An Implementation of the Call-By-Name $\lambda\mu\nu$-Calculus

Peter Selinger*
Department of Mathematics
University of Michigan
Ann Arbor, MI 48109-1109

Abstract

We describe an experimental, low-level implementation of the $\lambda\mu\nu$-calculus. The implementation is based on a Krivine-style abstract machine, which is in turns derived from a simple CPS semantics.

Parigot's $\lambda\mu$-calculus was originally invented as a proof-term calculus for classical logic [2]. One can also regard it as a prototypical call-by-name programming language with control primitives. However, its operational significance is not very well understood, particularly because it does not have a very satisfactory rewrite semantics. The $\lambda\mu\nu$-calculus was proposed by Pym and Ritter as an extension of the $\lambda\mu$-calculus with disjunction types [4], again motivated by proof theory.

Here, we hope to clarify the operational meaning of these calculi by giving a Krivine-style abstract machine semantics and an implementation. The meaning of the control operators and of the disjunction types can be explained in the machine model in terms of certain operations on stacks. Our abstract machine semantics is derived from a CPS semantics in the style of Hofmann and Streicher [1]. In [6], we have shown that this CPS semantics is in turns closely related, via a completeness theorem, to the category-theoretical models of the disjunctive $\lambda\mu\nu$-calculus.

We also describe a low-level implementation of the abstract machine. It translates $\lambda\mu\nu$-terms directly into a subset of the C language. The implementation, which is written in ML, is available from [5].

1 Syntax

1.1 The syntax of the $\lambda\mu$-calculus

The $\lambda\mu$-calculus is an extension of the simply-typed lambda calculus with certain control operators. In addition to the usual constructs of the simply-typed lambda calculus, there are two new term constructors: the term $\mu\alpha.M$ statically binds the current continuation to the name α before evaluating M. The term $[\alpha]M$ applies the continuation α to the term M. Thus, as a first approximation, one can think of $\mu\alpha.M$ as $\texttt{callcc}(\lambda\alpha.M)$ in Standard ML of New Jersey or Scheme, and of $[\alpha]M$ as $\texttt{throw}\ \alpha\ M$. However, there are some differences. First, the $\lambda\mu$-calculus is call-by-name, whereas ML and Scheme are call-by-value. The second difference lies in the typing: the $\lambda\mu$-calculus knows an "empty type" $\bot$, which is the type of an expression such as $[\alpha]M$ which never returns. In ML, such an expression has an arbitrary type. Dually, in $\mu\alpha.M$, we require M to be of type $\bot$. The third difference is that the $\lambda\mu$-calculus has two separate name spaces: *variables* $x, y, \ldots$ bind terms, whereas *names* $\alpha, \beta, \ldots$ bind continuations. This distinction, while operationally insignificant, is sometimes technically convenient — for instance, it simplifies the statement of duality in [6]. By convention, we say that an A-accepting continuation α has type A, and not type $A\ \texttt{cont}$ as in ML.

Formally, the $\lambda\mu$-calculus is defined as follows. Types, ranged over by $A, B, \ldots$, are built from a set $\mathcal{B}$ of basic types by the grammar:

$$A ::= \mathcal{B} \mid 1 \mid A \wedge B \mid A \to B \mid \bot$$

Let $\mathcal{V}$ and $\mathcal{N}$ be two given, disjoint, infinite sets of *variables* $x, y, \ldots$ and *names* $\alpha, \beta, \ldots$, respectively. Sometimes the former are called *object variables* and the latter *control variables*. Let $\mathcal{K}$ be a set of typed *constants* $c^A, d^B, \ldots$.

*This research was done while the author was visiting BRICS, Basic Research in Computer Science, Centre of the Danish National Research Foundation.

Table 1: The typing rules for the $\lambda\mu$-calculus

$$(\mathit{var})\ \frac{}{\Gamma, x{:}A \vdash x : A \mid \Delta} \qquad (\mathit{app})\ \frac{\Gamma \vdash M : A \to B \mid \Delta \qquad \Gamma \vdash N : A \mid \Delta}{\Gamma \vdash MN : B \mid \Delta}$$

$$(\mathit{const})\ \frac{}{\Gamma \vdash c^A : A \mid \Delta} \qquad (\mathit{abs})\ \frac{\Gamma, x{:}A \vdash M : B \mid \Delta}{\Gamma \vdash \lambda x^A.M : A \to B \mid \Delta}$$

$$(*)\ \frac{}{\Gamma \vdash * : 1 \mid \Delta} \qquad (\mathit{pass})\ \frac{\Gamma \vdash M : A \mid \alpha{:}A, \Delta}{\Gamma \vdash [\alpha]M : \bot \mid \alpha{:}A, \Delta}$$

$$(\mathit{pair})\ \frac{\Gamma \vdash M : A \mid \Delta \qquad \Gamma \vdash N : B \mid \Delta}{\Gamma \vdash \langle M, N\rangle : A \wedge B \mid \Delta} \qquad (\mu)\ \frac{\Gamma \vdash M : \bot \mid \alpha{:}A, \Delta}{\Gamma \vdash \mu\alpha^A.M : A \mid \Delta}$$

$$(\pi_1)\ \frac{\Gamma \vdash M : A \wedge B \mid \Delta}{\Gamma \vdash \pi_1 M : A \mid \Delta} \qquad (\mathit{weaken})\ \frac{\Gamma \vdash M : A \mid \Delta \qquad \Gamma \subseteq \Gamma' \qquad \Delta \subseteq \Delta'}{\Gamma' \vdash M : A \mid \Delta'}$$

$$(\pi_2)\ \frac{\Gamma \vdash M : A \wedge B \mid \Delta}{\Gamma \vdash \pi_2 M : B \mid \Delta}$$

The pair $(\mathcal{B}, \mathcal{K})$ is called a *signature of the* $\lambda\mu$*-calculus*, and sometimes denoted by Σ. Raw terms $M, N, \ldots$ are given by the grammar:

$$M ::= x \mid c^A \mid * \mid \langle M, N\rangle \mid \pi_1 M \mid \pi_2 M \mid MN \mid \lambda x^A.M \mid [\alpha]M \mid \mu\alpha^A.M$$

A term of the form $\mu\alpha^A.M$ is called a *μ-abstraction*, and a term of the form $[\alpha]M$ is called a *passification term*. In the terms $\lambda x^A.M$ and $\mu\alpha^A.M$, the variable x, respectively the name α, is bound. As usual, we identify raw terms up to renaming of bound variables and names.

The typing of the $\lambda\mu$-calculus is defined as follows. An *object context* $\Gamma = x_1{:}B_1, x_2{:}B_2, \ldots, x_n{:}B_n$ is a finite, possibly empty sequence of pairs of an object variable and a type, such that $x_i \neq x_j$ for all $i \neq j$. We write $\Gamma \subseteq \Gamma'$ if Γ is contained in Γ' as a set. A *control context* $\Delta = \alpha_1{:}A_1, \alpha_2{:}A_2, \ldots, \alpha_m{:}A_m$ is defined analogously. A *typing judgment* is an expression of the form $\Gamma \vdash M : A \mid \Delta$. The context Γ declares the types of the free variables, and Δ declares the types of the free names of M. If there are n variables in Γ and m names in Δ, then one can think of M as a function in n arguments which has $m + 1$ possible result channels. Valid typing judgments are derived by the rules in Table 1.

The reason that we write the control context on the right comes from logic: A sequent

$$x_1{:}B_1, \ldots, x_n{:}B_n \vdash M : A \mid \alpha_1{:}A_1, \ldots, \alpha_m{:}A_m$$

corresponds to a proof of the formula

$$B_1 \wedge \ldots \wedge B_n \Rightarrow A \vee A_1 \vee \ldots \vee A_m.$$

1.2 Adding classical disjunction: The $\lambda\mu\nu$-calculus

Pym and Ritter [4] propose the following straightforward way of adding a disjunction type to the $\lambda\mu$-calculus:

$$\begin{array}{lcl} A & ::= & \ldots \mid A \vee B \\ M & ::= & \ldots \mid \langle\alpha\rangle M \mid \nu\alpha^A.M \end{array}$$

with typing rules:

$$(\mathit{ang})\ \frac{\Gamma \vdash M : A \vee B \mid \alpha{:}A, \Delta}{\Gamma \vdash \langle\alpha\rangle M : B \mid \alpha{:}A, \Delta} \qquad (\nu)\ \frac{\Gamma \vdash M : B \mid \alpha{:}A, \Delta}{\Gamma \vdash \nu\alpha^A.M : A \vee B \mid \Delta}.$$

For technical reasons, these typing rules are the mirror images of those of Pym and Ritter [4]. Like μ-abstraction and passification, these two additional term constructors manipulate continuations. One can think of a term M of type

$A \vee B$ as a term of type B which has access to an unnamed continuation of type A. The term $[\alpha]M$ gives this unnamed continuation the name α. Dually, the term $\mu\alpha^A.M$ abstracts a continuation of name α in M. One way to use these constructs is to circumvent a problem that comes from static scoping: a function can never catch a term that one of its arguments throws. Consider the following term of the $\lambda\mu$-calculus:

$$(\lambda x^B.M(x))(N([\alpha]P))$$

Here, the α of the argument does not lie within the scope of any μ-abstractions that appear in M. Thus, M cannot "catch" (provide a continuation for) the term P that is thrown on α. For this reason, the control operators of the $\lambda\mu$-calculus, like `callcc`, are not suitable to model exception handling. On the other hand, in the $\lambda\mu\nu$-calculus, one can abstract from α as follows:

$$(\lambda x^{A\vee B}.M(\langle\alpha\rangle x))(\nu\alpha'^A.N([\alpha']P))$$

Now it is possible for M to "catch", on the name α, the term P that is thrown by its argument x on the name α'. This usage of the disjunction type $A \vee B$ is reminiscent of the way in which `throws`-clauses provide type-safe exception handling in Java. The operational meaning of the disjunction type will be stated more precisely in the CPS semantics and the abstract machine semantics below.

Pym and Ritter remark that in the call-by-name case, the disjunction type $A \vee B$ is not the same as the disjoint sum type $A + B$ which is customarily defined via `inl`/`inr`/`case` constructs. To distinguish them, we sometimes refer to $A \vee B$ as "classical" disjunction.

We close the section on syntax by remarking that the alternative, symmetric syntax for disjunction from [6] is in fact interdefinable with the syntax given here:

$$\begin{array}{lcl} \nu\alpha^A.M & = & \mu(\alpha^A,\beta^B).[\beta]M \\ \langle\alpha\rangle M & = & \mu\beta^B.[\alpha,\beta]M \end{array} \qquad \text{and} \qquad \begin{array}{lcl} \mu(\alpha^A,\beta^B).M & = & \nu\alpha^A.\mu\beta^B.M \\ [\alpha,\beta]M & = & [\beta]\langle\alpha\rangle M. \end{array}$$

2 The CPS interpretation of the call-by-name $\lambda\mu\nu$-calculus

We adopt the CPS semantics of [6] to the $\lambda\mu\nu$-calculus. The target of the CPS translation is a lambda calculus $\lambda^{R\times+}$ with sum, products, and a distinguished type R of responses. In the target of the translation, function types are restricted to the case $A \to R$, and consequently, lambda abstractions $\lambda x.M$ occur only when M has type R.

To keep the notation brief, we use various forms of syntactic sugar for the sums and products of the target calculus. We use patterned lambda abstraction $\lambda\langle x,y\rangle^{A\times B}.M$, which is customarily defined as an abbreviation for $\lambda z^{A\times B}.M[\pi_1 z/x, \pi_2 z/y]$. We also use the co-pairing notation $[M,N]$ as a shorthand for the expression $(\lambda k^{A+B}.\text{case } k \text{ of inl } k_1\colon Mk_1 \text{ or inr } k_2\colon Nk_2)$. Notice that $[M,N]$ is the term that corresponds to $\langle M,N\rangle$ under the canonical isomorphism $(A+B\to R) \cong (A\to R)\times(B\to R)$. We also use lambda abstraction patterns for co-pairing; thus $\lambda[x,y]^{A+B\to R}.M$ is a shorthand for $\lambda z^{A+B\to R}.M[\lambda a^A.z(\text{inl } a)/x, \lambda b^B.z(\text{inr } b)/y]$. The initial type 0 comes with a type cast operator: If M has type 0, then $\Box_A M$ has type A.

Definition (Call-by-name CPS translation). We assume that the target calculus has a basic type $\tilde{A}$ for each basic type $A \in \mathcal{B}$ of $\lambda\mu$. For each type A of the $\lambda\mu$-calculus, we define a pair of types K_A and C_A of the target calculus, called, respectively, the type of *continuations* and of *computations* of type A:

$$\begin{array}{lcll} K_A & = & \tilde{A}, & \text{if } A \text{ is a basic type,} \\ K_1 & = & 0, & \\ K_{A\wedge B} & = & K_A + K_B, & \\ K_{A\to B} & = & C_A \times K_B, & \\ K_\perp & = & 1, & \\ K_{A\vee B} & = & K_A \times K_B, & \\ C_A & = & K_A \to R. & \end{array}$$

For each variable x and each name α of the $\lambda\mu$-calculus, we assume a distinct variable $\tilde{x}$, respectively $\tilde{\alpha}$, of the target calculus. The call-by-name CPS translation $\underline{M}$ of a typed term M is given in Table 2. It respects the typing in the following sense:

$$\frac{x_1{:}B_1,\dots,x_n{:}B_n \vdash M : A \mid \alpha_1{:}A_1,\dots,\alpha_m{:}A_m}{\tilde{x}_1{:}C_{B_1},\dots,\tilde{x}_n{:}C_{B_n},\tilde{\alpha}_1{:}K_{A_1},\dots,\tilde{\alpha}_m{:}K_{A_m} \vdash \underline{M} : C_A}. \tag{1}$$

Table 2: The CPS translation of the call-by-name $\lambda\mu$-calculus

$$
\begin{array}{lcll}
\underline{x} & = & \lambda k^{K_A}.\tilde{x}k & \text{where } x : A \\
\underline{*} & = & \lambda k^{K_1}.\square_R k & \\
\underline{\langle M,N\rangle} & = & \lambda k^{K_{A\wedge B}}.[\underline{M},\underline{N}]k & \text{where } M : A,\, N : B \\
\underline{\pi_1 M} & = & \lambda k^{K_A}.\underline{M}(\text{inl } k) & \text{where } M : A \wedge B \\
\underline{\pi_2 M} & = & \lambda k^{K_B}.\underline{M}(\text{inr } k) & \text{where } M : A \wedge B \\
\underline{MN} & = & \lambda k^{K_B}.\underline{M}\langle \underline{N}, k\rangle & \text{where } M : A \to B,\, N : A \\
 & & & \\
\underline{\lambda x^A.M} & = & \lambda\langle\tilde{x},k\rangle^{K_{A\to B}}.\underline{M}k & \text{where } M : B \\
\underline{[\alpha]M} & = & \lambda k^{K_\perp}.\underline{M}\tilde{\alpha} & \text{where } M : A \\
\underline{\mu\alpha^A.M} & = & \lambda\tilde{\alpha}^{K_A}.\underline{M}* & \text{where } M : \perp \\
\underline{\langle\alpha\rangle M} & = & \lambda k^{K_B}.\underline{M}\langle\tilde{\alpha},k\rangle & \text{where } M : A \vee B \\
\underline{\nu\alpha^A.M} & = & \lambda\langle\tilde{\alpha},k\rangle^{K_{A\vee B}}.\underline{M}k & \text{where } M : B
\end{array}
$$

This CPS translation, for the fragment without conjunction and disjunction, was discovered by Hofmann and Streicher [1]. It differs from Plotkin's original call-by-name translation [3] by introducing one less double negation at function types, thus taking advantage of the richer target language. For details, see [6].

3 From CPS to an abstract machine

In this section, we describe how to pass from the CPS semantics to an abstract machine semantics. Recall that each continuation type comes with a set of canonical continuation constructors, shown in the following table:

Type:	Constructors:
$K_1 = 0$	$-$
$K_{A\wedge B} = K_A + K_B$	inl k, inr k
$K_{A\to B} = C_A \times K_B$	$\langle M, k\rangle$
$K_\perp = 1$	$*$
$K_{A\vee B} = K_A \times K_B$	$\langle k', k\rangle$.

Here, k ranges over continuations and M over terms. Moreover, we can have basic continuations at basic types. Here, we consider only one such basic continuation called *stop* of type K_A, where A is the type of the entire program. For the machine semantics, it is not necessary that A is a basic type.

Next, we forget the types and change the notation for continuations. We use the infix notation $x{::}k$ for pairing $\langle x, k\rangle$. We also write *inl*$::k$ and *inr*$::k$ for left and right injection. Moreover, we will write *stop* as *nil* and $*$ as $*$::*nil*. Thus, the syntactic class of continuations now looks like this:

$$k ::= \mathit{inl}{::}k \mid \mathit{inr}{::}k \mid M{::}k \mid *{::}\mathit{nil} \mid k'{::}k \mid nil.$$

As the notation suggests, a continuation is really a stack, whose elements are tags of the form *inl*, *inr*, and $*$, terms, and continuations.

In order to avoid substitutions in our abstract machine, we introduce *term closures*. A term closure is a pair M^σ, where M is a term and $\sigma = \langle\sigma_0, \sigma_1\rangle$ is a pair of maps such that σ_0 assigns term closures to the free variables of M, and such that σ_1 assigns continuations to the free names of M. σ is also called an *environment* or an *activation record*.

The states of the abstract machine are triples $\{M, \sigma, k\}$ of a term, an environment, and a stack. Informally, if a closure M^σ represents the term M', then the state $\{M, \sigma, k\}$ represents the term $\underline{M'}k$ of type R of the target language of the CPS transform. The transition rules for our abstract machine can be read directly off the corresponding transitions of the CPS. Both are shown in Table 3. The initial state for a closed program M is $\{M, \emptyset, \mathit{nil}\}$.

Notice that there is no transition from $\{*, \sigma, k\}$. This is because in a well-typed program, such a state can never occur unless $k = \mathit{nil}$, in which case the machine stops with result $*$. More generally, one can show that a well-typed program does not stop until it reaches a state with an empty stack.

Table 3: The transitions of the abstract machine

CPS			Abstract Machine		
$\underline{x}k$	$\to$	$\tilde{x}k$	$\{x,\sigma,k\}$	$\to$	$\{M,\tau,k\}$, where $\sigma_0(x)=M^\tau$.
$\underline{\langle M,N\rangle}(\text{inl } k)$	$\to$	$\underline{M}k$	$\{\langle M,N\rangle,\sigma,\mathit{inl}{::}k\}$	$\to$	$\{M,\sigma,k\}$
$\underline{\langle M,N\rangle}(\text{inr } k)$	$\to$	$\underline{N}k$	$\{\langle M,N\rangle,\sigma,\mathit{inr}{::}k\}$	$\to$	$\{N,\sigma,k\}$
$\underline{\pi_1 M}k$	$\to$	$\underline{M}(\text{inl } k)$	$\{\pi_1 M,\sigma,k\}$	$\to$	$\{M,\sigma,\mathit{inl}{::}k\}$
$\underline{\pi_2 M}k$	$\to$	$\underline{M}(\text{inr } k)$	$\{\pi_2 M,\sigma,k\}$	$\to$	$\{M,\sigma,\mathit{inr}{::}k\}$
$\underline{MN}k$	$\to$	$\underline{M}\langle\underline{N},k\rangle$	$\{MN,\sigma,k\}$	$\to$	$\{M,\sigma,N^\sigma{::}k\}$
$\underline{\lambda x^A.M}\langle N,k\rangle$	$\to$	$\underline{M}[N/\tilde{x}]k$	$\{\lambda x.M,\sigma,N^\tau{::}k\}$	$\to$	$\{M,\sigma(x\mapsto N^\tau),k\}$
$\underline{[\alpha]M}*$	$\to$	$\underline{M}\tilde{\alpha}$	$\{[\alpha]M,\sigma,*{::}\mathit{nil}\}$	$\to$	$\{M,\sigma,k\}$, where $\sigma_1(\alpha)=k$.
$\underline{\mu\alpha.M}k$	$\to$	$\underline{M}[k/\tilde{\alpha}]*$	$\{\mu\alpha.M,\sigma,k\}$	$\to$	$\{M,\sigma(\alpha\mapsto k),*{::}\mathit{nil}\}$
$\underline{\langle\alpha\rangle M}k$	$\to$	$\underline{M}\langle\tilde{\alpha},k\rangle$	$\{\langle\alpha\rangle M,\sigma,k\}$	$\to$	$\{M,\sigma,k'{::}k\}$, where $\sigma_1(\alpha)=k'$.
$\underline{\nu\alpha.M}\langle k',k\rangle$	$\to$	$\underline{M}[k'/\tilde{\alpha}]k$	$\{\nu\alpha.M,\sigma,k'{::}k\}$	$\to$	$\{M,\sigma(\alpha\mapsto k'),k\}$

To sketch a proof of the correctness of the machine, we reintroduce types. Typed term closures and typed continuations are defined by mutual recursion. A typed term closure is a pair $\{\Gamma\vdash M:A\mid\Delta,\sigma\}$, where $\Gamma\vdash M:A\mid\Delta$ is a valid typing judgment and σ is an environment that maps the variables and names from Γ and Δ to closures, respectively stacks, of the appropriate types. Stacks are typed as follows:

$$\frac{k:A}{\mathit{inl}_{A,B}{::}k:A\wedge B}\qquad\frac{k:B}{\mathit{inr}_{A,B}{::}k:A\wedge B}\qquad\frac{k:B}{\{\Gamma\vdash M:A\mid\Delta,\sigma\}{::}k:A\to B}$$

$$\frac{}{*{::}\mathit{nil}:\bot}\qquad\frac{k':A\qquad k:B}{k'{::}k:A\vee B}\qquad\frac{(A\text{ the top-level type})}{\mathit{nil}:A}$$

It is now straightforward to check the following:

1. The initial state $\{M,\emptyset,\mathit{nil}\}$ is typable, if M is a well-typed program.
2. The transitions of the abstract machine preserve types.
3. Every typable state of the abstract machine, for which the current stack is not *nil*, has a unique successor state.
4. The transitions of the abstract machine, starting from state $\{M,\emptyset,k\}$, correspond precisely to the top-most reduction sequence of the term $\underline{M}k$, where one never reduces under a λ.

4 Adding basic types and constants

We now show how to add some meaningful basic types and constants to the language. This complicates the semantics somewhat and leads away from the "pure" call-by-name discipline, because functions that operate on basic types must, by necessity, evaluate their arguments before operating on them. Nevertheless, there is a systematic way of adding basic constants and functions to the CPS semantics and to the abstract machine in the framework of the previous two sections.

4.1 CPS semantics with basic constants

Basic types, such as integers or booleans, differ from other types, because they come with a natural notion of *value*. In the semantics that we are about to give, we assume that basic functions, such as addition or logical "and", evaluate all their arguments before they operate on them. Thus we do not accommodate "lazy" basic functions, such as lazy multiplication, which does not evaluate the second argument if the first argument is zero.

We consider the $\lambda\mu\nu$-calculus over a given algebraic signature, i.e., over a set of basic types $A, B, \ldots$ and a set of basic constants $c : A$ and basic functions $f : B_1 \to \ldots \to B_n \to A$. For the CPS semantics, we consider the same target calculus as before. Moreover, we assume that each basic type A of the $\lambda\mu\nu$-calculus is interpreted by a given type V_A of the target calculus, together with interpretations $\tilde{c} : V_A$, respectively $\tilde{f} : V_{B_1} \to \ldots \to V_{B_n} \to V_A$, of the basic constants and functions. The type V_A is called the type of *values* of type A. We refine our CPS semantics by letting $\tilde{A} = V_A \to R$. Continuation and computation types are defined as before:

$$\begin{array}{lcll} K_A & = & V_A \to R, & \text{if } A \text{ is a basic type,} \\ K_1 & = & 0, & \\ K_{A\wedge B} & = & K_A + K_B, & \\ K_{A\to B} & = & C_A \times K_B, & \\ K_\perp & = & 1, & \\ K_{A\vee B} & = & K_A \times K_B, & \\ C_A & = & K_A \to R. & \end{array}$$

Notice that the value types V_A are only defined at basic types. To the interpretation of terms from Table 2, we add the following interpretation of basic constants $c : A$ and basic functions $f : B_1 \to \ldots \to B_n \to A$:

$$\begin{array}{lcl} \underline{c} & = & \lambda k.k\tilde{c}, \\ \underline{f} & = & \lambda\langle x_1, \ldots, x_n, k\rangle.x_1(\lambda v_1.x_2(\lambda v_2.\ldots x_n(\lambda v_n.k(\tilde{f}v_1\ldots v_n)))). \end{array}$$

Here $k : K_A$, $x_i : C_{B_i}$, and $v_i : V_{B_i}$. Notice that the interpretation of c is actually a special case of that of f for $n = 0$. The reader can convince herself that this CPS semantics behaves indeed as required: The term $fN_1\ldots N_n$ is evaluated by first evaluating all arguments from left to right, and then applying $\tilde{f}$ to the result.

So far, we have accounted only for "pure" basic functions that return basic values. It is also possible to add basic functions to the language whose result type is not a basic type. For instance, if B is the type of booleans, we may consider a family of basic functions $\mathit{if}_A : B \to (A \to A \to A)$ which maps *true* to $\lambda xy.x$ and *false* to $\lambda xy.y$. Thus, $\mathit{if}\,\mathit{true}MN$ reduces to M, and $\mathit{if}\,\mathit{false}MN$ reduces to N. We regard *if* as a unary basic function with return type $A \to A \to A$. For the CPS semantics, we assume that the target language has a type V_B of booleans, and we define $\tilde{\mathit{if}} : V_B \to C_{A\to A\to A}$ by $\tilde{\mathit{if}}\,\mathit{true} = \underline{\lambda xy.x}$ and $\tilde{\mathit{if}}\,\mathit{false} = \underline{\lambda xy.y}$. Then

$$\underline{\mathit{if}} \quad = \quad \lambda\langle x, k\rangle.x(\lambda b.\tilde{\mathit{if}}\,bk)$$

Another useful example of a basic function that returns a term is the *print* function. In call-by-name, one can model sequential composition $N; M$ by application NM, where N is a term that performs some effects and then returns $\lambda x.x$. Now consider a basic function $\mathit{print}_{N,A} : N \to (A \to A)$. The intended meaning is that $(\mathit{print}\,n); M$ prints the integer n and then behaves like M. For the CPS semantics, we need a function $\tilde{\mathit{pr}} : V_N \to R \to R$ of the target language to model the side effect of printing. We then have

$$\underline{\mathit{print}} \quad = \quad \lambda\langle x, k\rangle.x(\lambda n.\tilde{\mathit{pr}}\,n(\underline{\lambda x.x}k)).$$

4.2 Abstract machine semantics with basic constants

We extend the abstract machine semantics to accommodate basic types. In the CPS semantics, we have introduced a new kind of continuations that are functions. We need to fit these new continuations into our "continuations as stacks" paradigm. Fortunately, the CPS semantics only introduces very special kinds of continuation functions, namely ones of the form

$$\lambda v_j.N_{j+1}(\lambda v_{j+1}.\ldots N_n(\lambda v_n.k(\tilde{f}c_1\ldots c_{j-1}v_jv_{j+1}\ldots v_n))),$$

where $1 \leq j \leq n$. These can be represented as a particularly simple kind of closure, which we call a *frame*: we denote the above continuation by

$$[f\,c_1\ldots c_{j-1}\bullet N_{j+1}\ldots N_n]{::}k$$

Of course, we will also replace the terms $N_{j+1}, \ldots, N_n$ by closures.

Recall that a state $\{M, \sigma, k\}$ of our abstract machine corresponds to a term of the form $\underline{M}k$ under CPS. Because we have introduced values to the CPS semantics, we now add a new kind of state $\{c, k\}$, called a *value state*, to the abstract machine. Such a state needs no environment, and it corresponds to a term of the form $k\tilde{c}$ under CPS.

The CPS semantics of the "pure" basic functions has the following transitions:

$$
\begin{array}{lll}
\underline{c}k & \to & k\tilde{c} \\
\underline{f}\langle N_1, \ldots, N_n, k\rangle & \to & N_1(\lambda v_1. \ldots N_n(\lambda v_n.k(\tilde{f}v_1 \ldots v_n))) \\
\multicolumn{3}{l}{(\lambda v_j.N_{j+1}(\lambda v_{j+1}. \ldots N_n(\lambda v_n.k(\tilde{f}c_1 \ldots c_{j-1}v_jv_{j+1} \ldots v_n))))c} \\
 & \to & N_{j+1}(\lambda v_{j+1}. \ldots N_n(\lambda v_n.k(\tilde{f}c_1 \ldots c_{j-1}cv_{j+1} \ldots v_n))), \qquad \text{where } j < n \\
(\lambda v_n.k(\tilde{f}c_1 \ldots c_{n-1}v_n))c & \to & kd, \qquad \text{where } d = \tilde{f}c_1 \ldots c_{n-1}c.
\end{array}
$$

These translate directly to transitions of the abstract machine:

$$
\begin{array}{lll}
\{c, \sigma, k\} & \to & \{c, k\} \\
\{f, \sigma, N_1^{\sigma_1}{::} \ldots {::}N_n^{\sigma_n}{::}k\} & \to & \{N_1, \sigma_1, [f \bullet N_2^{\sigma_2} \ldots N_n^{\sigma_n}]{::}k\} \\
\{c, [f\, c_1 \ldots c_{j-1} \bullet N_{j+1}^{\sigma_{j+1}} \ldots N_n^{\sigma_n}]{::}k\} & \to & \{N_{j+1}, \sigma_{j+1}, [f\, c_1 \ldots c_{j-1}c \bullet N_{j+2}^{\sigma_{j+2}} \ldots N_n^{\sigma_n}]{::}k\} \\
\{c, [f\, c_1 \ldots c_{n-1} \bullet]{::}k\} & \to & \{d, k\},
\end{array}
$$

where $d = \tilde{f}c_1 \ldots c_{n-1}c$ and $j < n$. The rules for the non-pure basic functions *if* and *print* are

$$
\begin{array}{lll}
\{\mathit{true}, [\mathit{if}\bullet]{::}k\} & \to & \{\lambda x.\lambda y.x, \emptyset, k\} \\
\{\mathit{false}, [\mathit{if}\bullet]{::}k\} & \to & \{\lambda x.\lambda y.y, \emptyset, k\} \\
\{c, [\mathit{print}\bullet]{::}k\} & \xrightarrow{\text{output } c} & \{\lambda x.x, \emptyset, k\}
\end{array}
$$

5 Implementing the abstract machine

We give a low-level implementation of the abstract machine. The target language of the implementation is a certain fragment of the C language. This fragment does not use any of the built-in block structure of C, nor function calls (except for a few auxiliary functions which could be implemented by macros), nor the native machine stack. Control is strictly via `goto` and `if-then-else` statements. Thus, the output of the compiler is very close to machine language, except that we ignore issues of register allocation and spilling.

Terms M are represented by program points. The compiled code uses a stack to represent the k-component of a state $\{M, \sigma, k\}$, and a heap to hold term closures and stack closures. A term closure M^σ is a heap-allocated record of $|\sigma| + 1$ words, the first of which holds a pointer to M, and the remaining ones hold pointers to the values of σ. A stack closure is a heap-allocated data structure representing an immutable snapshot of a stack. A stack of size n is represented by a record of $n + 1$ words, the first of which holds the number n, and the remaining ones hold the actual stack data.

On the stack, the tokens $*$, *inl*, and *inr* are represented by the numbers 0, 1, and 2, respectively. Term closures and stack closures are represented by pointers to the heap. *nil* is, of course, the empty stack. A frame $[f\, c_1 \ldots c_{j-1} \bullet N_{j+1}^{\sigma_{j+1}} \ldots N_n^{\sigma_n}]$ is represented as a sequence of elements $\mathit{tag}_f, n, j, c_1, \ldots, c_{j-1}, p_{j+1}, \ldots p_n$ on the stack. Here, tag_f is an integer tag representing the function f, which could be a pointer to code in some implementations. n is the arity of f, j is the position of the $\bullet$ in the frame, $c_1, \ldots, c_{j-1}$ are representations of the respective values, and $p_{j+1}, \ldots, p_n$ are pointers to the closures $N_{j+1}^{\sigma_{j+1}}, \ldots, N_n^{\sigma_n}$.

Within the compiler, terms are represented by judgments $\Gamma \vdash M \mid \Delta$. Here Γ and Δ are functions that assign an *l-value* to each free variable, respectively name, that may occur in M. An l-value is represented as an expression, such as `clos[15]` or `reg5`, which will at runtime yield a pointer to a heap-object. Thus, if x is a variable in the domain of Γ, then $\Gamma(x)$ points to the heap-object that corresponds to x, and similarly for $\Delta(\alpha)$. Γ and Δ are implemented as lists of key-value pairs. If a closure for $\Gamma \vdash M \mid \Delta$ is build, then the order of the variables and names in Γ and Δ determines the order of the data in the activation record.

The translation of a judgment $[\![\Gamma \vdash M \mid \Delta]\!]_v$ is a piece of C-code that ends with a `goto`- or `return`-statement. It depends on an integer v, which is the number of the next unused "register", i.e., global variable. The compiler keeps track globally of the largest index v_{max} that was used by any subterm, and provides the variables $\mathit{reg}_0, \ldots, \mathit{reg}_{v_{max}}$. Thus, we assume that there is a potentially infinite supply of registers; of course, in real machine language, there would be a limited number of registers, and one would have to use spilling to allocate additional values in memory.

Table 4: The compilation of terms

$[\![\Gamma \vdash * \mid \Delta]\!]_v$ = `return "()";`

$[\![\Gamma \vdash x \mid \Delta]\!]_v$ = `clos =` $\Gamma(x)$`;`
`goto jump;`

$[\![\Gamma \vdash \langle M, N\rangle \mid \Delta]\!]_v$ =

```
if (stackempty()) return "pair";
if (pop()==1) {
  [[Γ ⊢ M | Δ]]_v
}
[[Γ ⊢ N | Δ]]_v
```

$[\![\Gamma \vdash \pi_1 M \mid \Delta]\!]_v$ = `push(1);`
$[\![\Gamma \vdash M \mid \Delta]\!]_v$

$[\![\Gamma \vdash \pi_2 M \mid \Delta]\!]_v$ = `push(2);`
$[\![\Gamma \vdash M \mid \Delta]\!]_v$

$[\![\Gamma \vdash MN \mid \Delta]\!]_v$ =

```
/* build closure for N */
tmp=alloc(n + m + 1);
tmp[0] = switch_N;
tmp[1] = Γ(x_1);
...
tmp[n] = Γ(x_n);
tmp[n+1] = Δ(α_1);
...
tmp[n+m] = Δ(α_m);
push(tmp);
[[Γ ⊢ M | Δ]]_v

label_N:
[[{x_i ↦ clos[i]}_i ⊢ N | {α_j ↦ clos[n + j]}_j]]_0
```

(where FV$(N) = x_1, \ldots, x_n$ and FN$(N) = \alpha_1, \ldots, \alpha_m$)

$[\![\Gamma \vdash \lambda x.M \mid \Delta]\!]_v$ =

```
if (stackempty()) return "fn";
reg_v = pop();
[[x:reg_v, Γ ⊢ M | Δ]]_{v+1}
```

$[\![\Gamma \vdash [\alpha]M \mid \Delta]\!]_v$ =

```
if (stackempty()) return "pass";
loadstack(Δ(α));
[[Γ ⊢ M | Δ]]_v
```

$[\![\Gamma \vdash \mu\alpha.M \mid \Delta]\!]_v$ =

```
reg_v = savestack();
push(0);
[[Γ ⊢ M | α:reg_v, Δ]]_{v+1}
```

$[\![\Gamma \vdash \langle\alpha\rangle M \mid \Delta]\!]_v$ =

```
push(Δ(α));
[[Γ ⊢ M | Δ]]_v
```

$[\![\Gamma \vdash \nu\alpha.M \mid \Delta]\!]_v$ =

```
if (stackempty()) return "nu";
reg_v = pop();
[[Γ ⊢ M | α:reg_v, Δ]]_{v+1}
```

Table 5: The compilation of basic constants and functions

$[\![\Gamma \vdash c \mid \Delta]\!]_v$ =
```
value = c;
goto intvalue;
```

$[\![\Gamma \vdash \mathit{true} \mid \Delta]\!]_v$ =
```
value = 1;
goto boolvalue;
```

$[\![\Gamma \vdash \mathit{false} \mid \Delta]\!]_v$ =
```
value = 0;
goto boolvalue;
```

$[\![\Gamma \vdash f \mid \Delta]\!]_v$ =
```
if (stackheight()<n) return "fn";
/* build frame for f */
clos = pop();
push(1);
push(n);
push(tag_f);
goto jump;
```

The translation of terms is defined in Tables 4 and 5, and the translation of a closed program is shown in Table 6. We have slightly simplified the translation here by omitting type casts between integers and pointers, as well as those parts of the code that are responsible for producing verbose output.

The global variable `clos` is used to hold a pointer to the current closure during a jump to a "regular" state (not a value-state). At the destination of the jump, execution will resume relative to this closure. In this implementation, jumps proceed in two steps: all jumps first go to a global "dispatch" section, labeled "`jump`" in Table 6, which reads `clos[0]` and jumps to the appropriate piece of code. Thus, we distinguish between a *switch*, which is an internal representation of a program point, and a *label*, which is a pointer to the actual code. This trick is necessary because in C, labels are not first-class objects. However, in an actual machine-language implementation, one could represent switches directly as pointers to code.

The implementation knows two basic types: integers and booleans. There are two special program points `boolvalue` and `intvalue`, also shown in Table 6, which correspond to value states of the abstract machine. At these program points, the global variable `value` is assumed to hold the current value. The translation of constants is shown in Table 5, where c is an integer constant, and f is an n-ary function constant.

In addition, each basic function has a handler which pops the appropriate values from the stack and performs the required calculation. The handlers for a generic unary function f and a binary function g on integers are shown in Table 7, as well as the handlers for `if` and `print`. The implementation also provides a function `printbool` which acts like `print`, except that it prints a boolean.

5.1 The runtime system

The compiled code relies on a small runtime system, which provides some auxiliary functions for handling the stack and the heap, as well as code for runtime errors, conversions, and reading command-line arguments. In a real implementation, these functions would be expanded to in-line code.

- `void push(void *)` and `void *pop(void)`: push, respectively pop, data to and from the stack.
- `int stackempty()`: tests whether the stack is empty.
- `int stackheight()`: returns the current size of the stack.
- `void *getstack(int n)`: gets the nth element from the stack, where $n = 0$ is the topmost element.
- `void modifystack(int n,void *x)`: sets the nth element of the stack to x.

Table 6: The compilation of a closed program

$[\![M]\!] \quad =$

```
#include "runtime.c"

char *compiled() {
  void **clos, **tmp;
  int value, arity, j, ftag;
  void *reg_0;
  ...
  void *reg_{v_max};

  [[∅ ⊢ M | ∅]]_0

  jump: switch(clos[0]) {
    case switch_{N_0}: goto label_{N_0};
    ...
    case switch_{N_k}: goto label_{N_k};
    default: runtimeerr("unknown label");
  }

  boolvalue:
    if (stackempty()) return value ? "true" : "false";
    goto applyfunction;

  intvalue:
    if (stackempty()) return int2str(value);

  applyfunction:
    arity = getstack(1);
    j = getstack(2);
    if (j<arity) {
      clos = getstack(2+j);
      modifystack(2+j,value);
      modifystack(2,j+1);
      goto jump;
      }
    ftag = pop();
    pop();
    pop();
    switch (ftag) {
    ...
    /* handlers for basic functions */
    ...
    default:
      runtimeerr("unknown function tag");
    }
}
```

Table 7: Handlers for some basic functions

case tag_f:
  value = f(value);
  goto intvalue;

case tag_g:
  reg_0 = pop();
  value = $g(reg_0, \texttt{value})$;
  goto intvalue;

case tag_{if}:
  if (value) {
    $[\![\emptyset \vdash \lambda x.\lambda y.x \mid \emptyset]\!]_0$
  }
  $[\![\emptyset \vdash \lambda x.\lambda y.y \mid \emptyset]\!]_0$

case tag_{print}:
  output(int2str(value));
  $[\![\emptyset \vdash \lambda x.x \mid \emptyset]\!]_0$

- `void *savestack(void)`: creates a new stack closure from the current stack, resets the stack, and returns a pointer to the new closure.
- `void loadstack(void *addr)`: restores the stack from the stack closure found at `addr`.
- `void *alloc(int n)`: allocates n consecutive words on the heap and returns a pointer to them.
- `void runtimeerr(char *)`: issues a runtime error and aborts execution.
- `char *int2str(int n)`: converts an integer to a string.
- `void output(char *s)`: writes the string s to standard output.
- `main()`: reads the command-line arguments and calls the compiled function.

In addition, there are a few routines that handle verbose output.

5.2 Running the compiler

The compiler is written in ML. It takes $\lambda\mu\nu$-terms to C code. No parsing or type-checking is performed in this experimental implementation, but the correctness of the implementation depends on the typability of the source term. $\lambda\mu\nu$-terms are represented as elements of the abstract datatype `term`:

```
type var = string;
type fnconst = string;
datatype term = Var of var | Unit | Pair of term*term | Proj1 of term
        | Proj2 of term | App of term*term | Lam of var*term
        | Pass of var*term | Mu of var*term | Ang of var*term
        | Nu of var*term | Intconst of int | True | False
        | Fnconst of fnconst;
```

Here, a variable x is represented as `Var "x"`. Application MN is `App(M,N)`, lambda abstraction $\lambda x.M$ is `Lam("x",M)`, and similar for μ- and ν-abstraction. Passification $[\alpha]M$ is `Pass("a",M)`, and $\langle\alpha\rangle M$ is `Ang("a",M)`. Integer and boolean constants are represented as `Intconst n`, `True`, and `False`, and basic functions as `Fnconst "+"`. The available basic functions are

`+`, `-`, `*`, `/`, `succ`, `~`, `and`, `or`, `not`, `iszero`, `<`, `>`, `=`, `>=`, `=<`, `if`, `print`, `printbool`

One can add additional basic functions by editing the definition of `fnconstdef` in the ML code.

The compiler is available in three dialects of ML: SML/NJ 0.93, Caml Light, and Objective Caml. The current version can be found at [5]. To load the compiler, type

`use "compiler.sml";`	from SML,
`include "compiler.ml";;`	from Caml Light, or
`#use "compiler.oml";;`	from Objective Caml.

Then type `compile term "myfile.c";;` to generate a C-program from a term `source`. The output is written to the file `myfile.c`. This can be compiled with any C-compiler, for instance by typing `gcc myfile.c -o myfile`. Make sure that the file `runtime.c` is in the same directory, because it is included. Finally, execute the code via `myfile [-v]`.

The `-v` option causes the program to write detailed information about the evaluation to standard output. In this mode, a representation of each state of the underlying abstract machine is printed, as well as information on each new term or stack closure that is created.

References

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

[2] M. Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In *Proceedings of the International Conference on Logic Programming and Automated Reasoning, St. Petersburg*, Springer LNCS 624, pages 190–201, 1992.

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

[4] D. Pym and E. Ritter. On the semantics of classical disjunction. Preprint, 1998.

[5] P. Selinger. An implementation of the call-by-name $\lambda\mu\nu$-calculus. Available from http://www.math.lsa.umich.edu/~selinger/lammunu/, 1998.

[6] P. Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Math. Struct. in Computer Science*, 2000. To appear.