The Next 700 Krivine Machines

Rémi Douence* and Pascal Fradet**

douence@emn.fr Pascal.Fradet@inria.fr

Abstract: The Krivine machine is a simple and natural implementation of the normal weak-head reduction strategy for pure λ-terms. While its original description has remained unpublished, this machine has served as a basis for many variants, extensions and theoretical studies. In this paper, we present the Krivine machine and some well-known variants in a common framework. Our framework consists of a hierarchy of intermediate languages that are subsets of the λ-calculus. The whole implementation process (compiler + abstract machine) is described via a sequence of transformations all of which express an implementation choice. We characterize the essence of the Krivine machine and locate it in the design space of functional language implementations. We show that, even within the particular class of Krivine machines, hundreds of variants can be designed.

Key-words: Krivine machine, abstract machines, program transformation, compilation, functional language implementations.

1 Introduction

The Krivine machine (or K-machine) is a simple and natural implementation of the weak-head call-by-name reduction strategy for pure λ-terms. It can be described in just three or four rules with minimal machinery (an environment and a stack). While its original description has remained unpublished, the K-machine has served as a basis for many variants, extensions and theoretical studies. For instance, Crégut used the K-machine as a basis for the implementation of other reduction strategies (call-by-need, head and strong reduction [4]); Leroy presents his Zinc abstract machine as a strict variant of the K-machine [24]; many others used it as a basis or framework for their work either practical or theoretical [8][16][23][31][32][34].

The presentations of the K-machine or its variants differ depending on the sources. The machine instructions may be de Bruijn's λ-expressions or completely compiled code. Environments and closures also have different representations. In this paper, we present the K-machine and some well-known variants in a common framework. This framework has been used to describe, prove, compare, classify and hybridize functional language implementations [9][10][11][12]. Here our main goal is:

* ÉCOLE DES MINES DE NANTES, 4 rue Alfred Kastler, BP 20722, 44307 Nantes Cedex 3, France.

** Work performed while at INRIA/IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France.
Author's current address: INRIA RHÔNE-ALPES, 655 av. de l'Europe, 38330 Montbonnot, France.

- To characterize the essence of the K-machine by making its fundamental implementation choices clear. This defines a class of abstract machines and suggests new variants.
- To locate the K-machine in the design space of functional language implementations. In particular, we mention alternative implementation choices and compare the K-machine with other abstract machines (e.g. Tim [13], Cam [3]).

2 Overview

The most common description of the K-machine [5] is given in Figure 1. It takes the form of an operational semantics whose state transition rules involve de Bruijn's λ-expressions*. A machine state (C, S, E) is made of a code C, an environment E and a stack S. Here, and throughout the paper, we represent stacks (and lists) using pairs. For instance, the expression $(\ldots(((S,X_n),X_{n-1}),X_{n-2}),\ldots,X_1)$ denotes a stack of (at least) n elements, with X_1 at the top. A closure is represented by a (code, environment) pair.

$(M\ N,\ S,\ E)$	$\rightarrow$	$(M,\ (S,(N,E)),\ E)$
$(\lambda M,\ (S,N),\ E)$	$\rightarrow$	$(M,\ S,\ (E,N))$
$(i+1,\ S,\ (E,N))$	$\rightarrow$	$(i,\ S,\ E)$
$(0,\ S,\ (E_1,(M,E_2)))$	$\rightarrow$	$(M,\ S,\ E_2)$

Figure 1 Usual description of the standard K-Machine

To evaluate an application $M\ N$, the K-machine builds a closure made of the argument N and the current environment E in the stack and proceeds with the reduction of the function M. This is the first characteristic of the K-machine: a closure is built in constant time and includes the complete current environment.

The evaluation of a λ-abstraction places the argument (the stack's top element) in the environment and proceeds with the body of the function. This is the second and more important characteristic of the K-machine: it strives not to build closures for functions. Other abstract machines return functions as closures before applying them.

The evaluation of a variable i amounts to following i links to find the corresponding closure in the environment. The closure's components become the current code and environment.

* In de Bruijn's notation [7], a variable occurrence is represented by the number of lambdas between this occurrence and the lambda binding the variable. For example, the λ-term $\lambda x.x\ (\lambda y.y\ x)$ is written in de Bruijn's notation as $\lambda 0(\lambda 0\ 1)$.

Other presentations use a more compiled code for their machine. Indeed, de Bruijn's notation compiles environment management, but applications remain interpreted by the machine. The representations of environments or closures also differ according to the machines.

In this paper, we model the K-machine and its variants as compositions of program transformations. Our framework consists of a hierarchy of intermediate languages all of which are subsets of the λ-calculus. The description of an implementation, such as the K-machine, consists of a series of transformations, each one compiling a particular task by mapping an expression from one intermediate language into another. The functional expressions obtained can be seen as machine code and their reduction can be seen as the state transitions of an abstract machine. This approach has several benefits:

- It is modular. Each transformation implements one compilation step and can be defined independently of the other steps. Transformations implementing different steps are composed to specify implementations. For instance, in this article, we model the canonical Krivine abstract machine as the composition of two program transformations. Implementation choices are modeled as distinct program transformations. Combinations of these transformations lead to many variants of the K-machine. Many classical abstract machines and new hybrid machines (e.g. that mix different implementation techniques for β-reduction) have been described in this framework [12].
- It has a strong formal basis. The functional framework is the only formal framework used. Instead of translations and state transitions, we use program transformation and functional reduction. This simplifies the correctness proofs and comparisons. Different implementation choices are represented by different transformations which can be related and compared. For instance, we have shown that, regarding the implementation of the reduction strategy, a G-machine is an interpretative version of the K-machine. The compilation of the reduction strategy used by the strict K-machine has also been formally compared to the compilation of the reduction strategy used by the SECD machine [12].
- It is (relatively) abstract although the intermediate languages come closer to an assembly language as we progress in the description. The combinators of the intermediate languages allow a more abstract description of notions such as instructions, sequencing or stacks. As a consequence, the compilation of control is expressed more abstractly than using CPS expressions, and the representation of components (e.g., data stack, environment stack) is a separate implementation step.
- It is extendable. New intermediate languages and transformations can be defined and inserted into the transformation sequence to model new compilation steps. For instance, some variants of the K-machine would be more faithfully described using additional transformations to compile control transfers (calls and returns) and/or register allocation.

The reduction of λ-terms comprises two main tasks: searching for the next redex (which depends on the reduction strategy) and β-reducing redexes. We describe an implementation as a transformation sequence $\Lambda \xrightarrow{T_1} \Lambda_s \xrightarrow{T_2} \Lambda_e$, where the transformations T_1 and T_2 compile the reduction strategy and the β-reduction respectively. The functional expressions obtained (in Λ_e) are sequences of combinators whose reduction can be seen as state transitions. In order to provide some intuition, consider the following definition of the 3-argument combinator **mkclos**:

$$\textbf{mkclos}\ C\ (S,N)\ E = C\ (S,(N,E))\ E$$

Its reduction can be seen as a state transition where (**mkclos** *C*), (*S*,*N*) and *E* represent the code, the current stack and environment respectively. The combinator **mkclos** can therefore be seen as a functional instruction building a closure made of the top stack element and the current environment and pushing this closure onto the stack.

In this paper, we consider only pure λ-expressions and the two aforementioned compilation steps. This is sufficient to present the fundamental choices that define the K-machine. There are other steps like the compilation of control transfers (calls and returns) and the implementation of sharing and updates in lazy implementations. Constants, primitive operators, recursion and data structures can also be taken into account by the K-machine. We will briefly mention the possible implementation choices for these extensions.

We focus on the description of the Krivine machine and some of its variants. We will introduce only the notions needed by this aim and will not provide correctness proofs. The interested reader can find a more complete presentation of the framework, descriptions and the classification of a dozen standard implementations, formal comparisons, correctness proofs and new implementation techniques in two technical reports [9][10], a journal article [12] and a PhD thesis [11].

We start in Section 3 by describing the classic, call-by-name, K-machine in our framework. Section 4 presents several variants of the K-machine for alternative reduction (call-by-value, call-by-need) and environment management strategies. In conclusion (Section 5), we review related work, the main characteristics of K-machines and the possible variations within this particular class.

3 Standard K-Machine

In this section, we describe the original and simplest version of the K-machine: it considers the call-by-name evaluation strategy and uses lists to represent the environments. We focus on pure λ-expressions and our source language Λ is

$$M ::= x \mid M_1\ M_2 \mid \lambda x.M$$

Extensions of the λ-calculus (e.g. constants, primitive operators, data structures) are discussed in the conclusion.

3.1 Framework

Our framework consists of a hierarchy of intermediate languages, all of which are subsets of the λ-calculus. We describe the implementation process via the transformation sequence $\Lambda \xrightarrow{T1} \Lambda_s \xrightarrow{T2} \Lambda_e$ starting with Λ and involving two intermediate languages.

The first intermediate language Λ_s is a subset of Λ defined using the three combinators **;**, $\textbf{push}_s$ and $\textbf{pop}_s$.

Λ_s $\quad M ::= x \mid M_1 \,;\, M_2 \mid \mathbf{push}_s\, M \mid \mathbf{pop}_s(\lambda x.M)$

Intuitively, **;** is a sequencing operator and M_1 **;** M_2 can be read "evaluate M_1 then evaluate M_2", $\mathbf{push}_s\, M$ returns M as a result and $\mathbf{pop}_s(\lambda x.M)$ binds the previous intermediate result to x before evaluating M. The combinators $\mathbf{push}_s$ and $\mathbf{pop}_s$ suggest a stack storing intermediate results. This argument stack will be denoted by s.

The language Λ_s is a subset of Λ since it rules out unrestricted applications and λ-abstractions always occur within a $\mathbf{pop}_s$. The three combinators **;**, $\mathbf{push}_s$ and $\mathbf{pop}_s$ are not linguistic extensions but only specific closed λ-expressions (to be defined later).

The substitution and the notion of free or bound variables are the same as in the λ-calculus. The basic combinators can be given different definitions (possible definitions are given in Section 3.5). We do not pick specific ones at this point; we simply impose the associativity of sequencing and that the combinators satisfy rules corresponding to the β and η-conversions (Figure 2, where "=" stands for λ-convertibility). To simplify the notation, we write $\lambda_s x.M$ for the expression $\mathbf{pop}_s(\lambda x.M)$.

(assoc)	$(M_1 \,;\, M_2) \,;\, M_3 = M_1 \,;\, (M_2 \,;\, M_3)$
(β_s)	$(\mathbf{push}_s\, N) \,;\, (\lambda_s x.M) = M[N/x]$
(η_s)	$\lambda_s x.(\mathbf{push}_s\, x \,;\, M) = M$ *if x does not occur free in M*

Figure 2 Conversion rules in Λ_s

We consider only one reduction rule corresponding to the classical β-reduction:

$$(\mathbf{push}_s\, N) \,;\, (\lambda_s x.M) \;\Rightarrow\!\!\!\!\Rightarrow\; M[N/x]$$

For example:

$$\mathbf{push}_s(\mathbf{push}_s(\lambda_s z.z) \,;\, \lambda_s y.y) \,;\, \lambda_s x.x \;\Rightarrow\!\!\!\!\Rightarrow\; \mathbf{push}_s(\lambda_s z.z) \,;\, \lambda_s y.y \;\Rightarrow\!\!\!\!\Rightarrow\; \lambda_s z.z$$

As with all standard abstract machines, we are only interested in modeling weak reductions. In our framework, a weak redex is a redex that does not occur inside an expression of the form $\mathbf{push}_s\, M$ or $\lambda_s x.M$. Weak reduction does not reduce under $\mathbf{push}_s$'s or λ_s's and, from here on, we write "redex" (resp. reduction, normal form) for weak redex (resp. weak reduction, weak normal form). Note that a redex cannot occur as a subexpression of another redex. So, a reduction $(\mathbf{push}_s\, N) \,;\, (\lambda_s x.M) \;\Rightarrow\!\!\!\!\Rightarrow\; M[N/x]$ cannot suppress nor duplicate another redex. The β_s-reduction is therefore strongly confluent and hence confluent. In Λ_s, the choice of the next redex is not relevant anymore: all redexes are needed and any reduction strategy is normalizing (i.e. reaches the normal form where there exists one). This is the key point to view transformations from Λ to Λ_s as compiling the evaluation strategy.

The next intermediate language Λ_e allows the encoding of environment management by introducing the combinators $\mathbf{push}_e$ and $\mathbf{pop}_e$.

Λ_e $\qquad M ::= x \mid M_1 \,;\, M_2 \mid \mathbf{push}_s\, M \mid \mathbf{pop}_s(\lambda x.M) \mid \mathbf{push}_e\, M \mid \mathbf{pop}_e(\lambda x.M)$

The combinators $\mathbf{push}_e$ and $\mathbf{pop}_e$ behave exactly as $\mathbf{push}_s$ and $\mathbf{pop}_s$ but they act on a (at least conceptually) different component *e* (e.g. a stack of environments). They obey the rules (β_e) and (η_e) similar to the ones in Figure 2. In Λ_e, variables will be only used to define (macro-)combinators for environment management and the expressions can be read as assembly code (see Section 3.3). More components (e.g. a call stack or a heap) can be introduced in the same way. Similarly to Λ_s, we write $\lambda_e x.M$ for the expression $\mathbf{pop}_e(\lambda x.M)$. We will also use pairs (x,y) and simple pattern matching $(\lambda_i(x,y).M)$. These notations are just syntactic sugar since they are easily translated into pure λ-expressions.

3.2 Evaluation strategy

The K-machine implements the call-by-name evaluation strategy. It uses a push-enter model where unevaluated functions are applied right away and application is an implicit operation. The transformation N in Figure 3 formalizes this choice.

$N : \Lambda \to \Lambda_s$

$N[\![M\ N]\!] = \mathbf{push}_s(N[\![N]\!])\ ;\ N[\![M]\!]$

$N[\![\lambda x.M]\!] = \lambda_s x.N[\![M]\!]$

$N[\![x]\!] = x$

Figure 3 Compilation of call-by-name in the push-enter model (N)

The transformation N compiles applications by pushing the unevaluated argument and applying the function right away. Functions are not returned as results (no closure is built). Variables are bound to arguments which are evaluated when accessed.

Example. Let $M \equiv (\lambda x.x)((\lambda y.y)(\lambda z.z))$ then

$$N[\![M]\!] \equiv \mathbf{push}_s(\mathbf{push}_s(\lambda_s z.z)\ ;\ \lambda_s y.y)\ ;\ \lambda_s x.x \ \Longrightarrow\ \mathbf{push}_s(\lambda_s z.z)\ ;\ \lambda_s y.y \ \Longrightarrow\ \lambda_s z.z \equiv N[\![\lambda z.z]\!]$$

This transformation compiles the call-by-name reduction strategy. Indeed, the transformed form of $(\lambda y.y)(\lambda z.z)$ is $\mathbf{push}_s(\lambda_s z.z)\ ;\ \lambda_s y.y$ which is not a redex because it occurs under a $\mathbf{push}_s$. So, in the transformed expression, the argument cannot be reduced before calling the function $\lambda_s x.x$.

In our framework, the correctness of a compilation step boils down to the proof of a simple program transformation and relies on classical techniques (e.g. structural induction). For example, the correctness of N is stated by Property 1.

Property 1 *For all closed Λ-expressions M, $M \xrightarrow[cbn]{} V$ if and only if $N[\![M]\!] \overset{*}{\Longrightarrow} N[\![V]\!]$*

In the rest of the article, we omit other correctness properties and their proofs which can be found in previous publications [9][10][12].

The transformation N is a very simple way to compile call-by-name. This option is also taken by Tim [13] and most graph-based implementations (e.g. [18][28]). Another choice is the *eval-apply* model, where a λ-abstraction is considered as a result and the application of a function to its argument is an explicit operation. For an expression $\lambda x_1 \ldots \lambda x_n.M$, the K-machine does not build any closure whereas the eval-apply model builds n temporary closures corresponding to the n partial applications of this function. Uncurrying (e.g. [1]) may remove some of this overhead but this optimization is not always possible for functions passed as arguments. On the other hand, the eval-apply model facilitates the compilation of call-by-value and call-by-need. This choice is taken by the SECD [22], the Cam [3] and also some non-strict implementations [14].

3.3 β-reduction

In the λ-calculus, β-reduction is defined as a textual substitution. This operation can be compiled using transformations from Λ_s to Λ_e. These transformations are akin to abstraction algorithms and consist of replacing variables with combinators [35]. In environment-based implementations, substitutions are compiled by storing values to be substituted in an environment. Values are accessed in the environment only when needed. This technique can be compared with the activation records used by imperative language compilers. Some graph based implementations do not use environments but encode each substitution separately [19][35].

The K-machine uses linked environments. Closures are built in constant time and include (a reference to) the complete environment. On the other hand, a chain of links has to be followed when accessing a value. This option is also taken by the SECD and the Cam.

The transformation A (Figure 4) formalizes this choice. The transformation is done with respect to a compile-time environment ρ (initially empty for a closed expression). We note x_i the variable occurring at the ith entry in the environment (i corresponds to the de Bruijn's index of the occurrence).

$A : \Lambda_s \rightarrow env \rightarrow \Lambda_e$

$A[\![M_1 \,;\, M_2]\!]\,\rho = \mathbf{dupl}_e \,;\, A[\![M_1]\!]\,\rho \,;\, \mathbf{swap}_{se} \,;\, A[\![M_2]\!]\,\rho$

$A[\![\mathbf{push}_s\, M]\!]\,\rho = \mathbf{push}_s\,(A[\![M]\!]\,\rho) \,;\, \mathbf{mkclos}$

$A[\![\lambda_s x.M]\!]\,\rho = \mathbf{mkbind} \,;\, A[\![M]\!]\,(\rho,x)$

$A[\![x_i]\!]\,(\ldots((\rho,x_i),x_{i-1})\ldots,x_0) = \mathbf{fst}^i \,;\, \mathbf{snd} \,;\, \mathbf{appclos}$ *with* $\mathbf{fst}^i = \mathbf{fst} \,;\, \ldots \,;\, \mathbf{fst}$ (i times)

Figure 4 Compilation of β-reduction using linked environments (A)

A uses seven new (macro-)combinators (Λ_e closed expressions) to express environment saving and restoring (**dupl**$_e$, **swap**$_{se}$), closure building and calling (**mkclos**, **appclos**), access to values (**fst**, **snd**) and adding a binding in the environment (**mkbind**).

A sequence M_1 **;** M_2 is evaluated by first reducing M_1 using a copy of the environment. Then the result of this evaluation and the environment are swapped so that M_2 can be evaluated. The combinators **dupl**$_e$ and **swap**$_{se}$ can be defined in Λ_e by:

$$\textbf{dupl}_e = \lambda_e e.\textbf{push}_e\ e\ \textbf{;}\ \textbf{push}_e\ e \qquad\qquad \textbf{swap}_{se} = \lambda_s x.\lambda_e e.\textbf{push}_s\ x\ \textbf{;}\ \textbf{push}_e\ e$$

When both components are implemented by the same stack, **swap**$_{se}$ is required to reorder the closure and the environment before reducing M_2. Note that **swap**$_{se}$ is useless when *s* and *e* are implemented by distinct components. This implementation choice is postponed to the instantiation step presented in Section 3.5.

Storing λ-expressions (**push**$_s$ M) and accessing variables (x_i) correspond to closure constructions (**mkclos**) and calls (**appclos**). These combinators can be defined in Λ_e by:

$$\textbf{mkclos} = \lambda_s x.\lambda_e e.\textbf{push}_s(x,e) \qquad\qquad \textbf{appclos} = \lambda_s(x,e).\textbf{push}_e\ e\ \textbf{;}\ x$$

A uses linked environments and adding a binding in the environment as well as building a closure is a constant time operation. On the other hand, a chain of links has to be followed when accessing a value. The corresponding combinators can be defined as follows:

$$\textbf{mkbind} = \lambda_e e.\lambda_s x.\textbf{push}_e(e,x) \qquad \textbf{fst} = \lambda_e(e,x).\textbf{push}_e\ e \qquad \textbf{snd} = \lambda_e(e,x).\textbf{push}_s\ x$$

Example. $A[\![\lambda_s x_1.\lambda_s x_0.\textbf{push}_s\ M\ \textbf{;}\ x_1]\!]\ \rho$

$$= \textbf{mkbind ; mkbind ; dupl}_e\ \textbf{;}\ \textbf{push}_s\ (A[\![M]\!]\ ((\rho,x_1),x_0))\ \textbf{;}\ \textbf{mkclos ;}$$

$$\textbf{swap}_{se}\ \textbf{;}\ \textbf{fst ; snd ; appclos}$$

The transformed expression is only composed of combinators and the β-reduction has been compiled. Variables are only used to define (macro-)combinators. In the example, two bindings (**mkbind ; mkbind**) are added to the current environment, a closure is built for M (**dupl**$_e$ **;** **push**$_s$ (…) **;** **mkclos**), and the closure denoted by x_1 is accessed in the environment by **fst ; snd**.

This implementation of the β-reduction is simple but prone to space leaks. A closure refers to the whole environment even if its code needs only one entry. Space leaks can be avoided by copying only the needed entries of the environment during closure building. This variant can be expressed by inserting code copying the environment before **mkclos** in *A*. In this case, each closure has its own environment which can be represented by a vector. Access to values is therefore a constant access time operation. This choice is taken by Tim [13], the SML-NJ compiler [1] and several other implementations [14][28].

3.4 Composition

The push-enter model and shared environments are natural options for the compilation of call-by-name and β-reduction respectively. In our view, these choices are the essence of the K-machine. The composition of N and A gives the compilation rules of the K-machine:

$K : \Lambda \to env \to \Lambda_e$

$K[\![M\ N]\!]\ \rho = \mathbf{dupl}_e\ \mathbf{;}\ \mathbf{push}_s\ (K[\![N]\!]\ \rho)\ \mathbf{;}\ \mathbf{mkclos}\ \mathbf{;}\ \mathbf{swap}_{se}\ \mathbf{;}\ K[\![M]\!]\ \rho$

$K[\![\lambda x.M]\!]\ \rho = \mathbf{mkbind}\ \mathbf{;}\ K[\![M]\!]\ (\rho,x)$

$K[\![x_i]\!]\ (\ldots((\rho,x_i),x_{i-1})\ldots,x_0) = \mathbf{fst}^i\ \mathbf{;}\ \mathbf{snd}\ \mathbf{;}\ \mathbf{appclos}$

Figure 5 Compiler for the K-machine ($K = A \circ N$)

Intuitively, the rules can be read as follows. Evaluating an application $M\ N$, amounts to building a closure made of N and a reference to the environment (**dupl**$_e$ **;** **push**$_s$($K[\![N]\!]\ \rho$) **;** **mkclos**), and evaluating M with the environment at the top (**swap**$_{se}$ **;** $K[\![M]\!]\ \rho$). The evaluation of a λ-abstraction binds its variable with the top of the data stack in the current environment (**mkbind**) and evaluates the body $K[\![M]\!]\ (\rho,x)$. The evaluation of a variable amounts to fetching (**fst**^{i} **;** **snd**) and executing (**appclos**) the corresponding closure in its environment.

This compilation process is obtained by composing two independent and generic transformations. Actually, this specific composition makes the stack of environments useless. The duplication of (the reference to) the environment is immediately followed by a closure construction which consumes it: a single environment suffices. The component e can be implemented by a register rather than a stack. Note that this is not always the case (see Section 4.1).

3.5 Instantiation

Until now, we have just assumed that combinators **;**, **push**$_i$ and **pop**$_i$ respect properties (assoc), (β_i) and (η_i) ($i \in \{s, e\}$). Their actual definitions are chosen as a last compilation step. This allows us to shift from the β_i-reduction in Λ_i to a state-machine-like expression reduction.

The most natural definition for the sequencing combinator is **;** = $\lambda a.\lambda b.\lambda c.a\ (b\ c)$, that is

$$M\ \mathbf{;}\ N = \lambda c.M\ (N\ c)$$

The (fresh) variable c can be seen as a continuation which implements the sequencing. The K-machine keeps the data stack s and the environment e separate. This is formalized by the following definitions:

$$\mathbf{push}_s = \lambda n.\lambda c.\lambda s.\lambda e.c\ (s,n)\ e \qquad\qquad \mathbf{pop}_s = \lambda f.\lambda c.\lambda(s,x).\lambda e.f\ x\ c\ s\ e$$

$\mathbf{push}_e = \lambda n.\lambda c.\lambda s.\lambda e.c\ s\ (e,n)$ $\qquad$ $\mathbf{pop}_e = \lambda f.\lambda c.\lambda s.\lambda(e,x).f\ x\ c\ s\ e$

It is easy to check that these definitions respect properties (β_i), (η_i) and (assoc). The reduction (using classical β-reduction and normal order) of our expressions can be seen as state transitions of an abstract machine with three components (code, data stack, environment), e.g.:

$$((M\ ;\ N)\ C)\ S\ E \rightarrow (M\ (N\ C))\ S\ E$$

$(\mathbf{push}_s\ N\ C)\ S\ E \rightarrow C\ (S,N)\ E$ $\qquad$ $((\lambda_s x.M)\ C)\ (S,N)\ E \rightarrow (M[N/x]\ C)\ S\ E$

$(\mathbf{push}_e\ N\ C)\ S\ E \rightarrow C\ S\ (E,N)$ $\qquad$ $((\lambda_e x.M)\ C)\ S\ (E,N) \rightarrow (M[N/x]\ C)\ S\ E$

These definitions* entail the following rewriting rules:

$\mathbf{dupl}_e\ C\ S\ (E_0,E_1)$	$\rightarrow$	$C\ S\ ((E_0,E_1),E_1)$
$\mathbf{swap}_{se}\ C\ (S,N)\ (E_0,E_1)$	$\rightarrow$	$C\ (S,N)\ (E_0,E_1)$
$\mathbf{mkclos}\ C\ (S,N)\ (E_0,E_1)$	$\rightarrow$	$C\ (S,(N,E_1))\ E_0$
$\mathbf{mkbind}\ C\ (S,N)\ (E_0,E_1)$	$\rightarrow$	$C\ S\ (E_0,(E_1,N))$
$\mathbf{fst}\ C\ S\ (E_0,(E_1,N))$	$\rightarrow$	$C\ S\ (E_0,E_1)$
$\mathbf{snd}\ C\ S\ (E_0,(E_1,N))$	$\rightarrow$	$C\ (S,N)\ E_0$
$\mathbf{appclos}\ C\ (S,(N,E_1))\ E_0$	$\rightarrow$	$N\ C\ S\ (E_0,E_1)$

The choice of keeping the data and environment stacks separate brings new properties. In particular, there is no need to swap the environment and the newly built closure. Indeed, the combinator $\mathbf{swap}_{se}$ is the identity function and can be discarded. In order to get closer to the usual descriptions of the K-machine, we use the following combinators:

$\mathbf{clos}\ N = \mathbf{dupl}_e\ ;\ \mathbf{push}_s\ N\ ;\ \mathbf{mkclos}$ $\qquad$ $\mathbf{access}(i) = \mathbf{fst}^i\ ;\ \mathbf{snd}\ ;\ \mathbf{appclos}$

We get the following reduction rules for the code produced by K:

$(\mathbf{clos}\ N\ ;\ M)\ C\ S\ (E_0,E_1)$	$\rightarrow$	$M\ C\ (S,(N,E_1))\ (E_0,E_1)$
$(\mathbf{mkbind}\ ;\ M)\ C\ (S,N)\ (E_0,E_1)$	$\rightarrow$	$M\ C\ S\ (E_0,(E_1,N))$
$\mathbf{access}(i)\ C\ S\ (E_0,(...((E,(M_i,E_i)),N_{i-1}),...,N_0))$	$\rightarrow$	$M_i\ C\ S\ (E_0,E_i)$

Figure 6 Reduction rules of the K-machine

* Note that the definitions use a few useless parentheses to make the three components more explicit. Everywhere else, the parentheses are dropped using the usual convention of association to the left.

It is easy to see from these rules that the continuation *C* is not used and can be replaced by *e.g.* a function **end** printing the result. The stack of environments is useless and can be replaced by a single environment (E_0 is not used and could be removed from the rules). With these two simplifications, we get the rules of the standard K-machine acting on compiled code. Our presentation is actually exactly the same as Leroy's [24] (p. 25) where **clos** *N* and **mkbind** are written **Push**(*N*) and **Grab**, respectively.

The classic presentation of Figure 1 is based on state transitions involving de Bruijn's λ-expressions. Source λ-expressions can be translated into de Bruijn's form using the following abstraction:

$$(M\ N)_\rho = \ M_\rho\ N_\rho \qquad (\lambda x.M)_\rho = \ \lambda M_{(\rho,x)} \qquad x_{\ (\ldots((\rho,x),xi\text{-}1)\ldots,x0)} = i$$

and the de Bruijn's form of a closed λ-expression *M* is M_0. It is easy to see that our functional machine code and the machine states of Figure 1 are related by the following relation:

$$K[\![M]\!]\ \rho\ C\ S\ (E_0,E) \approx (M_\rho,\ S,\ E)$$

for all closed *C, S,* E_0, *E* and for all *M* with all its free variables in ρ. The initial arguments/ configuration to reduce a closed expression *M* are

$$K[\![M]\!]\ ()\ \mathbf{end}\ ()\ ((),()) \approx (M_0,\ (),\ ())$$

where $K[\![M]\!]\ ()$ takes as parameters an initial continuation **end**, an empty argument stack () and a stack of environments that contains a single empty environment ((),()). The combinators **mkbind**, $\mathbf{fst}^{i+1}$ and (**snd ; appclos**) correspond respectively to the λ, *i*+1 and 0 of Figure 1. The definition of a relation between states is a standard technique to prove the correctness or equivalence of implementations [26]. A reduction step $E \rightarrow F$ of the machine in Figure 6 is simulated by a sequence of reduction steps of the machine of Figure 1 (i.e. a state related to *E* is rewritten into a state related to *F*). Actually, splitting the reduction rule of **access**(*i*) into two reduction rules (one for $\mathbf{fst}^{i+1}$ and one for **snd ; appclos**) is sufficient to get a one to one correspondence between the reduction steps of the two machines.

4 Variants

In the previous section, we focused on the standard call-by-name K-machine. We now describe several variants that appear in the literature. We present a strict (i.e. call-by-value) and a lazy (i.e. call-by-need) version of the machine as well as two alternatives for environment management.

4.1 A strict variant

In this section, we present a push-enter transformation for the call-by-value reduction strategy. The composition of this transformation with A yields a strict variant of the K-machine. With call-by-value, a function can be evaluated as an argument. In this case, it cannot be applied right away but must be returned as a result. In order to detect when its evaluation is over, there has to be a way to distinguish whether its argument is present or absent: this is the

role of *marks*. After a function is evaluated, a test is performed: if there is a mark, the function is returned as a result (and a closure is built); otherwise, the argument is present and the function is applied. This technique avoids building some closures, but at the price of performing dynamic tests.

The mark ε is supposed to be a value that can be distinguished from others. Functions are transformed into $\mathbf{grab}_s\ M$ which satisfies the following reduction rules. When a mark is present the function M is returned as a result:

$$\mathbf{push}_s\ \varepsilon\ ;\ \mathbf{grab}_s\ M \ \Rightarrow\!\!\!\!\Rightarrow \mathbf{push}_s\ M$$

When no mark is present, the function M is applied to its argument N:

$$\mathbf{push}_s\ N\ ;\ \mathbf{grab}_s\ M \ \Rightarrow\!\!\!\!\Rightarrow \mathbf{push}_s\ N\ ;\ M$$

The combinator $\mathbf{grab}_s$ and the mark ε can be defined in Λ_s. In practise, $\mathbf{grab}_s$ is implemented using a conditional testing the presence of a mark. The transformation for right-to-left call-by-value is described in Figure 7.

$V : \Lambda \to \Lambda_s$

$V[\![M\ N]\!] = \mathbf{push}_s\ \varepsilon\ ;\ V[\![N]\!]\ ;\ V[\![M]\!]$

$V[\![\lambda x.M]\!] = \mathbf{grab}_s\ (\lambda_s x.V[\![M]\!])$

$V[\![x]\!] = \mathbf{grab}_s\ x$

Figure 7 Compilation of call-by-value (V)

Example. Let $M \equiv (\lambda x.x)((\lambda y.y)(\lambda z.z))$; then

$$\begin{aligned} V[\![M]\!] &\equiv \mathbf{push}_s\ \varepsilon\ ;\ \mathbf{push}_s\ \varepsilon\ ;\ \mathbf{grab}_s\ (\lambda_s z.\mathbf{grab}_s\ z)\ ;\ \mathbf{grab}_s(\lambda_s y.\mathbf{grab}_s\ y)\ ;\ \mathbf{grab}_s(\lambda_s x.\mathbf{grab}_s\ x) \\ &\Rightarrow\!\!\!\!\Rightarrow \mathbf{push}_s\ \varepsilon\ ;\ \mathbf{push}_s(\lambda_s z.\mathbf{grab}_s\ z)\ ;\ \mathbf{grab}_s(\lambda_s y.\mathbf{grab}_s\ y)\ ;\ \mathbf{grab}_s(\lambda_s x.\mathbf{grab}_s\ x) \\ &\Rightarrow\!\!\!\!\Rightarrow \mathbf{push}_s\ \varepsilon\ ;\ \mathbf{grab}_s(\lambda_s z.\mathbf{grab}_s\ z)\ ;\ \mathbf{grab}_s(\lambda_s x.\mathbf{grab}_s\ x) \\ &\Rightarrow\!\!\!\!\Rightarrow \mathbf{push}_s(\lambda_s z.\mathbf{grab}_s\ z)\ ;\ \mathbf{grab}_s(\lambda_s x.\mathbf{grab}_s\ x) \\ &\Rightarrow\!\!\!\!\Rightarrow \mathbf{grab}_s(\lambda_s z.\mathbf{grab}_s\ z) \equiv V[\![\lambda z.z]\!] \end{aligned}$$

In this example, initially the function $\lambda z.z$ has no argument, so it is returned as a result. No closure is built for $\lambda y.y$ which takes $\lambda z.z$ as a parameter. Similarly, no closure is built for $\lambda x.x$ which takes its evaluated argument $\lambda z.z$ as a parameter. This transformation compiles call-by-value. Indeed, in the transformed expression, the function $\lambda x.x$ cannot be called before its argument is evaluated (the reduction rules of $\mathbf{grab}_s$ require *either* a mark $\mathbf{push}_s\ \varepsilon$ *or* a closure $\mathbf{push}_s\ N$).

A strict version of the K-machine can be modeled by $K_s = A \circ V$. Figure 8 gathers the rules obtained after the simplification of this composition. In particular, no closure is built for the constant ε. We use $\mathbf{grab}_e$ that satisfies the two following reduction rules:

$$\mathbf{push}_s\ \varepsilon\ ;\ \mathbf{push}_e\ X\ ;\ \mathbf{grab}_e\ M \ \Rightarrow\!\!\!\!\Rightarrow\ \mathbf{push}_s\ M\ ;\ \mathbf{push}_e\ X\ ;\ \mathbf{mkclos}$$

$$\mathbf{push}_s\ N\ ;\ \mathbf{push}_e\ X\ ;\ \mathbf{grab}_e\ M \ \Rightarrow\!\!\!\!\Rightarrow\ \mathbf{push}_s\ N\ ;\ \mathbf{push}_e\ X\ ;\ M$$

$K_s : \Lambda \rightarrow env \rightarrow \Lambda_e$

$K_s[\![M\ N]\!]\ \rho = \mathbf{dupl}_e\ ;\ \mathbf{push}_s\ \varepsilon\ ;\ \mathbf{swap}_{se}\ ;\ K_s[\![N]\!]\ \rho\ ;\ \mathbf{swap}_{se}\ ;\ K_s[\![M]\!]\ \rho$

$K_s[\![\lambda x.M]\!]\ \rho = \mathbf{grab}_e(\mathbf{mkbind}\ ;\ K_s[\![M]\!]\ (\rho,x))$

$K_s[\![x_i]\!]\ (\ldots((\rho,x_i),x_{i-1})\ldots,x_0) = \mathbf{grab}_e(\mathbf{access}(i))$

Figure 8 Strict K-Machine ($K_s = A \circ V$)

With the same definition for the basic combinators as in Section 3.5, we get the following reduction rules ($\mathbf{swap}_{se}$ is, as in Section 3.5, the identity function) for the strict K-machine:

$$(\mathbf{dupl}_e\ ;\ \mathbf{push}_s\ \varepsilon\ ;\ N\ ;\ M)\ C\ S\ (E_0,E_1) \rightarrow N\ (M\ C)\ (S,\varepsilon)\ ((E_0,E_1),E_1)$$

$$\mathbf{grab}_e\ M\ C\ (S,\varepsilon)\ (E_0,E_1) \rightarrow C\ (S,(M,E_1))\ E_0$$

$$\mathbf{grab}_e\ M\ C\ (S,N)\ (E_0,E_1) \rightarrow M\ C\ (S,N)\ (E_0,E_1)$$

$$(\mathbf{mkbind}\ ;\ M)\ C\ (S,N)\ (E_0,E_1) \rightarrow M\ C\ S\ (E_0,(E_1,N))$$

$$\mathbf{access}(i)\ C\ S\ (E_0,(\ldots((E,(M_i,E_i)),N_{i-1}),\ldots,N_0)) \rightarrow M_i\ C\ S\ (E_0,E_i)$$

Besides marks, there are two important differences between the strict variant and the standard call-by-name machine. Firstly, the continuation evolves: in the first rule, the evaluation of N takes place with the new continuation ($M\ C$) recording the fact that M should be reduced after N. Likewise, the evaluation of a $\mathbf{grab}_s\ M$ with a mark in the stack (second rule) returns M and the reduction proceeds with the code stored in the continuation. Secondly, the stack of environments is now needed and used. In the first rule, the current environment (E_1) is saved and will remain in the stack of environments throughout the evaluation of N.

A conventional machine executes linear sequences of basic instructions. In our framework, we could make calls and returns explicit using another component k (with its associated $\mathbf{push}_k$ and $\mathbf{pop}_k$ combinators) to represent the call stack. A transformation S can be designed in order to save explicitly the code following a function call using $\mathbf{push}_k$, and to return to it with **rts** ($= \lambda_k f.f$) when the function ends. Another solution is to transform expressions into CPS before the transformation A. Continuations are treated as regular functions by A so that return addresses are represented by closures. This solution is used in the SML-NJ compiler [1]. We do not describe this linearization process here (see [12] pp. 369-370).

There are several differences between our description and Leroy's [24]. The **Grab** instruction used by Zinc is a combination of our $\mathbf{grab}_e$ (in fact, a recursive version) and **mkbind** combinators. Control transfers are implemented by building closures in the data stack. This machine could be modeled precisely in our framework using a variant of the **grab** combinator.

4.2 A lazy variant

After the evaluation of a closure, a call-by-need (or lazy) implementation updates the closure with its normal form. The transformation N makes it impossible to distinguish results of closures (which have to be updated) from regular functions (which are applied right away). This problem is solved, as in V, with the help of marks.

The transformation $L_\uparrow$ in Figure 9 introduces marks in order to stop the normal evaluation process and update closures. This transformation implements a *caller-update* scheme. Each time a variable (i.e. a closure) is accessed, a mark is pushed in order to pause the evaluation when the closure is in normal form. Like $\mathbf{grab}_s$, the combinator $\mathbf{updt}_s$ tests the presence of a mark before the evaluation of a normal form. When no mark is present, $\mathbf{updt}_s\ M$ proceeds with the evaluation of M. When a mark is present, the last closure evaluated must be updated with its normal form M.

$L_\uparrow : \Lambda \to \Lambda_s$

$L_\uparrow [\![M\ N]\!] = \mathbf{push}_s(L_\uparrow [\![N]\!])\ ;\ L_\uparrow [\![M]\!]$

$L_\uparrow [\![\lambda x.M]\!] = \mathbf{updt}_s(\lambda_s x.L_\uparrow [\![M]\!])$

$L_\uparrow [\![x]\!] = \mathbf{push}_s\ \varepsilon\ ;\ x$

Figure 9 Compilation of call-by-need (caller-update, push-enter model)

The drawback of this scheme is that it updates a closure every time it is accessed. The *callee-update* scheme updates closures only the first time they are accessed. Once in normal form, all the subsequent accesses will not entail further (useless) updates.

$L_\downarrow : \Lambda \to \Lambda_s$

$L_\downarrow [\![M\ N]\!] = \mathbf{push}_s(\mathbf{push}_s\ \varepsilon\ ;\ L_\downarrow [\![N]\!])\ ;\ L_\downarrow [\![M]\!]$

$L_\downarrow [\![\lambda x.M]\!] = \mathbf{updt}_s(\lambda_s x.L_\downarrow [\![M]\!])$

$L_\downarrow [\![x]\!] = x$

Figure 10 Compilation of call-by-need (callee-update, push-enter model)

This last scheme is more efficient and is implemented by most environment-based implementations. This choice can be formalized by changing the compilation rules as shown in Figure 10. Closures are now responsible for updating themselves. They begin by pushing a mark ε so that $\mathbf{updt}_s$ can update it with its normal form. Code corresponding to normal forms does not push marks and the future accesses to updated closures will not trigger updating. Note that when the argument of an application is already a normal form (a λ-abstraction) at compile time then it is useless to build an updatable closure. This optimization could be expressed by the additional rule:

$$L_{\downarrow}[\![M\ (\lambda x.N)]\!] = \mathbf{push}_s(\lambda_s x.L_{\downarrow}[\![N]\!])\ ;\ L_{\downarrow}[\![M]\!]$$

The composition of $L_{\downarrow}$ and A gives the following compilation rules where $\mathbf{updt}_e$ is a variant of $\mathbf{updt}_s$ that takes into account the component e.

$K_l : \Lambda \rightarrow env \rightarrow \Lambda_e$

$K_l[\![M\ N]\!]\ \rho \quad = \mathbf{clos}(\mathbf{push}_s\ \varepsilon\ ;\ \mathbf{swap}_{se}\ ;\ K_l[\![N]\!]\ \rho)\ ;\ \mathbf{swap}_{se}\ ;\ K_l[\![M]\!]\ \rho$

$K_l[\![\lambda x.M]\!]\ \rho \quad = \mathbf{updt}_e(\mathbf{mkbind}\ ;\ K_l[\![M]\!]\ (\rho,x))$

$K_l[\![x_i]\!]\ (\ldots((\rho,x_i),x_{i-1})\ldots,x_0) = \mathbf{access}(i)$

Figure 11 Lazy K-Machine ($K_l = A \circ L_{\downarrow}$)

The complete description of the lazy K-machine requires modelling sharing and updating. A memory component (a heap) must be introduced in order to store (and share) closures. This can be done (see [12]) but encoding a state in a purely functional framework is intricate. We prefer not to detail this step here. Instead, we leave temporarily our framework and present intuitively the rules of the lazy K-machine. We use the notion of address (written @) and a component H whose modifications and accesses are written $H[@\leftarrow M]$ and H(@) respectively. With the same definition for the basic combinators as in Section 3.5, we get the following rules for the lazy K-machine:

$(\mathbf{clos}\ N\ ;\ M)\ C\ S\ (E_0,E_1)\ H$	$\rightarrow$	$M\ C\ (S,@_{new})\ (E_0,E_1)\ H[@_{new}\leftarrow (N,E_1)]$	
$(\mathbf{push}_s\ \varepsilon\ ;\ M)\ C\ S\ (E_0,E_1)\ H$	$\rightarrow$	$M\ C\ (S,(\varepsilon,@_{last}))\ (E_0,E_1)\ H$	
$\mathbf{updt}_e\ M\ C\ (S,(\varepsilon,@))\ (E_0,E_1)\ H$	$\rightarrow$	$\mathbf{updt}_e\ M\ C\ S\ (E_0,E_1)\ H[@\leftarrow (M,E_1)]$	
$\mathbf{updt}_e\ M\ C\ (S,N)\ (E_0,E_1)\ H$	$\rightarrow$	$M\ C\ (S,N)\ (E_0,E_1)\ H$	
$(\mathbf{mkbind}\ ;\ M)\ C\ (S,@)\ (E_0,E_1)\ H$	$\rightarrow$	$M\ C\ S\ (E_0,(E_1,@))\ H$	
$\mathbf{access}(i)\ C\ S\ (E_0,(\ldots((E_1,@_i),@_{i-1},,\ldots,@_0))\ H$	$\rightarrow$	$M_i\ C\ S\ (E_0,E_i)H$	with $H(@_i)=(M_i,E_i)$

Compared to the previous versions of the machine, the main differences lie in the updating ($\mathbf{updt}_e$ rules) and the representation of closures by addresses in the stack and environ-

ment. Closure building is done at a fresh address $@_{new}$ in the heap and its address is pushed onto *S*. Before its evaluation, a closure pushes a mark with its own address in the heap ($@_{last}$, the last accessed address is the closure's address). This pair (ε,$@_{last}$) is used by the first rule of $\mathbf{updt}_e$ to perform updating. The other rules are similar as before except that they manipulate addresses instead of closures directly. Our description is identical to the KP-machine of Crégut [4].

4.3 Refined environment managements

4.3.1 Two-level environments

Accesses in the environment of the standard K-machine are linear time operations. We present here another environment-based abstraction $\mathcal{A}_2$ (Figure 12) proposed by Krivine [21] that improves environment lookups. The transformation $\mathcal{A}_2$ relies on two-level environments to deal with sequences of λ-abstractions.

$\mathcal{A}_2 : \Lambda_s \to env \to \Lambda_e$

$\mathcal{A}_2\,[\![M_1\ ;\ M_2]\!]\,\rho = \mathbf{dupl}_e\ ;\ \mathcal{A}_2\,[\![M_1]\!]\,\rho\ ;\ \mathbf{swap}_{se}\ ;\ \mathcal{A}_2\,[\![M_2]\!]\,\rho$

$\mathcal{A}_2\,[\![\mathbf{push}_s\ M]\!]\,\rho = \mathbf{push}_s\ (\mathcal{A}_2\,[\![M]\!]\,\rho)\ ;\ \mathbf{mkclos}$

$\mathcal{A}_2\,[\![\lambda_s x_n \ldots \lambda_s x_0.M]\!]\,\rho = \mathbf{mkbind}_{n+1}\ ;\ \mathbf{mkenv}_e\ ;\ \mathcal{A}_2\,[\![M]\!]\,(\rho,(((),x_n)\ldots,x_0))$

$\mathcal{A}_2\,[\![x_{i,j}]\!]\,(\ldots((\rho,((e,x_j)\ldots,x_0)),\rho_{i-1})\ldots,\rho_0) = \mathbf{fst}^{\,i}\ ;\ \mathbf{snd}\ ;\ \mathbf{fst}^{\,j}\ ;\ \mathbf{snd}\ ;\ \mathbf{appclos}$

Figure 12 Compilation of β-reduction with two-level environments ($\mathcal{A}_2$)

The transformation of a λ_s-expression $\lambda_s x_n \ldots \lambda_s x_0.M$ entails the construction of a local environment of length $n+1$ ($\mathbf{mkbind}_{n+1} = \lambda_e e_g.\lambda_s x_n \ldots \lambda_s x_0.\mathbf{push}_e\ e_g\ ;\ \mathbf{push}_e\ (((),x_n)\ \ldots,\ x_0)$) which is then appended to the main environment ($\mathbf{mkenv}_e = \lambda_e e_l.\lambda_e e_g.\mathbf{push}_e\ (e_g,e_l)$). An environment is represented by a tree (or list of lists) of closures. The variable occurring at the *j*th entry of the *i*th local environment is denoted by $x_{i,j}$.

The benefit of two-level environments is to improve access time. Consider the λ-expression ($\lambda z.M\ (\lambda x_1 \ldots \lambda x_n.z)$). In the standard K-machine, the access to *z* would be compiled into (**fst**^{n} **; snd ; appclos**). With two-level environments, the access to *z* is constant time (**fst ; snd ; snd ; appclos**). On the other hand, this technique suffers space leak problems.

This treatment of environments suggests to use tuples (or vectors) of closures. This can be done by introducing a family of indexed combinators to build and access tuples:

$$\mathbf{mkbind}(n) = \lambda_e e.\lambda_s x_n \ldots \lambda_s x_1.\mathbf{push}_e(e,(x_1,\ldots,x_n))$$

$$\mathbf{access}(i,j) = \lambda_e(\ldots((e,(c_0,\ldots,(x_j,e_j),\ldots,c_n)),e_{i-1}),\ldots,e_0).\mathbf{push}_e\ e_j\ ;\ x_j$$

and by modifying the transformation accordingly:

$A_2 [\![\lambda_s x_n \ldots \lambda_s x_0.M]\!]\ \rho$ = **mkbind**$(n+1)$ **;** $A_2 [\![M]\!]\ (\rho,(((),x_n)\ldots,x_0))$

$A_2 [\![x_{i,j}]\!]\ (\ldots((\rho,((\rho',x_j)\ldots,x_0)),\rho_{i-1})\ldots,\rho_0)$ = **access**(i,j)

With the same definition for the basic combinators as in Section 3.5, we get the following rules for A_2 o N:

(**clos** N **;** M) $C\ S\ (E_0,E_1)$ $\rightarrow$ $M\ C\ (S,(N,E_1))\ (E_0,E_1)$

(**mkbind**(n) **;** M) $C\ (\ldots((S,N_n),N_{n-1}),\ldots,N_1))\ (E_0,E_1)$ $\rightarrow$ $M\ C\ S\ (E_0,(E_1,(N_1,\ldots,N_n)))$

access(i,j) $C\ S\ (E_0,(\ldots$
$((E,(V_0,\ldots,(M_j,E_j),\ldots,V_n)),N_{i-1}),\ldots,N_0))$ $\rightarrow$ $M_j\ C\ S\ (E_0,E_j)$

The second level of the environment (represented by an n-ary tuple) is accessed in constant time.

4.3.2 Super closures

The K-machine avoids the construction of some intermediate closures. However, it still builds n closures for the expression $M\ N_1 \ldots N_n$ (for each argument N_i), each one with the same environment. A variant, proposed by [23], builds only one *super-closure* of the form $(code_1, \ldots, code_n, env)$ made of a code vector and the environment. It replaces n pairs by a single n+1-uplet and avoids n-1 references to the environment. This variant of environment management is formalized in our framework by the transformation A_3 (Figure 13).

$A_3 : \Lambda_s \rightarrow env \rightarrow \Lambda_e$

$A_3 [\![\mathbf{push}_s\ N_n\ \mathbf{;}\ \ldots\ \mathbf{;}\ \mathbf{push}_s\ N_1\ \mathbf{;}\ M]\!]\ \rho$

$= \mathbf{dupl}_e\ \mathbf{;}\ \mathbf{push}_s\ (A_3 [\![N_n]\!]\ \rho)\ \mathbf{;}\ \ldots\ \mathbf{;}\ \mathbf{push}_s\ (A_3 [\![N_1]\!]\ \rho)\ \mathbf{;}\ \mathbf{mkclosS}(n)\ \mathbf{;}\ \mathbf{swap}_{se}\ \mathbf{;}\ A_3 [\![M]\!]\ \rho$

$A_3 [\![\lambda_s x_n \ldots \lambda_s x_1.M]\!]\ \rho$ = **mkbindS**(n) **;** $A_3 [\![M]\!]\ ((\rho,x_n)\ldots,x_1)$

$A_3 [\![x_i]\!]\ ((\rho,x_i)\ldots,x_1)$ = **accessS**(i)

Figure 13 Compilation of β-reduction with super-closures (A_3)

This transformation uses a new combinator to build a super-closure:

mkclosS$(n) = \lambda_s x_1 \ldots \lambda_s x_n.\lambda_e e.\mathbf{push}_s\ (e,x_1,\ldots,x_n)$

Such a super-closure is split by **mkbindS**(n) if required. This combinator relies on a run-time check to adapt the size of super-closures to the arity of functions.

$\mathbf{mkbindS}(n) = \lambda_e e_0.\lambda_s(e,x_1,\ldots,x_m).$

$$\begin{array}{lll} \textbf{case} & n{=}m \rightarrow \textbf{push}_s\,(e,x_1,\ldots,x_m)\ \textbf{;}\ \textbf{push}_e\ e_0\ \textbf{;}\ \textbf{mkbind} \\ & n{<}m \rightarrow \textbf{push}_s\,(e,x_1,\ldots,x_n)\ \textbf{;}\ \textbf{push}_e\ e_0\ \textbf{;}\ \textbf{mkbind}\ \textbf{;}\ \textbf{push}_s\,(e,x_{n+1},\ldots,x_m) \\ & n{>}m \rightarrow \textbf{push}_s\,(e,x_1,\ldots,x_m)\ \textbf{;}\ \textbf{push}_e\ e_0\ \textbf{;}\ \textbf{mkbind}\ \textbf{;}\ \textbf{mkbindS}(n{-}m) \end{array}$$

In the first case ($n{=}m$), the super-closure has the right size and it is added to the environment. In the second case ($n{<}m$), the super-closure is too large and is split. Finally, in the third case ($n{>}m$), the super-closure is too small, so it is added to the environment and the next super-closure is considered.

Environment accesses rely also on dynamic checks. In the first case, the first super-closure of the environment is skipped. In the second case, the super-closure is opened at the right index.

$$\begin{array}{lll} \mathbf{accessS}(n) = \lambda_e(e_0,(e,x_1,\ldots,x_m)).\ \textbf{case} & n{>}m & \rightarrow \textbf{push}_e\ e_0\ \textbf{;}\ \mathbf{accessS}(n{-}m) \\ & \textit{otherwise} & \rightarrow \textbf{push}_e\ e\ \textbf{;}\ x_n \end{array}$$

These combinators allow to delay (and sometimes to suppress) closure building at the expense of dynamic checks. However, repeated application of **mkbindS**(n) (second case) may lead to build more closures than A. Super-closures have also sharing and space leak problems [23]. Even if super-closures turn out not to be a practical optimization, the simplicity of the K-machine has clearly facilitated the study of such a complex feature.

5 Conclusion

In this paper, we modeled the K-machine as a sequence of two program transformations.

- The first transformation compiles the reduction strategy according to a push-enter model. Compared to the eval-apply model, this choice avoids useless closure building. For call-by-name, this is the most natural choice. When more realistic languages are considered (e.g. equipped with strict operators and a lazy or call-by-value strategy), the push-enter model becomes more complicated. Marks and dynamic tests become necessary to return intermediate results and/or to update closures.
- The second transformation compiles the β-reduction using linked closures and environments. Compared to other environment-based abstractions, this scheme promotes sharing and closure building is a constant time operation. The main drawback is that this choice leads to space leaks. In a real functional language implementation, space leaks should be avoided by copying only the needed part of the environment in closures.

In our view, these two choices are the essence of the Krivine machine. K-machines are push-enter, linked-environment machines. Still this class is large and contains many variants:

- We have considered in this paper three versions of the push-enter model for call-by-name, call-by-value and call-by-need. For this last strategy, updating can be implemented according two strategies (caller update or callee update).
- Three versions (basic, two-level environments and super closures) of environment management have been presented. All of them use linked environments.
- Call-by-value (actually, any reduction strategy as soon as the source language has strict operators) requires the implementation of function calls and returns. This can be implemented by a stack of return addresses or by closures in the heap.
- Finally, as described by the last instantiation step, an implementation can choose to merge or keep components separate (e.g. the environments can be stored in the argument stack). Depending whether there are only two components or also a control stack, this give rises to 2 or 5 possible choices.

These options are to a large extent independent and can be combined freely. Actually, just by combining the choices mentioned above, one could derive nearly 100 K-machines. Of course, the design space of functional languages implementations is much larger. For example:

- Like the K-machine, Tim [13] relies on the push-enter model but uses copied environments instead. Its call-by-name version can be described as the composition of the transformation N and a new transformation compiling environment management [12].
- Like the K-machine, the SECD [22] and the Cam [3] use linked environments but rely on the eval-apply model. They can be described as the composition of a new transformation compiling call-by-value and A [12].
- Graph-based implementations (e.g. the G-machine [18]) rewrite more or less interpretively a graph representation of the program. After optimisations, they become close to push-enter environment machines.
- The call-value and call-by-name machines of Fradet and Le Métayer [14] and the SML-NJ compiler [1] are based on the eval-apply model and use copied-environments.

Realistic functional languages include constants, data structures and recursion. These features bring new implementation choices. Constants can be stored into the stack *s* or in yet another component. The latter option has the advantage of marking a difference between pointers and values which can be exploited by the garbage collector. Recursion can be implemented using circular environments or jumps to functions' addresses. Data structures can be represented using tags or higher-order functions [29][13]. These new choices would definitely allow the derivation of more than 700 K-machines. Optimisations (e.g. unboxing [25], let-floating [30], sophisticated data representations [15][33], etc.) and other compilation steps (register allocation) would also bring new options.

In this paper, we have presented the K-machine from an implementer's point of view. We have focused on pure λ-expressions and weak-head reduction strategies. The K-machine has appeared in many other contexts.

The K-machine has served as a workbench to implement head and strong reduction strategies [5] and to study the integration and impact of static analyses [31]. The K-machine has

been presented in categorical terms [2] and its relationship with the Tim has been studied [4]. Not surprisingly, the K-machine has been one of the easiest machine to derive from formal semantics. The standard and lazy versions have been derived from operational semantics of call-by-name [16] or call-by-need [32]. λ-calculi with explicit substitutions aim at formalizing and proving functional language implementations. The K-machine has been a natural candidate to present and test some of these calculi [6][17]. The λμ-calculus [27] is an extension of the λ-calculus with control operators. Several implementations of this calculus are based on extensions of the K-machine [8] [34].

All this work demonstrates that the simplicity of the Krivine machine makes it a valuable tool to study new implementation techniques and λ-calculi extensions.

REFERENCES

[1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press. 1992.

[2] A. Asperti. A categorical understanding of environment machines. *Journal of Functional Programming*, 2(1), pp. 23-59,1992.

[3] G. Cousineau, P.-L. Curien and M. Mauny, The categorical abstract machine. *Science of Computer Programming*, 8(2), pp. 173-202, 1987.

[4] P. Crégut. *Machines à environnement pour la réduction symbolique et l'évaluation partielle*. Thèse de l'université de Paris VII, 1991.

[5] P. Crégut. An abstract machine for lambda-terms normalization, In *Proc. of LFP'90*, pp. 333-340, ACM Press, June 1990.

[6] P.-L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82, pp. 389-402, 1991.

[7] N. G. De Brujin. λ-calculus notation with nameless dummies: a tool for automatic formula manipulation, with application to Church Rosser theorem. In *Indagationes mathematicae*, 34, pp. 381-392, 1972.

[8] P. De Groote. An environment machine for the lambda-mu-calculus. *Mathematical Structures in Computer Science*, 8(6), pp. 637-669, 1998.

[9] R. Douence and P. Fradet. A taxonomy of functional language implementations. Part I: call-by-value. *INRIA research report 2783*, Jan. 1996.

[10] R. Douence and P. Fradet. A taxonomy of functional language implementations. Part II: call-by-name, call-by-need, and graph reduction. *INRIA research report 3050*, Nov. 1996.

[11] R. Douence. *Décrire et comparer les mises en œuvre de langages fonctionnels*. PhD Thesis, University of Rennes I, 1996.

[12] R. Douence and P. Fradet. A systematic study of functional language implementations. *ACM Trans. on Prog. Lang. and Sys.*, 20(2), pp. 344-387, 1998.

[13] J. Fairbairn and S. Wray. Tim: a simple, lazy abstract machine to execute supercombinators. In *Proc of FPCA'87*, LNCS 274, pp. 34-45, 1987.

[14] P. Fradet and D. Le Métayer. Compilation of functional languages by program transformation. *ACM Trans. on Prog. Lang. and Sys.*, 13(1), pp. 21-51, 1991.

[15] C. Hall. Using Hindley-Milner type inference to optimise list representation. In *Proc. of LFP'94*, pp. 162-172,1994.

[16] J. Hannan and D. Miller. From operational semantics to abstract machines: Preliminary results. In *Proc. of LFP'90*, pp. 323-332, Nice, France, 1990.

[17] T. Hardin, L. Maranget and B. Pagano. Functional back-ends within the lambda-sigma calculus. In Proc. of *ICFP'1996*, pp. 25-33, 1996.

[18] T. Johnsson. *Compiling Lazy Functional Languages.* PhD Thesis, Chalmers University, 1987.

[19] M. S. Joy, V. J. Rayward-Smith and F. W. Burton. Efficient combinator code. *Computer Languages*, 10(3), 1985.

[20] J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science*. Vol. 2, pp. 2-108, Oxford University Press, 1992.

[21] J.-L. Krivine. Un interprète du lambda-calcul. Unpublished draft, available at `ftp://ftp.logique.jussieu.fr/pub/distrib/krivine/interprt.pdf`.

[22] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4), pp. 308-320, 1964.

[23] F. Lang, Z. Benaissa and P. Lescanne. Super-Closures. In *Proc. of WPAM'98*, as Technical Report of the University of SaarBruck, number A 02/98, 1998.

[24] X. Leroy. The Zinc experiment: an economical implementation of the ML language. *INRIA Technical Report 117*, 1990.

[25] X. Leroy. Unboxed objects and polymorphic typing. In *ACM Symp. on Princ. of Prog. Lang.*, pp. 177-188, 1992.

[26] H.R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.

[27] M. Parigot. λμ-Calculus: An algorithmic interpretation of classical natural deduction. In *Proc. of LPAR'92*, LNAI, Vol. 624, pp. 190-201, 1992.

[28] S.L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Func. Prog.*, 2(2):127-202, 1992.

[29] S. L. Peyton Jones and D. Lester. *Implementing functional languages, a tutorial*. Prentice Hall, 1992.

[30] S. L. Peyton Jones, W. Partain and A. Santos. Let-floating: moving bindings to give faster programs. In *Proc. of ICFP'96*, pp. 1-12, 1996.

[31] P. Sestoft. *Analysis and efficient implementation of functional programs*. PhD Thesis, DIKU, University of Copenhagen, 1991.

[32] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3), pp. 231-264, 1997.

[33] Z. Shao, J. Reppy and A. Appel. Unrolling lists. In *Proc. of LFP'94*, pp. 185-195,1994.

[34] T. Streicher and B. Reus. Classical Logic, Continuation semantics and abstract machines. *Journal of Functional Programming*, 8(6), pp. 543-572, 1998.

[35] D.A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9, pp. 31-49, 1979.