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 that the $\lambda\mu$-calculus itself has a simple computational interpretation: it is a typed λ-calculus which is able to save and restore the runtime environment. This interpretation is best given as a single-step semantics which, in particular, leads to a relatively simple, but powerful, operational theory.

This is an expanded version of a paper presented at the 23rd International Symposium on Mathematical Foundations of Computer Science. August 24–28, 1998. Brno, Czech Republic.

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 [25] 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 amazingly well behaved, satisfying both strong normalisation and confluence.

However two questions remain. First, what does this extension to the λ-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 has operators to save and restore the runtime environment. 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 [12]. However these techniques do not often extend to (call-by-value) languages with control. In §7 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 designated to be the active type and the rest which are said to be passive.

Types are given by the grammar

$$\phi ::= \bot \mid \phi \to \phi$$

and raw $\lambda\mu$-terms are given by the grammar

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

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

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

$$\dfrac{}{\Gamma, x{:}\,\phi \rhd x{:}\,\phi, \Sigma}\ \text{Identity}$$

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

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

The new rules are called Passivate and 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 Activate rule works similarly but in the reverse direction.

As an example the following derivation gives a (well-typed) term whose type is the *Peirce formula*.

$$\dfrac{\dfrac{}{y{:}\,(\phi \to \psi) \to \phi \rhd y{:}\,(\phi \to \psi) \to \phi, b{:}\,\phi} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{x{:}\,\phi, y{:}\,(\phi \to \psi) \to \phi \rhd x{:}\,\phi, a{:}\,\psi}\ \text{Identity}}{x{:}\,\phi, y{:}\,(\phi \to \psi) \to \phi \rhd [b{:}\,\phi]x{:}\,\bot, b{:}\,\phi, a{:}\,\psi}\ \text{Passivate}}{x{:}\,\phi, y{:}\,(\phi \to \psi) \to \phi \rhd \mu a{:}\,\psi.[b{:}\,\phi]x{:}\,\psi, b{:}\,\phi}\ \text{Activate}}{y{:}\,(\phi \to \psi) \to \phi \rhd \lambda x{:}\,\phi.\mu a{:}\,\psi.[b{:}\,\phi]x{:}\,\phi \to \psi, b{:}\,\phi} \to_{\mathcal{I}}}{\dfrac{\dfrac{\dfrac{y{:}\,(\phi \to \psi) \to \phi \rhd y(\lambda x{:}\,\phi.\mu a{:}\,\psi.[b{:}\,\phi]x){:}\,\phi, b{:}\,\phi}{y{:}\,(\phi \to \psi) \to \phi \rhd [b{:}\,\phi]y(\lambda x{:}\,\phi.\mu a{:}\,\psi.[b{:}\,\phi]x){:}\,\bot, b{:}\,\phi}\ \text{Passivate}}{y{:}\,(\phi \to \psi) \to \phi \rhd \mu b{:}\,\phi.[b{:}\,\phi]y(\lambda x{:}\,\phi.\mu a{:}\,\psi.[b{:}\,\phi]x){:}\,\phi}\ \text{Activate}}{\rhd \lambda y.\mu b{:}\,\phi.[b{:}\,\phi]y(\lambda x{:}\,\phi.\mu a{:}\,\psi.[b{:}\,\phi]x){:}\,((\phi \to \psi) \to \phi) \to \phi} \to_{\mathcal{I}}} \to_{\mathcal{E}}$$

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

$$\begin{array}{rcll}
(\lambda x{:}\,\phi.M)N & \rightsquigarrow_{\beta} & M[x := N] & \\
\\
\mu a{:}\,\phi.[a{:}\,\phi]M & \rightsquigarrow_{s} & M & \text{where } a \notin \mu\text{FV}(M) \\
[a{:}\,\phi]\mu b{:}\,\phi.M & \rightsquigarrow_{s} & M[a/b] & \\
\\
(\mu a{:}\,\phi \to \psi.M)N & \rightsquigarrow_{c} & \mu b{:}\,\psi.M[a{:}\,\phi \to \psi \Leftarrow [b{:}\,\psi] \bullet N] &
\end{array}$$

The β-rule is familiar from the λ-calculus. The $\lambda\mu$-calculus introduces three new reduction rules. Two are known as *simplification rules* [26] and are written $\rightsquigarrow_s$. In the first simplification rule, $\mu\text{FV}(M)$ denotes the set of free μ-variables in M, which is defined as follows.

$$\begin{array}{rcl}
\mu\text{FV}(x) & \stackrel{\text{def}}{=} & \emptyset \\
\mu\text{FV}(\lambda x.M) & \stackrel{\text{def}}{=} & \mu\text{FV}(M) \\
\mu\text{FV}(MN) & \stackrel{\text{def}}{=} & \mu\text{FV}(M) \cup \mu\text{FV}(N) \\
\mu\text{FV}([a]M) & \stackrel{\text{def}}{=} & \mu\text{FV}(M) \cup \{a\} \\
\mu\text{FV}(\mu a.M) & \stackrel{\text{def}}{=} & \mu\text{FV}(M) - \{a\}
\end{array}$$

[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.

A term is said to be λ-*closed* if it has no free λ-variables; it is said to be μ-*closed* if it has no free μ-variables, and *closed* if it is both λ-closed and μ-closed. In the second simplification rule, $M[a/b]$ denotes the term M where all free occurrences of the μ-variable b are replaced with a.

The third new reduction rule is essentially a commuting conversion, and is written $\rightsquigarrow_c$. I have used the notation $M[a \Leftarrow P[\bullet]]$ to denote the term M where *all* occurrences of the subterm $[a]N$ have been replaced by the term $P[N]$ (where $P[\bullet]$ is a term with a single hole in it, and $P[N]$ is the result of replacing the hole with N). This complicated commuting conversion is often glossed over in other papers. In truth there are actually two different rules depending on the type of the μ-variable a. In detail, the commuting conversion is defined as follows.

$$(\mu a\colon \phi \to \psi.M)N \quad \rightsquigarrow_c \quad \begin{cases} \mu b\colon \psi.M[a\colon \phi \to \psi \Leftarrow [b\colon \psi] \bullet N] & \text{where } \psi \neq \perp \\ M[a\colon \phi \to \psi \Leftarrow \bullet N] & \text{where } \psi = \perp \end{cases}$$

where

$$\begin{array}{rcl} x[a\colon \phi \Leftarrow P[\bullet]] & \stackrel{\text{def}}{=} & x \\ (\lambda x.M)[a\colon \phi \Leftarrow P[\bullet]] & \stackrel{\text{def}}{=} & \lambda x.(M[a\colon \phi \Leftarrow P[\bullet]]) \\ (MN)[a\colon \phi \Leftarrow P[\bullet]] & \stackrel{\text{def}}{=} & (M[a\colon \phi \Leftarrow P[\bullet]])(N[a\colon \phi \Leftarrow P[\bullet]]) \\ (\mu b\colon \psi.M)[a\colon \phi \Leftarrow P[\bullet]] & \stackrel{\text{def}}{=} & \mu b\colon \psi.(M[a\colon \phi \Leftarrow P[\bullet]]) \\ ([b\colon \phi]M)[a\colon \phi \Leftarrow P[\bullet]] & \stackrel{\text{def}}{=} & \begin{cases} P[M[a\colon \phi \Leftarrow P[\bullet]]] & \text{if } a = b \\ [b\colon \phi](M[a\colon \phi \Leftarrow P[\bullet]]) & \text{o'wise} \end{cases} \end{array}$$

As a rewriting step, this commuting conversion is a highly unusual rule, the substitution involves a replacement of term for term, rather than the more familiar substitution of term for variable. Certainly one would hope not to implement this operation in practice. Fortunately the treatment given in later sections removes the need to implement this substitution, in contrast to the framework given by Ong and Stewart [24].

Term will be assumed to have been written so that all forms of substitution are non-capturing.

3 A Computational Interpretation

In contrast to the situation for the λ-calculus, there is little attention in the literature to computational aspects of the $\lambda\mu$-calculus. How do programs execute? How do we handle different evaluation orders? What is the computational significance of having two distinct variable spaces? How can we reason about programs? At the time of writing only the paper by Ong and Stewart [24] addresses these sorts of questions. This paper is an attempt to provide an alternative, and hopefully simpler, approach to these questions.

Before presenting this approach I need first to introduce some standard terminology from work on control operators. To formalise the notion of an evaluation order, Felleisen and Friedman [8] 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 $E[M]$. The idea is that the hole sits at the place where reduction will next occur. In other words, evaluation contexts are devised so that every closed term, M, is either a value (canonical) or can be written *uniquely* as $E[R]$, where R is a redex. The context $E[\bullet]$ essentially represents 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 and is often referred to as 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

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

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 'saved' by adding it to $\mathcal{E}$, indexed with a. In the second reduction rule the current continuation is thrown away and the appropriate indexed continuation is restored from $\mathcal{E}$. In summary, the Activate and Passivate rules are interpreted as (indexed) *save* and *restore* operators, respectively.[2]

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 [24]. Thus μPCF types are given by the grammar

$$\begin{array}{rcl} \phi & ::= & \mathtt{int} \\ & | & \bot \\ & | & \phi \to \phi \\ & | & \phi \times \phi \end{array}$$

and the additional typing rules are as follows.

$$\frac{}{\Gamma \rhd \underline{n} \colon \mathtt{int}, \Sigma}\ \text{Nat} \qquad \frac{\Gamma \rhd M \colon \mathtt{int}, \Sigma}{\Gamma \rhd \mathsf{suc}(M) \colon \mathtt{int}, \Sigma}\ \text{Suc}$$

$$\frac{\Gamma \rhd M \colon \mathtt{int}, \Sigma \qquad \Gamma \rhd N \colon \phi, \Sigma \qquad \Gamma \rhd P \colon \phi, \Sigma}{\Gamma \rhd \mathsf{ifz}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P \colon \phi, \Sigma}\ \text{Conditional}$$

$$\frac{\Gamma \rhd M \colon \phi, \Sigma \qquad \Gamma \rhd N \colon \psi, \Sigma}{\Gamma \rhd \langle M, N \rangle \colon \phi \times \psi, \Sigma}\ \times_{\mathcal{I}} \quad \frac{\Gamma \rhd M \colon \phi \times \psi, \Sigma}{\Gamma \rhd \mathsf{fst}(M) \colon \phi, \Sigma}\ \times_{\mathcal{E}} \quad \frac{\Gamma \rhd M \colon \phi \times \psi, \Sigma}{\Gamma \rhd \mathsf{snd}(M) \colon \psi, \Sigma}\ \times_{\mathcal{E}}$$

$$\frac{\Gamma, f \colon \phi \to \phi, x \colon \phi \rhd M \colon \phi, \Sigma \qquad \Gamma, f \colon \phi \to \phi \rhd N \colon \psi, \Sigma}{\Gamma \rhd \mathsf{letrec}\ f = \lambda x.M\ \mathsf{in}\ N \colon \psi, \Sigma}\ \text{Recursion}$$

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.[3] It is important to note that what is developed in this and following sections can easily be adjusted to reflect a call-by-name strategy; some details are given in §9. 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 [24]).

[2] The reader familiar with control operators will recognise the save operation as a form of 'catch' and restore as a form of 'throw'.

[3] This means that recursion is restricted to function definitions in the usual way [33, §11.1].

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

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

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

In this section I give a number of examples of μPCF-programs to give the reader a feel for the computational power of the calculus. In particular, in §5.4, I shall reconsider the examples of encodings given by Ong and Stewart [24].

5.1 Idealised Scheme

Felleisen *et al.* [8, 9] presented an extension to the (untyped) call-by-value λ-calculus, called *Idealised Scheme*. Two new operators are added, written $\mathcal{A}(M)$ and $\mathcal{C}(M)$, which are called abort and control, respectively. Both forms are considered to be redexes and their reduction behaviour is given by the following rules.

$$\begin{array}{rcl}
E[\mathcal{A}(M)] & \leadsto_{\mathcal{A}} & M \\
E[\mathcal{C}(M)] & \leadsto_{\mathcal{C}} & M(\lambda z.\mathcal{A}(E[z]))
\end{array}$$

Informally $\mathcal{A}(M)$ abandons the current continuation, $E[\bullet]$. $\mathcal{C}(M)$ also abandons the current continuation and M is applied to the abstraction of the current continuation. If this abstraction is invoked with the value v in an evaluation context $E_1[\bullet]$, then $E_1[\bullet]$ will be abandoned and evaluation will continue with $E[v]$.

More concretely, consider the term

$$E_0[\mathcal{C}(\lambda j.M)].$$

Evaluation of this term can be thought of as a 'catch' which labels the current continuation $E_0[\bullet]$ with j. If j doesn't get used in the evaluation of M then $E_0[\bullet]$ is effectively garbaged. If an application of j occurs in the evaluation of M, e.g. $E_1[jv]$, then the computation is 'thrown' back to the evaluation context labelled with j, along with the value v. Thus computation continues with the term $E_0[v]$.

Griffin [14] showed that control and abort can be typed as follows.

$$\frac{\Gamma \rhd M\colon (\phi \to\perp) \to\perp}{\Gamma \rhd \mathcal{C}_\phi(M)\colon \phi}\,\text{Control} \qquad \frac{\Gamma \rhd M\colon \perp}{\Gamma \rhd \mathcal{A}_\phi(M)\colon \phi}\,\text{Abort}$$

Thus control corresponds to the double negation elimination rule proposed by Gentzen [11] for classical logic, and abort corresponds to the (intuitionistic) $\perp$-elimination rule.

With these typings in mind, these operators can be encoded in μPCF as follows.

$$\begin{array}{rcl} \mathcal{C}_\phi(M) & \stackrel{\text{def}}{=} & \mu a\colon\phi.M(\lambda z\colon\phi.[a\colon\phi]z) \\ \mathcal{A}_\phi(M) & \stackrel{\text{def}}{=} & \mu b\colon\phi.M \end{array}$$

(In both encodings the μ-variables are assumed to be fresh.) Consider an evaluation of the encoding of an abort in μPCF.

$$\begin{array}{rl} & (E_0[\mathcal{A}_\phi(M)], \mathcal{E}) \\ \stackrel{\text{def}}{=} & (E_0[\mu b\colon\phi.M], \mathcal{E}) \\ \Rightarrow & (M, \mathcal{E} \uplus \{b \mapsto E_0[\bullet]\}) \end{array}$$

As b is assumed to be fresh, then it is easy to see that the evaluation context $E_0[\bullet]$ is abandoned.

Consider an evaluation of the encoding of an application of control in μPCF.

$$\begin{array}{rll} & (E_0[\mathcal{C}_\phi(\lambda j.M)], \mathcal{E}) & \\ \stackrel{\text{def}}{=} & (E_0[\mu a\colon\phi.(\lambda j.M)\,(\lambda z\colon\phi.[a\colon\phi]z)], \mathcal{E}) & \\ \Rightarrow & ((\lambda j.M)\,(\lambda z\colon\phi.[a\colon\phi]z), \mathcal{E} \uplus \{a \mapsto E_0[\bullet]\}) & \\ \Rightarrow & (M[j := (\lambda z\colon\phi.[a\colon\phi]z)], \mathcal{E} \uplus \{a \mapsto E_0[\bullet]\}) & \\ \vdots & & \\ & (E_1[(jv)[j := (\lambda z\colon\phi.[a\colon\phi]z)]], \mathcal{E} \uplus \{a \mapsto E_0[\bullet]\}) & (\dagger) \\ \stackrel{\text{def}}{=} & (E_1[(\lambda z\colon\phi.[a\colon\phi]z)v], \mathcal{E} \uplus \{a \mapsto E_0[\bullet]\}) & \\ \Rightarrow & (E_1[[a\colon\phi]v], \mathcal{E} \uplus \{a \mapsto E_0[\bullet]\}) & \\ \Rightarrow & (E_0[v], \mathcal{E} \uplus \{a \mapsto E_0[\bullet]\}) & \end{array}$$

At stage ($\dagger$) the substitution has been left explicit to aid comparison with the earlier discussion of the control operator, $\mathcal{C}$, in Idealised Scheme. Hopefully it is clear that the effect of this encoding is that evaluation has been thrown back to E_0, along with the value v.

Remark. It is also possible to encode the Scheme variant of the Control operator [4], which does not initially abandon the current continuation. This rule, as observed by Griffin, is typed using the Peirce formula, i.e.

$$\frac{\Gamma \rhd M\colon (\phi \to \psi) \to \phi}{\Gamma \rhd \mathcal{P}(M)\colon \phi}$$

Its reduction rule is

$$E[\mathcal{P}(M)] \rightsquigarrow_{\mathcal{P}} E[M(\lambda z.\mathcal{A}(E[z]))]$$

and it can be encoded into μPCF as follows.

$$\mathcal{P}(M) \stackrel{\text{def}}{=} \mu b{:}\,\phi.[b{:}\,\phi]M(\lambda x{:}\,\phi.\mu a{:}\,\psi.[b{:}\,\phi]x)$$

It is left as an exercise to the reader to verify that this encoding has the expected operational bahviour.

5.2 de Groote's Exception Handling Calculus

de Groote [7] presented a simply-typed λ-calculus extended with an ML-like exception handling mechanism. A new class of exception variable is introduced and typing judgements are of the form $\Gamma; \Delta \triangleright M{:}\,\phi$ where Γ is the typing environment for normal λ-variables and Δ the typing environment for exception variables. Two new term constructors are introduced whose typing rules are as follows.

$$\frac{\Gamma; \Delta \triangleright M{:}\,\phi}{\Gamma; \Delta, e{:}\,\phi \to \perp \triangleright \mathit{raise}(e, M){:}\,\psi}\ \text{Raise} \qquad \frac{\Gamma; \Delta, e{:}\,\neg\phi \triangleright M{:}\,\psi \qquad \Gamma, x{:}\,\phi; \Delta \triangleright N{:}\,\psi}{\Gamma; \Delta \triangleright \mathit{let}\ e\ \mathit{in}\ M\ \mathit{handle}\ e\ x \Rightarrow N{:}\,\psi}\ \text{Handle}$$

(where $\neg\phi \stackrel{\text{def}}{=} \phi \to \perp$). As de Groote notes, the typing rule Raise corresponds to the standard (intuitionistic) rules for falsity. The typing rule Handle is clearly related to classical logic: it corresponds to the rule of the excluded middle.

Associated with these rules are a number of reduction rules.[4]

$$\begin{array}{rcll}
v(\mathit{raise}(e,w)) & \rightsquigarrow & \mathit{raise}(e,w) & \\
(\mathit{raise}(e,v))M & \rightsquigarrow & \mathit{raise}(e,v) & \\
\mathit{raise}(e,\mathit{raise}(e',v)) & \rightsquigarrow & \mathit{raise}(e',v) & \\
\mathit{let}\ e\ \mathit{in}\ v\ \mathit{handle}\ e\ x \Rightarrow N & \rightsquigarrow & v & (e \notin \text{FV}(v)) \\
\mathit{let}\ e\ \mathit{in}\ \mathit{raise}(e,v)\ \mathit{handle}\ e\ x \Rightarrow N & \rightsquigarrow & N[x := v] & (e \notin \text{FV}(v, N)) \\
\mathit{let}\ e\ \mathit{in}\ \mathit{raise}(e',v)\ \mathit{handle}\ e\ x \Rightarrow N & \rightsquigarrow & \mathit{raise}(e',v) & (e \notin \text{FV}(v))
\end{array}$$

This exception handling mechanism can be encoded into μPCF quite simply, as follows.

$$\begin{array}{rcl}
[\![\mathit{raise}(e,M)]\!] & \stackrel{\text{def}}{=} & (\lambda x.\mu b.[a]x)[\![M]\!] \\
[\![\mathit{let}\ e\ \mathit{in}\ M\ \mathit{handle}\ e\ x \Rightarrow N]\!] & \stackrel{\text{def}}{=} & \mu b.[b](\lambda x.[\![N]\!])(\mu e.[b][\![M]\!])
\end{array}$$

It is quite easy to verify that this translation preserves the expected operational behaviour. Here are two examples.

$$\begin{array}{rl}
 & (E[[\![\mathit{let}\ e\ \mathit{in}\ v\ \mathit{handle}\ e\ x \Rightarrow N]\!]], \mathcal{E}) \\
\stackrel{\text{def}}{=} & (E[\mu b.[b](\lambda x.[\![N]\!])(\mu e.[b][\![v]\!])], \mathcal{E}) \\
\Rightarrow^2 & (E[(\lambda x.[\![N]\!])(\mu e.[b][\![v]\!])], \mathcal{E} \uplus \{b \mapsto E[\bullet]\}) \\
\Rightarrow & ([b][\![v]\!], \mathcal{E} \uplus \{b \mapsto E[\bullet], e \mapsto E[(\lambda x.[\![N]\!])\bullet]\}) \\
\Rightarrow & (E[[\![v]\!]], \mathcal{E} \uplus \{b \mapsto E[\bullet], e \mapsto E[(\lambda x.[\![N]\!])\bullet]\})
\end{array}$$

[4] de Groote also gives a second, more complicated, set of reduction rules.

$$
\begin{array}{ll}
& (E[\llbracket \mathit{let}\, e\, \mathit{in}\, \mathsf{raise}(e,v)\ \mathit{handle}\, e\, x \Rightarrow N\rrbracket], \mathcal{E}) \\
\stackrel{\text{def}}{=} & (E[\mu b.[b](\lambda x.\llbracket N\rrbracket)(\mu e.[b](\lambda x.\mu c.[e]x)\llbracket v\rrbracket)], \mathcal{E}) \\
\Rightarrow^2 & (E[(\lambda x.\llbracket N\rrbracket)(\mu e.[b](\lambda x.\mu c.[e]x)\llbracket v\rrbracket)], \mathcal{E} \uplus \{b \mapsto E[\bullet]\}) \\
\Rightarrow & ([b](\lambda x.\mu c.[e]x)\llbracket v\rrbracket, \mathcal{E} \uplus \{b \mapsto E[\bullet], e \mapsto E[(\lambda x.\llbracket N\rrbracket)\bullet]\}) \\
\Rightarrow & (E[(\lambda x.\mu c.[e]x)\llbracket v\rrbracket], \mathcal{E} \uplus \{b \mapsto E[\bullet], e \mapsto E[(\lambda x.\llbracket N\rrbracket)\bullet]\}) \\
\Rightarrow & (E[\mu c.[e]\llbracket v\rrbracket], \mathcal{E} \uplus \{b \mapsto E[\bullet], e \mapsto E[(\lambda x.\llbracket N\rrbracket)\bullet]\}) \\
\Rightarrow & ([e]\llbracket v\rrbracket, \mathcal{E} \uplus \{b \mapsto E[\bullet], e \mapsto E[(\lambda x.\llbracket N\rrbracket)\bullet] c \mapsto E[\bullet]\}) \\
\Rightarrow & (E[(\lambda x.\llbracket N\rrbracket)\llbracket v\rrbracket], \mathcal{E} \uplus \{b \mapsto E[\bullet], e \mapsto E[(\lambda x.\llbracket N\rrbracket)\bullet] c \mapsto E[\bullet]\}) \\
\Rightarrow & (E[\llbracket N\rrbracket[x := \llbracket v\rrbracket]], \mathcal{E} \uplus \{b \mapsto E[\bullet], e \mapsto E[(\lambda x.\llbracket N\rrbracket)\bullet] c \mapsto E[\bullet]\})
\end{array}
$$

Remark. The reader familiar with SML will recognise that this exception handling mechanism is less powerful than that in SML. This is to be expected as it known that the typed λ-calculus extended with SML exception handling can encode the untyped λ-calculus, and thus is not strongly normalising [21]. As the typed $\lambda\mu$-calculus is strongly normalising, it clearly can not encode true SML exception handling.

5.3 Pairing

It is easy to verify that $\phi \times \psi \equiv \neg(\phi \rightarrow \neg\psi)$ in **CL**. This logical equivalence can be used to simulate pairing in μPCF. The constructor and deconstructors are encoded as follows.[5]

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

It is simple to see that these encodings satisfy the expected (call-by-value) behaviour, e.g.

$$
\begin{array}{ll}
& (\mathit{fst}\ (\mathit{pair}\, v\ w), \mathcal{E}) \\
\stackrel{\text{def}}{=} & (\mathit{fst}\ ((\lambda mnf.f\ m\ n)\ v\ w), \mathcal{E}) \\
\Rightarrow^2 & (\mathit{fst}\ (\lambda f.f\ v\ w), \mathcal{E}) \\
\Rightarrow & (\mu a.((\lambda f.f\ v\ w)(\lambda x.\mu b.[a]x)), \mathcal{E}) \\
\Rightarrow & ((\lambda f.f\ v\ w)(\lambda x.\mu b.[a]x), \mathcal{E} \uplus \{a \mapsto \bullet\}) \\
\Rightarrow & ((\lambda x.\mu b.[a]x)\ v\ w, \mathcal{E} \uplus \{a \mapsto \bullet\}) \\
\Rightarrow & ((\mu b.[a]v)\ w, \mathcal{E} \uplus \{a \mapsto \bullet\}) \\
\Rightarrow & ([a]v, \mathcal{E} \uplus \{a \mapsto \bullet, b \mapsto (\bullet w)\}) \\
\Rightarrow & (v, \mathcal{E} \uplus \{a \mapsto \bullet, b \mapsto (\bullet w)\})
\end{array}
$$

5.4 Ong/Stewart Encodings

In their paper, Ong and Stewart [24] give variants of callcc and exceptions and show how they can be encoded in μPCF. I shall reconsider these encodings in the light of the simpler operational treatment offered by this paper.

5.4.1 Exceptions

Ong and Stewart considered extending PCF with a simple system of exception handling, which was inspired by (but less powerful than) work by Gunter *et al.* [15]. (Their system is actually quite similar to de Groote's system, considered in §5.2.) Typed exceptions are

[5] A similar encoding using Idealised Scheme was given by Griffin [14].

identified with names, thus typing judgements are now of the form $\Gamma;\Delta \triangleright M\colon\phi$ where Γ is the usual typing environment and Δ is the typing environment for the exception names. Two new operators are added to PCF whose typing rules are as follows.

$$\frac{\Gamma;\Delta \triangleright M\colon\phi}{\Gamma;\Delta, a\colon\phi \triangleright \mathit{raise}(a,M)\colon\psi} \qquad \frac{\Gamma;\Delta, a\colon\phi \triangleright M\colon\phi\to\psi \qquad \Gamma;\Delta, a\colon\phi \triangleright N\colon\psi}{\Gamma;\Delta \triangleright \mathit{handle}(a,M,N)\colon\psi}$$

The intended interpretation is that the term $\mathit{raise}(a,M)$ first evaluates M to a value v and then raises an exception named a associated with v. The term $\mathit{handle}(a,M,N)$ 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.

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

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

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

It is relatively simple to show that this translation preserves the operational behaviour, e.g. (where $[\![M]\!] \Rightarrow^* v$ and $[\![N]\!] \Rightarrow^* u$)

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

5.4.2 Callcc

Ong and Stewart also considered extending PCF with a variant of callcc (again inspired by work by Gunter *et al.* [15]). Continuations are both typed and associated with names, and so typing judgements are of the form $\Gamma;\Delta \triangleright M\colon\phi$, where Δ is the typing environment for continuation names. Three new operators are added to PCF, whose typing rules are as follows.

$$\frac{\Gamma;\Delta \triangleright M\colon(\phi\to\psi)\to\phi}{\Gamma;\Delta \triangleright \mathit{callcc}(M)\colon\phi} \qquad \frac{\Gamma;\Delta \triangleright M\colon\phi}{\Gamma;\Delta, a\colon\phi \triangleright \mathit{abort}(a,M)\colon\psi} \qquad \frac{\Gamma;\Delta, a\colon\phi \triangleright M\colon\phi}{\Gamma;\Delta \triangleright \mathit{set}(a,M)\colon\phi}$$

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). Given as reduction rules their intended operational behaviour is as follows.

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

These operators can be translated into μPCFas follows.

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

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

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

6 A Transition System

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 evaluation context and the redex are actually separated. 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.

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

(Clearly $\mathcal{E}$ is now a function from μ-variables to stacks.) The reduction rules essentially describe the transitions of a simple abstract machine.[6] In full they are as follows.

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

[6] Harper and Stone [16] give simple transition rules in their analysis of SML and Pitts [27] has used similar rules in work on functional languages with dynamic allocation of store.

$$
\begin{array}{rcl}
(S, \mathsf{ifz}\ \underline{0}\ \mathsf{then}\ M\ \mathsf{else}\ N, \mathcal{E}) & \longrightarrow & (S, M, \mathcal{E}) \\
(S, \mathsf{ifz}\ (\underline{n+1})\ \mathsf{then}\ M\ \mathsf{else}\ N, \mathcal{E}) & \longrightarrow & (S, N, \mathcal{E}) \\
(S, \mathsf{letrec}\ f = \lambda x.M\ \mathsf{in}\ N, \mathcal{E}) & \longrightarrow & (S, N[f := \lambda x.\mathsf{letrec}\ f = \lambda x.M\ \mathsf{in}\ M], \mathcal{E}) \\
(S, \mu a.M, \mathcal{E}) & \longrightarrow & ([], M, \mathcal{E} \uplus \{a \mapsto S\}) \\
(S, [a]M, \mathcal{E} \uplus \{a \mapsto T\}) & \longrightarrow & (T, M, \mathcal{E} \uplus \{a \mapsto T\})
\end{array}
$$

An example may make these reduction rules clearer. Consider an instance of the Ong/Stewart 'callcc' reduction rule given in §5.4.2.

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

The left hand term is translated into the following μPCF-term.

$$
\mu a.[a](\lambda x.[\![N]\!])(\mu b.[a][\![M]\!])
$$

The reduction of this term is as follows.

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

To compare these two formulations I shall 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.

$$
\begin{array}{rcl}
\ulcorner \bullet \urcorner & \stackrel{\mathrm{def}}{=} & [] \\
\ulcorner vE \urcorner & \stackrel{\mathrm{def}}{=} & \ulcorner E \urcorner @ [v\bullet] \\
\ulcorner EM \urcorner & \stackrel{\mathrm{def}}{=} & \ulcorner E \urcorner @ [\bullet M]
\end{array}
$$

$$
[]@M \stackrel{\mathrm{def}}{=} M
$$

For example

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

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

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

An important fact (first discovered by Pitts [27] in a different setting) is that the set

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

has a direct, inductive definition which is as follows.

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

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

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

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

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

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

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

These rules will form the basis of a notion of program equivalence given in the next section.

7 Program Equivalence

μPCF is a simple functional programming language for recursively defined, higher order functions along with facilities to save and restore the runtime environment. As for pure functional languages, we should like to develop methods for reasoning about properties of μPCF programs. One possibility is to devise a set of equations and reason equationally about programs. Such equations have been considered for Idealised Scheme by Felleisen *et al.* [10], Griffin [14] and Hofmann [18]. Another possibility is to develop a denotational model and reason about programs via their denotations. Cartwright *et al.* [3] give a (fully abstract) model of Idealised Scheme and some models of the $\lambda\mu$-calculus have been considered recently by Ong [23], Hofmann and Streicher [19] and Selinger [30].

In this paper I shall rather consider techniques based on the *operational behaviour* of μPCF programs. One advantage of such operationally based techniques is that they require relatively little mathematical overhead. (Further arguments in favour of operationally based techniques can be found in the literature, e.g. [13].)

Morris-style, contextual equivalence is accepted as a natural notion of equivalence for sequential languages. Essentially two programs are considered contextually equivalent if interchanging one for the other in any larger program does not affect the result. Before the formal definition, here are a couple of simple definitions.

$$
\begin{array}{rcl}
(M,\mathcal{E}) \Downarrow (v,\mathcal{E}') & \stackrel{\text{def}}{=} & (M,\mathcal{E}) \Rightarrow^{*} (v,\mathcal{E}') \text{ and } (v,\mathcal{E}') \not\Rightarrow \\
(M,\mathcal{E}) \Downarrow v & \stackrel{\text{def}}{=} & \exists \mathcal{E}'.(M,\mathcal{E}) \Downarrow (v,\mathcal{E}') \\
(M,\mathcal{E}) \Downarrow & \stackrel{\text{def}}{=} & \exists v.(M,\mathcal{E}) \Downarrow v \\
(M,\mathcal{E}) \Uparrow & \stackrel{\text{def}}{=} & \neg(\exists v.(M,\mathcal{E}) \Downarrow v)
\end{array}
$$

Definition 1. *Let M and N be terms and $\mathcal{C}$ a λ-closing context. M is said to* contextually refine N*, written $\Gamma \rhd M \sqsubseteq N\colon \phi, \Sigma$, when $\forall \mathcal{C}, \mathcal{E}$. if $(\mathcal{C}[M], \mathcal{E}) \Downarrow$ then $(\mathcal{C}[N], \mathcal{E}) \Downarrow$. They are said to be* contextually equivalent*, written $\Gamma \rhd M \approx N\colon \phi, \Sigma$, just when $\Gamma \rhd M \sqsubseteq N\colon \phi, \Sigma$ iff $\Gamma \rhd N \sqsubseteq M\colon \phi, \Sigma$.*

Reasoning about languages with control is difficult. One reason for this is that, in terms of contextual equivalence, control is invariably a non-conservative extension. By this, I mean that terms which are contextually equivalent in the call-by-value λ-calculus, may no longer be contextually equivalent in a call-by-value λ-calculus extended with control. For example, consider the following PCF-terms (this example is due to Meyer and Riecke [22]).

$$
\begin{array}{rcl}
M_1 & \stackrel{\text{def}}{=} & \lambda x.\lambda y.\lambda z.(\lambda w.(y\ x)\ w)(z\ x)\colon \phi \to (\phi \to \psi \to \varphi) \to (\phi \to \psi) \to \varphi \\
M_2 & \stackrel{\text{def}}{=} & \lambda x.\lambda y.\lambda z.(y\ x)(z\ x)\colon \phi \to (\phi \to \psi \to \varphi) \to (\phi \to \psi) \to \varphi
\end{array}
$$

It can be seen that these terms are PCF-contextually equivalent: Meyer and Riecke claim an complicated inductive argument can be used, although a more modern approach would be to show that they are applicatively bisimilar (some details of the definition are given by Pitts [28, §6]) and use the fact that applicative bisimilarity coincides with contextual equivalence.

Of course, M_1 and M_2 are valid μPCF-terms. However they are *not* μPCF-contextually equivalent: they can be distinguished by the context

$$\mathcal{C} \stackrel{\text{def}}{=} \mu b\colon \texttt{int}.[b\colon \texttt{int}]((\bullet\ 1)(\lambda u.\Omega^{\texttt{int}\to\texttt{int}})(\lambda v.\mu a.[b]1))$$

where $\Omega^{\texttt{int}\to\texttt{int}}$ is a non-terminating term.[7] Thus $\mathcal{C}[M_1]$ terminates by the following steps (where, for clarity, I have in places underlined the redex).

$$
\begin{array}{ll}
 & (\mathcal{C}[M_1], \mathcal{E}) \\
\stackrel{\text{def}}{=} & (\underline{\mu b\colon \texttt{int}.[b\colon \texttt{int}]((M_1\ 1)(\lambda u.\Omega)(\lambda v.\mu a.[b]1))}, \mathcal{E}) \\
\Rightarrow^2 & \underline{((M_1\ 1)(\lambda u.\Omega)(\lambda v.\mu a.[b]1), \mathcal{E} \uplus \{b \mapsto \bullet\})} \\
\stackrel{\text{def}}{=} & (((\lambda x.\lambda y.\lambda z.(\lambda w.(y\ x)\ w)(z\ x))\ 1)(\lambda u.\Omega)(\lambda v.\mu a.[b]1), \mathcal{E} \uplus \{b \mapsto \bullet\}) \\
\Rightarrow^3 & ((\lambda w.((\lambda u.\Omega)\ 1)\ w)(\underline{(\lambda v.\mu a.[b]1)\ 1}), \mathcal{E} \uplus \{b \mapsto \bullet\}) \\
\Rightarrow & ((\lambda w.((\lambda u.\Omega)\ 1)\ w)(\underline{\mu a.[b]1}), \mathcal{E} \uplus \{b \mapsto \bullet\}) \\
\Rightarrow & ([b]1, \mathcal{E} \uplus \{b \mapsto \bullet, a \mapsto (\lambda w.((\lambda u.\Omega)\ 1)\ w)\bullet\}) \\
\Rightarrow & (1, \mathcal{E} \uplus \{b \mapsto \bullet, a \mapsto (\lambda w.((\lambda u.\Omega)\ 1)\ w)\bullet\})
\end{array}
$$

Unfortunately $\mathcal{C}[M_2]$ does not terminate. It evaluates as follows.

$$
\begin{array}{ll}
 & (\mathcal{C}[M_2], \mathcal{E}) \\
\stackrel{\text{def}}{=} & (\underline{\mu b\colon \texttt{int}.[b\colon \texttt{int}]((M_2\ 1)(\lambda u.\Omega)(\lambda v.\mu a.[b]1))}, \mathcal{E}) \\
\Rightarrow^2 & \underline{((M_2\ 1)(\lambda u.\Omega)(\lambda v.\mu a.[b]1), \mathcal{E} \uplus \{b \mapsto \bullet\})} \\
\stackrel{\text{def}}{=} & (((\lambda x.\lambda y.\lambda z.(y\ x)(z\ x))\ 1)(\lambda u.\Omega)(\lambda v.\mu a.[b]1), \mathcal{E} \uplus \{b \mapsto \bullet\}) \\
\Rightarrow^3 & ((\underline{(\lambda u.\Omega)\ 1})((\lambda v.\mu a.[b]1)\ 1), \mathcal{E} \uplus \{b \mapsto \bullet\}) \\
\Rightarrow & (\underline{\Omega}\ ((\lambda v.\mu a.[b]1)\ 1), \mathcal{E} \uplus \{b \mapsto \bullet\}) \\
\Uparrow &
\end{array}
$$

[7]For example $\Omega^{\texttt{int}\to\texttt{int}} \stackrel{\text{def}}{=} \mathsf{letrec}\ f = \lambda x\colon \texttt{int} \to \texttt{int}.fx\ \mathsf{in}\ f(\lambda y\colon \texttt{int}.y)$.

This counter-example is quite serious. Both M_1 and M_2 are closed terms and so they are not distinguished by binding some free μ-variable in some complicated way. In fact they differ by having a slightly different order of evaluation in their bodies. In μPCF there is enough computational power to differentiate between these different orders of evaluation (a similar thing happens with languages with state [29]).

Here's another problem when reasoning about μPCF programs. Typically given two programs of function type, we might think that if they behave in the same way for all given arguments, then they can safely be thought of as indistinguishable, i.e. contextually equivalent. For example, notions of applicative bisimilarity of PCF-programs use this idea in their definition [1, 28, 12]. As one may have feared, things are more complicated for μPCF. Consider the closed programs

$$\begin{array}{lcl} T_1 & \stackrel{\text{def}}{=} & \mu a.[a](\lambda y.\mu c.[a](\lambda x.\mathsf{ifz}\ y\ \mathsf{then}\ \Omega\ \mathsf{else}\ \underline{0}))\colon \mathtt{int} \to \mathtt{int} \\ T_2 & \stackrel{\text{def}}{=} & \lambda z.\mu b.[b]((\lambda y.\mu c.[b]((\lambda x.\mathsf{ifz}\ y\ \mathsf{then}\ \Omega\ \mathsf{else}\ \underline{0})z))z)\colon \mathtt{int} \to \mathtt{int} \end{array}$$

where, again, Ω is a non-terminating term. It is easy to verify that for all natural numbers n and $\mathcal{E}$

$$(T_1\underline{n}, \mathcal{E}) \Downarrow \underline{0} \Longleftrightarrow (T_2\underline{n}, \mathcal{E}) \Downarrow \underline{0}$$

Thus given the argument above, one may conclude that T_1 and T_2 should be thought of as contextually equivalent. Unfortunately they are not, as the context

$$\mathcal{C} \stackrel{\text{def}}{=} (\lambda s.s(s\underline{1}))\bullet$$

can distinguish between T_1 and T_2! Indeed $\mathcal{C}[T_1]$ terminates in the following way.

$$\begin{array}{ll} & (\mathcal{C}[T_1], \mathcal{E}) \\ \stackrel{\text{def}}{=} & ((\lambda s.s(s\underline{1}))(\underline{\mu a.[a](\lambda y.\mu c.[a](\lambda x.\mathsf{ifz}\ y\ \mathsf{then}\ \Omega\ \mathsf{else}\ \underline{0}))}), \mathcal{E}) \\ \Rightarrow^2 & ((\lambda s.s(s\underline{1}))(\lambda y.\mu c.[a](\lambda x.\mathsf{ifz}\ y\ \mathsf{then}\ \Omega\ \mathsf{else}\ \underline{0})), \mathcal{E} \uplus \{a \mapsto (\lambda s.s(s\underline{1}))\bullet\}) \\ \Rightarrow & (U(U\underline{1}), \mathcal{E} \uplus \{a \mapsto (\lambda s.s(s\underline{1}))\bullet\}) \\ \Rightarrow & (U(\mu c.[a](\lambda x.\mathsf{ifz}\ \underline{1}\ \mathsf{then}\ \Omega\ \mathsf{else}\ \underline{0})), \mathcal{E} \uplus \{a \mapsto (\lambda s.s(s\underline{1}))\bullet\}) \\ \Rightarrow^2 & ((\lambda s.s(s\underline{1}))(\lambda x.\mathsf{ifz}\ \underline{1}\ \mathsf{then}\ \Omega\ \mathsf{else}\ \underline{0}), \mathcal{E} \uplus \{a \mapsto (\lambda s.s(s\underline{1}))\bullet, c \mapsto U\bullet\}) \\ \Rightarrow^3 & (\underline{0}, \mathcal{E} \uplus \{a \mapsto (\lambda s.s(s\underline{1}))\bullet, c \mapsto U\bullet\}) \end{array}$$

(U is used as shorthand for the term $\lambda y.\mu c.[a](\lambda x.\mathsf{ifz}\ y\ \mathsf{then}\ \Omega\ \mathsf{else}\ \underline{0})$.) However $\mathcal{C}[T_2]$ loops in the following way.

$$\begin{array}{ll} & (\mathcal{C}[T_2], \mathcal{E}) \\ \stackrel{\text{def}}{=} & ((\lambda s.s(s\underline{1}))T_2, \mathcal{E}) \\ \Rightarrow & (T_2(T_2\underline{1}), \mathcal{E}) \\ \Rightarrow & (T_2(\underline{\mu b.[b]((\lambda y.\mu c.[b]((\lambda x.\mathsf{ifz}\ y\ \mathsf{then}\ \Omega\ \mathsf{else}\ \underline{0})\underline{1}))\underline{1})}), \mathcal{E}) \\ \Rightarrow^2 & (T_2(\underline{(\lambda y.\mu c.[b]((\lambda x.\mathsf{ifz}\ y\ \mathsf{then}\ \Omega\ \mathsf{else}\ \underline{0})\underline{1}))\underline{1}}), \mathcal{E} \uplus \{b \mapsto T_2\bullet\}) \\ \Rightarrow & (T_2(\underline{\mu c.[b]((\lambda x.\mathsf{ifz}\ \underline{1}\ \mathsf{then}\ \Omega\ \mathsf{else}\ \underline{0})\underline{1})}), \mathcal{E} \uplus \{b \mapsto T_2\bullet\}) \\ \Rightarrow^2 & (T_2(\underline{(\lambda x.\mathsf{ifz}\ \underline{1}\ \mathsf{then}\ \Omega\ \mathsf{else}\ \underline{0})\underline{1}}), \mathcal{E} \uplus \{b \mapsto T_2\bullet, c \mapsto T_2\bullet\}) \\ \Rightarrow & (T_2\underline{0}, \mathcal{E} \uplus \{b \mapsto T_2\bullet, c \mapsto T_2\bullet\}) \\ \Rightarrow & (\mu b'.[b']((\lambda y.\mu c'.[b']((\lambda x.\mathsf{ifz}\ y\ \mathsf{then}\ \Omega\ \mathsf{else}\ \underline{0})\underline{0}))\underline{0}), \mathcal{E} \uplus \{b \mapsto T_2\bullet, c \mapsto T_2\bullet\}) \\ \Rightarrow^2 & ((\lambda y.\mu c'.[b']((\lambda x.\mathsf{ifz}\ y\ \mathsf{then}\ \Omega\ \mathsf{else}\ \underline{0})\underline{0}))\underline{0}), \mathcal{E} \uplus \{b \mapsto T_2\bullet, c \mapsto T_2\bullet, b' \mapsto \bullet\}) \\ \Rightarrow & (\mu c'.[b']((\lambda x.\mathsf{ifz}\ \underline{0}\ \mathsf{then}\ \Omega\ \mathsf{else}\ \underline{0})\underline{0}), \mathcal{E} \uplus \{b \mapsto T_2\bullet, c \mapsto T_2\bullet, b' \mapsto \bullet\}) \\ \Rightarrow^2 & ((\lambda x.\mathsf{ifz}\ \underline{0}\ \mathsf{then}\ \Omega\ \mathsf{else}\ \underline{0})\underline{0}, \mathcal{E} \uplus \{b \mapsto T_2\bullet, c \mapsto T_2\bullet, b' \mapsto \bullet, c' \mapsto \bullet\}) \\ \Rightarrow & (\mathsf{ifz}\ \underline{0}\ \mathsf{then}\ \Omega\ \mathsf{else}\ \underline{0}, \mathcal{E} \uplus \{b \mapsto T_2\bullet, c \mapsto T_2\bullet, b' \mapsto \bullet, c' \mapsto \bullet\}) \\ \Uparrow & \end{array}$$

A consequence of this example is the fact that simple definitions of applicative bisimilarity are unlikely to coincide with contextual equivalence. (Indeed as Ong and Stewart point out, the most obvious definition of applicative bisimilarity fails even to be a congruence!)

The problem with contextual equivalence is that it is very hard to reason about. As demonstrated above, it is easy to see when two programs are *not* contextually equivalent—one simply finds a context which distinguishes them. To show that two programs *are* contextually equivalent is much harder: the problem lies in the quantification over all program contexts. Indeed, the discussion above demonstrates how μPCF's save and restore features also complicate the problem.

Despite these difficulties let us consider a very simple notion of program equivalence which uses the notion of termination given at the end of §6. Informally, the idea is that two programs are considered equivalent if they have the same termination properties. The formal definition is as follows.

Definition 2. *Given two programs M and N, M is said to* ciu-refine *N, written $M \leq N\colon \phi, \Sigma$, 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 $M \simeq N\colon \phi, \Sigma$ just when $M \leq N\colon \phi, \Sigma$ and $M \leq N\colon \phi, \Sigma$.*

These can be extended to open terms as follows

$$\begin{array}{lcl} \vec{x}\colon \Gamma \rhd M \leq^{\circ} N\colon \phi, \Sigma & \stackrel{\text{def}}{=} & \forall \vec{v}. M[\vec{x} := \vec{v}] \leq N[\vec{x} := \vec{v}]\colon \phi, \Sigma \\ \vec{x}\colon \Gamma \rhd M \simeq^{\circ} N\colon \phi, \Sigma & \stackrel{\text{def}}{=} & \forall \vec{v}. M[\vec{x} := \vec{v}] \simeq N[\vec{x} := \vec{v}]\colon \phi, \Sigma \end{array}$$

The obvious question is in what way are ciu-equivalence and contextual equivalence related? It is possible to show that they coincide, or, put in another way, ciu-equivalence is an alternative (but relatively simpler) characterisation of contextual equivalence.

Theorem 1. (ciu theorem) $\Gamma \rhd M \approx N\colon \phi, \Sigma$ *iff* $\Gamma \rhd M \simeq^{\circ} N\colon \phi, \Sigma$.

Proof. Some details are given in Appendix A. ■

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 can be shown that the (call-by-value variants of the) reduction rules of the $\lambda\mu$-calculus given in §2 are contextual equivalences, by showing that they are, in fact, ciu-equivalences.

$$\begin{array}{rcll} (\lambda x\colon \phi.M)v & \approx & M[x := v]\colon \psi, \Sigma & \qquad (1) \\ \mu a\colon \phi.[a\colon \phi]M & \approx & M\colon \phi, \Sigma \qquad (a \notin \mu\text{FV}(M)) & \qquad (2) \\ [a\colon \phi]\mu b\colon \phi.[c\colon \psi]M & \approx & ([c\colon \psi]M)[a/b]\colon \bot, a\colon \phi, \Sigma & \qquad (3) \\ (\mu a\colon \phi \to \psi.M)N & \approx & \mu b\colon \psi.M[a \Leftarrow [b\colon \psi] \bullet N]\colon \psi, \Sigma & \qquad (4) \end{array}$$

For example, the equivalence (2) holds by observing

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

and the fact that $a \notin \mu\text{FV}(M)$ by assumption.

A number of other equivalences can be derived. For example, consider a term M such that $\Gamma, x\colon \phi \triangleright M\colon \psi, \Sigma$. An instance of equivalence (1) is

$$\Gamma, x\colon \phi \triangleright (\lambda x\colon \phi.M)x \simeq^\circ M\colon \psi, \Sigma \tag{5}$$

In Appendix A it is shown that ciu-equivalence is a congruence. In particular, this means that the following rule is valid.

$$\frac{\Gamma, x\colon \phi \triangleright M \simeq^\circ N\colon \psi, \Sigma}{\Gamma \triangleright \lambda x\colon \phi.M \simeq^\circ \lambda x\colon \phi.N\colon \phi \to \psi, \Sigma}$$

Applying this to equivalence 5 gives

$$\Gamma \triangleright \lambda x\colon \phi.(\lambda x\colon \phi.M)x \simeq^\circ \lambda x\colon \phi.M\colon \phi \to \psi, \Sigma.$$

Setting v for the term $\lambda x\colon \phi.M$, this implies the following contextual equivalence.

$$\Gamma \triangleright \lambda x\colon \phi.vx \approx v\colon \phi \to \psi, \Sigma \tag{6}$$

(where $x \notin \text{FV}(v)$). Hence the call-by-value η-rule is a contextual equivalence. Here are a couple more examples of contextual equivalences.

$$E[\mu a.[b]M] \approx [b]M\colon \bot \qquad (a \notin \mu\text{FV}(M)) \tag{7}$$

$$\mu a.[b](\mathsf{ifz}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P) \approx \mathsf{ifz}\ M\ \mathsf{then}\ \mu a.[b]N\ \mathsf{else}\ \mu a.[b]P\colon \phi \qquad (a \notin \mu\text{FV}(M)) \tag{8}$$

The first equivalence is a sort of 'abort' axiom, the second is a typical compiler optimisation. Further study of provable contextual equivalences is important future work.

8 Implementation

In this section I shall give details of two implementations. The first is simply the SML code for a tail recursive interpreter based on the transition system of §6. The second is a simple abstract machine, which is based on the framework proposed by Curien [5].

8.1 Interpreter

As mentioned in §6, the transition system essentially describes a tail recursive interpreter for μPCF. In this section I shall give details for such an interpreter, written in SML. The function mapping μ-variables to stacks is implemented as a list of pairs. The main datatypes are as follows.

```
datatype 'a exp = const   of int
                | suc     of 'a exp
                | var     of 'a
                | abs     of 'a * ('a exp)
                | app     of ('a exp) * ('a exp)
                | pair    of ('a exp) * ('a exp)
                | fst     of ('a exp)
                | snd     of ('a exp)
```

```
                | ifz     of ('a exp) * ('a exp) * ('a exp)
                | letrec  of 'a * 'a * ('a exp) * ('a exp)
                | save    of 'a * ('a exp)
                | restore of 'a * ('a exp);

type term = string exp;
(* type typed_term = (string*formula) exp *)

datatype hole = Bullet;

datatype frame = csuc  of hole
               | appl  of hole*term
               | appr  of term*hole
               | pairl of hole*term
               | pairr of term*hole
               | cfst  of hole
               | csnd  of hole
               | cifz  of hole*term*term;

type stack = frame list;

type stack_function = (string*stack) list;
(* type typed_stack_function = ((string*formula)*stack) list *)
```

The main interpreter consists of two mutually recursive routines `eval` and `unwind`. They call three subsidiary functions: `insert` and `get`, which handle the function update and access respectively, and `subs` which performs substitution of a term for a variable. The code for the interpreter is then as follows.

```
fun eval S (const i)         C = unwind S (const i) C
  | eval S (suc(e))          C = eval (csuc(Bullet)::S) e C
  | eval S (abs(x,e))        C = unwind S (abs(x,e)) C
  | eval S (app(e,f))        C = eval (appl(Bullet,f)::S) e C
  | eval S (fst(e))          C = eval (cfst(Bullet)::S) e C
  | eval S (snd(e))          C = eval (csnd(Bullet)::S) e C
  | eval S (pair(e,f))       C = eval (pairl(Bullet,f)::S) e C
  | eval S (ifz(e,f,g))      C = eval (cifz(Bullet,f,g)::S) e C
  | eval S (letrec(f,x,e,g)) C = unwind
                                   S
                                   subs(abs(x,letrec(f,x,e,e),f,g)
                                   C
  | eval S (restore(n,e))    C = let val T=get n C
                                 in eval T e C
                                 end
  | eval S (save(n,e))       C = eval [] e (insert (n,S) C)
and
  unwind S v C = case S of
  []                         => v
| (csuc(Bullet)::S)          => let val (const i) = v
                                in unwind S (const (i+1)) C
                                end
| (appl(Bullet,e)::S)        => eval (appr(v,Bullet)::S) e C
| (appr(abs(x,e),Bullet)::S) => eval S (subs(v,x,e)) C
```

```
| (pairl(Bullet,e)::S)        => eval (pairr(v,Bullet)::S) e C
| (pairr(w,Bullet)::S)        => unwind S (pair(w,v)) C
| (cfst(Bullet)::S)           => let val pair(u,w)=v
                                 in unwind S u C
                                 end
| (csnd(Bullet)::S)           => let val pair(u,w)=v
                                 in unwind S w C
                                 end
| (cifz(Bullet,f,g)::S)       => let val (const i) = v
                                 in if (i=0) then eval S f C
                                             else eval S g C
                                 end;
```

8.2 Towards an Abstract Machine

In his paper, Curien [5] describes a simple framework for abstract machines which evaluate functional programs using environments and closures. Both the CAM and Krivine's machine arise naturally as instances of this abstract framework. In this section, I shall extend Curien's "Eager Machine" to μPCF (actually without recursion).

The machine essentially consists of a triple. The first part is an *environment*. For simplicity I shall consider this to be a pair of symbol tables (functions), the first maps λ-variables to values and the second mapping μ-variables to stacks. In a realistic machine this indirection would be removed by compiling both λ- and μ-variables into de Bruijn indices. I have chosen not to do this for clarity. The second part of the triple is the term. The third part is a stack of values (I use :: as an infix 'push' operator). The evaluation rules for the abstract machine are as follows.

$$
\begin{array}{lcl}
(\langle\Gamma,\mathcal{E}\rangle,\underline{n},S) & \gg & (-,-,\underline{n}::S)\\
(\langle\Gamma,\mathcal{E}\rangle,\mathsf{suc}(M),S) & \gg & (\langle\Gamma,\mathcal{E}\rangle,M,\mathbf{SUC}::S)\\
(-,-,\underline{n}::\mathbf{SUC}::S) & \gg & (-,-,\underline{n+1}::S)\\
(\langle\Gamma\uplus\{x\mapsto v\},\mathcal{E}\rangle,x,S) & \gg & (-,-,v::S)\\
(\langle\Gamma,\mathcal{E}\rangle,\lambda x.M,S) & \gg & (-,-,\mathbf{cl}(\langle\Gamma,\mathcal{E}\rangle,\lambda x.M)::S)\\
(\langle\Gamma,\mathcal{E}\rangle,MN,S) & \gg & (\langle\Gamma,\mathcal{E}\rangle,M,\mathbf{L}::\mathbf{cl}(\langle\Gamma,\mathcal{E}\rangle,N)::S)\\
(-,-,v::\mathbf{L}::\mathbf{cl}(\langle\Gamma,\mathcal{E}\rangle,M)::S) & \gg & (\langle\Gamma,\mathcal{E}\rangle,M,\mathbf{R}::v::S)\\
(-,-,v::\mathbf{R}::\mathbf{cl}(\langle\Gamma,\mathcal{E}\rangle,\lambda x.M)::S) & \gg & (\langle\Gamma\uplus\{x\mapsto v\},\mathcal{E}\rangle,M,S)\\
(\langle\Gamma,\mathcal{E}\rangle,\mu a.M,S) & \gg & (\langle\Gamma,\mathcal{E}\uplus\{a\mapsto S\}\rangle,M,[])\\
(\langle\Gamma,\mathcal{E}\uplus\{a\mapsto S\}\rangle,[a]M,T) & \gg & (\langle\Gamma,\mathcal{E}\uplus\{a\mapsto S\}\rangle,M,S)\\
(\langle\Gamma,\mathcal{E}\rangle,\mathsf{ifz}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P,S) & \gg & (\langle\Gamma,\mathcal{E}\rangle,M,\mathbf{IFZ}::\mathbf{cl}(\langle\Gamma,\mathcal{E}\rangle,\langle N,P\rangle)::S)\\
(-,-,\underline{0}::\mathbf{IFZ}::\mathbf{cl}(\langle\Gamma,\mathcal{E}\rangle,\langle N,P\rangle)::S) & \gg & (\langle\Gamma,\mathcal{E}\rangle,N,S)\\
(-,-,\underline{n+1}::\mathbf{IFZ}::\mathbf{cl}(\langle\Gamma,\mathcal{E}\rangle,\langle N,P\rangle)::S) & \gg & (\langle\Gamma,\mathcal{E}\rangle,P,S)
\end{array}
$$

As the environment is separate from the term, the value of a function is naturally a closure, which has a code part and an environment part. Closures are written $\mathbf{cl}(\langle\Gamma,\mathcal{E}\rangle,M)$.

The feature of this machine (and related environment machines) is that the recursive calls are implemented using the stack and the "markers" (**L**, **R**, **SUC** and **IFZ**).[8] Consequently the stack contains either natural numbers, closures or markers.

Further details of such environment machines and more generally about calculi of closures can be found in Curien's paper [5]. Low level details of implementing other related control

[8] The **L** and **R** markers are not required in a call-by-name setting. The resulting machine in that case is essentially a Krivine machine—a related machine has been given by Selinger [31].

operators are given by Hieb *et al.* [17].

9 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. (As normal the call-by-value function restriction on recursion can be relaxed [33, §11.5].)

$$
\begin{array}{llcl}
\text{Values} & v & ::= & \underline{n} \mid \lambda x.M \mid \langle M, M\rangle \\[1em]
\text{Evaluation Contexts} & E & ::= & \bullet \\
 & & \mid & EM \\
 & & \mid & \mathsf{fst}(E) \mid \mathsf{snd}(E) \\
 & & \mid & \mathsf{suc}(E) \\
 & & \mid & \mathsf{ifz}\ E\ \mathsf{then}\ M\ \mathsf{else}\ M \\[1em]
\text{Redexes} & R & ::= & vM \\
 & & \mid & \mathsf{fst}(v) \mid \mathsf{snd}(v) \\
 & & \mid & \mathsf{suc}(v) \\
 & & \mid & \mathsf{ifz}\ v\ \mathsf{then}\ M\ \mathsf{else}\ M \\
 & & \mid & \mathsf{rec}\ x.M \\
 & & \mid & [a]M \mid \mu a.M
\end{array}
$$

The evaluation rules are essentially as before.

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

The development of the corresponding operational theory follows closely that outlined in §7. The differs from the treatment given by Ong and Stewart [24] who have to introduce completely new reduction rules to move from a call-by-name to a call-by-value setting.

10 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 save and restore the runtime environment. This is maybe not too surprising as Griffin [14] 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 transition system, which eliminated the need for the evaluation contexts. Using this simple transition system it is possible to define a notion of program equivalence based on a termination relation which can be proved to be equivalent to a natural definition of contextual equivalence.

Clearly the work by Ong and Stewart [24] 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 suggest 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.

This paper has given a computational interpretation directly to the $\lambda\mu$-calculus but de Groote [6] has worked in the other direction, using existing work on continuation passing to give an interpretation for the (call-by-name) $\lambda\mu$-calculus. A fuller comparison of these approaches is important future work.

The techniques used in this paper to define a ciu-equivalence and prove a ciu-theorem can be applied to a number of complicated languages.[9] Pitts has used this in recent work on various calculi with explicit state. In unpublished work, the author and Pitts have applied these techniques to both Idealised Scheme [8] and the generalised control language of Gunter *et al.* [15].

Acknowledgements

I am currently supported by EPSRC Grant GR/M04716 and Gonville and Caius College, Cambridge. I am grateful to Nick Benton, Søren Lassen, Luke Ong, Andrew Pitts and Peter Selinger for helpful comments and discussions.

An abridged version of this paper appears in the MFCS proceedings [2].

References

[1] S. Abramsky. The lazy lambda calculus. In D.A. Turner, editor, *Research Topics in Functional Programming*, chapter 4, pages 65–116. Addison-Wesley, 1990.

[2] G.M. Bierman. A computational interpretation of the $\lambda\mu$-calculus. In L. Brim, J. Gruska, and J. Zlatuška, editors, *Proceedings of Symposium on Mathematical Foundations of Computer Science*, volume 1450 of *Lecture Notes in Computer Science*, pages 336–345, August 1998.

[3] R. Cartwright, P.-L. Curien, and M. Felleisen. Fully abstract semantics for observably sequential languages. *Information and Computation*, 111(2):297–401, June 1994.

[4] W. Clinger and J. Rees. The revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, 1986.

[5] P.-L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82(2):389–402, May 1991.

[9] After completing the first draft of this paper, the work of Talcott [32] was brought to my attention. Talcott also proves a ciu-theorem for an untyped variant of Idealised Scheme as well as for a language with explicit memory effects.

[6] P. de Groote. On the relation between the $\lambda\mu$-calculus and the syntactic theory of sequential control. In *Proceedings of Conference on Logic Programming and Automated Reasoning*, volume 822 of *Lecture Notes in Computer Science*, pages 31–43, 1994.

[7] P. de Groote. A simple calculus of exception handling. In *Proceedings of Second International Conference on Typed λ-calculi and applications*, volume 902 of *Lecture Notes in Computer Science*, pages 201–215, 1995.

[8] M. Felleisen and D.P. Friedman. Control operators, the SECD-machine and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 131–141. North-Holland, 1986.

[9] M. Felleisen, D.P. Friedman, E.E. Kohlbecker, and B. Duba. Reasoning with continuations. In *Proceedings of Symposium on Logic in Computer Science*, pages 131–141, June 1986.

[10] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, September 1992.

[11] G. Gentzen. Investigations into logical deduction. In M.E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, 1969. English Translation of 1935 German original.

[12] 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.

[13] A.D. Gordon and A.M. Pitts, editors. *Higher Order Operational Techniques in Semantics*. Publications of the Newton Institute. Cambridge University Press, 1998.

[14] T.G. Griffin. A formulae-as-types notion of control. In *Proceedings of Symposium on Principles of Programming Languages*, pages 47–58, 1990.

[15] C.A. Gunter, D. Rémy, and J.G. Riecke. A generalisation of exceptions and control in ML-like languages. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture*, pages 12–23, 1995.

[16] 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.

[17] R. Hieb, R.K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 66–77, June 1990.

[18] M. Hofmann. Sound and complete axiomatisations of call-by-value control operators. *Mathematical Structures in Computer Science*, 5:461–482, 1995.

[19] M. Hofmann and T. Streicher. Continuation models are universal for $\lambda\mu$-calculus. In *Proceedings of Symposium on Logic in Computer Science*, pages 387–397, 1997.

[20] D.J. Howe. Equality in lazy computation systems. In *Proceedings of Symposium on Logic in Computer Science*, pages 198–203, August 1989.

[21] M. Lillibridge. Exceptions are strictly more powerful than call/cc. Technical Report CMU–CS–95–178, School of Computer Science, Carnegie Mellon University, July 1995.

[22] A.R. Meyer and J.G. Riecke. Continuations may be unreasonable (Preliminary Report). In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 63–71, July 1988.

[23] C.-H.L. Ong. A semantic view of classical proofs: type-theoretic, categorical and denotational characterizations. In *Proceedings of Symposium on Logic in Computer Science*, pages 230–241, 1996.

[24] C.-H.L. Ong and C.A. Stewart. A Curry-Howard foundation for functional computation with control. In *Proceedings of Symposium on Principles of Programming Languages*, pages 215–227, 1997.

[25] M. Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proceedings of Conference on Logic Programming and Automated Reasoning*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201, 1992.

[26] M. Parigot. Proofs of strong normalisation for second order classical natural deduction. *Journal of Symbolic Logic*, 62(4):1461–1479, December 1997.

[27] A.M. Pitts. Operational semantics for program equivalence. Slides from talk given at MFPS, 1997.

[28] A.M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A.M. Pitts, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 241–298. Cambridge University Press, 1997.

[29] A.M. Pitts and I.D.B. Stark. Operational reasoning for functions with local state. In A.D. Gordon and A.M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 227–273. Cambridge University Press, 1998.

[30] P. Selinger. Control categories: an axiomatic approach to the semantics of control in functional languages. Unpublished manuscript, May 1998.

[31] P. Selinger. An implementation of the call-by-name $\lambda\mu\nu$-calculus. Unpublished manuscript, July 1998.

[32] C. Talcott. Reasoning about functions with effects. In A.D. Gordon and A.M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 347–390. Cambridge University Press, 1998.

[33] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

A The ciu-theorem

In this appendix, I shall give details of the proof of the 'ciu-theorem': the proof that contextual equivalence coincides with ciu-equivalence. First recall these two definitions of program equivalence.

Definition 3. *Let M and N be terms and $\mathcal{C}$ a λ-closing context. M is said to* contextually refine *N, written $\Gamma \rhd M \sqsubseteq N\colon \phi, \Sigma$, when $\forall \mathcal{C}, \mathcal{E}.$ if $(\mathcal{C}[M], \mathcal{E}) \Downarrow$ then $(\mathcal{C}[N], \mathcal{E}) \Downarrow$. They are said to be* contextually equivalent, *written $\Gamma \rhd M \approx N\colon \phi, \Sigma$, just when $\Gamma \rhd M \sqsubseteq N\colon \phi, \Sigma$ iff $\Gamma \rhd N \sqsubseteq M\colon \phi, \Sigma$.*

Definition 4. *Given two programs M and N, M is said to* ciu-refine *N, written $M \leq N\colon \phi, \Sigma$, 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 $M \simeq N\colon \phi, \Sigma$ just when $M \leq N\colon \phi, \Sigma$ and $M \leq N\colon \phi, \Sigma$.*

These can be extended to open terms as follows

$$\begin{array}{rcl} \vec{x}\colon \Gamma \rhd M \leq^\circ N\colon \phi, \Sigma & \stackrel{\text{def}}{=} & \forall \vec{v}. M[\vec{x} := \vec{v}] \leq N[\vec{x} := \vec{v}]\colon \phi, \Sigma \\ \vec{x}\colon \Gamma \rhd M \simeq^\circ N\colon \phi, \Sigma & \stackrel{\text{def}}{=} & \forall \vec{v}. M[\vec{x} := \vec{v}] \simeq N[\vec{x} := \vec{v}]\colon \phi, \Sigma \end{array}$$

Two facts are almost immediate from the definition of ciu-refinement.

Lemma 2.

1. *$\forall M. M \leq M\colon \phi, \Sigma$.*
2. *If $M \leq M'\colon \phi, \Sigma$ and $M' \leq M''\colon \phi, \Sigma$ then $M \leq M''\colon \phi, \Sigma$.*

The following properties of relations between terms will be useful.

Definition 5. *A relation, $\mathcal{R}$, is said to be* compatible *if it satisfies the following rules.*

$$\frac{}{\Gamma, x\colon \phi \rhd x \;\mathcal{R}\; x\colon \phi, \Sigma} \qquad \frac{}{\Gamma \rhd \underline{n} \;\mathcal{R}\; \underline{n}\colon \mathtt{int}, \Sigma}$$

$$\frac{\Gamma \rhd M \;\mathcal{R}\; N\colon \phi, \Sigma}{\Gamma, x\colon \psi \rhd M \;\mathcal{R}\; N\colon \phi, \Sigma}\,\text{Weakening}_{\mathcal{L}} \qquad \frac{\Gamma \rhd M \;\mathcal{R}\; N\colon \phi, \Sigma}{\Gamma \rhd M \;\mathcal{R}\; N\colon \phi, \Sigma, a\colon \psi}\,\text{Weakening}_{\mathcal{R}}$$

$$\frac{\Gamma, x\colon \phi \rhd M \;\mathcal{R}\; M'\colon \psi, \Sigma}{\Gamma \rhd \lambda x.M \;\mathcal{R}\; \lambda x.M'\colon \phi \to \psi, \Sigma} \to_{\mathcal{I}}$$

$$\frac{\Gamma \rhd M \;\mathcal{R}\; N\colon \phi \to \psi, \Sigma \qquad \Gamma \rhd M' \;\mathcal{R}\; N'\colon \phi, \Sigma}{\Gamma \rhd MM' \;\mathcal{R}\; NN'\colon \psi, \Sigma} \to_{\mathcal{E}}$$

$$\frac{\Gamma \rhd M \;\mathcal{R}\; M'\colon \phi, \Sigma \qquad \Gamma \rhd N \;\mathcal{R}\; N'\colon \psi, \Sigma}{\Gamma \rhd \langle M, N\rangle \;\mathcal{R}\; \langle M', N'\rangle\colon \phi \times \psi, \Sigma} \times_{\mathcal{I}}$$

$$\frac{\Gamma \rhd M \;\mathcal{R}\; M'\colon \phi \times \psi, \Sigma}{\Gamma \rhd \mathsf{fst}(M) \;\mathcal{R}\; \mathsf{fst}(M')\colon \phi, \Sigma} \times_{\mathcal{E}} \qquad \frac{\Gamma \rhd M \;\mathcal{R}\; M'\colon \phi \times \psi, \Sigma}{\Gamma \rhd \mathsf{snd}(M) \;\mathcal{R}\; \mathsf{snd}(M')\colon \psi, \Sigma} \times_{\mathcal{E}}$$

$$\frac{\Gamma \triangleright M \;\mathcal{R}\; M' \colon \mathtt{int}, \Sigma}{\Gamma \triangleright \mathsf{suc}(M) \;\mathcal{R}\; \mathsf{suc}(M') \colon \mathtt{int}, \Sigma}\ \text{Suc}$$

$$\frac{\Gamma \triangleright M \;\mathcal{R}\; M' \colon \mathtt{int}, \Sigma \qquad \Gamma \triangleright N \;\mathcal{R}\; N' \colon \phi, \Sigma \qquad \Gamma \triangleright P \;\mathcal{R}\; P' \colon \phi, \Sigma}{\Gamma \triangleright \mathsf{ifz}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P \;\mathcal{R}\; \mathsf{ifz}\ M'\ \mathsf{then}\ N'\ \mathsf{else}\ P' \colon \phi, \Sigma}\ \text{Cond}$$

$$\frac{\Gamma, f\colon \phi \to \phi, x\colon \phi \triangleright M \;\mathcal{R}\; M' \colon \phi, \Sigma \qquad \Gamma, f\colon \phi \to \phi \triangleright N \;\mathcal{R}\; N' \colon \phi, \Sigma}{\Gamma \triangleright \mathsf{letrec}\ f = \lambda x.M\ \mathsf{in}\ N \;\mathcal{R}\; \mathsf{letrec}\ f = \lambda x.M'\ \mathsf{in}\ N' \colon \phi, \Sigma}\ \text{Recursion}$$

$$\frac{\Gamma \triangleright M \;\mathcal{R}\; N \colon \phi, \Sigma}{\Gamma \triangleright [a\colon \phi]M \;\mathcal{R}\; [a\colon \phi]N \colon \bot, \Sigma, a\colon \phi}\ \text{Passivate}$$

$$\frac{\Gamma \triangleright M \;\mathcal{R}\; N \colon \bot, \Sigma, a\colon \phi}{\Gamma \triangleright \mu a\colon \phi.M \;\mathcal{R}\; \mu a\colon \phi.N \colon \phi, \Sigma}\ \text{Activate}$$

A precongruence *is a compatible relation which is also transitive. A* congruence *is a precongruence which is also symmetric.*

It is easy to see that a compatible relation is reflexive. Another important property is the following.

Lemma 3. *If $\mathcal{R}$ is a precongruence and $\Gamma, \Gamma' \triangleright M \;\mathcal{R}\; N \colon \phi, \Sigma, \Sigma'$. Then for any context $\mathcal{C}[\bullet]$, $\Gamma \triangleright \mathcal{C}[M] \;\mathcal{R}\; \mathcal{C}[N] \colon \psi, \Sigma$.*

Proof. By induction over the structure of the context. ■

We should like to prove that ciu-refinement is a precongruence. As is usual (e.g. for pure PCF), this is extremely difficult to prove directly. Fortunately Howe [20] has given an ingenious method for proving (pre)congruences of PCF-like languages. The trick is to give another relation (which will be written $\leq^\star$), which is almost trivially compatible and rather less trivially coincides with ciu-refinement. As ciu-refinement is transitive, this is enough to show that it is a precongruence. Howe's method will be adopted here, although it has to be extended to handle the save and restore features of μPCF. First, the definition of the $\leq^\star$ relation.

Definition 6.

$$\begin{array}{rcl}
\Gamma \triangleright \underline{n} \leq^\star N \colon \mathtt{int}, \Sigma & \stackrel{\text{def}}{=} & \Gamma \triangleright \underline{n} \leq^\circ N \colon \mathtt{int}, \Sigma \\
\Gamma, x\colon \phi \triangleright x \leq^\star N \colon \phi, \Sigma & \stackrel{\text{def}}{=} & \Gamma, x\colon \phi \triangleright x \leq^\circ N \colon \phi, \Sigma \\
\Gamma \triangleright \mathsf{suc}(M) \leq^\star N \colon \mathtt{int}, \Sigma & \stackrel{\text{def}}{=} & \exists P.\, \Gamma \triangleright M \leq^\star P \colon \mathtt{int}, \Sigma \\
 & & \Gamma \triangleright \mathsf{suc}(P) \leq^\circ N \colon \mathtt{int}, \Sigma \\
\Gamma \triangleright \lambda x\colon \phi.M \leq^\star N \colon \phi \to \psi, \Sigma & \stackrel{\text{def}}{=} & \exists P.\, \Gamma, x\colon \phi \triangleright M \leq^\star P \colon \psi, \Sigma \\
 & & \Gamma \triangleright \lambda x\colon \phi.P \leq^\circ N \colon \phi \to \psi, \Sigma \\
\Gamma \triangleright M M' \leq^\star N \colon \psi, \Sigma & \stackrel{\text{def}}{=} & \exists P, P'.\, \Gamma \triangleright M \leq^\star P \colon \phi \to \psi, \Sigma \\
 & & \Gamma \triangleright M' \leq^\star P' \colon \phi, \Sigma \\
 & & \Gamma \triangleright P P' \leq^\circ N \colon \psi, \Sigma \\
\Gamma \triangleright \mathsf{ifz}\ M\ \mathsf{then}\ M'\ \mathsf{else}\ M'' \leq^\star N \colon \phi, \Sigma & \stackrel{\text{def}}{=} & \exists P, P', P''.\, \Gamma \triangleright M \leq^\star P \colon \mathtt{int}, \Sigma \\
 & & \Gamma \triangleright M' \leq^\star P' \colon \phi, \Sigma \\
 & & \Gamma \triangleright M'' \leq^\star P'' \colon \phi, \Sigma \\
 & & \Gamma \triangleright \mathsf{ifz}\ P\ \mathsf{then}\ P'\ \mathsf{else}\ P'' \leq^\circ N \colon \phi, \Sigma
\end{array}$$

$$
\begin{array}{rcl}
\Gamma \rhd \langle M, M' \rangle \leq^\star N\colon \phi \times \psi, \Sigma & \stackrel{\text{def}}{=} & \exists P, P'.\, \Gamma \rhd M \leq^\star P\colon \phi, \Sigma \\
& & \Gamma \rhd M' \leq^\star P'\colon \psi, \Sigma \\
& & \Gamma \rhd \langle P, P' \rangle \leq^\circ N\colon \phi \times \psi, \Sigma \\
\Gamma \rhd \mathsf{fst}(M) \leq^\star N\colon \phi, \Sigma & \stackrel{\text{def}}{=} & \exists P.\, \Gamma \rhd M \leq^\star P\colon \phi \times \psi, \Sigma \\
& & \Gamma \rhd \mathsf{fst}(P) \leq^\circ N\colon \phi, \Sigma \\
\Gamma \rhd \mathsf{snd}(M) \leq^\star N\colon \psi, \Sigma & \stackrel{\text{def}}{=} & \exists P.\, \Gamma \rhd M \leq^\star P\colon \phi \times \psi, \Sigma \\
& & \Gamma \rhd \mathsf{snd}(P) \leq^\circ N\colon \psi, \Sigma \\
\Gamma \rhd \mathsf{letrec}\ f = \lambda x.M\ \mathsf{in}\ M' \leq^\star N\colon \psi, \Sigma & \stackrel{\text{def}}{=} & \exists P, P'.\, \Gamma, f\colon \phi \to \phi, x\colon \phi \rhd M \leq^\star P\colon \phi, \Sigma \\
& & \Gamma, f\colon \phi \to \phi \rhd M' \leq^\star P'\colon \psi, \Sigma \\
& & \Gamma \rhd \mathsf{letrec}\ f = \lambda x.P\ \mathsf{in}\ P' \leq^\circ N\colon \psi, \Sigma \\
\Gamma \rhd \mu a\colon \phi.M \leq^\star N\colon \phi, \Sigma & \stackrel{\text{def}}{=} & \exists P.\, \Gamma \rhd M \leq^\star P\colon \bot, \Sigma, a\colon \phi \\
& & \Gamma \rhd \mu a\colon \phi.P \leq^\circ N\colon \phi, \Sigma \\
\Gamma \rhd [a\colon \phi]M \leq^\star N\colon \bot, a\colon \phi, \Sigma & \stackrel{\text{def}}{=} & \exists P.\, \Gamma \rhd M \leq^\star P\colon \phi, \Sigma \\
& & \Gamma \rhd [a\colon \phi]P \leq^\circ N\colon \bot, a\colon \phi, \Sigma
\end{array}
$$

This relation can be extended to frames, stacks of frames and functions from μ-variables to stacks. This is important for technical reasons. The definitions are as follows.

Definition 7.

1.

$$
\begin{array}{rcl}
\bullet M \leq^\star F[\bullet]\colon \psi, \Sigma & \stackrel{\text{def}}{=} & \exists N.M \leq^\star N\colon \phi, \Sigma \\
& & \forall P.PN \leq F[P]\colon \psi, \Sigma \\
v\bullet \leq^\star F[\bullet]\colon \psi, \Sigma & \stackrel{\text{def}}{=} & \exists N.v \leq^\star N\colon \phi \to \psi, \Sigma \\
& & \forall P.NP \leq F[P]\colon \psi, \Sigma \\
\langle \bullet, M \rangle \leq^\star F[\bullet]\colon \phi \times \psi, \Sigma & \stackrel{\text{def}}{=} & \exists N.M \leq^\star N\colon \psi, \Sigma \\
& & \forall P.\langle P, N \rangle \leq F[P]\colon \phi \times \psi, \Sigma \\
\langle v, \bullet \rangle \leq^\star F[\bullet]\colon \phi \times \psi, \Sigma & \stackrel{\text{def}}{=} & \exists N.v \leq^\star N\colon \phi, \Sigma \\
& & \forall P.\langle v, P \rangle \leq F[P]\colon \phi \times \psi, \Sigma \\
\mathsf{fst}(\ \bullet\ ) \leq^\star F[\bullet]\colon \psi, \Sigma & \stackrel{\text{def}}{=} & \forall P.\mathsf{fst}(P) \leq F[P]\colon \psi, \Sigma \\
\mathsf{snd}(\ \bullet\ ) \leq^\star F[\bullet]\colon \psi, \Sigma & \stackrel{\text{def}}{=} & \forall P.\mathsf{snd}(P) \leq F[P]\colon \psi, \Sigma \\
\mathsf{suc}(\bullet) \leq^\star F[\bullet]\colon \mathtt{int}, \Sigma & \stackrel{\text{def}}{=} & \forall P.\mathsf{suc}(P) \leq F[P]\colon \mathtt{int}, \Sigma \\
\mathsf{ifz}\ \bullet\ \mathsf{then}\ M\ \mathsf{else}\ M' \leq^\star F[\bullet]\colon \psi, \Sigma & \stackrel{\text{def}}{=} & \exists N, N'.\, M \leq^\star N\colon \psi, \Sigma \\
& & M' \leq^\star N'\colon \psi, \Sigma \\
& & \forall P.\ \mathsf{ifz}\ P\ \mathsf{then}\ N\ \mathsf{else}\ N' \leq F[P]\colon \psi, \Sigma
\end{array}
$$

2.

$$
[] \leq^\star [] \qquad \frac{F[\bullet] \leq^\star F'[\bullet] \qquad S \leq^\star S'}{(F[\bullet] :: S) \leq^\star (F'[\bullet] :: S')}
$$

3.

$$
\mathcal{E} \leq^\star \mathcal{E}' \text{ iff } \forall a.\mathcal{E}a \leq^\star \mathcal{E}'a.
$$

It is fairly easy to verify that all these relations are reflexive.

Lemma 4. (Reflexivity)

1. $\forall M.\Gamma \rhd M \leq^\star M\colon \phi, \Sigma.$

2. $\forall F[\bullet]. F[\bullet] \leq^\star F[\bullet]\colon \phi, \Sigma$.
3. $\forall S. S \leq^\star S$.
4. $\forall \mathcal{E}. \mathcal{E} \leq^\star \mathcal{E}$.

It is not hard to see that the $\leq^\star$ relation is *not* transitive. However the following property will suffice.

Lemma 5. *If $M \leq^\star M'\colon \phi, \Sigma$ and $M' \leq M''\colon \phi, \Sigma$ then $M \leq^\star M''\colon \phi, \Sigma$.*

Proof. By induction on $M \leq^\star M'\colon \phi, \Sigma$. ■

One half of the coincidence of the $\leq^\star$ relation and ciu-refinement is now immediate.

Proposition 2. *If $M \leq N\colon \phi, \Sigma$ then $M \leq^\star N\colon \phi, \Sigma$.*

Proof. We know that $M \leq^\star M\colon \phi, \Sigma$ and have that $M \leq N\colon \phi, \Sigma$. Hence by lemma 5 we conclude that $M \leq^\star N\colon \phi, \Sigma$. ■

The following two properties will be useful.

Lemma 6.

1. *If $v \leq^\star v'\colon \phi, \Sigma$ and $\Gamma, x\colon \phi \rhd M \leq^\star M'\colon \psi, \Sigma$ then $\Gamma \rhd M[x := v] \leq^\star M'[x := v']\colon \psi, \Sigma$.*
2. *If $v \leq^\star M\colon \phi, \Sigma$ then $\exists w. v \leq^\star w\colon \phi, \Sigma$ and $w \leq M\colon \phi, \Sigma$.*

Proof. Part 1 follows by induction on M. Part 2 by induction on v. ■

The following property relates the $\leq^\star$ relation and the termination relation.

Proposition 3. *If $S \leq^\star S'$, $M \leq^\star M'\colon \phi, \Sigma$, $\mathcal{E} \leq^\star \mathcal{E}'$ and $(S, M, \mathcal{E})\searrow$ then $(S', M', \mathcal{E}')\searrow$.*

Proof. By induction on depth of $(S, M, \mathcal{E})\searrow$. ■

Corollary 1. *If $(S, M, \mathcal{E})\searrow$ and $M \leq^\star N\colon \phi, \Sigma$ then $(S, N, \mathcal{E})\searrow$.*

We can now show the other direction of the coincidence of the $\leq^\star$ relation and ciu-refinement (cf. Proposition 2)

Proposition 4. *If $M \leq^\star N\colon \phi, \Sigma$ then $M \leq N\colon \phi, \Sigma$.*

Proof. Assume that $M \leq^\star N\colon \phi, \Sigma$ and that $(S, M, \mathcal{E})\searrow$. Then by corollary 1, it follows that $(S, N, \mathcal{E})\searrow$. ■

As the $\leq^\star$ relation is trivially compatible, Propositions 2 and 4 allow one to conclude that ciu-refinement is also compatible. In addition, ciu-refinement is transitive (Lemma 2), which allows one to conclude that it is a precongruence.

Proposition 5. *$\leq$ is a precongruence.*

This now allows us to conclude that ciu-refinement is included in contextual refinement.

Proposition 6. *If $M \leq N\colon \phi, \Sigma$ then $M \sqsubseteq N\colon \phi, \Sigma$.*

Proof. By Proposition 5 we have that $\mathcal{C}[M] \leq \mathcal{C}[N]\colon \psi$ which is sufficient. ∎

An important property of the termination relation and evaluation contexts is the following.

Lemma 7. *$(S, E[M], \mathcal{E})\searrow$ iff $(\lceil E[\bullet] \rceil :: S, M, \mathcal{E})\searrow$.*

Proof. By induction on the structure of $E[\bullet]$. ∎

Evaluation contexts and frame stacks can be seen to be in one-to-one correspondence as follows.

Lemma 8.

1. *$\forall E[\bullet], \exists S$ such that $S = \lceil E[\bullet] \rceil$.*
2. *$\forall S, \exists E[\bullet]$ such that $S = \lceil E[\bullet] \rceil$.*

We can now conclude that contextual refinement is included in ciu-refinement.

Proposition 7. *If $M \sqsubseteq N\colon \phi, \Sigma$ then $M \leq N\colon \phi, \Sigma$.*

Proof. By assumption we have that if $([], \mathcal{C}[M], \mathcal{E})\searrow$ then $([], \mathcal{C}[N], \mathcal{E})\searrow$. In particular we can take only the contexts which are evaluation contexts; thus, if $([], E[M], \mathcal{E})\searrow$ then $([], E[N], \mathcal{E})\searrow$. From Lemma 7 we have that if $(\lceil E[\bullet] \rceil, M, \mathcal{E})\searrow$ then $(\lceil E[\bullet] \rceil, N, \mathcal{E})\searrow$. From Lemma 8 we are done. ∎

Thus we have proved the ciu-theorem.

Theorem 2. *$\Gamma \rhd M \sqsubseteq N\colon \phi, \Sigma$ iff $\Gamma \rhd M \leq^{\circ} N\colon \phi, \Sigma$.*

Proof. The λ-closed instances follow from Propositions 6 and 7. It is relatively straightforward to extend these to the open versions. ∎