A Computational Interpretation of the $\lambda\mu$-calculus

G.M. Bierman

University of Cambridge

Abstract. This paper proposes a simple computational interpretation of Parigot's $\lambda\mu$-calculus. The $\lambda\mu$-calculus is an extension of the typed λ-calculus which corresponds via the Curry-Howard correspondence to classical logic. Whereas other work has given computational interpretations by translating the $\lambda\mu$-calculus into other calculi, I wish to propose here a direct computational interpretation. This interpretation is best given as a single-step semantics which, in particular, leads to a relatively simple, but powerful, operational theory.

1 Introduction

It is well-known that the typed λ-calculus can be viewed as a term assignment for natural deduction proofs in intuitionistic logic (**IL**). Consequently the set of types of all closed λ-terms enumerates all intuitionistic tautologies. This is known as the Curry-Howard correspondence, or the formulae-as-types principle. Thus one can talk of a computational interpretation of **IL**. A natural question is whether there is such a computational interpretation of classical logic (**CL**). A first step is to devise a well behaved natural deduction formulation for **CL** and give a term assignment. A number of proposals have been made but recently Parigot [9] introduced a extension of the typed λ-calculus, which he called the $\lambda\mu$-calculus. The set of types of all closed $\lambda\mu$-terms enumerates all classical tautologies and the calculus is particularly well behaved, satisfying both strong normalisation and confluence.

However two questions remain. First, what does the extension to the $\lambda\mu$-calculus mean computationally? Secondly, if the $\lambda\mu$-calculus is extended in much the same way as the λ-calculus is extended to yield PCF, what is its operational theory? Of course the answer to the second question is heavily dependent upon the answer to the first. In this paper I suggest that the $\lambda\mu$-calculus has a natural computational reading: it is a λ-calculus which is able to manipulate the runtime environment via indexed catch and throw operators. This can easily be expressed using evaluation contexts which are common in work on control operators.

Morris-style contextual equivalence is commonly accepted as the natural notion of equivalence for functional languages. There has been significant effort in devising alternative characterisations of contextual equivalence which are more amenable for constructing proofs. For PCF the common solution is to use some form of (applicative) bisimilarity [3]. However these techniques do not often extend to languages with control. In §6 I give a simple notion of program equivalence, based on transitions in an abstract machine, which coincides with contextual equivalence.

2 Parigot's $\lambda\mu$-calculus

In his seminal paper Parigot introduced an extension of the typed λ-calculus, which he called the $\lambda\mu$-calculus. The extension is such that terms no longer have a single type but a *sequence* of types, one of which is said to be the active type and the rest which are said to be passive. I shall not go into great detail here—the reader is referred to any one of a number of good introductions [1, 7, 8].

Types are given by the grammar $\phi ::= \bot \mid \phi \to \phi$, and raw $\lambda\mu$-terms are given by

$$\begin{array}{rcll} M & ::= & x & \text{Variable} \\ & \mid & \lambda x{:}\,\phi.M & \text{Abstraction} \\ & \mid & MM & \text{Application} \\ & \mid & [a{:}\,\phi]M & \text{Passivation} \\ & \mid & \mu a{:}\,\phi.M & \text{Activation;} \end{array}$$

where x is taken from a countable set of λ-variables, ϕ is a well-formed type (formula) and a is taken from a countable set of μ-variables.

Typing judgements are of the form, $\Gamma \triangleright M{:}\,\phi, \Sigma$, where Γ is a set of pairs of λ-variables and types written $x{:}\,\psi$, M is a term from the above grammar and Σ denotes a set of pairs of μ-variables and types written $a{:}\,\varphi$ (thus ϕ is the active type). The typing rules are as follows.

$$\frac{}{\Gamma, x{:}\,\phi \triangleright x{:}\,\phi, \Sigma}\ \text{Identity}$$

$$\frac{\Gamma, x{:}\,\phi \triangleright M{:}\,\psi, \Sigma}{\Gamma \triangleright \lambda x{:}\,\phi.M{:}\,\phi \to \psi, \Sigma}\to_{\mathcal{I}} \qquad \frac{\Gamma \triangleright M{:}\,\phi \to \psi, \Sigma \qquad \Gamma \triangleright N{:}\,\phi, \Sigma}{\Gamma \triangleright MN{:}\,\psi, \Sigma}\to_{\mathcal{E}}$$

$$\frac{\Gamma \triangleright M{:}\,\phi, \Sigma}{\Gamma \triangleright [a{:}\,\phi]M{:}\,\bot, a{:}\,\phi, \Sigma}\ \text{Passivate} \qquad \frac{\Gamma \triangleright M{:}\,\bot, a{:}\,\phi, \Sigma}{\Gamma \triangleright \mu a{:}\,\phi.M{:}\,\phi, \Sigma}\ \text{Activate}$$

The new rules are the $\mathrm{Passivate}$ and $\mathrm{Activate}$. The former takes a term whose active type is ϕ (where ϕ is not $\bot$) and passivates it, i.e. ϕ becomes a passive type (and is hence labelled with a). The resulting term has an active type of $\bot$.[1] The $\mathrm{Activate}$ rule works similarly but in the reverse direction.

There are a number of reduction rules associated with the $\lambda\mu$-calculus. In full they are as follows.

$$\begin{array}{rcll} (\lambda x{:}\,\phi.M)N & \rightsquigarrow_\beta & M[x := N] & \\ \mu a{:}\,\phi.[a{:}\,\phi]M & \rightsquigarrow_\beta & M & \text{where } a \notin \mu\mathrm{FV}(M) \\ (\mu a{:}\,\phi \to \psi.M)N & \rightsquigarrow_c & \mu a{:}\,\psi.M[[a{:}\,\phi \to \psi]P \Leftarrow [a{:}\,\psi]PN] & \\ \lambda x.Mx & \rightsquigarrow_\eta & M & \text{where } x \notin \mathrm{FV}(M) \\ [a{:}\,\phi]\mu b{:}\,\phi.M & \rightsquigarrow_\eta & M[a/b] & \end{array}$$

In the second β-rule, $\mu\mathrm{FV}(M)$ denotes the set of free μ-variables in the term M (I shall omit its rather obvious definition). In the commuting conversion ($\rightsquigarrow_c$) I have used the

[1] This ensures that every term has an active type. It is possible to give a formulation where terms need not have an active type.

notation $M[N \Leftarrow P]$ to denote the term M where *all* occurrences of the subterm N have been replaced by the term P. In the last η-rule, $M[a/b]$ denotes the term M where all free occurrences of the μ-variable b are replaced with a. All forms of substitution are assumed to be non-capturing.

3 A Computational Interpretation

As it stands it is unclear what this move to **CL** has given us—clearly we have terms at new types and new terms at old types, but what does this mean computationally? In order to find an answer I shall consider the operational behaviour of the calculus, namely the execution of closed terms (programs) to canonical values.

Before presenting the operational behaviour I need first to introduce some standard terminology from work on control operators, e.g. [2]. To formalise the notion of an evaluation order, Felleisen [*op. cit.*], defined an *evaluation context*. This is essentially a term with a single 'hole' in it, written $E[\bullet]$ (this will be defined formally in the next section). The result of placing a term, M, in that hole is written $E[M]$. Evaluation contexts are devised so that every closed term, M, is either a canonical value or can be written *uniquely* as $E[R]$, where R is a redex. The context $E[\bullet]$ can be thought of as representing the rest of the computation that remains to be done after R has been reduced. In this sense it can be seen as the *continuation* of R or, more simply, the *current continuation*.

Evaluation is then written as $(E[R], \mathcal{E}) \Rightarrow (M', \mathcal{E}')$, where $\mathcal{E}$ is a function from μ-variables to evaluation contexts—the need for this will become clear. The important evaluation rules are

$$(E[\mu a.M], \mathcal{E}) \Rightarrow (M, \mathcal{E} \uplus (a \mapsto E[\bullet]))$$
$$(E[[a]M], \mathcal{E} \uplus (a \mapsto E'[\bullet])) \Rightarrow (E'[M], \mathcal{E} \uplus (a \mapsto E'[\bullet]));$$

where $\mathcal{E} \uplus (a \mapsto E[\bullet])$ denotes the extension of the function $\mathcal{E}$ with the mapping $a \mapsto E[\bullet]$. Thus in the first reduction rule the current continuation is captured ('catch'), added to $\mathcal{E}$ and indexed with a. In the second reduction rule the appropriate indexed continuation is taken from $\mathcal{E}$, replacing the current continuation (i.e. the term M is 'thrown back' to an earlier continuation). In summary, the $\mathrm{Activate}$ and $\mathrm{Passivate}$ rules are interpreted as indexed catch and throw operators, respectively.

4 μPCF

Rather than develop an operational theory for the $\lambda\mu$-calculus, I shall first enrich it with natural numbers, a conditional, pairs and recursion. This is essentially what Ong and Stewart call μPCF [8]. The next step is to choose an evaluation strategy. Most work on control operators has considered a *call-by-value* strategy and to aid comparison I shall adopt the same. It is important to note that what is developed in this section can easily be adjusted to reflect a call-by-name strategy; some details are sketched in §7. This is in contrast with Ong and Stewart's framework, which requires significant changes to move from call-by-name to call-by-value (some details are in their paper [8]). For

completeness the typing rules for the new constructors are given below.

$$\frac{}{\Gamma \triangleright \underline{n} : \mathtt{int}, \Sigma} \qquad \frac{\Gamma \triangleright M : \mathtt{int}, \Sigma}{\Gamma \triangleright \mathsf{suc}(M) : \mathtt{int}, \Sigma} \qquad \frac{\Gamma, f : \phi \to \phi, x : \phi \triangleright M : \phi, \Sigma \quad \Gamma, f : \phi \to \phi \triangleright N : \psi, \Sigma}{\Gamma \triangleright \mathsf{letrec}\ f = \lambda x.M\ \mathsf{in}\ N : \psi, \Sigma}$$

$$\frac{\Gamma \triangleright M : \mathtt{int}, \Sigma \qquad \Gamma \triangleright N : \phi, \Sigma \qquad \Gamma \triangleright P : \phi, \Sigma}{\Gamma \triangleright \mathsf{ifz}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P : \phi, \Sigma}$$

$$\frac{\Gamma \triangleright M : \phi, \Sigma \qquad \Gamma \triangleright N : \psi, \Sigma}{\Gamma \triangleright \langle M, N \rangle : \phi \times \psi, \Sigma} \qquad \frac{\Gamma \triangleright M : \phi \times \psi, \Sigma}{\Gamma \triangleright \mathsf{fst}(M) : \phi, \Sigma} \qquad \frac{\Gamma \triangleright M : \phi \times \psi, \Sigma}{\Gamma \triangleright \mathsf{snd}(M) : \psi, \Sigma}$$

The syntactic classes of values, evaluation contexts and redexes are defined as follows.

$$\begin{array}{lll}
\text{Values} & v & ::= \underline{n} \mid \lambda x.M \mid \langle v, v \rangle \\
\text{Evaluation Contexts} & E & ::= \bullet \mid vE \mid EM \\
& & \quad \mid \langle E, M \rangle \mid \langle v, E \rangle \mid \mathsf{fst}(E) \mid \mathsf{snd}(E) \\
& & \quad \mid \mathsf{suc}(E) \mid \mathsf{ifz}\ E\ \mathsf{then}\ M\ \mathsf{else}\ M \\
\text{Redexes} & R & ::= vv \mid \mathsf{fst}(v) \mid \mathsf{snd}(v) \\
& & \quad \mid \mathsf{suc}(v) \mid \mathsf{ifz}\ v\ \mathsf{then}\ M\ \mathsf{else}\ M \\
& & \quad \mid \mathsf{letrec}\ f = \lambda x.M\ \mathsf{in}\ N \mid [a]M \mid \mu a.M
\end{array}$$

The fundamental property of evaluation contexts is the following.

Lemma 1. *Every closed term, M, is either a value, v, or is uniquely of the form $E[R]$, where $E[\bullet]$ is an evaluation context and R is a redex.*

We can now write out the (single-step) reduction rules in full, which are as follows.

$$\begin{array}{rcl}
(E[(\lambda x.M)v], \mathcal{E}) & \Rightarrow & (E[M[x := v]], \mathcal{E}) \\
(E[\mathsf{fst}(\langle v, w \rangle)], \mathcal{E}) & \Rightarrow & (E[v], \mathcal{E}) \\
(E[\mathsf{snd}(\langle v, w \rangle)], \mathcal{E}) & \Rightarrow & (E[w], \mathcal{E}) \\
(E[\mathsf{suc}(\underline{n})], \mathcal{E}) & \Rightarrow & (E[\underline{n+1}], \mathcal{E}) \\
(E[\mathsf{ifz}\ \underline{0}\ \mathsf{then}\ M\ \mathsf{else}\ N], \mathcal{E}) & \Rightarrow & (E[M], \mathcal{E}) \\
(E[\mathsf{ifz}\ (\underline{n+1})\ \mathsf{then}\ M\ \mathsf{else}\ N], \mathcal{E}) & \Rightarrow & (E[N], \mathcal{E}) \\
(E[\mathsf{letrec}\ f = \lambda x.M\ \mathsf{in}\ N], \mathcal{E}) & \Rightarrow & (E[N[f := \lambda x.\mathsf{letrec}\ f = \lambda x.M\ \mathsf{in}\ M]], \mathcal{E}) \\
(E[\mu a.M], \mathcal{E}) & \Rightarrow & (M, \mathcal{E} \uplus (a \mapsto E[\bullet])) \\
(E[[a]M], \mathcal{E} \uplus (a \mapsto E'[\bullet])) & \Rightarrow & (E'[M], \mathcal{E} \uplus (a \mapsto E'[\bullet]))
\end{array}$$

5 Examples

To demonstrate the expressive power of this computational interpretation I shall show the dynamics of particular ML-like exception handling and 'callcc' primitives are preserved by their encodings into μPCF (the encodings are due to Ong and Stewart [8]).

5.1 Exception Handling

ML can be extended with exceptions in a number of ways. One such method was given by Gunter *et al.* [5] and simplified by Ong and Stewart [8]. Typed exceptions are identified with names, thus typing judgements (for ML) are now of the form $\Gamma; \Delta \triangleright M : \phi$ where Γ is the usual typing environment and Δ is the typing environment for the exception names. Two new operators are added to ML whose typing rules are as follows.

$$\frac{\Gamma; \Delta \triangleright M : A}{\Gamma; \Delta, a : A \triangleright \mathit{raise}(a, M) : B} \qquad \frac{\Gamma; \Delta, a : A \triangleright M : A \to B \qquad \Gamma; \Delta, a : A \triangleright N : B}{\Gamma; \Delta \triangleright \mathit{handle}(a, M, N) : B}$$

The intended interpretation is that the first rule evaluates M to a value v and then raises an exception named a associated with v. The second rule evaluates M to a value (say v) and then evaluates N. If N evaluates to a value w then this is the overall result, but if it raises an exception named a with a value u, then this is applied to v. Given as reduction rules the intended interpretation is as follows.

$$\begin{array}{rl} \mathit{handle}(a, v, w) \rightsquigarrow w & (a \notin \mathrm{FN}(w)) \\ \mathit{handle}(a, v, E[\mathit{raise}(a, u)]) \rightsquigarrow vu & (a \notin \mathrm{FN}(v, u)) \end{array}$$

These operators can be translated into μPCF as follows (where b is a fresh μ-variable).

$$\begin{array}{c} [\![\mathit{raise}(a, M)]\!] \stackrel{\mathrm{def}}{=} (\lambda x.\mu b.[a]x)[\![M]\!] \\ [\![\mathit{handle}(a, M, N)]\!] \stackrel{\mathrm{def}}{=} \mu b.[b][\![M]\!](\mu a.[b][\![N]\!]) \end{array}$$

It is relatively easy to show that this translation preserves the operational behaviour, e.g.

$$\begin{array}{ll} & ([\![\mathit{handle}(a, M, E[\mathit{raise}(a, N)])]\!], \mathcal{E}) \\ \stackrel{\mathrm{def}}{=} & (\mu b.[b][\![M]\!](\mu a.[b]E[(\lambda x.\mu c.[a]x)[\![N]\!]]), \mathcal{E}) \\ \Rightarrow^2 & ([\![M]\!](\mu a.[b]E[(\lambda x.\mu c.[a]x)[\![N]\!]]), \mathcal{E} \uplus \{b \mapsto \bullet\}) \\ \Rightarrow^* & (v(\mu a.[b]E[(\lambda x.\mu c.[a]x)[\![N]\!]]), \mathcal{E} \uplus \{b \mapsto \bullet\}) \\ \Rightarrow & ([b]E[(\lambda x.\mu c.[a]x)[\![N]\!]], \mathcal{E} \uplus \{a \mapsto (v\bullet), b \mapsto \bullet\}) \\ \Rightarrow & (E[(\lambda x.\mu c.[a]x)[\![N]\!]], \mathcal{E} \uplus \{a \mapsto (v\bullet), b \mapsto \bullet\}) \\ \Rightarrow^+ & (E[\mu c.[a]u)], \mathcal{E} \uplus \{a \mapsto (v\bullet), b \mapsto \bullet\}) \\ \Rightarrow & ([a]u, \mathcal{E} \uplus \{a \mapsto (v\bullet), b \mapsto \bullet, c \mapsto E[\bullet]\}) \\ \Rightarrow & (vu, \mathcal{E} \uplus \{a \mapsto (v\bullet), b \mapsto \bullet, c \mapsto E[\bullet]\}) \end{array}$$

5.2 Call-with-current-continuation (callcc)

ML can be extended with operators to manipulate first-class continuations in a number of ways. I shall consider a proposal again due to Gunter *et al.* [5] and simplified by Ong and Stewart [8]. Here (typed) continuations are associated with names, and so typing judgements are of the form $\Gamma; \Delta \triangleright M : A$, where Δ is the typing environment for continuation names. Three new operators are added to ML, whose typing rules are as follows.

$$\frac{\Gamma; \Delta \triangleright M : (A \to B) \to A}{\Gamma; \Delta \triangleright \mathit{callcc}(M) : A} \qquad \frac{\Gamma; \Delta \triangleright M : A}{\Gamma; \Delta, a : A \triangleright \mathit{abort}(a, M) : B} \qquad \frac{\Gamma; \Delta, a : A \triangleright M : A}{\Gamma; \Delta \triangleright \mathit{set}(a, M) : A}$$

The $callcc$ operator applies the term M to an abstraction of the current continuation. The set serves as a delimiter for continuations, and the $abort$ discards the current continuation (delimited by a). Their intended operational behaviour is as follows.

$$\begin{array}{rll} set(a, E[abort(a, M)]) & \rightsquigarrow M & (a \notin \text{FN}(M)) \\ set(a, v) & \rightsquigarrow v & (a \notin \text{FN}(v)) \\ E[callcc(M)] & \rightsquigarrow set(a, E[M(\lambda x.abort(a, E[x]))]) & \end{array}$$

Ong and Stewart provided a translation of these operators into μPCF, which is as follows.

$$\begin{array}{rl} [\![callcc(M)]\!] & \stackrel{\text{def}}{=} \mu a.[a]([\![M]\!](\lambda x.\mu b.[a]x)) \\ [\![abort(a, M)]\!] & \stackrel{\text{def}}{=} \mu b.[a][\![M]\!] \qquad \text{where } b \notin \mu\text{FV}([\![M]\!]) \\ [\![set(a, M)]\!] & \stackrel{\text{def}}{=} \mu a.[a][\![M]\!] \end{array}$$

Again it is simple to check that this translation preserves the operational behaviour, e.g.

$$\begin{array}{rl} & ([\![set(a, E[abort(a, M)])]\!], \mathcal{E}) \\ \stackrel{\text{def}}{=} & (\mu a.[a]E[\mu b.[a][\![M]\!]], \mathcal{E}) \\ \Rightarrow^2 & (E[\mu b.[a][\![M]\!]], \mathcal{E} \uplus \{a \mapsto \bullet\}) \\ \Rightarrow & ([a][\![M]\!], \mathcal{E} \uplus \{a \mapsto \bullet, b \mapsto E[\bullet]\}) \\ \Rightarrow & ([\![M]\!], \mathcal{E} \uplus \{a \mapsto \bullet, b \mapsto E[\bullet]\}) \end{array}$$

5.3 Pairing

It is easy to verify that $\phi \times \psi \equiv \neg(\phi \to \neg\psi)$ in **CL**. This logical equivalence can be used to simulate pairing in μPCF. The constructor and deconstructors are encoded as follows.[2]

$$\begin{array}{rl} pair & \stackrel{\text{def}}{=} \lambda m{:}\,\phi.\lambda n{:}\,\psi.\lambda f{:}\,(\phi \to (\psi \to \bot)).f\ m\ n \\ fst & \stackrel{\text{def}}{=} \lambda p.\mu a.p(\lambda x.\mu b.[a]x) \\ snd & \stackrel{\text{def}}{=} \lambda p.\mu a.p(\lambda y.\lambda x.[a]x) \end{array}$$

It is left to the reader to verify that these encodings satisfy the expected operational behaviour.

6 Operational Theory

An implementation based on the reduction rules given in §4 would work as follows. Take a term M: if it is a value then we are done; if not it can be given uniquely as $E[R]$. One takes the relevant reduction step (determined by R)—the resulting term is either a value, in which case we are done, or it has to be re-written again as an evaluation context and a redex. This process is repeated until a value is reached. The continual intermediate step of rewriting a term into an evaluation context and a redex would be inefficient in practice and is quite cumbersome theoretically. Consequently I shall give a new set of reduction rules where the context and the redex are actually separated.

[2] A similar encoding using control operators was given by Griffin [4].

Reduction rules are now of the form $(S, M, \mathcal{E}) \longrightarrow (S', M', \mathcal{E}')$, where S is a stack of *evaluation frames*, which are defined as follows.

$$\begin{array}{rl} F ::= & \bullet M \mid v \bullet \mid \langle \bullet, M \rangle \mid \langle v, \bullet \rangle \\ & \mid \ \mathsf{fst}(\bullet) \mid \mathsf{snd}(\bullet) \mid \mathsf{suc}(\bullet) \mid \mathsf{ifz}\ \bullet\ \mathsf{then}\ M\ \mathsf{else}\ M \end{array}$$

(Clearly $\mathcal{E}$ is now a function from μ-variables to stacks.) The reduction rules essentially describe the transitions of a simple abstract machine.[3] In full they are as follows.

$$\begin{array}{rcll}
(F[\bullet] :: S, v, \mathcal{E}) & \longrightarrow & (S, F[v], \mathcal{E}) & \\
(S, MN, \mathcal{E}) & \longrightarrow & ((\bullet N) :: S, M, \mathcal{E}) & M \text{ not a value} \\
(S, vN, \mathcal{E}) & \longrightarrow & ((v\bullet) :: S, N, \mathcal{E}) & N \text{ not a value} \\
(S, (\lambda x.M)v, \mathcal{E}) & \longrightarrow & (S, M[x := v], \mathcal{E}) & \\
(S, \langle M, N \rangle, \mathcal{E}) & \longrightarrow & (\langle \bullet, N \rangle :: S, M, \mathcal{E}) & M \text{ not a value} \\
(S, \langle v, N \rangle, \mathcal{E}) & \longrightarrow & (\langle v, \bullet \rangle :: S, N, \mathcal{E}) & N \text{ not a value} \\
(S, \mathsf{fst}(M), \mathcal{E}) & \longrightarrow & (\mathsf{fst}(\bullet) :: S, M, \mathcal{E}) & M \text{ not a value} \\
(S, \mathsf{fst}(\langle v, w \rangle), \mathcal{E}) & \longrightarrow & (S, v, \mathcal{E}) & \\
(S, \mathsf{snd}(M), \mathcal{E}) & \longrightarrow & (\mathsf{snd}(\bullet) :: S, M, \mathcal{E}) & M \text{ not a value} \\
(S, \mathsf{snd}(\langle v, w \rangle), \mathcal{E}) & \longrightarrow & (S, w, \mathcal{E}) & \\
(S, \mathsf{suc}(M), \mathcal{E}) & \longrightarrow & (\mathsf{suc}(\bullet) :: S, M, \mathcal{E}) & M \text{ not a value} \\
(S, \mathsf{suc}(\underline{n}), \mathcal{E}) & \longrightarrow & (S, \underline{n+1}, \mathcal{E}) & \\
(S, \mathsf{ifz}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P, \mathcal{E}) & \longrightarrow & ((\mathsf{ifz}\ \bullet\ \mathsf{then}\ N\ \mathsf{else}\ P) :: S, M, \mathcal{E}) & M \text{ not a value} \\
(S, \mathsf{ifz}\ \underline{0}\ \mathsf{then}\ M\ \mathsf{else}\ N, \mathcal{E}) & \longrightarrow & (S, M, \mathcal{E}) & \\
(S, \mathsf{ifz}\ (\underline{n+1})\ \mathsf{then}\ M\ \mathsf{else}\ N, \mathcal{E}) & \longrightarrow & (S, N, \mathcal{E}) & \\
(S, \mathsf{letrec}\ f = \lambda x.M\ \mathsf{in}\ N, \mathcal{E}) & \longrightarrow & (S, N[f := \lambda x.\mathsf{letrec}\ f = \lambda x.M\ \mathsf{in}\ M], \mathcal{E}) & \\
(S, \mu a.M, \mathcal{E}) & \longrightarrow & ([], M, \mathcal{E} \uplus (a \mapsto S)) & \\
(S, [a]M, \mathcal{E} \uplus (a \mapsto T)) & \longrightarrow & (T, M, \mathcal{E} \uplus (a \mapsto T)) &
\end{array}$$

An example may make these reduction rules clearer. Consider an instance of the 'callcc' reduction rule given in §5.2.

$$set(a, (\lambda x.N)(abort(a, M))) \rightsquigarrow M$$

The left hand term is translated to the μPCF-term $\mu a.[a](\lambda x.[\![N]\!])(\mu b.[a][\![M]\!])$, which reduces as follows.

$$\begin{array}{lll}
(S, & \mu a.[a](\lambda x.[\![N]\!])(\mu b.[a][\![M]\!]), & \mathcal{E}) \\
\longrightarrow ([], & [a](\lambda x.[\![N]\!])(\mu b.[a][\![M]\!]), & \mathcal{E} \uplus \{a \mapsto S\}) \\
\longrightarrow (S, & (\lambda x.[\![N]\!])(\mu b.[a][\![M]\!]), & \mathcal{E} \uplus \{a \mapsto S\}) \\
\longrightarrow (((\lambda x.[\![N]\!])\bullet) :: S, & \mu b.[a][\![M]\!], & \mathcal{E} \uplus \{a \mapsto S\}) \\
\longrightarrow ([], & [a][\![M]\!], & \mathcal{E} \uplus \{a \mapsto S, b \mapsto ((\lambda x.[\![N]\!])\bullet) :: S\}) \\
\longrightarrow (S, & [\![M]\!], & \mathcal{E} \uplus \{a \mapsto S, b \mapsto ((\lambda x.[\![N]\!])\bullet) :: S\})
\end{array}$$

It is easy to define a function $\ulcorner E \urcorner$ which converts a given evaluation context, E to a stack of frames, and a function $S@M$ which takes a stack of frames, S, and a term, M, and converts the stack back to an evaluation context before inserting M. For example

$$\begin{array}{rcl}
\ulcorner(((\lambda x.M)\bullet)P)Q\urcorner & \stackrel{\text{def}}{=} & ((\lambda x.M)\bullet) :: ((\bullet P) :: ((\bullet Q) :: [])) \\
((\lambda x.M)\bullet) :: ((\bullet P) :: ((\bullet Q) :: []))@N & \stackrel{\text{def}}{=} & (((\lambda x.M)N)P)Q
\end{array}$$

The two sets of reduction rules can be related in the following sense.

[3] Harper and Stone [6] give similar transition rules in their analysis of SML and Pitts [10] has used similar rules in work on functional languages with dynamic allocation of store.

Proposition 1 $(S@M, \mathcal{E}) \Rightarrow (N, \mathcal{E}')$ *iff* $\exists S', M'.N = S'@M', (S, M, \lceil\, \mathcal{E}\rceil) \longrightarrow^* (S', M', \lceil\, \mathcal{E}'\rceil)$

An important fact (first discovered by Pitts [10] in a different setting) is that the set

$$\searrow \stackrel{\text{def}}{=} \{(S, M, \mathcal{E}) \mid \exists v, \mathcal{E}'.(S, M, \mathcal{E}) \longrightarrow^* ([], v, \mathcal{E}')\}$$

has a direct, inductive definition which is as follows.

$$([], v, \mathcal{E})\searrow \qquad \frac{(S, F[v], \mathcal{E})\searrow}{(F[\bullet] :: S, v, \mathcal{E})\searrow}$$

$$\frac{((\bullet N) :: S, M, \mathcal{E})\searrow}{(S, MN, \mathcal{E})\searrow} M \text{ not a value} \qquad \frac{((v\bullet) :: S, N, \mathcal{E})\searrow}{(S, vN, \mathcal{E})\searrow} M \text{ not a value}$$

$$\frac{(S, M[x := v], \mathcal{E})\searrow}{(S, (\lambda x.M)v, \mathcal{E})\searrow} \qquad \frac{(S, N[f := \lambda x.\mathsf{letrec}\ f = \lambda x.M\ \mathsf{in}\ M], \mathcal{E})\searrow}{(S, \mathsf{letrec}\ f = \lambda x.M\ \mathsf{in}\ N, \mathcal{E})\searrow}$$

$$\frac{(\langle \bullet, N\rangle :: S, M, \mathcal{E})\searrow}{(S, \langle M, N\rangle, \mathcal{E})\searrow} M \text{ not a value} \qquad \frac{(\langle v, \bullet\rangle :: S, N, \mathcal{E})\searrow}{(S, \langle v, N\rangle, \mathcal{E})\searrow} N \text{ not a value}$$

$$\frac{(\mathsf{fst}(\,\bullet\,) :: S, M, \mathcal{E})\searrow}{(S, \mathsf{fst}(M), \mathcal{E})\searrow} M \text{ not a value} \qquad \frac{(S, v, \mathcal{E})\searrow}{(S, \mathsf{fst}(\langle v, w\rangle), \mathcal{E})\searrow}$$

$$\frac{(\mathsf{snd}(\,\bullet\,) :: S, M, \mathcal{E})\searrow}{(S, \mathsf{snd}(M), \mathcal{E})\searrow} M \text{ not a value} \qquad \frac{(S, w, \mathcal{E})\searrow}{(S, \mathsf{snd}(\langle v, w\rangle), \mathcal{E})\searrow}$$

$$\frac{(T, M, \mathcal{E} \uplus (a \mapsto T))\searrow}{(S, [a]M, \mathcal{E} \uplus (a \mapsto T))\searrow} \qquad \frac{([], M, \mathcal{E} \uplus (a \mapsto S))\searrow}{(S, \mu a.M, \mathcal{E})\searrow}$$

Given two terms M and N such that $\emptyset \triangleright M\colon \phi, \Sigma$ and $\emptyset \triangleright N\colon \phi, \Sigma$, they are said to be *ciu-similar*, written $M \leq_{\phi,\Sigma} N$, just when $\forall S, \mathcal{E}.$ if $(S, M, \mathcal{E})\searrow$ then $(S, N, \mathcal{E})\searrow$. They are said to be *ciu-equivalent*, written $M \simeq_{\phi,\Sigma} N$ just when $M \leq_{\phi,\Sigma} N$ and $N \leq_{\phi,\Sigma} M$. Both these relations are extended to open terms in the obvious way.

This notion of equivalence is quite refined, consider the following terms (where Ω is a looping term, which can be defined using the recursion operator).

$$T_1 \stackrel{\text{def}}{=} \mu a.[a](\lambda y.\mu c.[a](\lambda x.\mathsf{ifz}\ y\ \mathsf{then}\ \Omega\ \mathsf{else}\ \underline{0}))$$
$$T_2 \stackrel{\text{def}}{=} \lambda z.\mu b.[b]((\lambda y.\mu c.[b]((\lambda x.\mathsf{ifz}\ y\ \mathsf{then}\ \Omega\ \mathsf{else}\ \underline{0})z))z)$$

It is easy to verify that $T_1\underline{n} \simeq_{\mathtt{int}} T_2\underline{n}$ for all natural numbers n. However they are *not* ciu-equivalent as $([(\lambda s.s(s\underline{1}))\bullet], T_1, \emptyset)\searrow$, but it is *not* the case that $([(\lambda s.s(s\underline{1}))\bullet], T_2, \emptyset)\searrow$. This is an important example as T_1 and T_2 *are* equivalent given the definition of applicative bisimilarity by Ong and Stewart [8]. (Their notion of bisimilarity is hence not a congruence.)

We can make the following definitions.

$$(M, \mathcal{E}) \Downarrow (v, \mathcal{E}') \stackrel{\text{def}}{=} (M, \mathcal{E}) \Rightarrow^* (v, \mathcal{E}') \text{and} (v, \mathcal{E}') \not\Rightarrow$$
$$(M, \mathcal{E}) \Downarrow \stackrel{\text{def}}{=} \exists v, \mathcal{E}'.(M, \mathcal{E}) \Downarrow (v, \mathcal{E}')$$

Let $\mathcal{C}$ be a context, which is a μPCF-term with (possibly many) hole(s) in it (not to be confused with an evaluation context). We say that two terms M and N are *contextually equivalent*, written $M \approx N$, when $\forall \mathcal{C}, \mathcal{E}.(\mathcal{C}[M], \mathcal{E}) \Downarrow$ iff $(\mathcal{C}[N], \mathcal{E}) \Downarrow$. In other words, two terms are contextually equivalent if no larger program can tell them apart.

The two terms given above (T_1 and T_2) are not contextually equivalent, as the context $(\lambda s.s(s\underline{1}))\bullet$ distinguishes them. Clearly this notion of contextual equivalence is highly desirable but awkward to work with given the quantification over all contexts. However the notion of ciu-equivalence is more usable and an interesting question is in what sense they are related. In fact we find that they coincide!

Theorem 1. $\forall M, N.M \approx N$ *iff* $M \simeq N$.

Proof. The proof is adapted from the standard one for purely functional languages (see, for example, the chapter by Pitts [11]). It uses a variant of Howe's method.

This means that to prove two terms contextually equivalent we need only to show that they are ciu-equivalent, which is significantly easier. For example, it is simple to show the following ciu-equivalences.

$$\begin{array}{ll} (\lambda x.M)v \simeq M[x := v] & \\ \mu a.[a]M \simeq M & a \notin \mu\mathrm{FV}(M) \\ (\mu a.M)N \simeq \mu b.M[[a]P \Leftarrow [b]PN] & \end{array}$$

For example, the second equivalence holds by the assumption that $a \notin \mu\mathrm{FV}(M)$ and by observing

$$\dfrac{\dfrac{(S, M, \mathcal{E} \uplus (a \mapsto S))\searrow}{([], [a]M, \mathcal{E} \uplus (a \mapsto S))\searrow}}{(S, \mu a.[a]M, \mathcal{E})\searrow}$$

7 Call-by-Name

This paper has so far considered only call-by-value computation. However it is very simple to provide a computational interpretation for a call-by-name evaluation strategy. The main difference is in the (new) definition of values, evaluation contexts and redexes, which are as follows.

Values	$v ::= \underline{n} \mid \lambda x.M \mid \langle M, M\rangle$
Evaluation Contexts	$E ::= \bullet \mid EM \mid \mathsf{fst}(E) \mid \mathsf{snd}(E) \mid \mathsf{suc}(E) \mid \mathsf{ifz}\ E\ \mathsf{then}\ M\ \mathsf{else}\ M$
Redexes	$R ::= vM \mid \mathsf{fst}(v) \mid \mathsf{snd}(v) \mid \mathsf{suc}(v) \mid \mathsf{ifz}\ v\ \mathsf{then}\ M\ \mathsf{else}\ M$ $\mid\ \mathsf{rec}\ x.M \mid [a]M \mid \mu a.M$

The evaluation rules are as before except for the following.

$$\begin{array}{rcl} (E[(\lambda x.M)N], \mathcal{E}) & \Rightarrow & (E[M[x := N]], \mathcal{E}) \\ (E[\mathsf{fst}(\langle M, N\rangle)], \mathcal{E}) & \Rightarrow & (E[M], \mathcal{E}) \\ (E[\mathsf{snd}(\langle M, N\rangle)], \mathcal{E}) & \Rightarrow & (E[N], \mathcal{E}) \\ (E[\mathsf{rec}\ x.M], \mathcal{E}) & \Rightarrow & (E[M[x := (\mathsf{rec}\ x.M)]], \mathcal{E}) \end{array}$$

The development of the corresponding operational theory follows closely that outlined in §6. The differs sharply from the treatment given by Ong and Stewart [8] who have to introduce completely new reduction rules to move from a call-by-name to a call-by-value setting.

8 Conclusion

In this paper I have given a simple computation interpretation of the $\lambda\mu$-calculus: it is a λ-calculus which is extended with indexed operators to manipulate the runtime environment. This is maybe not too surprising as Griffin [4] has shown the close relationship between classical logic and languages with control. This interpretation can be expressed as a single-step reduction semantics using environment contexts. In turn I gave an equivalent semantics expressed as steps of a simple abstract machine, which eliminated the need for the evaluation contexts. Using this simple abstract machine it is possible to define a notion of program equivalence based on a termination relation which coincides with a natural definition of contextual equivalence.

Clearly the work by Ong and Stewart [8] is most closely related to that reported here. Their thesis is that μPCF is a foundational language for call-by-value functional computation with control and this paper can be seen as further evidence to that claim. However I would claim that the operational treatment given here is more intuitive, more flexible (in that different calling mechanisms can be handled easily) and leads to a more refined notion of program equivalence.

References

1. G.M. BIERMAN. A classical linear λ-calculus. Technical Report 401, Cambridge Computer Laboratory 1996.
2. M. FELLEISEN. The theory and practice of first-class prompts. POPL 1988.
3. A.D. GORDON. Bisimilarity as a theory of functional programming: Mini-course. Technical Report NS–95–2, BRICS, Department of Computer Science, University of Århus, July 1995.
4. T.G. GRIFFIN. A formulae-as-types notion of control. POPL 1990.
5. C.A. GUNTER, D. RÉMY, AND J.G. RIECKE. A generalisation of exceptions and control in ML-like languages. FPCA 1995.
6. R. HARPER AND C. STONE. An interpretation of Standard ML in type theory. Technical Report CMU–CS–97–147, School of Computer Science, Carnegie Mellon University, June 1997.
7. M. HOFMANN AND T. STREICHER. Continuation models are universal for $\lambda\mu$-calculus. LICS 1997.
8. C.-H.L. ONG AND C.A. STEWART. A Curry-Howard foundation for functional computation with control. POPL 1997.
9. M. PARIGOT. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. LPAR 1992. LNCS 624.
10. A.M. PITTS. Operational semantics for program equivalence. Slides from talk given at MFPS, 1997.
11. A.M. PITTS. Operationally-based theories of program equivalence. In *Semantics and Logics of Computation*, CUP, 1997.