From Continuation Passing Style to Krivine's Abstract Machine

Peter Selinger
Department of Mathematics and Statistics
University of Ottawa
Ottawa, ON K1N 6N5, Canada

Abstract

We describe, for three different extensions of typed lambda calculus, how the rules for a version of Krivine's abstract machine can be derived from those of continuation passing style (CPS) semantics. The three extensions are: Parigot's $\lambda\mu$-calculus, Pym and Ritter's $\lambda\mu\nu$-calculus, and an extension of the call-by-name lambda calculus with built-in types and primitive functions. We also show how Krivine's abstract machine can be implemented on realistic hardware by compiling it into an idealized assembly language.

1 Introduction

Abstract machines play an important role in the implementation of programming languages. Examples include Warren's 1983 abstract machine for Prolog, which is the basis for most modern Prolog implementations [12], and Cousineau's 1990 categorical abstract machine for ML, on which the original Caml implementation was based (and from which it derives its name) [2]. The reason abstract machines are so useful is because, on the one hand, they are sufficiently "abstract" to relate easily to other kinds of mathematical semantics, such as equational semantics or continuation passing style (CPS) semantics. On the other hand, they are sufficiently "machine-like" to be easily implementable on real machines.

A particularly nice example of an abstract machine is Krivine's machine for the call-by-name lambda calculus [6]. In this paper, we show how it is possible to "derive" the rules of Krivine's abstract machine, in a semi-formal but systematic way, from a CPS semantics in the style of Hofmann and Streicher [5]. We do this for three extensions of the lambda calculus: the $\lambda\mu$-calculus, the $\lambda\mu\nu$-calculus, and an extension of lambda calculus with built-in basic types and primitive functions. For each of these extensions, we also give an implementation of Krivine's abstract machine, which takes the form of a compiler into an idealized assembly language.

It is interesting to note that Hofmann and Streicher's CPS semantics can itself be derived, via a categorical completeness theorem, from a yet more abstract category-theoretical semantics. This semantics is based on the interpretation of the $\lambda\mu\nu$-calculus

in a so-called *control category*, and it generalizes the familiar interpretation of the simply-typed lambda calculus in cartesian-closed categories [10]. Thus, one obtains the following sequence of constructions, leading systematically from the very abstract to the very concrete:

$$\text{Categorical Semantics} \to \text{CPS Semantics} \to \text{Abstract Machine} \to \text{Compiler}$$

Krivine's abstract machine therefore fits nicely into a multi-step process for designing implementations which are essentially "correct by contruction", relative to a given high-level semantics. In this paper, we only consider the last two steps in this sequence; the first step, namely the relationship between the categorical semantics and the CPS semantics, is discussed elsewhere [10].

It should be stressed that, from a practical point of view, the implementation of the call-by-name lambda calculus derived in this paper is too inefficient to be of much use. Because our implementation follows the design of Krivine's abstract machine very closely, it embodies a "naive" version of call-by-name evaluation, in which each subterm is possibly evaluated many times. More realistic implementations of call-by-name languages typically use a form of "lazy" evaluation to avoid this problem.

The development of CPS semantics, abstract machine, and a compiler, as presented in this paper, is a rational reconstruction and does not reflect the historical development of these concepts. As a matter of fact, Krivine's formulation of his abstract machine predates the CPS semantics of Hofmann and Streicher, which in turns predates the categorical semantics in terms of control categories. Also, the connection between continuation semantics and abstract machines is well-known; for example, a treatment in the context of denotational semantics was given in [11]. We do not claim originality for any of the results presented in this article; rather, we hope to present them under a unique and unifying point of view.

2 The $\lambda\mu$-calculus

The $\lambda\mu$-calculus was originally introduced by Parigot as a proof-term calculus for classical logic [7]. Following Griffin's earlier work, who showed that under the Curry-Howard isomorphism, classical logic corresponds to languages with control operators [4], the $\lambda\mu$-calculus can also be regarded as a prototypical call-by-name language with control primitives for handling continuations. In this respect, it is similar to programming languages with $callcc$ or Felleisen's $\mathcal{C}$ operator [3], except that the latter languages are call-by-value. The rewrite semantics of the $\lambda\mu$-calculus is not very intuitive, and Krivine's abstract machine offers a more easily accessible way to understand its operational behavior. The control primitives are given a natural interpretation as certain manipulations of stack closures.

The $\lambda\mu$-calculus extends the simply-typed lambda calculus with a pair of control operators which can influence the sequential flow of control during the evaluation of a term. Normally, in call-by-name, a term M represents a computation which, upon demand, returns some result to its environment. For instance, if the term M appears in a context $C[-]$, then the result which M computes will be returned to $C[-]$.

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

$$(\textit{var})\quad \frac{}{\Gamma \vdash x : A \mid \Delta} \quad \text{if } x{:}A \in \Gamma$$

$$(*)\quad \frac{}{\Gamma \vdash * : \top \mid \Delta}$$

$$(\textit{pair})\quad \frac{\Gamma \vdash M : A \mid \Delta \qquad \Gamma \vdash N : B \mid \Delta}{\Gamma \vdash \langle M, N\rangle : A \wedge B \mid \Delta}$$

$$(\pi_i)\quad \frac{\Gamma \vdash M : A_1 \wedge A_2 \mid \Delta}{\Gamma \vdash \pi_i M : A_i \mid \Delta}$$

$$(\textit{app})\quad \frac{\Gamma \vdash M : A \to B \mid \Delta \qquad \Gamma \vdash N : A \mid \Delta}{\Gamma \vdash MN : B \mid \Delta}$$

$$(\textit{abs})\quad \frac{\Gamma, x{:}A \vdash M : B \mid \Delta}{\Gamma \vdash \lambda x^A.M : A \to B \mid \Delta}$$

$$(\textit{name})\quad \frac{\Gamma \vdash M : A \mid \Delta}{\Gamma \vdash [\alpha]M : \bot \mid \Delta} \quad \text{if } \alpha{:}A \in \Delta$$

$$(\mu)\quad \frac{\Gamma \vdash M : \bot \mid \alpha{:}A, \Delta}{\Gamma \vdash \mu\alpha^A.M : A \mid \Delta}$$

In the $\lambda\mu$-calculus, terms are given the ability to ignore their immediate context and to return a result someplace else. Intuitively, this can be thought of as "sending" a result on a "channel". We introduce a set of channel names α, β, etc., which are distinct from the usual lambda calculus variables x, y, z. The term $[\alpha]M$ causes the result of M to be sent on channel α. Dually, the term $N = \mu\alpha^A.P$ will start by evaluating P, but if in the process of doing so, anything is sent on the channel α, then this immediately becomes the result of N. Channel names are typed, and we say that a channel α has type A if values of type A can be sent along it. As we are in a sequential world, channels are refered to as *continuations*, and channel names α, β are refered to as *control variables*, or simply *names*.

This first interpretation of the $\lambda\mu$-calculus in terms of "channels" is only an intuitive approximation; a more accurate interpretation can be found in the description of the CPS translation or Krivine's abstract machine below. Modulo some minor differences in typing, the term $\mu\alpha^A.M$ is a call-by-name analogue of $\mathit{callcc}(\lambda\alpha^{\neg A}.M)$ in the call-by-value world, where callcc is the call-with-current-continuation operator as it appears for instance in Scheme or Standard ML.

2.1 Syntax

We start from a simply-typed lambda calculus with finite products. Binary products are denoted $A \wedge B$, and the terminal type (or empty product) is denoted by $\top$.

To obtain the $\lambda\mu$-calculus, we first add a new type $\bot$. The type $\bot$ is thought of as the "empty type", or the type of a term which never returns a result to its immediate

context. Thus, the types of the $\lambda\mu$-calculus are given as follows, where σ ranges over a set of *basic types*:

$$A, B ::= \sigma \mid \top \mid A \wedge B \mid A \to B \mid \bot$$

As usual, we sometimes write $\neg A$ as an abbreviation for the type $A \to \bot$. The $\lambda\mu$-calculus uses two sets of identifiers, *variables* and *names*, which are ranged over by $x, y, \ldots$ and $\alpha, \beta, \ldots$, respectively. Variables and names belong to two separate name spaces, which are usually assumed to be disjoint. Semantically, variables are bound to terms, whereas names are bound to continuations. The terms of the $\lambda\mu$-calculus are obtained from the terms of the simply-typed lambda calculus by adding two new term constructors, $[\alpha]M$ and $\mu\alpha^A.M$. Thus, terms are given as follows:

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

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 *named term*. In the terms $\lambda x^A.M$ and $\mu\alpha^A.M$, the variable x, respectively the name α, is bound. As usual, terms are identified up to capture-free renaming of bound variables and names. We write FV(M) and FN(M) for the set of free variables and free names of M, respectively. For simplicity, we do not consider basic term constants at this point; we will show how to add them in Section 4.

The typing rules for $\lambda\mu$-terms are shown in Table 1. Here Γ ranges over *variable contexts* and Δ ranges over *name contexts*, which are (finite) assignments of types to variables and names, respectively. A *typing judgment* is an expression of the form $\Gamma \vdash M : A \mid \Delta$. It asserts that the term M is well-typed of type A, assuming that its free variables and names have the types declared in Γ, respectively Δ. Note that the turnstile "$\vdash$", the colon ":", and the vertical bar "$|$" are all part of the syntax of typing judgments; thus, a typing judgment is a 4-tuple consisting of a variable context, a term, a type, and a name context. Valid typing judgments are those which can be derived from the rules in Table 1.

Note the typing rules for (*name*) and (μ). The term $[\alpha]M$ has type $\bot$, reflecting the fact that such a term never returns anything to its immediate environment. Similarly, in the term $\mu\alpha^A.M$, we assume that the subterm M has type $\bot$, as we have no use for its value. These typing conventions differ slightly from Parigot's original formulation of the $\lambda\mu$-calculus, where the type $\bot$ only occured implicitly, and only at the top level.

One notable difference between the $\lambda\mu$-calculus and its call-by-value cousins is that we use a separate name space for continuations, rather than identifying them with variables of type $\neg A$ (or A `cont`, as this type is known in ML). While this distinction would make no difference in call-by-value, it turns out to be an important optimization in call-by-name.

Another difference is that in ML, the term which is analogous to $[\alpha]M$ would be given an arbitrary type B, and in $\mu\alpha^A.M$, the subterm M would have type A. However, this difference is unimportant, as we can replace the first term by $\mu\beta^B.[\alpha]M$, for a dummy name β, and the second one by $\mu\alpha^A.[\alpha]M$, in cases where the alternate typing is required.

The fact that we write the name context Δ on the right-hand side of a typing judgment is motivated by logic: under the formulas-as-types correspondence, a typing judg-

Table 2: Axioms of the call-by-name $\lambda\mu$-calculus

Axioms for the lambda calculus with products:				
$(\beta_\to)$	$(\lambda x^A.M)N$	$=$	$M[N/x] : B$	
$(\eta_\to)$	$\lambda x^A.Mx$	$=$	$M : A \to B$	if $x \notin \mathrm{FV}(M)$
$(\beta_\wedge)$	$\pi_i\langle M_1, M_2\rangle$	$=$	$M_i : A_i$	
$(\eta_\wedge)$	$\langle \pi_1 M, \pi_2 M\rangle$	$=$	$M : A \wedge B$	
$(\eta_\top)$	$*$	$=$	$M : \top$	
Axioms for $\lambda\mu$:				
$(\zeta_\to)$	$(\mu\alpha^{A\to B}.M)N$	$=$	$\mu\beta^B.M[[\beta](-)N/[\alpha](-)] : B$	if $\beta \notin \mathrm{FN}(M,N)$
$(\zeta_\wedge)$	$\pi_i(\mu\alpha^{A_1\times A_2}.M)$	$=$	$\mu\beta^{A_i}.M[[\beta]\pi_i(-)/[\alpha](-)] : A_i$	if $\beta \notin \mathrm{FN}(M)$
(β_μ)	$[\alpha']\mu\alpha^A.M$	$=$	$M[\alpha'/\alpha] : \bot$	
(η_μ)	$\mu\alpha^A.[\alpha]M$	$=$	$M : A$	if $\alpha \notin \mathrm{FN}(M)$
$(\beta_\bot)$	$[\xi^\bot]M$	$=$	$M : \bot$	

ment $x_1{:}A_1, \ldots, x_n{:}A_n \vdash M : B \mid \alpha_1{:}B_1, \ldots, \alpha_m{:}B_m$ corresponds to a logical implication $A_1 \wedge \ldots \wedge A_n \Rightarrow B \vee B_1 \vee \ldots \vee B_m$. Operationally, we think of M as a function in n arguments, with $m+1$ alternative ways of returning a result.

2.2 Equational theory

The equational theory of the $\lambda\mu$-calculus is an extension of that of the call-by-name lambda calculus. The axioms are shown in Table 2. These axioms use three kinds of substitution. We write $M[N/x]$ for the usual substitution of a term N for a variable x in M. We write $M[\alpha'/\alpha]$ for the substitution of a name α' for another name α in M. Finally, we consider the so-called *mixed substitution*: If M is a term, $C(-)$ is a context, and α is a name, then the *mixed substitution* $M[C(-)/[\alpha](-)]$ is the result of recursively replacing any subterm of the form $[\alpha](-)$ by $C(-)$ in M. For all three kinds of substitution, appropriate care must be taken to avoid the capture of free variables. Also note that technically, each equation $M = N$ is understood to be stated within a particular typing context, and equations are only between well-typed terms. However, we usually omit the typing context from the notation. For more details, see e.g. [10].

It is possible to give an operational semantics of the $\lambda\mu$-calculus in terms of a reduction relation based on a directed version of the axioms of Table 2. However, this notion of reduction is neither intuitive nor particularly enlightening. We prefer to discuss the operational semantics of the $\lambda\mu$-calculus in terms of a CPS translation (in Section 2.3) and via an abstract machine (in Section 2.4).

Table 3: The CPS translation of the call-by-name $\lambda\mu$-calculus

$\underline{x}$	$=$	$\lambda k^{K_A}.\tilde{x}k$	where $x : A$
$\underline{*}$	$=$	$\lambda k^{K_\top}.\Box_R k$	
$\underline{\langle M, N\rangle}$	$=$	$\lambda k^{K_{A\wedge B}}.[\underline{M}, \underline{N}]k$	where $M : A$, $N : B$
$\underline{\pi_1 M}$	$=$	$\lambda k^{K_A}.\underline{M}(\mathrm{inl}\ k)$	where $M : A \wedge B$
$\underline{\pi_2 M}$	$=$	$\lambda k^{K_B}.\underline{M}(\mathrm{inr}\ k)$	where $M : A \wedge B$
$\underline{MN}$	$=$	$\lambda k^{K_B}.\underline{M}\langle \underline{N}, k\rangle$	where $M : A \to B$, $N : A$
$\underline{\lambda x^A.M}$	$=$	$\lambda\langle \tilde{x}, k\rangle^{K_{A\to B}}.\underline{M}k$	where $M : B$
$\underline{[\alpha]M}$	$=$	$\lambda k^{K_\bot}.\underline{M}\tilde{\alpha}$	where $M : A$
$\underline{\mu\alpha^A.M}$	$=$	$\lambda\tilde{\alpha}^{K_A}.\underline{M}*$	where $M : \bot$

2.3 CPS semantics

We give a continuation passing style (CPS) semantics of the $\lambda\mu$-calculus in the style of Hofmann and Streicher [5]. The target language of this CPS translation is a lambda calculus $\lambda^{R\times+}$ with finite sums, products, and a distinguished type R, called the type of *responses*. Function types in the target calculus are restricted to the case $A \to R$. Thus, every application MN in the target calculus is of type R, as is the body of any lambda abstraction.

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 $\lambda\langle x, y\rangle^{A\times B}.M$ 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 an abbreviation for the term

$$\lambda k^{A+B}.\mathrm{case}\ k\ \mathrm{of}\ \mathrm{inl}\ k_1 \to Mk_1 \mid \mathrm{inr}\ k_2 \to Nk_2.$$

Notice that $[M, N]$ is the term that corresponds to $\langle M, N\rangle$ under the canonical isomorphism $(A + B) \to R \cong (A \to R) \times (B \to R)$. The initial type 0 is equipped 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 chosen type $\tilde{\sigma}$ for each basic constant σ of the $\lambda\mu$-calculus. For each type A of the $\lambda\mu$-calculus, we define a pair of types K_A and C_A of the target calculus, which are respectively called the type of *continuations* and of *computations* of type A:

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

For each variable x and each name α of the $\lambda\mu$-calculus, we assume a distinct chosen variable $\tilde{x}$, respectively $\tilde{\alpha}$, of the target calculus. The call-by-name CPS translation $\underline{M}$

of a typed term M is defined in Table 3. It respects the typing in the following sense:

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

This CPS translation, for the fragment without product types, is due to Hofmann and Streicher [5]. It differs from Plotkin's original call-by-name translation [8] by introducing one less double negation at function types, thus taking advantage of the products of the target language.

The CPS translation respects the equational theory in the sense that $M = N$ holds in the equational theory of the $\lambda\mu$-calculus if and only if $\underline{M} = \underline{N}$ holds in the equational theory of the target calculus.

Remark. The above CPS translation for the $\lambda\mu$-calculus can be derived abstractly, via a categorical representation theorem, from a category-theoretic interpretation of the $\lambda\mu$-calculus. This interpretation takes place in a class of so-called "control categories", and it generalizes the well-known interpretation of the simply-typed lambda calculus in cartesian-closed categories. For details, see [10].

2.4 From the CPS semantics to Krivine's abstract machine

In this section, we describe a rational reconstruction of Krivine's abstract machine directly from the CPS semantics, adopted to the $\lambda\mu$-calculus. Note that an abstract machine interpretation was already sketched in the very last paragraph of Parigot's original paper on the $\lambda\mu$-calculus [7].

We start by observing that each continuation type K_A is equipped with a set of canonical term constructors, shown in the following table. Here, k ranges over continuations and M over computations.

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

There is also a *top-level continuation* κ, which is the first continuation passed (presumably by the operating system) to the entire program.

Next, we change the notation for continuations. A pair $\langle M, k\rangle$ will be written in infix notation $M{::}k$. Instead of inl k and inr k, we will write $\mathit{tag}_1{::}k$ and $\mathit{tag}_2{::}k$, respectively. We write *nil* for $*$, and also for κ, the top-level continuation. To summarize, we arrive at the following syntax for continuations:

$$k ::= \mathit{tag}_1{::}k \mid \mathit{tag}_2{::}k \mid M{::}k \mid \mathit{nil}.$$

As this notation suggests, we will think of a continuation as an ordered list, which will be used as a *stack*. The elements of this stack are the tags tag_1 and tag_2, as well as computations M. The symbol *nil* represents the empty stack.

Table 4: The transitions of the abstract machine

CPS		Abstract Machine	
$\underline{x}k$	$\to \tilde{x}k$	$\{x, \sigma, k\}$	$\to \{M, \tau, k\}$, where $\sigma(x) = M^\tau$.
$\underline{\langle M, N\rangle}(\text{inl } k)$	$\to \underline{M}k$	$\{\langle M, N\rangle, \sigma, \mathit{tag}_1{::}k\}$	$\to \{M, \sigma, k\}$
$\underline{\langle M, N\rangle}(\text{inr } k)$	$\to \underline{N}k$	$\{\langle M, N\rangle, \sigma, \mathit{tag}_2{::}k\}$	$\to \{N, \sigma, k\}$
$\underline{\pi_1 M}k$	$\to \underline{M}(\text{inl } k)$	$\{\pi_1 M, \sigma, k\}$	$\to \{M, \sigma, \mathit{tag}_1{::}k\}$
$\underline{\pi_2 M}k$	$\to \underline{M}(\text{inr } k)$	$\{\pi_2 M, \sigma, k\}$	$\to \{M, \sigma, \mathit{tag}_2{::}k\}$
$\underline{MN}k$	$\to \underline{M}\langle \underline{N}, k\rangle$	$\{MN, \sigma, k\}$	$\to \{M, \sigma, N^\sigma{::}k\}$
$\underline{\lambda x^A.M}\langle N, k\rangle$	$\to \underline{M}[N/\tilde{x}]k$	$\{\lambda x.M, \sigma, N^\tau{::}k\}$	$\to \{M, \sigma(x \mapsto N^\tau), k\}$
$\underline{[\alpha]M}k$	$\to \underline{M}\tilde{\alpha}$	$\{[\alpha]M, \sigma, k\}$	$\to \{M, \sigma, k'\}$, where $\sigma(\alpha) = k'$.
$\underline{\mu\alpha.M}k$	$\to \underline{M}[k/\tilde{\alpha}]*$	$\{\mu\alpha.M, \sigma, k\}$	$\to \{M, \sigma(\alpha \mapsto k), \mathit{nil}\}$

After having changed the notation for continuations, we will now also change the notation for computations, i.e., for translated terms. In order to avoid having to do substitutions, we introduce the notion of a closure. A *closure* is a pair M^σ of a term M and an environment σ. An *environment* for M is a map from the free variables of M to closures, and from the free names of M to continuation, i.e., stacks. An environment σ is also sometimes called an *activation record*.

The states of Krivine's abstract machine are triples $\{M, \sigma, k\}$, consisting of a term, an environment, and a stack. Informally, a state $\{M, \sigma, k\}$ represents the term $\underline{M'}k$ of type R of the target language of the CPS transform, where M' is the term represented by the closure M^σ. The transition rules of the abstract machine can be read off directly from the corresponding transitions of the CPS semantics. Both sets of transitions are shown in Table 4.

Note how the continuation-manipulating operations of the $\lambda\mu$-calculus, namely the terms $\mu\alpha^A.M$ and $[\alpha]M$, correspond to manipulations of entire stacks, rather than individual stack elements. In particular, the $\mu\alpha$ construction requires saving an image of the entire current stack into a variable α. In actual implementations, such an operation can be implemented in several different ways. One possibility, which we will follow in Section 5, is to make an actual copy of the current stack somewhere on the heap, and to store a pointer to it in the variable α. Such a stack copy is called a *stack closure*. This implementation is conceptually simple, but potentially expensive if the stack tends to be large. Another possibility is to implement stacks as linked lists, and to use sharing instead of copying to implement the μ-operation. This reduces the cost of each μ-operation, but it can lead to an increased load for the garbage collector. See [1] for a thorough discussion of the tradeoffs of the various implementations.

The initial state for a closed program M is $\{M, \emptyset, \mathit{nil}\}$. In other words, a program starts executing in the empty environment, and with an empty stack. It is easy to see from Table 4 that the transition relation of the abstract machine is deterministic, i.e., each state has at most one successor state. On the other hand, there are clearly some states from which no transition is possible. Several such states are designated as special *halting states*, and we write:

$$\begin{array}{ll}\{*,\sigma,\mathit{nil}\} & \to \mathit{halt}\,\text{"unit"}\\ \{\langle M,N\rangle,\sigma,\mathit{nil}\} & \to \mathit{halt}\,\text{"pair"}\\ \{\lambda x.M,\sigma,\mathit{nil}\} & \to \mathit{halt}\,\text{"function"}\end{array}$$

In these cases, we say that the machine *halts* and outputs a *result*, which is one of the strings "unit", "pair", or "function". This indicates that the $\lambda\mu$-expression has been reduced to a unit term, to a pair, or to a lambda abstraction (neither of which will be evaluated further).

A state which neither allows a valid transition nor is a designated halting state is called an *error state*. An example of an error state is $\{\langle M,N\rangle,\sigma,P^{\tau}{::}k\}$. This state represents a run-time typing error, because if the current term is a pair $\langle M,N\rangle$, then the abstract machine expects either tag_1 or tag_2 on top of the stack, to indicate which of two possible branches is to be taken. It does not make sense, in this situation, to find P^{τ} on top of the stack. We imagine that the abstract machine will abort execution when it encounters an error state; a real machine might engage in undefined behavior or even crash.

Note that, as we can see from Table 4, the transitions of the abstract machine, starting from an initial state $\{M,\emptyset,\mathit{nil}\}$, correspond precisely to the top-most reduction sequence of the term $\underline{M}\kappa$ (modulo some administrative reductions).

2.5 Type soundness

A crucial property of the abstract machine is that a well-typed program does not reach an error state.

Proposition 2.1 (Type soundness). *If M is a well-typed, closed term of the $\lambda\mu$-calculus, then there is no sequence of transitions leading from state $\{M,\emptyset,\mathit{nil}\}$ to an error state.*

As a matter of fact, the simply-typed $\lambda\mu$-calculus without explicit recursion is strongly normalizing, and thus a halting state is always reached in a finite number of steps. However, once recursion is added, it is possible to obtain a non-terminating sequence of reductions.

Type soundness is best proved by giving a typed version of the abstract machine. Typed closures and typed stacks are defined by mutual recursion. A typed closure is a pair $\{\Gamma\vdash M:A\,|\,\Delta,\sigma\}$, where $\Gamma\vdash M:A\,|\,\Delta$ is a valid typing judgment and σ is an environment that maps the variables and names from Γ and Δ to typed closures, respectively typed stacks, of the appropriate types. Stacks are typed as follows:

$$\frac{k:A}{\mathit{tag}_1^{A,B}{::}k : A\wedge B}\qquad \frac{k:B}{\mathit{tag}_2^{A,B}{::}k : A\wedge B}$$

$$\frac{k:B}{\{\Gamma\vdash M:A\,|\,\Delta,\sigma\}{::}k : A\to B}\qquad \frac{}{\mathit{nil}^{\perp}:\perp}\qquad \frac{}{\mathit{nil}^{A_{top}}:A_{top}}$$

Here, A_{top} is the top-level type of the entire program. Note that not only term closures, but also the tags tag_1 and tag_2 and the empty stack nil carry type annotations. Finally, a typed abstract machine state is $\{\Gamma\vdash M:A\,|\,\Delta,\sigma,k\}$, where $\{\Gamma\vdash M:A\,|\,\Delta,\sigma\}$ is a typed closure and k is a typed stack of type A. Note that the type of k matches that of M. It is now straightforward to check the following:

1. The initial state $\{M,\emptyset,\mathit{nil}\}$ is typable, if M is a well-typed closed $\lambda\mu$-term.
2. The transitions of the abstract machine preserve well-typedness.
3. Every well-typed abstract machine state is either a halting state, or else it has a unique successor state. In particular, a well-typed state cannot be an error state.

3 Adding classical disjunction

The $\lambda\mu\nu$-calculus is an extension of the $\lambda\mu$-calculus with a type $A\vee B$ of classical disjunctions, first introduced by Pym and Ritter [9]. In call-by-name languages, the type of classical disjunctions is distinct from the more familiar intuitionistic "sum" type $A+B$, which is usually defined via left and right injections and case distinctions. In fact, the two disjunctions (intuitionistic and classical) are related by the type isomorphism $A+B\cong(\neg\neg A)\vee(\neg\neg B)$. This implies that classical disjunctions can be regarded as more primitive than sum types. As we will see, classical disjunctions can be naturally interpreted in Krivine's abstract machine as the ability to push and pop entire stack closures to and from the current stack.

3.1 The $\lambda\mu\nu$-calculus

Pym and Ritter [9] propose the following straightforward way of adding a disjunction type to the $\lambda\mu$-calculus:

$$\begin{array}{llll}\text{Types:} & A,B & ::= & \ldots \mid A\vee B\\ \text{Terms:} & M,N & ::= & \ldots \mid \langle\alpha\rangle M \mid \nu\alpha^A.M\end{array}$$

with typing rules:

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

Like μ-abstractions and named terms, 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 $\langle\alpha\rangle M$ gives this unnamed continuation the name α. Dually, the term $\nu\alpha^A.M$ abstracts a continuation of name α in M. The resulting calculus is known as the $\lambda\mu\nu$-calculus. Its equational theory is obtained from that of the $\lambda\mu$-calculus by adding the following three axioms:

$$\begin{array}{llcll}(\zeta_\vee) & [\beta]\langle\alpha\rangle\mu\gamma^{A\vee B}.M & = & M[[\beta]\langle\alpha\rangle(-)/[\gamma](-)]:\perp & \\ (\beta_\vee) & \langle\alpha'\rangle\nu\alpha^A.M & = & M[\alpha'/\alpha]:\perp & \\ (\eta_\vee) & \nu\alpha^A.\langle\alpha\rangle M & = & M:A\vee B & \text{if }\alpha\notin\mathrm{FN}(M)\end{array}$$

We also need to extend the definition of a mixed substitution $M[C(-)/[\alpha](-)]$ to replace any subterm of the form $\langle\alpha\rangle(-)$ by $\mu\beta^B.C(\mu\alpha^A.[\beta]\langle\alpha\rangle(-))$, where β is a fresh name.

3.2 Classical and intuitionistic disjunction

In the lambda calculus, one usually defines a “disjoint sum type” $A + B$ via the “inl”, “inr”, and “case” constructs. Pym and Ritter remark that in the call-by-name case, the disjunction type $A \vee B$, as defined in the previous section, does not coincide with the disjoint sum type $A + B$. To distinguish them, we sometimes refer to $A \vee B$ as “classical” disjunction and to $A + B$ as “intuitionistic” disjunction.

An interesting fact is that intuitionistic disjunction can be defined in terms of classical disjunction. Namely, we can define

$$\begin{array}{lcl} A + B & := & \neg\neg A \vee \neg\neg B \\ \text{inl } M & := & \nu\alpha.\mu\beta.[\alpha]\lambda k.kM \\ \text{inr } M & := & \nu\alpha.\mu\beta.[\beta]\lambda k.kM \\ \text{case } M \text{ of inl } x \to N \mid \text{inr } y \to P & := & \mu\gamma.(\mu\alpha.(\langle\alpha\rangle M)(\lambda y.[\gamma]P))(\lambda x.[\gamma]N) \end{array}$$

Here, $\neg A$ is an abbreviation for the function type $A \to \bot$. With these definitions, the usual equational call-by-name laws for “inl”, “inr”, and “case” are derivable from those of the $\lambda\mu\nu$-calculus. On the other hand, the classical disjunction $A \vee B$ is not definable in terms of the intuitionistic disjunction type. Thus, classical disjunction should be throught of as a very primitive operation, a low-level building block from which more high-level constructs can be built.

To further illustrate the difference between the two disjunctions, we remark that classical disjunction satisfies certain type isomorphisms such as associativity $(A \vee B) \vee C \cong A \vee (B \vee C)$ and domination $A \vee \top \cong \top$. The corresponding isomorphisms do not hold for intuitionistic disjunction. For a more in-depth discussion of type isomorphisms, see e.g. [10].

3.3 Alternative syntax

A different, more symmetric syntax for the classical disjunction type was used in [10]. Readers who are familiar with [10] may appreciate knowing that the two notations are interdefinable as follows:

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

3.4 CPS semantics and abstract machine interpretation

The CPS translation of Section 2.3 easily extends to classical disjunction: We define

$$\begin{array}{lcll} K_{A\vee B} & = & K_A \times K_B, & \\ & & & \\ \underline{\langle\alpha\rangle M} & = & \lambda k^{K_B}.\underline{M}\langle\tilde{\alpha}, k\rangle & \text{where } M : A \vee B, \\ \underline{\nu\alpha^A.M} & = & \lambda\langle\tilde{\alpha}, k\rangle^{K_{A\vee B}}.\underline{M}k & \text{where } M : B. \end{array}$$

To derive an abstract machine model from this CPS semantics, observe that the disjunction introduces a new kind of continuation of the form $\langle k', k\rangle$. In the context of abstract machines, we write this continuation as $k'{::}k$, and we interpret it as a stack whose topmost element is (a pointer to) a stack closure. The corresponding abstract machine transitions are derived directly from the CPS semantics:

CPS	Abstract Machine
$\underline{\langle\alpha\rangle M}k \to \underline{M}\langle\tilde{\alpha}, k\rangle$	$\{\langle\alpha\rangle M, \sigma, k\} \to \{M, \sigma, k'{::}k\}$, where $\sigma(\alpha) = k'$.
$\underline{\nu\alpha.M}\langle k', k\rangle \to \underline{M}[k'/\tilde{\alpha}]k$	$\{\nu\alpha.M, \sigma, k'{::}k\} \to \{M, \sigma(\alpha \mapsto k'), k\}$

Thus, we see that the connectives of classical disjunction correspond to the ability to push and pop stack closures to/from the current stack. We also introduce a new halting state, which applies in case a ν-abstraction encounters an empty stack:

$$\{\nu\alpha.M, \sigma, \mathit{nil}\} \to \textit{halt}\text{“disjunction”}.$$

An alternative way to think of the classical disjunction type is as a kind of function type, where the argument is a continuation variable instead of a term. Thus, a term of type $A \vee B$ can be thought of as a kind of function which accepts a continuation variable of type A and turns into a term of type B. Note the perfect analogy between the following pairs of reduction rules of Krivine’s abstract machine:

$$\begin{array}{lcll} \{MN, \sigma, k\} & \to & \{M, \sigma, N^\sigma{::}k\}, & \text{where } M : A \to B, \\ \{\langle\alpha\rangle M, \sigma, k\} & \to & \{M, \sigma, k'{::}k\}, & \text{where } M : A \vee B, \text{ and} \\ \{\lambda x.M, \sigma, N^\tau{::}k\} & \to & \{M, \sigma(x \mapsto N^\tau), k\}, & \text{where } \lambda x.M : A \to B, \\ \{\nu\alpha.M, \sigma, k'{::}k\} & \to & \{M, \sigma(\alpha \mapsto k'), k\}, & \text{where } \nu\alpha.M : A \vee B. \end{array}$$

This helps explain why, in call-by-name, there is a type isomorphism between $A \to (B \vee C)$ and $B \vee (A \to C)$. A term of either type can be regarded as expecting an argument of type A and a continuation of type B; the only difference is the order in which these two items are expected.

4 Adding basic types and operations

We now consider how the addition of built-in datatypes, such as integers or booleans, affects the CPS semantics and Krivine’s abstract machine. Basic types complicate the semantics somewhat, because they lead away from a “pure” call-by-name discipline. This is because primitive operations on basic types, for instance addition or multiplication, must necessarily evaluate their arguments before operating on them. Thus, even in a call-by-name language, basic operations are necessarily call-by-value.

It is therefore necessary to extend Krivine’s machine with a call-by-value evaluation mechanism at basic types. It is interesting that the rules for the abstract machine can again be derived systematically from the corresponding CPS semantics.

4.1 CPS semantics

In call-by-name languages, built-in basic types, such as integers or booleans, differ from other types, because they are equipped with a natural notion of *value*. These values are never stored in variables, but they are computed just before a built-in operation is applied. For simplicity, we assume for the moment that all built-in functions, such as addition or logical "and", are *strict*, i.e., they evaluate all their arguments before they operate on them. Thus we do not at first consider "lazy" basic operations such as lazy multiplication, which evaluates its second argument only if the first argument is non-zero. We will get back to the question of lazy functions in Section 4.3.

We consider the $\lambda\mu\nu$-calculus over a given *algebraic signature*, i.e., over a set of basic types $\sigma, \tau, \ldots$ and a set of typed constant symbols $c : \sigma$ and of typed function symbols $f : \tau_1 \to \ldots \to \tau_n \to \sigma$. As usual, n is called the *arity* of the function symbol f. For the CPS semantics, we consider the same target calculus as before. Moreover, we assume that each basic type σ of the $\lambda\mu\nu$-calculus is interpreted by a chosen type V_σ of the target calculus, together with chosen interpretations $\tilde{c} : V_\sigma$, respectively $\tilde{f} : V_{\tau_1} \to \ldots \to V_{\tau_n} \to V_\sigma$, of the primitive constants and functions. The type V_σ is called the type of *values* of type σ. We refine the CPS semantics from Sections 2.3 and 3.4 by letting $K_\sigma = V_\sigma \to R$, when σ is a basic type. Thus, continuation and computation types are defined as before:

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

Notice that a value type V_A is only defined when A is a basic type, and not when A is an arbitrary type. We extend the CPS translation of Table 3 with the following interpretation of primitive constants $c : \sigma$ and functions $f : \tau_1 \to \ldots \to \tau_n \to \sigma$:

$$\underline{c} \;=\; \lambda k.k\tilde{c}, \tag{1}$$

$$\underline{f} \;=\; \lambda\langle x_1, \ldots, x_n, k\rangle.x_1(\lambda v_1.x_2(\lambda v_2.\ldots x_n(\lambda v_n.k(\tilde{f}v_1 \ldots v_n)))). \tag{2}$$

Here $k : K_\sigma$, $x_i : C_{\tau_i}$, and $v_i : V_{\tau_i}$. Notice that the interpretation of a constant symbol c is actually a special case of the interpretation of an n-ary function symbol f, namely the case when $n = 0$. The reader should check that this CPS translation does indeed have the required behavior. In particular, 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.

4.2 Abstract machine interpretation

We extend the abstract machine interpretation to accommodate basic types and functions. As usual, we start by examining the kinds of continuations introduced by the new language feature. The CPS translation of primitive functions, shown in equation (2), introduces a new kind of continuation which is a function. We need to fit this into the "continuations as stacks" paradigm of Section 2.4. Fortunately, a careful examination of the CPS semantics reveals that, all the continuation functions which occur during the β-reduction of the CPS translation of a term are of one particular form:

$$\lambda v_j.N_{j+1}(\lambda v_{j+1}.\ldots N_n(\lambda v_n.k(\tilde{f}c_1 \ldots c_{j-1}v_jv_{j+1} \ldots v_n))), \tag{3}$$

where $1 \le j \le n$. In the abstract machine, each term N_i is represented by a closure $N_i^{\sigma_i}$, and we will represent a continuation of the form (3) by the formal expression

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

The expression $[f\,c_1 \ldots c_{j-1} \bullet N_{j+1}^{\sigma_{j+1}} \ldots N_n^{\sigma_n}]$ is called a *frame*, and it is typically implemented as a fixed-size array of data on top of the current stack (i.e., whose size depends only on the symbol f). This is analogous to the notion of a stack frame in imperative programming languages, i.e., a data structure on the stack, containing variables belonging to a particular scope or procedure. The symbol "•" is a special place holder which corresponds to a memory location which previously contained the closure $N_j^{\sigma_j}$, and where the value c_j is going to be stored next.

Before giving the transition rules of the extended abstract machine, we need to introduce one more new feature, and that is the notion of a *value state*. Recall that a state $\{M, \sigma, k\}$ of Krivine's abstract machine corresponds to a term of the form $\underline{M}k$ under CPS. Because of the presence of values in the CPS semantics, there is now a new kind of state which is of the form kc, where k is a continuation of a basic type A, and c is a value of type A, i.e., an element of V_A. We call a state of the form kc a *value state*, and we denote it in the abstract machine as a pair $\{c, k\}^{\mathrm{v}}$. Note that, unlike an ordinary state of the form $\{M, \sigma, k\}$, a value state $\{c, k\}^{\mathrm{v}}$ does not require an environment.

The CPS semantics of primitive constants and functions, as embodied in equations (1) and (2), has the following transitions, where $j < n$ and $d = \tilde{f}c_1 \ldots c_{n-1}c$:

$$\begin{array}{ll}
\underline{c}k & \to k\tilde{c} \\
\underline{f}\langle N_1, ..., N_n, k\rangle & \to N_1(\lambda v_1....N_n(\lambda v_n.k(\tilde{f}v_1...v_n))) \\
(\lambda v_j.N_{j+1}(...(\lambda v_n.k(\tilde{f}c_1...c_{j-1}v_j...v_n))))c & \to N_{j+1}(...(\lambda v_n.k(\tilde{f}c_1...c_{j-1}cv_{j+1}...v_n))) \\
(\lambda v_n.k(\tilde{f}c_1...c_{n-1}v_n))c & \to kd,
\end{array}$$

These can now be immediately translated to transition rules of the abstract machine:

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

In these rules, it is again assumed that f is an n-ary function symbol, that $j < n$, and that $d = \tilde{f}c_1 \ldots c_{n-1}c$. We also introcuce two new halting states: a value state with empty stack is a halting state with result c, and an n-ary built-in function f will halt with value "function" if the stack contains fewer than the required n arguments.

$$\begin{array}{l}
\{c, \mathit{nil}\}^{\mathrm{v}} \to \mathit{halt}\, c, \\
\{f, \sigma, k\} \to \mathit{halt}\,\text{“function”}, \quad \text{if size}(k) < n.
\end{array}$$

4.3 "Impure" functions

So far, we have only considered primitive functions of the form $f : \tau_1 \to \ldots \to \tau_n \to \sigma$, where all of $\tau_1, \ldots, \tau_n$ and σ are basic types. Sometimes, it is useful to allow primitive functions with arbitrary result type, i.e., of the form $f : \tau_1 \to \ldots \to \tau_n \to A$, where A is any type. We refer to these more general basic functions as "impure".

One example of an impure basic function is the if-then-else function $\mathit{if}_B : \mathit{bool} \to (B \to B \to B)$ which maps *true* to $\lambda xy.x$ and *false* to $\lambda xy.y$. Here, *bool* is a built-in type of booleans, and B is any type. Another example is the "lazy multiplication" function *lazymult* : *int* $\to$ (*int* $\to$ *int*), where *int* is the type of integers. By definition, the function *lazymult* maps 0 to the constant function $\lambda x.0$, and any other integer n to $\lambda x.$*mult n x*, where *mult* is the usual strict multipliction operation. We can regard both *if* and *lazymult* as impure, strict basic functions in one argument.

Another useful example of an impure function is the side-effecting *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 $\lambda x.x$. As an application of this idea, we can consider a family of basic functions $\mathit{print}_B : \mathit{int} \to (B \to B)$. The intended meaning is that $(\mathit{print}\, n); M$ prints the integer n and then behaves like M.

4.4 Semantics of impure functions

The CPS semantics of impure basic functions is straightforward. For each impure basic function symbol $f : \tau_1 \to \ldots \to \tau_n \to A$, we need a chosen term $\tilde{f} : V_{\tau_1} \to \ldots \to V_{\tau_n} \to C_A$ of the target language of the CPS translation. The term f is then translated as follows:

$$\underline{f} \quad = \quad \lambda\langle x_1, \ldots, x_n, k\rangle . x_1(\lambda v_1 . x_2(\lambda v_2 . \ldots x_n(\lambda v_n . (\tilde{f} v_1 \ldots v_n)k))). \qquad (4)$$

Here, $k : K_A$, $x_i : C_{\tau_i}$, and $v_i : V_{\tau_i}$. Note that the only difference between equations (2) and (4) is the order of the terms $\tilde{f} v_1 \ldots v_n$ and k. For impure functions, the term $\tilde{f} v_1 \ldots v_n$ is of type C_A, whereas for pure functions, it is of type V_A. It follows that the interpretation of an impure function does not coincide with that of a pure function, even in the case where A happens to be a basic type: the interpretation of a pure function always produces a value, whereas the interpretation of an impure function potentially produces an arbitrary computation.

In concrete cases, we rely on the target language of the CPS transform to supply us with "native" implementations of the required functionality. To interpret the basic function *if* : *bool* $\to (B \to B \to B)$, we assume that V_{bool} is the type of booleans of the target language, and we define the function $\widetilde{\mathit{if}} : V_{\mathit{bool}} \to C_{B \to B \to B}$ such that $\widetilde{\mathit{if}}\,\mathit{true} = \underline{\lambda xy.x}$ and $\widetilde{\mathit{if}}\,\mathit{false} = \underline{\lambda xy.y}$. The interpretation of *lazymult* is similar.

The easiest way to interpret the side-effecting *print* function (although there are better ways) is to assume that the target language of the CPS transform also allows side effects. In this case, we need a primitive function $\widetilde{\mathit{print}} : V_{\mathit{int}} \to C_{B \to B}$ of the target language, such that $\widetilde{\mathit{print}}\, n$ has the behavior of printing n and then returning $\underline{\lambda x.x}$.

For the abstract machine interpretation, we will overload the frame notation by writing $[f\, c_1 \ldots c_{j-1} \bullet N_{j+1}^{\sigma_{j+1}} \ldots N_n^{\sigma_n}]{::}k$ for the expression

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

in the case where f is an impure basic function. Note that this is not quite the same as equation (3). From the CPS semantics of the impure basic functions *if* and *print*, we

have

$$\begin{array}{lcl}
(\lambda v.(\widetilde{\mathit{if}}\, v)k)\mathit{true} & \to & \underline{\lambda xy.x}k \\
(\lambda v.(\widetilde{\mathit{if}}\, v)k)\mathit{false} & \to & \underline{\lambda xy.y}k \\
(\lambda v.(\widetilde{\mathit{print}}\, v)k)c & \xrightarrow{\text{output } c} & \underline{\lambda x.x}k.
\end{array}$$

Here the label "output c" denotes a side effect taking place as part of the reduction. This immediately gives rise to the corresponding abstract machine rules:

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

5 Implementing the abstract machine

In this section, we give an implementation of Krivine's abstract machine, and its various extensions, in an idealized, low-level assembly language. This illustrates that, despite its name, the abstract machine is not as "abstract" as one might think; it can be implemented, with relatively little effort, on a standard von Neumann style "concrete" machine. Note that the implementation takes the form of a *compiler*, and not of an *interpreter*; thus, the final program does not run by updating a data structure, but by executing actual code.

As already pointed out in the introduction, the implementation given here is not efficient enough to be useful in practice. Its main flaw is that it uses a naive call-by-name evaluation strategy, in which each subterm is possibly evaluated many times. This is the same evaluation strategy which is embodied in Krivine's abstract machine, and since our goal is to follow the abstract machine model as faithfully as possible, we resist the temptation to optimize. It can be argued that any substantial improvement to the implementation is best carried out at the abstract machine level, or even at the level of CPS translations, rather than at the compiler level.

We also take the liberty to ignore certain practical aspects of implementations, such as garbage collection and efficient register allocation. In our "ideal" implementation, we simply assume that there are infinitely many registers and an infinite amount of memory available.

5.1 Target assembly language

The target language of our compiler is an idealized assembly language whose instruction set shown in Table 5. It differs from actual assembly languages in several respects. First, we assume that there are infinitely many registers. Second, we assume that there are built-in instructions for certain high-level operations such as memory allocation (ALLOC) and the manipulation of stack closures (SAVE, RESTORE); these would not normally be available as separate instructions, but would be implemented as macros or system calls.

The only data type of the assembly language is a *word*, which can be interpreted as an integer, a boolean (with $0 = \mathit{false}$, $1 = \mathit{true}$), or as a pointer. We assume that

Instruction	Meaning
MOVE w, v	Store the value v in location w
ADD w, v	Add v to w
PUSH v	Push the value v onto the stack
POP w	Pop a value from the stack and store it in w
CMP v_1, v_2	Compare the two values, and remember the result
BNE v	If previous CMP resulted in not equal, jump to location v
BGE v	If previous CMP resulted in greater than or equal jump to location v
JUMP v	Jump to address v
CALL v	Call subroutine at address v
ALLOC w, v	Allocate v words of memory and store a pointer to them in w
SAVE w	Make a stack closure from the current stack, and store a pointer to it in w
RESTORE v	Replace the current stack by a copy of the stack closure pointed to by v
EXIT v	Exit with result v

Table 5: Instruction set of the idealized assembly language

there are infinitely many *registers* $R_1, R_2, \ldots$, as well as four distinguished registers SP, SS, C, and V, each of which can hold a word.

We assume that there are infinitely many addressable memory cells, each of which holds a word. A *memory reference* takes the form $[R, n]$, where R is a register and n is a literal integer. The expression $[R, n]$ refers to the contents of the memory cell at address $R+n$. An ℓ*-value* (assignable value) is either a register or a memory reference. A *value* is either an ℓ-value or a literal integer. Literal integers are often written as $\#n$ in assembly language instructions.

The memory is divided into two separate regions: the *stack* and the *heap*. The stack is manipulated in the usual way via the PUSH and POP instructions, and also via the two special registers SP and SS, which represent the *stack pointer* and the *stack size*, respectively. We assume that the stack grows downward (towards lower memory addresses), and that the stack pointer SP points to the memory cell just below the stack, so that $[SP, 1]$ refers to the topmost element on the stack. Setting the register SS to 0 has the effect of emptying the stack.

The instruction set of the assembly language is shown in Table 5. Here, the letter w ranges over ℓ-values and the letter v ranges over values. The meaning of most instructions should be clear. Note that there is only one MOVE instruction, which can be used, among other things, to copy a value from memory to a register or vice versa. The PUSH and POP instructions implicitly update the registers SP and SS. Some high-level operations are included for convenience: ALLOC is used to allocate memory from the heap. SAVE and RESTORE are used to manipulate stack closures and will be explained in more detail later, and the EXIT instruction ends the computation and returns a result which is a word; it is up to the environment to interpret this word correctly as an integer, a boolean, or a pointer to a literal string, depending on the type of the program being run.

When writing assembly language code, each instruction can be preceded by an optional label, which provides a symbolic reference to the address of the instruction. We use a semicolon ";" to introduce a comment.

5.2 Data representation

We need to specify how the various kinds of data of Krivine's abstract machine are represented in memory. Specifically, we need to fix a representation for term closures, stack closures, and for items on the stack. Terms themselves are represented as code, and will be discussed in Section 5.3.

Term closures and stack closures are allocated on the heap. A term closure N^σ is represented as $n+1$ consecutive words $a_0, \ldots, a_n$ in memory. Here a_0 is a pointer to the code for the term N, and $a_1, \ldots, a_n$ are pointers to representations of the closures $\sigma(x_1), \ldots, \sigma(x_n)$, where $x_1, \ldots, x_n$ are the free variables and names of N.

A stack closure is represented by a record of $n+1$ words, of which the first one holds the number n, and the remaining ones hold the actual stack data. For convenience, we provide a SAVE w instruction, which makes a heap-allocated closure from the current stack and returns a pointer to it in w. We also provide a RESTORE v instruction, which erases the current stack and replaces it by a copy of the stack closure pointed to by v.

The stack of the abstract machine is of course implemented as the native stack of the assembly language. Most individual items on the stack are represented as single words, except for frames, which are represented as records of several words. The tags *tag*$_1$ and *tag*$_2$ are represented as the integers 1 and 2, respectively. Term closures and stack closures are represented as pointers to the respective objects on the heap. The representation of *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 $n+1$ words $f_j, c_1, \ldots, c_{j-1}, b, p_{j+1}, \ldots, p_n$. Here, f_j is a special tag which uniquely determines f and j (actually, we will implement f_j as a pointer to code). $c_1, \ldots, c_{j-1}$ are literal values, b is an undefined word (occupying the position of the "$\bullet$" in the frame), and $p_{j+1}, \ldots, p_n$ are pointers to representations of the closures $N_{j+1}^{\sigma_{j+1}}, \ldots, N_n^{\sigma_n}$.

5.3 Compilation of terms

Terms are not represented as data structures, but rather as code to be executed. Since a term needs to be able to access the values of its free variables, it is executed in the context of a particular closure, the *current closure* of the term. By convention, we assume that there is a special register C which always contains a pointer to the current closure. Thus, the calling convention for invoking a specific closure is to store a pointer to it in the register C, then jump to the address $[C, 0]$.

When the abstract machine is in a value state, the current value needs to be stored somewhere; by convention, we store it in the special register V. (As a matter of fact, the registers C and V are never used simultaneously, so it would be possible to use just one register for both purposes. However, doing so would add no conceptual clarity). The representation of stack frames was arranged in such a way that, when the machine reaches a value state, the topmost item on the stack is a tag f_j. We interpret this as an

Lambda calculus

$\llbracket x \rrbracket_s =$

```
      MOVE    $C, s(x)$
      JUMP    $[C, 0]$
```

$\llbracket \lambda x.M \rrbracket_s =$

```
      CMP     $SS, \#0$
      BNE     $l$
      EXIT    “function”
$l$ : POP     $R$
      $\llbracket M \rrbracket_{s(x \mapsto R)}$
```

(where l is a fresh label and R is a fresh register.)

$\llbracket MN \rrbracket_s =$

```
      ; build closure for $N$
      ALLOC   $R, \#(n+1)$
      MOVE    $[R, 0], \#l$
      MOVE    $[R, 1], s(x_1)$
      ...
      MOVE    $[R, n], s(x_n)$
      PUSH    $R$
      $\llbracket M \rrbracket_s$
$l$ : $\llbracket N \rrbracket_{(x_1 \mapsto [C,1], \ldots, x_n \mapsto [C,n])}$
```

(where l is a fresh label, R is a fresh register, and $\mathrm{FV}(N) \cup \mathrm{FN}(N) = \{x_1, \ldots, x_n\}$.)

$\llbracket \langle M, N \rangle \rrbracket_s =$

```
        CMP     $SS, \#0$
        BNE     $l_1$
        EXIT    “pair”
$l_1$ : POP     $R$
        CMP     $R, \#1$
        BNE     $l_2$
        $\llbracket M \rrbracket_s$
$l_2$ : $\llbracket N \rrbracket_s$
```

(where l_1, l_2 are fresh labels, and R is a fresh register.)

$\llbracket \pi_i M \rrbracket_s =$

```
      PUSH    $\#i$
      $\llbracket M \rrbracket_s$
```

$\llbracket * \rrbracket_s =$

```
      EXIT    “unit”
```

$\lambda\mu$-Calculus

$\llbracket \mu\alpha.M \rrbracket_s =$

```
      ; build a stack closure
      SAVE    $R$
      ; clear the stack
      MOVE    $SP, \#0$
      $\llbracket M \rrbracket_{s(\alpha \mapsto R)}$
```

(where R is a fresh register.)

$\llbracket [\alpha]M \rrbracket_s =$

```
      RESTORE $s(\alpha)$
      $\llbracket M \rrbracket_s$
```

Classical disjunction ($\lambda\mu\nu$-calculus)

$\llbracket \nu\alpha.M \rrbracket_s =$

```
      CMP     $SS, \#0$
      BNE     $l$
      EXIT    “disjunction”
$l$ : POP     $R$
      $\llbracket M \rrbracket_{s(\alpha \mapsto R)}$
```

(where l is a fresh label and R is a fresh register.)

$\llbracket \langle \alpha \rangle M \rrbracket_s =$

```
      PUSH    $s(\alpha)$
      $\llbracket M \rrbracket_s$
```

Table 6: The compilation of terms

Basic constants

$\llbracket n \rrbracket_s =$

```
      MOVE    $V, \#n$
      CMP     $SS, \#0$
      BNE     $l$
      EXIT    $V$
$l$ : POP     $R$
      JUMP    $R$
```

(where l is a fresh label and R is a fresh register.)

$\llbracket \mathit{true} \rrbracket_s = \llbracket 1 \rrbracket_s$

$\llbracket \mathit{false} \rrbracket_s = \llbracket 0 \rrbracket_s$

“Pure” basic functions

$\llbracket f \rrbracket_s =$

```
            ; check for sufficient arguments
            CMP     $SS, \#n$
            BGE     $f_0$
            EXIT    “function”
$f_0$ :     MOVE    $C, [SP, 1]$
            PUSH    $\#f_1$
            JUMP    $[C, 0]$

(repeat following code for $j = 1 \ldots n-1$)
$f_j$ :     MOVE    $[SP, j], V$
            MOVE    $C, [SP, j+1]$
            PUSH    $\#f_{j+1}$
            JUMP    $[C, 0]$

$f_n$ :     MOVE    $[SP, n], V$
            ; $n$ values are now on top of stack
            CALL    native$_f$
            POP     $V$
            CMP     $SS, \#0$
            BNE     $l$
            EXIT    $V$
$f_{n+1}$ : POP     $R$
            JUMP    $R$
```

(where $n \geq 1$ is the arity of f, $f_0, \ldots, f_{n+1}$ are fresh labels, and R is a fresh register.)

Some “impure” basic functions

$\llbracket \mathit{if} \rrbracket_s =$

```
            CMP     $SS, \#1$
            BGE     $\mathit{if}_0$
            EXIT    “function”
$\mathit{if}_0$ :  POP     $C$
            PUSH    $\#\mathit{if}_1$
            JUMP    $[C, 0]$
$\mathit{if}_1$ :  CMP     $V, \#0$
            BNE     $\mathit{if}_2$
            $\llbracket \lambda xy.y \rrbracket_\emptyset$
$\mathit{if}_2$ :  $\llbracket \lambda xy.x \rrbracket_\emptyset$
```

(where $\mathit{if}_0, \ldots, \mathit{if}_{n+1}$ are fresh labels.)

$\llbracket \mathit{print} \rrbracket_s =$

```
              CMP     $SS, \#1$
              BGE     $\mathit{print}_0$
              EXIT    “function”
$\mathit{print}_0$ : POP     $C$
              PUSH    $\#\mathit{print}_1$
              JUMP    $[C, 0]$
$\mathit{print}_1$ : PUSH    $V$
              CALL    native$_{\mathit{print}}$
              $\llbracket \lambda x.x \rrbracket_\emptyset$
```

(where print_0, print_1 are fresh labels.)

Table 7: The compilation of terms, continued

address to jump to. Thus, the convention in a value state is to put the value into the register V, then pop the topmost address from the stack and jump to it. If a value state encounters an empty stack, then the program halts and the current value is the result of the computation.

A compiled term must know where to find the values of its free variables, either as offsets within the current closure, or in registers or elsewhere in memory. Therefore, the translation of a term M is defined relative to a *symbol table* s, which is a function from the free variables of M to symbolic values. For example, the symbol table might specify that the value of the free variables x, y, and z can be found in $[C, 1]$, $[C, 2]$, and in the register R_4, respectively. Note that a symbol table s is a compile-time concept and maps variables to *symbolic* values, unlike an environment σ, which is a run-time concept and maps variables to *actual* values. We write $s(x \mapsto v)$ for the symbol table obtained from s by adding a mapping of the variable x to the symbolic value v.

We use the notation $[\![M]\!]_s$ to denote the assembly code for the term M under the symbol table s. The rules of translation are derived directly from the corresponding rules of the abstract machine, and they are shown in Tables 6 and 7. Note that the translation proceeds by recursion on the structure of terms. Also note that the translation $[\![M]\!]_s$ of a term is always a piece of assembly code which ultimately ends in a JUMP or EXIT instruction.

5.4 The translation of individual terms

The code for a variable x simply invokes the closure that x points to. According to our calling convention for closures, this is done by loading a pointer to the closure into the register C, then jumping to the address $[C, 0]$.

The code for a lambda abstraction $\lambda x.M$ simply pops a value from the stack and binds it to the variable x; thereafter, it behaves like M. Three additional lines of code are needed to test whether the stack is empty, in which case the program halts.

The code for an application MN builds a term closure for N; this is done by allocating $n + 1$ words of memory, and storing in them the address of the code for N, as well as the values of the free variables $x_1, \ldots, x_n$ of N. A pointer to the term closure is then pushed onto the stack before M is executed. Note that the code for the term N is given separately, and is generated relative to a new symbol table where the variables $x_1, \ldots, x_n$ are mapped to the respective offsets into the "current" closure (i.e., the closure which will be current when N is invoked).

The translations of pairs, projections, and unit are straightforward and follow directly from the corresponding rules of the abstract machine. The code for a pair pops a tag from the stack, whereas the code for a projection pushes a tag onto the stack.

The code for a μ-abstraction $\mu\alpha.M$ saves the current stack into a new stack closure, and then executes M in the context of an empty stack, and with the name α bound to the stack closure just created.

The code for $[\alpha]M$ replaces the current stack by the stack closure pointed to by α.

The code for a ν-abstraction pops a pointer to a stack closure from the stack and binds it to the variable α. Note that this code is almost identical to that of a λ-abstraction, except that, in case of an empty stack, the result of the program is "disjunction" instead of "function".

The code for $\langle\alpha\rangle M$ pushes a pointer to a stack closure onto the stack.

The code for basic integer and boolean constants, which is shown in Table 7, reflects our convention for value states. Namely, the convention specifies to put the value into the special register V, then jump to the address on top of the stack, if any. If the stack is empty, V is returned as the result of the program.

The code for a "pure" basic function is interesting. We first check whether there are enough closures on the stack to form a frame. Note that the data representation of a frame was chosen in such a way that the rule

$$\{f, \sigma, N_1^{\sigma_1} :: \ldots :: N_n^{\sigma_n} :: k\} \to \{N_1, \sigma_1, [f \bullet N_2^{\sigma_2} \ldots N_n^{\sigma_n}] :: k\}$$

does not require any rearrangment of the n closures; the first frame is simply built by pushing the address f_1 onto the stack. The first closure $N_1^{\sigma_1}$, a pointer to which is stored in the stack frame, is then invoked. Eventually, this closure reaches a value state, and following the convention for value states, it will jump to the address on top of the stack, in this case f_1, with V being the value just computed. This value is stored in the current frame, and then the remaining closures $N_2^{\sigma_2}, \ldots, N_n^{\sigma_n}$ are evaluated in the same way, until the top n items on the stack contain the actual arguments to the function f. At this point, we call a subroutine which contains some native implementation of the function f. The convention is that this native implementation expects its n arguments on top of the stack, and returns its result on top of the stack as well. After the subroutine call returns, we simply pop the result value off the stack and follow the protocol for value states.

Finally, Table 7 shows the implementation of two "impure" basic functions, the "*if*" and "*print*" functions which were already discussed in Section 4.3. Both these functions use a simplified form of the mechanism for pure basic functions (specialized for unary functions) to evaluate the closure on top of the stack and to obtain a value V. The "*if*" function then simply executes $\lambda xy.x$ or $\lambda xy.y$, depending whether $V = \mathit{true}$ or $V = \mathit{false}$. The "*print*" function calls a subroutine to print the value V, then executes $\lambda x.x$.

References

[1] W. D. Clinger, A. H. Hartheimer, and E. M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12:7–45, 1999.

[2] G. Cousineau. The categorical abstract machine. In G. Huet, editor, *Logical Foundations of Functional Programming*, pages 25–45. Addison-Wesley, 1990.

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

[4] T. G. Griffin. A formulae-as-types notion of control. In *POPL '90: Proceedings of the 17th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1990.

[5] 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.

[6] J.-L. Krivine. Un interpreteur du lambda-calcul. Draft, available from ftp://ftp.logique.jussieu.fr/pub/distrib/krivine/interprt.pdf.

[7] 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.

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

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

[10] P. Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Math. Struct. in Computer Science*, 11(2), 2001.

[11] T. Streicher and B. Reus. Classical logic, continuation semantics and abstract machines. *Journal of Functional Programming*, 8(6):543–572, Nov. 1998.

[12] D. H. D. Warren. An abstract prolog instruction set. Technical Note 309, SRI International, 1983.