[*s*/*k*]^T
| *subst-Mu*: (μ *T* : *c*)[*s*/*k*]^T = μ *T* : (*c*[(*liftM-trm s 0*) / *k*]^C)
| *subst-MVar*: (<*i*> *t*)[*s*/*k*]^C = <*i*> (*t*[*s*/*k*]^T)

Substituting a term for the hole in a context.

primrec *ctxt-subst* :: *ctxt* ⇒ *trm* ⇒ *trm* **where**
ctxt-subst ◊ *s* = *s* |
ctxt-subst (*E* • *t*) *s* = (*ctxt-subst E s*)° *t*

lemma *ctxt-app-subst*:
shows *ctxt-subst E* (*ctxt-subst F t*) = *ctxt-subst* (*E* . *F*) *t*
⟨*proof*⟩

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

primrec
struct-subst-trm :: [*trm*, *nat*, *nat*, *ctxt*] ⇒ *trm* (‹-[-=- -]^T› [*300*, *0*, *0*, *0*] *300*) **and**
struct-subst-cmd :: [*cmd*, *nat*, *nat*, *ctxt*] ⇒ *cmd* (‹-[-=- -]^C› [*300*, *0*, *0*, *0*] *300*)
where
struct-LVar: (‘*i*)[*j*=*k E*]^T = (‘*i*) |
struct-Lbd: (λ *T* : *t*)[*j*=*k E*]^T = (λ *T* : (*t*[*j*=*k* (*liftL-ctxt E 0*)]^T)) |
struct-App: (*t*°*s*)[*j*=*k E*]^T = (*t*[*j*=*k E*]^T)°(*s*[*j*=*k E*]^T) |
struct-Mu: (μ *T* : *c*)[*j*=*k E*]^T = μ *T* : (*c*[(*j*+*1*)=(*k*+*1*) (*liftM-ctxt E 0*)]^C) |
struct-MVar: (<*i*> *t*)[*j*=*k E*]^C =

$(\mathit{if}\ i{=}j\ \mathit{then}\ (<k>\ (\mathit{ctxt\text{-}subst}\ E\ (t[j{=}k\ E]^T)))$
$\ \mathit{else}\ (\mathit{if}\ j{<}i \wedge i{\leq}k\ \mathit{then}\ (<i{-}1>\ (t[j{=}k\ E]^T))$
$\ \ \mathit{else}\ (\mathit{if}\ k{\leq}i \wedge i{<}j\ \mathit{then}\ (<i{+}1>\ (t[j{=}k\ E]^T))$
$\ \ \ \mathit{else}\ (<i>\ (t[j{=}k\ E]^T)))))$

Lifting of lambda and mu variables commute with each other

lemma *liftLM-comm*:
$\mathit{liftL\text{-}trm}\ (\mathit{liftM\text{-}trm}\ t\ n)\ m = \mathit{liftM\text{-}trm}\ (\mathit{liftL\text{-}trm}\ t\ m)\ n$
$\mathit{liftL\text{-}cmd}\ (\mathit{liftM\text{-}cmd}\ c\ n)\ m = \mathit{liftM\text{-}cmd}\ (\mathit{liftL\text{-}cmd}\ c\ m)\ n$
⟨*proof*⟩

lemma *liftLM-comm-ctxt*:
$\mathit{liftL\text{-}ctxt}\ (\mathit{liftM\text{-}ctxt}\ E\ n)\ m = \mathit{liftM\text{-}ctxt}\ (\mathit{liftL\text{-}ctxt}\ E\ m)\ n$
⟨*proof*⟩

Lifting of μ-variables (almost) commutes.

lemma *liftMM-comm*:
$n{\geq}m \Longrightarrow \mathit{liftM\text{-}trm}\ (\mathit{liftM\text{-}trm}\ t\ n)\ m = \mathit{liftM\text{-}trm}\ (\mathit{liftM\text{-}trm}\ t\ m)\ (\mathit{Suc}\ n)$
$n{\geq}m \Longrightarrow \mathit{liftM\text{-}cmd}\ (\mathit{liftM\text{-}cmd}\ c\ n)\ m = \mathit{liftM\text{-}cmd}\ (\mathit{liftM\text{-}cmd}\ c\ m)\ (\mathit{Suc}\ n)$
⟨*proof*⟩

lemma *liftMM-comm-ctxt*:
$\mathit{liftM\text{-}ctxt}\ (\mathit{liftM\text{-}ctxt}\ E\ n)\ 0 = \mathit{liftM\text{-}ctxt}\ (\mathit{liftM\text{-}ctxt}\ E\ 0)\ (n{+}1)$
⟨*proof*⟩

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*:
$\mathit{liftM\text{-}trm}\ t\ i[i{=}i\ F]^T = \mathit{liftM\text{-}trm}\ t\ i$
$\mathit{liftM\text{-}cmd}\ c\ i[i{=}i\ F]^C = \mathit{liftM\text{-}cmd}\ c\ i$
⟨*proof*⟩

lemma *liftM-ctxt-struct-subst*:
$(\mathit{ctxt\text{-}subst}\ (\mathit{liftM\text{-}ctxt}\ E\ i)\ t)[i{=}i\ F]^T = \mathit{ctxt\text{-}subst}\ (\mathit{liftM\text{-}ctxt}\ E\ i)\ (t[i{=}i\ F]^T)$
⟨*proof*⟩

end

1.5 Reduction relation

theory *Reduction*
imports *Substitution*
begin

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

where
beta [*intro*]: $(\lambda\ T : t)\circ r \longrightarrow t[r/0]^T$ |
struct [*intro*]: $(\mu\ (T1 \to T2) : c)\circ s \longrightarrow \mu\ T2 : (c[0 = 0\ (\Diamond \bullet (\mathit{liftM\text{-}trm}\ s\ 0))]^C)$ |
rename [*intro*]: $[\![\ 0 \notin (\mathit{fmv\text{-}trm}\ t\ 0)\ ]\!] \Longrightarrow (\mu\ T : (<0> t)) \longrightarrow \mathit{dropM\text{-}trm}\ t\ 0$ |
mueta [*intro*]: $<i> (\mu\ T : c)\ {}_C\!\longrightarrow (\mathit{dropM\text{-}cmd}\ (c[0 = i\ \Diamond]^C)\ i)$ |

lambda [*intro*]: $[\![\ s \longrightarrow t\ ]\!] \Longrightarrow (\lambda\ T : s) \longrightarrow (\lambda\ T : t)$ |
appL [*intro*]: $[\![\ s \longrightarrow u\ ]\!] \Longrightarrow (s\circ t) \longrightarrow (u\circ t)$ |
appR [*intro*]: $[\![\ t \longrightarrow u\ ]\!] \Longrightarrow (s\circ t) \longrightarrow (s\circ u)$ |
mu [*intro*]: $[\![\ c\ {}_C\!\longrightarrow d\ ]\!] \Longrightarrow (\mu\ T : c) \longrightarrow (\mu\ T : d)$ |
cmd [*intro*]: $[\![\ t \longrightarrow s\ ]\!] \Longrightarrow (<i> t)\ {}_C\!\longrightarrow (<i> s)$

inductive-cases *redE* [*elim*]:
$`i \longrightarrow s$
$(\lambda\ T : t) \longrightarrow s$
$s\circ t \longrightarrow u$
$(\mu\ T : c) \longrightarrow t$
$<i> t\ {}_C\!\longrightarrow c$

Reflexive transitive closure

inductive *beta-rtc-term* :: [*trm*, *trm*] ⇒ *bool* (**infixl** ‹$\longrightarrow^*$› 50)
where
refl-term [*iff*]: $s \longrightarrow^* s$ |
step-term: $[\![s \longrightarrow t;\ t \longrightarrow^* u]\!] \Longrightarrow s \longrightarrow^* u$

lemma *step-term2*: $[\![s \longrightarrow^* t;\ t \longrightarrow u]\!] \Longrightarrow s \longrightarrow^* u$
⟨*proof*⟩

inductive *beta-rtc-command* :: [*cmd*, *cmd*] ⇒ *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*]: $s \longrightarrow t \Longrightarrow s \longrightarrow^* t$
⟨*proof*⟩

lemma [*intro*]: $c\ {}_C\!\longrightarrow d \Longrightarrow c\ {}_C\!\longrightarrow^* d$
⟨*proof*⟩

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

lemma *rtc-term-trans* [*intro*]: $s \longrightarrow^* t \Longrightarrow t \longrightarrow^* u \Longrightarrow s \longrightarrow^* u$
⟨*proof*⟩

lemma *rtc-command-trans*[*intro*]: $c\ {}_C\!\longrightarrow^*\ d \Longrightarrow d\ {}_C\!\longrightarrow^*\ e \Longrightarrow c\ {}_C\!\longrightarrow^*\ e$
⟨*proof*⟩

Congruence rules for the reflexive transitive closure.

lemma *rtc-lambda*: $s \longrightarrow^* t \Longrightarrow (\lambda\ T : s) \longrightarrow^* (\lambda\ T : t)$
⟨*proof*⟩

lemma *rtc-appL*: $s \longrightarrow^* u \Longrightarrow (s \circ t) \longrightarrow^* (u \circ t)$
⟨*proof*⟩

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* :: $(nat \Rightarrow type) \Rightarrow (nat \Rightarrow type) \Rightarrow ctxt \Rightarrow type \Rightarrow type \Rightarrow bool$
$(\langle\text{-}\ ,\ \text{-} \vdash_{ctxt} \text{-} : \text{-} \Leftarrow \text{-}\rangle\ [50,\ 50,\ 50,\ 50,\ 50]\ 50)$
where
type-ctxtEmpty [*intro*!]: $\Gamma,\ \Delta \vdash_{ctxt} \Diamond : T \Leftarrow T\ |$
type-ctxtApp [*intro*!]: $[\![\ \Gamma,\ \Delta \vdash_{ctxt} E : (T1 \to T2) \Leftarrow U;\ \Gamma,\ \Delta \vdash_T t : T1\ ]\!] \Longrightarrow \Gamma,\ \Delta \vdash_{ctxt} (E \bullet t) : T2 \Leftarrow U$

inductive-cases *typing-ctxt-elims* [*elim*!]:
$\Gamma,\ \Delta \vdash_{ctxt} \Diamond : T \Leftarrow T$
$\Gamma,\ \Delta \vdash_{ctxt} (E \bullet t) : T \Leftarrow U$

lemma *typing-ctxt-correct1*:
shows $\Gamma,\ \Delta \vdash_T (ctxt\text{-}subst\ E\ r) : T \Longrightarrow \exists\, U.\ (\Gamma,\ \Delta \vdash_T r : U \wedge \Gamma,\ \Delta \vdash_{ctxt} E : T \Leftarrow U)$
⟨*proof*⟩

lemma *typing-ctxt-correct2*:
shows $\Gamma,\ \Delta \vdash_{ctxt} E : T \Leftarrow U \Longrightarrow \Gamma,\ \Delta \vdash_T r : U \Longrightarrow \Gamma,\ \Delta \vdash_T (ctxt\text{-}subst\ E\ r) : T$
⟨*proof*⟩

lemma *ctxt-subst-basecase*:
$\forall\, n.\ c[n = n\ \Diamond]^C = c$

$\forall\, n.\ t[n = n\ \Diamond]^T = \ t$
⟨*proof*⟩

lemma *ctxt-subst-caseApp*:
$\forall\, n\ E\ s.\ (c[n{=}n\ (\mathit{liftM\text{-}ctxt}\ E\ n)]^C)[n{=}n\ (\Diamond \bullet (\mathit{liftM\text{-}trm}\ s\ n))]^C = c[n{=}n\ ((\mathit{liftM\text{-}ctxt}\ E\ n) \bullet (\mathit{liftM\text{-}trm}\ s\ n))]^C$
$\forall\, n\ E\ s.\ (t[n{=}n\ (\mathit{liftM\text{-}ctxt}\ E\ n)]^T)[n{=}n\ (\Diamond \bullet (\mathit{liftM\text{-}trm}\ s\ n))]^T = t[n{=}n\ ((\mathit{liftM\text{-}ctxt}\ E\ n) \bullet (\mathit{liftM\text{-}trm}\ s\ n))]^T$
⟨*proof*⟩

lemma *ctxt-subst*:
assumes $\Gamma, \Delta \vdash_{ctxt} E : U \Leftarrow T$
shows $(\mathit{ctxt\text{-}subst}\ E\ (\mu\ T : c)) \longrightarrow^* \mu\ U : (c[0 = 0\ (\mathit{liftM\text{-}ctxt}\ E\ 0)]^C)$
⟨*proof*⟩

end

1.7 Type preservation

theory *TypePreservation*
imports
ContextFacts
begin

Shifting lambda variables preserves well-typedness.

lemma *liftL-type*:
$\Gamma, \Delta \vdash_T t : T \Longrightarrow \forall\, k.\ \Gamma\langle k{:}U\rangle, \Delta \vdash_T (\mathit{liftL\text{-}trm}\ t\ k) : T$
$\Gamma, \Delta \vdash_C c \Longrightarrow \forall\, k.\ \Gamma\langle k{:}U\rangle, \Delta \vdash_C (\mathit{liftL\text{-}cmd}\ c\ k)$
⟨*proof*⟩

Shifting mu variables preserves well-typedness.

lemma *liftM-type*:
$\Gamma, \Delta \vdash_T t : T \Longrightarrow \forall\, k.\ \Gamma, \Delta\langle k{:}U\rangle \vdash_T (\mathit{liftM\text{-}trm}\ t\ k) : T$
$\Gamma, \Delta \vdash_C c \Longrightarrow \forall\, k.\ \Gamma, \Delta\langle k{:}U\rangle \vdash_C (\mathit{liftM\text{-}cmd}\ c\ k)$
⟨*proof*⟩

lemma *dropM-type*:
$\Gamma\ , \Delta 1 \vdash_T t : T \Longrightarrow k \notin \mathit{fmv\text{-}trm}\ t\ 0 \Longrightarrow (\forall\, x.\ x{<}k \longrightarrow \Delta 1\ x = \Delta\ x)$
$\Longrightarrow (\forall\, x.\ x{>}k \longrightarrow \Delta 1\ x = \Delta\ (x{-}1)) \Longrightarrow \Gamma\ , \Delta \vdash_T \mathit{dropM\text{-}trm}\ t\ k : T$
$\Gamma\ , \Delta 1 \vdash_C c \Longrightarrow k \notin \mathit{fmv\text{-}cmd}\ c\ 0 \Longrightarrow (\forall\, x.\ x{<}k \longrightarrow \Delta 1\ x = \Delta\ x)$
$\Longrightarrow (\forall\, x.\ x{>}k \longrightarrow \Delta 1\ x = \Delta\ (x{-}1)) \Longrightarrow \Gamma\ , \Delta \vdash_C \mathit{dropM\text{-}cmd}\ c\ k$
⟨*proof*⟩

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

lemma *liftL-ctxt-type*:
assumes $\Gamma, \Delta \vdash_{ctxt} E : T \Leftarrow U$
shows $\forall k.\ \Gamma\langle k{:}T1\rangle, \Delta \vdash_{ctxt} (\mathit{liftL\text{-}ctxt}\ E\ k) : T \Leftarrow U$
⟨*proof*⟩

lemma *liftM-ctxt-type*:
assumes $\Gamma, \Delta \vdash_{ctxt} E : T \Leftarrow U$
shows $\Gamma, \Delta\langle k{:}T1\rangle \vdash_{ctxt} (\mathit{liftM\text{-}ctxt}\ E\ k) : T \Leftarrow U$
⟨*proof*⟩

Substitution lemma for logical substitution.

theorem *subst-type*:
$\Gamma 1, \Delta \vdash_T t : T \Longrightarrow \Gamma, \Delta \vdash_T r : T1 \Longrightarrow \Gamma 1 = \Gamma\langle k{:}T1\rangle \Longrightarrow \Gamma, \Delta \vdash_T t[r/k]^T : T$
$\Gamma 1, \Delta \vdash_C c \Longrightarrow \Gamma, \Delta \vdash_T r : T1 \Longrightarrow \Gamma 1 = \Gamma\langle k{:}T1\rangle \Longrightarrow \Gamma, \Delta \vdash_C c[r/k]^C$
⟨*proof*⟩

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

lemma *struct-subst-command*:
assumes $\Gamma, \Delta \vdash_T t : T\ \Delta\ x = T\ \Gamma\ , \Delta' \vdash_{ctxt} E : U \Leftarrow T1\ \Delta = \Delta'\langle\alpha{:}T1\rangle$
$(\Gamma\ , \Delta' \vdash_{ctxt} E : U \Leftarrow T1 \Longrightarrow \Delta = \Delta'\langle\alpha{:}T1\rangle \Longrightarrow \Gamma\ , \Delta'\langle\beta{:}U\rangle \vdash_T t[\alpha{=}\beta\ (\mathit{liftM\text{-}ctxt}\ E\ \beta)]^T : T)$
shows $\Gamma,(\ \Delta'\langle\beta{:}U\rangle) \vdash_C (<x>\ t)[\alpha{=}\beta\ (\mathit{liftM\text{-}ctxt}\ E\ \beta)]^C$
⟨*proof*⟩

theorem *struct-subst-type*:
$\Gamma, \Delta 1 \vdash_T t : T \Longrightarrow \Gamma, \Delta \vdash_{ctxt} E : U \Leftarrow T1 \Longrightarrow \Delta 1 = \Delta\langle\alpha{:}T1\rangle \Longrightarrow \Gamma, \Delta\langle\beta{:}U\rangle \vdash_T t[\alpha{=}\beta\ (\mathit{liftM\text{-}ctxt}\ E\ \beta)]^T : T$
$\Gamma, \Delta 1 \vdash_C c \Longrightarrow \Gamma, \Delta \vdash_{ctxt} E : U \Leftarrow T1 \Longrightarrow \Delta 1 = \Delta\langle\alpha{:}T1\rangle \Longrightarrow \Gamma, \Delta\langle\beta{:}U\rangle \vdash_C c[\alpha{=}\beta\ (\mathit{liftM\text{-}ctxt}\ E\ \beta)]^C$
⟨*proof*⟩

lemma *struct-subst-type-command*: $\Gamma, \Delta 1 \vdash_C c \Longrightarrow \Gamma, \Delta \vdash_{ctxt} E : U \Leftarrow T1$
$\Longrightarrow \Delta 1 = \Delta\langle\alpha{:}T1\rangle$
$\Longrightarrow \Gamma, \Delta\langle\beta{:}U\rangle \vdash_C c[\alpha{=}\beta\ (\mathit{liftM\text{-}ctxt}\ E\ \beta)]^C$
⟨*proof*⟩

lemma *dropM-env*:
$\Gamma, \Delta 1 \vdash_T t[k{=}x\ \Diamond]^T : T \Longrightarrow \Delta 1 = \Delta\langle x{:}(\Delta\ x)\rangle \Longrightarrow \Gamma\ , \Delta \vdash_T \mathit{dropM\text{-}trm}\ (t[k{=}x\ \Diamond]^T)\ x : T$
$\Gamma, \Delta 1 \vdash_C c[k{=}x\ \Diamond]^C \Longrightarrow \Delta 1 = \Delta\langle x{:}(\Delta\ x)\rangle \Longrightarrow \Gamma\ , \Delta \vdash_C \mathit{dropM\text{-}cmd}\ (c[k{=}x\ \Diamond]^C)\ x$
⟨*proof*⟩

theorem *type-preservation*:
$\Gamma, \Delta \vdash_T t : T \Longrightarrow t \longrightarrow s \Longrightarrow \Gamma, \Delta \vdash_T s : T$
$\Gamma, \Delta \vdash_C c \Longrightarrow c\ {}_C\!\longrightarrow d \Longrightarrow \Gamma, \Delta \vdash_C d$
⟨*proof*⟩

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 $\alpha \in fcv(v)$ whenever $\alpha = 0$.

fun (*sequential*) *is-nf* :: *trm* ⇒ *bool* **where**
is-nf (μ *U*: (<β> *v*)) = (*is-val v* ∧ (β = *0* ⟶ *0* ∈ *fmv-trm v 0*)) |
is-nf v = *is-val v*

lemma *progress′*:
$\Gamma, \Delta \vdash_T t : T \Longrightarrow$ *lambda-closed t* ⟹ (∀ *s*. ¬(*t* ⟶ *s*)) ⟹ *is-nf t*
$\Gamma, \Delta \vdash_C c \Longrightarrow$ *lambda-closedC c* ⟹ (∀ β *t*. *c* = (<β> *t*) ⟶ (∀ *d*. ¬(*t* ⟶ *d*)) ⟶ *is-nf t*)
⟨*proof*⟩

theorem *progress*:
assumes $\Gamma, \Delta \vdash_T t : T$ *lambda-closed t*
shows *is-nf t* ∨ (∃ *s*. *t* ⟶ *s*)
⟨*proof*⟩

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: $((A \to B) \to A) \to A$.

lemma $\Gamma, \Delta \vdash_T$ λ (*A*→*B*)→*A*: (μ *A*:(<*0*>((‘*0*) ° (λ *A*: (μ *B*:(<*1*> (‘*0*)))))))

$: ((A \to B) \to A) \to A$
$\langle proof \rangle$

end

References

[1] H. Geuvers, R. Krebbers, and J. McKinna. The $\lambda\mu^{\mathrm{T}}$-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.