The hidden exception handler of Parigot’s $\lambda\mu$-calculus and its completeness properties

Observationally (Böhm) complete

\[ \uparrow \]

Saurin’s extension of $\lambda\mu$-calculus = call-by-name Danvy-Filinski shift-reset “calculus”

\[ \downarrow \]

call-by-value version complete for representing syntactic monads (exceptions, references, ...)

1
Contents

A tour of computational classical logic
- callcc vs try-with
- Operational semantics: the need for $\text{tp}$
- Felleisen $\lambda_c$-calculus and Parigot $\lambda\mu$-calculus
- Curry-Howard and classical logic
- Danvy-Filinski shift/reset calculus vs extending $\lambda\mu$ with (pure) try

Böhm completeness
- David-Py incompleteness of $\lambda\mu$-calculus vs Saurin’s observational completeness
- Saurin’s calculus = call-by-name $\lambda\mu$ + pure try = call-by-name shift/reset calculus
Part I

Computing with classical logic

(a tour of callcc, $\mathcal{A}$, $\mathcal{C}$, try-with/raise, shift/reset, $\mu$, $\hat{\mu}$, ...)
Computing with \texttt{callcc} and \texttt{ptry}


definition Result of \texttt{int}

\begin{verbatim}
let product l =
  try
    let rec aux = function
    | []    -> 1
    | 0 :: l -> raise (Result 0)
    | n :: l -> n * aux l
  in aux l
  with
  Result n -> n
\end{verbatim}

\texttt{ptry} sets a marker and an associated handler in the evaluation stack and \texttt{raise} jumps to the nearest enclosed marker.

\begin{verbatim}
let product l =
  callcc (fun k =>
    let rec aux = function
    | []    -> 1
    | 0 :: l -> throw k 0
    | n :: l -> n * aux l
  in aux l)
\end{verbatim}

\texttt{callcc} memorises the evaluation stack and \texttt{throw} restores the memorised evaluation stack.
ptry binds raise dynamically

exception Result of int

let product l =
  try
    let rec aux = function
    | [] -> 1
    | 0 :: l -> raise (Result 0)
    | n :: l -> n * aux l
    in aux l
  with
    Result n -> n

= exception Result of int

let product l =
  try
    let rec aux = function
    | [] -> 1
    | 0 :: l -> raise (Result 0)
    | n :: l -> n * aux l
    in
    aux l
  with
    Result n -> n
callcc binds its corresponding throw statically

let product l =
callcc (fun k =>
  let rec aux = function
  | []       -> 1
  | 0 :: l   -> throw k 0
  | n :: l   -> n * aux l
  in aux l)

let product l =
let rec aux = function
| []       -> 1
| 0 :: l   -> throw k 0
| n :: l   -> n * aux l
in
  callcc (fun k => aux l)

which is syntactically ill-formed!
A primitive form of try-with: ptry

Syntax

\[
\begin{align*}
t & ::= V \mid tt \mid \text{raise } t \mid \text{ptry } t \\
V & ::= x \mid \lambda x.t
\end{align*}
\]

(terms)

(\[
\begin{align*}
F & ::= \Box \mid F[V \Box] \mid F[\Box t] \mid F[\text{raise } \Box] \\
E & ::= \Box \mid E[\text{ptry } F \Box]
\end{align*}
\]

(local evaluation contexts)

(\[
\begin{align*}
F & ::= \Box \mid F[V \Box] \mid F[\Box t] \mid F[\text{raise } \Box] \\
E & ::= \Box \mid E[\text{ptry } F \Box]
\end{align*}
\]

(global evaluation contexts)

Operational Semantics (aka weak-head reduction)

\[
\begin{align*}
E[(\lambda x.t) u] & \rightarrow E[t[u/x]] \\
E[\text{ptry } F[\text{raise } V]] & \rightarrow E[V] \\
E[\text{ptry } V] & \rightarrow E[V]
\end{align*}
\]

Simulation of ptry/raise from Ocaml’s try-with/raise

\[
\begin{align*}
\text{raise } t & \triangleq \text{raise } (\text{Exc } t) \\
\text{ptry } t & \triangleq \text{try } t \text{ with } \text{Exc } x \rightarrow x
\end{align*}
\]

Simulation of OCaml’s try-with/raise from ptry/raise

\[
\begin{align*}
\text{raise } t & \triangleq \text{raise } t \\
\text{try } t \text{ with } E x \rightarrow u & \triangleq \text{match } (\text{ptry } (\text{Val } t)) \text{ with } \text{Val } x \rightarrow x \mid E x \rightarrow u \mid e \rightarrow \text{raise } e
\end{align*}
\]
Typing callcc/throw and ptry/raise
(standard presentation)

\[
\frac{\Gamma, k : \text{cont } A \vdash t : A}{\Gamma \vdash \text{callcc (fun } k \to t) : A} \quad \frac{\Gamma, k : \text{cont } A \vdash t : A}{\Gamma, k : \text{cont } A \vdash \text{throw } k \; t : B}
\]

\[
\frac{\Gamma \vdash t : \text{exn}}{\Gamma \vdash \text{ptry } t : \text{exn}} \quad \frac{\Gamma \vdash t : \text{exn}}{\Gamma \vdash \text{raise } t : B}
\]
Typing `callcc/throw` and `ptry/raise`  
(generalising the type of exceptions)
Typing \texttt{callcc/throw} and \texttt{ptry/raise}  
(naming the dynamic \texttt{ptry} continuation)

\[
\frac{\Gamma, k : \text{cont} A \vdash t : A}{\Gamma \vdash \text{callcc} (\text{fun} k \to t) : A} \quad \frac{\Gamma, k : \text{cont} A \vdash t : A}{\Gamma, k : \text{cont} A \vdash \text{throw} k \ t : B}
\]

\[
\frac{\Gamma, \text{tp} : \text{cont} A \vdash t : A}{\Gamma \vdash \text{ptry}_{\text{tp}} \ t : A} \quad \frac{\Gamma, \text{tp} : \text{cont} A \vdash t : A}{\Gamma, \text{tp} : \text{cont} A \vdash \text{raise}_{\text{tp}} \ t : B}
\]
Typing `callcc`/`throw` and `ptry`/`raise` (naming the dynamic `ptry` continuation)

\[
\begin{align*}
\Gamma, k : \text{cont } A &\vdash t : A & \Gamma, k : \text{cont } A &\vdash t : A \\
\Gamma &\vdash \text{callcc } (\text{fun } k \to t) : A & \Gamma &\vdash \text{throw } k t : B \\
\Gamma, t_p : \text{cont } A &\vdash t : A & \Gamma, t_p : \text{cont } A &\vdash t : A \\
\Gamma, t_p : \text{cont } T &\vdash \text{ptry}_{t_p} t : A & \Gamma, t_p : \text{cont } A &\vdash \text{raise}_{t_p} t : B
\end{align*}
\]

Generalisation of the type of `ptry` needs type effects on arrows to ensure the type correctness of dynamic binding:

\[
\begin{align*}
\Gamma, t_p : \text{cont } T &\vdash t : A \to_T B & \Gamma, t_p : \text{cont } T &\vdash u : A \\
\Gamma &\vdash tu : B & \Gamma, x : B, t_p : \text{cont } T &\vdash t : C \\
\Gamma, t_p : \text{cont } U &\vdash \lambda x.t : B \to_T C \\
\Gamma, x : A, t_p : \text{cont } T &\vdash x : A
\end{align*}
\]
Simple types: \texttt{callcc} is more expressive than \texttt{ptry} \\
(computing with infinity)

E.g. proof of $\forall f : (\text{nat} \rightarrow \text{bool}) \: \exists b : \text{bool} \: \forall m \: \exists n \geq n \: f(m) = b$

\begin{verbatim}
let pseudo_decide_infinity f =
  callcc (fun k -> (true, fun m1 ->
    callcc (fun k1 -> throw k (false (fun m2 ->
      callcc (fun k2 ->
        let n = max m1 m2 in
        if f n then throw k1 n else throw k2 n))))))
\end{verbatim}

This is executable in SML, Objective Caml (and Scheme).

Any program whose result is a non-functional value and that uses \texttt{pseudo\_decide\_infinity} will yield a (correct) result.

Each call to \texttt{throw} will induce backtracking on the progress of the program that uses \texttt{pseudo\_decide\_infinity}.


Simple types: \texttt{callcc} is more expressive than \texttt{ptry}
(drinkers’ paradox)

E.g. proof of $\forall P : (\text{human} \rightarrow \text{Prop}) \ \exists x : \text{human} \ \forall y : \text{human}, P \ x \rightarrow P \ y$

\begin{verbatim}
(* drinkers : human * (human -> 'a -> 'a)
let drinkers =
    callcc (fun k -> (adam, fun y px ->
        callcc (fun k' -> throw k (y, fun y' py -> throw k' py))))
\end{verbatim}
Simple types: ptry is more expressive than callcc

Derivation of a fixpoint using exceptions of functional type:

type dom = unit -> unit

(* lam : (dom -> dom) -> dom *)
let lam f = fun () -> raise f

(* app : dom -> dom -> dom *)
let app t u = (try (let () = t () in t)) u

(* delta : dom *)
let delta = lam (fun x -> app x x)

(* omega : dom *)
let omega = app delta delta

Typing ptry shows that it raises exceptions of type dom ≜ unit →dom unit which is a recursive type.
Which possible Curry-Howard interpretation for \( \text{ptry/raise} \)?

How to interpret type effects in the logical framework?
Use a canonical type effect? Use \( \perp \)?
Type \( \text{ptry/raise} \) with the rules

\[
\begin{align*}
\Gamma, tp : \text{cont} & \vdash t : \perp \\
\Gamma, tp : \text{cont} & \vdash \text{ptry}_{tp} \ t : \perp \\
\Gamma, tp : \text{cont} & \vdash t : \perp \\
\Gamma, tp : \text{cont} & \vdash \text{raise}_{tp} \ t : B
\end{align*}
\]
The Curry-Howard interpretation for callcc

We have $\lambda x.\text{callcc}(\lambda k.x k) : ((A \rightarrow B) \rightarrow A) \rightarrow A$ which corresponds to Peirce law

Minimal logic + Peirce law is called minimal classical logic

$\lambda$-calculus + callcc and its reduction semantics corresponds to minimal classical logic
The hidden *toplevel* of callcc operational semantics

# let y = callcc (fun k -> fun x -> throw k (fun y -> x+y));;
val y : int -> int = <fun>
# y 3;;
val y : int -> int = <fun> (* !!!! *)

The reason is that the continuation of the definition of \( y \) is “print the value of \( y \)”. Let’s call it \( k_0 \).

The value of \( y \) is \( \text{fun} \ x \to \text{throw} \ k_0 \ (\text{fun} \ y \to \ x+y) \).

When \( y \) is applied, throw is called and it returns to \( k_0 \).

Conventionally, this semantics is expressed using an abortion operator \( A \) which itself hides a call to the toplevel continuation.
Intermezzo: Felleisen’s $C$ operator

Motivated by the possibility to reason on operators such as callcc in Scheme, Felleisen et al introduced the $C$ operator.

**Syntax**

\[ t ::= x \mid \lambda x.t \mid tt \mid C(\lambda k.t) \]

$C$ is equivalent to the combination of callcc and $A$.

\[
C(\lambda k.t) = \text{callcc}(\lambda k.A t) \\
\text{callcc}(\lambda k.t) = C(\lambda k.k t) \\
A t = C(\lambda_.t)
\]

Remark: in $C(\lambda k.t)$, “$\lambda k$.” is part of the syntax. Alternative equivalent definitions of the language are

\[ t ::= x \mid \lambda x.t \mid tt \mid C t \]

or

\[ t ::= x \mid \lambda x.t \mid tt \mid C \]
Parigot’s $\lambda\mu$-calculus
(minimal version)

Use special variables $\alpha$, $\beta$ for denoting continuations.

Syntax

\[
\begin{align*}
t & ::= V \mid t \mid \mu \alpha.c \quad \text{(terms)} \\
c & ::= [\beta]t \quad \text{(commands or states)} \\
V & ::= x \mid \lambda x.t \quad \text{(values)}
\end{align*}
\]

Use structural substitution for continuations:

\[
E[\mu \alpha.c] \rightarrow \mu \alpha.c[[\alpha]E/\alpha]\]

More concisely:

\[
E[\mu \alpha.c] \rightarrow \mu \alpha.c[[\alpha]E/\alpha]\]
Parigot’s $\lambda\mu$-calculus
(expressing callcc)

callcc is approximable:

$$\text{callcc}(\lambda k.(\ldots k\ t \ldots)) \simeq \mu k.[k](\ldots[k]t \ldots)$$

In fact, the actual operational semantics of callcc is:

$$E[\text{callcc}(\lambda k.t)] \rightarrow t[\lambda x.A(E[x])/k] \quad (\ast)$$

The previous approximation of callcc is “too efficient” compared to the actual semantics (such as implemented in Scheme). The correct simulation, in Scheme, is

$$\text{callcc} \triangleq \lambda z.\alpha.[\alpha](z\ \lambda x.\mu_.[\alpha]x)$$

which corresponds to an operator of reification of the evaluation context as a function.

Yet, the full semantics of callcc above requires $A$ and $\lambda\mu$-calculus is not strong enough to express it. Especially, it is not strong enough to simulate the capture of the toplevel continuation performed by the toplevel rule $(\ast)$ above.
Parigot’s $\lambda\mu$-calculus
(adding a denotation for the toplevel continuation)

Syntax

$$
\begin{align*}
t & ::= V \mid tt \mid \mu\alpha.c \quad \text{(terms)} \\
c & ::= [\beta]t \mid [tp]t \quad \text{(commands or states)} \\
V & ::= x \mid \lambda x.t \quad \text{(values)}
\end{align*}
$$

The operator $A$ can now be expressed: $A \triangleq \lambda x.\mu_.[tp]x$ and the toplevel reduction rule of callcc gets simulable:

$$
\begin{align*}
[tp] E[\text{callcc}(\lambda k.t)] & \equiv [tp] E[(\lambda z.\mu\alpha.[\alpha](z \lambda x.\mu_.[\alpha]x)) \lambda k.t] \\
& \downarrow^* \\
[tp] E[t[\lambda x.A(E[x])/k]] & \equiv [tp] E[t[\lambda x.\mu_.[tp]E[x]/k]]
\end{align*}
$$

$\lambda\mu$-calculus extended with tp can faithfully simulate callcc or Felleisen's $\lambda_c$-calculus. Moreover, it
- is able to substitute continuations directly as evaluation contexts
- allows notations for continuation constants
- is able to express states of abstract machine (tp plays the rôle of the bottom of the stack)
- has nice reduction and operational properties (almost as nice as $\tilde{\lambda}\mu\tilde{\mu}$!) [Ariola-Herbelin 2007]

An example of “inefficiency” in call-by-value:

```latex
let \text{loop}() = \text{callcc}(\lambda k.&)x(\text{loop}()) : \text{loop}() \rightarrow (\lambda x.Ax) (\text{loop}()) \rightarrow (\lambda x.Ax)(\lambda x.Ax)(\text{loop}()) \rightarrow \ldots
```

```latex
let \text{loop}() = \mu k.\kx(\text{loop}()) : [tp](\text{loop}()) \rightarrow [tp]\text{loop}() \rightarrow [tp]\text{loop}() \rightarrow \ldots
```

21
Danvy and Filinski’s shift and reset [1989]

The operator reset locally “resets” the toplevel and delimits the current continuation of a computation. The operator shift captures the current delimited continuation and composes it with the continuation at the places it is invoked.

**Syntax**

\[
\begin{align*}
t & ::= V \mid tt \mid S(\lambda k.t) \mid \langle t \rangle & \text{(terms)} \\
V & ::= x \mid \lambda x.t & \text{(values)} \\
F\Box & ::= \Box \mid F[V\Box] \mid F[t\Box] & \text{(local ev. contexts)} \\
E\Box & ::= \Box \mid E[\text{reset}F\Box] & \text{(global ev. contexts)}
\end{align*}
\]

Historically, the semantics of shift/reset was defined by continuation-passing-style translation (CPS). Its operational semantics is now well established.

**Operational Semantics**

\[
\begin{align*}
E[(\lambda x.t) u] & \rightarrow E[t[u/x]] \\
E[\langle F[S(\lambda k.t)] \rangle] & \rightarrow E[t[\lambda x.\langle F[x]\rangle/k]] \\
E[\langle V \rangle] & \rightarrow E[V]
\end{align*}
\]
Application: normalisation by evaluation with boolean type [Danvy 1996]

type term =
| Abs of string * term
| Var of string | App of term * term
| True | False | If of term * term * term

type types =
| Atom (* |Atom| = term *)
| Arrow of types * types (* |Arrow(T1,T2)| = |T1| -> |T2| *)
| Bool (* |Bool| = bool *)

(* up : (T:types)term(T)->|T| *)
let rec up tt t = match tt with
| Atom -> t
| Arrow (tt1,tt2) -> fun x -> up tt2 (App (t,down tt1 x))
| Bool -> shift (fun k -> If (t,k true,k false))

(* down : (T:types)|T|->term(T) *)
and down tt mt = match tt with
| Atom -> mt
| Arrow (tt1,tt2) -> let s = fresh () in
         Abs (s,reset (down tt2 (mt (up tt1 (Var s)))))
| Bool -> if mt then True else False

23
Filinski [1994]: shift/reset can simulate all concrete monads in direct style
(the example of references)

The monad that simulates a reference of type $S$

$$T(A) = S \rightarrow A \times S$$
$$\eta = \lambda x.\lambda s. (x, s) : A \rightarrow T(A)$$
$$* = \lambda f.\lambda x.\lambda s. \text{let } (x, s) = x \; s \; \text{in } (f \; x \; s) : (A \rightarrow T(B)) \rightarrow T(A) \rightarrow T(B)$$

The resulting encoding of read and write

$$\text{read} \triangleq \lambda().S(\lambda k.\lambda s. (k \; s \; s)) : \text{unit} \rightarrow S$$
$$\text{write} \triangleq \lambda s. S(\lambda k.\lambda _. (k () s)) : S \rightarrow \text{unit}$$

The operators can be safely added to (call-by-value) classical logic preserving subject reduction. In general, if $T$ is atomic, normalisability is preserved. In the case of the state monad, $T$ is functional and nothing prevents (a priori) to derive a fixpoint.
Comparing shift and reset to the other operators

One can observe that reset behaves the same as ptry and that \(A\) behaves as raise. The correspondences are as follows:

\[
\begin{align*}
  A t & \triangleq S(\lambda_.t) \\
  \text{callcc } (\lambda k.t) & \triangleq S(\lambda k.k [\lambda x. A (k x)/k]) \\
  C (\lambda k.t) & \triangleq S(\lambda k.t [\lambda x. A (k x)/k]) \\
  \text{ptry } t & \triangleq \langle t \rangle \\
  \text{raise } t & \triangleq A t
\end{align*}
\]

Conversely, shift can be macro-defined from callcc, \(A/\text{raise}\) and ptry/\text{reset}:

\[
S(\lambda k.t) \triangleq \text{callcc } (\lambda k.A(t[\lambda x.(k x)/k]))
\]

The shift/reset calculus can hence be seen as the marriage between the ptry/raise calculus and the callcc/throw calculus.

Warning: callcc, as it occurs in programming languages (that do have exceptions) captures the full continuations and not just the delimited one. It is better to use the name \(K\) to denote the variant that captures the current delimited continuation.
\(\lambda\mu\widehat{\mu}tp\)-calculus
(a fine-grained shift/reset calculus)

The structural substitution, the presence of continuation variables and the distinction between commands and terms make of \(\lambda\mu\)-calculus a good candidate for finely analysing the shift/reset calculus.

Syntax

\[
\begin{align*}
t &::= V | tt | \mu\alpha.c | \widehat{\mu}tp.c \\
c &::= \beta t | [tp]t \\
V &::= x | \lambda x.t
\end{align*}
\]

Macro-definability

\[
\begin{align*}
\langle t \rangle &\triangleq \widehat{\mu}tp.[tp]t \\
A t &\triangleq \mu_.[tp]t \\
S(\lambda k.t) &\triangleq \mu\alpha.[tp](t[\lambda x.\widehat{\mu}tp.[\alpha]x/k]) \\
C(\lambda k.t) &\triangleq \mu\alpha.[tp](t[\lambda x.\mu_.[\alpha]x/k]) \\
callcc(\lambda k.t) &\triangleq \mu\alpha.[\alpha](t[\lambda x.\mu_.[\alpha]x/k])
\end{align*}
\]

The fourth combination \(\mu\alpha.[\alpha](t[\lambda x.\widehat{\mu}tp.[\alpha]x/k])\) has no name (to our knowledge), it is equivalent to \(S(\lambda k.t)\).
\[ \lambda \mu \hat{\mu} \text{-calculus} \]
\( \text{(a fine-grained shift/reset calculus)} \)

**Semantics**

\[
\begin{align*}
(\beta_v) \quad & (\lambda x.t) V & \rightarrow & \ t[V/x] \\
(\eta_v) \quad & \lambda x.(V x) & \rightarrow & \ t & \text{if } x \text{ not free in } V \\
(\mu_{app}) \quad & (\mu \alpha. c) u & \rightarrow & \mu \alpha. c[[\alpha](\Box u)]/\alpha \\
(\mu'_{app}) \quad & V (\mu \alpha. c) & \rightarrow & \mu \alpha. c[[\alpha](V \Box)]/\alpha \\
(\mu_{var}) \quad & [\beta] \mu \alpha. c & \rightarrow & \ c[\beta/\alpha] & \text{also if } \beta \text{ is tp} \\
(\eta_{\mu}) \quad & \mu \alpha.[\alpha]t & \rightarrow & \ t & \text{if } \alpha \text{ not free in } t \\
(\hat{\mu}_{var}) \quad & [\text{tp}] \hat{\mu} \text{tp}.c & \rightarrow & \ c \\
(\eta_{\hat{\mu}v}) \quad & \hat{\mu} \text{tp.}[\text{tp}]V & \rightarrow & \ V & \text{even if tp occurs in } t \\

(\text{let}_{\hat{\mu}}) \quad & \hat{\mu} \text{tp.}[\beta](\lambda x.t) \hat{\mu} \text{tp}.c & \rightarrow & \ (\lambda x.\hat{\mu} \text{tp.}[\beta]t) \hat{\mu} \text{tp}.c \\
(\text{let}_{\mu}) \quad & (\lambda x.\mu \alpha.[\beta]t) u & \rightarrow & \mu \alpha.[\beta](\lambda x.t) u \\
(\text{let}_{app}) \quad & (\lambda x.t) u u' & \rightarrow & \ (\lambda x.(t u')) u \\
(\text{let}'_{app}) \quad & V ((\lambda x.t) u) & \rightarrow & \ (\lambda x.(V t)) u \\
(\eta_{\text{let}}) \quad & (\lambda x.x) t & \rightarrow & \ t
\end{align*}
\]
There are several possible systems of simple types and they all depend on a toplevel type, say $T$. They assign the following types to operators:

\[
\begin{align*}
\langle t \rangle & : T \to T \\
A t & : T \to A \\
S(\lambda k.t) & : ((A \to T) \to T) \to A \\
C(\lambda k.t) & : ((A \to B) \to T) \to A \\
callcc(\lambda k.t) & : ((A \to B) \to A) \to A
\end{align*}
\]

A Curry-Howard correspondence holds if the toplevel type $T$ is taken to be $\bot$, in which case, we get types compatible with Griffin's seminal observations [1990] on Curry-Howard for classical logic.

\[
\begin{align*}
\langle t \rangle & : \bot \to \bot \quad \text{(no logical content)} \\
A t & : \bot \to A \quad \text{(ex falso quodlibet)} \\
S(\lambda k.t) & : ((A \to \bot) \to \bot) \to A \quad \text{(double negation elimination)} \\
C(\lambda k.t) & : ((A \to B) \to \bot) \to A \quad \text{(an instance of it is double negation elimination)} \\
callcc(\lambda k.t) & : ((A \to B) \to A) \to A \quad \text{(Peirce law)}
\end{align*}
\]
Part II

Observational (Böhm) completeness in (call-by-name) $\lambda\mu$-calculus
Failure of separability in call-by-name $\lambda\mu$-calculus

The original syntax of $\lambda\mu$-calculus:

$$
t ::= x \mid \lambda x.t \mid t t \mid \mu\alpha.c \\
c ::= [\alpha]t
$$

Extending the call-by-name reduction semantics with $\eta$ rules:

$$(\beta) \quad (\lambda x.t) u \rightarrow t[u/x]$$

$$(\eta) \quad \lambda x.(t x) \rightarrow t \quad \text{if } x \text{ not free in } t$$

$$(\mu_{\text{app}}) \quad (\mu\alpha.c) u \rightarrow \mu\alpha.c[\alpha](\square u)/\alpha$$

$$(\mu_{\text{var}}) \quad [\beta]\mu\alpha.c \rightarrow c[\beta/\alpha]$$

$$(\eta_{\mu}) \quad \mu\alpha.[\alpha]t \rightarrow t \quad \text{if } \alpha \text{ not free in } t$$

David-Py [2001]: There exist two closed terms $W_0$ and $W_1$ in $\lambda\mu$-calculus that are not equal w.r.t. the equalities $\beta$, $\eta$, $\mu_{\text{app}}$, $\mu_{\text{var}}$ and $\eta_{\mu}$ but whose observational behaviour is not separable.
Success of separability in Saurin’s $\lambda\mu$-calculus

A slightly different syntax (originally from de Groote):

\[ t ::= x \mid \lambda x.t \mid tt \mid \mu\alpha.t \mid [\beta]t \]

The same (apparent) reduction rules:

\[
\begin{align*}
(\beta) \quad (\lambda x.t) u & \rightarrow t[u/x] \\
(\eta) \quad \lambda x.(tx) & \rightarrow t & \text{if } x \text{ not free in } t \\
(\mu_{\text{app}}) \quad (\mu\alpha.t) u & \rightarrow \mu\alpha.t[\alpha](\Box u)/\alpha \\
(\mu_{\text{var}}) \quad [\beta]\mu\alpha.t & \rightarrow t[\beta/\alpha] \\
(\eta_{\mu}) \quad \mu\alpha.[\alpha]t & \rightarrow t & \text{if } \alpha \text{ not free in } t
\end{align*}
\]

But a major difference: \( t \; [\beta]\mu\alpha.u \rightarrow t(u[\beta/\alpha]) \). We can get rid of $\mu$ what was not possible in the original $\lambda\mu$-calculus (indeed the left-hand side is even not expressible).

Saurin [2005]: The modified syntax of $\lambda\mu$-calculus with the equalities $\beta$, $\eta$, $\mu_{\text{app}}$, $\mu_{\text{var}}$ and $\eta_{\mu}$ has the Böhm separability property.
From Saurin’s \( \lambda \mu \)-calculus to call-by-name \( \lambda \mu \hat{\mu}tp \)-calculus

In Saurin’s calculus, the syntactic distinction between terms and commands is lost, making difficult to understand it computationally (e.g., in an abstract machine, i.e., in \( \overline{\lambda \mu \tilde{\mu}} \)-calculus).

The constructions \( \hat{\mu}tp \) and \( [tp] \) can be proved to be adequate coercions from Saurin’s calculus to a calculus well-suited for computation.

\begin{align*}
\mu \alpha . t & \triangleq \mu \alpha . [tp] t \\
[\alpha] t & \triangleq \hat{\mu}tp . [\alpha] t
\end{align*}

Macro-definition of Saurin’s calculus on top of \( \lambda \mu \hat{\mu}tp \)
Call-by-name $\lambda\mu\widehat{\mu}\text{tp}$-calculus

$$
\begin{align*}
(\beta) & \quad (\lambda x. t) u \rightarrow t[u/x] \\
(\eta) & \quad \lambda x. (tx) \rightarrow t & \text{if } x \text{ not free in } t \\
(\mu_{\text{app}}) & \quad (\mu\alpha. c) u \rightarrow \mu\alpha. c[[\alpha](\Box u))/\alpha] \\
(\mu_{\text{var}}^n) & \quad [\beta]\mu\alpha. c \rightarrow c[\beta/\alpha] & \beta \neq \text{tp} \\
(\eta\mu) & \quad \mu\alpha.[\alpha]t \rightarrow t & \text{if } \alpha \text{ not free in } t \\
(\widehat{\mu}_{\text{var}}) & \quad [\text{tp}]\widehat{\mu}\text{tp}. c \rightarrow c \\
(\eta\widehat{\mu}) & \quad \widehat{\mu}\text{tp}.[\text{tp}]t \rightarrow t & \text{even if tp occurs in } t
\end{align*}
$$

Obviously, we have:

**Proposition** $t = u$ in Saurin's $\lambda\mu$-calculus iff $t = u$ in $\lambda\mu\widehat{\mu}\text{tp}$-calculus.

**Corollary** $\lambda\mu\widehat{\mu}\text{tp}$-calculus is observationally complete on finite normal forms.

That the rules above are relevant for what can be considered as a call-by-name version of the shift/reset-calculus can be seen from the operational semantics and from the continuation-passing-style semantics of call-by-name $\lambda\mu\widehat{\mu}\text{tp}$-calculus.
Classification of the reduction semantics of $\lambda\mu\widehat{\mu}$tp-calculus

The fundamental critical pair of computation

$$(\lambda x.t)\ (\mu\alpha.c)$$

$(\beta_v) + (\mu'_{app}) + (\eta_{\widehat{\mu}v}) + (\eta_v)$

$(\beta) + (\eta_{\widehat{\mu}}) + (\eta)$

Subsidiary choice

$$(\lambda x.t)\ (\widehat{\mu}tp.c)$$

$(\widehat{\mu}$ not value) $\nearrow$

$\langle$ shift/lazy reset $\rangle$

(Sabry)

cps-completion (Sabry)

$\Lambda\mu$

(CBN shift/reset)

(Danvy)

(Böhm-completion (Saurin))

$(\widehat{\mu}$ value) $
earrow$

$\langle$ shift/reset $\rangle$

(Danvy-Filinski)

cps-completion (Kameyama-Hasegawa)

typed “domain”-completion (Sitaram-Felleisen)

(CBN)

$(\beta) + (\eta_{\widehat{\mu}}) + (\eta)$

$(\mu\alpha.c)$

(Sabry) (Danvy-Filinski)

Sabry (Danvy) (de Groote/Saurin)
Abstract machine for call-by-name $\lambda\mu\hat{\mu}tp$-calculus

The language of the call-by-name abstract machine is an extension with explicit environments of the language of $\lambda\mu\hat{\mu}tp$. We need an extra constant of evaluation context that we write $\epsilon$. It is defined by:

$$
\begin{align*}
K & ::= \alpha[e] | t[e] \cdot K \\
[S] & ::= [] | [tp = K; S] \\
[e] & ::= [] | [x = t[e]; e] | [\alpha = K; e] \\
s & ::= c[e] [S] | t[e] K [S]
\end{align*}
$$

(linear ev. contexts) (dynamic environment) (environments) (states)
Abstract machine for call-by-name $\lambda\mu\hat{\mu}tp$-calculus (continued)

The evaluation rules can be split into two categories: the rules giving priority to the evaluation of context (commands of the form $[k]t[ e] S$) and the ones giving priority to the term (commands of the form $t[e] K S$). We write $e(\alpha)$ for the binding of $\alpha$ in $e$ and similarly for $e(x)$.

Control given to the evaluation context

1. $[tp]t[ e] \quad tp = K; S \quad \rightarrow \quad t[ e] K \quad [S]$
2. $[\alpha]t[ e] \quad [S] \quad \rightarrow \quad t[ e] \alpha[e] \quad [S]$
3. $[tp]t[ e] [] \quad \rightarrow \quad \text{stop on} \ [tp]t[ e]$

Control given to the term

1. $x [ e] K \quad [S] \quad \rightarrow \quad t \quad [e'] K \quad [S] \quad \text{if} \ e(x) = t[e']$
2. $x [ e] K \quad [S] \quad \rightarrow \quad \text{stop on} \ S*[K*[x]] \quad \text{if} \ x \text{ not bound in } e$
3. $\lambda x.t[ e] \quad K \quad [S] \quad \rightarrow \quad \lambda x.t[ e] \quad K \quad [S]$
4. $tu[ e] \quad K \quad [S] \quad \rightarrow \quad t \quad [e] \ u[e] \cdot K \quad [S]$
5. $\mu\alpha.c[ e] \quad K \quad [S] \quad \rightarrow \quad c \quad [\alpha = K; e] \quad [S]$
6. $\hat{\mu}tp.c[ e] \quad K \quad [S] \quad \rightarrow \quad c \quad [e] \quad [tp = K; S]$

Control given to the “linear” evaluation context

1. $\lambda x.t[e] \quad u \cdot K \quad [S] \quad \rightarrow \quad t \quad [x = u; e] \quad K \quad [S]$
2. $\lambda x.t[e] \quad \alpha[e'] \quad [S] \quad \rightarrow \quad \lambda x.t \quad [e] \quad K \quad [S] \quad \text{if} \ e'(\alpha) = K$
3. $\lambda x.t[e] \quad \alpha[e'] \quad [S] \quad \rightarrow \quad \text{stop on} \ S*[[\alpha](\lambda x.t[e])] \quad \text{if} \ \alpha \text{ not bound in } e'$

To evaluate $t$, we need a linear toplevel free variables distinct from tp (which is not linear). Let $e$ be this variables. Then, the machine starts with the following initial state:

$$t[ ] \ e[ ] \ [ ]$$

Note: terminal states are defined by

- $[\alpha]^* = [\alpha][\square]$  
- $(t[e] \cdot K)^* = K^*[\square \ t[e]]$  
- $[]^* = \square$  
- $[tp = K; S]^* = [S]^*\hat{\mu}tp.K^*$
Abstract machine for call-by-value $\lambda\mu\tilde{\mu}tp$-calculus

The language of the abstract machine is an extension with explicit environments of the language of $\lambda\mu\tilde{\mu}tp$. It is defined by:

$$
K ::= k[e] \mid t[e] \cdot K \mid \tilde{x}.(W \cdot x \cdot K) \quad \text{(ev. contexts)}
$$

$$
[S] ::= [] \mid [\text{tp} = K; S] \quad \text{(dynamic environment)}
$$

$$
W ::= V[e] \quad \text{(closure)}
$$

$$
[e] ::= [] \mid [x = W; e] \mid [\alpha = K; e] \quad \text{(environments)}
$$

$$
k ::= \alpha \mid \text{tp} \quad \text{(ev. context variables)}
$$

$$
s ::= W \cdot K \cdot [S] \mid t[e] \cdot K \cdot [S] \quad \text{(states)}
$$
Abstract machine for call-by-value $\lambda\mu\hat{\mu}\text{tp}$-calculus (continued)

The evaluation rules can be split into two categories: the rules giving priority to the evaluation of context (commands of the form $W\ K\ S$) and the ones giving priority to the term (commands of the form $t[e]\ K\ S$). We write $e(\alpha)$ for the binding of $\alpha$ in $e$ and similarly for $e(x)$.

Control given to the evaluation context

\[
\begin{align*}
W\ \text{tp}[e] & \quad [\text{tp} = K; S] \rightarrow W\ K\ [S] \\
W\ \text{tp}[e] & \quad [\ ] \quad \rightarrow \ \text{stop on } [\text{tp}]W \\
W\ \alpha[e] & \quad [S] \rightarrow W\ K\ [S] \quad \text{if } e(\alpha) = K \\
W\ \alpha[e] & \quad [S] \rightarrow \text{stop on } S^*[\alpha][W] \quad \text{if } \alpha \text{ not bound in } e \\
W\ t[e]\ K & \quad [S] \rightarrow t\ [e]\ \tilde{\mu}x.(W\ x\ K)\ [S] \\
W\ \tilde{\mu}x.(V[e]\ x\ K) & \quad [S] \rightarrow V\ [e]\ W\ K\ [S]
\end{align*}
\]

Control given to the term

\[
\begin{align*}
V\ [e]\ K\ [S] & \rightarrow V[e]\ K\ [S] \\
t\ u & \quad [e]\ K\ [S] \rightarrow t\ [e]\ u[e]\ K\ [S] \\
\mu\alpha.[k]t & \quad [e]\ K\ [S] \rightarrow t\ [\alpha = K; e]\ k[\alpha = K; e]\ [S] \\
\tilde{\mu}\text{tp}.[k]t & \quad [e]\ K\ [S] \rightarrow t\ [e]\ k[e]\ [\text{tp} = K; S]
\end{align*}
\]

Control given to the functional value

\[
\begin{align*}
\lambda x.t & \quad [e]\ W\ K\ [S] \rightarrow t\ [x = W; e]\ K\ [S] \\
x & \quad [e]\ W\ K\ [S] \rightarrow V\ [e']\ W\ K\ [S] \quad \text{if } e(x) = V[e'] \\
x & \quad [e]\ W\ K\ [S] \rightarrow \text{stop on } S^*[K^*[x\ W]] \quad \text{otherwise}
\end{align*}
\]

To evaluate $t$, the machine starts with the following initial state:

\[t\ [\ ]\ \text{tp}[\ ]\ [\ ]\]
References


