A Third-Order Representation of the $\lambda\mu$-Calculus

Andreas Abel*

Carnegie Mellon University, Department of Computer Science
5000 Forbes Avenue, Pittsburgh, PA 15213, USA
phone: +1(412)268-2582, email: abel@cs.cmu.edu

Abstract. Higher-order logical frameworks provide a powerful technology to reason about object languages with binders. This will be demonstrated for the case of the $\lambda\mu$-calculus with two different binders which can most elegantly be represented using a third-order constant. Since cases of third- and higher-order encodings are very rare in comparison with those of second order, a second-order representation is given as well and equivalence to the third-order representation is proven formally.

1 Introduction

The $\lambda\mu$-calculus [Par92,OS97,Bie98], a proof theory for the implicational fragment of classical logic, has been established as a general tool to reason about functional programming languages with control, e.g. continuations and exceptions. It is basically an extension of the λ-calculus by a second binder. Some of its properties like strong normalization and confluence are very fundamental for its use in functional programming and proof systems; a formal verification of these basic properties is therefore desirable. Human proofs are error-prone; even these which have undergone the scientific review process. For instance, the first published proof of confluence for the $\lambda\mu$-calculus contained a flaw that was only recently corrected [BHF01].

When reasoning about programming languages and logics with binders—like the λ-calculus—the use of a *higher-order* logical framework can greatly reduce the size formalized proofs require. This is due to its built-in notions of α-equivalence and substitution which make several technical lemmata obsolete if a clever representation of the object language is chosen.

To our knowledge, properties of the $\lambda\mu$-calculus have not been verified mechanically so far. This article intends to provide a working point by giving two possible encodings of the $\lambda\mu$-calculus in a higher-order abstract syntax. As it will be seen, the $\lambda\mu$-calculus is one of the rare examples that can best be represented in a way that involves a third-order constructor. This representation will enable a natural implementation of the *structural* or *mixed substitution* that has been discussed controversially in the literature [OS97,SR98,dG98,Bie98].

* This work was supported by the Graduiertenkolleg Logik in der Informatik (GKLI), Munich, and the Office of Technology in Education, Carnegie Mellon University.

The remainder of this article is organized as follows: In Sect. 2 we will introduce the $\lambda\mu$-calculus in its original formulation by Parigot [Par92]. We will proceed to interpret it computationally in Sect. 3 by giving a small-step semantics and an application in exception handling. Sect. 4 contains a development of a third-order representation for the untyped and the typed $\lambda\mu$-calculus together with the formalization of the small-step semantics. We will point out a difficulty with the representation and give an alternative, second-order encoding and prove equivalence of both representations in Sect. 5. An outlook on further work will conclude this article.

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

In his seminal 1992 paper Parigot [Par92] introduced the $\lambda\mu$-calculus as a proof theory for classical logic. In the following we will present the first-order fragment which, in its computational interpretation, provides a general tool to model control in functional programming.

The raw terms are given by this grammar:

$$\begin{aligned} M &::= x \mid \lambda x.M \mid M\,M \mid \mu a.N && \textit{(Unnamed) terms} \\ N &::= [a]M && \textit{Named terms} \end{aligned}$$

It is basically the untyped λ-calculus plus a second binder μ and *μ-application* $[a]M$. Furthermore there is a second class of variables which we call *μ-variables* and denote by a, b and c, in contrast to the λ-variables denoted by x, y and z. The constructor μ binds a μ-variable in the following term; as for λ-variables, α-equivalence can be defined. In the following we will not distinguish between α-equivalent terms.

The raw terms can be typed by the following rules which we present in natural deduction style:

$$\frac{\begin{array}{c}\overline{x : A} \\ \vdots \\ M : B\end{array}}{\lambda x.M : A \to B}\,{\to}\mathcal{I}^x \qquad \frac{M_1 : A \to B \qquad M_2 : A}{M_1\,M_2 : B}\,{\to}\mathcal{E} \qquad \frac{a : \overline{A} \qquad M : A}{[a]M : -}\,\textsc{contra}$$

$$\frac{\begin{array}{c}\overline{a : \overline{A}} \\ \vdots \\ N : -\end{array}}{\mu a.N : A}\,\textsc{raa}^a$$

In the rule RAA the judgment $N : -$ means that the term N has *no type*. The symbol $-$ denotes the absurdity, which is *not* considered to be a type. The formula $\overline{A}$ denotes the *cotype* of A; in the realm of logic $\overline{A}$ would be the *negated* proposition A. Thus the μ-introduction is isomorphic to the *reductio ad absurdum* rule in classical logic, which states that A holds if $\neg A$ leads to a contradiction.

The CONTRA rule is analogous to the fact in logic that $\neg A$ and A entail a contradiction. Note that there is no rule which constructs an inhabitant of a cotype. Thus, if $C : \overline{A}$ holds, C must be a μ-variable a. The term $[a]M$ resulting from the application of a μ-variable a to a term M has no type, but it is said to be named by a and hence is called a *named term*.

As closed λ-terms inhabit types that are isomorphic to tautologies in the implicational fragment of *intuitionistic* logic, closed $\lambda\mu$-terms inhabit types that are isomorphic to tautologies in the implicational fragment of *classical* logic. The primary example is the Pierce-type $((A \to B) \to A) \to A$ which is inhabited by the $\lambda\mu$-term

$$\lambda x^{(A\to B)\to A}.\mu a^{\overline{A}}.[a](x\,\lambda y^{A}.\mu b^{\overline{B}}.[a]y)$$

but empty in the λ-calculus. (For better readability we have annotated the abstracted variables with their type.)

2.1 Reductions

In the simply-typed λ-calculus each *normal* term M has the *subformula* property: if $x_1 : A_1, \ldots, x_n : A_n \vdash M : A$ then each subterm of M is an inhabitant of a type given by a subformula of A or A_i for some i. Each term M has a *normal form* which can be obtained by a finite number of β-reductions. The reduction relation is given by the compatible closure of the β-axiom.

$$(\lambda x.M_1)M_2 \longrightarrow_\beta [M_2/x]M_1$$

In the $\lambda\mu$-calculus, we ignore negation when determining subformulas, that is, $\overline{A}$ is considered a subformula of A. Then we can define a reduction relation that computes normal forms which have the subformula property as in the λ-calculus. Besides β-reduction we need permutative conversions induced by the following axiom:

$$(\mu a.N)\,M \longrightarrow_\pi \mu b.[\lambda^\circ z.[b](z\,M)/a]N$$

This involves a new kind of substitution $[\lambda^\circ z.[b](z\,M)/a]N$, Parigot's *structural substitution*, which has the following effect: In N, replace every subterm of the form $[a]M'$ by $[b](M'\,M)$ for any term M'. Our presentation of the structural substitution follows de Groote [dG98], who introduces internal extensions of the syntax:

$M ::= x \mid \lambda x.M \mid M\,M \mid \mu a.N$	*Terms*
$N ::= [C]M$	*Named terms*
$C ::= a \mid \lambda^\circ z.N \quad$ *(z appears exactly once in N)*	*Contexts*

Structural substitution is generalized to a substitution of contexts $[C/a]M$. After a context is substituted for a μ-variable, appearances of λ° are eliminated *silently* by *linear β-reductions*:

$$[\lambda^\circ z.N]M \longrightarrow_{\beta^\circ} [M/z]N.$$

The effect of the π-reduction is to uncover *hidden β-redices*; it is best explained in an example. Consider the following open term of type A:

$$y{:}A \vdash (\mu a^{\overline{A\to A}}.[a]\lambda x^{A}.x)\,y : A$$

Because it contains the subterm $\lambda x.x$ of type $A \to A$, this term does not fulfill the subformula property. It is destroyed by a β-redex $(\lambda x.x)\, y$ which is separated by a μ-abstraction. This hidden redex can be uncovered by a permutation:

$$\begin{aligned}(\mu a^{\overline{A\to A}}.[a]\lambda x^A.x)\, y &\longrightarrow_\pi \mu b^{\overline{A}}.[\lambda^\circ z^{A\to A}.[b](z\, y)/a]([a]\lambda x^A.x)\\ &= \mu b^{\overline{A}}.[\lambda^\circ z^{A\to A}.[b](z\, y)]\lambda x^A.x\\ &\longrightarrow_{\beta^\circ} \mu b^{\overline{A}}.[\lambda x^A.x/z]([b](z\, y))\\ &= \mu b^{\overline{A}}.[b](\lambda x^A.x)\, y\\ &\longrightarrow_\beta \mu b^{\overline{A}}.[b]y\end{aligned}$$

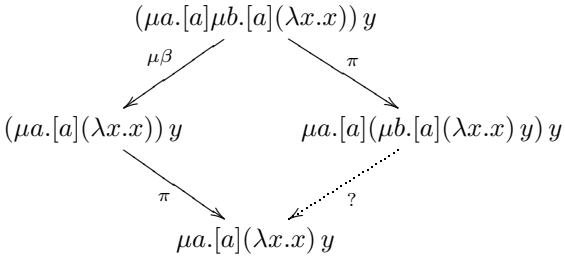
Note that the $\longrightarrow_{\beta^\circ}$-reduction step counts as part of the structural substitution.

Finally, we introduce a third kind of reduction which is analogous to β-reduction:

$$[a]\mu b.N \longrightarrow_{\mu\beta} [a/b]N$$

Parigot calls this reduction "renaming" of named terms. It requires a third kind of substitution—μ-variables for μ-variables—as Ong and Steward point out [OS97]. However, in our formulation it is just a special case of context substitution.

The reduction relation induced by the three axioms β, π and $\mu\beta$ is confluent. However, a straightforward adaption of the Tait and Martin-Löf parallel reduction method fails. This was overlooked in the original confluence proof [Par92], and only after years a correct proof was given by Baba, Hirokawa and Fujita [BHF01]. They found that in its straightforward definition parallel reduction does not have the diamond property. This is due to the twofold effect of π-reduction: It may create a β-redex but at the same time disrupt a $\mu\beta$-redex. For example,



The $\mu\beta$-redex $[a]\mu b...$ is hidden after the π-reduction step. Another π-reduction is required to restore it.

3 A Computational Interpretation

Since the pioneering work of Griffin [Gri90] it is known that control constructs for functional programming like `call/cc` or Felleisen's $\mathcal{C}$ [FFKD87] can be typed by classical tautologies. Therefore we expect that the $\lambda\mu$-calculus can be interpreted operationally such that known control constructs can be encoded as $\lambda\mu$-terms. This is indeed the case—as demonstrated by Ong and Stewart [OS97]

and Bierman [Bie98]. In the following we present Ong and Stewart's call-by-value $\lambda\mu$-calculus, Bierman's small step semantics and an encoding of de Groote's exception handling calculus [dG95] into the $\lambda\mu$-calculus.

3.1 Call-by-Value $\lambda\mu$-Calculus

The $\lambda\mu$-calculus can be extended by datatypes, case-distinction and recursion to form the core of a programming language. This has been done by Ong and Stewart who named the resulting toy language $\mu\mathrm{PCF}^{-}_{\mathrm{V}}$. We will stick to its core $\lambda\mu_{\mathrm{v}}$ which is the $\lambda\mu$-calculus with a call-by-value reduction strategy.

A *value* v is simply a λ-abstraction. We obtain a call-by-value strategy by adding two rules and restricting the β-axiom to values.

$$\begin{aligned}
&(\lambda x.M)\, v \longrightarrow_{\beta_{\mathrm{v}}} [v/x]M \\
&v\,(\mu a.N) \longrightarrow_{\pi_{\mathrm{v}}} \mu b.[\lambda^{\circ} z.[b](v\, z)/a]N \\
&\mu a.[a]M \longrightarrow_{\mu\eta} M \qquad\qquad (a \notin \mu\mathsf{FV}(M))
\end{aligned}$$

A deterministic evaluation strategy for $\lambda\mu_{\mathrm{v}}$ is given by the following small-step semantics.

3.2 A Small-Step Semantics

Bierman [Bie98] gives a simple call-by-value small-step semantics that sheds some light on the meaning of μ-abstraction and μ-application. Before we present it, we define redices and evaluation contexts with a single hole $\bullet$:

$$\begin{aligned}
E &::= \bullet \mid E\, M \mid v\, E && \textit{Evaluation context} \\
R &::= v\, v \mid \mu a.N && \textit{Redex}
\end{aligned}$$

Lemma 1 (Decomposition). *Every λ-closed unnamed term M is either a value or can be uniquely decomposed into an evaluation context $E[\bullet]$ and a redex R such that $M = E[R]$.*

Furthermore, named terms N are always of the form $[a]M$. Hence, using an environment $\mathcal{E}$ which maps μ-variables into evaluation contexts, we can spell out the small-step semantics $\Rightarrow$ by three axioms:

$$\begin{array}{llll}
(E[(\lambda x.M)\, v],\ \mathcal{E}) & & \Rightarrow (E[[v/x]M],\ \mathcal{E}) & \\
(E[\mu a.N],\ & \mathcal{E}) & \Rightarrow (N,\ & \mathcal{E} \uplus \{a \mapsto E[\bullet]\}) \\
([a]M,\ & \mathcal{E} \uplus \{a \mapsto E[\bullet]\}) & \Rightarrow (E[M],\ & \mathcal{E} \uplus \{a \mapsto E[\bullet]\})
\end{array}$$

Note the invariant that the environment $\mathcal{E}$ binds all free μ-variables in M or N.

This semantics suggests that, in the same way as λ-variables are placeholder for terms, μ-variables stand for *contexts* or *continuations*. Binding a μ-variable a means saving the current context in a, and applying a to a term M means restoring the context denoted by a and continuing the evaluation of M in this context. This insight will be critical in finding the best higher-order representation later in this article.

3.3 de Groote's Exception Handling Calculus

To demonstrate the capability of the $\lambda\mu$-calculus to model control we present an SML-like exception handling mechanism given by de Groote [dG95]. Exceptions are added to the λ-calculus via two constructs: one that declares a new exception and provides a handler for it; another one that raises the exception. They are typed as follows:

$$\frac{\begin{array}{cc}\overline{\begin{array}{c}e : \overline{A}\\ \vdots\\ M : B\end{array}} & \overline{\begin{array}{c}x : A\\ \vdots\\ M_e : B\end{array}}\end{array}}{(\mathsf{exception}\, e\ \mathsf{in}\ M\ \mathsf{handle}\ e(x).M_e) : B}\ \text{Handle} \qquad \frac{e : \overline{A} \qquad M : A}{\mathsf{raise}\, e(M) : B}\ \text{Raise}$$

We sketch the desired computational behavior by two examples:

$$\begin{array}{ll}\mathsf{exception}\, e\ \mathsf{in}\ E[\mathsf{raise}\, e(v)]\ \mathsf{handle}\ e(x).M_e & \rightsquigarrow [v/x]M_e\\ \mathsf{exception}\, e\ \mathsf{in}\ v\ \mathsf{handle}\ e(x).M_e & \rightsquigarrow v\end{array}$$

Bierman gives the following translation of the two constructs into $\lambda\mu$-terms. Note that the encoding of raise introduces a β-redex to ensure call-by-value evaluation.

$$\begin{array}{ll}\ulcorner \mathsf{raise}\, e(M) \urcorner & = (\lambda y.\mu\_.[e]y)\ulcorner M\urcorner\\ \ulcorner \mathsf{exception}\, e\ \mathsf{in}\ M\ \mathsf{handle}\ e(x).M_e \urcorner & = \mu a.[a](\lambda x.\ulcorner M_e\urcorner)\,(\mu e.[a]\ulcorner M\urcorner)\end{array}$$

The reader is invited to convince himself that this translation has the stipulated evaluation behavior, e.g. by verifying that

$$(\ulcorner \mathsf{exception}\, e\ \mathsf{in}\ \mathsf{raise}\, e(v)\ \mathsf{handle}\ e(x).x\urcorner, \mathcal{E}) \Rightarrow^* (v, \mathcal{E}').$$

Assuming the reader has gained some familiarity with the $\lambda\mu$-calculus by now, in the next section we shall proceed by discussing its formal representation in a reasoning framework.

4 A Third-Order Representation

Formal reasoning about logics and programming languages, in the following called *object* theories, must take place in a framework, the *meta* theory. A possible choice for such a *logical framework* is predicate logic: first-order terms, made up from function symbols and variables, encode the syntactic entities of the object theory, and predicates represent properties of these entities, for example the wellformedness of a formula or the validity of a proof. However, reasoning about languages with binders in a first-order representation is a tedious business and requires numerous technical lemmata concerning variable renaming, substitution etc. (see e.g. [Sha88], [Alt93]). More suitable is a logical framework with a *higher-order* term language which has binders itself. Then object variables can be encoded by meta variables and substitution in the object theory can be expressed by substitution in the logical framework.

The primary candidate for a higher-order logical framework is the simply-typed λ-calculus $\lambda^{\to}$ with $\beta\eta$-equality, in which object languages with binders can be encoded in a direct way. For example, the untyped λ-calculus can be represented by the following signature.

Base types	tm	*Terms*
Constants	$\mathsf{lam} : (\mathsf{tm} \to \mathsf{tm}) \to \mathsf{tm}$	*Abstraction*
	$\mathsf{app} : \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm}$	*Application*

The representation function $\ulcorner\cdot\urcorner$ maps any untyped λ-term M into a term t of the logical framework. It is defined by recursion over M as follows:

$$\begin{aligned} \ulcorner x \urcorner &= x \\ \ulcorner \lambda x.M \urcorner &= \mathsf{lam}\, \lambda x{:}\mathsf{tm}.\ulcorner M \urcorner \\ \ulcorner M_1\, M_2 \urcorner &= \mathsf{app}\, \ulcorner M_1 \urcorner\, \ulcorner M_2 \urcorner \end{aligned}$$

The given representation is *adequate*, that is, there exists a one-to-one correspondence between untyped λ-terms and their canonical representation in the logical framework.

Theorem 1 (Adequacy). *Let $\Gamma = x_1{:}\mathsf{tm}, \ldots, x_n{:}\mathsf{tm}$ be a context. Then*

1. *for every untyped λ-term M with free variables in $\{x_1, \ldots, x_n\}$ it holds that $\Gamma \vdash \ulcorner M \urcorner : \mathsf{tm}$,*
2. *for every canonical (i.e. β-normal η-long) term t with $\Gamma \vdash t : \mathsf{tm}$ there exists an untyped λ-term M s.th. $\ulcorner M \urcorner = t$, and*
3. *the representation function is* compositional *in the sense that $\ulcorner [M_1/x] M_2 \urcorner = [\ulcorner M_1 \urcorner / x] \ulcorner M_2 \urcorner$.*

Proof. By induction on (1.) M, (2.) t canonical, and (3.) M_2. The cases for abstraction $\lambda x \ldots$ refer to the induction hypothesis with an extended context $(\Gamma, x{:}\mathsf{tm})$.

We observe these benefits of representing the untyped λ-calculus in a higher-order logical framework like $\lambda^{\to}$: (i) α-equivalent object terms M_1 and M_2 translate into equivalent terms of the logical framework and, (ii) substitution does not have to be implemented. More on logical frameworks can be found in [Pfe01b].

4.1 Untyped $\lambda\mu$-Calculus

The untyped $\lambda\mu$-calculus extends the untyped λ-calculus by μ-abstraction and μ-application. How shall we represent the new binder? Let us analyse the general structure of a binder *bind*:

$$\mathit{bind} : (\rho \to \sigma) \to \tau$$

The binder *bind* generates an expression of type τ from a context of type σ with a free variable of type ρ. We know that the binder μ takes a named term N with

a free μ-variable a and returns a term. Thus we add the following constants to our signature.

Base type	nam	*Named terms*
Constant	$\mathsf{mu} : (\rho \to \mathsf{nam}) \to \mathsf{tm}$	*μ-abstraction*

In this signature ρ has to be replaced by the type of the variables that are bound by mu. In the case of lam we had $\rho = \mathsf{tm}$ since λ-variables just stand for terms. The crucial question is: "What do μ-variables stand for?" The small-step semantics suggested that they are placeholder for evaluation contexts. This is confirmed by the nature of the structural substitution which replaces *contexts with contexts*. We make use of the fact that contexts can be represented directly in $\lambda^{\to}$ as functions from terms to terms and set $\rho = \mathsf{tm} \to \mathsf{nam}$. Hence, the type of the constant mu is third-order:

$$\mathsf{mu} : ((\mathsf{tm} \to \mathsf{nam}) \to \mathsf{nam}) \to \mathsf{tm}$$

Surprisingly, we do not have to add a constant for μ-application; it is represented by application in $\lambda^{\to}$. The translation function for $\lambda\mu$-terms M, N and contexts C is an extension of $\ulcorner\cdot\urcorner$ by the following definitions:

$$\begin{array}{lcrcl}
\ulcorner \mu a.N \urcorner & = & \mathsf{mu}\, \lambda a{:}\mathsf{tm} \to \mathsf{nam}. \ulcorner N \urcorner & : & \mathsf{tm} \\
\ulcorner [C]M \urcorner & = & \ulcorner C \urcorner \ulcorner M \urcorner & : & \mathsf{nam} \\
\ulcorner a \urcorner & = & a & : & \mathsf{tm} \to \mathsf{nam} \\
\ulcorner \lambda^{\circ} z.N \urcorner & = & \lambda z{:}\mathsf{tm}. \ulcorner N \urcorner & : & \mathsf{tm} \to \mathsf{nam}
\end{array}$$

Theorem 2 (Adequacy). *Let $\Gamma = x_1{:}\mathsf{tm}, \ldots, x_n{:}\mathsf{tm}, a_1{:}\mathsf{tm} \to \mathsf{nam}, \ldots, a_m{:}\mathsf{tm} \to \mathsf{nam}$. Additionally to an adaption of Thm. 1 for $\lambda\mu$ it holds that*

4. *for every named term N with free variables in Γ it holds that $\Gamma \vdash \ulcorner N \urcorner : \mathsf{nam}$,*
5. *for every canonical term s with $\Gamma \vdash s : \mathsf{nam}$ there exists a named term N s.th. $\ulcorner N \urcorner = s$,*
6. *for every (not necessarily linear) context C with free variables in Γ it holds that $\Gamma \vdash \ulcorner C \urcorner : \mathsf{tm} \to \mathsf{nam}$,*
7. *for every canonical term r with $\Gamma \vdash r : \mathsf{tm} \to \mathsf{nam}$ there exists a (not necessarily linear) context C s.th. $\ulcorner C \urcorner = r$, and,*
8. *the representation is compositional for context substitution, that is,*

$$\ulcorner [C/a]M \urcorner = [\ulcorner C \urcorner / a] \ulcorner M \urcorner .$$

Proof. We prove 1., 4. and 6. simultaneously by induction on $M/N/C$, 2., 5. and 7. by induction on the canonical forms of tm, nam and $\mathsf{tm} \to \mathsf{nam}$, and 8. by induction on M.

Note that we only represented general contexts adequately, not *linear* contexts C. However, all contexts we *de facto* write down in the encoding of reduction rules and small-step semantics will actually be of the required form. In the given

encoding the structural substitution, which looks complicated in the literature (e.g. [OS97]), is just $\lambda^{\rightarrow}$-substitution for μ-variables (see 8.). For example,

$$\ulcorner[\lambda^{\circ}z.[b](z\,y)/a]([a]x)\urcorner = \ulcorner[b](xy)\urcorner = b\,(\mathsf{app}\,x\,y) = (\lambda z{:}\mathsf{tm}\,.b\,(\mathsf{app}\,z\,y))\,x$$
$$= [\lambda z{:}\mathsf{tm}\,.b\,(\mathsf{app}\,z\,y)/a](a\,x) = [\ulcorner\lambda^{\circ}z.[b](z\,y)\urcorner/a]\ulcorner[a]x\urcorner\,.$$

Well-typed terms:

ty	$:\mathsf{type}$	*Types*
tm	$:\mathsf{ty}\to\mathsf{type}$	*Typed terms*
nam	$:\mathsf{type}$	*Named terms*
i	$:\mathsf{ty}$	*Base type*
$\Rightarrow$	$:\mathsf{ty}\to\mathsf{ty}\to\mathsf{ty}$	*Function space*
lam	$:(\mathsf{tm}\,A\to\mathsf{tm}\,B)\to\mathsf{tm}(A\Rightarrow B)$	*λ-abstraction*
app	$:\mathsf{tm}(A\Rightarrow B)\to\mathsf{tm}\,A\to\mathsf{tm}\,B$	*Application*
mu	$:((\mathsf{tm}\,A\to\mathsf{nam})\to\mathsf{nam})\to\mathsf{tm}\,A$	*μ-abstraction*

Reduction:

$\longrightarrow$	$:\mathsf{tm}\,A\to\mathsf{tm}\,A\to\mathsf{type}$	*Red. for typed terms*
$\longrightarrow_{\mathsf{n}}$	$:\mathsf{nam}\to\mathsf{nam}\to\mathsf{type}$	*Red. for named terms*
$\longrightarrow_{\beta}$	$:\Pi M{:}\mathsf{tm}\,A\to\mathsf{tm}\,B.\,\Pi N{:}\mathsf{tm}\,A.$ $\mathsf{app}\,(\mathsf{lam}\,M)\,N\longrightarrow M\,N$	*β-axiom*
$\longrightarrow_{\pi}$	$:\Pi M{:}(\mathsf{tm}(A\Rightarrow B)\to\mathsf{nam})\to\mathsf{nam}.\,\Pi N{:}\mathsf{tm}\,A.$ $\mathsf{app}\,(\mathsf{mu}\,M)\,N\longrightarrow\mathsf{mu}\,\lambda a{:}\mathsf{tm}\,B\to\mathsf{nam}.$ $M\,(\lambda z{:}\mathsf{tm}(A\Rightarrow B).a\,(\mathsf{app}\,z\,N))$	*π-axiom*
$\longrightarrow_{\mathsf{lam}},\ \longrightarrow_{\mathsf{app1}},\ \longrightarrow_{\mathsf{app2}}$	$:\ldots$	lam- *and* app-*congruences*

(i) Direct representation (not adequate):

$\longrightarrow_{\mu\beta}$	$:\Pi a{:}\mathsf{tm}\,A\to\mathsf{nam}.\,\Pi M{:}(\mathsf{tm}\,A\to\mathsf{nam})\to\mathsf{nam}.$ $a\,(\mathsf{mu}\,M)\longrightarrow_{\mathsf{n}} M\,a$	*$\mu\beta$-axiom*
$\longrightarrow_{\mathsf{nam}}$	$:\Pi a{:}\mathsf{tm}\,A\to\mathsf{nam}.\ M\longrightarrow M'$ $\to a\,M\longrightarrow_{\mathsf{n}} a\,M'$	nam-*congruence*
$\longrightarrow_{\mathsf{mu}}$	$:(\Pi a{:}\mathsf{tm}\,A\to\mathsf{nam}.\ N\,a\longrightarrow_{\mathsf{n}} N'a)$ $\to\mathsf{mu}\,N\longrightarrow\mathsf{mu}\,N'$	mu-*congruence*

(ii) Representation with local rules (adequate):

$\longrightarrow_{\mathsf{mu}}$	$:(\Pi a{:}\mathsf{tm}\,A\to\mathsf{nam}.$ $\Pi\longrightarrow_{\mu\beta}:(\Pi M{:}(\mathsf{tm}\,A\to\mathsf{nam})\to\mathsf{nam}.\ a\,(\mathsf{mu}\,M)\longrightarrow_{\mathsf{n}} M\,a).$ $\Pi\longrightarrow_{\mathsf{nam}}:(\Pi M,M'{:}\mathsf{tm}\,A.\ M\longrightarrow M'\to a\,M\longrightarrow_{\mathsf{n}} a\,M').$ $\to N\,a\longrightarrow_{\mathsf{n}} N'a)$ $\to\mathsf{mu}\,N\longrightarrow\mathsf{mu}\,N'$	mu-*congruence*

Table 1. The Signature Σ^{3rd}: Well-Typed Terms and Reduction.

4.2 Typed $\lambda\mu$-calculus

Table 1 lists the signature Σ^{3rd} which encodes the *well-typed* $\lambda\mu$-terms in the dependently typed λ-calculus λ^{Π}—the core of logical frameworks like LF [HHP93]. To represent terms of type A we instantiate the *type family* $\mathsf{tm} : \mathsf{ty} \to \mathsf{type}$ to $\mathsf{tm}\, A$. Note that in the constant declarations A and B are considered to be implicitly quantified. For example, the full type of the constant lam would be $\Pi A{:}\mathsf{ty}.\Pi B{:}\mathsf{ty}.(\mathsf{tm}\, A \to \mathsf{tm}\, B) \to \mathsf{tm}(A \Rightarrow B)$.

Reduction is represented by a pair of type families, $\longrightarrow$ and $\longrightarrow_{\mathsf{n}}$. In part (i), the representation of the two rules $\longrightarrow_{\mu\beta}$ and $\longrightarrow_{\mathsf{nam}}$ is not adequate yet, since context C could be instantiated for a, but only μ-variables should be allowed. This problem can be circumvented by removing these rules and making them *local*: Whenever a new parameter $a{:}\mathsf{tm}\, A \to \mathsf{nam}$ is introduced, add these rules dynamically for this specific a. The only rule that introduces such parameters is $\longrightarrow_{\mathsf{mu}}$; we replace it by the version in part (ii). Adequacy of our representation can be stated and proven in a similar way to Thm. 2.

4.3 Small-Step Semantics

In the following we will refine and encode the small-step semantics $(M, \mathcal{E}) \Rightarrow (M', \mathcal{E}')$ given in Sect. 3.2. The original formulation has a little flaw: during the process of evaluation the environment $\mathcal{E}$ grows monotonically and accumulates contexts that will never be used again. We give a modification that uses substitution instead of environments and includes the decomposition of the subject M into evaluation context C and redex R. The μ-constant eval denotes the top level evaluation context.

Values	$\mathsf{val} \;: \mathsf{tm}\, A \to \mathsf{type}$
$v ::= \lambda x.M$	$\mathsf{vlam} : \Pi M{:}\mathsf{tm}\, A \to \mathsf{tm}\, B.\ \mathsf{val}\,(\mathsf{lam}\, M)$
Evaluation Contexts	
$C ::= \lambda^\circ z.[\mathsf{eval}]z$	$\mathsf{eval} \;: \mathsf{tm}\, A \to \mathsf{nam}$
$\mid \lambda^\circ z.[C](z\, M)$	$\lambda z.\ulcorner C \urcorner\,(\mathsf{app}\, z\, \ulcorner M \urcorner)$
$\mid \lambda^\circ z.[C](v\, z)$	$\lambda z.\ulcorner C \urcorner\,(\mathsf{app}\, \ulcorner v \urcorner\, z)$

Again not each inhabitant of $\mathsf{tm}\, A \to \mathsf{nam}$ stands for a valid evaluation context. However, a judgment $\mathsf{ecxt} : (\mathsf{tm}\, A \to \mathsf{nam}) \to \mathsf{type}$ which singles out the evaluation contexts can be defined easily.

The small-step semantics can be defined by the four rules given below. Formally, it maps one named term into another. We say a term M evaluates to a value v iff $[\mathsf{eval}]M \Rightarrow^* [\mathsf{eval}]v$.

$$\Rightarrow_\beta \frac{}{[C]((\lambda x.M)\, v) \Rightarrow [C]([v/x]M)} \qquad \Rightarrow_{\mu\beta} \frac{}{[C](\mu a.N) \Rightarrow [C/a]N}$$

$$\Rightarrow_{\mathsf{appl}} \frac{[\lambda^\circ z.[C](z\, M_2)]M_1 \Rightarrow N}{[C](M_1\, M_2) \Rightarrow N} \qquad \Rightarrow_{\mathsf{appr}} \frac{[\lambda^\circ z.[C](v\, z)]M \Rightarrow N}{[C](v\, M) \Rightarrow N}$$

On the left hand side of $\Rightarrow$ an occurrence of $[C]M$ is to be read as "term M in context C" but on the right hand side it is just the named term $[C]M$. After each step we decompose the reduct N afresh into $[\mathsf{eval}]M$. This is possible since the following invariant holds: If C and M are closed and $[C]M \Rightarrow N$, then $N = [\mathsf{eval}]M'$ for a closed term M'. Obviously it holds for the rules $\Rightarrow_\beta$, $\Rightarrow_{\mathsf{appl}}$ and $\Rightarrow_{\mathsf{appr}}$. To justify it for the rule $\Rightarrow_{\mu\beta}$ we first notice that the right hand side $[C/a]N$ is a closed named term. Hence, it must be equal to $[c]M'$ for some μ-constant c, which can only be eval.

The representation of "$\Rightarrow$" maps a context-term pair $C\ M$ into another context-term pair $C'\ M'$. The decomposition of a named term into context and term we did silently in the informal treatment is performed by the auxiliary judgment "$\Rightarrow_{\mathsf{n}}$".

$$
\begin{aligned}
\Rightarrow \quad &: (\mathsf{tm}\,A \to \mathsf{nam}) \to \mathsf{tm}\,A \to (\mathsf{tm}\,A' \to \mathsf{nam}) \to \mathsf{tm}\,A' \to \mathsf{type} \\
\Rightarrow_{\mathsf{n}} \quad &: \mathsf{nam} \to (\mathsf{tm}\,A' \to \mathsf{nam}) \to \mathsf{tm}\,A' \to \mathsf{type} \\
\Rightarrow_{\mathsf{eval}} &: \Pi M{:}\mathsf{tm}\,A'.\ (\mathsf{eval}\,M) \Rightarrow_{\mathsf{n}} \mathsf{eval}\ M
\end{aligned}
$$

The four computation rules for "$\Rightarrow$" are given in the following. Note that "$\Rightarrow$" is written infix and appears after two of its arguments.

$$
\begin{aligned}
\Rightarrow_\beta \quad &: \Pi C{:}\mathsf{tm}\,B \to \mathsf{nam}.\ \Pi M{:}\mathsf{tm}\,A \to \mathsf{tm}\,B.\ \Pi V{:}\mathsf{tm}\,A.\ \mathsf{val}\,V \to \\
&\quad (C\,(M\,V)) \Rightarrow_{\mathsf{n}} C'\ M' \to C\ (\mathsf{app}\,(\mathsf{lam}\,M)\,V) \Rightarrow C'\ M' \\
\Rightarrow_{\mu\beta} \quad &: \Pi N{:}(\mathsf{tm}\,A \to \mathsf{nam}) \to \mathsf{nam}.\ \Pi C{:}\mathsf{tm}\,A \to \mathsf{nam}. \\
&\quad (NC) \Rightarrow_{\mathsf{n}} C'\ M' \to C\ (\mathsf{mu}\,N) \Rightarrow C'\ M' \\
\Rightarrow_{\mathsf{appl}} \quad &: (\lambda z.C(\mathsf{app}\,z\,M_2))\ M_1 \Rightarrow C'\ M' \\
&\quad \to C\ (\mathsf{app}\,M_1\,M_2) \Rightarrow C'\ M' \\
\Rightarrow_{\mathsf{appr}} \quad &: (\lambda z.C(\mathsf{app}\,V\,z))\ M \Rightarrow C'\ M' \to \mathsf{val}\,V \\
&\quad \to C\ (\mathsf{app}\,V\,M) \Rightarrow C'\ M'
\end{aligned}
$$

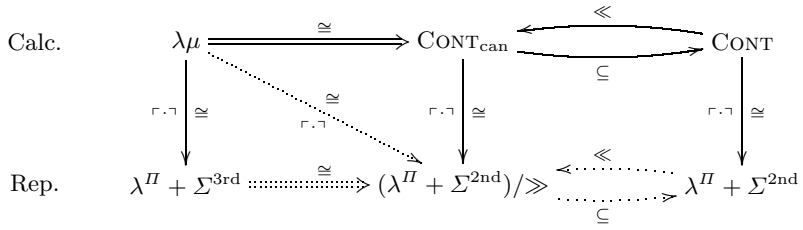
5 A Second-Order Representation

Even though we managed to show that the third-order representation of the $\lambda\mu$-calculus is elegant and adequate, it is not without pitfalls: As we have seen in Sect. 4.2, the direct representation of the reduction relation is not adequate; the rules for named terms have to be locally introduced for a new parameter $a : \mathsf{tm}\,T \to \mathsf{nam}$. This works for ordinary reduction, but cannot be applied to a representation of *parallel reduction*. Consider the rule $\Longrightarrow_{\mu\beta}$:

$$
\begin{aligned}
\Longrightarrow_{\mu\beta} &: \Pi a{:}\mathsf{tm}\,T \to \mathsf{nam}.\,(\Pi b{:}\mathsf{tm}\,T \to \mathsf{nam}.M\,b \Longrightarrow_{\mathsf{n}} M'b) \\
&\quad \to a(\mathsf{mu}\,\lambda b.M\,b) \Longrightarrow_{\mathsf{n}} M'\,a
\end{aligned}
$$

An application of this rule has the following effect: $[a]\mu b.M$ is reduced by the $\mu\beta$-rule, and additional reductions may occur within M. The hypothesis introduces a new parameter b since we step under the binder μ. To make this rule local, it has to be added for each μ-variable that is introduced as a parameter and

Adequacy of the second-order representation of $\lambda\mu$-terms:



Adequacy of the second-order representation of reduction $\longrightarrow$ (Bisimulation):

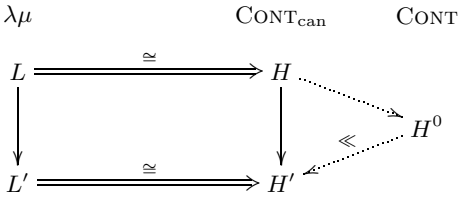


Fig. 1. Overview: Adequacy of the Second-Order Representation

therefore also for b. Thus, $\Longrightarrow_{\mu\beta}$ has to be inserted into itself. This leads to an infinite chain of local rules and cannot be implemented.

These problems do not occur with a second-order representation, which we present in this section. We define the representation function indirectly and also prove adequacy indirectly as outlined in Fig. 1. First, we define the *calculus of continuations* $\textsc{Cont}$, a modification of the $\lambda\mu$-calculus which allows general continuation terms K in a μ-application, now called $\mathsf{throw}\, K\, E$. In principle, we make λ° an explicit constructor klam and do not treat $\longrightarrow_{\beta^\circ}$ silently any more. Furthermore, in the representation Σ^{2nd} we add a base type cont for continuations. Then, we define the *canonical* expressions $H \in \textsc{Cont}_{\text{can}}$ as those which only contain continuation *variables*, that is, no klam. Every expression $H \in \textsc{Cont}$ can be made canonical by applying *normalization* $H \gg H'$.

For the three representations of these calculi we can show adequacy in a straightforward manner. Hence, in each case a one-to-one correspondence exists, and we can restrict ourselves to reason about the original calculi to obtain analogous results for their representations. Particularly, if we construct a bijection between $\lambda\mu$ and $\textsc{Cont}_{\text{can}}$, we implicitly define an adequate second-order representation of $\lambda\mu$.

As a second part, we define reduction in $\textsc{Cont}_{\text{can}}$ and prove that it is the second-order representation of the reduction in $\lambda\mu$. It is sufficient to show bisimulation of reduction in $\lambda\mu$ ($L \longrightarrow L'$) and reduction in $\textsc{Cont}_{\text{can}}$ ($H \longrightarrow H'$) as sketched in Fig. 1. The only case where we have to be careful is $H \longrightarrow_\pi H'$ because applying structural substitution on H temporarily creates a non-canonical

expression H^0 on which we have to apply normalization. Thus, we incorporate normalization into π-reduction.

In the following we will sketch the proof of adequacy of the second-order representation of $\lambda\mu$. The whole proof has been carried out in Twelf [PS98] and is available electronically [Abe01]. We consider it as a case study for formal reasoning about a system by a third-order representation. To my knowledge, this is the first formal proof in a representation of order strictly greater than two. This may be because cases are rare where third- or higher-order representations can be applied.

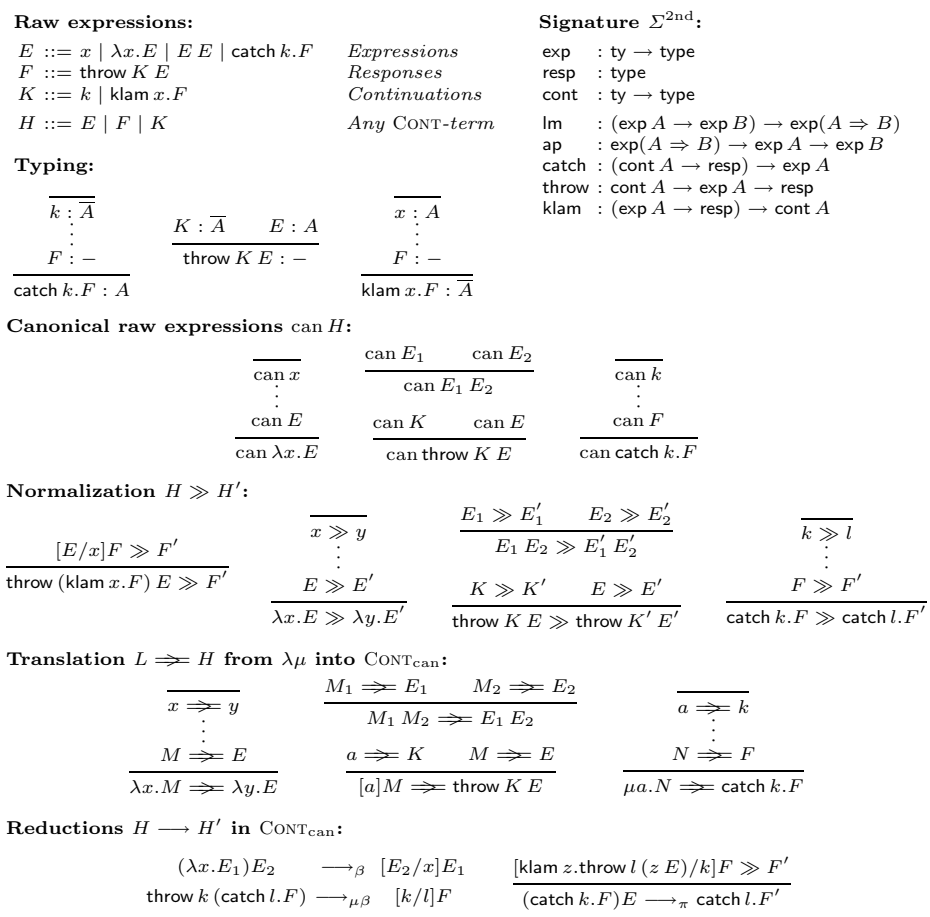


Table 2. Calculus of Continuations CONT and Canonical Fragment CONT_{can}

5.1 The Calculus of Continuations CONT

Table 2 defines expressions, continuations and *responses* (name due to Streicher and Reus [SR98]) of CONT all of which we will refer to as raw expressions H.

Canonical raw expressions H are those which do not contain a klam. The judgment $\mathrm{can}\, H$ is established by recursion on H; there are congruence rules for all constructs except klam. CONT_{can} is the quotient of CONT w.r.t. the equality induced by the axiom $\mathsf{throw}\,(\mathsf{klam}\,x.F)\,E = [E/x]F$. We obtain the canonical representative H' of a CONT-term H by applying the big-step call-by-name normalization procedure $H \gg H'$.

Lemma 2 (Properties of $\gg$).

1. *If* $H_1 \gg H_2$ *then* $\mathrm{can}\, H_2$.
2. *If* $H_1 \gg H_2$ *and* $\mathrm{can}\, H$ *then* $[H/x]H_1 \gg [H/x]H_2$.

Proof (of both assertions). By induction on $H_1 \gg H_2$. □

5.2 Bisimulation

The relation $L \Rrightarrow H$ constitutes a bijective translation between terms of the $\lambda\mu$-calculus L and CONT_{can}-expressions H. The following rectangle theorem (cf. Fig. 1) states that $\lambda\mu$-reductions can be simulated by CONT_{can}-reductions.

Theorem 3 (Simulation). *If* $L \Rrightarrow H$ *and* $L \longrightarrow L'$ *then* $L' \Rrightarrow H'$ *and* $H \longrightarrow H'$ *for some* H'.

Proof. By induction on $L \longrightarrow L'$. The only difficult case is $\longrightarrow_\pi$ for which we need Lemma 3. □

Lemma 3 (Substitution).
If

$$\begin{array}{ccc} a \Rrightarrow k & & x \Rrightarrow z \ll y \\ \vdots & \textit{and} & \vdots \\ L \Rrightarrow H & & C \Rrightarrow D' \ll D \end{array}$$

then

$$[\lambda^\circ x.C/a]L \Rrightarrow H' \ll [\mathsf{klam}\, y.D/k]H$$

Proof. By induction on L. We spell out the hard case $L = [a]M$.

$[a]M \Rrightarrow \mathsf{throw}\, k\, E$	by ass. and def. of $\Rrightarrow$
$M \Rrightarrow E$	by inversion
$[\lambda^\circ x.C/a]M \Rrightarrow E'$	by induction hypothesis
$[\lambda^\circ x.C/a]([a]M) = [[\lambda^\circ x.C/a]M/x]C \Rrightarrow [E'/z]D'$	by assumption
$E' \ll [\mathsf{klam}\, y.D/k]E$	by induction hypothesis
$[E'/z]D' \ll [[\mathsf{klam}\, y.D/k]E/y]D$	by assumption
$[E'/z]D' \ll \mathsf{throw}\,(\mathsf{klam}\, y.D)\,([\mathsf{klam}\, y.D/k]E)$	by def. of $\gg$
$= [\mathsf{klam}\, y.D/k](\mathsf{throw}\, k\, E)$	□

Note that the proof makes use of substitution of deductions. For example, we show $[[\lambda^\circ x.C/a]M/x]C \Rightarrow\!\!\!\!= [E'/z]D'$ by instantiating the deduction $x \Rightarrow\!\!\!\!= z$ of the assumption $C \Rightarrow\!\!\!\!= D'$ with $[\lambda^\circ x.C/a]M \Rightarrow\!\!\!\!= E'$.

Analogously, it can be shown that the reduction in $\textsc{Cont}_{\text{can}}$ simulates the reduction in the $\lambda\mu$-calculus which gives us bisimulation.

6 Conclusion

We have presented the $\lambda\mu$-calculus with an application and discussed two different encodings. The third-order representation seems to fit the $\lambda\mu$-calculus best since all three kinds of substitutions are reduced to substitution of the logical framework. Furthermore, we could formalize the small-step semantics economically. However, a direct representation of parallel reduction fails. The second-order representation is unaffected by these problems but requires auxiliary notions like canonical term and normalization which blow up the proofs considerably in practice.

Two further directions of research open up from here: On the side of the third-order representation, alternative formulations of parallel reduction have to be investigated. A concrete idea is to introduce an auxiliary judgment "C is a μ-variable" which makes the localizations of rules superfluous.

To improve support of the second-order representation, the logical framework could be extended to allow type refinement. The type of canonical expressions would be a refinement (a subtype) of the type of expressions. The fact whether a term H is canonical could be decided and the proof of can H would be *irrelevant* and could be hidden. The theoretical foundations for such an extension of the logical framework have been laid by Pfenning [Pfe01a].

The third-order representation can be used to prove many properties of the $\lambda\mu$-calculus. For a start, I have formally shown soundness of the big-step semantics given by Ong and Stewart [OS97] wrt. to an evaluation-frames-stack small-step semantics. I expect more applications of the encoding in the future.

Acknowledgments. It was Ralph Matthes who first interested me in the $\lambda\mu$-calculus. Frank Pfenning and Brigitte Pientka deserve my gratitude for many discussions and ideas. For comments on the draft I thank Ralph, Frank and the two anonymous referees. Last but not least, I thank the Creator for the framework in which human life and thinking can take place.

References

[Abe01] Andreas Abel. A third-order representation of the $\lambda\mu$-calculus. Twelf code, available under http://www.cs.cmu.edu/~abel/merlin01.tar.gz, 2001.

[Alt93] Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications, TLCA'93*, volume 664 of *Lecture Notes in Computer Science*, pages 13–28. Springer-Verlag, 1993.

[BHF01] Kensuke Baba, Sachio Hirokawa, and Ken-etsu Fujita. Parallel reduction in type free $\lambda\mu$-calculus. In *Proceedings of CATS 2001 (Computing: the Australasian Theory Symposium)*, volume 42 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science B. V., 2001. Also appeared as Technical Report DOI-TR-177, Kyushu University, Fukuoka, Japan.

[Bie98] Gavin M. Bierman. A computational interpretation of the $\lambda\mu$-calculus. In L. Brim, J. Gruska, and J. Zlatuska, editors, *Proceedings of Symposium on Mathematical Foundations of Computer Science*, volume 1450 of *Lecture Notes in Computer Science*, pages 336–345, Brno, Czech Republic, August 1998.

[dG95] Philippe de Groote. A simple calculus of exception handling. In M. Dezani and G. Plotkin, editors, *Second International Conference on Typed Lambda Calculi and Applications, TLCA'95*, volume 902 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 1995.

[dG98] Philippe de Groote. An environment machine for the $\lambda\mu$-calculus. To appear in MSCS, 1998.

[FFKD87] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce F. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987.

[Gri90] Timothy G. Griffin. A formulæ-as-types notion of control. In *Proceedings of the Seventeenth ACM/SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM Press, January 1990.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *Journal of the Association of Computing Machinery*, 40(1):143–184, January 1993.

[OS97] C.-H. L. Ong and C. A. Stewart. A Curry-Howard foundation for functional computation with control. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principle of Programming Languages*, pages 215–227, Paris, Januar 1997. ACM Press.

[Par92] M. Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In A. Voronkov, editor, *Logic Programming and Automated Reasoning: Proc. of the International Conference LPAR'92*, pages 190–201. Springer, Berlin, Heidelberg, 1992.

[Pfe01a] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *LICS 2001: IEEE Symposium on Logic in Computer Science*, June 2001.

[Pfe01b] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers B.V. and MIT Press, 2001. In preparation. Draft available from http://www.cs.cmu.edu/~fp.

[PS98] Frank Pfenning and Carsten Schürmann. Twelf user's guide. Technical report, Carnegie Mellon University, 1998.

[Sha88] Natarajan Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of the ACM*, 35(3):475–522, July 1988.

[SR98] Thomas Streicher and Bernhard Reus. Classical logic, continuation semantics and abstract machines. *Journal of Functional Programming*, 8(6):543–572, 1998.