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, ...)

Contents

A tour of computational classical logic

- `callcc` vs `try-with`
- Operational semantics: the need for tp
- Felleisen $\lambda_{\mathcal{C}}$-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`, μ, $\widehat{\mu}$, ...)

Computing with `callcc` and `ptry`

```
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
```

$\simeq$

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

`ptry` sets a marker and an associated handler in the evaluation stack and `raise` jumps to the nearest enclosed marker.

`callcc` memorises the evaluation stack and `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 =
  let rec aux = function
  | []     -> 1
  | 0 :: l -> raise (Result 0)
  | n :: l -> n * aux l
  in
  try
    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)
```

$\neq$

```
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

$$
\begin{array}{llll}
t & ::= & V \mid t\,t \mid \texttt{raise}\,t \mid \texttt{ptry}\,t & \text{(terms)} \\
V & ::= & x \mid \lambda x.t & \text{(values)} \\
\\
F & ::= & \square \mid F[V\ \square] \mid F[\square\ t] \mid F[\texttt{raise}\ \square] & \text{(local evaluation contexts)} \\
E & ::= & \square \mid E[\texttt{ptry}\,F] & \text{(global evaluation contexts)}
\end{array}
$$

Operational Semantics (aka weak-head reduction)

$$
\begin{array}{lcl}
E[(\lambda x.t)\ u] & \rightarrow & E[t[u/x]] \\
E[\texttt{ptry}\ F[\texttt{raise}\ V]] & \rightarrow & E[V] \\
E[\texttt{ptry}\ V] & \rightarrow & E[V]
\end{array}
$$

Simulation of `ptry`/`raise` *from Ocaml's* `try-with`/`raise`

$$
\begin{array}{lcl}
\texttt{raise}\ t & \triangleq & \texttt{raise}\ (\texttt{Exc}\ t) \\
\texttt{ptry}\,t & \triangleq & \texttt{try}\ t\ \texttt{with}\ \texttt{Exc}\ x\ \rightarrow\ x
\end{array}
$$

Simulation of OCaml's `try-with`/`raise` *from* `ptry`/`raise`

$$
\begin{array}{lcl}
\texttt{raise}\ t & \triangleq & \texttt{raise}\ t \\
\texttt{try}\ t\ \texttt{with}\ E\ x\ \rightarrow\ u & \triangleq & \texttt{match}\ (\texttt{ptry}\ (\texttt{Val}\ t))\ \texttt{with}\ \texttt{Val}\ x\ \rightarrow\ x\ \mid\ E\ x\ \rightarrow u \mid e \rightarrow \texttt{raise}\ e
\end{array}
$$

Typing `callcc`/`throw` and `ptry`/`raise` (standard presentation)

$$\frac{\Gamma, k : \texttt{cont}\ A \vdash t : A}{\Gamma \vdash \texttt{callcc}\ (\texttt{fun}\ k\ \rightarrow\ t) : A} \qquad \frac{\Gamma, k : \texttt{cont}\ A \vdash t : A}{\Gamma, k : \texttt{cont}\ A \vdash \texttt{throw}\ k\ t : B}$$

$$\frac{\Gamma \vdash t : \texttt{exn}}{\Gamma \vdash \texttt{ptry}\ t : \texttt{exn}} \qquad \frac{\Gamma \vdash t : \texttt{exn}}{\Gamma \vdash \texttt{raise}\ t : B}$$

Typing `callcc`/`throw` and `ptry`/`raise`
(generalising the type of exceptions)

$$\frac{\Gamma, k : \texttt{cont}\ A \vdash t : A}{\Gamma \vdash \texttt{callcc}\ (\texttt{fun}\ k\ \rightarrow\ t) : A} \qquad \frac{\Gamma, k : \texttt{cont}\ A \vdash t : A}{\Gamma, k : \texttt{cont}\ A \vdash \texttt{throw}\ k\ t : B}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \texttt{ptry}\ t : A} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash \texttt{raise}\ t : B}$$

Typing `callcc`/`throw` and `ptry`/`raise`
(naming the dynamic `ptry` continuation)

$$\frac{\Gamma, k : \texttt{cont}\ A \vdash t : A}{\Gamma \vdash \texttt{callcc}\ (\texttt{fun}\ k\ \rightarrow\ t) : A} \qquad \frac{\Gamma, k : \texttt{cont}\ A \vdash t : A}{\Gamma, k : \texttt{cont}\ A \vdash \texttt{throw}\ k\ t : B}$$

$$\frac{\Gamma, \texttt{tp} : \texttt{cont}\ A \vdash t : A}{\Gamma \qquad\qquad\qquad \vdash \texttt{ptry}_{\texttt{tp}}\ t : A} \qquad \frac{\Gamma, \texttt{tp} : \texttt{cont}\ A \vdash t : A}{\Gamma, \texttt{tp} : \texttt{cont}\ A \vdash \texttt{raise}_{\texttt{tp}}\ t : B}$$

Typing `callcc`/`throw` and `ptry`/`raise`
(naming the dynamic `ptry` continuation)

$$\frac{\Gamma, k : \texttt{cont}\ A \vdash t : A}{\Gamma \vdash \texttt{callcc}\ (\texttt{fun}\ k\ \rightarrow\ t) : A} \qquad \frac{\Gamma, k : \texttt{cont}\ A \vdash t : A}{\Gamma, k : \texttt{cont}\ A \vdash \texttt{throw}\ k\ t : B}$$

$$\frac{\Gamma, \mathsf{tp} : \texttt{cont}\ A \vdash t : A}{\Gamma, \mathsf{tp} : \texttt{cont}\ T \vdash \texttt{ptry}_{\texttt{tp}}\ t : A} \qquad \frac{\Gamma, \mathsf{tp} : \texttt{cont}\ A \vdash t : A}{\Gamma, \mathsf{tp} : \texttt{cont}\ A \vdash \texttt{raise}_{\texttt{tp}}\ t : B}$$

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

$$\frac{\Gamma, \mathsf{tp} : \texttt{cont}\ T \vdash t : A \rightarrow_T B \qquad \Gamma, \mathsf{tp} : \texttt{cont}\ T \vdash u : A}{\Gamma, \mathsf{tp} : \texttt{cont}\ T \vdash t\, u : B} \qquad \frac{\Gamma, x : B, \mathsf{tp} : \texttt{cont}\ T \vdash t : C}{\Gamma, \mathsf{tp} : \texttt{cont}\ U \vdash \lambda x.t : B \rightarrow_T C}$$

$$\frac{}{\Gamma, x : A, \mathsf{tp} : \texttt{cont}\ T \vdash x : A}$$

Simple types: `callcc` is more expressive than `ptry` (computing with infinity)

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

```
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))))))
```

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

Any program whose result is a non-functional value and that uses `pseudo_decide_infinity` will yield a (correct) result.

Each call to `throw` will induce backtracking on the progress of the program that uses `pseudo_decide_infinity`.

Simple types: `callcc` is more expressive than `ptry` (drinkers' paradox)

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

```
(* 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))))
```

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 = (ptry (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` $\triangleq$ `unit` $\rightarrow_{\texttt{dom}}$ `unit` which is a *recursive* type.

Which possible Curry-Howard interpretation for `ptry`/`raise`?

How to interpret type effects in the logical framework?
Use a canonical type effect? Use $\bot$?
Type `ptry`/`raise` with the rules

$$\frac{\Gamma, \mathsf{tp} : \mathtt{cont}\ \bot \vdash t : \bot}{\Gamma, \mathsf{tp} : \mathtt{cont}\ \bot \vdash \mathtt{ptry}_{\mathsf{tp}}\ t : \bot} \qquad \frac{\Gamma, \mathsf{tp} : \mathtt{cont}\ \bot \vdash t : \bot}{\Gamma, \mathsf{tp} : \mathtt{cont}\ \bot \vdash \mathtt{raise}_{\mathsf{tp}}\ t : B}$$

The Curry-Howard interpretation for `callcc`

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

Minimal logic + Peirce law is called *minimal classical logic*

λ-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 `fun x -> throw` k_0 `(fun y -> x+y)`.
When y is applied, `throw` is called and it returns to k_0.

Conventionally, this semantics is expressed using an abortion operator $\mathcal{A}$ which itself hides a call to the toplevel continuation.

Intermezzo: Felleisen's $\mathcal{C}$ operator

Motivated by the possibility to reason on operators such as `callcc` in Scheme, Felleisen *et al* introduced the $\mathcal{C}$ operator.

Syntax

$$t \;::=\; x \mid \lambda x.t \mid t\,t \mid \mathcal{C}(\lambda k.t)$$

$\mathcal{C}$ is equivalent to the combination of `callcc` and $\mathcal{A}$.

$\mathcal{C}(\lambda k.t) = \mathtt{callcc}(\lambda k.\mathcal{A}\,t)$

$\mathtt{callcc}(\lambda k.t) = \mathcal{C}(\lambda k.k\,t)$
$\mathcal{A}\,t = \mathcal{C}(\lambda\_.t)$

Remark: in $\mathcal{C}(\lambda k.t)$, "$\lambda k.$" is part of the syntax. Alternative equivalent definitions of the language are

$$t \;::=\; x \mid \lambda x.t \mid t\,t \mid \mathcal{C}\,t$$

or

$$t \;::=\; x \mid \lambda x.t \mid t\,t \mid \mathcal{C}$$

Parigot's $\lambda\mu$-calculus
(minimal version)

Use special variables α, β for denoting continuations.

Syntax

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

Use *structural substitution* for continuations:

$$E[\mu\alpha.c] \to \mu\alpha.c[[\alpha]E/[\alpha]\Box]$$

More concisely:

$$E[\mu\alpha.c] \to \mu\alpha.c[[\alpha]E/\alpha]$$

Parigot's $\lambda\mu$-calculus
(expressing `callcc`)

`callcc` is approximable:

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

In fact, the actual operational semantics of `callcc` is:

$$E[\texttt{callcc}(\lambda k.t)] \to t[\lambda x.\mathcal{A}(E[x])/k] \qquad (*)$$

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

$$\texttt{callcc} \triangleq \lambda z.\mu\alpha.[\alpha](z\ \lambda x.\mu\_.[\alpha]x)$$

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

Yet, the full semantics of `callcc` above requires $\mathcal{A}$ 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 $(*)$ above.

Parigot's $\lambda\mu$-calculus
(adding a denotation for the toplevel continuation)

Syntax

$$
\begin{array}{lcll}
t & ::= & V \mid t\,t \mid \mu\alpha.c & \text{(terms)} \\
c & ::= & [\beta]t \mid [\mathsf{tp}]t & \text{(commands or states)} \\
V & ::= & x \mid \lambda x.t & \text{(values)}
\end{array}
$$

The operator $\mathcal{A}$ can now be expressed: $\mathcal{A} \triangleq \lambda x.\mu\_.[\mathsf{tp}]x$ and the toplevel reduction rule of `callcc` gets simulable:

$$
\begin{array}{rcl}
[\mathsf{tp}]\ E[\mathtt{callcc}(\lambda k.t)] & \equiv & [\mathsf{tp}]E[(\lambda z.\mu\alpha.[\alpha](z\ \lambda x.\mu\_.[\alpha]x))\ \lambda k.t] \\
 & & \qquad\qquad\qquad\qquad \downarrow * \\
[\mathsf{tp}]\ E[t[\lambda x.\mathcal{A}(E[x])/k]] & \equiv & [\mathsf{tp}]\ E[t[\lambda x.\mu\_.[\mathsf{tp}]E[x]/k]]
\end{array}
$$

$\lambda\mu$-calculus extended with tp can faithfully simulate `callcc` or Felleisen's $\lambda_{\mathcal{C}}$-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 $\overline{\lambda}\mu\tilde{\mu}$!) [Ariola-Herbelin 2007]

An example of "inefficiency" in call-by-value:

$$
\begin{array}{llclclcl}
\mathtt{let}\ \mathit{loop}\,() = \mathtt{callcc}(\lambda k.k(\mathit{loop}\,())) : & \mathit{loop}\,() & \rightarrow & (\lambda x.\mathcal{A}x)\,(\mathit{loop}\,()) & \rightarrow & (\lambda x.\mathcal{A}x)\,((\lambda x.\mathcal{A}x)\,(\mathit{loop}\,())) & \rightarrow & \ldots \\
\mathtt{let}\ \mathit{loop}\,() = \mu k.[k](\mathit{loop}\,()) : & [\mathsf{tp}](\mathit{loop}\,()) & \rightarrow & [\mathsf{tp}]\mathit{loop}\,() & \rightarrow & [\mathsf{tp}](\mathit{loop}\,()) & \rightarrow & \ldots
\end{array}
$$

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

$$
\begin{array}{llll}
t & ::= & V \mid t\,t \mid \mathcal{S}(\lambda k.t) \mid \langle t \rangle & \text{(terms)} \\
V & ::= & x \mid \lambda x.t & \text{(values)} \\
\\
F\Box & ::= & \Box \mid F[V\ \Box] \mid F[\Box\ t] & \text{(local ev. contexts)} \\
E\Box & ::= & \Box \mid E[\mathtt{reset} F\Box] & \text{(global ev. contexts)}
\end{array}
$$

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

Operational Semantics

$$
\begin{array}{lcl}
E[(\lambda x.t)\ u] & \rightarrow & E[t[u/x]] \\
E[\langle F[\mathcal{S}(\lambda k.t)] \rangle] & \rightarrow & E[\langle t[\lambda x.\langle F[x] \rangle / k] \rangle] \\
E[\langle V \rangle] & \rightarrow & E[V]
\end{array}
$$

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
```

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

$$\begin{array}{lcll} T(A) & = & S \to A \times S & \\ \eta & = & \lambda x.\lambda s.(x,s) & : A \to T(A) \\ {}^* & = & \lambda f.\lambda x.\lambda s.\mathtt{let}\ (x,s) = x\ s\ \mathtt{in}\ (f\ x\ s) & : (A \to T(B)) \to T(A) \to T(B) \end{array}$$

The resulting encoding of `read` and `write`

$$\begin{array}{lcllc} \mathtt{read} & \triangleq & \lambda().\mathcal{S}(\lambda k.\lambda s.(k\ s\ s)) & : & \mathtt{unit} \to S \\ \mathtt{write} & \triangleq & \lambda s.\mathcal{S}(\lambda k.\lambda\_.(k\ ()\ s)) & : & S \to \mathtt{unit} \end{array}$$

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 $\mathcal{A}$ behaves as `raise`. The correspondences are as follows:

$$
\begin{array}{lcl}
\mathcal{A}\, t & \triangleq & \mathcal{S}(\lambda\_.t) \\
\texttt{callcc}\ (\lambda k.t) & \triangleq & \mathcal{S}(\lambda k.k\ t[\lambda x.\mathcal{A}\,(k\ x)/k]) \\
\mathcal{C}\ (\lambda k.t) & \triangleq & \mathcal{S}(\lambda k.t[\lambda x.\mathcal{A}\,(k\ x)/k]) \\
\texttt{ptry}\ t & \triangleq & \langle t \rangle \\
\texttt{raise}\ t & \triangleq & \mathcal{A}\ t
\end{array}
$$

Conversely, `shift` can be macro-defined from `callcc`, $\mathcal{A}$/`raise` and `ptry`/`reset`:

$$\mathcal{S}(\lambda k.t) \ \triangleq \ \texttt{callcc}\ (\lambda k.\mathcal{A}(t[\lambda x.\langle k\ x\rangle/k]))$$

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 continuation and not just the delimited one. It is better to use the name $\mathcal{K}$ 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

$$\begin{array}{lcl} t & ::= & V \mid t\,t \mid \mu\alpha.c \mid \widehat{\mu}\mathsf{tp}.c \\ c & ::= & [\beta]t \mid [\mathsf{tp}]t \\ V & ::= & x \mid \lambda x.t \end{array}$$

Macro-definability

$$\begin{array}{lcl} \langle t\rangle & \triangleq & \widehat{\mu}\mathsf{tp}.[\mathsf{tp}]t \\ \mathcal{A}\,t & \triangleq & \mu\_.[\mathsf{tp}]t \\ \mathcal{S}(\lambda k.t) & \triangleq & \mu\alpha.[\mathsf{tp}](t[\lambda x.\widehat{\mu}\mathsf{tp}.[\alpha]x/k]) \\ \mathcal{C}(\lambda k.t) & \triangleq & \mu\alpha.[\mathsf{tp}](t[\lambda x.\mu\_.[\alpha]x/k]) \\ \mathtt{callcc}(\lambda k.t) & \triangleq & \mu\alpha.[\alpha](t[\lambda x.\mu\_.[\alpha]x/k]) \end{array}$$

The fourth combination $\mu\alpha.[\alpha](t[\lambda x.\widehat{\mu}\mathsf{tp}.[\alpha]x/k])$ has no name (to our knowledge), it is equivalent to $\mathcal{S}(\lambda k.k\,t)$.

$\lambda\mu\widehat{\mu}$tp-calculus
(a fine-grained `shift`/`reset` calculus)

Semantics

(β_v)	$(\lambda x.t)\, V$	$\rightarrow$	$t[V/x]$	
(η_v)	$\lambda x.(V\ x)$	$\rightarrow$	t	if x not free in V
(μ_{app})	$(\mu\alpha.c)\, u$	$\rightarrow$	$\mu\alpha.c[[\alpha](\Box\ u))/\alpha]$	
(μ'_{app})	$V\ (\mu\alpha.c)$	$\rightarrow$	$\mu\alpha.c[[\alpha](V\ \Box))/\alpha]$	
(μ_{var})	$[\beta]\mu\alpha.c$	$\rightarrow$	$c[\beta/\alpha]$	also if β is tp
(η_μ)	$\mu\alpha.[\alpha]t$	$\rightarrow$	t	if α not free in t
$(\widehat{\mu}_{var})$	$[\mathsf{tp}]\widehat{\mu}\mathsf{tp}.c$	$\rightarrow$	c	
$(\eta_{\widehat{\mu}v})$	$\widehat{\mu}\mathsf{tp}.[\mathsf{tp}]V$	$\rightarrow$	V	even if tp occurs in t
$(let_{\widehat{\mu}})$	$\widehat{\mu}\mathsf{tp}.[\beta]((\lambda x.t)\ \widehat{\mu}\mathsf{tp}.c)$	$\rightarrow$	$(\lambda x.\widehat{\mu}\mathsf{tp}.[\beta]t)\ \widehat{\mu}\mathsf{tp}.c$	
(let_μ)	$(\lambda x.\mu\alpha.[\beta]t)\, u$	$\rightarrow$	$\mu\alpha.[\beta]((\lambda x.t)\, u)$	
(let_{app})	$(\lambda x.t)\, u\, u'$	$\rightarrow$	$(\lambda x.(t\, u'))\, u$	
(let'_{app})	$V\ ((\lambda x.t)\, u)$	$\rightarrow$	$(\lambda x.(V\ t))\, u$	
(η_{let})	$(\lambda x.x)\, t$	$\rightarrow$	t	

$\lambda\mu\widehat{\mu}$tp-calculus and types

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:

$$
\begin{array}{lcl}
\langle t \rangle & : & T \to T \\
\mathcal{A}\, t & : & T \to A \\
\mathcal{S}(\lambda k.t) & : & ((A \to T) \to T) \to A \\
\mathcal{C}(\lambda k.t) & : & ((A \to B) \to T) \to A \\
\mathtt{callcc}(\lambda k.t) & : & ((A \to B) \to A) \to A
\end{array}
$$

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

$$
\begin{array}{lcll}
\langle t \rangle & : & \perp \to \perp & \text{(no logical content)} \\
\mathcal{A}\, t & : & \perp \to A & \text{(ex falso quodlibet)} \\
\mathcal{S}(\lambda k.t) & : & ((A \to \perp) \to \perp) \to A & \text{(double negation elimination)} \\
\mathcal{C}(\lambda k.t) & : & ((A \to B) \to \perp) \to A & \text{(an instance of it is double negation elimination)} \\
\mathtt{callcc}(\lambda k.t) & : & ((A \to B) \to A) \to A & \text{(Peirce law)}
\end{array}
$$

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:

$$\begin{array}{lcl} t & ::= & x \mid \lambda x.t \mid t\,t \mid \mu\alpha.c \\ c & ::= & [\alpha]t \end{array}$$

Extending the call-by-name reduction semantics with η rules:

$$\begin{array}{llcll} (\beta) & (\lambda x.t)\,u & \rightarrow & t[u/x] & \\ (\eta) & \lambda x.(t\,x) & \rightarrow & t & \text{if } x \text{ not free in } t \\ & & & & \\ (\mu_{app}) & (\mu\alpha.c)\,u & \rightarrow & \mu\alpha.c[[\alpha](\Box\; u))/\alpha] & \\ (\mu_{var}) & [\beta]\mu\alpha.c & \rightarrow & c[\beta/\alpha] & \\ (\eta_\mu) & \mu\alpha.[\alpha]t & \rightarrow & t & \text{if } \alpha \text{ not free in } t \end{array}$$

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 β, η, μ_{app}, μ_{var} and η_μ but whose observational behaviour is not separable.

Success of separability in Saurin's $\lambda\mu$-calculus

A slightly different syntax (originally from de Groote):

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

The same (apparent) reduction rules:

$$\begin{array}{lllll}
(\beta) & (\lambda x.t)\,u & \to & t[u/x] & \\
(\eta) & \lambda x.(t\,x) & \to & t & \text{if } x \text{ not free in } t \\
 & & & & \\
(\mu_{app}) & (\mu\alpha.t)\,u & \to & \mu\alpha.t[[\alpha](\Box\ u))/\alpha] & \\
(\mu_{var}) & [\beta]\mu\alpha.t & \to & t[\beta/\alpha] & \\
(\eta_\mu) & \mu\alpha.[\alpha]t & \to & t & \text{if } \alpha \text{ not free in } t
\end{array}$$

But a major difference: $t\ [\beta]\mu\alpha.u \to t\,(u[\beta/\alpha])$. We can get rid of μ 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 β, η, μ_{app}, μ_{var} and η_μ has the Böhm separability property.

From Saurin's $\lambda\mu$-calculus to call-by-name $\lambda\mu\widehat{\mu}$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 $\widehat{\mu}$tp and [tp] can be proved to be adequate coercions from Saurin's calculus to a calculus well-suited for computation.

Macro-definition of Saurin's calculus on top of $\lambda\mu\widehat{\mu}$tp

$$\begin{array}{lcl} \mu\alpha.t & \triangleq & \mu\alpha.[\mathsf{tp}]t \\ [\alpha]t & \triangleq & \widehat{\mu}\mathsf{tp}.[\alpha]t \end{array}$$

Call-by-name $\lambda\mu\widehat{\mu}$tp-calculus

(β)	$(\lambda x.t)\,u$	$\rightarrow$	$t[u/x]$	
(η)	$\lambda x.(t\,x)$	$\rightarrow$	t	if x not free in t
(μ_{app})	$(\mu\alpha.c)\,u$	$\rightarrow$	$\mu\alpha.c[[\alpha](\Box\ u))/\alpha]$	
(μ^n_{var})	$[\beta]\mu\alpha.c$	$\rightarrow$	$c[\beta/\alpha]$	$\beta \neq \mathsf{tp}$
(η_μ)	$\mu\alpha.[\alpha]t$	$\rightarrow$	t	if α not free in t
$(\widehat{\mu}_{var})$	$[\mathsf{tp}]\widehat{\mu}\mathsf{tp}.c$	$\rightarrow$	c	
$(\eta_{\widehat{\mu}})$	$\widehat{\mu}\mathsf{tp}.[\mathsf{tp}]t$	$\rightarrow$	t	even if tp occurs in t

Obviously, we have:

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

Corollary $\lambda\mu\widehat{\mu}$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 $\lambda\mu\widehat{\mu}$tp-calculus.

Classification of the reduction semantics of $\lambda\mu\widehat{\mu}$tp-calculus

the fundamental critical pair of computation

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

$\swarrow$ (CBV)   (CBN) $\searrow$

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

subsidiary choice

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

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

`shift/lazy reset`
(Sabry)

cps-completion (Sabry)

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

`shift/reset`
(Danvy-Filinski)

cps-completion (Kameyama-Hasegawa)

typed "domain"-completion (Sitaram-Felleisen)

subsidiary choice

$[\mathrm{tp}]\mu\alpha.c$

(tp co-value) $\swarrow$

CBN `shift/reset`
(Danvy)

$\searrow$ (tp not co-value)

$\Lambda\mu$
(de Groote/Saurin)

Böhm-completion (Saurin)

Abstract machine for call-by-name $\lambda\mu\widehat{\mu}$tp-calculus

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

$$
\begin{array}{llll}
K & ::= & \alpha[e] \mid t[e] \cdot K & \text{(linear ev. contexts)} \\
[S] & ::= & [\,] \mid [\mathsf{tp} = K; S] & \text{(dynamic environment)} \\
[e] & ::= & [\,] \mid [x = t[e]; e] \mid [\alpha = K; e] & \text{(environments)} \\
s & ::= & c\ [e]\ [S] \mid t\ [e]\ K\ [S] & \text{(states)}
\end{array}
$$

Abstract machine for call-by-name $\lambda\mu\widehat{\mu}$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 α in e and similarly for $e(x)$.

Control given to the evaluation context

$$
\begin{array}{llllllll}
[\mathsf{tp}]t & [e] & [\mathsf{tp}=K;S] & \rightarrow & t & [e] & K & [S] \\
[\alpha]t & [e] & [S] & \rightarrow & t & [e] & \alpha[e] & [S] \\
[\mathsf{tp}]t & [e] & [\,] & \rightarrow & \multicolumn{4}{l}{\text{stop on } [\mathsf{tp}]t[e]}
\end{array}
$$

Control given to the term

$$
\begin{array}{lllllllll}
x & [e] & K & [S] & \rightarrow & t & [e'] & K & [S] & \text{if } e(x)=t[e'] \\
x & [e] & K & [S] & \rightarrow & \multicolumn{4}{l}{\text{stop on } S^*[K^*[x]]} & \text{if } x \text{ not bound in } e \\
\lambda x.t & [e] & K & [S] & \rightarrow & \lambda x.t[e] & & K & [S] \\
t\,u & [e] & K & [S] & \rightarrow & t & [e] & u[e]\cdot K & [S] \\
\mu\alpha.c & [e] & K & [S] & \rightarrow & c & & [\alpha=K;e] & [S] \\
\widehat{\mu}\mathsf{tp}.c & [e] & K & [S] & \rightarrow & c & & [e] & [\mathsf{tp}=K;S]
\end{array}
$$

Control given to the "linear" evaluation context

$$
\begin{array}{llllllllll}
\lambda x.t[e] & u\cdot K & [S] & \rightarrow & t & [x=u;e] & K & [S] \\
\lambda x.t[e] & \alpha[e'] & [S] & \rightarrow & \lambda x.t & [e] & K & [S] & \text{if } e'(\alpha)=K \\
\lambda x.t[e] & \alpha[e'] & [S] & \rightarrow & \multicolumn{4}{l}{\text{stop on } S^*[[\alpha](\lambda x.t[e])]} & \text{if } \alpha \text{ not bound in } e'
\end{array}
$$

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

$$t\ [\,]\ \epsilon[\,]\ [\,]$$

Note: terminal states are defined by

$$
\begin{array}{lll}
[\alpha]^* & = & [\alpha](\Box) \\
(t[e]\cdot K)^* & = & K^*[\Box\ t[e]]
\end{array}
\qquad
\begin{array}{lll}
[\,]^* & = & \Box \\
[\mathsf{tp}=K;S]^* & = & [S]^*[\widehat{\mu}\mathsf{tp}.K^*]
\end{array}
$$

Abstract machine for call-by-value $\lambda\mu\widehat{\mu}$tp-calculus

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

$$
\begin{array}{llll}
K & ::= & k[e] \mid t[e] \cdot K \mid \tilde{\mu}x.(W\ x \cdot K) & \text{(ev. contexts)} \\
[S] & ::= & [\,] \mid [\mathsf{tp} = K; S] & \text{(dynamic environment)} \\
W & ::= & V[e] & \text{(closure)} \\
[e] & ::= & [\,] \mid [x = W; e] \mid [\alpha = K; e] & \text{(environments)} \\
k & ::= & \alpha \mid \mathsf{tp} & \text{(ev. context variables)} \\
s & ::= & W\ K\ [S] \mid t\ [e]\ K\ [S] & \text{(states)}
\end{array}
$$

Abstract machine for call-by-value $\lambda\mu\widehat{\mu}$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 $W\ K\ S$) and the ones giving priority to the term (commands of the form $t[e]\ K\ S$). We write $e(\alpha)$ for the binding of α in e and similarly for $e(x)$.

Control given to the evaluation context

$$
\begin{array}{llllllll}
W & \mathsf{tp}[e] & [\mathsf{tp} = K; S] & \to & W & K & [S] & \\
W & \mathsf{tp}[e] & [\,] & \to & \text{stop on } [\mathsf{tp}]W & & & \\
W & \alpha[e] & [S] & \to & W & K & [S] & \text{if } e(\alpha) = K \\
W & \alpha[e] & [S] & \to & \text{stop on } S^*[[\alpha]W] & & \text{if } \alpha \text{ not bound in } e & \\
W & t[e] \cdot K & [S] & \to & t\ \ [e] & \tilde{\mu}x.(W\ x \cdot K) & [S] & \\
W & \tilde{\mu}x.(V[e]\ x \cdot K) & [S] & \to & V\ \ [e]\ \ W\ \cdot & K & [S] &
\end{array}
$$

Control given to the term

$$
\begin{array}{llllllll}
V & [e] & K & [S] & \to & V[e] & K & [S] \\
t\,u & [e] & K & [S] & \to & t\ \ [e] & u[e] \cdot K & [S] \\
\mu\alpha.[k]t & [e] & K & [S] & \to & t\ \ [\alpha = K; e] & k[\alpha = K; e] & [S] \\
\widehat{\mu}\mathsf{tp}.[k]t & [e] & K & [S] & \to & t\ \ [e] & k[e] & [\mathsf{tp} = K; S]
\end{array}
$$

Control given to the functional value

$$
\begin{array}{lllllllll}
\lambda x.t & [e] & W \cdot K & [S] & \to & t & [x = W; e] & K\ \ [S] & \\
x & [e] & W \cdot K & [S] & \to & V & [e'] & W \cdot K\ \ [S] & \text{if } e(x) = V[e'] \\
x & [e] & W \cdot K & [S] & \to & \text{stop on } S^*[K^*[x\,W]] & & \text{otherwise} &
\end{array}
$$

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

$$t\ [\,]\ \mathsf{tp}[\,]\ [\,]$$

References

[1] Zena M. Ariola and Hugo Herbelin. Control Reduction Theories: The Benefit of Structural Substitution. *Journal of Functional Programming*, 2007. To appear.

[2] Zena M. Ariola, Hugo Herbelin and Amr Sabry. A type-theoretic foundation of delimited continuations. In *Higher-Order and Symbolic Computation*, 2007. To appear.

[3] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU, University of Copenhagen, Copenhagen, Denmark, August 1989.

[4] Olivier Danvy and Andrzej Filinski. Abstracting Control. Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice, pages 151–160, 1990.

[5] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker and Bruce F. Duba. Reasoning with continuations. *First symposium on logic and computer science*, pages 131–141, 1986.

[6] Andrzej Filinski. Representing monads. In *Conf. Record 21st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '94), Portland, OR, USA*, pages 446–457. January 1994.

[7] Michel Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In *Logic Programming and Automated Reasoning: International Conf. LPAR '92 Proceedings*, pages 190–201, 1992.

[8] Amr Sabry. Note on axiomatizing the semantics of control operators. Technical Report CIS-TR-96-03, Dept of Information Science, Univ. of Oregon, 1996.

[9] Alexis Saurin. Separation with streams in the $\lambda\mu$-calculus. In *Proceedings, 20th Annual IEEE Symposium on Logic in Computer Science (LICS '05)*, pages 356–365. IEEE Computer Society Press, 2005.