Functional Back-Ends within the Lambda-Sigma Calculus

Thérèse Hardin*†, Luc Maranget†and Bruno Pagano*†

Abstract

We define a weak λ-calculus, $\lambda\sigma_w$, as a subsystem of the full λ-calculus with explicit substitutions $\lambda\sigma_{\Uparrow}$. We claim that $\lambda\sigma_w$ could be the archetypal output language of functional compilers, just as the λ-calculus is their universal input language. Furthermore, $\lambda\sigma_{\Uparrow}$ could be the adequate theory to establish the correctness of simplified functional compilers. Here, we illustrate these claims by proving the correctness of two simplified compilers and runtime systems modeled as abstract machines. We first present the Krivine machine. Then, we give the first formal proofs of Cardelli's FAM and of its compiler.

1 Introduction

It is folklore to define a compiler as a translator from a high-level language intended for humans to a low-level language intended for machines. For mostly theoretical issues such as semantics or correctness of high-level program transformations, real programming languages are too complicated and lack generality. Instead, it is convenient to use an archetypal language, standing as a suitable abstraction of a whole class of programming languages. The λ-calculus is widely accepted as such a paradigm of all functional programming languages, due to its simplicity, consistency and generality. More precisely, the λ-calculus captures the essence of functionality. It is a non-ambiguous (i.e., Church-Rosser) reduction system where any strategy can be specified, yielding call-by-value or call-by-name functional languages. Moreover, it can be extended by adding extra rewriting rules to treat arithmetic or data structures [22].

In opposition to this commonly accepted view of λ-calculus as universal abstract syntax, there is no consensus among writers of functional compilers about the choice of an archetypal target language. With respect to the formal description of their output, published compilers for functional languages fall into three classes: they either compile to combinator or supercombinator [5] terms, to λ-terms in continuation passing style (CPS) [2], or to abstract machines [16, 7, 6, 8, 18]. These different approaches are praised for their peculiarities: combinators for their rewriting aspects and adequation to lazy evaluation [23], CPS for its ability to encode explicitly a given strategy and abstract machines for their closeness to real computers. None of these frameworks is designed to express the others. In fact, they do not claim to be universal, but each claims to be the best.

Nevertheless, a few common concepts arise here. The functions are to be compiled, that is, a fixed code should perform the actions specified in the body of a function, this code remaining unchanged at every invocation of the function. The variables in a function are of two kinds: either formal parameters or free variables. The values of parameters change at every function call, whereas the values of free variables remain the same. Thus, the low-level object that represents a function is a *closure*: a pair of a code and an *environment* that collects the values of the free variables at function creation time. Therefore, our universal target language should be a calculus of closures. Furthermore, the basic operations performed by the various existing runtime systems are the same: applying a closure, creating a closure, or retrieving the value of a variable in some environment. These operations are best unveiled in abstract machines. Thus, in the rest of this paper we focus mostly on them, as a still widely accepted formal description of functional runtime systems, which we intend to surpass.

Closures are naturally expressed in the λ-calculus with explicit substitutions [1, 10] as a term $(\lambda M)[s]$, where M is a term standing for a piece of code, and s is a substitution, that is, an environment, collecting the values of free variables. We now need some rules to compute on closures. First of all, we need to apply a closure $(\lambda M)[s]$ to an argument N, yielding the explicit application of the new substitution $t = N \cdot s$ to the body M, written $M[N \cdot s]$. Then, we have to propagate the substitution t inside the body M, until the substitution t reaches a variable, which should then be replaced by its value, or a λ abstraction, whose body is code for a new closure. The rule for applying closures along with simple rules to propagate substitutions define the weak $\lambda\sigma$-calculus, $\lambda\sigma_w$. While designing $\lambda\sigma_w$ as yet another calculus of explicit substitutions, we took particular care to select only the term constructs and rewriting rules that are actually required to express the basic steps performed by the existing abstract machines. We are satisfied that this pragmatic approach yields a confluent subcalculus of $\lambda\sigma_{\Uparrow}$, one of the $\lambda\sigma$-calculi introduced in [10].

In this paper, we first introduce the weak $\lambda\sigma$-calculus and give a unified presentation of abstract machines. Afterwards, we show that the basic operations of abstract machines correspond to certain rewriting steps in the weak $\lambda\sigma$-calculus. More precisely, the deterministic evaluation strategy implemented by an abstract machine is identified as a rewriting strategy in $\lambda\sigma_w$. We make this correspondence fully explicit for the Krivine machine and the FAM. The lat-

*IBP-LITP, Université Pierre et Marie Curie, 75252 Paris Cedex 05, France.
†INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex France.
—Email: {Therese.Hardin,Luc.Maranget,Bruno.Pagano}@inria.fr
—This work was partially supported by the ESPRIT Basic Research Project 6454 - CONFER.

terexample involves a true compilation of the input λ-term to a $\lambda\sigma$-term. Thereby, we give the first known proof of the correctness of a FAM-based compiler and runtime system. Other execution models are briefly discussed in section 5.

We thus illustrate our claim that weak λ-calculus with explicit substitutions is an adequate tool to study the execution of compiled functional programs [1, 10, 20]. Moreover, as shown by the FAM, the full λ-calculus with explicit substitutions may be a good formal language to describe the whole compilation process. This confirms the versatility of $\lambda\sigma$, which has been used recently to study advanced topics in the λ-calculus, such as higher order unification [12], or issues in logic, such as the interpretation of sequent calculus [15].

2 Preliminaries

2.1 The lambda-calculus with explicit substitutions

The traditional weak λ-calculus is an attempt to model the execution of machine code within the λ-calculus; it conforms with the basic intuition that functions, once compiled, are code and cannot change (otherwise, there would be no compilation). A tentative definition of weak reduction is thus to *suppress* the (ξ) rule from the definition of the λ-calculus.

$$\frac{M \to M'}{\lambda x.M \to \lambda x.M'}\ (\xi)$$

This negative definition has the major drawback that it does not lead to a consistent definition of the weak λ-calculus as a Church-Rosser rewriting system. To see this, consider the following derivations:

$$\begin{tikzcd}
& (\lambda xy.y\ x)\ ((\lambda z.z)\ (\lambda z.z)) \arrow[dl] \arrow[dr] & \\
\lambda y.y\ ((\lambda z.z)\ (\lambda z.z)) & & (\lambda xy.y\ x)\ (\lambda z.z) \arrow[dl] \\
& \lambda y.y\ (\lambda z.z) &
\end{tikzcd}$$

The problem lies in a discrepancy between intuition and formalism. Using ordinary λ-terms only, what is intuitively perceived as the invariable code $\lambda y.y\ x$ with respect to the possibly changing binding $[x\backslash(\lambda z.z)\ (\lambda z.z)]$ has to be represented by the fully substituted abstraction $\lambda y.y\ ((\lambda z.z)\ (\lambda z.z))$, so that the redex $(\lambda z.z)\ (\lambda z.z)$ is now located under a λ and cannot be contracted without invoking the (ξ) rule. This undesirable divergence can be corrected by delaying substitution, that is, by introducing explicit closures. Then we get:

$$\begin{tikzcd}
& (\lambda xy.y\ x)\ ((\lambda z.z)\ (\lambda z.z)) \arrow[dl] \arrow[dr] & \\
(\lambda y.y\ x)[x\backslash(\lambda z.z)\ (\lambda z.z)] \arrow[dr] & & (\lambda xy.y\ x)\ (\lambda z.z) \arrow[dl] \\
& (\lambda y.y\ x)[x\backslash\lambda z.z] &
\end{tikzcd}$$

Informally, "reduction is not allowed under λ's" is replaced by "substitution does not cross λ's".

A natural setting for a formal treatment of closures is $\lambda\sigma$, the λ-calculus with explicit substitutions [1, 10]. We first recall the definition of Λ_σ, the two sorted algebra of $\lambda\sigma$-terms.

$$\begin{array}{lrcl}
\text{TERMS:} & M & ::= & \mathtt{n} \mid (MM) \mid \lambda M \mid M[s] \\
\text{SUBSTITUTIONS:} & s & ::= & \mathrm{id} \mid \uparrow \mid M \cdot s \mid s \circ s
\end{array}$$

Note that variables are represented by De Bruijn indices. The new term construct $M[s]$ represents explicitly the application of substitution s to term M. The substitutions themselves are made explicit: we have two special substitutions id and $\uparrow$, whereas substitutions are structured as lists of terms $M \cdot s$ or as compositions $s \circ s$.

We note Λ_{DB} the subset of Λ_σ that coincides with ordinary λ-terms.

$$\lambda_{\mathrm{DB}}\text{-TERMS:} \quad M \quad ::= \quad \mathtt{n} \mid (MM) \mid \lambda M$$

The propagation of substitutions inside terms and substitutions is defined by the following rewriting system σ_w:

$$\begin{array}{lrcl}
(\mathrm{App}) & (M\ N)[s] & \to & (M[s])\ (N[s]) \\
(\mathrm{FVar}) & \mathtt{1}[M \cdot s] & \to & M \\
(\mathrm{RVar}) & (\mathtt{n+1})[M \cdot s] & \to & \mathtt{n}[s] \\
(\mathrm{Clos}) & (M[s])[t] & \to & M[s \circ t] \\
(\mathrm{AssEnv}) & (s \circ t) \circ u & \to & s \circ (t \circ u) \\
(\mathrm{MapEnv}) & (M \cdot s) \circ t & \to & M[t] \cdot (s \circ t) \\
(\mathrm{ShiftCons}) & \uparrow \circ (M \cdot s) & \to & s \\
(\mathrm{IdL}) & \mathrm{id} \circ s & \to & s
\end{array}$$

There is one rule per term construct in the algebra of $\lambda\sigma$-terms, except for λ. The propagation of substitutions through application nodes and accesses inside list-structured substitutions are handled by the first three rules above in a straightforward manner. The case of functions is more subtle. In the simplest case of the so-called "shared environment machines" (Krivine machine or SECD, for instance), functions are compiled as abstractions — i.e., as code —, and execution will only pair these abstractions with an environment, producing closures. The "copied environment machines", such as the FAM, are more sophisticated: a function λM is compiled into a closure $(\lambda M')[s]$, where s collects the references of M to a global environment, whereas the free variables in $\lambda M'$ refer only to s. At run-time, applying some substitution t to the closure $(\lambda M')[s]$ will result in applying t to s, thus composing the two substitutions into one new substitution $s \circ t$ — as illustrated by the rewriting rule (Clos). Hence, we need rules to substitute inside substitutions; in other words, rules to compose substitutions. These rules are the remaining four rules above. Here again, there is one rule per term construct in the sort of substitutions.

As we need the composition operator on substitutions "$\circ$" to express certain computations, we differ significantly from recent calculus of explicit substitutions without composition [19].

Since there is no rule for crossing λ's, a $\lambda\sigma$-closure, i.e. a term of the form $(\lambda M)[s]$, cannot be reduced at its root. Hence, $\lambda\sigma$-closures are (weak) *values*, similar to weak head normal forms. In our case of an archetypal target language, $\lambda\sigma$-closures are the only values. In a more general setting that would also consider arithmetic and data structures, additional values would be integers, lists,...

In the weak setting, $\lambda\sigma$-closures are destructured only by applying them to arguments:

$$\text{(Beta)} \quad ((\lambda M)[s])\, N \quad \rightarrow \quad M[N \cdot s]$$

The system $\lambda\sigma_w$ is defined by the rules of σ_w plus the rule (Beta). From [10], where a system very close to $\lambda\sigma_w$ is studied, we deduce that the weak substitution system σ_w is both strongly normalizing and confluent and, by using the interpretation method, that the reduction system $\lambda\sigma_w$ is confluent. Moreover, our system $\lambda\sigma_w$ is a subcalculus of $\lambda\sigma_{\Uparrow}$, the most general λ-calculus with explicit substitutions.

The terms of $\lambda\sigma_{\Uparrow}$ are the terms of $\lambda\sigma$ plus the additional "lifted" substitution construct $\Uparrow(s)$, whereas the rules of the strong substitution system $\sigma_{\Uparrow}$ are the rule of σ_w plus the following strong substitution rules:

(Lambda)	$(\lambda M)[s]$	$\rightarrow$	$\lambda(M[\Uparrow(s)])$
(VarShift1)	$\mathtt{n}[\uparrow]$	$\rightarrow$	$\mathtt{n+1}$
(VarShift2)	$\mathtt{n}[\uparrow \circ s]$	$\rightarrow$	$\mathtt{n+1}[s]$
(FVarLift1)	$\mathtt{1}[\Uparrow(s)]$	$\rightarrow$	$\mathtt{1}$
(FVarLift2)	$\mathtt{1}[\Uparrow(s) \circ t]$	$\rightarrow$	$\mathtt{1}[t]$
(RVarLift1)	$\mathtt{n+1}[\Uparrow(s)]$	$\rightarrow$	$\mathtt{n}[s \circ \uparrow]$
(RVarLift2)	$\mathtt{n+1}[\Uparrow(s) \circ t]$	$\rightarrow$	$\mathtt{n}[s \circ (\uparrow \circ t)]$
(ShiftLift1)	$\uparrow \circ \Uparrow(s)$	$\rightarrow$	$s \circ \uparrow$
(ShiftLift2)	$\uparrow \circ (\Uparrow(s) \circ t)$	$\rightarrow$	$s \circ (\uparrow \circ t)$
(Lift1)	$\Uparrow(s) \circ \Uparrow(t)$	$\rightarrow$	$\Uparrow(s \circ t)$
(Lift2)	$\Uparrow(s) \circ (\Uparrow(t) \circ u)$	$\rightarrow$	$\Uparrow(s \circ t) \circ u$
(LiftEnv)	$\Uparrow(s) \circ (M \cdot t)$	$\rightarrow$	$M \cdot (s \circ t)$
(LiftId)	$\Uparrow(\mathrm{id})$	$\rightarrow$	id
(IdR)	$s \circ \mathrm{id}$	$\rightarrow$	s
(Id)	$M[\mathrm{id}]$	$\rightarrow$	M

With respect to strong substitution, $\lambda\sigma$-closures are not values any more, since any $\lambda\sigma$-closure $(\lambda M)[s]$ now immediately reduces to $\lambda(M[\Uparrow(s)])$, by the rule (Lambda). Most of the other rules of $\sigma_{\Uparrow}$ explicit the De Bruijn indices adjustements. The remaining two rules —(IdR) and (Id)— define the substitution id as the identity. As it can be expected, $\sigma_{\Uparrow}$ is a terminating and confluent rewriting system [10].

The full system $\lambda\sigma_{\Uparrow}$ is defined by the strong substitution rules of $\sigma_{\Uparrow}$ plus the following rule for applying λ-abstractions to their arguments:

$$\text{(BetaStrong)} \quad (\lambda M)\, N \quad \rightarrow \quad M[N \cdot \mathrm{id}]$$

The system $\lambda\sigma_{\Uparrow}$ is both confluent and correct with respect to the λ-calculus [10].

One easily sees that $\lambda\sigma_w$ is a subcalculus of $\lambda\sigma_{\Uparrow}$, since the weak β-rule (Beta) is a shortcut for the following $\lambda\sigma_{\Uparrow}$-derivation:

$$((\lambda M)[s])\, N \overset{\text{(Lambda)}}{\longrightarrow} (\lambda(M[\Uparrow(s)]))\, N \overset{\text{(BetaStrong)}}{\longrightarrow}$$
$$(M[\Uparrow(s)])[N \cdot \mathrm{id}] \overset{\text{(Clos)}}{\longrightarrow} M[\Uparrow(s) \circ (N \cdot \mathrm{id})] \overset{\text{(LiftEnv)}}{\longrightarrow}$$
$$M[N \cdot (s \circ \mathrm{id})] \overset{\text{(IdR)}}{\longrightarrow} M[N \cdot s]$$

Thus, as a subsystem of the strong system $\lambda\sigma_{\Uparrow}$, the weak system $\lambda\sigma_w$ is also correct with respect to the λ-calculus.

2.2 Abstract machines

In the rest of this paper, we describe two machines, while unifying their presentations.

Instructions differ between machines, but, for any machine, the code always is a possibly empty list of instructions:

$$\text{CODE} \;::=\; () \quad | \quad \text{INSTRUCTION}; \text{CODE}$$

A closure is a piece of code associated with an environment, an environment being a list of closures:

$$\text{CLOSURE} \;::=\; (\text{CODE}/\text{ENVIRONMENT})$$
$$\text{ENVIRONMENT} \;::=\; () \quad | \quad \text{CLOSURE} \cdot \text{ENVIRONMENT}$$

A typical code will be written C, a typical environment e and a typical closure f or (C/e).

The states of the machines are non-empty lists of frames, the exact structure of frames depending upon the machines.

$$\text{STATE} \;::=\; \text{FRAME} \quad | \quad \text{FRAME} :: \text{STATE}$$

The behavior of an abstract machine is specified by a deterministic transition system. Briefly, a transition system is a triple $(E, E_t, \rightarrow)$, where E is a set of states, E_t is the subset of terminal states and $\rightarrow$ is the transition relation defined over $(E - E_t) \times E$. The transitive closure of $\rightarrow$ is written $\rightarrow^*$. A transition system is deterministic when, for every state D, there exists at most one state D' such that $D \rightarrow D'$.

At this point, the puzzled reader may have a look at the machine descriptions of sections 3 and 4, keeping in mind the following notations:

Convention: *We will make intensive use of lists. When appropriate, we freely interpret them as sequences, words or even arrays. That is, given n elements $x_1, x_2, \ldots, x_n$, the list $S = x_1 : x_2 : \ldots : x_n : ()$ is simply written $x_1 : x_2 : \ldots : x_n$. A sublist $x_i : \ldots : x_j$ is written $\overrightarrow{x_{i,j}}$ and with a slight notational abuse we sometimes decompose S as follows: $S = \overrightarrow{x_{1,i-1}} : \overrightarrow{x_{i,j}} : \overrightarrow{x_{j+1,n}}$*

2.3 Implementation of a strategy by a machine

In [24], bisimulations are used to establish the equivalence of two transition systems Σ_1 and Σ_2. In our work, Σ_1 always defines an abstract machine whereas Σ_2 is a subsystem of $\lambda\sigma_w$. The states of Σ_2 are $\lambda\sigma$-terms and its transition relation is a rewriting strategy $\overset{\mathrm{S}}{\longrightarrow}$, a strategy being a deterministic subrelation of the general $\lambda\sigma_w$-reduction relation.

We present a simplified setting, where a bisimulation is given by two partial functions, the compile-and-load function $\mathcal{L}$ that translates $\lambda\sigma$-terms into machine states, and the decompilation function $\overline{\cdot}$ that translates machine states into $\lambda\sigma$-terms. We say that a machine implements a strategy S when the following three conditions are satisfied:

1. *Initial condition:* Let M be a $\lambda\sigma$-term such that $\mathcal{L}(M)$ exists. Then $\overline{\mathcal{L}(M)} = M[\mathrm{id}]$.
2. *The machine follows the strategy S:* If $D_1 \rightarrow D_2$ and $\overline{D_1}$ exists, then $\overline{D_2}$ exists and we have either $\overline{D_2} = \overline{D_1}$ or $\overline{D_1} \overset{\mathrm{S}}{\longrightarrow} \overline{D_2}$.

3. *Terminal states translate to normal forms:* Let M be a $\lambda\sigma$-term such that $\mathcal{L}(M)$ exists. If $\mathcal{L}(M) \to^* D$ and D is a terminal state, then $\overline{D}$ is a S-normal form.

The condition 1 is justified by the rule (Id): $M\,[\mathrm{id}] \to M$ of $\lambda\sigma_{\Uparrow}$. Basically, the rule (Id) states that "id" is the identity substitution that maps variables to themselves. This point is important, since it is a first illustration of using the strong $\lambda\sigma$-calculus to assert a correctness property.

3 The Krivine Machine

First, we describe the Krivine Machine [8]. This machine is very simple:

$$\begin{array}{rcl} \textsc{instruction} & ::= & \mathsf{Grab} \mid \mathsf{Push}(\textsc{code}) \mid \mathsf{Access}(n) \\ \textsc{frame} & ::= & \textsc{closure} \end{array}$$

A typical state D is thus a stack of closures, which we write $f_1 :: f_2 :: \ldots :: f_n$.

A λ_{DB}-term is compiled as follows:

$$\begin{array}{rcl} [\![\, \mathtt{n} \,]\!] & = & \mathsf{Access}(n) \\ [\![\, \lambda M \,]\!] & = & \mathsf{Grab}; [\![\, M \,]\!] \\ [\![\, (M\ N) \,]\!] & = & \mathsf{Push}([\![\, N \,]\!]); [\![\, M \,]\!] \end{array}$$

Loading of compiled code is just pairing with the empty environment: $\mathcal{L}(M) = ([\![\, M \,]\!]/())$.

The execution of programs is defined by the following transition rules:

$$\begin{array}{rcl} (\mathsf{Access}(1)/(C_0/e_0)\cdot e) :: D & \stackrel{\textbf{lvar}}{\longrightarrow} & (C_0/e_0) :: D \\ (\mathsf{Access}(n+1)/(C_0/e_0)\cdot e) :: D & \stackrel{\textbf{rvar}}{\longrightarrow} & (\mathsf{Access}(n)/e) :: D \\ (\mathsf{Push}(C'); C/e) :: D & \stackrel{\textbf{push}}{\longrightarrow} & (C/e) :: (C'/e) :: D \\ (\mathsf{Grab}; C/e) :: (C'/e') :: D & \stackrel{\textbf{grab}}{\longrightarrow} & (C/(C'/e')\cdot e) :: D \end{array}$$

Then, we define the decompilation procedure from machine states to $\lambda\sigma$-terms. First, we just reverse the compilation scheme $[\![\ ]\!]$, and extend the resulting decompilation procedure to closures and environments:

$$\begin{array}{rcl} \overline{\mathsf{Access}(n)} & = & \mathtt{n} \\ \overline{\mathsf{Grab}; C} & = & \lambda\, \overline{C} \\ \overline{\mathsf{Push}(C'); C} & = & (\overline{C}\ \overline{C'}) \\ & & \\ \overline{(C/e)} & = & \overline{C}\,[\overline{e}] \\ & & \\ \overline{()} & = & \mathrm{id} \\ \overline{f\cdot e} & = & \overline{f}\cdot\overline{e} \end{array}$$

Finally, observing that new frames are introduced by the execution of the Push instruction, which is the code equivalent of an application, the state constructor :: is decompiled as an application:

$$\overline{f_1 :: f_2 :: \ldots :: f_n} = (\ldots(\overline{f_1}\ \overline{f_2})\ldots\ \overline{f_n})$$

In the expression above, the $\lambda\sigma$-term $\overline{f_1}$ is said to be in head position. The head position is the leftmost position with respect to application nodes, since $\overline{f_1} = \overline{C_1}\,[\overline{e_1}]$ is not an application.

Now, we show several properties of the compilation and decompilation functions. First, these two transformations are another inverse:

Lemma 1 *$\overline{[\![\, M \,]\!]} = M$, for any λ_{DB}-term M.*

As a corollary, we get the initial condition 1. Then, we show a weakened condition 2, in order to make the strategy of the Krivine machine appear naturally.

Lemma 2 *Let D and D' be two machine states such that $\overline{D}$ is defined and $D \to D'$. Then $\overline{D'}$ exists and we have the reduction $\overline{D} \xrightarrow{\lambda\sigma_w *} \overline{D'}$.*

Proof: We give the example of a **push** transition:

$$\begin{tikzcd} (\mathsf{Push}(C'); C/e) :: D_0 \arrow[r, "\text{push}"] \arrow[d, "\overline{\cdot}"'] & (C/e) :: (C'/e) :: D_0 \arrow[d, "\overline{\cdot}"'] \\ (\ldots(\overline{C}\ \overline{C'})\,[\overline{e}]\ \ldots \overline{f_n}) \arrow[r, "\lambda\sigma_w"] & (\ldots(\overline{C}\,[\overline{e}])\ (\overline{C'}\,[\overline{e}])\ \ldots \overline{f_n}) \end{tikzcd}$$

Therefore, the execution of the instruction Push is equivalent to the application of the $\lambda\sigma_w$-reduction rule (App). Similarly, the transitions **lvar**, **rvar**, and **grab** implement the reduction rules (FVar), (RVar) and (Beta). Finally, all reductions are performed in head position. □

By the detailed proof of the previous lemma, it is not difficult to see that the Krivine machine follows the weak leftmost strategy, or K-strategy, described below in the small step formalism:

$$\mathtt{1}\,[M\cdot s] \xrightarrow{\mathrm{K}} M \qquad (\mathtt{n{+}1})\,[M\cdot s] \xrightarrow{\mathrm{K}} \mathtt{n}\,[s]$$

$$(M\ N)\,[s] \xrightarrow{\mathrm{K}} (M\,[s])\ (N\,[s]) \quad ((\lambda M)\,[s])\ N \xrightarrow{\mathrm{K}} M\,[N\cdot s]$$

$$\frac{M \xrightarrow{\mathrm{K}} M'}{M\ N \xrightarrow{\mathrm{K}} M'\ N}$$

It remains to show condition 3 on terminal states.

Lemma 3 *Let D be a reachable terminal state, then $\overline{D}$ is a K-normal form:*

Proof: The Krivine machine may stop for two reasons:

- When $D = (\mathsf{Grab}; C/e)$. Then, we get $\overline{D} = (\lambda\overline{C})\,[\overline{e}]$, which is a $\lambda\sigma$-closure and a K-normal form.
- When $D = (\mathsf{Access}(m)/()) :: D_0$, i.e, when an access fails. Then, $\overline{D} = (\ldots(\mathtt{m}\ \overline{f_2})\ldots\ \overline{f_n})$, which is also a K-normal form. This case cannot happen if the initial program is a closed λ_{DB}-term. □

Finally, as a corollary of the previous three lemmas, we get the final result:

Proposition 1 *The Krivine machine implements the K-strategy.*

4 The Functional Abstract Machine

4.1 Basics

The Functional Abstract Machine (FAM) was designed by Cardelli [6] as a "SECD machine optimized to allow very fast function application and the use of true stack".

$$
\begin{array}{rcl}
(AS : f, e, \mathsf{Local}; C) :: D & \xrightarrow{\textbf{local}} & (f : AS : f, e, C) :: D \\
(AS, \overrightarrow{f_{1,n}}, \mathsf{Global}(i); C) :: D & \xrightarrow{\textbf{global}} & (f_i : AS, \overrightarrow{f_{1,n}}, C) :: D \\
((C_0/e_0) : g : AS, e, \mathsf{Apply}; C) :: D & \xrightarrow{\textbf{apply}} & (g, e_0, C_0) :: (AS, e, C) :: D \\
(\overrightarrow{f_{n,1}} : AS, e, \mathsf{Fun}(n, C_0); C) :: D & \xrightarrow{\textbf{closure}} & ((C_0/\overrightarrow{f_{1,n}}) : AS, e, C) :: D \\
(f : \_, \_, ()) :: (AS, e, C) :: D & \xrightarrow{\textbf{return}} & (f : AS, e, C) :: D
\end{array}
$$

Figure 1: Transition rules for the FAM

The FAM has four instructions:

$$
\begin{array}{lcll}
\text{INSTRUCTION} & ::= & \mathsf{Local} & \\
 & | & \mathsf{Global}(n) & \\
 & | & \mathsf{Apply} & \\
 & | & \mathsf{Fun}(m, \text{CODE}) & (n \geq 1, m \geq 0)
\end{array}
$$

The frames of the FAM consist in an argument stack, an environment and a code.

$$
\begin{array}{lcl}
\text{STACK} & ::= & () \quad | \quad \text{CLOSURE} : \text{STACK} \\
\text{FRAME} & ::= & (\text{STACK}, \text{ENVIRONMENT}, \text{CODE})
\end{array}
$$

The transitions of the FAM are given in figure 1. Specifically, observe how the instruction Local selects the bottom element f of the argument stack $(AS : f)$ and how the n arguments $\overrightarrow{f_{n,1}}$ that the instruction $\mathsf{Fun}(n, C_0)$ pops are taken in reverse order to build a new environment $\overrightarrow{f_{1,n}}$. By contrast with the Krivine machine, the FAM builds a full environment when it creates a closure. One says that the FAM has "copied environments" (whereas the Krivine machine has "shared environments").

In [6], Cardelli only gives a few "compilation hints" for the FAM. One of these hints consists in compiling a function λM as a closure (C/e), where the environment e has been optimized to retain only the global variables of λM. This idea is described as a simple transformation in $\lambda\sigma$. For instance the λ-abstraction $M = \lambda(\mathtt{1}\ (\mathtt{5}\ \mathtt{7}))$ is transformed into the $\lambda\sigma$-closure $M' = (\lambda(\mathtt{1}\ (\mathtt{2}\ \mathtt{3})))\,[\mathtt{4} \cdot \mathtt{6} \cdot \mathrm{id}]$. That is, the free variables $\mathtt{5}$ and $\mathtt{7}$ are "abstracted out" or "lifted" and regrouped in the closure environment $\mathtt{4} \cdot \mathtt{6} \cdot \mathrm{id}$, whereas, in the body of M, the lifted variables $\mathtt{5}$ and $\mathtt{7}$ are replaced by $\mathtt{2}$ and $\mathtt{3}$ respectively, the new indices reflecting the final positions of lifted variables in the closure environment. As a first intuition of the correctness of such a transformation, observe that the terms $M\,[\mathrm{id}]$ and M' are the same function. For any argument N, we get:

$$
\begin{array}{l}
(M\,[\mathrm{id}])\ N \stackrel{(\mathrm{Beta})}{\longrightarrow} (\mathtt{1}\ (\mathtt{5}\ \mathtt{7}))\,[N \cdot \mathrm{id}] \xrightarrow{\sigma}{}^{*} N\ (\mathtt{4}\ \mathtt{6}) \\
M'\ N \stackrel{(\mathrm{Beta})}{\longrightarrow} (\mathtt{1}\ (\mathtt{2}\ \mathtt{3}))\,[N \cdot \mathtt{4} \cdot \mathtt{6} \cdot \mathrm{id}] \xrightarrow{\sigma}{}^{*} N\ (\mathtt{4}\ \mathtt{6})
\end{array}
$$

We now describe our general free variable abstraction procedure. Formally, the set of free variables in a λ_{DB}-term M are collected by calculating $\mathcal{F}_0(M)$, where $\mathcal{F}$ is defined by:

$$
\begin{array}{rcll}
\mathcal{F}_d(\mathtt{n}) & = & \emptyset & \text{if } n \leq d \\
\mathcal{F}_d(\mathtt{n}) & = & \{n - d\} & \text{if } n > d \\
\mathcal{F}_d(M\ N) & = & \mathcal{F}_d(M) \cup \mathcal{F}_d(N) & \\
\mathcal{F}_d(\lambda M) & = & \mathcal{F}_{d+1}(M) &
\end{array}
$$

The set $\mathcal{F}_0(M) = \{n_1, \ldots, n_m\}$ can now be arbitrarily ordered as a list $\overrightarrow{n_{1,m}} = n_1 : \ldots : n_m$. This list is then given as a first argument to our abstraction scheme $\mathcal{C}$, which, given any λ_{DB}-term M, outputs a $\lambda\sigma$-term $\mathcal{C}(\mathcal{F}_0(M)\,, M)$:

$$
\begin{array}{rcl}
\mathcal{C}(\overrightarrow{n_{1,m}}\,, \mathtt{n_i}) & = & \mathtt{i} \\
\mathcal{C}(\overrightarrow{n_{1,m}}\,, (M\ N)) & = & (\mathcal{C}(\overrightarrow{n_{1,m}}\,, M)\ \mathcal{C}(\overrightarrow{n_{1,m}}\,, N)) \\
\mathcal{C}(\overrightarrow{n_{1,m}}\,, \lambda M) & = & \\
\multicolumn{3}{l}{\left\{ \begin{array}{l} (\lambda\mathcal{C}(1 : p_1{+}1 : \cdots p_k{+}1\,, M))\,[\mathcal{C}(\overrightarrow{n_{1,m}}\,, \mathtt{p_1} \cdots \mathtt{p_k} \cdot \mathrm{id})] \\ \text{where } p_1 : p_2 : \cdots : p_k = \mathcal{F}_0(\lambda M) \end{array} \right.} \\
\mathcal{C}(\overrightarrow{n_{1,m}}\,, M \cdot s) & = & \mathcal{C}(\overrightarrow{n_{1,m}}\,, M) \cdot \mathcal{C}(\overrightarrow{n_{1,m}}\,, s) \\
\mathcal{C}(\overrightarrow{n_{1,m}}\,, \mathrm{id}) & = & \uparrow^m
\end{array}
$$

To get the intuition behind the scheme $\mathcal{C}$, think that the output $\lambda\sigma$-term is to be executed at run-time in an environment $s = N_1 \cdots N_i \cdots N_m \cdot \mathrm{id}$, where N_i is the run-time value of the variable $\mathtt{n_i}$ of the input term.

The translation $(\lambda M')\,[t]$ of a λ-abstraction λM is also to be interpreted in this environment s. At run-time the current substitution s will be applied to t, in order to yield a new current substitution that only retains the values of the free variables of λM. Thus, t is the list of the positions in s of these free variables. The substitution t ends with the new special substitution $\uparrow^m$, which ultimately discards s. This discarding operator is an ordinary $\lambda\sigma$-substitution:

$$
\uparrow^0 = \mathrm{id}, \quad \uparrow^1 = \uparrow, \quad \uparrow^m = \uparrow \circ \uparrow^{m-1} \text{ when } m > 1
$$

The action of $\uparrow^m$ is then expressed by the following σ_w-derivation:

$$
\begin{array}{rcl}
\uparrow^m \circ (N_1 \cdot N_2 \cdots N_m \cdot \mathrm{id}) & \stackrel{(\mathrm{AssEnv})}{\longrightarrow} & \\
\uparrow \circ (\uparrow^{m-1} \circ (N_1 \cdot N_2 \cdots N_m \cdot \mathrm{id})) & \xrightarrow{\sigma_w}{}^{*} & \ldots \\
\uparrow \circ (N_m \cdot \mathrm{id}) & \stackrel{(\mathrm{ShiftCons})}{\longrightarrow} & \mathrm{id}
\end{array}
$$

The correctness of the procedure $\mathcal{C}$ is stated with respect to the strong substitution rules of $\sigma_{\Uparrow}$.

Lemma 4 *The $\mathcal{C}$ compilation scheme never fails and is correct. More precisely, given a closed λ_{DB}-term M, then $\mathcal{C}(\emptyset\,, M)$ is a $\lambda\sigma$-term M' such that $M'\,[id]$ and M are $\sigma_{\Uparrow}$-equal.*

Proof: We show the following property by induction on M: given a λ_{DB}-term M, a substitution s and a vector $\overrightarrow{n_{1,m}}$ such that $\mathcal{F}_0(M) \subseteq \overrightarrow{n_{1,m}}$, $\mathcal{C}(\overrightarrow{n_{1,m}}\,, M)$ exists and we have:

$$
\mathcal{C}(\overrightarrow{n_{1,m}}\,, M)\,[\mathtt{n_1} \cdot \mathtt{n_2} \cdots \mathtt{n_m} \cdot s] =_{\sigma_{\Uparrow}} M
$$

□

It is important to notice that the $\lambda\sigma$-terms M and $M'\,[\mathrm{id}]$ only differ by *substitutions steps.* As a consequence, $M'\,[\mathrm{id}]$

and M' are more than just β-equivalent λ-terms, they are the same λ-term.

The second phase $[\![\ ]\!]$ of our compilation scheme applies to $\lambda\sigma$-terms of the kind produced by the first phase $\mathcal{C}$ and produces FAM code:

$$\begin{array}{rcl} [\![\, \mathtt{1} \,]\!] & = & \mathsf{Local} \\ [\![\, \mathtt{n+1} \,]\!] & = & \mathsf{Global}(n) \\ [\![\, (M\ N) \,]\!] & = & [\![\, N \,]\!]; [\![\, M \,]\!]; \mathsf{Apply} \\ [\![\, (\lambda M)\,[N_1 \cdots N_n \cdot \uparrow^m] \,]\!] & = & [\![\, N_1 \,]\!]; \ldots; [\![\, N_n \,]\!]; \mathsf{Fun}(n, [\![\, M \,]\!]) \end{array}$$

Finally, a closed λ_{DB}-term M is compiled to the code $C = [\![\, \mathcal{C}(\emptyset, M) \,]\!]$ and its execution starts from the initial state $\mathcal{L}(M) = ((), (), C)$.

4.2 The decompilation

First, we inverse the compilation procedure $[\![\ ]\!]$. We do so by proving judgements $C \Downarrow^k M$, which read "the code segment C stands for the $\lambda\sigma$-term M in an environment of size k".

$$\mathsf{Local} \Downarrow^k \mathtt{1}\ \text{(Local)} \qquad \mathsf{Global}(i) \Downarrow^k \mathtt{i{+}1}\ \text{(Global)}$$

$$\frac{C \Downarrow^k N \quad C' \Downarrow^k M}{C; C'; \mathsf{Apply} \Downarrow^k (M\ N)}\ \text{(Apply)}$$

$$\frac{C_1 \Downarrow^k N_1 \quad \cdots \quad C_i \Downarrow^k N_i \quad C \Downarrow^{i+1} M}{C_1; \cdots; C_i; \mathsf{Fun}(i, C) \Downarrow^k (\lambda M)\,[N_1 \cdots N_i \cdot \uparrow^k]}\ (\mathrm{Fun}(i))$$

The decompilation of FAM closures, environments and stacks naturally follows from code decompilation.

$$\begin{array}{rcl} \overline{(C/f_1 \cdot f_2 \cdots f_n)} & = & (\lambda M)\,[\overline{f_1 \cdot f_2 \cdots f_n}],\ \text{where } C \Downarrow^{n+1} M \\ \overline{f_1 \cdot f_2 \cdots f_n} & = & \overline{f_1} \cdot \overline{f_2} \cdots \overline{f_n} \cdot \mathrm{id} \\ \overline{f_n : \ldots : f_2 : f_1} & = & \overline{f_n} : \ldots : \overline{f_2} : \overline{f_1} \end{array}$$

The decompilation of machine states is best understood as a two-stage process. In a first step, we translate the frames (AS, e, C) into triples (S, s, C), where the stack S is the translation of the argument stack AS and the substitution s is the translation of the environment e, up to the bottom element of AS, whose translation may be incorporated either in S or in s.

$$\begin{array}{c} \Phi((AS_n : f_n, e_n, C_n) :: \ldots :: (AS_2 : f_2, e_2, C_2) :: (AS_1, e_1, C_1)) \\ \parallel \\ (\overline{AS_n}, \overline{f_n} \cdot \overline{e_n}, C_n) :: \ldots :: (\overline{AS_2}, \overline{f_2} \cdot \overline{e_2}, C_2) :: (\overline{AS_1}, \overline{e_1}, C_1) \end{array}$$

Then, the value of $\overline{D}$ is computed by proving a judgement $\Phi(D) \Downarrow \overline{D}$, with the axioms and inference rules given in figure 2. At first, our state decompilation procedure may seem a bit complicated. However, it is a simple extension of the code decompilation procedure. The basic idea is as follows: in a triple $D = (S, s, C)$, the stack S is the list of the subterms of $\overline{D}$ that have already been computed by the machine, whereas the code segment C encodes the part of $\overline{D}$ whose computation is still pending, the external references in C being relative to the current substitution s.

To see this, first consider the extreme cases, where everything has already been computed (rule (Res)), or where nothing has been computed yet (rule (Code)). Then, consider a mixed case, supposing, for instance, that $\overline{D}$ is an application $(M_1\ M_2)$ (i.e., assuming that the last instruction of C is Apply). If M_2 is a reduced term, then it stands at the bottom of the stack S, while the term M_1 is the decompilation of a substate of D (rule (AppLeft)). If M_2 is not fully reduced yet, then it is the decompilation of a substate, whereas M_1 is $M\,[s]$, where M is the decompilation of a code segment whose execution has not even started (rule (AppRight)).

The decompilation of a triple $(S,\ s,\ C'; \mathsf{Fun}(n, C))$ is performed by the rule $(\mathrm{Fun}(n,i))$. This operation resembles the decompilation of a triple $(S,\ s,\ C'; \mathsf{Apply})$. More specifically, the $\lambda\sigma$-substitution $N_1 \cdots N_i \cdot ((N_{i+1} \cdots N_n \cdot \uparrow^m) \circ s)$ stands for a closure environement that is not fully computed yet, where the term N_i is the subterm being currently computed, while the subterms $N_1, \ldots,\ N_{i-1}$ are fully reduced and the computation of the subterms $N_{i+1}, \ldots,\ N_n$ is yet to be started. Finally, the rule (State), which performs the decompilation of multi-frame states, is inspired by the FAM transition **return**.

We now prove that the rules of figure 2 effectively define a deterministic procedure for decompiling FAM states. First, we prove a strong non-ambiguity property for code segments.

Lemma 5 *A decompilable code is any code segment C, such that there exists a integer n and a $\lambda\sigma$-term M, with $C \Downarrow^n M$. A strict suffix is any code segement C, such that there exists a non-empty code segment C' and that the concatenation $C'; C$ is a decompilable code.*

1. *In a proof tree, all the code segments that appear in judgments $S, s, C \Downarrow M$, where S is a non-empty stack, are strict suffixes.*
2. *Strict suffixes cannot be decompiled.*

Proof: The first proposition is proved by induction on proof trees, starting from the fact that the empty code is a strict suffix. The second proposition is proved by induction on the length of decompilable codes. □

Lemma 6 *The decompilation of machines states is non-ambiguous.*

Proof: The proof is by induction on state size. Consider any state D, if D is made of two or more frames, the deterministic decompilation rule (State) applies. Now, suppose that D is a single frame, i.e., $\Phi(D) = (S, s, C)$.

If the code C is empty or ends by a Local or $\mathsf{Global}(i)$ instruction, only one non-recursive rule may apply and decompilation is over.

Otherwise, we have two subcases, either C ends by an Apply or by a $\mathsf{Fun}(n, C_0)$ instruction. As far as ambiguity is concerned, these subcases are the same. Thus, for instance, we assume $\Phi(D) = (S, s, C'; \mathsf{Fun}(n, C_0))$. If the stack S is empty, then only the rule (Code) may apply. If S contains at least one elememt, then we must apply the decompilation rule $(\mathrm{Fun}(n,i))$. By the previous lemma 5-2, there is at most one way to cut C' into $C_i; C_{i+1}; \cdots; C_n$, where C_i is a strict suffix and $C_{i+1}, \ldots,\ C_n$ are decompilable code segments. In other words, the rule $(\mathrm{Fun}(n,i))$ is applied unambiguously. □

$$M, s, () \Downarrow M \ \text{(Res)} \qquad\qquad \frac{C \Downarrow^m M}{(), s, C \Downarrow M\,[s]}\ \text{(Code)}$$

$$\frac{S, s, C \Downarrow N \quad C' \Downarrow^m M}{S,\ s,\ C; C'; \mathsf{Apply} \Downarrow (M\,[s]\ N)}\ \text{(AppRight)} \qquad \frac{S, s, C \Downarrow M}{S{:}N,\ s,\ C; \mathsf{Apply} \Downarrow (M\ N)}\ \text{(AppLeft)}$$

$$\frac{S, s, C_i \Downarrow N_i \quad C_{i+1} \Downarrow^m N_{i+1} \quad \cdots \quad C_n \Downarrow^m N_n \quad C \Downarrow^{n+1} M, \quad \text{for } 1 \le i \le n}{S{:}N_{i-1}{:}\ldots{:}N_1,\ s,\ C_i; C_{i+1}; \ldots; C_n; \mathsf{Fun}(n, C) \Downarrow (\lambda M)\,[N_1 \cdots N_i \cdot ((N_{i+1} \cdots N_n \cdot {\uparrow}^m) \circ s)]}\ (\text{Fun}(n,i))$$

$$\frac{S_n, s_n, C_n \Downarrow M \quad (M : S_{n-1}, s_{n-1}, C_{n-1}) :: \ldots :: (S_1, s_1, C_1) \Downarrow N}{(S_n, s_n, C_n) :: (S_{n-1}, s_{n-1}, C_{n-1}) :: \ldots :: (S_1, s_1, C_1) \Downarrow N}\ \text{(State)}$$

(Where S is a non-empty stack and m is the length of the substitution s)

Figure 2: The decompilation of machine states

4.3 Strategy and correctness

This section is devoted to the C-strategy that the FAM implements. This strategy is described in the small step formalism (see figure 3).

The axioms of the C-strategy are deduced from the FAM transition rules. The most significant axiom is (Beta), which we deduce from a simplified transition **apply**.

$$\begin{tikzcd}
((C_0/e_0) : g, e, \mathsf{Apply}) \arrow[r, "\text{apply}"] \arrow[d, dashed, "\overline{\ \ }"'] & (g, e_0, C_0) :: ((), e, ()) \arrow[d, dashed, "\overline{\ \ }"'] \\
(((\lambda \overline{C_0})\,[\overline{e_0}])\ \overline{g}) \arrow[r] & \overline{C_0}\,[\overline{g} \cdot \overline{e_0}]
\end{tikzcd}$$

Thus, we know that both function and argument are FAM closures when a function call occurs. Moreover, given a closure f, the $\lambda\sigma$-term $\overline{f}$ is both a $\lambda\sigma$-closure and a normal form. Such a $\lambda\sigma$-term is called a C-value.

C-VALUES:	V	$::=$	$(\lambda M)\,[e]$
C-ENVIRONMENTS:	e	$::=$	id | $V \cdot e$

Hence, we get the rule (Beta) of figure 3, which characterizes the C-strategy as call-by-value.

The compilation scheme $[\![\ ]\!]$ provides some intuition about the inductive rules of the C-strategy. Consider, for instance, that an application $(M\ N)$ compiles as $[\![N]\!]; [\![M]\!]; \mathsf{Apply}$. Thus, code being executed left-to-right, the C-strategy should be *right-to-left* call-by-value.

However, the exact rules of the C-strategy can be found only by closely examining the state decompilation rules of the previous section. Consider a FAM transition $D \to D'$, such that D is a single frame that decompiles to an application. First, assume that the last steps of both decompilations $\overline{D}$ and $\overline{D'}$ are applications of the rule (AppRight). Then, with the notations of figure 2, we get $\overline{D} = (M\,[s]\ N)$ and $\overline{D'} = (M\,[s]\ N')$, with $N \xrightarrow{\text{C}} N'$ by induction. Hence, we get the rule (AppRight) of the C-strategy (figure 3). Assuming that both $\overline{D}$ and $\overline{D'}$ are computed by the rule (AppLeft), we get the rule (AppLeft) of the C-strategy, by an identical argument. A more interesting case occurs when $\overline{D}$ is computed by the rule (Code). Suppose $\Phi(D) = ((), s, i; C; C'; \mathsf{Apply})$, we have:

$$\frac{\dfrac{C \Downarrow^m N \quad C' \Downarrow^m M}{i; C; C'; \mathsf{Apply} \Downarrow^m (M\ N)}\ \text{(Apply)}}{(), s, i; C; C'; \mathsf{Apply} \Downarrow (M\ N)\,[s]}\ \text{(Code)}$$

Then, D' is computed from D by executing the instruction i. As for $\overline{D'}$, we get:

$$\frac{S, s, C \Downarrow N' \quad C' \Downarrow^m M}{S, s, C; C'; \mathsf{Apply} \Downarrow (M\,[s]\ N')}\ \text{(AppRight)}$$

By induction hypothesis applied to the substates D_0 and D'_0, such that $\Phi(D_0) = ((), s, i; C)$ and $\Phi(D'_0) = (S, s, C)$, we get $N\,[s] \xrightarrow{\text{C}} N'$. Hence we get the rule (MapApp) of the C-strategy. Observe that, using this rule, the C-strategy may combine several σ_w reductions in one step.

For space concern, we do not prove here that the FAM implements the C-strategy. Intuitively, the C-strategy computes substitutions left-to-right in the environment part of closures (cf. the rules (ConsLeft), (ConsRight) and (ClosRight)). Moreover, the propagation of substitutions through closures and cons nodes "$\cdot$" is combined with other σ_w-reduction steps (cf. the rules (MapClos) and (MapEnv)). The full proof, which we omit, generalizes the case of application nodes given in this section. It does so in two directions: first we must consider closures nodes (i.e., the decompilation rule (Fun(n,i))), second we must consider multi-frame states, (i.e., the decompilation rule (State)).

5 Other execution models

In the previous sections, we have exhibited the $\lambda\sigma_w$-calculus strategies that hidden inside the Krivine machine and the FAM. In [21], the strategies implemented by two other environment machines, the SECD machine [16] and the CAM [7] are described. We also made explicit the strategy of the ZAM [18]. Briefly, the SECD and the ZAM implement the right-to-left call-by-value strategy, whereas the CAM implements the left-to-right call-by-value strategy.

The G-machine and the TIM [13] can also be understood in the $\lambda\sigma$-calculus, although these machines look quite different from environment machines. In order to simplify the

$$\frac{(\lambda M)\,[s] \text{ and } N \text{ are C-values}}{((\lambda M)\,[s])\ N \xrightarrow{\mathrm{C}} M\,[N \cdot s]}\ (\mathrm{Beta}) \qquad \mathtt{n}\,[N_1 \cdots N_n \cdot s] \xrightarrow{\mathrm{C}} N_n\ (\mathrm{Var}^n)$$

$$\uparrow^n \circ (N_1 \cdots N_n \cdot \mathrm{id}) \xrightarrow{\mathrm{C}} \mathrm{id}\ (\mathrm{Shift}^n)$$

$$\frac{N \xrightarrow{\mathrm{C}} N'}{M\ N \xrightarrow{\mathrm{C}} M\ N'}\ (\mathrm{AppRight}) \qquad \frac{M \xrightarrow{\mathrm{C}} M' \quad N \text{ is a C-value}}{M\ N \xrightarrow{\mathrm{C}} M'\ N}\ (\mathrm{AppLeft})$$

$$\frac{s \xrightarrow{\mathrm{C}} s'}{(\lambda M)\,[s] \xrightarrow{\mathrm{C}} (\lambda M)\,[s']}\ (\mathrm{ClosRight})$$

$$\frac{M \xrightarrow{\mathrm{C}} M'}{M \cdot s \xrightarrow{\mathrm{C}} M' \cdot s}\ (\mathrm{ConsLeft}) \qquad \frac{M \text{ is a C-value} \quad s \xrightarrow{\mathrm{C}} s'}{M \cdot s \xrightarrow{\mathrm{C}} M \cdot s'}\ (\mathrm{ConsRight})$$

$$\frac{N\,[s] \xrightarrow{\mathrm{C}} N'}{(M\ N)\,[s] \xrightarrow{\mathrm{C}} (M\,[s])\ N'}\ (\mathrm{MapApp}) \qquad \frac{s \circ t \xrightarrow{\mathrm{C}} s'}{((\lambda M)\,[s])\,[t] \xrightarrow{\mathrm{C}} (\lambda M)\,[s']}\ (\mathrm{MapClos}) \qquad \frac{N\,[s] \xrightarrow{\mathrm{C}} N'}{(N \cdot t) \circ s \xrightarrow{\mathrm{C}} N' \cdot (t \circ s)}\ (\mathrm{MapEnv})$$

Figure 3: The C-strategy

management of variables at run-time, compilers for these machines translate input λ-terms into supercombinators, by the so-called λ-lifting operation [23]. Supercombinators are n-ary functions without free variables, that is, in terms of $\lambda\sigma_w$, closures $(\lambda\lambda\ldots\lambda M)\,[\mathrm{id}]$, where M is a $\lambda\sigma$-term whose variables are all λ-bound. We call suchzz closures $\lambda\sigma$-supercombinators. In the $\lambda\sigma$-framework, λ-lifting is actually quite similar to the first phase $\mathcal{C}$ of the FAM compilation. Where, from a function, the transformation $\mathcal{C}$ produces a closure, the λ-lifting will produce the partial application of a $\lambda\sigma$-supercombinator. For instance, our scheme $\mathcal{C}$ translates the abstraction $\lambda(\mathtt{1}\ (\mathtt{5}\ \mathtt{7}))$ into $(\lambda(\mathtt{1}\ (\mathtt{2}\ \mathtt{3})))\,[\mathtt{4} \cdot \mathtt{6} \cdot \mathrm{id}]$, whereas a simple λ-lifting procedure would transform this abstraction into the $\lambda\sigma$-term $((\lambda\lambda\lambda(\mathtt{1}\ (\mathtt{2}\ \mathtt{3})))\,[\mathrm{id}])\ \mathtt{6}\ \mathtt{4}$.

Any $\lambda\sigma_w$-strategy that accounts for supercombinator reduction must include a n-ary (Beta) rule, which expresses the application of a $\lambda\sigma$-supercombinator to all its arguments in one step:

$$(\mathrm{Beta}_n) \quad ((\lambda\ldots\lambda M)\,[\mathrm{id}])\ N_1\ \ldots\ N_n \to M\,[N_n \cdots N_1 \cdot \mathrm{id}]$$

The G-machine and the TIM implement very similar strategies, which basically amounts to first contracting the leftmost-outermost (Beta_n) redex and then propagating the generated substitution. This simplified term-based presentation is sufficient for establishing the correctness of both machines.

The SML/NJ compiler departs from the abstract machine approach [2]. Roughly speaking, a schematic SML/NJ compiler would first translate a source λ-term into a λ-term in continuation-passing style (CPS). Then, this CPS λ-term would be further transformed by the so-called closure conversion. This conversion transforms functions into record data structures, which encode closures. Such records can easily be expressed in our framework by adopting a more direct encoding of closures. The resulting modified schematic compiler would now produce $\lambda\sigma$-terms in continuation-passing style. Note that the above schematic description of the SML/NJ compiler is ours and only intends to show that $\lambda\sigma$ can also account for a schematic CPS-based compiler. By no way, do we attempt to render the complexity of a full-fledged compiler as [2] does, using enriched λ-calculus (in CPS) as a formal language.

6 Related work and conclusion

The main contribution of this paper resides in the introduction the $\lambda\sigma_w$-calculus as "the" weak λ-calculus, that is, as the adequate framework for the formal study of the execution of compiled functional programs. Additionally, the full $\lambda\sigma$-calculus appears as an adequate formalism for proving the correctness of skeleton compilers. Presently, the most salient illustration of this claim is our complete description and proof of a schematic FAM based compiler.

Our work is to be compared first with similar attempts to prove, formalize or derive several functional back-ends in an unified formalism. In [9], the Krivine machine and the CAM, two shared environment abstract machines, are "derived" from deterministic strategies of $\lambda\rho$, a calculus of closures. The system $\lambda\rho$ is a *conditional* term rewriting system (see also [20]) and can be seen as a predecessor of our standard term rewriting system $\lambda\sigma_w$. A recent publication [11] resembles our work, since it models many compilers and abstract machines, using the λ-calculus extended with appropriate combinators as a formal language. We differ from this work on an important point: we insist on what all functional runtime system have in common, to the point of proposing a definition for compiled functionality in a relatively well established formalism, whereas [11] focuses more on modeling the exact structure of abstract machines, in order to establish their "taxonomy" of functional languages implementation.

Our work is to be compared also with other works that formally prove one or a few abstract machines. Here, a first benefit of our approach of describing abstract machines in terms of a $\lambda\sigma$-calculus rewriting strategy is that their correctness is a direct consequence of the correctness of the $\lambda\sigma$-calculus with respect to the λ-calculus. As a consequence, our correctness proofs appear to be quite simple. By contrast, the correctness proofs of the CAM in [4] and of the SECD machine in [24] were complicated. Moreover, our simple technique enabled us to prove the correctness of the FAM, which has never been done before. We believe that this simplicity owes much to the fact that our overall framework (i.e., the $\lambda\sigma_{\Uparrow}$ calculus) includes both our archetypal source and target languages as consistent (i.e., closed by reduction and Church-Rosser) subcalculi.

The second benefit of our approach lies in the generality and precision of our correctness results: all machines are described in the same framework and we describe every step of their execution. Here, we differ from [14], which relied upon natural or "big-step" semantics and from [8, 18], which proved the Krivine machine and the ZAM in $\lambda\sigma$ but do not specify their $\lambda\sigma$-strategies. Our "small-step" approach to semantics enables us to compare the termination properties of different machines naturally. For instance, we can say that the Krivine machine terminates more often than the CAM or the SECD, since it follows the leftmost-outermost strategy, which terminates more often than any other strategy of the orthogonal weak λ-calculus [20]. As a second example, given the same λ-term as input, the SECD and the CAM compute exactly the same closure, whereas the FAM computes a different, $\lambda\sigma_{\Uparrow}$-equivalent closure.

A first direction for future work is to study graph-based implementations. Considering that the call-by-need strategy is the natural implementation of call-by-name in graph rewriting systems, such a strategy can be modeled in a simple extension to term-graphs of the weak λ-calculus with explicit substitutions. To render sharing while preserving the desirable simplicity of terms, several techniques already exist, such as subterms labeling [20], explicit recursive equations [3], or specialized bindings [17].

A second direction is to examine how the full $\lambda\sigma$-calculus can be used to assert the correctness of some phases of realistic compilers. Various optimizations at the closure level are a first natural target for such a study. Considering skeleton compilers that are closer to real compilers would require first to extend $\lambda\sigma$ to handle common programming constructs, such as data structures, recursive bindings, exceptions,...

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy, "*Explicit Substitutions*", POPL'90.

[2] A. W. Appel, "*Compiling with Continuations*", Cambridge University Press, 1992.

[3] Z. Ariola, M. Feleisen. J. Maraist, M. Odersky, and P. Wadler, "A call-by-need lambda-calculus". POPL'95.

[4] A. Asperti, "*A Categorical Understanding of Environment Machines*", JFP, 2(1), 1992.

[5] L. Augustsson, "*A Compiler For Lazy ML*", LFP'84.

[6] L. Cardelli, "*Compiling a Functional Language*", LFP'84.

[7] G. Cousineau, P.-L. Curien and M. Mauny, "*The Categorical Abstract Machine*", FPCA'85.

[8] P. Crégut, "*An Abstract Machine for the Normalization of Lambda-terms*", LFP'90.

[9] P.-L. Curien, "An Abstract Framework for Environment Machines", TCS. 82, 1991.

[10] P.-L. Curien, T. Hardin and J.-J. Lévy, "*Confluence Properties of Weak and Strong Calculi of Explict Substitutions*", INRIA Research Report No 1617, 1992. Also to appear in JACM in april 1996.

[11] R. Douence and P. Fradet, "*Towards a Taxonomy of Functional Language Implementations*". PILP'95.

[12] G. Dowek, T. Hardin and C.Kirchner, "*Higher-Order Unification via explicit Substitutions*", LICS'95.

[13] J. Fairbairn and S. Wray, "*Tim: A Simple, Lazy Abstract Machine to Execute Supercombinators*", FPCA'87.

[14] J. Hannan, D. Miller, "*From Operational Semantics to Abstract Machines*", Journal of Mathematical Structures in Computer Science , 2(4):415-459, 1992

[15] H. Herbelin, "*A λ-Calculus Structure Isomorphic to Sequent Calculus Structure*", CSL'94.

[16] P. J. Landin, "*The Mechanical Evaluation of Expressions*", Computer Journal, Vol 6, No 4, 1964.

[17] J. Launchbury, "*A Natural Semantics for Lazy Evaluation*", POPL'93.

[18] X. Leroy, "*The ZINC Experiment: An Economical Implementation of the ML Language*", INRIA Technical Report 117, 1990.

[19] P. Lescanne, "*From $\lambda\sigma$ to $\lambda\upsilon$: a Journey through Calculi of Explicit Substitutions*", POPL'94.

[20] L. Maranget, "*Optimal derivation in Orthogonal Rewriting Systems and in Weak Lambda Calculi*", POPL'91.

[21] B. Pagano, "*Bi-simulation de machines abstraites en lambda-sigma-calcul*", Journées Francophone des Langages Applicatifs 1995, INRIA Éditions (In french).

[22] G.D. Plotkin, "*LCF Considered as a Programming Language*". TCS, 5:225-255, 1977.

[23] S. L. Peyton Jones, "*The implementation of Functional Programming Languages*", Prentice-Hall, 1987.

[24] M. Rittri, "*Proving the Correctness of a Virtual Machine by a Bisimulation*", Phd thesis, University of Göteborg and Chalmers University of Technology, 1988.