Classical logic and control operators

Hugo Herbelin

13 February 2012

Logic and Interaction 2012

Proofs and Programs

The proof-as-programs correspondence

Hilbert-style minimal propositional logic = Simply-typed combinatory logic [Curry, 1958]

Intuitionistic natural deduction = typed λ-calculus [Howard, 1969]

Griffin's revelation: control operators provide with a computational content to classical logic [1990]

Outline

- Preliminary: how to define classical logic?
- Preliminary: intuitionistic second-order logic
- Control operators in programming languages, SML programs at work
- A core calculus for representing control: $\lambda\mu$-calculus
- Adding toplevel continuation constants: $\lambda\mu$tp-calculus
- Other calculi of control

A possible definition of classical logic?

A logic is *classical* iff **Peirce's law**:

$$PL \triangleq ((P \to B) \to P) \to P$$

holds for any P positive (i.e. P is $A_1 \vee A_2$ or $\exists x\, A$ or an atom $P(\vec{t})$).

In particular, Peirce's law for a negative connective (i.e. $\forall$, $\to$, $\wedge$) reduces to Peirce's law on subformulae (reversibility of negative formulas).

In the presence of a connective $\bot$ such that **ex falso quodlibet** ($= \bot$-elimination) holds:

$$EFQ \triangleq \bot \to A$$

then the common principles of **double negation elimination** and **excluded-middle** hold:

$$DN \triangleq \neg\neg A \to A$$
$$EM \triangleq A \vee \neg A$$

As a matter of fact:

$$DN = PL + EFQ$$

Hence, PL (for positive formulas) is the essence of (axiomatic) classical logic. We will later see how $\lambda\mu$ itself "implements" the essence of PL.

Remark

By analogy with the definition

$$\textit{minimal logic} = \textit{intuitionistic logic w/o EFQ}$$

we could define

$$\textit{classical minimal logic} = \textit{classical logic w/o EFQ}$$

Examples: some intuitionistic logics that are essentially classical

- Heyting Arithmetic with only negative connectives (i.e. $\forall$, $\rightarrow$, $\wedge$) *is* essentially classical in this sense.

- Intuitionistic second-order logic with $\rightarrow$ and $\forall$ is classical as soon as all atoms are negated.

- Heyting Arithmetic such that all positive connectives and atoms are prefixed by double negations is essentially classical.

- Coherent logic [Bezem, Coquand] is an intuitionistic logic made of formulae of the form $\forall \vec{x}\,(\bigwedge \vec{A} \rightarrow \bigvee \exists \vec{y}\, \bigwedge \vec{B})$. It is essentially classical (neither PL nor EM can be stated!).

Intuitionistic second-order logic as a (Church-style) programming language

$$\frac{(a : A) \in \Gamma}{\Gamma \vdash a : A}\text{AXIOM}$$

$$\frac{\Gamma, a : A \vdash p : B}{\Gamma \vdash \lambda a.p : A \to B}{\to}_I \qquad \frac{\Gamma \vdash p : A \to B \qquad \Gamma \vdash q : A}{\Gamma \vdash p\, q : B}{\to}_E$$

$$\frac{\Gamma \vdash p : A(x) \qquad x \text{ fresh in } \Gamma}{\Gamma \vdash \lambda x.p : \forall x\, A(x)}\forall^1_I \qquad \frac{\Gamma \vdash p : \forall x\, A(x)}{\Gamma \vdash p\, t : A(t)}\forall^1_E$$

$$\frac{\Gamma \vdash p : A(X) \qquad X \text{ fresh in } \Gamma}{\Gamma \vdash \lambda X.p : \forall X\, A(X)}\forall^2_I \qquad \frac{\Gamma \vdash p : \forall X\, A(X)}{\Gamma \vdash p\, P : A(P)}\forall^2_E$$

Thanks to Church-style, can be equipped with either call-by-name or call-by-value semantics.

Intuitionistic second-order logic as a (Curry-style) programming language

$$\frac{(a : A) \in \Gamma}{\Gamma \vdash a : A}\text{AXIOM}$$

$$\frac{\Gamma, a : A \vdash p : B}{\Gamma \vdash \lambda a.p : A \to B}{\to}_I \qquad \frac{\Gamma \vdash p : A \to B \qquad \Gamma \vdash q : A}{\Gamma \vdash p\, q : B}{\to}_E$$

$$\frac{\Gamma \vdash p : A(x) \qquad x \text{ fresh in } \Gamma}{\Gamma \vdash p : \forall x\, A(x)}\forall^1_I \qquad \frac{\Gamma \vdash p : \forall x\, A(x)}{\Gamma \vdash p : A(t)}\forall^1_E$$

$$\frac{\Gamma \vdash p : A(X) \qquad X \text{ fresh in } \Gamma}{\Gamma \vdash p : \forall X\, A(X)}\forall^2_I \qquad \frac{\Gamma \vdash p : \forall X\, A(X)}{\Gamma \vdash p : A(P)}\forall^2_E$$

When treating quantifiers in Curry-style, i.e. as intersections, it becomes inconsistent with call-by-value reduction [Pierce, 1991].

Connectives can be added by second-order encoding

E.g. disjunction:

$$\begin{array}{ll} A_1 \vee A_2 & \triangleq\ \forall Y\,(A_1 \to Y) \to (A_2 \to Y) \to Y \\ \texttt{case}\ p\ \texttt{of}\ [a_1.p_1 \mid a_2.p_2]\ \texttt{end} & \triangleq\ p\,(\lambda a_1.p_1)\,(\lambda a_2.p_2) \\ \iota_i\,p & \triangleq\ \lambda f_1.\,\lambda f_2.(f_i\,p) \end{array}$$

Strong evaluation can be forced using "storage operators". E.g., if A_1 is itself $A_{11} \vee A_{12}$, then

$$\iota_1\,p \triangleq \lambda f_1\,\lambda f_2.\texttt{case}\ p\ \texttt{of}\ [a_1.f_i(\iota_1 p_1) \mid a_2.f_1(\iota_2 p_2)]\ \texttt{end}$$

defines injections whose argument is forced to be evaluated.

E.g. falsity:

$$\begin{array}{ll} \bot & \triangleq\ \forall Y\,Y \\ \texttt{exfalso}\ p & \triangleq\ p \end{array}$$

Other abbreviations:

$$\begin{array}{ll} A \wedge B & \triangleq\ \forall Y\,(A \to B \to Y) \to Y \\ \exists x\,A(x) & \triangleq\ \forall Y\,(\forall x\,A(x) \to Y) \to Y \\ \exists X\,A(X) & \triangleq\ \forall Y\,(\forall X\,A(X) \to Y) \to Y \\ \neg A & \triangleq\ A \to \bot \end{array}$$

Control operators in programming languages

There are many variants of *statically-binding* control operators

- Landin's J [1964]: an abstraction of `goto`
- Reynolds' escape [1972]
- Scheme's `catch/throw` [1975]
- Scheme's `call-with-current-continuation` (= `call/cc`) [1985]
- Felleisen *et al*'s $\mathcal{C}$, $\mathcal{K}$ and $\mathcal{A}$ [1986]: abstracting control for reasoning about it
- SML's `callcc/throw`: introducing a typed variant of Scheme's `call/cc`

We will later see how $\lambda\mu$ provides with a uniform framework for talking about all these operators.

Note: *dynamically-binding* control operators (i.e. exceptions) to be considered later on

Control operators in SML

SML of New Jersey offers the following operators:

```
module SMLofNJ.Cont
 type 'a cont;;                          (* an abstract type of continuation *)
 val callcc : ('a cont -> 'a) -> 'a;; (* operator to capture current continuation *)
 val throw : 'a cont -> 'a -> 'b;;    (* operator to reinstall given continuation *)
end
```

Control operators in SML: Operational semantics

In Felleisen's style (i.e. *contextual semantics*), the description of the operational semantics needs the following:

- a notion of *evaluation context*, i.e. a term with one hole denoting where the evaluation happens

- the restriction of evaluation to full programs, i.e. *toplevel evaluation*

In call-by-value λ-calculus, *evaluation contexts* E can typically defined by:

$$\begin{array}{lrcl}
\text{Ev. cont.} & E & ::= & [\,] \mid E[F] \\
\text{Elem. ev. cont.} & F & ::= & [\,]\,p \mid [\,]\,t \mid V\,[\,] \mid \iota_i([\,]) \\
& & \mid & \texttt{case}\ [\,]\ \texttt{of}\ [a_1.p_1 \mid a_2.p_2]\ \texttt{end} \mid \texttt{exfalso}\ [\,] \\
& & \mid & \texttt{throw}\ [\,]\,p \mid \texttt{throw}\ V\ [\,] \mid \ldots
\end{array}$$

Operational semantics

$$\begin{array}{lcl}
E[(\lambda x.p)\,V] & \mapsto & E[p[x \leftarrow V]] \\
E[\texttt{callcc}\,(\lambda k.p)] & \mapsto & E[p[k \leftarrow \lambda x.E[x]]] \\
E'[\texttt{throw}\ V\ V'] & \mapsto & V\,V'
\end{array}$$

Example: A program of type $A \vee \neg A$

$$\texttt{em} \triangleq \texttt{callcc}\, \lambda k.\iota_2(\lambda a.\texttt{throw}\ k\ (\iota_1\, a)) \ : \ A \vee \neg A$$

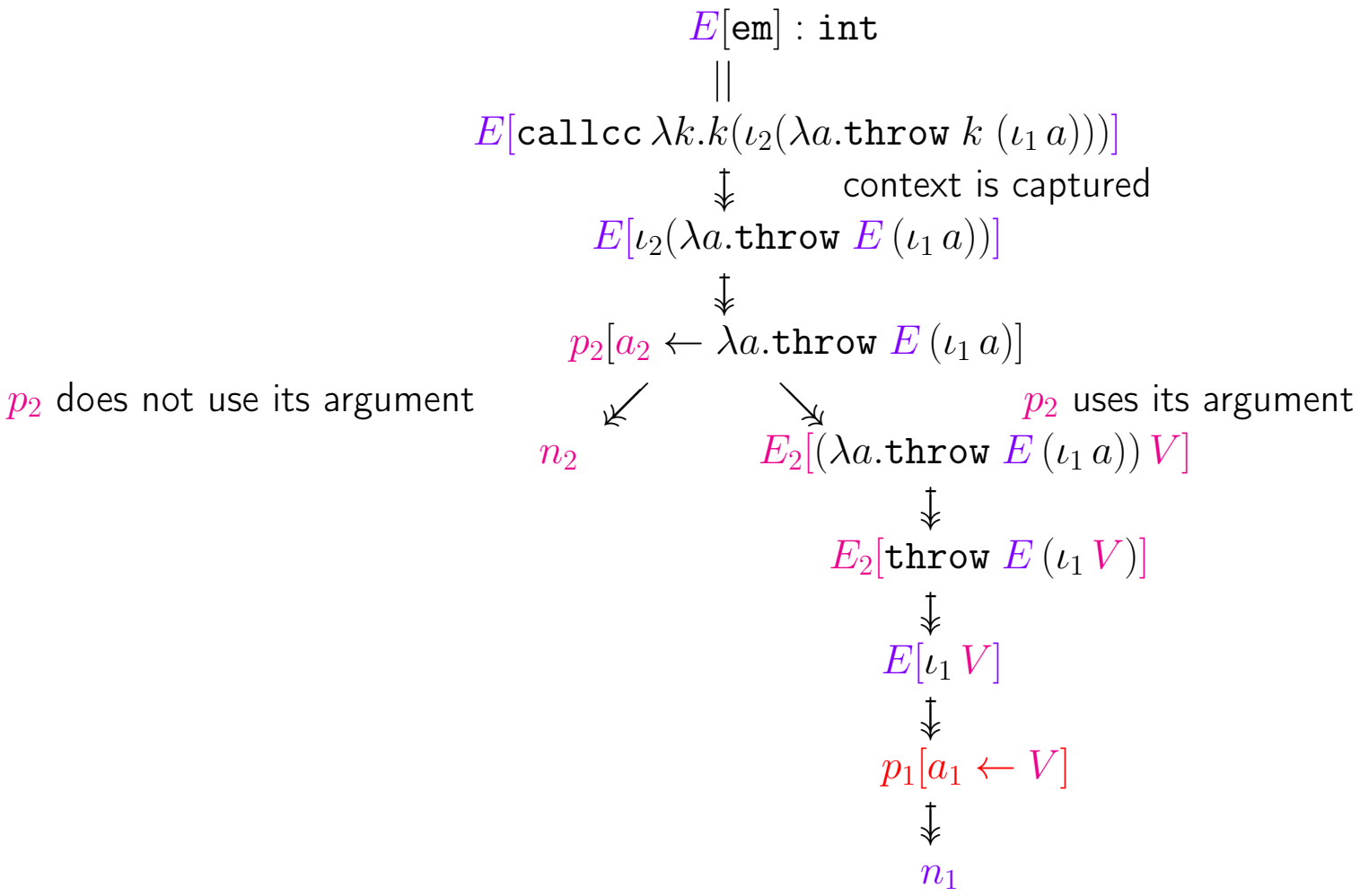
where

$k : A\,\texttt{cont}$

$a : A$

A running example

Assume we have a program of type `int` that uses `em`; let $E[\ ]$ be `case` $[\ ]$ `of` $[a_1.p_1 \mid a_2.p_2]$ `end` such that at some time of the execution we have:



Other examples

A program of type $((A \to B) \to A) \to A$ (**Peirce's law**):

$$\texttt{pl} \triangleq \lambda x.\texttt{callcc}\,\lambda k.(x\,\lambda a.\texttt{throw}\;k\;a) \;:\; ((A \to B) \to A) \to A$$

where
$x : (A \to B) \to A$
$k : A\;\texttt{cont}$
$a : A$

A program of type $((A \to \bot) \to \bot) \to A$ (**Double-negation elimination**):

$$\texttt{dn} \triangleq \lambda x.\texttt{callcc}\,\lambda k.(\texttt{exfalso}\;\;(x\,\lambda a.\texttt{throw}\;k\;a)) \;:\; ((A \to \bot) \to \bot) \to A$$

where
$x : (A \to \bot) \to \bot$
$k : A\;\texttt{cont}$
$a : A$

Drinker's paradox

A program of type $\exists x\,(\mathtt{drink}(x) \to \forall y\,\mathtt{drink}(y))$ **(Drinker's paradox)**:

$$\begin{array}{rcl} \mathtt{dl} & \triangleq & \mathtt{callcc}\,\lambda k.(x_0, \lambda H_x.\lambda y.\mathtt{callcc}\,\lambda k'.\mathtt{throw}\,k\,(y, \lambda H_y.\lambda y'.\mathtt{throw}\;k'\;H_y)) \\ & : & \exists x\,(\mathtt{drink}(x) \to \forall y\,\mathtt{drink}(y)) \end{array}$$

where

x_0 is an arbitrary variable

$k : \exists x\,(\mathtt{drink}(x) \to \forall y\,\mathtt{drink}(y))\ \mathtt{cont}$

$k' : \mathtt{drink}(y)\ \mathtt{cont}$

$\lambda\mu$-calculus: A fine-grained calculus for control operators and classical logic (Parigot, 1992)

New kinds of variables α, β for referring to evaluation contexts.

the "pure" binding part of `callcc`

$\downarrow$

Terms	p, q	$::=$	$x \mid \lambda x.p \mid p\,q \mid \mu\alpha.c$	
Commands	c	$::=$	$[\alpha]p$	(a.k.a. processes, states, statements ...)

$\uparrow$

the "pure" referring part of `throw`

Tentative re-interpretation of `callcc`/`throw`:

context is captured and immediately reinstalled, henceforth keeping a copy

$\downarrow$

$$\texttt{callcc}\ \lambda\alpha.p \triangleq \mu\alpha.[\alpha]\,p$$
$$\texttt{throw}\,\alpha\,p \triangleq \mu\_.[\alpha]\,p$$

$\uparrow$

current context is bound to a fresh variable hence thrown away

context bound to α is restored

$\lambda\mu$-calculus: A fine-grained calculus for control operators and classical logic (Parigot, 1992)

New kinds of variables α, β for referring to evaluation contexts.

the "pure" binding part of `callcc`

$\downarrow$

Terms $\quad p, q ::= x \mid \lambda x.p \mid p\,q \mid \mu\alpha.c$

Commands $\quad c ::= [\alpha]p$ (a.k.a. processes, states, statements ...

$\uparrow$

the "pure" referring part of `throw`

More precise re-interpretation of `callcc`/`throw`:

context is captured and immediately reinstalled, henceforth keeping a copy

$\downarrow$

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

$\uparrow$

current context is bound to a fresh variable hence thrown away

context bound to α is restored as a functional reification of the ev. context

$\lambda\mu$-calculus: A fine-grained calculus for control operators and classical logic (Parigot, 1992)

New kinds of variables α, β for referring to evaluation contexts.

the "pure" binding part of `callcc`

$\downarrow$

Terms $\quad p, q ::= x \mid \lambda x.p \mid p\,q \mid \mu\alpha.c$

Commands $\quad c ::= [\alpha]p$ (a.k.a. processes, states, statements ...

$\uparrow$

the "pure" referring part of `throw`

Ultimate re-interpretation of `callcc`/`throw`:

context is captured, immediately reinstalled, and its functional reification bound to k

$\downarrow$

$$\texttt{callcc}\ \lambda k.p \triangleq \mu\alpha.[\alpha]\,(p[k \leftarrow \lambda x.\mu\_.[\alpha]x])$$
$$\texttt{throw}\,k\,p \triangleq k\,p$$

$\uparrow$

k has in fact to be interpreted as an ordinary variable

$\lambda\mu$-calculus: Simple typing

Two kinds of declarations:

$$\Gamma ::= \emptyset \mid \Gamma, x : A \mid \Gamma, \alpha : A^{\perp\!\!\!\perp}$$

ordinary hypotheses ↑ refuted conclusions (the "A cont" of SML)

Two kinds of sequents:

$\Gamma \vdash p : A$ for terms

$\Gamma \vdash c : \perp\!\!\!\perp$ for commands

↑

symbol meaning "no type"

Extra inference rules:

$$\frac{\Gamma, \alpha : A^{\perp\!\!\!\perp} \vdash c : \perp\!\!\!\perp}{\Gamma \vdash \mu\alpha.c : A}\mathit{Activate} \qquad \frac{\Gamma \vdash p : A \qquad (\alpha : A^{\perp\!\!\!\perp}) \in \Gamma}{\Gamma \vdash [\alpha]p : \perp\!\!\!\perp}\mathit{Passivate}$$

Alternatively, one could have written refuted conclusions on the right-hand side of $\vdash$: $\mathit{Activate}$ rule is a right contraction rule! It expresses in a purely structural way the contraction operating in Peirce's law.

$\lambda\mu$-calculus: Call-by-name semantics

Definition of call-by-name evaluation contexts

$$\begin{array}{lll} \text{Ev. cont.} & E & ::= [\,] \mid E[F] \\ \text{Elem. ev. cont.} & F & ::= [\,]\,p \end{array}$$

The key innovation: substitution of evaluation contexts (a.k.a. *structural substitution*)

$$\begin{array}{lll} x[\alpha \leftarrow [\beta]E] & \triangleq & x \\ (\lambda x.p)[\alpha \leftarrow [\beta]E] & \triangleq & \lambda x.(p[\alpha \leftarrow [\beta]E]) \qquad x \text{ chosen fresh} \\ (p\,q)[\alpha \leftarrow [\beta]E] & \triangleq & (p[\alpha \leftarrow [\beta]E])\,(q[\alpha \leftarrow [\beta]E]) \\ (\mu\gamma.c)[\alpha \leftarrow [\beta]E] & \triangleq & \mu\gamma.(c[\alpha \leftarrow [\beta]E]) \qquad \gamma \text{ chosen fresh} \\ ([\alpha]p)[\alpha \leftarrow [\beta]E] & \triangleq & [\beta]E[p[\alpha \leftarrow [\beta]E]] \\ ([\gamma]p)[\alpha \leftarrow [\beta]E] & \triangleq & [\gamma](p[\alpha \leftarrow [\beta]E]) \qquad \gamma \neq \alpha \end{array}$$

Abbreviation: $p[\alpha \leftarrow \beta] \triangleq p[\alpha \leftarrow [\beta]([\,])]$

Rewriting semantics

$$\begin{array}{lll} (\lambda x.p)\,q & \rightarrow & p[x \leftarrow q] \\ F[\mu\alpha.c] & \rightarrow & \mu\alpha'.(c[\alpha \leftarrow [\alpha']F]) \\ [\beta]\mu\alpha.c & \rightarrow & c[\alpha \leftarrow \beta] \end{array}$$

$\lambda\mu$ semantics is not restricted to closed expressions at toplevel

Note: observational rules includes usual η and $\mu\alpha.[\alpha]p = p$ for α not in p

Excluded-middle and Peirce's law rephrased in $\lambda\mu$

$$\texttt{em} \triangleq \mu\alpha.[\alpha](\iota_2(\lambda a.\mu\_.[\alpha](\iota_1\, a))) \;:\; A \vee \neg A$$

where
- $\alpha : A^{\perp\!\!\perp}$
- $a : A$

$$\texttt{pl} \triangleq \lambda x.\mu\alpha.[\alpha](x\, \lambda a.\mu\_.[\alpha](k\; a)) \;:\; ((A \to B) \to A) \to A$$

where
- $x : (A \to B) \to A$
- $\alpha : A^{\perp\!\!\perp}$
- $a : A$

The need for "toplevel" continuation constants

Let us consider again the previous program of type `int` that used control.

$$
\begin{array}{c}
E[\mathtt{em}] : \mathtt{int} \\
\| \\
E[\mu\alpha.[\alpha](\iota_2(\lambda a.\mu\_.[\alpha](\iota_1\, a)))] \\
\Downarrow \\
\mu\alpha.[\alpha](E[\iota_2(\lambda a.\mu\_.[\alpha]E[\iota_1\, a])]) \\
\Downarrow \\
\text{We cannot remove the outermost } \mu\alpha!
\end{array}
$$

In $\lambda\mu$, the reduction rules are local. Nothing says that there is no further pieces of contexts to capture and it is not "safe" to remove the outermost $\mu\alpha$.

Commands are those kind of expressions that indicate where the evaluation context stops. In the presence of control operators, what is meaningful is the evaluation of *commands*.

Then, how to evaluate a closed expression? By consistency, there is no *closed* proof of $\vdash c : \perp\!\!\!\perp$. To technically solve the problem, we add a continuation constant standing for the "toplevel", whatever it is.

Note that the same phenomemon happens in Felleisen-Hieb's reduction semantics of control [1992].

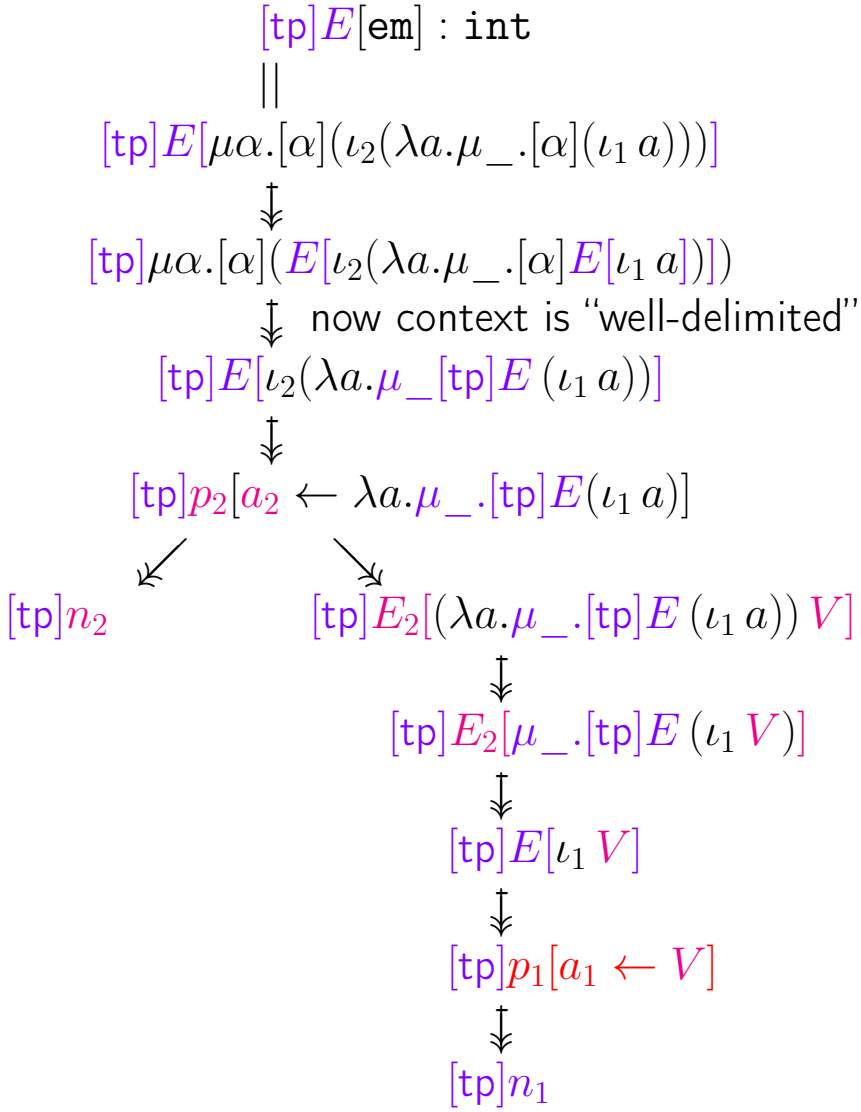
$\lambda\mu$tp: extending $\lambda\mu$ with a "toplevel" continuation constant

Syntax:

Terms	p, q	$::=$	$x \mid \lambda x.p \mid p\,q \mid \mu\alpha.c$
Commands	c	$::=$	$[\xi]p$
Atomic ev. cont.	ξ	$::=$	$\alpha \mid \mathsf{tp}$

Krivine: the "thread" model will populate the language with one toplevel continuation constant per closed program

Evaluation of closed expressions, rephrased



$\lambda\mu$tp-calculus: Its strengths

- Thanks to the distinction between term variables (the x, y, ...) and ev. cont. variables (the α, β, ...), $\lambda\mu$ is able to express by confluent local rules not only the call-by-value semantics of open expressions but also the call-by-name semantics.

  $\hookrightarrow$ Calculi such as Felleisen's $\lambda_{\mathcal{C}}$ or $\lambda_{\mathcal{KA}}$, or Krivine's λ_c can only express call-by-name evaluation of closed expressions: to know if $(k\,\texttt{callcc}(\lambda k'.M))$ is the same as $M[k' \leftarrow k]$ or not one needs to know if k is bound by a cc or by a λ [Hofmann-Streicher, 2002]

- Thanks to the use of structural substitution, it can express call-by-value semantics without computational overhead

  $\hookrightarrow$ Calculi such as Felleisen's $\lambda_{\mathcal{C}}$ or $\lambda_{\mathcal{KA}}$ have a "space leak" in call-by-value, due to the rule $F[\mathcal{C}(\lambda k.t)] \rightarrow \mathcal{C}(\lambda k'.t[k \leftarrow \lambda x.k'(F[x])])$: subexpressions of the form $(\lambda x.k'(F[x]))\,p$ needs p to be uselessly evaluated to V before being able to go to $k'(F[V])$. Also, iterated application of the rule do not compose [Ariola-H-Sabry 2007, Ariola-H 2007]

- Thanks to the explicit use of commands and ev. cont. variables, evaluation of closed expressions can easily be expressed as computations of the form $[\mathrm{tp}]p$ where tp is an ev. cont. constant denoting the toplevel continuation.

$\lambda\mu$tp-calculus: Other basic properties

- **Progress**: for closed expressions, in $[\mathrm{tp}]M$ either reduces or M is a value
- **Unique decomposition**: If $[\mathrm{tp}]M$ reduces then there is a unique decomposition of M under the form $E[N]$ such that N is a redex
- **Subject reduction**: reduction is compatible with typing
- **Termination** in the presence of simple or second-order typing

$\lambda\mu$tp-calculus: Selected expressiveness

$\texttt{call/cc}\, p$	$\triangleq$	$\mu\alpha.[\alpha](p\, \lambda x.\mu\_.[\alpha]x)$	(Scheme)
$\texttt{callcc}\, p$	$\triangleq$	$\mu\alpha.[\alpha](p\, \lambda x.\mu\_.[\alpha]x)$	(SML, ocaml)
$\texttt{throw}\, p\, q$	$\triangleq$	$p\, q$	(SML, ocaml)
$\texttt{escape}_k\, p$	$\triangleq$	$\mu\alpha.[\alpha]((\lambda k.p)\, \lambda x.\mu\_.[\alpha]x)$	(Reynolds)
$k_{[\alpha]E}$	$\triangleq$	$\lambda x.\mu\_.[\alpha](E[x])$	(Krivine)
$\texttt{cc}\, p$	$\triangleq$	$\mu\alpha.[\alpha](p\, k_{[\alpha]([\ ])})$	(Krivine)
$\texttt{catch}_\alpha\, p$	$\triangleq$	$\mu\alpha.[\alpha]p$	(Crolard)
$\texttt{throw}_\alpha\, p$	$\triangleq$	$\mu\_.[\alpha]p$	(Crolard)
$\mathcal{A}\, p$	$\triangleq$	$\mu\alpha.[\mathsf{tp}]p$	(Felleisen's $\mathcal{A}$bort)
$\mathcal{C}\, p$	$\triangleq$	$\mu\alpha.[\mathsf{tp}](p\, \lambda x.\mu\_.[\alpha]x)$	(Felleisen)
$\mathcal{K}\, p$	$\triangleq$	$\mu\alpha.[\alpha](p\, \lambda x.\mu\_.[\alpha]x)$	(Felleisen's name for `callcc`)
$\lambda y.E[\texttt{return}\, p]$	$\triangleq$	$\lambda y.\mu\alpha.E[\mu\_.[\alpha]\, p]$	(C, Java, ... up to the definition of E)
$\lambda y.E[\mathtt{J}\, f]$	$\triangleq$	$\lambda y.\mu\alpha.E[\lambda x.\mu\_.[\alpha](f\, x)]$	(Landin, up to the definition of E)

Note 1: In SML, `callcc` and `escape` do not behave the same w.r.t. to exceptions
Note 2: `catch`/`throw` have a different behavior in Lisp where, because of dynamic binding, they behave like an exceptions mechanism.
Note 3: None of operators with dynamic bindings, such as `try` or # (reset) are definable... one would need delimited control

Some other calculi of control based on "structural substitution" of evaluation contexts

- $\Lambda\mu$ [de Groote, 1994; Saurin, 2005]: turns to be a calculus of delimited control in the untyped case! [H-Ghilezan 2008]
- The `catch`/`throw` calculus [Crolard, 1999]: equivalent to $\lambda\mu$ but more "natural" bricks
- The $\overline{\lambda}\mu\tilde{\mu}$-calculus / dual calculus [Curien-H, 2000, Wadler, 2003]
  - left-right symmetry
  - a correspondence between "abstract machine" and sequent calculus
  - μ is dual to `let-in`
  - call-by-name is dual to call-by-value
  - evaluation contexts are syntactically dual to programs (even though "real world" is oriented from hypotheses to conclusions)

Also noticeable is λ_{sym} [Barbanera, Berardi, 1994] which amounts to a one-sided folding of the $\mu\tilde{\mu}$ / dual calculus.

$\Lambda\mu$

Syntax:

$$\text{Terms } p, q ::= x \mid \lambda x.p \mid p\,q \mid \underset{\uparrow}{\mu\alpha.p} \mid \underset{\uparrow}{[\alpha]p}$$

can live w/o restoring any context after saving the current one

can restore a context even if current one is still there

Call-by-name semantics: exactly as in $\lambda\mu$

$$\begin{array}{rcl}(\lambda x.p)\,q & \to & p[x \leftarrow q] \\ F[\mu\alpha.c] & \to & \mu\alpha'.(c[\alpha \leftarrow [\alpha']F]) \\ [\beta]\mu\alpha.c & \to & c[\alpha \leftarrow \beta]\end{array}$$

However, the language is more expressive. E.g.:

$$([\alpha]\mu\beta.\lambda x.t)\,u \twoheadrightarrow t[x \leftarrow u]$$

which is impossible in $\lambda\mu$.

Studied in the typed case by Ong [1996], Ong-Stewart [1997], Selinger [2000], Hofmann-Streicher [2002], ... ($\mu\alpha.p$ requires p of type $\bot$ and $[\alpha]p$ is of type $\bot$)

Studied and proved observationally complete (for finite n.f.) in the untyped case by Saurin [2005]

Crolard's `catch`/`throw`-calculus

Syntax

$$\text{Terms } p, q ::= x \mid \lambda x.p \mid p\,q \mid \mathtt{catch}_\alpha.p \mid \mathtt{throw}_\alpha\, p$$

Interpretation in $\lambda\mu$:

context is captured and immediately reinstalled, henceforth keeping a copy

$$\begin{array}{rcl} & \downarrow & \\ \mathtt{catch}_\alpha.p & \triangleq & \mu\alpha.[\alpha]\,p \\ \mathtt{throw}_\alpha\, p & \triangleq & \mu\_.[\alpha]\,p \\ & \uparrow & \end{array}$$

current context is bound to a fresh variable hence thrown away

context bound to α is restored as a functional reification of the ev. context

The interpretation is bijective up to the rule $[\alpha]\mu\beta.c = c[\beta \leftarrow \alpha]$ because of the following equalities in $\lambda\mu$:

$$\begin{array}{lcll} \mu\alpha.[\beta]p & = & \mu\alpha.[\alpha]\mu\_.[\beta]p & = \ \mathtt{catch}_\alpha\mathtt{throw}_\beta p \\ \mu\alpha.[\alpha]p & = & & \mathtt{catch}_\alpha p \\ \mu\_.[\alpha]p & = & & \mathtt{throw}_\alpha p \end{array}$$

$\overline{\lambda}\mu\tilde{\mu}$

Syntax

$$\begin{array}{lllll} \text{Commands} & c & ::= & \langle v \| e \rangle & \\ \text{Terms} & p, q & ::= & \mu\alpha.c \mid x \mid \lambda x.p & \\ \text{Ev. cont.} & e & ::= & \tilde{\mu}x.c \mid \alpha \mid p \cdot e & \end{array}$$

Basic (non-deterministic) semantics

$$\begin{array}{lrcl} (\beta) & \langle \lambda x.p \| q \cdot e \rangle & \to_v & \langle q \| \tilde{\mu}x.\langle p \| e \rangle \rangle \\ (\mu) & \langle \mu\alpha.c \| e \rangle & \to_v & c[\alpha \leftarrow e] \\ (\tilde{\mu}) & \langle p \| \tilde{\mu}x.c \rangle & \to_v & c[x \leftarrow p] \end{array}$$

Different deterministic semantics, call-by-name, call-by-value, call-by-value with strong evaluation, call-by-need, dual call-by-need, ... can be obtained by restricting (μ) and $(\tilde{\mu})$ appropriately [Curien-H, 2000; Wadler 2003; H, 2005; Ariola-Herbelin-Saurin, 2011]

Final quizz

In ocaml extended with `callcc` (or SML), what does the following interactive session display (fill in the dots):

```
# let f = callcc (fun k -> fun x -> throw k (fun y -> x+y));;
val f : int -> int = <fun>
# f 1;;
val ...
# f 2;;
val ...
# f 3;;
val ...
```