A Curry-Howard foundation for functional computation with control

C.-H. L. Ong* C. A. Stewart†

Oxford University Computing Laboratory

Abstract

We introduce the type theory $\boldsymbol{\lambda\mu}_{\mathrm{v}}$, a call-by-value variant of Parigot's $\lambda\mu$-calculus, as a Curry-Howard representation theory of classical propositional proofs. The associated rewrite system is Church-Rosser and strongly normalizing, and definitional equality of the type theory is consistent, compatible with cut, congruent and decidable. The attendant call-by-value programming language $\mu\mathrm{PCF}_{\mathrm{v}}$ is obtained from $\boldsymbol{\lambda\mu}_{\mathrm{v}}$ by augmenting it by basic arithmetic, conditionals and fixpoints. We study the behavioural properties of $\mu\mathrm{PCF}_{\mathrm{v}}$ and show that, though simple, it is a very general language for functional computation with control: it can express all the main control constructs such as exceptions and first-class continuations. Proof-theoretically the dual $\boldsymbol{\lambda\mu}_{\mathrm{v}}$-constructs of naming and μ-abstraction witness the introduction and elimination rules of absurdity respectively. Computationally they give succinct expression to a kind of generic (forward) "jump" operator, which may be regarded as a unifying control construct for functional computation. Our goal is that $\boldsymbol{\lambda\mu}_{\mathrm{v}}$ and $\mu\mathrm{PCF}_{\mathrm{v}}$ respectively should be to functional computation with first-class access to the flow of control what λ-calculus and PCF respectively are to pure functional programming: $\boldsymbol{\lambda\mu}_{\mathrm{v}}$ gives the logical basis via the Curry-Howard correspondence, and $\mu\mathrm{PCF}_{\mathrm{v}}$ is a prototypical language albeit in purified form.

1 Introduction

Most modern functional languages provide powerful operations for altering the normal flow of control. Exceptions and continuations are the two most common examples. Both mechanisms, particularly exceptions, are useful: ML-style exceptions are an efficient, elegant and systematic way to recover from "errors"; and continuations can be used to implement a host of sophisticated control features such as coroutines [19], multiple threads of control [43], logic variables [18], and even concurrent constructs [35].

Early theoretical work in the area is mainly concerned with developing reasoning principles for functional programs with control constructs based on a minimal programming calculus. The research of Felleisen and his coworkers [11, 12] (see also [41]) exemplify this type-free, operational approach. They introduce a simple untyped language $\boldsymbol{\lambda\mathcal{C}}$, which is Plotkin's call-by-value (CBV) λ-calculus [32] augmented by a unary operator $\mathcal{C}$, and study its rewrite properties with a view to deriving a well-behaved theory of contextual or observational equivalence. Others seek to understand control constructs by program (including CPS-) transformation e.g. [1, 8], and in an ML-style polymorphically typed regime e.g. [17].

In an influential paper [15] that opened a new and rich line of enquiry, Griffin put forth the remarkable idea that the well-known Curry-Howard correspondence linking intuitionistic natural deduction proofs with functional programs *can* be extended to classical logic provided one augments functions by appropriate control constructs. In particular he proposed the tautology $\neg\neg A \Rightarrow A$ as the type for Felleisen's $\mathcal{C}$-operator. A spate of research into the semantics and computational contents of classical proofs ensued (some of which quite independently of Griffin's): [6, 13, 25, 28, 2, 21, 4, 7, 42], etc.

Church's λ-calculus is by now widely accepted as the logical basis of functional programming. A goal of our research is to find the "λ-calculus" of functional computation with first-class access to the flow of control, or functional computation with control, for short. In **Sec-**

*Luke.Ong@comlab.ox.ac.uk

†Charles.Stewart@comlab.ox.ac.uk

To appear in *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, January 15-17, 1997.* ACM Press.

tion 2 of this paper we introduce $\boldsymbol{\lambda\mu}_{\mathrm{v}}$, a CBV[1] variant of Parigot's $\lambda\mu$-calculus, as a Curry-Howard style representation theory of classical propositional proofs. Viewed as a rewrite system, reduction in the type theory $\boldsymbol{\lambda\mu}_{\mathrm{v}}$ (corresponding to cut elimination) is Church-Rosser and strongly normalizing, thus giving rise to an intrinsic notion of *equality* of classical proofs that is consistent, compatible with cut, congruent (i.e. compatible with term formation rules), and decidable. To our knowledge this is the first *confluent* $\lambda\mu$-calculus that admits a kind of "symmetric" structural-rules (i.e. what we call (ζ_{fun}) and (ζ_{arg})) in the sense of Parigot [28], and which satisfies the *uniqueness of data representation* property (i.e. the only normal forms of the type of natural numbers are the Church numerals). This property does not hold for the call-by-name $\lambda\mu$-calculus.

It is folklore that Scott's PCF [37, 33], that "mother of all toy languages" after Pitts' memorable turn of phrase, is a prototypical functional language. Another goal of our research is to develop a theory of functional computation with control based on a simple but expressive language; and then to give a precise and systematic account of its connection with classical proof theory. To this end we introduce, in **Section 3**, $\mu\mathrm{PCF}_{\mathrm{v}}$ (and its $\bot$-free subsystem $\mu\mathrm{PCF}_{\mathrm{v}}^{-}$), which is obtained from the simply-typed $\boldsymbol{\lambda\mu}_{\mathrm{v}}$ by augmenting it by basic arithmetic, conditionals and fixpoint operators, and study its properties as a CBV programming language equipped with a sequential and deterministic reduction strategy. For such a language to be useful for foundational analysis of the kind we envisage, several criteria may be identified:

- it should be conceptually simple and syntactically economical, and yet expressively rich
- it should have a sound logical basis
- it should have a well understood model theory.

Our aim is that $\boldsymbol{\lambda\mu}_{\mathrm{v}}$ and $\mu\mathrm{PCF}_{\mathrm{v}}$ should provide appropriate logical and computational foundations[2] respectively for functional computation with control, in much the same way as the λ-calculus and PCF respectively do for pure functional computation.

Previous work The CBN version of $\boldsymbol{\lambda\mu}$, which we shall henceforth call $\boldsymbol{\lambda\mu}_{\mathrm{n}}$, has been presented in [27] which is a purely proof-theoretic study. The work reported here sets out the computational relevance of the $\boldsymbol{\lambda\mu}$ approach, and should be regarded as a sequel to [27]. A reading of the first two sections of [27] is probably necessary for a proper understanding of section 2 of this paper.

[1] The adjective "call-by-value" qualifies only the β-redex rule and *not* the reduction strategy.

[2] The $\boldsymbol{\lambda\mu}$ approach works equally well for both the CBN and CBV regimes. In this paper we choose to focus on the latter (and on $\mu\mathrm{PCF}_{\mathrm{v}}$ instead of $\mu\mathrm{PCF}_{\mathrm{n}}$), which is the more complex of the two from the viewpoint of reasoning about flow of control.

2 Classical proof semantics and $\boldsymbol{\lambda\mu}_{\mathrm{v}}$

Parigot's $\lambda\mu$-calculus [28] is a natural deduction (formation rules take the form of introduction and elimination of logical connectives) system of classical proofs but formulated in a sequent style. In this section we shall focus on $\boldsymbol{\lambda\mu}_{\mathrm{v}}$, a CBV variant[3] of Parigot's system for the propositional $\{\bot, \Rightarrow\}$-fragment but presented as a type theory.

The type theory $\boldsymbol{\lambda\mu}_{\mathrm{v}}$ has two kinds of judgement:

- *term assignment*: $\quad \Gamma;\ \Delta \vdash s : A$
- *term equality*: $\quad \Gamma;\ \Delta \vdash s = t : A$

where Γ, Γ', etc., range over λ-contexts, and Δ, Δ', etc., over μ-contexts. **Types** of the system, ranged over by A, B, etc., are built up by the constructor $\Rightarrow$ from a set of atomic types a, b, etc., including a distinguished base type $\bot$ called **absurdity**. Standardly we use $\neg A$ as a shorthand for $A \Rightarrow \bot$.

There are two kinds of "variable" in $\boldsymbol{\lambda\mu}_{\mathrm{v}}$. The first is just the usual λ-*variable* or simply **variable**. There are denumerably many variables x^A, x_1^A, etc., for each type A, and the associated binding operator is the λ-abstraction $\boldsymbol{\lambda} x^A.\bot$. We shall refer to the other kind of variable, ranged over by $\alpha^A, \beta^B, \gamma^C$, etc., as μ-*names*, or simply **names**, in the sense of the π-calculus [23]. There are denumerably many names $\alpha^A, \alpha_1^A, \alpha_2^A$, etc., for each non-$\bot$ type A; but it is a crucial feature of our $\boldsymbol{\lambda\mu}_{\mathrm{v}}$ (as opposed to Parigot's $\lambda\mu$-calculus) that there is *no* name of type $\bot$. In addition, in contrast to variable, the only thing that can be substituted for a name is a name. The associated binding operator is the μ-abstractor $\boldsymbol{\mu}\alpha^A.\bot$. Corresponding to the two kinds of "variable", there are two kinds of context: λ-*context* and μ-*context*. We find it convenient to define a λ-context as a finite sequence of "type-variable" pairs, written A^x as opposed to $x : A$, in which no variable may occur more than once; similarly for μ-context. A λ-context (respectively μ-context) has the typical form

$$A^x, A^{x'}, B^y, \cdots, C^z \quad (\text{resp. } A^\alpha, B^\beta, B^{\beta'}, \cdots, C^\gamma)$$

Note that $\bot$ can never occur in a μ-context. *Raw* $\boldsymbol{\lambda\mu}_{\mathrm{v}}$-*terms*, ranged over by s, t, u, etc., are defined in Figure 1.

The rules defining the **term assignment judgements** are displayed in Table 1. Observe the symmetry

[3] We refer the reader to [27] for an analysis of the differences between our system and Parigot's. We shall distinguish the two by referring to the latter as $\lambda\mu$-*calculus* (Greek letters in light font), as opposed to our $\boldsymbol{\lambda\mu}_{\mathrm{v}}$ and $\boldsymbol{\lambda\mu}_{\mathrm{n}}$.

$$(\lambda\textbf{-perm})\quad \frac{\Gamma, A^x, B^y, \Gamma';\ \Delta \vdash s : C}{\Gamma, B^y, A^x, \Gamma';\ \Delta \vdash s : C} \qquad (\mu\textbf{-perm})\quad \frac{\Gamma;\ \Delta, A^\alpha, B^\beta, \Delta' \vdash s : C}{\Gamma;\ \Delta, B^\beta, A^\alpha, \Delta' \vdash s : C}$$

$$(\lambda\textbf{-wk})\quad \frac{\Gamma;\ \Delta \vdash s : C}{\Gamma, A^x;\ \Delta \vdash s : C}\ A^x \notin \Gamma \qquad (\mu\textbf{-wk})\quad \frac{\Gamma;\ \Delta \vdash s : C}{\Gamma;\ \Delta, A^\alpha \vdash s : C}\ A^\alpha \notin \Delta$$

$$(\lambda\textbf{-ctr})\quad \frac{\Gamma, A^{x_1}, A^{x_2};\ \Delta \vdash s : C}{\Gamma, A^x;\ \Delta \vdash s[x/x_1][x/x_2] : C}\ A^x \notin \Gamma \qquad (\mu\textbf{-ctr})\quad \frac{\Gamma;\ \Delta, A^{\alpha_1}, A^{\alpha_2} \vdash s : C}{\Gamma;\ \Delta, A^\alpha \vdash s[\alpha/\alpha_1][\alpha/\alpha_2] : C}\ A^\alpha \notin \Delta$$

$$(\textbf{axiom})\quad A^x;\ \vdash x : A$$

$$(\Rightarrow\textbf{-intro})\quad \frac{\Gamma, A^x;\ \Delta \vdash s : B}{\Gamma;\ \Delta \vdash \boldsymbol{\lambda} x^A.s : A \Rightarrow B} \qquad (\Rightarrow\textbf{-elim})\quad \frac{\Gamma;\ \Delta \vdash s : A \Rightarrow B \qquad \Gamma;\ \Delta \vdash t : A}{\Gamma;\ \Delta \vdash st : B}$$

$$(\bot\textbf{-intro})\quad \frac{\Gamma;\ \Delta \vdash s : A}{\Gamma;\ \Delta, A^\alpha \vdash [\alpha^A]s : \bot}\ A \neq \bot, A^\alpha \notin \Delta \qquad (\bot\textbf{-elim})\quad \frac{\Gamma;\ \Delta, B^\beta \vdash s : \bot}{\Gamma;\ \Delta \vdash \boldsymbol{\mu}\beta^B.s : B}$$

Table 1: Rules defining term assignment judgements of $\boldsymbol{\lambda\mu}_{\mathrm{v}}$.

s	$:=$	x	variable
	$\mid$	$\boldsymbol{\lambda} x^A.s$	λ-abstraction
	$\mid$	st	application
	$\mid$	$[\alpha^A]s$	*named-term*
	$\mid$	$\boldsymbol{\mu}\beta^B.s$	*μ-abstraction.*

Figure 1: Terms of $\boldsymbol{\lambda\mu}_{\mathrm{v}}$.

between the structural rules for λ-contexts and those for μ-contexts. A $\boldsymbol{\lambda\mu}_{\mathrm{v}}$-term, or simply **term**, is a well-typed raw term. In the following we shall often write s^A to mean that s is of type A. Note that a well-typed named-term $[\alpha^A]s$ is necessarily of type $\bot$, and s has type A; a well-typed μ-abstraction of the form $\boldsymbol{\mu}\beta^B.s$ is of type B, and s must be of type $\bot$.

How should one think of $\boldsymbol{\lambda\mu}_{\mathrm{v}}$? As a first approximation, there is no harm in reading the $\boldsymbol{\lambda\mu}_{\mathrm{v}}$-sequent $C;\ D \vdash s : A$ as "$C \wedge \neg D \Rightarrow A$", and accordingly, the rule ($\bot$-**elim**) would correspond to the classical absurdity rule

$$\frac{\begin{array}{c}[\neg A]\\ \vdots \\ \bot\end{array}}{A}$$

and ($\bot$-**intro**) to implication elimination-rule whose hypotheses have the forms $A \Rightarrow \bot$ and A respectively. In fact a correspondence between $\boldsymbol{\lambda\mu}_{\mathrm{n}}$ and Gentzen's natural deduction for classical logic with four rules, namely, axiom, $\Rightarrow$-introduction, $\Rightarrow$-elimination, and classical absurdity, can be established (see [30] or [27] for an account); the rule (ζ_{fun}) then corresponds to a rule of Prawitz [34] that "pushes" double negation of an implicational formula into its components. Computationally the dual constructs of μ-abstraction $\boldsymbol{\mu}\alpha.\bot$ and naming $[\beta](\bot)$ (which witness the elimination and introduction of absurdity respectively) give expression to a kind of generic jump operator. See the discussion after Proposition 3.2 for an explanation.

The least that we should require from any formal presentation of classical proofs is *completeness for provability* in the following sense.

Proposition 2.1 (Parigot) *A is a tautology if and only if A is inhabited in $\boldsymbol{\lambda\mu}_{\mathrm{v}}$ i.e. for some closed term (no free variables nor names) s, the sequent "$;\ \vdash s : A$" is derivable.* □

Example 2.2 (i) The term

$$\boldsymbol{\kappa} \stackrel{\mathrm{def}}{=} \boldsymbol{\lambda} y^{(A\Rightarrow B)\Rightarrow A}.\boldsymbol{\mu}\alpha^A.[\alpha](y(\boldsymbol{\lambda} x^A.\boldsymbol{\mu}\beta^B.[\alpha]x))$$

is a (in fact, the simplest) witness for Peirce's Law, $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$, the famous (classical) tautology that is not intuitionistically valid. See Table 2 for a proof.

(ii) The (closed) $\lambda\mu$-term[4]

$$\boldsymbol{\aleph} \stackrel{\mathrm{def}}{=} \boldsymbol{\lambda} z^{\neg\neg A}.\boldsymbol{\mu}\alpha^A.z(\boldsymbol{\lambda} x^A.[\alpha^A]x)$$

[4] According to Parigot's syntax of 1992 (see [28, §3.4]) the term that corresponds to $\boldsymbol{\aleph}$ is $\boldsymbol{\lambda} z^{\neg\neg A}.\boldsymbol{\mu}\alpha^A.[\delta^\bot](z(\boldsymbol{\lambda} x^\alpha.\boldsymbol{\mu}\phi^\bot.[\alpha]x))$ which is not closed – it has a free name δ of type $-$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A^x;\ \vdash x : A}}{A^x;\ A^\alpha \vdash [\alpha^A]x : \bot}\,\bot\textbf{-intro}}{A^x;\ A^\alpha, B^\beta \vdash [\alpha^A]x : \bot}\,\mu\textbf{-wk}}{A^x;\ A^\alpha \vdash \boldsymbol{\mu}\beta^B.[\alpha^A]x : B}\,\bot\textbf{-elim}}{;\ A^\alpha \vdash \boldsymbol{\lambda}x^A.\boldsymbol{\mu}\beta^B.[\alpha^A]x : A \Rightarrow B}\,\Rightarrow\textbf{-intro}}{((A \Rightarrow B) \Rightarrow A)^y;\ \vdash y : (A \Rightarrow B) \Rightarrow A \qquad ;\ A^\alpha \vdash \boldsymbol{\lambda}x^A.\boldsymbol{\mu}\beta^B.[\alpha^A]x : A \Rightarrow B}\,\Rightarrow\textbf{-elim}}{((A \Rightarrow B) \Rightarrow A)^y;\ A^\alpha \vdash y(\boldsymbol{\lambda}x^A.\boldsymbol{\mu}\beta^B.[\alpha^A]x) : A}\,\bot\textbf{-intro}}{((A \Rightarrow B) \Rightarrow A)^y;\ A^\alpha, A^{\alpha'} \vdash [\alpha'^A](y(\boldsymbol{\lambda}x^A.\boldsymbol{\mu}\beta^B.[\alpha^A]x)) : \bot}\,\mu\textbf{-ctr}}{((A \Rightarrow B) \Rightarrow A)^y;\ A^\alpha \vdash [\alpha^A](y(\boldsymbol{\lambda}x^A.\boldsymbol{\mu}\beta^B.[\alpha^A]x)) : \bot}\,\bot\textbf{-elim}}{((A \Rightarrow B) \Rightarrow A)^y;\ \vdash \boldsymbol{\mu}\alpha^A.[\alpha^A](y(\boldsymbol{\lambda}x^A.\boldsymbol{\mu}\beta^B.[\alpha^A]x)) : A}\,\Rightarrow\textbf{-elim}}{;\ \vdash \boldsymbol{\lambda}y^{(A\Rightarrow B)\Rightarrow A}.\boldsymbol{\mu}\alpha^A.[\alpha^A](y(\boldsymbol{\lambda}x^A.\boldsymbol{\mu}\beta^B.[\alpha^A]x)) : ((A \Rightarrow B) \Rightarrow A) \Rightarrow A}$$

Table 2: A proof of Peirce's Law

is a proof of the tautology $\neg\neg A \Rightarrow A$. There are other witnesses of the same proposition, e.g.,

$$\boldsymbol{\lambda}z^{\neg\neg A}.\boldsymbol{\mu}\alpha^A.z(\boldsymbol{\lambda}x^A.z(\boldsymbol{\lambda}y^A.[\alpha]x)).$$

It may be worth clarifying the status of the absurdity type in categorical terms. One can show (see [27]) that absurdity is not (modelled by) an initial but weakly initial object; for each proposition A, $\neg\neg A$ is not isomorphic to A, rather, the canonical map $A \xrightarrow{\bot} \neg\neg A$ has a retraction $\aleph$ (see Example 2.2).

Definition 2.3 The **reduction relation** $\rightarrow$ of $\boldsymbol{\lambda\mu}_{\rm v}$, viewed as a rewrite system, is defined to be the compatible (i.e. contextual) closure of the notion of reduction defined by four groups of redex rules, namely, (β_v), (μ), (ζ) and $(\bot)$. We let v range over **values** which are λ-abstractions; and let K range over (call-by-value) **singular contexts** which are single-holed contexts that have either the shape [-]s or v[-]. As usual, for any context C, we write $C[u]$ to mean the standard non capture-avoiding context substitution.

$\boldsymbol{\beta_{\rm v}}$-reduction: $\quad (\beta_{\rm v}) \quad (\boldsymbol{\lambda}x^A.s)v \quad \rightarrow \quad s[v/x]$

$\boldsymbol{\mu}$-reduction:

$$\begin{array}{llll} (\mu\text{-}\beta) & [\alpha^A](\boldsymbol{\mu}\gamma^A.e) & \rightarrow & e[\alpha/\gamma] \\ (\mu\text{-}\eta) & \boldsymbol{\mu}\alpha^A.[\alpha^A]s & \rightarrow & s \qquad \alpha \notin \mathrm{FN}(s) \end{array}$$

$\boldsymbol{\zeta}$-reduction[5]:

$$(\zeta) \quad K^B[\boldsymbol{\mu}\alpha^A.e] \quad \rightarrow \quad \begin{cases} \boldsymbol{\mu}\beta^B.e[\beta, K/\alpha] & \text{if } B \neq \bot \\ e[\bot, K/\alpha] & \text{otherwise} \end{cases}$$

$\bot$-reduction:

$$(\bot) \quad v^{-\Rightarrow B}s^- \quad \rightarrow \quad \begin{cases} \boldsymbol{\mu}\beta^B.s^- & \text{if } B \neq \bot \\ s^- & \text{otherwise.} \end{cases}$$

We shall refer to the above as **$\boldsymbol{\lambda\mu}_{\rm v}$-redex rules** and terms on the l.h.s. of the redex rules as **$\boldsymbol{\lambda\mu}_{\rm v}$-redexes**.

Three kinds of **substitution** can be distinguished in μPCF. The first $s[v/x]$ is the usual "substitution of a term v for all free occurrences of the variable x in s", as in the λ-calculus; the second $s[\alpha/\gamma]$, as in rule $(\mu\text{-}\beta)$, is just *renaming* (of names). The third, which we shall call **mixed substitution**[6], is a somewhat unusual syntactic operation. Given a context C^A (not necessarily singular) of type A with a B-typed "hole" [-], the term $s[\alpha^A, C^A/\beta^B]$ is obtained from s by "recursively replacing every named subterm of s of the shape $[\beta^B](u)$ by $[\alpha^A](C[u])$". (We omit the formal definition.) For example $\boldsymbol{\mu}\gamma^C.[\alpha^A](y(\boldsymbol{\lambda}x^A.\boldsymbol{\mu}\beta^B.[\alpha]x))[\delta^D, K^D/\alpha^A]$ is

$$\boldsymbol{\mu}\gamma^C.[\delta^D]K[y(\boldsymbol{\lambda}x^A.\boldsymbol{\mu}\beta^B.[\delta](K[x]))].$$

In the case of $\bot$-typed context K^- with a B-typed hole, the term $s[\bot, K^-/\beta^B]$ is obtained from s by recursively

[5] This rule, or rather group of rules, is to be distinguished from the ζ-rule of [27]

$$\boldsymbol{\mu}\alpha^{A\Rightarrow B}.e \quad \rightarrow \quad \boldsymbol{\lambda}x^A.\boldsymbol{\mu}\beta^B.e[\beta, [\text{-}]x/\alpha]$$

which (call it $\zeta^{\rm ext}_{\rm fun}$ here) is the *extensional* version of $(\zeta_{\rm fun})$.

[6] This generalizes the mixed substitution introduced in [27].

replacing every named subterm of s of the shape $[\beta^B](u)$ by $K[u]$. We adopt the variable convention of Barendregt's book [3] and extend it to names in the natural way.

Remark 2.4 Having explained mixed substitution, it may now be worth unpacking the **ζ-redex rule** according to the two cases of singular contexts, corresponding to whether the μ-abstraction is in the *fun*ction or *arg*ument position of the application: for $B \neq \bot$

$$\begin{array}{lrcl} (\zeta_{\mathrm{fun}}) & (\boldsymbol{\mu}\alpha^{A\Rightarrow B}.e)t^A & \to & \boldsymbol{\mu}\beta^B.e[\beta,[\text{-}]t/\alpha] \\ & (\boldsymbol{\mu}\alpha^{A\Rightarrow -}.e)t^A & \to & e[\bot,[\text{-}]t/\alpha^{A\Rightarrow -}] \\ (\zeta_{\mathrm{arg}}) & v^{A\Rightarrow B}(\boldsymbol{\mu}\alpha^A.e) & \to & \boldsymbol{\mu}\beta^B.e[\beta,v[\text{-}]/\alpha] \\ & v^{A\Rightarrow -}(\boldsymbol{\mu}\alpha^A.e) & \to & e[\bot,v[\text{-}]/\alpha^A]. \end{array}$$

The rule (ζ_{arg}) is an instance of what Parigot refers to as a rule "symmetric" to what we call (ζ_{fun}).

The first property we can state at once about $\boldsymbol{\lambda\mu}_{\mathrm{v}}$ is *subject reduction*: if $\Gamma;\ \Delta \vdash s : A$ and $s \to s'$ then $\Gamma;\ \Delta \vdash s' : A$. In fact, the rewrite system is very well-behaved:

Theorem 2.5 *The reduction system $\boldsymbol{\lambda\mu}_{\mathrm{v}}$ is Church-Rosser and strongly normalizing.* □

The standard Tait and Martin-Löf parallel reduction method gives an essentially straightforward proof of confluence. Strong normalization is harder. We use a candidates of reducibility method but require an induction hypothesis stronger than that in [30] in order to handle the ζ_{arg}-rule.

Remark 2.6 In [29] Parigot observed that, in contrast to the Curry-Howard representation theory of intuitionistic logic, with respect to his CBN $\lambda\mu$-calculus, numerals are not uniquely represented by *normal* forms. For instance both the Church numeral $\boldsymbol{\lambda}xf.f(fx)$ and the term $\boldsymbol{\lambda}xf.\boldsymbol{\mu}\alpha.[\alpha](f(f\boldsymbol{\mu}\beta.[\alpha](f(fx))))$ denote 2. A natural way to cause the latter to reduce to the former, thus clawing back uniqueness by admitting fewer normal forms, is to introduce the rule:

$$s^{A\Rightarrow B}(\boldsymbol{\mu}\alpha^A.t) \quad \to \quad \boldsymbol{\mu}\beta^B.t[\beta,s[\text{-}]/\alpha]$$

with s ranging over all $A \Rightarrow B$-typed terms. Unfortunately, as Parigot noted, such a rule would break the confluence of the (CBN) system. But contrary to Parigot's view, uniqueness of data representation can be reconciled with confluence provided s in the rule is restricted to abstractions *and variables*, and β-reduction be required to call by value. A proper account of how this can be done will be given elsewhere.

The valid **equational judgements** of $\boldsymbol{\lambda\mu}_{\mathrm{v}}$ are those that are derivable in the proof system as defined in Table 7. As an easy corollary of Theorem 2.5, we have:

Corollary 2.7 *Definitional equality of the type theory $\boldsymbol{\lambda\mu}_{\mathrm{v}}$ is consistent, compatible with cut, congruent (i.e. compatible with all term formation rules) and decidable.* □

3 $\mu\mathrm{PCF}_{\mathrm{v}}$ as a programming language

In this section we introduce a deterministic, sequential, CBV programming language $\mu\mathrm{PCF}_{\mathrm{v}}$. Syntactically $\mu\mathrm{PCF}_{\mathrm{v}}$ is obtained from $\boldsymbol{\lambda\mu}_{\mathrm{v}}$ by augmenting it by basic arithmetic, conditionals, and fixed-point operators. (So $\mu\mathrm{PCF}_{\mathrm{v}}$ is to $\boldsymbol{\lambda\mu}_{\mathrm{v}}$ what Scott's programming language PCF [37, 33] is to pure simply-typed λ-calculus.) We distinguish two strengths of the language. First the more expressive version.

Programming language $\mu\mathrm{PCF}_{\mathrm{v}}$

Types of $\mu\mathrm{PCF}_{\mathrm{v}}$ are generated by the function space constructor $\Rightarrow$ from three **program types**, namely, natural number ι, boolean o, and absurdity $\bot$. (Later we shall identify a sublanguage called $\mu\mathrm{PCF}_{\mathrm{v}}^-$ whose types are generated from just ι and o. Whenever there is a need to distinguish the two, we shall refer to $\mu\mathrm{PCF}_{\mathrm{v}}$ as **$\bot$-enriched** and its sublanguage as **$\bot$-free**.) Let $\mathcal{T}$ denote the set of $\mu\mathrm{PCF}_{\mathrm{v}}$-types. *Raw $\mu\mathrm{PCF}_{\mathrm{v}}$-terms* are obtained from (the rules in Figure 1 that define) $\lambda\mu$-terms by adding the following rules:

$$\begin{array}{lll} \mid & c & \text{constant} \\ \mid & \mathbf{Y}^A(s) & \text{Y-term} \end{array}$$

where c ranges over the set $\mathcal{A}$ consisting of numerals 1, 2, 3, etc., booleans t,f, and first-order constants: successor, predecessor, test for zero, and conditionals, whose types are as follows:

$$\begin{array}{lcllcl} \mathsf{succ} & : & \iota \Rightarrow \iota & \mathsf{cond}^\iota & : & o \Rightarrow (\iota \Rightarrow (\iota \Rightarrow \iota)) \\ \mathsf{pred} & : & \iota \Rightarrow \iota & \mathsf{cond}^o & : & o \Rightarrow (o \Rightarrow (o \Rightarrow o)) \\ \mathsf{iszero} & : & \iota \Rightarrow o. & & & \end{array}$$

In the following we shall omit the type annotation in $\mathbf{Y}^A(s)$, cond^o and cond^ι.

The **typing** (or term assignment) judgements are defined by the rules in Table 1 augmented by the appropriate rules that assign types to constants, and by the following:

$$\frac{\Gamma;\ \Delta \vdash s : A \Rightarrow A}{\Gamma;\ \Delta \vdash \mathbf{Y}(s) : A}$$

where A is a function type. **Terms** of μPCF_v are just the type-correct raw terms.

The **operational semantics** of μPCF_v is given in terms of a Plotkin-style transition relation ("small-step semantics") $>$, which defines a deterministic and sequential reduction *strategy* on closed μPCF_v-terms. **Programs** of μPCF_v are closed terms of program type. **Values**, ranged over by v, v', are constants and abstractions.

Definition 3.1 (small-step semantics $>$) The transition relation $>$ is defined by five rules in Table 4. In the table, we let $\theta > \theta'$ range over the following redex-rules:

- $\boldsymbol{\lambda\mu}_\text{v}$-redex rules $(\beta_\text{v}), (\zeta)$ and $(\bot)$
- basic arithmetic and conditionals redex rules as defined in Table 3
- fixpoint: $(\mathbf{Y}_\text{v}) \quad \mathbf{Y}(s) \quad > \quad s(\boldsymbol{\lambda} x.\mathbf{Y}(s)x)$.

We define **active contexts**, ranged over by E, as follows:

$$E \quad ::= \quad [\text{-}] \quad | \quad Es \quad | \quad v^{A\Rightarrow B}E$$

where A is not $\bot$ (for if it were then $v^{-\Rightarrow B}s$ would be a $\bot$-redex). Note that an active context E has exactly one hole; its role is similar but not identical to that of *evaluation context*[7]. Active contexts E should *not* be confused with singular contexts K introduced in Definition 2.3.

It is easy to check that $>$ is a deterministic reduction strategy (for closed terms) i.e. it defines a partial function. Write $\gg$ as the reflexive, transitive closure of $>$. For example (recall that $\boldsymbol{\kappa}$ is the simplest proof of Peirce's Law as defined in Example 2.2)

$$\begin{array}{lll}
 & \Theta \equiv \boldsymbol{\kappa}(\boldsymbol{\lambda} k^{\iota\Rightarrow\iota}.(\boldsymbol{\lambda} l^{\iota}.1)(k2)) & \\
\gg & \boldsymbol{\mu}\alpha^{\iota}.[\alpha]((\boldsymbol{\lambda} l.1)((\boldsymbol{\lambda} x.\boldsymbol{\mu}\beta^{\iota}.[\alpha]x)2)) & E[\beta_\text{v}], E_\mu[\beta_\text{v}] \\
> & \boldsymbol{\mu}\alpha^{\iota}.[\alpha]((\boldsymbol{\lambda} l.1)(\boldsymbol{\mu}\beta^{\iota}.[\alpha]2)) \equiv \Xi & E_\mu[\beta_\text{v}] \\
> & \boldsymbol{\mu}\alpha^{\iota}.[\alpha](\boldsymbol{\mu}\gamma.[\alpha]2) & E_\mu[\zeta_\text{arg}] \\
\gg & 2. & \mu\text{-}\beta, \mu\text{-}\eta
\end{array}$$

(By $E_\mu[\beta_\text{v}]$ (say) above, we mean that the reduction step is by the rule E_μ where the redex rule in question is β_v.) If the program Θ were evaluated in a CBN regime, then the CBN β-rule would apply at Ξ, and so, the reduction would yield the value 1 instead. In contrast to pure functional computation, CBV-β and CBN-β are inconsistent redex rules in the presence of control features.

We say that $s >$-*diverges*, or simply **diverges**, just in case for an infinite collection of s_i, we have $s > s_1 > s_2 > \cdots$.

Proposition 3.2 *For any closed μPCF_v-term s, $s \not>$ (i.e. $\neg[\exists s'.s > s']$) if and only if s is a value or it has the form $\boldsymbol{\mu}\alpha^{A\Rightarrow B}.[\alpha](\boldsymbol{\lambda} x^A.t)$ where $\alpha \in \text{FN}(\boldsymbol{\lambda} x.t)$. We shall refer to the latter as* μ**-canonical form**. *Hence for any closed μPCF_v-term s, either s converges to a value, or to a μ-canonical form, or s diverges.* □

As a corollary if s is a program, then either s converges to a numeral or boolean, or it diverges.

What do the dual constructs of μ-abstraction and naming mean computationally? Consider a functional language $\mathcal{L}$ whose operational semantics is given by a small-step semantics $>$, whose reflexive, transitive closure is denoted $\gg$. Let E range over the *evaluation contexts* of $\langle \mathcal{L}, > \rangle$. Informally we say that $\mathcal{L}$ has a **generic jump operator** if there is a context operator θ (which we can think of as a function $C \mapsto C_\theta$) such that for any evaluation context E, and for any C from an appropriate class of contexts (which should be as general as possible), a *commutation* of contexts E and C_θ can always be achieved after finitely many steps of reduction in the following sense

$$E^A[C_\theta^B[u^A]] \quad \gg \quad C_\theta^B[E^A[u]].$$

An immediate problem with typing arises: unless there is a systematic way to transform the context C_θ^B on the right to an appropriate context of type A, subject reduction will be lost. Remarkably the dual $\boldsymbol{\lambda\mu}_\text{v}$-constructs of naming and μ-abstraction provide a good approximation to just such a generic control operator in a systematic and type-consistent way.

Proposition 3.3 (first-class control) *Let E^A be an active context of type A, and e a $\bot$-typed term. It is straightforward to verify that*

$$\begin{array}{lrcl}
(1) & E^A[\boldsymbol{\mu}\beta^B.e] & \gg & \boldsymbol{\mu}\alpha^A.e[\alpha, E/\beta] \\
(2) & \boldsymbol{\mu}\alpha^A.[\alpha^A](E^A[\boldsymbol{\mu}\beta^B.e]) & \gg & \boldsymbol{\mu}\alpha^A.e[\alpha, E/\beta] \\
(3) & \boldsymbol{\mu}\alpha^A.E^-[\boldsymbol{\mu}\beta^B.e] & \gg & \boldsymbol{\mu}\alpha^A.e[\bot, E/\beta]
\end{array}$$

where $\alpha \in \text{FN}(E^A[\boldsymbol{\mu}\beta^B.e])$ in clause (2). (Note that in the case of the $\bot$-free sublanguage μPCF_v^-, (1) and (2) are valid.) □

Take E^A to be the active context $(v^{B\Rightarrow(D\Rightarrow A)}[\text{-}])s^D$.

[7] A one-step reduction $>$ is said to be characterized by a set $\mathcal{E}$ of **evaluation contexts** and a set $\mathcal{R}$ of redex rules just in case for any s and s', $s > s'$ iff there is a unique $E \in \mathcal{E}$ such that $s \equiv E[\theta]$, $s' \equiv E[\theta']$ and $\theta > \theta'$ is an instance of an $\mathcal{R}$-redex rule.

$\mathsf{pred}\ n+1 \to n \quad \mathsf{pred}\ 0 \to 0 \quad \mathsf{cond\ t} \to \boldsymbol{\lambda} xy.x \quad \mathsf{cond\ f} \to \boldsymbol{\lambda} xy.y$
$\mathsf{succ}\ n \to n+1 \quad \mathsf{iszero}\ 0 \to \mathsf{t} \quad \mathsf{iszero}\ n+1 \to \mathsf{f}$

Table 3: Redex rules for basic arithmetic and definition by cases (or conditional)

(E)	$E[\theta]$	$>$	$E[\theta']$	
(E_μ)	$\boldsymbol{\mu}\alpha^A.[\alpha^A](E[\theta])$	$>$	$\boldsymbol{\mu}\alpha^A.[\alpha^A](E[\theta'])$	$\alpha \in \mathrm{FN}(E[\theta])$
(E_-)	$\boldsymbol{\mu}\alpha^A.E^-[\theta]$	$>$	$\boldsymbol{\mu}\alpha^A.E^-[\theta']$	
$(\mu\text{-}\eta)$	$\boldsymbol{\mu}\alpha^A.[\alpha^A]t$	$>$	t	$\alpha \notin \mathrm{FN}(t)$
$(\mu\text{-}\beta)$	$\boldsymbol{\mu}\alpha^A.[\alpha^A](\boldsymbol{\mu}\beta^A.e)$	$>$	$\boldsymbol{\mu}\alpha.e[\alpha/\beta]$	$\alpha \in \mathrm{FN}(\boldsymbol{\mu}\beta.e)$

Table 4: Rules defining Plotkin-style transition (or "small-step") semantics of $\mu\mathrm{PCF}_\mathrm{v}$: $s > s'$ where s is closed.

For an illustration we see that

$$\begin{array}{lcll} E^A[\boldsymbol{\mu}\beta^B.e] & > & (\boldsymbol{\mu}\gamma^{D\Rightarrow A}.e[\gamma, v[\text{-}]/\beta])s & E[\zeta_{\mathrm{arg}}] \\ & > & \boldsymbol{\mu}\alpha^A.e[\gamma, v[\text{-}]/\beta][\alpha, [\text{-}]s/\gamma] & E[\zeta_{\mathrm{fun}}] \\ & \equiv & \boldsymbol{\mu}\alpha^A.e[\alpha, E/\beta]; & \end{array}$$

the term $\boldsymbol{\mu}\beta^B.e$, initially embedded in the active context $E[\bot]$ has been "percolated" to the top-level, and transformed into $\boldsymbol{\mu}\alpha^A.e[\alpha, E/\beta]$. The transformed term has a new type which is the type of the top-level, thus preserving subject reduction. From a computational viewpoint, we can think of *naming* as a way of denoting subterm-positions (or more accurately in terms of abstract syntax, a way of identifying nodes in a tree) where, e.g, the current control context may subsequently be placed. The dual construct of μ-abstraction is the associated binding and parameter-passing operator, where the parameter in question should be thought of as control context.

Programming language $\mu\mathrm{PCF}_\mathrm{v}^\bot$

Though (as we shall see in section 4) the full power of the $\bot$-enriched $\mu\mathrm{PCF}$ is needed to express Felleisen's theory of sequential control, a weaker "$\bot$-free" subsystem $\mu\mathrm{PCF}_\mathrm{v}^-$ suffices for the interpretation of standard control constructs. The types of $\mu\mathrm{PCF}_\mathrm{v}^-$ are exactly the PCF-types (i.e. types generated from ι and o by $\Rightarrow$). Raw terms of $\mu\mathrm{PCF}_\mathrm{v}^-$ are defined by the grammatical rules of $\mu\mathrm{PCF}_\mathrm{v}$ except that the two rules corresponding to μ-abstraction and named term respectively are replaced by

$$\mid \quad \boldsymbol{\mu}\alpha^A.[\beta^B]s \quad \mu\text{-abstraction.}$$

Similarly the typing judgements are defined by the corresponding rules of $\mu\mathrm{PCF}_\mathrm{v}$ except that the two rules ($\bot$-**intro**) and ($\bot$-**elim**) are replaced by

$$(\mu\textbf{-abs}) \quad \frac{\Gamma;\ \Delta, B^\beta \vdash s : A}{\Gamma;\ \Delta, A^\alpha \vdash \boldsymbol{\mu}\beta^B.[\alpha^A]s : B} \quad A^\alpha \notin \Delta$$

Hence in $\mu\mathrm{PCF}_\mathrm{v}^-$ the body e of a μ-abstraction $\boldsymbol{\mu}\alpha.e$ necessarily has the shape $[\beta]s$; there is no term of type $\bot$. In particular, named-terms $[\beta]s$ are not well-formed terms of $\mu\mathrm{PCF}_\mathrm{v}^-$.

The operational semantics of $\mu\mathrm{PCF}_\mathrm{v}^-$, in terms of a Plotkin-style transition semantics $s > s'$ where s is closed, is inherited from that of $\mu\mathrm{PCF}_\mathrm{v}$. Note that only the first of each of the pairs (ζ_{fun}) and (ζ_{arg}) is applicable; and we need only consider rules $(E), (E_\mu), (\mu\text{-}\beta)$ and $(\mu\text{-}\eta)$ in Table 4. Note that the relevant parts, i.e. (1) and (2), of Proposition 3.3 are valid for $\mu\mathrm{PCF}_\mathrm{v}^-$.

Alternatively we may define a Martin-Löf style evaluation semantics ("big-step semantics") $s \Downarrow k$ with k ranging over values and μ-canonical forms, which we refer to collectively as **canonical forms**.

Definition 3.4 (big-step semantics $\Downarrow$) The relation $\Downarrow$ is defined in Table 5 by rule induction in terms of an intermediate relation $s \downarrow f$ where f ranges over values and μ-abstractions. (We think of $\downarrow$ as the first stage of evaluation, a kind of preprocessing.) In case $\downarrow$ reduces a term to a μ-abstraction, the first four rules of $\Downarrow$ then take over, reducing it further to either a value or a μ-canonical form.

The small-step and big-step semantics coincide for $\mu\mathrm{PCF}_\mathrm{v}^-$: for closed term s,

$$[s \gg k \ \ \& \ \ k \not>] \iff s \Downarrow k.$$

This can be shown rather straightforwardly by an induction on the length of reduction. Hence in the following we shall switch from one to the other freely, and use whichever is more convenient in describing reduction in $\mu\mathrm{PCF}_{\mathrm{v}}^{-}$. We say that s **converges** just in case $s \Downarrow k$ for some k.

Contextual equivalences

Consider two natural contextual or observational equivalences for $\mu\mathrm{PCF}_{\mathrm{v}}^{-}$:
(i) Two terms s and t of the same type (not necessarily closed) are said to be **program-type contextually equivalent**, written $s \sim^{\mathrm{prog}} t$, just in case for every context C such that both $C[s]$ and $C[t]$ are programs, $C[s]$ converges to a value v if and only if $C[t]$ converges to v.
(ii) Two terms s and t are **all-type contextually equivalent**, written $s \sim^{\mathcal{T}} t$, just in case for every closed context C, $C[s]$ converges if and only if $C[t]$ converges.

Proposition 3.5 *Program-type contextual equivalence of $\mu\mathrm{PCF}_{\mathrm{v}}^{-}$ is not a conservative extension of that of $\mathrm{PCF}_{\mathrm{v}}$ (call-by-value* PCF*).* □

Proof Set s_0 and s_1 to be $\boldsymbol{\lambda} f^{\iota\Rightarrow\iota}.F(f0)(F(f1)1)$ and $\boldsymbol{\lambda} f^{\iota\Rightarrow\iota}.F(f1)(F(f0)1)$ respectively where F stands for $\boldsymbol{\lambda} xy.y$. s_0 and s_1 are applicatively bisimilar in $\mathrm{PCF}_{\mathrm{v}}$, and hence, contextually equivalent. Define C to be $\boldsymbol{\mu}\alpha^{\iota}.[\alpha]([\text{-}](\boldsymbol{\lambda} x^{\iota}.\boldsymbol{\mu}\beta^{\iota}.[\alpha]x))$; then $C[s_0] \gg 0$ but $C[s_1] \gg 1$. □

It is difficult to reason about contextual equivalence from first principles. Generally it is not at all trivial to check if two given terms are contextually equivalent. It would therefore be desirable if contextual equivalence can be described in terms of other behavioural equivalences that are easier to understand and calculate with. A natural candidate to consider is applicative bisimilarity. We say that s and $t : A_1 \Rightarrow (\cdots (A_n \Rightarrow a))$ are *program-type applicatively bisimilar*, written $s \simeq^{\mathrm{prog}} t$, just in case for every closed u_i of type A_i, for every program value v, $s\vec{u} \Downarrow v$ if and only if $t\vec{u} \Downarrow v$. There is an associated notion of *all-type applicative bisimilarity* denoted $\simeq^{\mathcal{T}}$ which should be clear and so we will not bother to define Unfortunately both notions of bisimilarity, though entirely plausible, are not congruent; hence they are of little interest.

Proposition 3.6

$$\begin{array}{ccc} \sim^{\mathcal{T}} & \subset & \simeq^{\mathcal{T}} \\ \cap\mid & & \cap \\ \sim^{\mathrm{prog}} & \subset & \simeq^{\mathrm{prog}} \end{array}$$

Hence program-type and all-type applicative bisimilarity of $\mu\mathrm{PCF}_{\mathrm{v}}^{-}$ are both not congruences.

Proof The four inclusions are straightforward to verify. Consider

$$\boldsymbol{\Omega}^{\iota\Rightarrow\iota} \stackrel{\mathrm{def}}{=} \mathbf{Y}(\boldsymbol{\lambda} x^{\iota\Rightarrow\iota}.x) \quad \text{and} \quad \boldsymbol{\lambda} x^{\iota}.\boldsymbol{\Omega}^{\iota\Rightarrow\iota}2$$

which are program-type applicatively bisimilar, but not all-type applicatively bisimilar, and so the right inclusion is strict. As $\boldsymbol{\mu}\alpha^{\iota}.[\alpha]([\text{-}](\boldsymbol{\mu}\beta^{\iota}.[\alpha]2))$ distinguishes the two terms, the bottom inclusion is strict. Let s be $\boldsymbol{\lambda} v^{\iota\Rightarrow\iota}.v(v2)$ and

$$\begin{aligned} t &\equiv \boldsymbol{\mu}\alpha^{\iota\Rightarrow\iota}.[\alpha](\boldsymbol{\lambda} y^{\iota}.\boldsymbol{\mu}\gamma^{\iota}.[\alpha](\boldsymbol{\lambda} x.\mathsf{succ}\, y)) \\ \hat{t} &\equiv \boldsymbol{\lambda} z^{\iota}.\boldsymbol{\mu}\beta^{\iota}.[\beta]((\boldsymbol{\lambda} y.\boldsymbol{\mu}\gamma.[\beta]((\boldsymbol{\lambda} x.\mathsf{succ}\, y)z))z). \end{aligned}$$

t and $\hat{t}$ (obtained from t by reducing it by $(\zeta_{\mathrm{fun}}^{\mathrm{ext}})$, see footnote 5) are all-type applicatively bisimilar but $st \Downarrow 3$ and $s\hat{t} \Downarrow 4$. Hence all-type applicative bisimilarity is not a congruence, and so, the top inclusion is strict. □

Thus *operational extensionality* (or equivalently the context lemma) fails hopelessly for $\mu\mathrm{PCF}_{\mathrm{v}}^{-}$. This is perhaps to be expected since imperative features such as control spoils pure functional behaviour, as is the case in e.g. ν-calculus [31].

4 Expressing control constructs

Parigot already pointed out a connection between certain $\lambda\mu$-terms and control operators in [28]; he stated that $\boldsymbol{\aleph}$ and $\boldsymbol{\kappa}$ (see Example 2.2) "have behaviour close to" Felleisen's $\mathcal{C}$-operator and Scheme `callcc` respectively but offered no explanation. de Groote was the first to study $\lambda\mu$-calculus from a computational viewpoint: in [9] he considered a CPS translation. Various researchers e.g. [26, 36] have studied control operators in relation to classical proofs. More recently Bierman [5] has sketched an explanation of $\lambda\mu$ in terms of a kind of abstract machine with "control environment". In this section we demonstrate the expressive richness of $\mu\mathrm{PCF}_{\mathrm{v}}$ by using it to interpret Felleisen's $\boldsymbol{\lambda}\mathcal{C}_{\mathrm{v}}$ and versions of $\mathrm{PCF}_{\mathrm{v}}$ (CBV PCF) augmented by various control constructs. We shall take an informal approach, and be content with just giving the encoding, omitting the essentially straightforward but tedious soundness proofs from this summary.

Felleisen's theory of sequential control

In [12] Felleisen and Hieb develop an untyped CBV theory of sequential control $\boldsymbol{\lambda}\mathcal{C}_{\mathrm{v}}$ based on Plotkin's CBV

$$v \downarrow v \qquad \boldsymbol{\mu}\alpha.e \downarrow \boldsymbol{\mu}\alpha.e \qquad \frac{s \downarrow n}{\mathsf{succ}\, s \downarrow n+1} \qquad \cdots \qquad \frac{s \downarrow \mathsf{t}}{\mathsf{cond}\, s \downarrow \boldsymbol{\lambda}xy.x} \qquad \cdots$$

$$\frac{s(\boldsymbol{\lambda}x.\mathbf{Y}(s)x) \downarrow f}{\mathbf{Y}(s) \downarrow f} \qquad \frac{s \downarrow \boldsymbol{\mu}\alpha.e}{st \downarrow \boldsymbol{\mu}\beta.e[\beta, [\text{-}]t/\alpha]} \qquad \frac{s \downarrow v \quad t \downarrow \boldsymbol{\mu}\alpha.e}{st \downarrow \boldsymbol{\mu}\beta.e[\beta, v[\text{-}]/\alpha]} \qquad \frac{s \downarrow \boldsymbol{\lambda}x.p \quad t \downarrow v \quad p[v/x] \downarrow v'}{st \downarrow v'}$$

$$\frac{}{\boldsymbol{\mu}\alpha.[\alpha](\boldsymbol{\lambda}x.s) \Downarrow \boldsymbol{\mu}\alpha.[\alpha](\boldsymbol{\lambda}x.s)} \quad \alpha \in \mathrm{FN}(s) \qquad \frac{s \downarrow f \quad \boldsymbol{\mu}\alpha.[\alpha]f \Downarrow k}{\boldsymbol{\mu}\alpha.[\alpha]s \Downarrow k} \quad \alpha \in \mathrm{FN}(s), s \equiv pq \text{ or } \mathbf{Y}(p)$$

$$\frac{\boldsymbol{\mu}\alpha.e[\alpha/\beta] \Downarrow k}{\boldsymbol{\mu}\alpha.[\alpha](\boldsymbol{\mu}\beta.e) \Downarrow k} \quad \alpha \in \mathrm{FN}(\boldsymbol{\mu}\beta.e) \qquad \frac{s \Downarrow k}{\boldsymbol{\mu}\alpha.[\alpha]s \Downarrow k} \quad \alpha \notin \mathrm{FN}(s) \qquad \frac{s \downarrow f \quad f \Downarrow k}{s \Downarrow k} \quad s \equiv pq \text{ or } \mathbf{Y}(p) \qquad v \Downarrow v$$

Table 5: Rules defining Martin-Löf style evaluation (or "big-step") semantics of $\mu\mathrm{PCF}_{\mathrm{v}}^{-}$: $s \Downarrow k$ with s closed.

λ-calculus as an idealization[8] of Scheme's treatment of continuation `callcc` (call with current continuation). For a fairer comparison with μPCF, we shall consider a simply-typed (generated from a collection of base types containing a distinguished absurdity type $\perp$) version of $\boldsymbol{\lambda}\mathcal{C}_{\mathrm{v}}$, and of $\boldsymbol{\lambda}\mathcal{C}_{\mathrm{n}}$, the CBN version of $\boldsymbol{\lambda}\mathcal{C}$. The syntax of $\boldsymbol{\lambda}\mathcal{C}$ is just that of the simply-typed λ-calculus augmented by a single rule: for each type A, and for each $\boldsymbol{\lambda}\mathcal{C}$-term s of type $\neg\neg A$, $\mathcal{C}^A s$ is a term of type A. The CBV equational theory $\boldsymbol{\lambda}\mathcal{C}_{\mathrm{v}}$[9] has four axioms[10] : (β_{v}) and

$$\begin{array}{lrcl}
(\mathcal{C}_{\mathrm{lift}}) & K^B[\mathcal{C}^A s] & = & \mathcal{C}^B(\boldsymbol{\lambda}y^{\neg B}.s(\boldsymbol{\lambda}a^A.yK[a])) \\
(\mathcal{C}_{\mathrm{top}}) & \mathcal{C}^A s & = & \mathcal{C}^A(\boldsymbol{\lambda}x^{\neg A}.s(\boldsymbol{\lambda}a^A.xa)) \\
(\mathcal{C}_{\mathrm{idem}}) & \mathcal{C}^A(\boldsymbol{\lambda}x^{\neg A}.\mathcal{C}^- s) & = & \mathcal{C}^A(\boldsymbol{\lambda}x^{\neg A}.s(\boldsymbol{\lambda}z^-.z))
\end{array}$$

where K ranges over CBV singular contexts. The CBN theory, $\boldsymbol{\lambda}\mathcal{C}_{\mathrm{n}}$, has axioms (β_{n}), $(\mathcal{C}_{\mathrm{lift}})$ but in which K ranges only over CBN singular contexts of the form $[\text{-}]s$, $(\mathcal{C}_{\mathrm{top}})$ and $(\mathcal{C}_{\mathrm{idem}})$. We define a translation $s \mapsto \ulcorner s \urcorner$ of $\boldsymbol{\lambda}\mathcal{C}$ to $\boldsymbol{\lambda}\boldsymbol{\mu}$ by setting $\ulcorner \mathcal{C}^A s \urcorner$ to be $\boldsymbol{\mu}\alpha^A.\ulcorner s \urcorner(\boldsymbol{\lambda}x^A.[\alpha]x)$ (which is just the outermost β-contractum of $\aleph\, s$). There is also a translation in the reverse direction. Take a $\boldsymbol{\lambda}\boldsymbol{\mu}$-term s, we define a $\boldsymbol{\lambda}\mathcal{C}$-term $\llcorner s \lrcorner$ by recursion. The map $s \mapsto \llcorner s \lrcorner$ preserves the λ-formation rules. Assuming a given injection from names to variables so that $\alpha^A \mapsto x^{\neg A}$ etc.,

$$\begin{array}{rcl}
\llcorner [\alpha^A] s \lrcorner & \stackrel{\mathrm{def}}{=} & x^{\neg A} \llcorner s \lrcorner \\
\llcorner \boldsymbol{\mu}\alpha^A.s \lrcorner & \stackrel{\mathrm{def}}{=} & \mathcal{C}^A(\boldsymbol{\lambda}x^{\neg A}.\llcorner s \lrcorner).
\end{array}$$

[8] Felleisen's $\mathcal{C}$-operator differs from Scheme `callcc` in that the former aborts the current control context whereas the latter leaves it intact.

[9] This is the simply-typed version of $\boldsymbol{\lambda}_v$-$\mathbf{C}(\mathbf{d})$ in [12].

[10] There is no harm in omitting the `abort` construct (`abort` s is defined in [12] as $\mathcal{C}(\boldsymbol{\lambda}x.s)$) since it has per force type $- \Rightarrow -$ in the simply-typed version and behaves as the identity. Note that $(\mathcal{C}_{\mathrm{lift}})$, in our formulation, is in effect the two axioms $(\mathcal{F}_L)$ and $(\mathcal{F}_R)$ in [10] taken together.

Now let $\boldsymbol{\lambda}\boldsymbol{\mu}_{\mathrm{n}}^{\flat}$ be the theory consisting of equational axioms (β_{n}) and (ζ_{arg}). de Groote has established the relationship between $\boldsymbol{\lambda}\boldsymbol{\mu}_{\mathrm{n}}$ and $\boldsymbol{\lambda}\mathcal{C}_{\mathrm{n}}$ in [9]:

Proposition 4.1 (de Groote) *Let s and s' be $\boldsymbol{\lambda}\mathcal{C}$-terms and t and t' $\boldsymbol{\lambda}\boldsymbol{\mu}$-terms. Then*

(i) $\boldsymbol{\lambda}\mathcal{C}_{\mathrm{n}} \vdash t = t'$ *iff* $\boldsymbol{\lambda}\boldsymbol{\mu}_{\mathrm{n}}^{\flat} \vdash \ulcorner t \urcorner = \ulcorner t' \urcorner$

(ii) $\boldsymbol{\lambda}\boldsymbol{\mu}_{\mathrm{n}}^{\flat} \vdash s = s'$ *iff* $\boldsymbol{\lambda}\mathcal{C}_{\mathrm{n}} \vdash \llcorner s \lrcorner = \llcorner s' \lrcorner$. □

The relationship between the respective CBV systems with respect to the same translations is unfortunately not as neat.

Remark 4.2 In [12, §3.3] Felleisen and Hieb proposed two desirable axioms $(\mathcal{C}_E)$ and $(\mathcal{C}_{\mathrm{elim}})$ and asked how they can be added to $\boldsymbol{\lambda}\mathcal{C}_{\mathrm{v}}$ without sacrificing Church-Rosser and other good rewriting properties. The former axiom "adds equational power to the calculus" and the latter gives rise to syntactically tidier and more regular normal forms. The question was left unsolved. Our approach based on $\boldsymbol{\lambda}\boldsymbol{\mu}$ may be regarded as a solution: $(\mathcal{C}_E)$ is valid in $\boldsymbol{\lambda}\boldsymbol{\mu}_{\mathrm{v}}$ — it is just rule (1) of Proposition 3.3, and $(\mathcal{C}_{\mathrm{elim}})$ translates to $(\mu\text{-}\beta)$ exactly.

ML-style exception mechanism

All dialects of ML have an exception mechanism, including the original ML of the theorem prover LCF [14], CAML [22] and Standard ML (SML) [24]. We introduce a simplified ML-style exception mechanism on top of $\mathrm{PCF}_{\mathrm{v}}$ along the lines of the simple exceptions considered in [16, §4.1]. Names are used to identify exceptions. We add new constructs to $\mathrm{PCF}_{\mathrm{v}}$ as follows:

$$\frac{\Gamma;\ \Delta \vdash u : A}{\Gamma;\ \Delta, A^{\alpha} \vdash \mathtt{raise}\, \alpha^A\, u : B}$$

$$\frac{\Gamma;\ \Delta \vdash s : A \Rightarrow B \quad \Gamma;\ \Delta, A^{\alpha} \vdash t : B}{\Gamma;\ \Delta \vdash \mathtt{handle}\, \alpha^A\, s\, t : B}$$

The term $\mathtt{raise}\,\alpha^A\,u$ raises an exception known as α^A with value u. Exceptions are handled by $\mathtt{handle}\,\alpha^A\,s\,t$, which binds any free occurrence of α^A in t, evaluates s to value v_1, and then evaluates t. If t evaluates to a value v then the whole expression evaluates to v, or else, if an exception named α^A is raised with a value v_2, then the handler v_1 is applied to v_2. To set down the formal semantics, we need two new redex rules:

$$\begin{array}{rcl}\mathtt{handle}\,\alpha^A\,v_1^{A\Rightarrow B}\,v^A & > & v\\ \mathtt{handle}\,\alpha^A\,v_1\,E[\mathtt{raise}\,\alpha^A\,v_2] & > & v_1 v_2\end{array}$$

where E ranges over evaluation contexts augmented by

$$\mathtt{handle}\,\alpha^A\,E\,t \quad | \quad \mathtt{handle}\,\alpha^A\,v\,E.$$

The exception constructs can be macro expanded into $\mu\mathrm{PCF}_{\mathrm{v}}^{-}$ as follows: pick a fresh name δ^B

$$\begin{array}{rcll}\ulcorner\mathtt{raise}\,\alpha^A\,u\urcorner & \stackrel{\mathrm{def}}{=} & \boldsymbol{\mu}\delta^B.[\alpha^A]\ulcorner u\urcorner & \delta\notin \mathrm{FN}(\ulcorner u\urcorner)\\ \ulcorner\mathtt{handle}\,\alpha^A\,s\,t\urcorner & \stackrel{\mathrm{def}}{=} & \boldsymbol{\mu}\beta^B.[\beta](\ulcorner s\urcorner(\boldsymbol{\mu}\alpha^A.[\beta]\ulcorner t\urcorner)).\end{array}$$

We check that the two redex rules are "simulated" correctly by $\mu\mathrm{PCF}_{\mathrm{v}}^{-}$. It is easy to see that if $\ulcorner s\urcorner \Downarrow v_1$ and $\ulcorner t\urcorner \Downarrow v$ then $\ulcorner\mathtt{handle}\,\alpha^A\,s\,t\urcorner \Downarrow v$ as required. Next suppose $\ulcorner s\urcorner \Downarrow v_1$ and $\ulcorner t\urcorner \gg E[\mathtt{raise}\,\alpha^A\,v_2]$. We then have

$$\begin{array}{rl}& \ulcorner\mathtt{handle}\,\alpha^A\,s\,t\urcorner\\ \stackrel{\mathrm{def}}{=} & \boldsymbol{\mu}\beta^B.[\beta](\ulcorner s\urcorner(\boldsymbol{\mu}\alpha^A.[\beta]\ulcorner t\urcorner))\\ \gg & \boldsymbol{\mu}\beta^B.[\beta](v_1(\boldsymbol{\mu}\alpha^A.[\beta]E[\mathtt{raise}\,\alpha^A\,v_2]))\\ > & \boldsymbol{\mu}\beta^B.[\beta](\boldsymbol{\mu}\beta'^B.[\beta](E[\boldsymbol{\mu}\delta^B.[\beta'](v_1 v_2)]))\\ > & \boldsymbol{\mu}\beta^B.[\beta](E[\boldsymbol{\mu}\delta^B.[\beta](v_1 v_2)])\\ \gg & \boldsymbol{\mu}\beta^B.[\beta](v_1 v_2) > v_1 v_2\end{array}$$

as required.

First-class continuations: `callcc`, `throw`, `abort`

SML/NJ, the New Jersey implementation of Standard ML, has primitives for reifying (or capturing), invoking and discarding continuations, namely, `callcc`, `throw` and `abort`. We shall add (simplified versions of) such constructs to $\mathrm{PCF}_{\mathrm{v}}$. The formation rules are:

$$\frac{\Gamma;\ \Delta\vdash s : (A\Rightarrow B)\Rightarrow A}{\Gamma;\ \Delta\vdash \mathtt{callcc}\,s : A}$$

$$\frac{\Gamma;\ \Delta\vdash s : A}{\Gamma;\ \Delta, A^\alpha\vdash \mathtt{abort}^B_{\alpha^A} s : B.}$$

$$\frac{\Gamma;\ \Delta, A^\alpha\vdash s : A}{\Gamma;\ \Delta\vdash \mathtt{set}\,\alpha^A\,\mathtt{in}\,s : A}$$

The redex rules setting out the (standard) behaviour of the control constructs are presented in Table 6 where E ranges over the appropriate evaluation contexts. `callcc` takes a term s of type $(A\Rightarrow B)\Rightarrow A$ as argument, and applies it to an abstraction of the current evaluation (or control) context, the continuation. A continuation[11] has the same first-class status as a λ-abstraction; upon invocation it discards the evaluation context of the application and resumes the abstracted evaluation context with its argument. Names are used to identify (or delimit) continuations earmarked for discarding by `abort`. So the construct $\mathtt{set}\,\alpha^A\,\mathtt{in}\,\bot$ is an operator that binds the name α^A. See [17] and [38] for similar formulations of first-class continuation constructs.

We can translate $\mathrm{PCF}_{\mathrm{v}}$ augmented by such constructs into $\mu\mathrm{PCF}_{\mathrm{v}}$ as before by macro-expanding the continuation constructs into $\mu\mathrm{PCF}_{\mathrm{v}}^{-}$-terms. Define an encoding

$$\begin{array}{rcll}\ulcorner\mathtt{callcc}\,s\urcorner & \stackrel{\mathrm{def}}{=} & \boldsymbol{\mu}\alpha^A.[\alpha](\ulcorner s\urcorner(\boldsymbol{\lambda}x^A.\boldsymbol{\mu}\beta^B.[\alpha]x))\\ \ulcorner\mathtt{abort}^B_{\alpha^A}\,s\urcorner & \stackrel{\mathrm{def}}{=} & \boldsymbol{\mu}\beta^B.[\alpha^A]\ulcorner s\urcorner & \beta\notin \mathrm{FN}(\ulcorner s\urcorner)\\ \ulcorner\mathtt{set}\,\alpha^A\,\mathtt{in}\,s\urcorner & \stackrel{\mathrm{def}}{=} & \boldsymbol{\mu}\alpha^A.[\alpha]\ulcorner s\urcorner.\end{array}$$

Note that the translate of $\mathtt{callcc}\,s$ is just (the outmost β-contractum of) $\boldsymbol{\kappa}s$ where $\boldsymbol{\kappa}$ is the simplest witness of Peirce's Law as identified in Example 2.2. It is easy to see that the encoding correctly simulates the redex rules in $\mu\mathrm{PCF}_{\mathrm{v}}^{-}$.

5 Further directions and conclusions

One of the most promising ML-style general accounts of control is the core language presented in [16] which uses the additional feature of prompts. Although there are similarities between prompts and μ-names (both introduce state into the computation), $\mu\mathrm{PCF}_{\mathrm{v}}^{-}$ cannot properly represent the core language as it lacks a name *generating* or "new name" facility. A consequence of using prompts is some restriction of polymorphism (similar to that required for references) becomes necessary. One solution is to subject control constructs to the discipline of Tofte's weak-polymorphism, which will give type soundness [16]. Our name-based system does not appear to run into these difficulties. We plan to extend $\mu\mathrm{PCF}_{\mathrm{v}}$ to recursive types as in Plotkin's FPC (Fixed Point Calculus), or to Damas-Milner style or System F polymorphism. These extension will be presented elsewhere and in [39].

Two challenging **open problems** germane to the $\boldsymbol{\lambda\mu}_{\mathrm{v}}$ approach are worth stating:

[11] We regard continuations as functions rather than objects of a distinguished form (see the discussion in [17]). So continuations that are typed "A `cont`" in SML/NJ are just terms of type $A\Rightarrow B$ in our system, for some B.

$$
\begin{array}{rcll}
E[\mathtt{callcc}\, s] & > & \mathtt{set}\ \alpha^A\ \mathtt{in}\ E[s(\boldsymbol{\lambda} x^A.\mathtt{abort}^B_\alpha E[x])] & \\
\mathtt{set}\ \alpha^A\ \mathtt{in}\ E[\mathtt{abort}^B_\alpha s] & > & s & \\
\mathtt{set}\ \alpha^A\ \mathtt{in}\ v & > & v & \alpha \notin \mathrm{FN}(v)
\end{array}
$$

Table 6: Redex rules defining ML-style continuation constructs.

- *Reasoning principles for* $\mu\mathrm{PCF}_{\mathrm{v}}$. We know little about $\mu\mathrm{PCF}^{-}_{\mathrm{v}}$-contextual equivalence (so far only negative results) and how to reason about it. Powerful reasoning principles (of the variety of Scott induction or of the various co-inductive principles of late) are needed to help us understand the important notion of program equivalence.
- *Full abstraction for* $\mu\mathrm{PCF}^{-}_{\mathrm{v}}$. The complete game model for $\boldsymbol{\lambda\mu}_{\mathrm{n}}$ in [27] should extend straightforwardly to a (presumably fully abstract) model of $\mu\mathrm{PCF}_{\mathrm{n}}$. The construction of (fully abstract) game model for the CBV case remains open.

It would be good also to have continuation based classical domain models of $\boldsymbol{\lambda\mu}_{\mathrm{v}}$ and $\mu\mathrm{PCF}_{\mathrm{v}}$. Hofmann's analysis [20], and Streicher and Reus' recent work [40] in the area look promising.

In summary we have given a confluent and strongly normalizing version of $\lambda\mu$-calculus with "symmetric" structural rules (or (ζ_{fun}) and (ζ_{arg}) as we call them) as a Curry-Howard style representation theory for classical propositional proofs. The theory is extended to a simple PCF-style prototypical language capable of expressing control in functional programming to a high degree of generality.

Acknowledgements We are grateful to Tony Hoare for detailed comments on an earlier draft of the paper. Discussions with Martin Hyland, Philippe de Groote, Gavin Bierman, Pierre-Louis Curien, Thomas Streicher, Vincent Danos and Hugo Herbelin, and comments from the anonymous referees and Didier Remy, have much improved our understanding of the subject. The first author acknowledges financial support in the form of EU KIT Project 143 (ConFuPro); the second author is supported by a British EPSRC doctoral studentship.

References

[1] A. Aiken, J. H. Williams, and E. L. Wimmers. Program transformation in the presence of errors. In *Proc. 17th ACM POPL*, pages 210–217. ACM Press, 1990.

[2] F. Barbanera and S. Berardi. Extracting constructive content from classical logic via control-like reductions. In *Proc. Int. Conf. TLCA*, pages 45–59. Springer, 1993. LNCS Vol. 664.

[3] H. Barendregt. *The Lambda Calculus*. North-Holland, revised edition, 1984.

[4] U. Berger and H. Schwichtenberg. Program extraction from classical proofs. In D. Leivant, editor, *Proceedings LCC 94*. 1995.

[5] G. M. Bierman. Towards a classical linear λ-calculus. In *Proceedings of Tokyo Conference on Linear Logic*, volume 3 of *Electronic Notes in Computer Science*. Elsevier, 1996.

[6] Th. Coquand. A semantics of evidence for classical arithmetic. *Journal of Symbolic Logic*, 60:325–337, 1995.

[7] V. Danos, J.-B. Joinet, and H. Schellinx. A new deconstructive logic: linear logic. In J.-Y. Girard et al, editor, *Advances in Linear Logic*. Cambridge Univ. Press, 1995.

[8] O. Danvy and A. Filinski. Representing control: A study of the CPS transformation. *Math. Struct. Comp. Sc.*, 2(4):361–391, 1992.

[9] P. de Groote. A CPS-translation of the $\lambda\mu$-calculus. In S. Tison, editor, *Proceedings of the 19th International Colloquium on Trees in Algebra and Programming (CAAP'94)*, pages 85–138. Springer-Verlag, 1994.

[10] M. Felleisen. The theory and practice of first-class prompts. In *Proc. 15th Annual ACM Symp. Principles of Programming Languages, San Diego*, pages 180–190. ACM Press, 1988.

[11] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987.

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

[13] J.-Y. Girard. A new constructive logic: classical logic. *Math. Struct. in Comp. Science*, 1:255–296, 1991.

[14] M. J. C. Gordon, A. J. R. Milner, and C. P. Wadsworth. *Edinburgh LCF*. LNCS 78, Springer-Verlag, 1979.

[15] T. Griffin. A formulae-as-types notion of control. In *Proc. ACM Conf. Principle of Programming Languages*. ACM Press, 1990.

[16] C. A. Gunter, D. Rémy, and J. G. Riecke. A generalization of exceptions and control in ML-like languages. In *Proc. ACM Conf. Functional Programming and Computer Architecture*, pages 12–23. ACM Press, 1995.

[17] R. Harper, B. Duba, and D. MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993.

[18] C. T. Haynes. Logic continuations. *Journal of Logic Programming*, 4:157–176, 1987.

[19] C. T. Haynes, D. P. Friedman, and M. Wand. Obtaining coroutines from continuations. *Journal of Computer Languages*, 11:143–153, 1986.

[20] M. Hofmann. Sound and complete axiomatisations of call-by-value control operators. preprint, 1994.

[21] J.-L. Krivine. Classical logic, storage operators and second order λ-calculus. *Annals of Pure and Applied Logic*, 68:53–78, 1994.

[22] X. Leroy. *The Caml Light System, release 0.6: Documentation and User's Manual*. INRIA, 1993. Included in the Caml Light distribution.

[23] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100:1–77, 1992.

[24] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachussetts, 1990.

[25] C. Murthy. An evaluation semantics for classical proofs. In *Proc. 5th* IEEE *Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1991.

[26] H. Nakano. *Logical Structure of Catch and Throw Mechanism*. PhD thesis, University of Tokyo, 1995.

[27] C.-H. L. Ong. A semantic view of classical proofs: type-theoretic, categorical, denotational characterizations. In *Proc. 11th* IEEE *Symp. Logic in Computer Science, New Jersey, July 1996*, pages 230–241. IEEE Computer Society Press, 1996.

[28] M. Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proc. Int. Conf. Logic Prog. Automated Reasoning*, pages 190–201. Springer, 1992. LNCS Vol. 624.

[29] M. Parigot. Classical proofs as programs. In *Proc. 3rd Kurt Gödel Colloquium*, pages 263–276. Springer-Verlag, 1993. LNCS Vol. 713.

[30] M. Parigot. Strong normalization for second order classical natural deduction. In *Proc. 8th Annual IEEE Symposium on Logic in Computer Science*, pages 39–46. IEEE Computer Society Press, 1993.

[31] A. M. Pitts and I. Stark. On the observable properties of higher order functions that dynamically create local names, or: What's new? In *Proc. 18th Int. Symp. on Math. Foundations of Computer Science*, pages 122–141. Springer-Verlag, 1993. LNCS Vol. 711.

[32] G. D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[33] G. D. Plotkin. LCF as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[34] D. Prawitz. *Natural Deduction*. Almqvist and Wiksell, 1965. Stockholm Studies in Philosophy 3.

[35] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1995.

[36] M. Sato. Intuitionistic and classical natural deduction systems with the catch and throw rules. *Theoretical Computer Science*, 1996. To appear.

[37] D. S. Scott. A type-theoretical alternative to CUCH, ISWIM and OWHY. *Theoretical Computer Science*, 121:411–440, 1993.

[38] D. Sitaram and M. Felleisen. Reasoning with continuations II: Full abstraction for models of control. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 161–175, 1990.

[39] C. A. Stewart. *Classical proofs and a theory of functional control*. PhD thesis, PRG, University of Oxford, 1997. In preparation.

[40] T. Streicher and B. Reus. Continuation semantics: control operators and abstract machines. Submitted for publication, 1996.

[41] C. Talcott. Rum: an intensional theory of functions and control abstraction. In *Proc. 1987 Workshop of Foundations of Logic and Functional Programming*. Springer-Verlag, 1988. LNCS Vol. 306.

[42] J. L. Underwood. *Aspects of the Computational Contents of Proofs*. PhD thesis, Cornell University, 1994.

[43] M. Wand. Continuation-based multiprogramming. In *Record of the 1980 Lisp Conference*, pages 19–28. Springer-Verlag, 1990.

(**reflexivity**)	$\Gamma;\ \Delta \vdash s = s : A$
(**symmetry**)	$\dfrac{\Gamma;\ \Delta \vdash s = t : A}{\Gamma;\ \Delta \vdash t = s : A}$
(**transitivity**)	$\dfrac{\Gamma;\ \Delta \vdash s = t : A \qquad \Gamma;\ \Delta \vdash t = u : A}{\Gamma;\ \Delta \vdash s = u : A}$
(λ-**abstraction**)	$\dfrac{\Gamma, A^x;\ \Delta \vdash s = s' : B}{\Gamma;\ \Delta \vdash \boldsymbol{\lambda} x^A.s = \boldsymbol{\lambda} x^A.s' : A \Rightarrow B}$
(**application**)	$\dfrac{\Gamma;\ \Delta \vdash s = s' : A \Rightarrow B \qquad \Gamma;\ \Delta \vdash t = t' : A}{\Gamma;\ \Delta \vdash st = s't' : B}$
(μ-**abstraction**)	$\dfrac{\Gamma;\ \Delta, B^\beta \vdash s = s' : \bot}{\Gamma;\ \Delta \vdash \boldsymbol{\mu}\beta^B.s = \boldsymbol{\mu}\beta^B.s' : B}$
(**named-term**)	$\dfrac{\Gamma;\ \Delta \vdash s = s' : A}{\Gamma;\ \Delta, A^\alpha \vdash [\alpha^A]s = [\alpha^A]s' : \bot}$
($\boldsymbol{\beta_{\mathrm{v}}}$)	$\Gamma;\ \Delta \vdash (\boldsymbol{\lambda} x^A.s)v = s[v/x^A] : B$
($\boldsymbol{\mu}$-$\boldsymbol{\beta}$)	$\Gamma;\ \Delta, A^\alpha \vdash [\alpha^A](\boldsymbol{\mu}\gamma^A.s) = s[\alpha^A/\gamma^A] : \bot$
($\boldsymbol{\mu}$-$\boldsymbol{\eta}$)	$\Gamma;\ \Delta \vdash \boldsymbol{\mu}\alpha^A.[\alpha^A]s = s : A \quad \alpha \notin \mathrm{FN}(s)$
($\boldsymbol{\zeta}$)	$\Gamma;\ \Delta \vdash K^B[\boldsymbol{\mu}\alpha^A.e] = \begin{cases} \boldsymbol{\mu}\beta^B.e[\beta, K/\alpha] : B & \text{if } B \neq \bot \\ e[\bot, K/\alpha] : \bot & \text{otherwise} \end{cases}$
($\bot$)	$\Gamma;\ \Delta \vdash v^{-\Rightarrow B}s^- = \begin{cases} \boldsymbol{\mu}\beta^B.s^- : B & \text{if } B \neq \bot \\ s^- : \bot & \text{otherwise.} \end{cases}$

Table 7: Rules defining equality judgements of $\boldsymbol{\lambda\mu}_{\mathrm{v}}$.