An Approach to Call-by-Name Delimited Continuations
(version with errata - mid-Nov 2007)

Hugo Herbelin
INRIA Futurs, France
Hugo.Herbelin@inria.fr

Silvia Ghilezan
Faculty of Engineering, University of Novi Sad, Serbia
gsilvia@uns.ns.ac.yu

Abstract
We show that a variant of Parigot’s $\lambda\mu$-calculus, originally due to de Groot and proved to satisfy Böhm’s theorem by Saurin, is canonically interpretable as a call-by-name calculus of delimited control. This observation is expressed using Ariola et al’s call-by-value calculus of delimited control, an extension of $\lambda\mu$-calculus with delimited control known to be equational equivalent to Danvy and Filinski’s calculus with $shift$ and $reset$. Our main result then is that de Groot and Saurin’s variant of $\lambda\mu$-calculus is equivalent to a canonical call-by-name variant of Ariola et al’s calculus. The rest of the paper is devoted to a comparative study of the call-by-name and call-by-value variants of Ariola et al’s calculus, covering in particular the questions of simple typing, operational semantics, and continuation-passing-style semantics. Finally, we discuss the relevance of Ariola et al’s calculus as a uniform framework for representing different calculi of delimited continuations, including “lazy” variants such as Sabry’s $shift$ and lazy $reset$ calculus.

Categories and Subject Descriptors F.3.3 [Studies of Program Constructs]: Control primitives; F.4.1 [Mathematical Logic]: Lambda calculus and related systems

Keywords Delimited control, Observational completeness, Böhm separability, Classical logic.

1. Introduction
Control calculi emerged as an attempt to abstractly characterise the semantics of operators like Scheme’s $call/cc$ that capture the current continuation of a computation. One first such calculus is the $\lambda_c$-calculus of Felleisen et al. (1986). Control operator are connected to classical logic, as first investigated by Griffin (1990). Hence, it is not a surprise that the “cleanest” such $\lambda$-calculus of control, namely $\lambda\mu$-calculus of Parigot (1992) comes from a computational analysis of classical natural deduction: as shown by Ariola and Herbelin (2007), $\lambda\mu$-calculus extended with a single “toplevel” continuation constant $tp$ provides a fine-grained calculus able, among other things, to faithfully express the operational semantics of $call/cc$, $C$, etc, including its own operational semantics, a property that $\lambda_c$-calculus achieves only at observational level. The reason for this success is that $\lambda\mu$-calculus treats evaluation contexts as stand-alone first-class objects while $\lambda_c$ manages evaluation contexts through their reification as regular functions.

Delimited control and completeness properties
If we concentrate on call-by-value control calculi, the introduction of delimiters can be traced back to Johnson (1987), Johnson and Duggan (1988), Felleisen (1988), and Danvy and Filinski (1989). It has then been shown in different contexts that adding such delimiters increases the expressiveness of control calculi. For instance, Sitaram and Felleisen (1990) showed how to recover a full abstraction result for call-by-value PCF with control by adding a control delimiter. As another striking example, Filinski (1994) showed that delimited control is complete for representing concrete monads, hence to simulate side-effects such as states, exceptions, etc.

Historically, delimited control came with ad hoc operators for composing continuations: Felleisen had a calculus that included a delimiter $prompt$ and a control operator $control$ (also respectively written $#$ and $F$); Danvy and Filinski had an operator $shift$ to compose continuations and an operator $reset$ to delimit them (these were also written $S$ and $(\langle \rangle )$). From Filinski (1994), it is known that $shift$ and $reset$ are equivalent to the combination of Scheme’s $call/cc$, Felleisen’s $abort$ and $reset$, and hence equivalent to $C$ and $reset$. From Shan (2004), it is known that $control$ and $prompt$ are also equivalent to $shift$ and $reset$, in spite that $control$ is semantically more complex to study than $C$ or $shift$. The simplicity of the semantics of $shift$ together with its relevance for some programing applications contributed to set $shift$ as a reference in delimited control. And this is so in spite (it seems that) it has never been studied until now as part of a dedicated $\lambda$-calculus of delimited control.

As shown by Ariola et al. (2007), a fine-grained calculus of delimited control of the strength of $shift$ and $reset$ is obtained if one starts from $\lambda\mu$-calculus and extends it first by a notation $tp$ for the “toplevel” continuation, then by a toplevel delimiter. A possible interpretation for this toplevel delimiter is as a dynamic binder of $tp$, what justifies to interpret the resulting call-by-value calculus, called $\lambda\mu tp$, as an extension of call-by-value $\lambda\mu$-calculus with a single dynamically bound continuation variable $tp$, where the hat on $tp$ emphasises the dynamic treatment of the variable. A typical analogy for the dynamic continuation variable here is an exception handling: each call to $tp$ is dynamically bound to the closest surrounding $tp$ binder, in exactly the same way as a raised exception is dynamically bound to the closest surrounding handler.

On the call-by-name side, we know no explicit mention of delimited continuations, but two results related to Böhm’s theorem (a form of observational completeness stated as a separability property) raised interesting questions: David and Py (2001) showed that Parigot’s $\lambda\mu$-calculus does not satisfy Böhm’s theorem while Saurin (2005) showed that an apparently inoffensive variant of $\lambda\mu$-calculus due to de Groote (1994) does satisfy Böhm’s theorem.
Until Saurin’s result, de Groote’s variant of Parigot’s $\lambda\mu$-calculus was merely considered in typed settings, and more particularly in settings where the continuation calls had type $\bot$. Hence, Saurin’s result is the first result revealing that de Groote’s calculus is strictly stronger than Parigot’s one in the untyped setting. In our opinion, this justifies to refer to this calculus as de-Groote–Saurin’s calculus. Our main result then is that de-Groote–Saurin’s calculus can be interpreted as a canonical call-by-name variant of call-by-value $\lambda\mu\varphi$. Capitulating on the equational correspondence between call-by-value $\lambda\mu\varphi$ and an axiomatic of Danvy and Filinski’s shift and reset given by (Kameyama and Hasegawa 2003), we can then assert that the calculus with shift and reset and de-Groote–Saurin’s calculus are two facets of the very same notion of delimited control.

Outline of the paper
Section 2 is a brief survey of Parigot’s $\lambda\mu$ and de Groote’s variant of $\lambda\mu$, including the separability properties studied by David and Py, and by Saurin. Section 3 presents call-by-value $\lambda\mu\varphi$ and its relation with shift and reset. It reviews the results by Ariola et al. (2007) and completes them by a formal presentation of the operational semantics of call-by-value $\lambda\mu\varphi$. In Section 4 we introduce a call-by-name $\lambda\mu\varphi$ and show that it is equivalent to de-Groote–Saurin’s calculus. Especially, it directly inherits separability from it. We study call-by-name $\lambda\mu\varphi$ in comparison with the call-by-value $\lambda\mu\varphi$: we propose a system of simple types for which subject reduction holds and we study the operational and continuation-passing-style semantics. A further analysis of $\lambda\mu\varphi$ leads to a classification of four calculi of delimited continuations which is discussed in Section 5. Concluding remarks are given in Section 6.

2. Parigot’s $\lambda\mu$-Calculus and Saurin’s $\Lambda\mu$-Calculus

Failure of separability in $\lambda\mu$-calculus

The $\lambda\mu$-calculus (Parigot 1992), for short $\lambda\mu$, is an untyped calculus designed to computationally interpret proofs of classical natural deduction. Its syntax is defined by the following grammar:

Parigot’s $\lambda\mu$-calculus

\[
M ::= x \mid \lambda x.M \mid M M \mid \mu c c \quad \text{terms}
\]

\[
c ::= \alpha[M] \quad \text{commands}
\]

where $x, y, z$ and their notational derivatives range over an infinite set of term variables and $\alpha, \beta, \gamma, \delta$ and their notational derivatives range over an infinite set of continuation variables (also called evaluation context variables). Expressions contain terms (called unnamed terms in Parigot) and commands (called named terms in Parigot). The operators $\lambda$ and $\mu$ are binders. Free and bound variables are defined as usual and we reason modulo renaming of bound variables. A term or command is closed if it contains no free variables.

The calculus is equipped with the call-by-name reduction rules shown in Figure 1. The notations $M[N/x]$ and $c[\beta/\alpha]$ denote usual capture-free substitutions, whereas the expression $c[[\beta]/[\square N]/\alpha]$, called structural substitution, denotes the capture-free substitution of every subterm of the form $[\alpha]M$ in $c$ by $[\beta](M N)$. For instance, we have $[[\alpha]x \lambda y.M[\alpha y]](x y N) \equiv [\gamma](\lambda x.\lambda y.M)[\alpha y]$. We equip $\lambda\mu$ with the equational theory given in Figure 2. Up to the use of $\tau_\alpha$, the rule $\eta$ is equivalent to the combination of $\nu$-rules and $\mu$-app so that the equational theory is correctly an extension with $\eta$-rules of the reflexive-symmetric-transitive closure of $\rightarrow$.

David and Py investigated Böhm’s separability for the equational theory of $\lambda\mu$ and showed that it does not satisfy Böhm’s separability\(^\text{2}\).

\begin{figure}
\centering
\begin{align*}
\beta : & \quad (\lambda x.M) N \rightarrow M[N/x] \\
\mu_{\text{app}} : & \quad (\mu c c) N \rightarrow \mu \beta \cdot c[[\beta]/[\square N]/\alpha] \quad \beta \text{ fresh} \\
\mu_{\text{var}} : & \quad [\beta] \mu c c \rightarrow c[\beta/\alpha]
\end{align*}
\caption{Reduction rules of $\lambda\mu$-calculus}
\end{figure}

\begin{figure}
\centering
\begin{align*}
\beta : & \quad (\lambda x.M) N \equiv M[N/x] \\
\mu_{\text{app}} : & \quad [\beta](E_\alpha[\mu c c]) \equiv c[[\beta]/E[\alpha]] \\
\eta : & \quad \lambda x.(M x) \equiv M \quad \text{if } \alpha \text{ not free in } M \\
\eta : & \quad \lambda x.(M x) \equiv M \quad \text{if } x \text{ not free in } M
\end{align*}
\caption{Equational theory of $\lambda\mu$-calculus}
\end{figure}

Separability in $\Lambda\mu$-calculus

Saurin (2005) showed that completeness can be recovered by relaxing the syntax of $\lambda\mu$ so that the category of commands (i.e. named terms) becomes a subcategory of the one of terms. The syntax used by Saurin was already considered by de Groote (1994), Ong (1996), Selinger (2001). This syntax was considered as an alternative to Parigot’s original $\lambda\mu$. Saurin’s result sheds new light on the relation between the two calculi. Following Saurin, we call $\Lambda\mu$ the calculus based on de Groote’s syntax equipped with the same reduction rules and equational theory as in $\lambda\mu$. The syntax is\(^\text{3}\):

\[
\Lambda\mu\text{-calculus}
\]

\(^2\)David and Py actually had $\mu_{\text{var}}$ and $\mu_{\text{app}}$ instead of $(\mu_{\text{var}})$ and their rules were oriented as rewrite rules. They also considered the rule $\nu : \mu c c \rightarrow \lambda x.\mu c c((x z)/x)/\alpha$ but this rule is redundant for equational reasoning as it derives from $(\mu_{\text{var}})$ and $\eta$. The initial motivation for $\nu$ was to turn their system of reduction rules $(\beta), (\eta), (\mu_{\text{app}}), (\mu_{\text{var}})$ into a confluent system of reduction. In fact, $\nu$ hides an $\eta$-expansion and it is enough to formulate $\eta$ in the expansion way to get a confluent system, without any need for $\nu$.

\(^3\)Saurin’s syntax is a bit different as he writes $M \alpha$ for what we write $[\alpha]M$ but that is really here a matter of notation.
\[
M \ ::= \ x \mid \lambda x. M \mid M \cdot M \mid \mu \alpha. M \mid [\alpha]M
\]

In \(\Lambda\mu\), there are more evaluation contexts. They are defined by:
\[
D_n := \square \mid D_n[\square]E_v[\square].
\]

**Theorem 2** (Saurin 2005). A normal term is called canonical if it contains no subterms of the form \([\alpha]\lambda x. M\). Now, if the closed canonical normal terms \(M\) and \(N\) in \(\Lambda\mu\) are not equal by the equational theory in Figure 2, then they are separable, i.e., for any two variables \(x\) and \(y\), there exists a context \(D_n\), such that \(D_n[M] = x\) and \(D_n[N] = y\).

This may look strange as the only change is a change in the syntax of the terms. In fact, the difference lies in the rule (\(\mu_{\text{var}}\)) which in the case of \(\lambda\mu\) can only occur in a configuration of the form:
\[
M(\mu\gamma,\beta)\mu\alpha.c \rightarrow M(\mu\gamma,\beta(\alpha/\gamma))
\]
while in the case of \(\lambda\mu\), it can also occur in a configuration of the form:
\[
M(\beta)\mu\alpha.c \rightarrow M(\epsilon(\beta/\gamma))
\]
so that the computational effect of any \(\mu\alpha.c\) can be cancelled if we succeed in putting it in a context of the form \([\beta]\square\). This last property is the main reason why Saurin’s completeness theorem works.

### 3. A Review of Call-by-Value \(\lambda\mu\)-Calculus

The \(\lambda\mu\)-calculus was introduced by Ariola et al. (2007). It is an extension of the call-by-value variant of \(\lambda\mu\) obtained by adding a single dynamically bound continuation variable \(\top\). Ariola et al’s \(\lambda\mu\)-calculus is a fine-grained calculus of delimited continuations in which, as an example, the semantics of Danvy and Filinski’s shift and reset operators can be simulated. In the original formulation of \(\lambda\mu\), the control operator of the language was called \(C\) in spite that \(\lambda\mu\) is based on Parigot’s structural substitution, as in \(\lambda\mu\), rather than on substitution of continuations reified as ordinary functions, as it is the case in Felleisen \(\lambda\mu\)-calculus. Here we redefine \(\lambda\mu\) using the \(\mu\) notation instead of a \(C\) notation. The so-reformulated calculus is called the call-by-value \(\lambda\mu\)-\(\top\).

#### 3.1 Syntax and reduction rules

\[
\begin{align*}
M, N & ::= V \mid M \cdot M \mid \mu q.e \quad (\text{terms}) \\
V & ::= x \mid \lambda x. M \quad (\text{values}) \\
e & ::= [\square].M \quad (\text{commands}) \\
q & ::= \square \mid \square[\square]E_v[\square] \quad (\text{ev. context variables})
\end{align*}
\]

**Figure 3. Syntax of \(\lambda\mu\)-\(\top\)**

The syntax of \(\lambda\mu\)-\(\top\) is given in Figure 3. We also define call-by-value evaluation contexts by
\[
E_v := \square \mid E_v[\square]M[\square] \mid E_v[V \square] \quad (\text{eval. contexts}) \\
D_v := \square \mid D_v[q.E_v[\square]\top\square]\quad (\text{nested eval. context})
\]

and the notations \(E_v[M]\) and \(D_v[e]\) stand for the terms and commands obtained by plugging \(M\) into \(E_v\) and \(e\) into \(D_v\) seen as expressions with one-place-holder.

The reduction semantics of call-by-value \(\lambda\mu\)-\(\top\) is given in Figure 4. The notation \(e[\square]E_v/\alpha\) stands for structural substitution of evaluation contexts as in \(\lambda\mu\) (see Section 2). The substitutions are capture-free for term and continuation variables but \(\top\) gets captured (e.g. the substitution of \(x\) by \(h(\mu\delta.\top\square)\)) \((\mu\delta.\beta)z\) in \([\alpha]\{f \mu\beta.(g \mu\top.(\beta x))\}) gives as result the expression \([\alpha]\{f \mu\beta.(g \mu\top.(\beta\beta)(g \mu\gamma.(\beta x)))(h(\mu\top.(\beta\square)(\mu\delta.\beta)z))\})).

\[
\begin{align*}
\beta_v & : (\lambda x. M) V \rightarrow M[V/x] \\
\mu_v & : [\square].E_v[\mu q.e]/\alpha \rightarrow c[q/\alpha] \\
\eta_v & : \mu\top.(\top\square) V \rightarrow V \\
\mu_{\top} & : \top.(\top\square) \rightarrow c \\
\mu_{\text{let}} & : \mu visible.\top.[\text{let}(\lambda x. M)] \rightarrow (\lambda x.\mu\top.[\square]M)(\mu\top.c) \\
\mu_{\text{let}} & : \mu\alpha.\top.[\text{let}(\lambda x. \lambda\alpha. M)] \rightarrow (\lambda x.\mu\alpha.\top.[\square]M)(\mu\top.c) \\
\eta_v & : \mu\alpha.\square \rightarrow M \\
\beta_v & : (\lambda x. E_v[x]) M \rightarrow E_v[M/x] \\
\beta_v & : (\lambda x. M) V \rightarrow M[V/x]
\end{align*}
\]

**Figure 4. Reductions of call-by-value \(\lambda\mu\)-\(\top\)**

A simple analysis of the syntax and rules shows that the unique context lemma (see Felleisen and Friedman 1986) holds: any closed command which is not of the form \([\top]\lambda x. M\) has a unique decomposition as \(D_v[[\top]\square]E_v[M]\) where either \(M\) or \([\top]\square]E_v[M]\) is a redex. Hence the reduction system is complete for the evaluation of closed programs. A term that contains no redex at all is said to be normal.

#### 3.2 Equational theory

\[
\begin{align*}
\beta_v & : (\lambda x. M) V \rightarrow M[V/x] \\
\mu_v & : [\square].E_v[\mu q.e]/\alpha \rightarrow c[q/\alpha] \\
\eta_v & : \mu\top.(\top\square) V \rightarrow V \\
\mu_{\top} & : \top.(\top\square) \rightarrow c \\
\mu_{\text{let}} & : \mu\alpha.\top.[\text{let}(\lambda x. \lambda\alpha. M)] \rightarrow (\lambda x.\mu\alpha.\top.[\square]M)(\mu\top.c) \\
\mu_{\text{let}} & : \mu\alpha.\top.[\text{let}(\lambda x. \lambda\alpha. M)] \rightarrow (\lambda x.\mu\alpha.\top.[\square]M)(\mu\top.c) \\
\eta_v & : \mu\alpha.\square \rightarrow M \\
\beta_v & : (\lambda x. E_v[x]) M \rightarrow E_v[M/x]
\end{align*}
\]

1. \(\alpha\) not free in \(M\) 2. \(x\) not free in \(V\) 3. \(x\) not free in \(E_v\)

**Figure 5. CPS-complete theory of call-by-value \(\lambda\mu\)-\(\top\)**

The equational theory of \(\lambda\mu\)-\(\top\) is given in Figure 5. Note that the equation (\(\mu_v\)) generalises the effect of the rules (\(\mu_{\text{app}}\)) (\(\mu_{\text{let}}\)) (up to the use of \(\eta_v\)).

#### 3.3 Simple typing

\[
\begin{align*}
X & \in \text{TypeConstants} \\
A, B, T, U & ::= X \mid A \rightarrow B \\
\Gamma & ::= \cdot \mid \Gamma, x : A \\
\Delta & ::= \cdot \mid \Delta, \alpha : A \\
\Gamma \vdash x : A \rightarrow M : B \rightarrow U; b : \top : T \\
\Gamma \vdash \lambda x.M : (A \rightarrow B) \rightarrow U; b : \top : T \\
\Gamma \vdash \lambda x.M : A \rightarrow B \\
\Gamma \vdash \lambda \alpha.e : A; b : \top : T \\
\Gamma \vdash \lambda \alpha.e : A; b : \top : T \\
\Gamma \vdash \lambda \alpha.e : A; b : \top : T \\
\Gamma \vdash \lambda \alpha.e : A; b : \top : T \\
\Gamma \vdash \lambda \alpha.e : A; b : \top : T \\
\Gamma \vdash \lambda \alpha.e : A; b : \top : T
\end{align*}
\]

**Figure 6. Simple typing of call-by-value \(\lambda\mu\)-\(\top\)-calculus**

The calculus \(\lambda\mu\)-\(\top\) is basically an untyped calculus. Still, it is possible to constrain it with a type system. Ariola et al’s adaptation
of Danvy and Filinski’s system of simple types (Danvy and Filinski 1989) is given in Figure 6. As in Parigot, the typing context of continuation variables is on the right of the sequent. We use the symbol \( \bot \) in the typing judgements of commands to emphasise that they have no type.

A continuation of type \( A_T \) is a continuation whose own continuation is a call to a top-level continuation \( \hat{t}p \) expected of type \( T \), i.e. whose own continuation is expected to be called in a context where the surrounding \( \mu tp \) has type \( T \). A judgement \( \Gamma \vdash M : A_T ; \Delta \vdash \hat{t}p : U \) says that \( M \) is a term which expects a continuation of type \( A_T \); the possible capture by \( M \) of its evaluation context will be dispatched in contexts where the dynamically closest surrounding \( \mu tp \) is of type \( T \). In the judgement, \( U \) is the type of the actual closest surrounding top-level \( \mu tp \). To propagate the type information of the dynamically bound \( \hat{t}p \), arrows have effects: a term of type \( A_T \rightarrow B_U \) is a term that expects a value of type \( A \) and returns a code of type \( B \) which:

- may capture its surrounding context and move it in a place where the top-level has type \( U \),
- eventually itself calls the top-level continuation with a value of type \( T \).

### 3.4 Continuation-passing-style semantics

\[
x^+ \triangleq x \\
(\lambda x. M) + \triangleq \lambda k.\lambda \nu.\lambda x. [M] \ k \ \nu \\
[V] k \ \nu \triangleq k \ V^+ \\
[M] \ N \ k \ \nu \triangleq [M] (\lambda \nu'. \lambda f. ([N] (f k) \ \nu')) \ \nu \\
[\mu \nu. \rho] \ k \ \nu \triangleq \{ \rho k \}/[\alpha] \\
[\mu \nu. \rho \ p] \ k \ \nu \triangleq \{ \rho k \}(k \ \nu) \\
([\mu \nu. \rho] p) \ M \ \nu \triangleq [M] (\lambda k. \ \nu) \\
([\alpha] p) \ M \ \nu \triangleq [M] \ \alpha \ \nu
\]

**Figure 7.** Call-by-value CPS translation of \( \lambda \mu \hat{t}p \)

We give in Figure 7 a continuation-passing-style (CPS) semantics in Fischer style (see Fischer 1972) for call-by-value \( \lambda \mu \hat{t}p \). Ariola et al. (2007) showed that this CPS translation can be factorised (up to currying and \( \eta \)-conversion) as the composition first of a state-passing-style transformation to call-by-value \( \lambda \mu \) with subtraction, then of a standard call-by-value CPS translation to \( \lambda \)-calculus with pairs.

### 3.5 Equational correspondence with Kameyama and Hasegawa’s axiomatisation of a calculus with shift and reset

Danvy and Filinski originally defined the operators \( \text{shift} \) and \( \text{reset} \) by their continuation-passing semantics. We show in this section that call-by-value \( \lambda \mu \hat{t}p \) contains \( \text{shift} \) and \( \text{reset} \) in the sense that they contain operators of which the CPS semantics is the defining semantics of \( \text{shift} \) and \( \text{reset} \).

In a second step, we show that call-by-value \( \lambda \mu \hat{t}p \) contains no more than call-by-value \( \lambda \)-calculus extended with \( \text{shift} \) and \( \text{reset} \). This is shown by exhibiting an equational correspondence with Kameyama and Hasegawa’s theory of call-by-value \( \lambda \)-calculus with \( \text{shift} \) and \( \text{reset} \), a theory known to exactly capture the CPS semantics of \( \lambda \)-calculus with \( \text{shift} \) and \( \text{reset} \) (Kameyama and Hasegawa 2003).

The operators \( \text{shift} \) and \( \text{reset} \) are defined as follows:

\[
\begin{align*}
\text{shift} & : S M \triangleq \mu \alpha.\{tp\}(\lambda x. \mu \hat{t}p.\alpha x) \\
\text{reset} & : \langle M \rangle \triangleq \mu \hat{t}p.\{tp\}M
\end{align*}
\]

The justification that these encodings define \( \text{shift} \) and \( \text{reset} \) is given by the following proposition taken from Ariola et al. (2007):

**Proposition 3 (Simulation of \( \text{shift} \) and \( \text{reset} \) in \( \lambda \mu \hat{t}p \)).** The CPS semantics of \( S M \) and \( \langle M \rangle \) are:

\[
\begin{align*}
[S(\lambda q. M)] k \ \nu & = [M][\lambda k'.\lambda \nu. k'(\nu)/q](\lambda k) \ \nu \\
[\langle M \rangle] k \ \nu & = [M](\lambda k. k(\nu))
\end{align*}
\]

which coincide with the defining CPS semantics of \( \text{shift} \) and \( \text{reset} \) in Danvy and Filinski (1989).

Let now \( (\lambda s, =_{KH}) \) be \( \lambda \)-calculus equipped with \( \text{shift} \) and \( \text{reset} \) and with the axioms of Kameyama and Hasegawa (2003). Let \( (\lambda \mu \hat{t}p, =) \) be call-by-value \( \lambda \mu \hat{t}p \) equipped with the axioms given in Figure 5. Let \( \lambda \mu \hat{t}p_0 \) be the subset of expressions of \( \lambda \mu \hat{t}p \) that do not contain free continuation variables. We define \( (\lambda \mu \hat{t}p_0, =) \) as the restriction of \( (\lambda \mu \hat{t}p, =) \) to the expressions of \( \lambda \mu \hat{t}p_0 \).

The interpretation of \( \lambda \mu \hat{t}p \) in Kameyama and Hasegawa’s calculus works as follows: each continuation variable \( \alpha \) is injectively mapped to a fresh ordinary variable \( k_\alpha \), \( \mu \hat{t}p.\{tp\} t \) is interpreted as \([M]\), \( \mu \hat{t}p.\{tp\} t \) as \([k_\alpha M], \mu \alpha.\{\beta\} M \) as \((S(k_\alpha, k_\beta) M)\) and \( \mu \alpha.\{tp\} M \) as \([S(\lambda k_\alpha, \lambda k_\beta) M]\).

The next theorem expresses the equational correspondence (in the sense of Sabry and Felleisen (1993)) between call-by-value \( \lambda \mu \hat{t}p \) and Kameyama and Hasegawa’s calculus:

**Theorem 4 (Ariola et al. 2007).** The theories \( (\lambda s, =_{KH}) \) and \( (\lambda \mu \hat{t}p_0, =) \) are isomorphic.

**Corollary 5 (Ariola et al. 2007).** The theory \( (\lambda \mu \hat{t}p, =) \) is complete with respect to \( \beta \) and \( \eta \) through the CPS semantics of call-by-value \( \lambda \mu \hat{t}p \).

The addition of a continuation delimiter was used in Sitaram and Felleisen (1990) to recover some completeness property that was lost in the move from \( \lambda \)-calculus to \( \lambda c \)-calculus. Our analysis of Saurin’s separability result for call-by-name \( \lambda \mu \) in Section 4.3 shows that Böhm’s theorem, which amounts to observational completeness for normal terms, is also recovered by the addition of a continuation delimiter. This suggests the following conjecture:

**Conjecture 6 (Ariola et al. 2007).** The theory \( (\lambda \mu \hat{t}p, =) \) satisfies Böhm’s theorem, i.e., for any equationally distinct closed normal forms \( M \) and \( N \), there is a context \( D_\alpha[\{q\} E_\alpha] \) such that \( \mu \hat{t}p. D_\alpha[\{q\} E_\alpha[M]] = x \) and \( \mu \hat{t}p. D_\alpha[\{q\} E_\alpha[N]] = y \).

### 3.6 Operational semantics

The operational semantics in “natural” style is characterised by a deterministic application of the reduction rules at the head of a computation (so-called weak-head reduction). It is common to formulate the operational semantics on terms but we rather do it on commands what allows for a more uniform characterisation of normal forms. Typically, when formulated on terms, a term like \( \mu \alpha.\{\alpha \} V \) can be reduced further to \( V \) only if \( \alpha \) does not occur in \( V \) but it cannot be reduced further if \( \alpha \) does occur. To the contrary, if the same term is reduced as part of a command, as in \( [\beta][\mu \alpha.\{\alpha \} V] \), then the resulting command uniformly reduces to \( [\beta][V]\{\beta/\alpha\} \) independently of whether \( \alpha \) occurs or not in \( V \). The operational semantics, that we do not only define on closed terms as it is common but also on terms with free variables, is given by the equations:

\[
\begin{align*}
\beta : D_\alpha[\{q\} E_\alpha[(\lambda x. M) V]] & \mapsto D_\alpha[\{q\} E_\alpha[M[V/x]]] \\
\mu \alpha : D_\alpha[\{q\} E_\alpha[\mu \beta.\{c\}]] & \mapsto D_\alpha[c[\{q\} E_\alpha/\beta]] \\
\eta \alpha : D_\alpha[\{q\} E_\alpha[\mu \hat{t}p.\{tp\} V]] & \mapsto D_\alpha[\{q\} E_\alpha[V]]
\end{align*}
\]

Obviously \( \mapsto \) is included in \( \mapsto^* \) of which it constitutes on commands a convenient level of abstraction. We say that \( c \) is a weak-head normal command if for no \( c', c \mapsto c' \). Weak-head normal
commands are either of the form $[[p]]V$, or of the form $D_v[[\alpha]]V$, or of the form $D_v[[q]]E_v[x V]]$.

Operational semantics can be also described by using an abstract machine. Evaluation in an abstract machine is closely related to cut-elimination in Gentzen’s sequent calculus (see e.g. Herbelin 1995, 1997; Danos et al. 1996) while, contrastingly, operational semantics in “natural” style is related to Gentzen’s natural deduction. Sequent calculus proofs can be represented in $\lambda\mu\bar{\mu}$-calculus (Curien and Herbelin 2000; Herbelin 2005) extended with a calculus of explicit substitutions (see e.g. Herbelin 2001) to represent closures and environments, in the spirit of Hardin et al. (1996). The language of the abstract machine for call-by-value $\lambda\mu\bar{\mu}$-calculus is shown in Figure 8 and the reduction steps are given in Figure 9. The syntax for evaluation contexts and states is reminiscent of $\lambda\mu\bar{\mu}$-calculus. Stacks are identified with evaluation contexts. The construction $q[e]$ refers to the continuation bound to $q$ in the environment $e$. The construction $M[e] \cdot K$ denotes the continuation which first applies $M[e]$ before continuing with continuation $K$. The construction $\mu\bar{\mu}.(V[e] || \alpha)K$ denotes the continuation that binds $\alpha$ to the current result, say $W$, so that computation continues with code $V$ in environment $e$ and continuation $W \cdot K$. The construction $\langle W || K \rangle$ denotes the interaction of a term $M$ in context $K$. This construction comes in three flavours. In $\langle M[e] || K \rangle_{eval} [S]$, the term $M$ in environment $e$ has the control on what is going next.

At some point of the evaluation process, the term gets evaluated and the control is transfered to the evaluation context. This corresponds to a state $\langle W || K \rangle_{cont} [S]$. At some point, both the term and the evaluation context are in “evaluated” form and a “logical” interaction happens. This corresponds to states of the form $\langle V[e] || W \cdot K \rangle_{logic} [S]$. Specific evaluation rules correspond to each of these different states. We write $e(\alpha)$ for the binding of $\alpha$ in $e$ and similarly for $e(x)$. The dynamically bound variables are bound in an environment $S$ that remains global (it is not stored in closures). Note that when the dynamic continuation variable $tp$ is referred to, not only the continuation bound to $tp$ is restored but the binding is removed so that the next call to $tp$ will refer to the next binding of the global environment.

Observe that the abstract machine is designed to return the weak-head normal form not only of closed programs but also of terms with free variables (see the “stop” transitions). Final result reconstruction in terminal states turns explicit substitutions into effective substitutions. Result reconstruction turns closures of values into ordinary values. It also uses the operation $S^T$ that builds contexts for command of the form $D_v$ and the operation $K^T$ that builds contexts of the form $[q]E_v$. These operations are defined by the fol-

Figure 8. Specific components of the abstract machine for call-by-value $\lambda\mu\bar{\mu}$-calculus

| $K$ | ::= $q[e] \cdot M[e] \cdot K$ | (evaluation contexts) |
| $[S]$ | ::= $[]$ | (dynamic environment) |
| $W$ | ::= $V[e]$ | (closure of value) |
| $e$ | ::= $[\cdot] || W[e] || e$ | (environments) |
| $s$ | ::= $\langle W || K \rangle_{cont} [S] || (M[e] || K)_{eval} [S] || (V[e] || W \cdot K)_{logic} [S]$ | (states) |

\[
\begin{align*}
\text{Control owned by the evaluation context} & \quad \langle W || N[e] \cdot K \rangle_{cont} [S] \rightarrow \langle N || e \rangle_{eval} [S] \quad \langle W || \mu\bar{\mu}.(W || \alpha \cdot K) \rangle_{eval} [S] \\
\text{Control owned by the term} & \quad \langle M \cdot N || e \rangle_{eval} [S] \rightarrow \langle M || N[e] \cdot K \rangle_{eval} [S] \\
\text{Control owned by the value} & \quad \langle \lambda x. M || W \cdot K \rangle_{logic} [S] \rightarrow \langle M || x = W \cdot e \rangle_{eval} [S] \\
\end{align*}
\]

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

\[
\langle M [] || \text{tp} [] \rangle_{eval} []
\]
call/cc \(\mu\) of the which would mean that reifies the whole undelimited continuation including the exception handlers, language, assuming that a 4 substitution, we have:

\begin{align*}
&\mu = \lambda_\mu.M \\
&\mu_\alpha.M \\
&\mu_{\alpha/\beta}.M
\end{align*}

\[\mu_{\alpha/\beta}.M\]

which coincide with the rules \((\text{S}_\lambda)\) and \((\text{val})\) in Biernacka et al. (2003, Section 4.4) through the identification of \(C_2\) with \(D_2\) and of \(C_1\) with, \([\text{tp}]E_\rel\) if in position of context, or \(\lambda_\text{tp}.(E_\rel(x))\) if in position of term. As for the rules \((\beta_\lambda)\) and \((\beta_{\text{cc}})\) in Biernacka et al. (2003) they both are instances of \((\beta)\).

### 3.7 Expressiveness

Call-by-value \(\lambda_\mu\text{tp}\) is fine-grained enough to directly simulate the operational semantics of most standard control operators. Let \(C\) and \(A\) be Felleisen’s \(C\) and \(\text{abort}\) operators. Let \(\text{call/cc}\) be the implementation of \(\text{call-cc}\) in Scheme. Let \(M\) handle patterns and \(\text{raise}\) \(M\) be the constructions of the exception mechanism in SML (i.e. of \text{try} \(M\) with \text{patterns} and \text{raise} \(M\) in O’Caml). Let \(\text{Val}\) be a special exception with one argument. In addition to the definition of \(\mathcal{S}\) \(M\) and \(\langle M \mid \mid \text{tp}\rangle\) above, we have the following encodings:

\[\begin{array}{l}
\text{A} M \triangleq \mu_{\text{tp}}M \\
\text{C} (\lambda_\mu. M) \triangleq \mu\alpha.\beta.M \lambda_\text{tp}.\mu\alpha.\beta.\alpha x \\
\text{call/cc} (\lambda_\mu. M) \triangleq \mu\alpha.\beta.M \lambda_\text{tp}.\mu\alpha.\beta.\alpha x \\
\text{raise} M \triangleq \text{case} \mu\text{tp}.(\text{tp})(\text{Val} M) \text{ of } \\
\quad | \text{Val} x \Rightarrow x \\
\quad | \text{patterns} \\
\quad | x \Rightarrow \mu_\alpha.\beta.\text{tp} x
\end{array}\]

Let us show for instance how the operational semantics of Scheme’s \text{call/cc} is faithfully simulated\(^5\). Thanks to structural substitution, we have:

\[E_\rel[\text{call/cc} \lambda_\mu. M] \Rightarrow E_\rel[M \lambda_\text{tp}.A E_\rel[x]/k]\]

\[\beta : (\lambda_\mu. M) N \Rightarrow M[N/x] \\
\mu^{\text{app}} : (\mu_\text{tp}. c) M \Rightarrow \mu_\beta.\text{cc}[[\beta](\text{tp} M)/\alpha] \text{ fresh} \\
\mu^{\text{var}} : [\beta]\mu_\text{tp}. c \Rightarrow c[\beta/\alpha] \\
\eta^{\text{tp}}_\alpha : \mu_\text{tp}.(\text{tp}) M \Rightarrow M \text{ even if \text{tp} occurs in } M
\]

### 4. Call-by-Name \(\lambda_\mu\text{tp}\)-Calculus

In this section we introduce a call-by-name variant of \(\lambda_\mu\text{tp}\). We formalise a reduction semantics, an equational theory, a system of simple types, a continuation-passing-style semantics and an operational semantics. Call-by-name \(\lambda_\mu\text{tp}\) is an extension of \(\lambda_\mu\) that we show to be isomorphic to \(\lambda_\mu\). From the programming viewpoint of view, call-by-name \(\lambda_\mu\text{tp}\), is a bizarre calculus. In an attempt to clarify how it behaves, we end the section by an example.

### 4.1 Syntax and reduction rules

The syntax of \(\lambda_\mu\text{tp}\) was given in Figure 3. The reduction rules of the call-by-name variant of \(\lambda_\mu\text{tp}\) are in Figure 10. They extend the reduction rules of \(\lambda_\mu\) in Figure 1 by one rule, called \(\eta^{\text{tp}}_\alpha\), that is similar to the equation \(\eta_\alpha\) but without any constraints on whether \text{tp} occurs or not in \(M\).

\[\quad \beta : (\lambda_\mu. M) N \Rightarrow M[N/x] \\
\quad \mu^{\text{app}} : (\mu_\text{tp}. c) M \Rightarrow \mu_\beta.\text{cc}[[\beta](\text{tp} M)/\alpha] \text{ fresh} \\
\quad \mu^{\text{var}} : [\beta]\mu_\text{tp}. c \Rightarrow c[\beta/\alpha] \\
\quad \eta^{\text{tp}}_\alpha : \mu_\text{tp}.(\text{tp}) M \Rightarrow M \text{ even if \text{tp} occurs in } M
\]

**Figure 10.** Reductions of call-by-name \(\lambda_\mu\text{tp}\)

We say that a term or a command is normal if it contains no reduct. Call-by-name evaluation contexts are defined as follows:

\begin{align*}
E_n & \triangleq \square | E_n[\square M] \text{ (linear eval. contexts)} \\
D_n & \triangleq \square | D_n[\alpha E_n[\text{tp} \square]] \text{ (nested linear eval. context)}
\end{align*}
Call-by-name evaluation contexts are called *linear* because they do not erase or duplicate the terms they expect in their hole.

An analysis of the syntax and rules shows that a formal context lemma holds: any command with α as free variable is either of the form \(\Delta \alpha \), or of the form \(\mu \delta M\), or of the form \(\Delta \alpha \). This is in fact a curious result: \(\mu \) blocks the reduction and there is not much hope to compute something interesting without at least one free continuation variable at hand.

### 4.2 Equational theory

The equational theory of call-by-name \(\lambda \mu \tilde{\text{p}}\) extends the equational theory of \(\lambda \mu\) with \((\mu\nu\text{ar})\) (an analog of the rule \((\mu\text{var})\)) for the special continuation \(\tilde{\text{p}}\) and with \((\eta\nu\theta)\) seen as equation. It is given in Figure 11.

\begin{align*}
\beta : \quad (\lambda x.M) \, N & \quad = \quad M[N/x] \\
\mu_\alpha : \quad [\beta](E_\alpha[\mu_\alpha . c]) & \quad = \quad c[[\beta]E_\alpha/\alpha] \\
\eta_\alpha : \quad \mu_\alpha.\alpha M & \quad = \quad M \quad \text{if \(\alpha\) not free in \(M\)} \\
\eta : \quad \lambda x.(M \, x) & \quad = \quad M \quad \text{if \(x\) not free in \(M\)} \\
\mu_\theta : \quad [\tilde{\text{p}}]\mu_\theta . c & \quad = \quad c \\
\eta_\theta : \quad \mu_\theta .[\tilde{\text{p}}]\mu_\theta . M & \quad = \quad M
\end{align*}

**Figure 11.** Equational theory of call-by-name \(\lambda \mu \tilde{\text{p}}\)-calculus

### 4.3 Equational correspondence with \(\Lambda \mu\)

The calculus \(\Lambda \mu\) is derived from \(\lambda \mu\) by relaxing the syntax and keeping the same theory. We now show how \(\Lambda \mu\) can be contrastingly restated as a strict extension of \(\lambda \mu\). This extension is precisely our call-by-name variant of \(\lambda \mu \tilde{\text{p}}\).

**Embedding of \(\Lambda \mu\) into call-by-name \(\lambda \mu \tilde{\text{p}}\)** A naive way to interpret \(\Lambda \mu\) in \(\lambda \mu\), actually in any extension \(\lambda \mu \tilde{\text{p}}\), is to interpret any \(\lambda \mu\)-term \(\mu_\alpha M\) as the \(\lambda \mu\)-term \(\mu_\alpha .[\tilde{\text{p}}]M\) and to interpret any \(\lambda \mu\)-term \([\alpha]M\) as the \(\lambda \mu\)-term \(\mu_\alpha .[\tilde{\text{p}}][\alpha]M\). Formally, this corresponds to the following embedding \(\Pi\) of \(\Lambda \mu\) into \(\lambda \mu \tilde{\text{p}}\):

\[\Pi(\beta) \quad \triangleq \quad \beta\]
\[\Pi((\lambda x.M) \, N) \quad \triangleq \quad \lambda x.\Pi(M) \, \Pi(N)\]
\[\Pi([\alpha]M) \quad \triangleq \quad \Pi([\alpha]M)\]

This translation is not defined on continuation variables, since they are not part of the formal syntax. Nevertheless we can derive the following property:

**LEMMA 8.**

\[\Pi(M[\beta/\alpha]) = \Pi(M)[\beta/\alpha]\]

We then check that all rules of \(\Lambda \mu\) can be simulated in \(\lambda \mu\), all but the \((\mu\text{var})\) rule. Indeed,

\[\Pi([\beta]\mu_\alpha . M) \quad \equiv \quad \mu_\theta .[\tilde{\text{p}}]\mu_\alpha .[\tilde{\text{p}}]\Pi(M)\]
\[\quad \overset{\mu\text{var}}{\longrightarrow} \quad \mu_\theta .[\tilde{\text{p}}]\Pi(M)[\beta/\alpha]\]
\[\quad \equiv \quad \mu_\theta .[\tilde{\text{p}}]\Pi(M)[\beta/\alpha]\]

but \(\mu_\theta .[\tilde{\text{p}}]\Pi(M)[\beta/\alpha]\) has no reason to be equal to \(\Pi(M[\beta/\alpha])\) in \(\lambda \mu\). This is actually expected since \(\Lambda \mu\) is observationally complete for normal terms but \(\lambda \mu\) is not. However, in the extended calculus \(\lambda \mu \tilde{\text{p}}\), this equality holds. Indeed, we now have:

**PROPOSITION 9.** If \(M = N\) in \(\Lambda \mu\) then \(\Pi(M) = \Pi(N)\) in \(\lambda \mu \tilde{\text{p}}\)

**Embedding of call-by-name \(\lambda \mu \tilde{\text{p}}\) into \(\Lambda \mu\)** We now want to show that our call-by-name \(\lambda \mu \tilde{\text{p}}\), i.e. \(\lambda \mu\) extended with rules \((\mu_\nu\theta)\) and \((\eta\nu\theta)\), is indeed equivalent to \(\Lambda \mu\). Let us define the following converse translation:

\[\Sigma(x) \quad \triangleq \quad x\]
\[\Sigma((\lambda x.M) \, N) \quad \triangleq \quad \lambda x.\Sigma(M) \, \Sigma(N)\]
\[\Sigma([\mu_\alpha .[\tilde{\text{p}}]M]) \quad \triangleq \quad \Sigma([\mu_\alpha .[\tilde{\text{p}}][\alpha]M])\]
\[\Sigma(\mu_\theta .[\tilde{\text{p}}][\theta]M) \quad \triangleq \quad \mu_\theta .[\tilde{\text{p}}]\Sigma(M)\]

**PROPOSITION 10.** If \(M = N\) in call-by-name \(\lambda \mu \tilde{\text{p}}\), then \(\Sigma(M) = \Sigma(N)\) in \(\Lambda \mu\).

Since moreover, \(\Sigma(\Pi(M)) = M\) and \(\Sigma(\Pi(N)) = M\) (using \((\mu_\nu\theta)\)), we get:

**THEOREM 11** (Equational correspondence). \(\Lambda \mu\) equipped with the equations of Figure 2 and call-by-name \(\lambda \mu \tilde{\text{p}}\) equipped with the equations of Figure 11 equationally correspond.

**REMARK.** Call-by-name \(\lambda \mu \tilde{\text{p}}\) and \(\Lambda \mu\) form more than an equational correspondence: their reduction systems are also bisimilar: \(M \rightarrow N\) if \(\Pi(M) \rightarrow \Pi(N)\) and \(\Sigma(M) \rightarrow \Sigma(N)\) iff \(M \rightarrow N\). In particular, normal forms match.

**Observational completeness of normal forms in \(\lambda \mu \tilde{\text{p}}\)** As a consequence of the isomorphism, we have:

**COROLLARY 12.** Call-by-name \(\lambda \mu \tilde{\text{p}}\) is observationally complete for closed normal forms: Calling a normal form canonical if it contains no subterms of the form \([\alpha]\lambda x.M\), we have that for any closed canonical normal forms \(M\) and \(N\) not equal in call-by-name \(\lambda \mu \tilde{\text{p}}\), there exists an evaluation context \(\Delta\), such that, in \(\lambda \mu \tilde{\text{p}}\), \(\mu_\theta .[\tilde{\text{p}}]M = \mu_\theta .[\tilde{\text{p}}]N\) for \(x\) and \(y\) arbitrary fresh variables.

Interestingly, this shows that if Böhm’s theorem in \(\Lambda \mu\) (Theorem 2) was apparently obtained by allowing more contexts (namely contexts of the form \([\alpha]M\)) which were not allowed in Parigot’s syntax, it is alternatively obtained by adding not only more contexts but by adding new rules that were hidden by the fact that \(\lambda \mu\) and \(\Lambda \mu\) apparently share the same rules.

One may wonder whether the equational theory of call-by-name \(\lambda \mu \tilde{\text{p}}\) is complete with respect to its CPS semantics. This is answered positively in Section 4.5.

### 4.4 Simple typing

We propose a system of simple types for call-by-name \(\lambda \mu \tilde{\text{p}}\). Like for typing \(\lambda \mu\), we have two kinds of sequents, one for each category of expressions:

\[\Gamma \vdash_\Sigma M : A ; \Delta \quad \text{(for terms)}\]
\[\Gamma \vdash_\Sigma c : \mathcal{L} ; \Delta \quad \text{(for commands)}\]

Like for \(\lambda \mu\), we have a context of hypotheses \(\Gamma\) that assigns types to term variables and a context of conclusions \(\Delta\) that assigns types to continuation variables. But we have also to take care of the \(\mu\tilde{\text{p}}\) dynamic binder.

Like for Ariola et al’s adaptation to call-by-value \(\lambda \mu \tilde{\text{p}}\) of Danvy and Filinski’s typing system in Section 3.3, we have an extra data to type the dynamic effects. Each use of \(\mu\tilde{\text{p}}\) pushes the current continuation on a stack of dynamically bound continuations. Each call to \(\tilde{\text{p}}\) pops the top continuation from this stack.

To the contrary of Ariola et al’s typing system, the extra information needed to type the dynamic binding is not a single formula.
but the ordered list $\Sigma$ of the types of the continuations present in
the stack.

Like for Ariola et al's typing system, functions can encapsulate occurrences of $\mathsf{tp}$ that may be called in a different typing context than the one that was active at the time $\mathsf{tp}$ was typed. For type consistency, rows have to remember the types of the dynamic continuation stack that calls the temps expect to see. We write $A_S \rightarrow B$ for an arrow annotated with the list $\Sigma$ of effect types.

To the contrary of Ariola et al's typing system, calls to $\mathsf{tp}$ are associated to terms and hence effects are assigned to the types of $\Gamma$ rather than to the types of $\Delta$. The typing system is given in Figure 12.

A very similar system of simple types has been given by Saurin (2007) on top of $\lambda\mu$. In $\lambda\mu$, judgements $\Gamma \vdash x : \cdot$; $\Delta$ are absent since there are no commands in the calculus. Judgements $\Gamma \vdash M : A ; \Delta$ are written $\Gamma, \Delta \vdash M : A \Rightarrow \Sigma$. Moreover, $\Gamma, \Delta \vdash M : A \Rightarrow \Sigma$. In addition, $\Gamma, \Delta \vdash M : A \Rightarrow \Sigma$ denotes that $\Gamma$ returns an object of type $B$ when applied to a linear evaluation context of type $A$ (a stream in Saurin's terminology). Logically, the type $A \Rightarrow B$ is equivalent to $\neg A \ightarrow B$, where the use of a negation emphasises that $\neg A$ is the type of an evaluation context expecting an argument of type $A$. Hence, $A \Rightarrow B$ is logically equivalent to a disjunction. Note that $\Rightarrow$ is a connective in Saurin's typing system, a conversion rule from $(A \Rightarrow \Sigma) \Rightarrow B \Rightarrow \Sigma$ to $(A \Rightarrow \Sigma) \Rightarrow (B \Rightarrow \Xi)$ is needed to type abstraction and application. This latter conversion rule has no computational content.

In order to prove subject reduction of the type system in Figure 12 we state two auxiliary lemmas (Generation and Substitution Lemma).

**Lemma 13 (Generation Lemma).** 1. $\Gamma, x : A_S \vdash x : B ; \Delta$ implies $\Xi \equiv \Sigma$ if $\Sigma$ and $\Xi$ are $\equiv$. 2. $\Gamma \vdash {\mathsf{lx}}.M : C ; \Delta$ implies $C \equiv A_S \rightarrow B$ and $\Gamma, x : A_S \vdash \Xi \equiv M : B ; \Delta$. 3. $\Gamma \vdash M N ; B ; \Delta$ implies $\Gamma \vdash M : A_S \rightarrow B ; \Delta$ and $\Gamma \vdash N : A ; \Delta$ for some $A_S$ and $\Sigma$. 4. $\Gamma \vdash \mu \alpha.c : A ; \Delta$ implies $\Gamma \vdash {\mathsf{c}} \equiv \Xi ; \Delta ; \alpha : A$. 5. $\Gamma \vdash [\alpha]M : A ; \Delta$ implies $\Xi \equiv \Delta, \alpha : A$ and $\Gamma \vdash {\mathsf{c}} \equiv M ; \Delta$. 6. $\Gamma \vdash {\mathsf{tp}}.c : A ; \Delta$ implies $\Gamma \vdash M \equiv c ;\Delta$.

**Lemma 14 (Substitution Lemma).** 1. Let $\Gamma, x : A_S \vdash M : B ; \Delta$ and $\Gamma \vdash N : A ; \Delta$. Then $\Gamma \vdash [\Xi \vdash M[N/x]] : B ; \Delta$. 2. Let $\Gamma \vdash c : \Xi ; \Delta, \alpha : A_S \rightarrow B$ and $\Gamma \vdash N : A ; \Delta$ and let $\beta$ be a fresh variable. Then $\Gamma \vdash \epsilon[\beta(\Xi[N]/\alpha)] \equiv \Delta, \beta : B$. 3. Let $\Gamma \vdash [\Xi \vdash c : \Xi ; \Delta, \alpha : A \Rightarrow B ; \Delta$ and $\alpha \vdash c \equiv \Xi, \Delta$. Then $\Gamma \vdash [\Xi \vdash c[\beta/\alpha]] \equiv \Delta, \beta : A$.

Subject reduction follows directly.

**Proposition 15 (Subject reduction).**

(i) If $\Gamma \vdash M : A ; \Delta$ and $M \Rightarrow N$, then $\Gamma \vdash N : A ; \Delta$.

(ii) If $\Gamma \vdash c : \Xi, \Delta$ and $c \equiv \Xi$, then $\Gamma \vdash [\Xi \vdash c : \Xi, \Delta$.

### 4.5 Continuation-passing-style semantics

De Groote (1994) defined a CPS transformation to $\lambda$-calculus for $\lambda\mu$. We give here an alternative CPS transformation that is based on a call-by-name CPS translation to $\lambda$-calculus with pairs (Lafont, Reus, and Streicher 1993). The $\lambda$-calculus with pairs is defined by the syntax:

$$M ::= x | \lambda x. M | M M | (M, M) | \mathsf{let} (y, y) = M in M$$

and we use $\lambda(x, y).t$ as an abbreviation for $\lambda x. \lambda y. t$. The following reduction rules:

$$\begin{align*}
\&: \mathsf{let} (x, y) = (M, N) in M' & M' & M' [N/y; M/x] \\
\&\&: \mathsf{let} (x, y) = M in N & \mathsf{let} (x, y) = M in F[N] \\
\&\&\&: \mathsf{let} (x, y) = \square in M & M
\end{align*}$$

for $F ::= \square N$ or $\mathsf{let} (x, y) = \square$.

We assume to have an injection $k_\alpha$ from continuation variables to term variables. The CPS transformation is shown in Figure 13. To the exception of some uses of $\eta$-conversion, it differs from de Groote’s transformation on $\lambda\mu$ only in the application and abstraction cases.

$$\begin{align*}
(\lambda x. M)^* & \equiv \chi x \\
(\lambda x. M)^* & \equiv \lambda x. M^* k \\
(M N)^* & \equiv \lambda x.M^* (N^* k) \\
(\mu \alpha.c)^* & \equiv \lambda \alpha.M^* k_\alpha \\
(\mu \mathsf{tp}.c)^* & \equiv c^* \\
([\mathsf{tp}]M)^* & \equiv M^*
\end{align*}$$

**Figure 13.** Call-by-name CPS translation of $\lambda\mu$
then, we get the following compatibility result:

**Proposition 16.**
(i) If \( \Gamma \vdash \Delta \) then \( \Gamma \vdash \Delta \).
(ii) If \( \Gamma \vdash \Sigma \vdash \Delta \) then \( \Gamma \vdash \Sigma \vdash \Delta \).

Unfortunately, the CPS above does not simulate the reduction. As it is common, we would have needed a CPS that takes care of administrative redex to get a simulation result. Still, the CPS above is compatible with equality in the \( \lambda \)-calculus with pairs:

**Proposition 17.** If \( M \to N \) then \( M^* = N^* \).

We can also state a completeness result (this is an adaptation of standard proofs, see e.g. de Groot 1994; Fujita 2003):

**Proposition 18.** If \( M^* = \beta \eta \mu \xi \eta \mu \xi \alpha \beta \gamma \delta \) then \( M = N \).

**Remark:** In the very same way as for call-by-value \( \lambda \mu \xi \) the call-by-name CPS translation can be factorised as the composition of a state-passing-style transformation to call-by-name \( \lambda \mu \xi \) extended with an asymmetric disjunction (because the type effects in call-by-name \( \lambda \mu \xi \) “naturally” take the form of an asymmetric disjunction; for asymmetric disjunction, see Pym and Ritter 2001), then of a call-by-name CPS translation to the \( \lambda \)-calculus with pairs.

### 4.6 Operational semantics

We first give the operational semantics of call-by-name \( \lambda \mu \xi \) as a set of reduction rules applicable to the term as a whole. This kind of operational semantics in “natural” style is defined on commands by the following rules:

\[
\begin{align*}
\beta : & \quad D_n(\alpha)[E_n][\lambda x.M][N] \\ \mu_n : & \quad D_n(\alpha)[E_n][\mu \beta \gamma \delta][c] \\ \eta_n : & \quad D_n(\alpha)[E_n][\mu \xi \eta \mu \xi \alpha \beta \gamma \delta]\] \\
\end{align*}
\]

As in the call-by-value case, \( \Rightarrow \) is included in \( \Rightarrow \) of which it constitutes on commands a level of abstraction. We say that \( c \) is a **weak-head normal command** if for no \( c', c \Rightarrow c' \). Weak-head normal commands are either of the form \([\mu \xi \eta \mu \xi \alpha \beta \gamma \delta]M\), or of the form \( D_n(\alpha)[E_n][x] \) or of the form \( D_n(\alpha)[\lambda x.M] \).

We then present the operational semantics by means of a call-by-name abstract machine. The language of the abstract machine for call-by-name is shown in Figure 14 and the reduction steps are given in Figure 15. As for the call-by-value machine in Section 3, the language of the machine is an extension with explicit environments of the language of \( \lambda \xi \). To initiate the computation, we need an extra condition of evaluation context that we write \( c \).

As in the call-by-value machine, the evaluation rules are split into three categories. However, the control is first owned by the evaluation context, so that the “logical” steps are controlled not by the value but by the linear evaluation context. Final result reconstruction in terminal states uses almost the same operations as for the call-by-value machine.

\[
\begin{align*}
[\alpha[e]] & \Rightarrow L \quad \text{if } c(\alpha) = L \\
[\alpha[e]] & \Rightarrow [\alpha][\square] \\
(M[e] \cdot L) & \Rightarrow L[\square M[e]] \\
[.] & \Rightarrow \square \\
[\mu \xi \eta \mu \xi \alpha \beta \gamma \delta] & \Rightarrow S[L[\mu \xi \eta \mu \xi \alpha \beta \gamma \delta]] \\
M(x) & \Rightarrow N[e] \\
M(\alpha) & \Rightarrow L[e] \\
\end{align*}
\]

**Proposition 19.** If \( c \) is a weak-head normal form, then \( [c]M \Rightarrow c \) if the evaluation starting from \( \langle M \mid [] \mid [] \rangle \) eval \( [] \) stops with result \( c \).

### 4.7 An example

How does call-by-name \( \lambda \xi \mu \xi \) behave on standard examples that uses delimited control? We consider the example of list traversal that Biermann and Danvy (2005) used to emphasise the differences between operator \( \mathcal{F} \) (Felleisen 1988) and shift. We extend \( \lambda \xi \mu \xi \) with a fixpoint operator, list constructors and a list destructor:

\[
M, N ::= \ldots \mid \nu_M.M \mid [] \mid M:N \mid \text{if } M \text{ is } x:y \text{ then } M \text{ else } M
\]

and we extend call-by-name reduction with the rules

\[
\begin{align*}
\nu_M.M & \Rightarrow M[\nu_x.M/x] \\
\nu_M.M & \Rightarrow M_1 \\
\nu_M.M & \Rightarrow M_2[M/x][N/y] \\
\nu_M.M & \Rightarrow M_3[\nu_c.M[\nu_x.M/x]/[c][x]] \\
\nu_M.M & \Rightarrow M_3[\nu_c.M[\nu_x.M/x]/[c][x]] \\
\end{align*}
\]

In informal ML syntax, the example is the following:

```ml
let rec visit l = match l with
| [] -> [] \\
| a::l' -> visit (\text{shift (fun k -> a :: k l'}}) \\
\end{align*}

in \text{reset (visit l)}

in \text{traverse \{1;2;3\}}

Translated into \( \Lambda \mu \xi \), it gives:

\[
\begin{align*}
v & \equiv \nu_M. M \text{ if } l \equiv a::l' \text{ then } f(\mu \xi. a::[a][l'] \text{ else } []) \\
\end{align*}
\]

Let \( \epsilon \) be an arbitrary continuation distinct from \( \xi \). We write \( li \) for \( n_1::n_2::n_3::[] \). We list the steps of the reduction of \( \epsilon[v l_1] \):

\[
\begin{align*}
\epsilon[v l_1] & \Rightarrow \epsilon(\nu_M. M \text{ if } l \equiv a::l' \text{ then } f(\mu \xi. a::[a][l'] \text{ else } [])) l_1 \\
\epsilon[v l_1] & \Rightarrow \epsilon(\text{if } l_1 \equiv a::l' \text{ then } f(\mu \xi. a::[a][l'] \text{ else } [])) v \\
\epsilon[v l_1] & \Rightarrow \epsilon(\nu_M. \text{if } l_2 \equiv a::l' \text{ then } f(\mu \xi. a::[a][l'] \text{ else } [])) l_2 \\
\epsilon[v l_1] & \Rightarrow \epsilon(\text{if } l_2 \equiv a::l' \text{ then } f(\mu \xi. a::[a][l'] \text{ else } [])) v \\
\epsilon[v l_1] & \Rightarrow \epsilon(\mu \xi. n_1::a::n_2::n_3::[a][l'] v) l_1 \\
\epsilon[v l_1] & \Rightarrow \epsilon(\mu \xi. n_1::a::n_2::n_3::[a][l'] v) l_2 \\
\epsilon[v l_1] & \Rightarrow \epsilon(\mu \xi. n_1::a::n_2::n_3::[a][l'] v) l_3 \\
\end{align*}
\]

Otherwise said, the list traversal program copies its argument and shifts its continuation to the tail of the list.

### 5. Discussion on a General Framework for Calculi of Delimited Continuations

We review below two variants of the original calculus with shift and reset. Together with \( \Lambda \mu \xi \), we then obtain four calculi of delimited continuations. We show how these four calculi are related.

**Lazy reset** A variant of call-by-value \( \lambda \xi \) can be obtained by considering that terms of the form \( \mu \xi \) are values. In this case, one obtains a calculus equivalent to the \( \lambda \) and shift and lazy reset, a calculus for which Sabry gave an axiomisation complete with respect to its CPS semantics (Sabry 1996).
Call-by-name shift studied in the paper in that the rules in call-by-name discipline. Expressed in the language of $\lambda\mu$ machine for shift $\lambda$ of call-by-value $\lambda\mu$ for incremental substitution of the new kind of context $e$. Choosing call-by-value requires delimited continuations are classified in Figure 16.

Fundamental dilemma of computation (e.g., in Curien and Herbelin 2000). Choosing call-by-value requires to restrict $\beta$-reduction into $\beta_{\eta}$-reduction and to add a rule $\mu_{\eta_{\alpha}}$ for incremental substitution of the new kind of context $(\lambda x.M) \Box$. In each variant, a subsidiary choice has to be made to decide if $\mu_{\alpha}c$ is a value or not and if $\mu_{\eta_{\alpha}}c$ behaves like a linear continuation variable or not.

In call-by-value, the extra critical pair is $(\lambda x.t) (\mu_{\eta_{\alpha}}c)$. If $\mu_{\eta_{\alpha}}c$ is considered as non evaluated, the call-by-value discipline expects that priority is given to it and one obtains the original shift and reset calculus from Danvy and Filinski. If otherwise $\mu_{\eta_{\alpha}}c$ is considered as evaluated, it yields its priority to its evaluation context, i.e. to the function, and $\beta$ is applicable. One then obtains the calculus with lazy reset that was studied in Sabry (1996).

In call-by-name, the extra critical pair is $(\mu_{\alpha}c)[\mu_{\eta_{\alpha}}c]$. If priority is given to the evaluation context, i.e. $\mu_{\eta_{\alpha}}c$, one has first to know to what it is bound before to continue the computation. One then obtains Danvy’s call-by-name variant of the shift and reset calculus.

### Conclusions

**Summary**

We showed that de Groote variant of $\lambda\mu$-calculus, here called $\Lambda_{\mu}$ after Saurin, while apparently similar to Parigot’s $\lambda_{\mu}$, can be interpreted as an extension of $\lambda_{\mu}$ with call-by-name delimited control. Especially, we showed the following points:

- The four calculi of delimited continuations are classified in Figure 16.
- Choosing between call-by-name and call-by-value amounts to decide the fundamental dilemma of computation (as emphasised, e.g., in Curien and Herbelin 2000). Choosing call-by-value requires to restrict $\beta$-reduction into $\beta_{\eta}$-reduction and to add a rule $\mu_{\eta_{\alpha}}$ for incremental substitution of the new kind of context $(\lambda x.M) \Box$.
Fundamental critical pair of computation
\((\lambda x.t) (\mu \alpha.c)\)

\[ (\beta_e) + (\mu_{app}) + (\mu_{app}) + (\mu_{var}) + (\eta_\mu) \]

(CBV)

\((\mu_\alpha.c)\) shifted/reset

(CBN)

\[ (\beta) + (\mu_{app}) + (\mu_{var}) + (\eta_\mu) \]

\[ (\mu_{var} \text{ moved to } \mu_{app}) \]

(subsidiary choice)

\[ (\mu_\alpha.c) \]

(shift/reset)

\[ (\eta_\mu \text{ moved to } \eta_\mu') \]

(subsidiary choice)

\[ \Lambda \mu \]

\[ \Lambda \mu \]

Figure 16. Calculi of delimited continuations - a classification

- \(\Lambda \mu\) can be interpreted as a call-by-name variant of Ariola et al’s extension of call-by-value \(\lambda \mu\) with delimited control, namely call-by-value \(\lambda \mu tp\).
- The abstract machine for \(\Lambda \mu\) relies on a global stack for the dynamic continuation as the abstract machine for call-by-name \(\lambda \mu tp\) does.
- There is a system of simple types with effects for \(\Lambda \mu\) for which subject reduction holds.

The \(\Lambda \mu\) is a surprising calculus. On one side, its syntax and CPS semantics are very simple, and in particular simpler than the syntax and CPS semantics of call-by-value calculus of delimited continuations. On the other side, its “canonical” system of types and its operational semantics keep the complexity of a calculus of delimited control. The absence of an explicit control delimiter in \(\Lambda \mu\) is at first glance surprising, but if we admit that the definition of \((M)\) is \(\mu \text{tp} \cdot [\text{tp}] M\) as it is in call-by-value \(\lambda \mu \text{tp}\), then it is normal that no explicit \((M)\) is needed in \(\Lambda \mu\) since it collapses in call-by-name \(\lambda \mu \text{tp}\) to an identity operator. Another lesson is that the \(\mu\) operator of \(\Lambda \mu\) is indeed a shift operator\(^6\).

One could ask whether the syntax of \(\Lambda \mu\) can be used for call-by-value delimited control. The answer is yes if one adds an explicit \((M)\). Indeed, in \(\Lambda \mu\) extended with \((M)\), the four combinations \(\mu \alpha \cdot [\beta] M, \mu \alpha \cdot [\beta] M, \mu \alpha \cdot [\beta] M, \mu \alpha \cdot [\beta] M, [\alpha] M\) and \(\mu \text{tp} \cdot [\text{tp}] M\) are equivalently expressible in \(\Lambda \mu\) by \(\mu \alpha \cdot [\beta] M, \mu \alpha \cdot [\beta] M, [\alpha] M\) and \((M)\) respectively.

Section 5 showed that \(\Lambda \mu\) is not the only call-by-name delimited control. Further investigations into the four different calculi need to be done to better understand the relative strengths of each of the calculi.

The separability property in classical logic

The \(\lambda \mu \mu\)-calculus is the calculus of choice to study the kind of duality given in Figure 16. Uniformly investigating the completeness properties of the four calculi and completing the picture in the framework of \(\lambda \mu \mu\)-calculus would be interesting. Up to our knowledge, there are no results on Böhm’s separability property in other proof calculi for classical logic. We believe that the separability property for call-by-name \(\lambda \mu \text{tp}\) would directly transfer to call-by-name untyped \(\lambda \mu \mu\)-calculus but Böhm’s separability property in the untyped call-by-value and in the typed versions of \(\lambda \mu \mu\)-calculus are open problems. The question of separability in the Dual Calculus Wadler (2003) is a topic for future research, as well.

\(^6\) In passing, this suggests that de Groote’s use of \(\Lambda \mu\) for representing quantifier scope in linguistic is not so far from the shift/reset-based approach of quantifier scope by Barker and Shan.

An other question is also the investigation of Böhm’s theorem in the simply typed fragments of the four calculi (see e.g. Böhm’s theorem in the simply typed \(\lambda\)-calculus by Došen and Petrić (2001), Statman (1982), Simpson (1995), Joly (2000)).

Finally, how far the study of Böhm’s theorem in call-by-value calculus with control can help for investigating separation in Moggi’s extension of Plotkin’s \(\lambda_e\) (see Paolini 2001).

Acknowledgements We wish to thank Olivier Danvy and Alexis Saurin for fruitful discussions we had during the work on this paper.

References


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


