

## *Functional runtime systems within the lambda-sigma calculus\**

THÉRÈSE HARDIN

*LIP6, Université Pierre et Marie Curie, 75252 Paris Cedex 05, France  
INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France  
(e-mail: Therese.Hardin@lip6.fr)*

LUC MARANGET

*INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France  
(e-mail: Luc.Maranget@inria.fr)*

BRUNO PAGANO

*LIP6, Université Pierre et Marie Curie, 75252 Paris Cedex 05, France  
(e-mail: Bruno.Pagano@lip6.fr)*

---

### **Abstract**

We define a weak  $\lambda$ -calculus,  $\lambda\sigma_w$ , as a subsystem of the full  $\lambda$ -calculus with explicit substitutions  $\lambda\sigma_\eta$ . We claim that  $\lambda\sigma_w$  could be the archetypal output language of functional compilers, just as the  $\lambda$ -calculus is their universal input language. Furthermore,  $\lambda\sigma_\eta$  could be the adequate theory to establish the correctness of functional compilers. Here we illustrate these claims by proving the correctness of four simplified compilers and runtime systems modelled as abstract machines. The four machines we prove are the Krivine machine, the SECD, the FAM and the CAM. Thus, we give the first formal proofs of Cardelli's FAM and of its compiler.

---

### **Capsule Review**

This paper describes a single framework in which one can carry out the formalizations and proofs of correctness for a variety of abstract machines. Although the basic ideas are known, this is the first thorough treatment of several abstract machines within lambda-sigma. Moreover, the paper contains the first correctness proof for one machine, the FAM, that uses environments in a more sophisticated way; it seems that this proof would be hard if carried out without the support that lambda-sigma offers for manipulating environments. The paper is useful also because, by virtue of working in a single framework, it can bring out the similarities and differences between the different machines: a direct comparison between the machines is possible by inspecting the lambda-sigma reduction strategy that each implements.

---

\* This work was partially supported by the ESPRIT Basic Research Project 6454–CONFER.

## 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  $\lambda$ -calculus is widely accepted as such a paradigm of all functional programming languages, due to its simplicity, consistency and generality. More precisely, the  $\lambda$ -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 (Plotkin, 1977).

In opposition to this commonly accepted view of  $\lambda$ -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 (Augustsson, 1984) terms, to  $\lambda$ -terms in continuation passing style (CPS) (Appel, 1992), or to abstract machines (Landin, 1964; Cousineau *et al.*, 1985; Cardelli, 1984; Crégut, 1990; Leroy, 1990). These different approaches are praised for their peculiarities: combinators for their rewriting aspects and adequacy for lazy evaluation (Peyton-Jones, 1987), 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 the true instructions of 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  $\lambda$ -calculus with explicit substitutions (Abadi *et al.*, 1996; Curien *et al.*, 1996) as a term  $(\lambda x.M)[s]$ , 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, we

need to apply a closure  $(\lambda x.M)[s]$  to an argument  $N$ , yielding the explicit application of the new substitution  $t = x \setminus N \cdot s$  to the body  $M$ , written  $M[x \setminus N \cdot s]$ . 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  $\lambda$ -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,  $\lambda\sigma_w$ .

Compilation takes as input a  $\lambda$ -term  $N$  and produces a  $\lambda\sigma_w$ -term  $M$ . It turns out that both terms are valid terms of  $\lambda\sigma_{\uparrow}$ , a general calculus of explicit substitutions (Curien *et al.*, 1996) and that the compilation process can be expressed as a rewriting in  $\lambda\sigma_{\uparrow}$ . While designing  $\lambda\sigma_w$ , 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 functional runtime systems. We are satisfied that this pragmatic approach yields a confluent calculus, which moreover is a subcalculus of  $\lambda\sigma_{\uparrow}$ .

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, the SECD, the FAM and the CAM. The FAM example involves a true compilation of the input  $\lambda$ -term to a  $\lambda\sigma_w$ -term, which requires the full power of  $\lambda\sigma_{\uparrow}$  to assert its correctness. 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 7.

We thus illustrate our claim that weak  $\lambda$ -calculus with explicit substitutions is an adequate tool to study the execution of compiled functional programs (Abadi *et al.*, 1996; Curien *et al.*, 1996; Maranget, 1991). More ambitiously, we claim that  $\lambda\sigma_w$  is a good candidate for being the universal formalism of *compiled* functional languages: it expresses the essence of compiled functionality (closures and step by step substitutions), it is simple (as an ordinary, first order term rewriting system), and it is non-ambiguous (it has the Church–Rosser property). Moreover, as shown by the FAM, the full  $\lambda$ -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  $\lambda$ -calculus, such as higher order unification (Dowek *et al.*, 1995), or issues in logic, such as the interpretation of sequent calculus (Herbelin, 1994).

## 2 Preliminaries

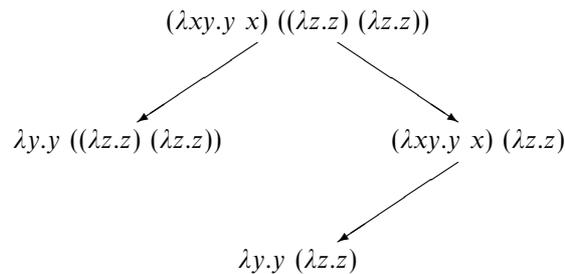
### 2.1 The lambda-calculus with explicit substitutions

The traditional weak  $\lambda$ -calculus is an attempt to model the execution of machine code within the  $\lambda$ -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  $(\xi)$  rule from the

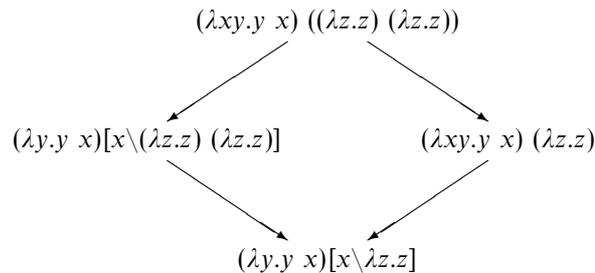
definition of the  $\lambda$ -calculus.

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

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



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



Informally, ‘reduction is not allowed under  $\lambda$ ’s’ is replaced by ‘substitution does not cross  $\lambda$ ’s’. Closing the Church-Rosser diagram for the weak  $\lambda$ -calculus is not anecdotic. It means that weak  $\lambda$ -calculus is a consistent formalism, in which various computations results and strategies to reach them can be compared.

Variable names do not appear at runtime, they have been compiled into memory accesses. We represent such accesses by De Bruijn indices. The De Bruijn notation avoids the  $\alpha$ -conversion burden, replacing a given occurrence of a variable by the number of  $\lambda$ ’s that separates this occurrence from its binder.

A natural setting for a formal treatment of closures is  $\lambda\sigma$ , the  $\lambda$ -calculus with explicit substitutions (Abadi *et al.*, 1996; Curien *et al.*, 1996). We first recall the definition of  $\Lambda_\sigma$ , the two sorted algebra of  $\lambda\sigma$ -terms.

TERMS:  $M ::= n \mid (MM) \mid \lambda M \mid M[s]$ , with  $n \geq 1$   
 SUBSTITUTIONS:  $s ::= \text{id} \mid \uparrow \mid M \cdot s \mid s \circ s$

(App)	$(M_1 M_2) [s]$	$\rightarrow$	$(M_1 [s] M_2 [s])$
(FVar)	$1 [M \cdot s]$	$\rightarrow$	$M$
(RVar)	$n+1 [M \cdot s]$	$\rightarrow$	$n [s]$
(Clos)	$(M [s]) [t]$	$\rightarrow$	$M [s \circ t]$
(AssEnv)	$(s \circ t) \circ u$	$\rightarrow$	$s \circ (t \circ u)$
(MapEnv)	$(M \cdot s) \circ t$	$\rightarrow$	$M [t] \cdot (s \circ t)$
(ShiftCons)	$\uparrow \circ (M \cdot s)$	$\rightarrow$	$s$
(IdL)	$\text{id} \circ s$	$\rightarrow$	$s$

 Fig. 1. System  $\sigma_w$  for weak substitution rules.

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  $\text{id}$  and  $\uparrow$ , whereas substitutions are structured as lists of terms  $M \cdot s$  or as compositions  $s \circ s$ .

We note  $\Lambda_{\text{DB}}$  the subset of  $\Lambda_\sigma$  that coincides with ordinary  $\lambda$ -terms.

$$\lambda_{\text{DB-TERMS}}: N ::= n \mid (NN) \mid \lambda N, \quad \text{with } n \geq 1$$

The  $\lambda\sigma$ -calculi that we use are plain (i.e. non-conditional) term rewriting systems. This means that any subterm can be reduced as soon as it matches the left-hand side of a rule. The propagation of substitutions inside terms and substitutions is defined by the term rewriting system  $\sigma_w$  (see figure 1).

There is one rule per term construct in the algebra of  $\lambda\sigma$ -terms, except for  $\lambda$ . The propagation of substitutions through application nodes and accesses inside list-structured substitutions are handled by the first three rules of figure 1 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  $\lambda M$  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 of  $\sigma_w$ . 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 calculi of explicit substitutions without composition (Lescanne, 1994).

Since there is no rule for crossing  $\lambda$ ’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, etc.

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

$$\text{(Beta)} \quad ((\lambda M_1) [s] M_2) \rightarrow M_1 [M_2 \cdot s]$$

The system  $\lambda\sigma_w$  is defined by the rules of  $\sigma_w$  plus the rule (Beta). From (Curien *et al.*, 1996), where a system also named  $\lambda\sigma_w$ , very close to ours, is studied, we deduce that the weak substitution system  $\sigma_w$  is both strongly normalizing and confluent and, by using the Yokouchi lemma, that the reduction system  $\lambda\sigma_w$  is confluent. Moreover, our system  $\lambda\sigma_w$  is a subcalculus of  $\lambda\sigma_{\uparrow}$ , a very general  $\lambda$ -calculus with explicit substitutions that enjoys confluence on open terms and can encode most, if not all, other calculi of explicit substitutions (Curien *et al.*, 1996).

(Lambda)	$(\lambda M) [s]$	$\rightarrow$	$\lambda(M [\uparrow(s)])$
(VarShift1)	$n [\uparrow]$	$\rightarrow$	$n+1$
(VarShift2)	$n [\uparrow \circ s]$	$\rightarrow$	$n+1 [s]$
(FVarLift1)	$1 [\uparrow(s)]$	$\rightarrow$	$1$
(FVarLift2)	$1 [\uparrow(s) \circ t]$	$\rightarrow$	$1 [t]$
(RVarLift1)	$n+1 [\uparrow(s)]$	$\rightarrow$	$n [s \circ \uparrow]$
(RVarLift2)	$n+1 [\uparrow(s) \circ t]$	$\rightarrow$	$n [s \circ (\uparrow \circ t)]$
(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)	$\uparrow(\text{id})$	$\rightarrow$	$\text{id}$
(IdR)	$s \circ \text{id}$	$\rightarrow$	$s$
(Id)	$M [\text{id}]$	$\rightarrow$	$M$

Fig. 2. System  $\sigma_{\uparrow}$  for strong substitution rules.

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 rules of  $\sigma_w$  plus the strong substitution rules of figure 2. 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  $\sigma_{\uparrow}$  make explicit the de Bruijn indices adjustments. The remaining two rules – (IdR) and (Id) – define the substitution  $\text{id}$  as the identity. As it can be expected,  $\sigma_{\uparrow}$  is a terminating and confluent rewriting system (Curien *et al.*, 1996).

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

$$\text{(BetaStrong)} \quad ((\lambda M_1) M_2) \rightarrow M_1 [M_2 \cdot \text{id}]$$

The system  $\lambda\sigma_{\uparrow}$  is both confluent and correct with respect to the  $\lambda$ -calculus (Curien *et al.*, 1996).

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

$$\begin{aligned} ((\lambda M_1) [s] M_2) &\xrightarrow{\text{(Lambda)}} (\lambda(M_1 [\uparrow (s)])) M_2 \xrightarrow{\text{(BetaStrong)}} (M_1 [\uparrow (s)]) [M_2 \cdot \text{id}] \xrightarrow{\text{(Clos)}} \\ &M_1 [\uparrow (s) \circ (M_2 \cdot \text{id})] \xrightarrow{\text{(LiftEnv)}} M_1 [M_2 \cdot (s \circ \text{id})] \xrightarrow{\text{(IdR)}} M_1 [M_2 \cdot s] \end{aligned}$$

Thus, as a subsystem of the strong system  $\lambda\sigma_{\uparrow}$ , the weak system  $\lambda\sigma_w$  is also correct with respect to the  $\lambda$ -calculus. This correctness is to be understood as follows: given any  $\lambda\sigma$ -term  $M$ , the  $\sigma_{\uparrow}$ -normal form  $\sigma_{\uparrow}(M)$  is a  $\lambda_{\text{DB}}$ -term. Furthermore, if  $M$  reduces to  $M'$  by the rule (StrongBeta), then  $\sigma_{\uparrow}(M)$   $\beta$ -reduces to  $\sigma_{\uparrow}(M')$  in one or more steps. Conversely,  $\beta$ -reduction in  $\lambda_{\text{DB}}$  can be simulated by a (StrongBeta) rewriting step followed by  $\sigma_{\uparrow}$ -normalization. Again, refer to Curien *et al.* (1996, Section 5) for details and proofs.

In this paper, functional programs are modelled as closed  $\lambda$ -terms. More precisely, given a  $\lambda$ -term in De Bruijn notation  $N$ , the free variables in  $N$  are collected by calculating  $\mathcal{F}_0(N)$ , where, for any integer  $d$ ,  $\mathcal{F}_d$  is defined by:

$$\begin{aligned} \mathcal{F}_d(\mathbf{n}) &= \emptyset && \text{if } n \leq d \\ \mathcal{F}_d(\mathbf{n}) &= \{n - d\} && \text{if } n > d \\ \mathcal{F}_d(N_1 N_2) &= \mathcal{F}_d(N_1) \cup \mathcal{F}_d(N_2) \\ \mathcal{F}_d(\lambda N) &= \mathcal{F}_{d+1}(N) \end{aligned}$$

By definition, a  $\lambda_{\text{DB}}$ -term  $N$  is closed, if and only if, the set  $\mathcal{F}_0(N)$  is empty. Furthermore, given any  $\lambda\sigma$ -term  $M$ , we say that  $M$  is closed, if and only if its  $\sigma_{\uparrow}$ -normal form is closed. This closeness property is preserved by reduction:

*Lemma 1*

Let  $M$  be a closed  $\lambda\sigma$ -term. Then, for all  $M'$  such that  $M \xrightarrow{\lambda\sigma_{\uparrow}} M'$ , the  $\lambda\sigma$ -term  $M'$  is also closed.

*Proof*

The property holds for  $\lambda_{\text{DB}}$ -terms and  $\beta$ -reduction. It smoothly extends to  $\lambda\sigma$ -terms and  $\lambda\sigma_{\uparrow}$  reduction.  $\square$

### 2.2 Abstract machines

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

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

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

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

$$\begin{aligned} \text{CLOSURE} &::= (\text{CODE}/\text{ENVIRONMENT}) \\ \text{ENVIRONMENT} &::= () \quad | \quad \text{CLOSURE} \cdot \text{ENVIRONMENT} \end{aligned}$$

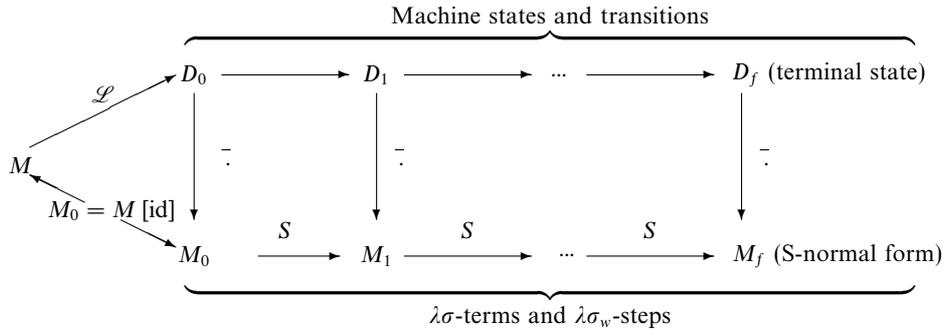


Fig. 3. Summary of bisimulation conditions.

A typical code segment 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 behaviour 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 in sections 3–6, bearing in mind the following notations:

**Convention:**Our stacks grow right to left. For instance, pushing the element  $x$  onto the stack  $S$  yields the new stack  $x : S$ . When appropriate, we freely interpret stacks as sequences or arrays. That is, given  $n$  elements  $x_1, x_2, \dots, x_n$ , the stack  $S = x_n : \dots : x_2 : x_1 : ()$  is simply written  $x_n : \dots : x_2 : x_1$ . A subsequence  $x_j : \dots : x_i$  is written  $\overline{x_{j,i}}$ . For instance,  $\overline{x_{n,n-i}} : S$  is a stack with the  $i$  elements  $x_n, x_{n-1}, \dots, x_{n-i}$  standing on top of it.

### 2.3 Implementation of a strategy by a machine

In Rittri (1988), bisimulations are used to establish the equivalence of two transition systems  $\Sigma_1$  and  $\Sigma_2$ . In our work,  $\Sigma_1$  always defines an abstract machine whereas  $\Sigma_2$  is a subsystem of  $\lambda\sigma_w$ . The states of  $\Sigma_2$  are  $\lambda\sigma$ -terms and its transition relation is a rewriting strategy  $\xrightarrow{S}$ , 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  $\bar{\cdot}$  that translates machine states into  $\lambda\sigma$ -terms.

For simplicity, we retain the word ‘bisimulation’ to name the relations we define between abstract machines and  $\lambda\sigma$ . We say that a machine implements a strategy  $S$  when the following four conditions are satisfied:

1. *Initial condition*: Let  $M$  be a  $\lambda\sigma$ -term such that  $\mathcal{L}(M)$  exists. Then  $\overline{\mathcal{L}(M)} = M [\text{id}]$ .
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  $\overline{D_2} = \overline{D_1}$  —and we say the machine performs a *silent transition*— or  $\overline{D_1} \xrightarrow{S} \overline{D_2}$ .
3. *Terminal states translate to normal forms*: Let  $M$  be a  $\lambda\sigma$ -term such that  $\mathcal{L}(M)$  exists. If  $\mathcal{L}(M) \rightarrow^* D$  and  $D$  is a terminal state, then  $\overline{D}$  is a  $S$ -normal form.
4. *Machine and strategy progress at the same pace*. There cannot be infinitely many consecutive silent transitions.

The diagram in figure 3 summarizes the bisimulation conditions, in the ideal case when there are no silent transitions.

Condition 1 is justified by the strong rule (Id):  $M [\text{id}] \rightarrow M$  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  $\lambda\sigma_{\uparrow}$  to assert the correctness of the compilation.

As defined in condition 2, silent transitions perform only administrative work on the abstract machinery.

Observe that, by the last three bisimulation conditions and by the determinacy of machines and strategies, the converse property of condition 2 holds. Let  $D_1$  be a machine state, such that  $\overline{D_1}$  exists and that  $\overline{D_1} \xrightarrow{S} M_2$ . Then, there exists a machine state  $D_2$ , with  $\overline{D_2} = M_2$  and  $D_1 \rightarrow^* D_2$ , where  $\rightarrow^*$  is zero or more silent transitions followed by one non-silent transition.

### 3 The Krivine Machine

As a gentle introduction to our framework, we describe the Krivine machine — see (Crégut, 1990), for instance. This machine is very simple:

INSTRUCTION ::= Grab | Push(CODE) | Access( $n$ )  
 FRAME ::= CLOSURE

A typical state  $D$  is thus a stack of closures, which we write  $f_n :: f_{n-1} :: \dots :: f_1$ .

A  $\lambda_{\text{DB}}$ -term is compiled as follows:

$\llbracket n \rrbracket = \text{Access}(n)$   
 $\llbracket \lambda N \rrbracket = \text{Grab}; \llbracket N \rrbracket$   
 $\llbracket (N_1 N_2) \rrbracket = \text{Push}(\llbracket N_2 \rrbracket); \llbracket N_1 \rrbracket$

Loading of compiled code is just pairing with the empty environment:  $\mathcal{L}(N) = (\llbracket N \rrbracket / ()$ .

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

$$\begin{aligned}
(\text{Access}(1)/(C_0/e_0) \cdot e) :: D &\xrightarrow{\text{lvar}} (C_0/e_0) :: D \\
(\text{Access}(n+1)/(C_0/e_0) \cdot e) :: D &\xrightarrow{\text{rvar}} (\text{Access}(n)/e) :: D \\
(\text{Push}(C'); C/e) :: D &\xrightarrow{\text{push}} (C/e) :: (C'/e) :: D \\
(\text{Grab}; C/e) :: (C'/e') :: D &\xrightarrow{\text{grab}} (C/(C'/e') \cdot e) :: D
\end{aligned}$$

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

$$\begin{aligned}
\overline{\text{Access}(n)} &= \mathbf{n} & \overline{(C/e)} &= \overline{C} [\overline{e}] \\
\overline{\text{Grab}; C} &= \lambda \overline{C} & \overline{()} &= \text{id} \\
\overline{\text{Push}(C'); C} &= (\overline{C} \overline{C'}) & \overline{f \cdot e} &= \overline{f} \cdot \overline{e}
\end{aligned}$$

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:

$$\overline{f_n :: f_{n-1} :: \dots :: f_1} = (\dots (\overline{f_n} \overline{f_{n-1}}) \dots \overline{f_1})$$

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

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

*Lemma 2*

$\llbracket \llbracket N \rrbracket \rrbracket = N$ , for any  $\lambda_{\text{DB}}$ -term  $N$ .

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 3*

Let  $D$  and  $D'$  be two machine states such that  $\overline{D}$  is defined and  $D \rightarrow D'$ . 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:

$$\begin{array}{ccc}
(\text{Push}(C'); C/e) :: D_0 &\xrightarrow{\text{push}} & (C/e) :: (C'/e) :: D_0 \\
\downarrow \cdot & & \downarrow \cdot \\
(\dots (\overline{C} \overline{C'}) [\overline{e}] \dots \overline{f_1}) &\xrightarrow{\lambda\sigma_w} & (\dots (\overline{C} [\overline{e}] \overline{C'} [\overline{e}]) \dots \overline{f_1})
\end{array}$$

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.  $\square$

By the detailed proof of the previous lemma, it is not difficult to see that condition 2 holds, once the weak leftmost strategy, or K-strategy, is adopted. Additionally, there are no silent transitions. The K-strategy is described below in the small step formalism:

$$\begin{array}{l}
 1 [M \cdot s] \xrightarrow{K} M \qquad n+1 [M \cdot s] \xrightarrow{K} n [s] \\
 (N_1 N_2) [s] \xrightarrow{K} (N_1 [s] N_2 [s]) \qquad ((\lambda N) [s] M) \xrightarrow{K} N [M \cdot s] \\
 \\
 \frac{M_1 \xrightarrow{K} M'_1}{(M_1 M_2) \xrightarrow{K} (M'_1 M_2)}
 \end{array}$$

It remains to show condition 3 on terminal states.

*Lemma 4*

Let  $D$  be a reachable, terminal state, then  $\bar{D}$  is a K-normal form:

*Proof*

The Krivine machine may stop for two reasons:

- When  $D = (\text{Grab}; C/e)$ . Then, we get  $\bar{D} = (\lambda \bar{C}) [\bar{e}]$ , which is a  $\lambda\sigma$ -closure and a K-normal form.
- When  $D = (\text{Access}(m)/()) :: D_0$ , i.e. when an access fails. Then, we get  $\bar{D} = (\dots(m \bar{f}_{n-1}) \dots \bar{f}_1)$ , which is also a K-normal form. Moreover, since we have  $\sigma_{\uparrow}(M_1 M_2) = (\sigma_{\uparrow}(M_1) \sigma_{\uparrow}(M_2))$ , the  $\sigma_{\uparrow}$ -normal form  $\sigma_{\uparrow}(\bar{D})$  admits at least  $m$  as a free variable. Thus, by Lemma 1, this case can occur only when the initial program is not a closed  $\lambda_{\text{DB}}$ -term.  $\square$

Finally, as the Krivine machine does not perform silent transition, the final result of this section immediately follows from the previous lemmas.

*Theorem 1*

The Krivine machine implements the K-strategy.

### 4 The SECD machine

#### 4.1 SECD in the $\lambda_{\text{DB}}$ -calculus

The original SECD machine of Landin (1994) used named variables. In our presentation, we consider a slightly modified SECD machine that reduces  $\lambda_{\text{DB}}$ -terms. Our choice to define machine states as lists of frames also induces minor syntactic modifications with respect to usual presentations of the SECD machine.

An instruction is a  $\lambda_{\text{DB}}$ -term or a new symbol @.

$$\text{INSTRUCTION} := \lambda_{\text{DB}}\text{-TERM} \mid @$$

An argument stack  $AS$  is a list of closures.

$$\text{STACK} := () \mid \text{CLOSURE} : \text{STACK}$$

Finally, a frame of the SECD machine is a  $\langle AS \cdot e \cdot C \rangle$  triple:

$$\text{FRAME} := \langle \text{STACK} \cdot \text{ENVIRONMENT} \cdot \text{CODE} \rangle$$

Thus, a SECD state is written  $D = \langle AS_n \cdot e_n \cdot C_n \rangle :: \dots :: \langle AS_2 \cdot e_2 \cdot C_2 \rangle :: \langle AS_1 \cdot e_1 \cdot C_1 \rangle$ .

The transition rules are as follows:

$$\begin{aligned} \langle AS \cdot e \cdot (N_1 N_2); C \rangle :: D &\xrightarrow{\text{app}} \langle AS \cdot e \cdot N_2; N_1; @; C \rangle :: D \\ \langle AS \cdot e \cdot \lambda N; C \rangle :: D &\xrightarrow{\text{lam}} \langle (N/e) : AS \cdot e \cdot C \rangle :: D \\ \langle (N_0/e_0) : f : AS \cdot e \cdot @; C \rangle :: D &\xrightarrow{@} \langle () \cdot f \cdot e_0 \cdot N_0 \rangle :: \langle AS \cdot e \cdot C \rangle :: D \\ \langle f \cdot e \cdot () \rangle :: \langle AS' \cdot e' \cdot C' \rangle :: D &\xrightarrow{\text{dump}} \langle f : AS' \cdot e' \cdot C' \rangle :: D \\ \langle AS \cdot f_1 \dots f_n \cdot e \cdot n; C \rangle :: D &\xrightarrow{\text{var}} \langle f_n : AS \cdot f_1 \dots f_n \cdot e \cdot C \rangle :: D \end{aligned}$$

The SECD machine looks very much like an interpreter and the compile and load function is minimal: for any  $\lambda_{\text{DB}}$ -term  $N$ , we define  $\mathcal{L}(N) = \langle () \cdot () \cdot N \rangle$ .

#### 4.2 The decompilation

As in the previous case of the Krivine machine we view the decompilation function as an inverse of the compilation function. For the most simple structures – environments, closures and stacks – there is very little to do.

$$\text{Environments: } \begin{cases} \bar{()} = \text{id} \\ \overline{f \cdot e} = \bar{f} \cdot \bar{e} \end{cases}$$

$$\text{Closures: } \overline{(N/e)} = (\lambda N)[\bar{e}]$$

$$\text{Stacks: } \overline{f_n : f_{n-1} : \dots : f_1} = \bar{f}_n : \bar{f}_{n-1} : \dots : \bar{f}_1$$

From the SECD point of view, the results of computations are closures  $(N/e)$ . From the general  $\lambda\sigma$  point of view, results are weak values, i.e. closure terms  $\lambda N [s]$ , where  $N$  is a  $\lambda_{\text{DB}}$ -term and  $s$  is a substitution. In the more precise case of the interpretation of the SECD in the  $\lambda\sigma_w$ -calculus, values are translations of SECD closures. These Landin values or L-values are defined by the following grammar:

$$\begin{aligned} \text{L-VALUES:} \quad V &::= (\lambda N)[e] \\ \text{L-ENVIRONMENTS:} \quad e &::= \text{id} \mid V \cdot e \end{aligned}$$

Observe that a L-value  $(\lambda N)[e]$  is a  $\lambda\sigma_w$ -normal form, since  $N$  is a  $\lambda_{\text{DB}}$ -term, which is irreducible by the rules of  $\lambda\sigma_w$ .

The decompilation of frames and states is a bit more complicated, it is best described as the composition of two functions. The first decompilation phase  $\Phi$  decompiles the closures appearing inside environments and stacks.

$$\Phi(\langle AS_n \cdot e_n \cdot C_n \rangle :: \dots :: \langle AS_1 \cdot e_1 \cdot C_1 \rangle) = \langle \overline{AS_n} \cdot \bar{e}_n \cdot C_n \rangle :: \dots :: \langle \overline{AS_1} \cdot \bar{e}_1 \cdot C_1 \rangle$$

Given a state  $D = \langle AS_n \cdot e_n \cdot C_n \rangle :: \dots :: \langle AS_1 \cdot e_1 \cdot C_1 \rangle$ , we write  $\Phi(D) = \langle S_n \cdot s_n \cdot C_n \rangle :: \dots :: \langle S_1 \cdot s_1 \cdot C_1 \rangle$ , where the new  $S_i$ 's are stacks of L-values and the  $s_i$ 's are  $\lambda\sigma$ -substitutions, that stand for the respective translations of argument stacks  $\overline{AS_i}$  and environments  $\overline{e_i}$ .

Then, the decompilation  $\overline{D}$  of a state  $D$  is computed by proving a judgment  $\Phi(D) \Downarrow \overline{D}$ , using the following rules:

$$\begin{array}{c} \langle M \cdot s \cdot () \rangle \Downarrow M \text{ (Res)} \qquad \langle () \cdot s \cdot N \rangle \Downarrow N [s] \text{ (Code)} \\ \frac{\langle S \cdot s \cdot C \rangle \Downarrow M}{\langle S \cdot s \cdot C; N; @ \rangle \Downarrow (N [s] M)} \text{ (AppRight)} \quad \frac{\langle S \cdot s \cdot C \rangle \Downarrow M_1 \text{ (where } S \neq ())}{\langle S : M_2 \cdot s \cdot C; @ \rangle \Downarrow (M_1 M_2)} \text{ (AppLeft)} \\ \frac{\langle S_n \cdot s_n \cdot C_n \rangle \Downarrow M_n \quad \langle M_n : S_{n-1} \cdot s_{n-1} \cdot C_{n-1} \rangle :: \dots :: \langle S_1 \cdot s_1 \cdot C_1 \rangle \Downarrow M}{\langle S_n \cdot s_n \cdot C_n \rangle :: \langle S_{n-1} \cdot s_{n-1} \cdot C_{n-1} \rangle :: \dots :: \langle S_1 \cdot s_1 \cdot C_1 \rangle \Downarrow M} \text{ (State)} \end{array}$$

In the decompilation rules above, the stacks  $S$  grow right-to-left and we use shortcuts in notations: the empty stack is written  $()$  (rule (Code)), a stack with a single element  $M$  is simply written  $M$  (rule (Res)), in a stack  $S : M_2$ ,  $M_2$  is the bottom element of the stack (rule (AppLeft)) and in a stack  $(M_1 : S_2)$ ,  $M_1$  is the topmost element of the stack (rule (State)). Finally, it is worth noticing that a side condition  $S \neq ()$  (i.e.  $S$  is not empty) applies to the premise of the rule (AppLeft).

The basic idea of our decompilation procedure is as follows: in a triple  $\langle S \cdot s \cdot C \rangle$  such that  $\langle S \cdot s \cdot C \rangle \Downarrow M$  holds, the stack  $S$  is a list of the subterms of  $M$  that have already been computed by the machine, whereas the code segment  $C$  represents the part of  $M$  whose computation is still pending, the external references in  $C$  being relative to the current substitution  $s$ . Decompilation is by induction on the  $\langle S \cdot s \cdot C \rangle$  triple structure. At each inductive decompilation step, some subterms are combined. One of these subterms is neither fully reduced nor yet-to-be-computed: it is being computed. As such, it is produced, by the decompilation of a *sub-state* of  $\langle S \cdot s \cdot C \rangle$  (cf. the rules (AppLeft) and (AppRight)).

The simplest cases are when everything has been computed (rule (Res)) and when computations have not even started (rule (Code)). Things get more interesting in the intermediate situation where a computation is being performed. Consider a state  $D = \langle AS \cdot e \cdot C; @ \rangle$ . We get  $\Phi(D) = \langle S \cdot s \cdot C; @ \rangle$  and thus  $\overline{D} = (P_1 P_2)$ . If  $P_2$  is not fully reduced yet, then it is the decompilation of a sub-state of  $D$ , whereas  $P_1$  is  $N [s]$  where  $N$  is an instruction (here a  $\lambda_{DB}$ -term) whose execution has not even started (rule (AppRight)). If  $P_2$  is a reduced term, then it stands at the bottom of the stack  $S : P_2$  (i.e.  $M_2 = P_2$ ), while the term  $P_1$  is the decompilation of a sub-state of  $D$  (rule (AppLeft)). The last rule (State), which performs the decompilation of multi-frame states, is inspired by the transition **dump**.

In the next section we prove that, given a state  $D$  such that  $\overline{D}$  exists, then, for any state  $D'$  that can be computed from  $D$  by the SECD machine, the  $\lambda\sigma$ -term  $\overline{D}'$  also exists. At present, we just state two simple results on judgments and proof trees:

*Lemma 5*

Let  $D$  be a state of the SECD machine, such that  $\overline{D}$  exists. Then the following properties hold:

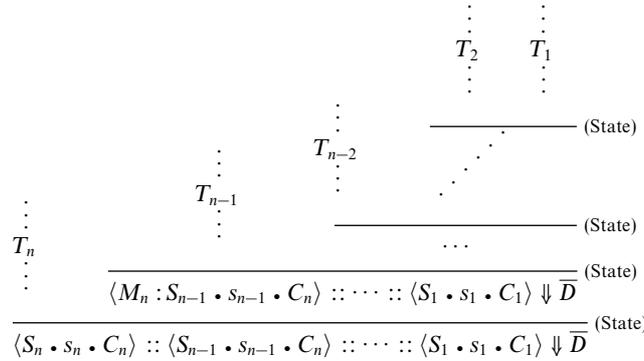


Fig. 4. Structure of the proof of  $\Phi(D) \Downarrow \bar{D}$ .

1. Let  $\langle S . s . C \rangle \Downarrow M$  be a judgment that appears in the proof tree of  $\Phi(D) \Downarrow \bar{D}$ . Then, all the  $\lambda\sigma$ -terms in  $S$  except, possibly, the topmost one are L-values.
2. The  $\lambda\sigma$ -term  $\bar{D}$  is unique.

*Proof*

The first proposition is easy, once one understands the structure of the proof of  $\Phi(D) \Downarrow \bar{D}$ . Consider a state  $D = \langle AS_n . e_n . C_n \rangle :: \langle AS_{n-1} . e_{n-1} . C_{n-1} \rangle :: \dots :: \langle AS_1 . e_1 . C_1 \rangle$ , we get  $\Phi(D) = \langle S_n . s_n . C_n \rangle :: \langle S_{n-1} . s_{n-1} . C_{n-1} \rangle :: \dots :: \langle S_1 . s_1 . C_1 \rangle$ . The proof tree of  $\Phi(D) \Downarrow \bar{D}$  is schematized in figure 4. In this figure,  $T_n$  stands for a proof tree whose conclusion is  $\langle S_n . s_n . C_n \rangle \Downarrow M_n$ . The only inference rules that can appear inside  $T_n$  are either rules (AppRight) or rules (AppLeft). In the first case, the stack component of the premise is the same as the stack component of the conclusion. In the second case, the stack component of the premise is built by taking all the elements of the stack component of the conclusion except the bottom one. Thus, any judgment  $\langle S . s . C \rangle \Downarrow M$  occurring inside  $T_n$  is such that  $S$  is a prefix of  $S_n$ . Therefore, since  $S_n$  is  $\overline{AS_n}$ , all the  $\lambda\sigma$ -terms in  $S$  are L-values. Moreover, given an integer  $i \in [1 \dots n - 1]$ , the proof tree  $T_i$  admits the judgment  $\langle M_{i+1} : S_i . s_i . C_i \rangle \Downarrow M_i$  as its conclusion. Thus, given any judgment  $\langle S . s . C \rangle \Downarrow M$  occurring inside  $T_i$ , the stack  $S$  is a prefix of  $M_{i+1} : S_i$ , where  $S_i$  is  $\overline{AS_i}$ . Therefore, all the  $\lambda\sigma$ -terms in  $S$  except, possibly, the first one are L-values. The  $\lambda\sigma$ -term  $M_{i+1}$  is a decompilation result. In general, this is not a L-value.

The second proposition follows from a more general result: given any triple  $\langle S . s . C \rangle$ , there exists at most one proof tree of a judgment  $\langle S . s . C \rangle \Downarrow M$ . Ambiguity may only occur when  $C = C'; @$  and we just need to check that the decompilation rules (AppLeft) and (AppRight) may not apply simultaneously. Otherwise, there would exist a state  $\langle S . s . C'; @ \rangle = \langle S' : M'_2 . s . C''; N''; @ \rangle$ , such that the following two proof nodes hold:

$$\frac{\langle S' . s . C''; N'' \rangle \Downarrow M'_1}{\langle S' : M'_2 . s . C''; N''; @ \rangle \Downarrow (M'_1 M'_2)} \text{ (AppLeft)}$$

$$\frac{\langle S' : M'_2 . s . C'' \rangle \Downarrow M''}{\langle S' : M'_2 . s . C''; N''; @ \rangle \Downarrow (N'' [s] M'')} \text{ (AppRight)}$$

However, the judgment  $\langle S' \cdot s \cdot C''; N'' \rangle \Downarrow M'_1$  can only be proved by the axiom (Code), because  $N''$  is a  $\lambda_{\text{DB}}$ -term and that no other rule applies in this case. Thus, we get  $S' = ()$ , which is impossible because of the side condition of (AppLeft)  $S' \neq ()$ .  $\square$

In the following, we assume that all the proof trees we consider are proofs of judgments  $\Phi(D) \Downarrow \bar{D}$ . Thus they have the structure pictured in figure 4, where all the stacks in  $T_n$  hold L-values and all the elements except, possibly, the topmost in the stacks in  $T_{n-1}, \dots, T_1$  are L-values.

### 4.3 Correctness

In this section, we show the correctness of the SECD machine by establishing a bisimulation between this machine and a strategy in the  $\lambda\sigma$ -calculus. Thus, we review the conditions of section 2.3, one after the other:

*Lemma 6 (Initial condition)*

Let  $N$  be a  $\lambda_{\text{DB}}$ -term. Then, we have  $\overline{\mathcal{L}(N)} = N [\text{id}]$

*Proof*

Quite straightforward, since we have  $\mathcal{L}(M) = \langle () \cdot () \cdot N \rangle$  and thus we get  $\langle () \cdot \bar{()} \cdot N \rangle \Downarrow N [\text{id}]$  by the decompilation rule (Code).  $\square$

Our idea is first to guess the strategy of the SECD machine (the L-strategy), and then to prove formally that the SECD implements the L-strategy.

We guess the axioms of the L-strategy, by considering some simple SECD transitions  $D \rightarrow D'$ . First consider the state  $D = \langle () \cdot e \cdot (N_1 \ N_2) \rangle$ , by the decompilation rule (Code), we get  $\bar{D} = (N_1 \ N_2) [\bar{e}]$ . Moreover, we have the SECD transition  $\langle () \cdot e \cdot (N_1 \ N_2) \rangle \xrightarrow{\text{app}} \langle () \cdot e \cdot N_2; N_1; @ \rangle$ . The  $\lambda\sigma$ -term  $\bar{D}' = (N_1 [\bar{e}] \ N_2 [\bar{e}])$  is computed by the following proof tree:

$$\frac{\langle () \cdot \bar{e} \cdot N_2 \rangle \Downarrow N_2 [\bar{e}] \text{ (Code)}}{\langle () \cdot \bar{e} \cdot N_2; N_1; @ \rangle \Downarrow (N_1 [\bar{e}] \ N_2 [\bar{e}]) \text{ (AppRight)}}$$

Hence we get the strategy axiom

$$(N_1 \ N_2) [s] \xrightarrow{\text{L}} (N_1 [s] \ N_2 [s]) \text{ (App)}$$

Similarly, from the transitions var and @, we guess the following two extra axioms:

$$\text{n} [M_1 \cdot M_2 \cdots M_n \cdot s] \xrightarrow{\text{L}} M_n \text{ (Var}^n\text{)}$$

$$\frac{M \text{ is a L-value}}{((\lambda N) [s] \ M) \xrightarrow{\text{L}} N [M \cdot s]} \text{ (Beta)}$$

Notice that the remaining two transitions **lam** and **dump** do not suggest any new axiom, since, for these transitions  $D \rightarrow D'$ , we get  $\bar{D} = \bar{D}'$ . Therefore, these two transitions are silent transitions.

Our presentation of the axiom (Beta) highlights an important point: at the time of function application, the argument  $M$  is not any  $\lambda\sigma$ -term, it is a L-value.

Then, we guess the inference rules of the L-strategy. We consider a state  $D = \langle AS \cdot e \cdot C; N; @ \rangle$  such that  $D \rightarrow D'$  with  $D' = \langle AS' \cdot e \cdot C'; N; @ \rangle$  (i.e, the code  $C$  is not empty and we do not consider the case of a transition  $@$ ). Thus we get:

$$\frac{\langle \overline{AS} \cdot \bar{e} \cdot C \rangle \Downarrow M}{\langle \overline{AS} \cdot \bar{e} \cdot C; N; @ \rangle \Downarrow (N [\bar{e}] M)} \text{ (AppRight)}$$

and

$$\frac{\langle \overline{AS'} \cdot \bar{e} \cdot C' \rangle \Downarrow M'}{\langle \overline{AS'} \cdot \bar{e} \cdot C'; N; @ \rangle \Downarrow (N [\bar{e}] M')} \text{ (AppRight)}$$

Assume that the reduction  $M \xrightarrow{L} M'$  holds. We get a first context rule:

$$\frac{M_2 \xrightarrow{L} M'_2}{(M_1 M_2) \xrightarrow{L} (M_1 M'_2)} \text{ (AppRight)}$$

By considering the case where both  $\lambda\sigma$ -terms  $\overline{D}$  and  $\overline{D}'$  are computed using the decompilation rule (AppLeft), we get a second context rule:

$$\frac{M_1 \xrightarrow{L} M'_1 \text{ and } M_2 \text{ is a L-value}}{(M_1 M_2) \xrightarrow{L} (M'_1 M_2)} \text{ (AppLeft)}$$

Here again, we enforce the condition that the argument  $M_2$  is a L-value, to ensure that the L-strategy is deterministic.

The rule (State) plugs the term  $M_n$  into the hole  $X$  of the context  $\langle X : S_{n-1} \cdot s_{n-1} \cdot C_{n-1} \rangle :: \dots :: \langle S_1 \cdot s_1 \cdot C_1 \rangle$ . The following lemma shows that this combination of subterm and context is compatible with the L-strategy.

*Lemma 7*

Consider any provable judgment  $\langle P : S \cdot s \cdot C \rangle :: D \Downarrow M$  and any  $\lambda\sigma$ -term  $P'$ , such that  $P \xrightarrow{L} P'$ . Then, there exists a  $\lambda\sigma$ -term  $M'$  such that the judgment  $\langle P' : S \cdot s \cdot C \rangle :: D \Downarrow M'$  holds and we have  $M \xrightarrow{L} M'$ .

*Proof*

Simple induction on the proof of  $\langle P : S \cdot s \cdot C \rangle :: D \Downarrow M$ .

1. The base case of rule (Res) is obvious.
2. In the case of the rule (AppRight), we have:

$$\frac{\langle P : S \cdot s \cdot C \rangle \Downarrow M}{\langle P : S \cdot s \cdot C; N; @ \rangle \Downarrow (N [s] M)}$$

By induction hypothesis and by the decompilation rule (AppRight), there exists  $M'$ , such that we get:

$$\frac{\langle P' : S \cdot s \cdot C \rangle \Downarrow M'}{\langle P' : S \cdot s \cdot C; N; @ \rangle \Downarrow (N [s] M')}$$

Hence the result, by the rule (AppRight) of the L-strategy.

3. In the case of the decompilation rule (AppLeft), by a similar argument, we get:

$$\frac{\langle P : S \cdot s \cdot C \rangle \Downarrow M_1}{\langle P : S : M_2 \cdot s \cdot C ; @ \rangle \Downarrow (M_1 \ M_2)} \quad \text{and} \quad \frac{\langle P' : S \cdot s \cdot C \rangle \Downarrow M'_1}{\langle P' : S : M_2 \cdot s \cdot C ; @ \rangle \Downarrow (M'_1 \ M_2)}$$

Observe that  $M_2$  cannot be the topmost element of the stack  $P : S : M_2$ . Therefore,  $M_2$  is a L-value (cf. lemma 5-1). Hence the result, by the rule (AppLeft) of the L-strategy.

4. In the case of the rule (State), assuming  $D = \langle S' \cdot s' \cdot C' \rangle :: D'$ , we have:

$$\frac{\langle P : S \cdot s \cdot C \rangle \Downarrow M_1 \quad \langle M_1 : S' \cdot s' \cdot C' \rangle :: D' \Downarrow M}{\langle P : S \cdot s \cdot C \rangle :: D \Downarrow M}$$

By induction hypothesis, there exists  $M'_1$ , such that  $\langle P' : S \cdot s \cdot C \rangle \Downarrow M'_1$  holds and  $M_1 \xrightarrow{L} M'_1$ . Therefore, still by induction hypothesis, there exists  $M'$ , such that  $\langle M'_1 : S' \cdot s' \cdot C' \rangle :: D' \Downarrow M'$  holds and  $M \xrightarrow{L} M'$ . Hence the result, since  $\langle P' : S \cdot s \cdot C \rangle :: D \Downarrow M'$  holds, by the decompilation rule (State).  $\square$

Then, our idea is to interpret the instruction to be executed next as a  $\lambda\sigma$ -term to be plugged in the same context  $\langle X : S \cdot s \cdot C \rangle :: D$  that holds the terms  $P$  and  $P'$  in the previous lemma. In a first case, the instruction itself is a  $\lambda\sigma$ -term or, more precisely, a  $\lambda_{DB}$ -term.

#### Lemma 8

Let  $N$  be any  $\lambda_{DB}$ -term, such that the judgment  $\langle S \cdot s \cdot N ; C \rangle :: D \Downarrow M$  holds. Then the judgment  $\langle N [s] : S \cdot s \cdot C \rangle :: D \Downarrow M$  holds.

#### Proof

By induction on states. There are two base cases. First assume that  $C$  is empty. We get:

$$\langle () \cdot s \cdot N \rangle \Downarrow N [s] \text{ (Code)} \quad \langle N [s] \cdot s \cdot () \rangle \Downarrow N [s] \text{ (Res)}$$

The second base case is when  $C = @$ , we get:

$$\frac{\langle P \cdot s \cdot () \rangle \Downarrow P}{\langle P \cdot s \cdot N ; @ \rangle \Downarrow (N [s] \ P)} \text{ (AppRight)} \quad \frac{\langle N [s] \cdot s \cdot () \rangle \Downarrow N [s]}{\langle N [s] : P \cdot s \cdot @ \rangle \Downarrow (N [s] \ P)} \text{ (AppLeft)}$$

Then, we consider the inductive cases, first assuming that  $D$  is empty. That is, we consider a judgment  $\langle S \cdot s \cdot N ; C' ; @ \rangle \Downarrow M$ , where  $C'$  is not empty. We have two subcases:

- If we have the proof node:

$$\frac{\langle S \cdot s \cdot N ; C'' \rangle \Downarrow Q}{\langle S \cdot s \cdot N ; C'' ; P ; @ \rangle \Downarrow (P [s] \ Q)} \text{ (AppRight)}$$

Then, by induction hypothesis, the judgment  $\langle N [s] : S \cdot s \cdot C'' \rangle \Downarrow Q$  holds. Therefore, so does the judgment  $\langle N [s] : S \cdot s \cdot C'' ; P ; @ \rangle \Downarrow (P [s] \ Q)$ , by the inference rule (AppRight).

- The other case, where the judgments are proved by the rule (AppLeft), is similar.

Finally, consider the case where  $D = \langle S' . s' . C' \rangle :: D'$  is not empty. We have:

$$\frac{\langle S . s . N ; C \rangle \Downarrow P \quad \langle P : S' . s' . C' \rangle :: D' \Downarrow M}{\langle S . s . N ; C \rangle :: \langle S' . s . C' \rangle :: D' \Downarrow M} \text{ (State)}$$

By a simple inductive argument, the judgment  $\langle N [s] : S . s . C \rangle \Downarrow P$  hold and we get:

$$\frac{\langle N [s] : S . s . C \rangle \Downarrow P \quad \langle P : S' . s' . C' \rangle :: D' \Downarrow M}{\langle N [s] : S . s . C \rangle :: \langle S' . s . C' \rangle :: D' \Downarrow M} \text{ (State)}$$

□

When the next instruction to be executed is @, it must be given two arguments to be interpreted as an application node in  $\Lambda_\sigma$ .

*Lemma 9*

If  $\langle M_1 : M_2 : S . s . @ ; C \rangle :: D \Downarrow M$  holds, then  $\langle (M_1 M_2) : S . s . C \rangle :: D \Downarrow M$  also holds.

*Proof*

Easy induction on proof trees. □

Now we show that the SECD machine indeed follows the L-strategy.

*Lemma 10*

Let  $D$  and  $D'$  be two states of the SECD machine such that  $\bar{D}$  exists and  $D \rightarrow D'$  by one transition step. Then,  $\bar{D}'$  exists and we have two possibilities:

1.  $\bar{D}' = \bar{D}$ , if  $\rightarrow$  is a transition **dump** or **lam**.
2.  $\bar{D} \xrightarrow{L} \bar{D}'$ , otherwise.

*Proof*

First consider the transition **dump**, which is special. We have  $D = \langle f . e . () \rangle :: \langle AS' . e' . C' \rangle :: D_0$  and  $D' = \langle f : AS' . e' . C' \rangle :: D_0$ . Computing  $\bar{D}$ , we have:

$$\frac{\langle \bar{f} . \bar{e} . () \rangle \Downarrow \bar{f} \quad \langle \bar{f} : \bar{AS}' . \bar{e}' . C' \rangle :: \Phi(D_0) \Downarrow M}{\langle \bar{f} . \bar{e} . () \rangle :: \langle \bar{AS}' . \bar{e}' . C' \rangle :: \Phi(D_0) \Downarrow M} \text{ (State)}$$

Observe that the right premise of the rule above is  $\Phi(D') \Downarrow M$ . In other words, we get  $\bar{D}' = \bar{D}$ .

All other transitions correspond to the execution of an instruction. First, consider the case of the instruction @. Thus, we state  $D = \langle (N_0/e_0) : f : AS . e . @ ; C \rangle :: D_0$  and  $D' = \langle () . f . e_0 . N_0 \rangle :: \langle AS . e . C \rangle :: D_0$ . On the one hand, by lemma 9, we get  $\langle ((\lambda N_0) [\bar{e}_0] \bar{f}) : \bar{AS} . \bar{e} . C \rangle :: \Phi(D_0) \Downarrow \bar{D}$ . On the other hand, by the compilation rule (State), we get  $\langle N_0 [\bar{f} . \bar{e}_0] : \bar{AS} . \bar{e} . C \rangle :: \Phi(D_0) \Downarrow \bar{D}'$ . Moreover, by the axiom (Beta) and since  $\bar{f}$  is a L-value, we get:

$$(((\lambda N_0) [\bar{e}_0] \bar{f}) \xrightarrow{L} N_0 [\bar{f} . \bar{e}_0])$$

Hence the result, by Lemma 7.

Then, in the three remaining cases, we have  $D = \langle AS . e . N ; C \rangle :: D_0$ , where  $N$  is a  $\lambda_{DB}$ -term.

1. If  $N$  is a variable  $n$  (i.e.,  $D = \langle AS \cdot e \cdot n; C \rangle :: D_0$  where  $e = f_1 \cdots f_n \cdot e_r$ ), then, on the one hand, by lemma 8, we get:

$$\langle n [\bar{e}] : \overline{AS} \cdot \bar{e} \cdot C \rangle :: \Phi(D_0) \Downarrow \bar{D}$$

On the other hand, we get:

$$\langle \bar{f}_n : \overline{AS} \cdot \bar{e} \cdot C \rangle :: \Phi(D_0) \Downarrow \bar{D}'$$

Hence the result, by the strategy axiom (Var<sup>n</sup>) and lemma 7.

2. If  $N$  is an abstraction  $\lambda N_0$ , then, on the one hand, by lemma 8, we get:

$$\langle (\lambda N_0) [\bar{e}] : \overline{AS} \cdot \bar{e} \cdot C \rangle :: \Phi(D_0) \Downarrow \bar{D}$$

Thus, since  $D' = \langle (N_0/e) : AS \cdot e \cdot C \rangle :: D_0$  and  $\overline{(N_0/e)} = (\lambda N_0) [\bar{e}]$ , we get  $\bar{D} = \bar{D}'$ .

3. If  $N$  is an application  $(N_1 N_2)$ , then, by lemma 8, we get:

$$\langle (N_1 N_2) [\bar{e}] : \overline{AS} \cdot \bar{e} \cdot C \rangle :: \Phi(D_0) \Downarrow \bar{D}$$

Here, we have  $D' = \langle AS \cdot e \cdot N_2; N_1; @; C \rangle :: D_0$ . Thus, by two applications of Lemma 8, first to  $N_2$  and then to  $N_1$ , followed by one application of Lemma 9, we get:

$$\langle (N_1 [\bar{e}] N_2 [\bar{e}]) : \overline{AS} \cdot \bar{e} \cdot C \rangle :: \Phi(D_0) \Downarrow \bar{D}'$$

Hence, the result, by the strategy axiom (App) and by lemma 7.  $\square$

Now, we prove the final Condition 3.

#### Lemma 11

Let  $N$  be  $\lambda_{DB}$ -term, and  $D$  be a terminal state, with  $\mathcal{L}(N) \rightarrow^* D$ . Then,  $\bar{D}$  is a L-normal form.

#### Proof

Let us state  $D = \langle AS_n \cdot e_n \cdot C_n \rangle :: \cdots :: \langle AS_1 \cdot e_1 \cdot C_1 \rangle$ . First observe that, by Lemma 10,  $\bar{D}$  exists. Then, by studying the SECD transitions, we distinguish three possibilities for  $D$  to be a terminal state:

1.  $n = 1$  and  $C_1 = ()$ , then, the stack  $AS_1$  holds exactly one closure  $f$  (Otherwise it cannot be decompiled) and we get  $\bar{D} = \overline{\langle f \cdot () \cdot () \rangle} = \bar{f}$ , which is a L-value and a L-normal form.
2.  $C_n = @; C'_n$  and the stack  $AS_n$  has zero or one element. In fact, this case cannot occur here, since  $\Phi(D) \Downarrow M$  holds. The proof of this judgment must include a proof  $\langle \overline{AS_n} \cdot \bar{e}_n \cdot @; C'_n \rangle \Downarrow M_n$ , which in turn must include a proof of  $\langle S \cdot \bar{e}_n \cdot @ \rangle \Downarrow P$ , where  $S$  is a stack with less elements than  $\overline{AS_n}$ . This latter judgment can be proved only by the rule (AppLeft). Thus, the stack  $S$  has at least two elements and so does the stack  $AS_n$ .

3.  $C_n = m; C'_n$  and  $e_n = f_1 \cdots f_k \cdot \text{id}$ , with  $k < m$ . Then,  $\bar{D}$  is a L-failure term  $W$ , where L-failure terms are defined as follows:

$$W ::= m [\bar{f}_1 \cdots \bar{f}_k \cdot \text{id}] \quad \text{with } k < m \\ | M W \quad | W \bar{f}$$

where  $M$  stands for a  $\lambda\sigma$ -term and  $\bar{f}$  for a L-value. The subterm  $m [\bar{f}_1 \cdots \bar{f}_k \cdot \text{id}]$  is a L-normal form which is not a L-value. It stands in L-redex position inside  $\bar{D}$ . Thus,  $\bar{D}$  is a L-normal form which is not a value. Moreover, the  $\sigma_{\uparrow}$ -normal form  $\sigma_{\uparrow}(\bar{D})$  is not a closed  $\lambda_{\text{DB}}$ -term, since it admits at least one free variable  $m-k = \sigma_{\uparrow}(m [\bar{f}_1 \cdots \bar{f}_k \cdot \text{id}])$ . Therefore, by Lemma 1, this case may only occur when the initial program is not a closed  $\lambda_{\text{DB}}$ -term.  $\square$

*Lemma 12*

The SECD cannot perform infinitely many consecutive silent transitions.

*Proof*

Let  $\mathcal{S}$  be a measure on states, code segments and  $\lambda_{\text{DB}}$ -terms, defined as follows:

$$\begin{aligned} \mathcal{S}(\langle AS_n \cdot s_n \cdot C_n \rangle :: \cdots :: \langle AS_1 \cdot s_1 \cdot n \rangle) &= n + \mathcal{S}(C_1) + \cdots + \mathcal{S}(C_n) \\ \mathcal{S}(@; C) &= \mathcal{S}(C) \\ \mathcal{S}(N; C) &= \mathcal{S}(N) + \mathcal{S}(C) \\ \mathcal{S}() &= 0 \\ \mathcal{S}(N_1 N_2) &= 1 + \mathcal{S}(N_1) + \mathcal{S}(N_2) \\ \mathcal{S}(\lambda N) &= 1 \\ \mathcal{S}(\mathfrak{n}) &= 1 \end{aligned}$$

Now, given a transition  $D \rightarrow D'$  that is not **app**, we have  $\mathcal{S}(D) > \mathcal{S}(D')$ . Thus, any computation of the SECD that does not include the transition **app** must be finite. Since the transition **app** corresponds to the rewriting axiom (Beta), it is not silent. Hence the result.  $\square$

Finally, we conclude:

*Theorem 2*

The SECD machine implements the L-strategy.

## 5 The Functional Abstract Machine

### 5.1 Basics

The Functional Abstract Machine (FAM) was designed by Cardelli (Cardelli, 1984) as a ‘‘SECD machine optimized to allow very fast function application and the use of true stack’’.

The FAM has four instructions:

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

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

$$\begin{aligned} \text{STACK} & ::= () \quad | \quad \text{CLOSURE} : \text{STACK} \\ \text{FRAME} & ::= \langle \text{STACK} \cdot \text{ENVIRONMENT} \cdot \text{CODE} \rangle \end{aligned}$$

The transitions rules of the FAM are defined as follows:

$$\begin{aligned} \langle AS : f \cdot e \cdot \text{Local}; C \rangle :: D & \xrightarrow{\text{local}} \langle f : AS : f \cdot e \cdot C \rangle :: D \\ \langle AS \cdot \vec{f}_{1,n} \cdot \text{Global}(i); C \rangle :: D & \xrightarrow{\text{global}} \langle f_i : AS \cdot \vec{f}_{1,n} \cdot C \rangle :: D \\ \langle (C_0/e_0) : g : AS \cdot e \cdot \text{Apply}; C \rangle :: D & \xrightarrow{\text{apply}} \langle g \cdot e_0 \cdot C_0 \rangle :: \langle AS \cdot e \cdot C \rangle :: D \\ \langle \vec{f}_{n,1} : AS \cdot e \cdot \text{Fun}(n, C_0); C \rangle :: D & \xrightarrow{\text{closure}} \langle (C_0/\vec{f}_{1,n}) : AS \cdot e \cdot C \rangle :: D \\ \langle f : \_ \cdot \_ \cdot () \rangle :: \langle AS \cdot e \cdot C \rangle :: D & \xrightarrow{\text{return}} \langle f : AS \cdot e \cdot C \rangle :: D \end{aligned}$$

Specifically, observe how the instruction `Local` selects the bottom element  $f$  of the argument stack  $AS : f$  and how the  $n$  arguments  $\vec{f}_{n,1}$  that the instruction `Fun`( $n, C_0$ ) pops are taken in reverse order to build a new environment  $\vec{f}_{1,n}$ . 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’).

Cardelli (1984) only gives a few ‘compilation hints’ for the FAM. One of these hints consists in compiling a function  $\lambda N$  as a closure  $(C/e)$ , where the environment  $e$  has been optimized to retain only the global variables of  $\lambda N$ . This idea is described as a simple transformation in  $\lambda\sigma$ . For instance the  $\lambda$ -abstraction  $N = \lambda(1 (5 7))$  is transformed into the  $\lambda\sigma$ -closure  $M = (\lambda(1 (2 3))) [4 \cdot 6 \cdot \text{id}]$ . That is, the free variables 5 and 7 are ‘abstracted out’ or ‘lifted’ and regrouped in the closure environment  $4 \cdot 6 \cdot \text{id}$ , whereas in the body of  $M$ , the lifted variables 5 and 7 are replaced by 2 and 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  $N [\text{id}]$  and  $M$  are the same function. For any argument  $P$ , we get:

$$\begin{aligned} (N [\text{id}] P) & \xrightarrow{(\text{Beta})} (1 (5 7)) [P \cdot \text{id}] \xrightarrow{\sigma_w} P \quad (4 \ 6) \\ M P & \xrightarrow{(\text{Beta})} (1 (2 3)) [P \cdot 4 \cdot 6 \cdot \text{id}] \xrightarrow{\sigma_w} P \quad (4 \ 6) \end{aligned}$$

We now describe our general free variable abstraction procedure. The set  $\mathcal{F}_0(N) = \{n_1, \dots, n_m\}$  of the free variables in a  $\lambda_{\text{DB}}$ -term  $N$  can be arbitrarily ordered as a list  $\vec{n}_{1,m} = n_1 : \dots : n_m$ . This list is then given as a first argument to our abstraction scheme  $\mathcal{C}$ , which, given any  $\lambda_{\text{DB}}$ -term  $N$ , outputs a  $\lambda\sigma$ -term  $\mathcal{C}(\mathcal{F}_0(N), N)$ .

$$\begin{aligned} \mathcal{C}(\vec{n}_{1,m}, \mathbf{i}) & = \mathbf{i} \\ \mathcal{C}(\vec{n}_{1,m}, (N_1 N_2)) & = (\mathcal{C}(\vec{n}_{1,m}, N_1) \mathcal{C}(\vec{n}_{1,m}, N_2)) \\ \mathcal{C}(\vec{n}_{1,m}, \lambda N) & = \left\{ \begin{array}{l} (\lambda \mathcal{C}(1 : p_1 + 1 : \dots : p_k + 1, N)) [\mathcal{C}(\vec{n}_{1,m}, p_1 \cdots p_k \cdot \text{id})] \\ \text{where } p_1 : \dots : p_k = \mathcal{F}_0(\lambda N) \end{array} \right. \\ \mathcal{C}(\vec{n}_{1,m}, N \cdot s) & = \mathcal{C}(\vec{n}_{1,m}, N) \cdot \mathcal{C}(\vec{n}_{1,m}, s) \\ \mathcal{C}(\vec{n}_{1,m}, \text{id}) & = \uparrow^m \end{aligned}$$

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  $s = M_1 \cdots M_i \cdots M_m \cdot \text{id}$ , where  $M_i$  is the run-time value of the variable  $n_i$  of the input term. The translation  $(\lambda M) [t]$  of a  $\lambda$ -abstraction  $\lambda N$  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  $\lambda N$ . 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:

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

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

$$\begin{array}{ccc} \uparrow^m \circ (M_1 \cdot M_2 \cdots M_m \cdot \text{id}) & \xrightarrow{(\text{AssEnv})} & \\ \uparrow \circ (\uparrow^{m-1} \circ (M_1 \cdot M_2 \cdots M_m \cdot \text{id})) & \xrightarrow{\sigma_w^*} & \dots \\ & \xrightarrow{(\text{ShiftCons})} & \text{id} \end{array}$$

The correctness of the procedure  $\mathcal{C}$  with respect to the substitution rules of the full (strong)  $\sigma_{\uparrow}$ -calculus can be stated quite simply. We first prove two technical lemmas on free variables and substitutions.

*Lemma 13*

For any  $\lambda_{\text{DB}}$ -term  $N$  and any integer  $d$ , we have the following implication:

$$n \in \mathcal{F}_d(N) \quad \Rightarrow \quad n = 1 \text{ or } n - 1 \in \mathcal{F}_{d+1}(N)$$

*Proof*

By induction on  $N$ :

- If  $N = n$ , then we have three subcases. If  $n \leq d$ , then  $\mathcal{F}_d(N) = \mathcal{F}_{d+1}(N) = \emptyset$ . If  $n = d + 1$  then  $\mathcal{F}_d(N) = \{1\}$  and  $\mathcal{F}_{d+1}(N) = \emptyset$ . If  $n > d + 1$ , then  $\mathcal{F}_d(N) = \{n - d\}$  and  $\mathcal{F}_{d+1}(N) = \{n - d - 1\}$ .
- If  $N = (N_1 N_2)$ , then we get the result by direct induction.
- If  $N = \lambda N_0$ , then, by definition, we get  $\mathcal{F}_d(N) = \mathcal{F}_{d+1}(N_0)$  and  $\mathcal{F}_{d+1}(N) = \mathcal{F}_{d+2}(N_0)$ . Hence the result, since we have  $n \in \mathcal{F}_{d+1}(N_0) \Rightarrow n = 1 \text{ or } n - 1 \in \mathcal{F}_{d+2}(N_0)$  by induction hypothesis.  $\square$

*Lemma 14*

Let  $N$  be a  $\lambda_{\text{DB}}$ term. The following  $\sigma_{\uparrow}$  equality holds, for any substitution  $s$  and integer  $d$ :

$$N [\uparrow^d (s)] =_{\sigma_{\uparrow}} N [1 \cdot 2 \cdots d \cdot (s \circ \uparrow^d)]$$

(Where  $\uparrow^d (s)$  stands for  $\overbrace{\uparrow (\cdots \uparrow (s) \cdots)}^{d \text{ times}}$ )

*Proof*

A simple proof by induction on  $N$  is possible. In fact, as exposed in Curien *et al.* (1996, Sections 3.2 and 4.1),  $\uparrow^d(s)$  and  $1 \cdot 2 \cdots d \cdot (s \circ \uparrow^d)$  perform the same replacements on ground terms. Operationally, both substitutions behave as  $s$  after it went through  $d$  nested levels of  $\lambda$ 's.  $\square$

*Proposition 1 (Correctness of compilation)*

The  $\mathcal{C}$  compilation scheme never fails and is correct. More precisely, given a  $\lambda_{\text{DB}}$ -term  $N$ , let  $\vec{n}_{1,m}$  such that  $\mathcal{F}_0(N) \subseteq \vec{n}_{1,m}$ . Then  $\mathcal{C}(\vec{n}_{1,m}, N)$  is a  $\lambda\sigma$ -term  $M$ , such that  $M[\mathbf{n}_1 \cdots \mathbf{n}_m \cdot \text{id}]$  and  $N$  are  $\sigma_{\uparrow}$ -equivalent.

*Proof*

We prove the following proposition: for any  $\lambda_{\text{DB}}$ -term  $N$  and any vector of indices  $\vec{n}_{1,m}$ , such that  $\mathcal{F}_0(N) \subseteq \vec{n}_{1,m}$ , the  $\lambda\sigma$ -term  $M = \mathcal{C}(\vec{n}_{1,m}, N)$  exists. Moreover, given any substitution  $s$ , we have the following conversion:

$$M[\mathbf{n}_1 \cdots \mathbf{n}_m \cdot s] \xrightarrow{\sigma_{\uparrow}^*} N$$

The proof is by induction on  $N$ . In the proof, we state  $t(\vec{n}_{1,m}, s) = \mathbf{n}_1 \cdots \mathbf{n}_m \cdot s$ .

- If  $N$  is a variable  $\mathbf{n}$ , then, by hypothesis,  $\mathbf{n}$  is one of the  $n_i$  and we have  $\mathcal{C}(\vec{n}_{1,m}, N) = \mathbf{i}$ . Furthermore, by the  $\sigma_w$ -rules (RVar) and (FVar), we get

$$\mathbf{i}[t(\vec{n}_{1,m}, s)] \xrightarrow{\sigma_w^*} \mathbf{n}_i$$

- If  $N = (N_1 N_2)$ , then, by definition of  $\mathcal{F}$ , we have  $\mathcal{F}_0(N) = \mathcal{F}_0(N_1) \cup \mathcal{F}_0(N_2)$ , and thus  $\mathcal{F}_0(N_1) \subseteq \vec{n}_{1,m}$  and  $\mathcal{F}_0(N_2) \subseteq \vec{n}_{1,m}$ . Thus, we can apply the induction hypothesis and both  $M_1 = \mathcal{C}(\vec{n}_{1,m}, N_1)$  and  $M_2 = \mathcal{C}(\vec{n}_{1,m}, N_2)$  exist. So does  $M = \mathcal{C}(\vec{n}_{1,m}, N) = (M_1 M_2)$ . Furthermore, we have

$$M[t(\vec{n}_{1,m}, s)] \xrightarrow{\sigma_{\uparrow}^*} N$$

by the  $\sigma_w$ -rule (App) and by induction hypothesis.

- If  $N = \lambda N_0$ , then let  $\vec{p}_{1,k}$  be  $\mathcal{F}_0(N)$ , the set of the free variables of  $N$ , expressed as De Bruijn indices with respect to the scope of  $N$  itself.

By definition of  $\mathcal{F}$ , we have  $\vec{p}_{1,k} = \mathcal{F}_1(N_0)$  and thus, by lemma 13, we have  $\mathcal{F}_0(N_0) \subseteq 1 : p_1 + 1 : \cdots : p_k + 1$ . Therefore,  $M_0 = \mathcal{C}(1 : p_1 + 1 : \cdots : p_k + 1, N_0)$  exists, by induction hypothesis. Now, all the variables in the substitution  $u = \mathbf{p}_1 \cdots \mathbf{p}_k \cdot \text{id}$  belong to  $\mathcal{F}_0(N)$ . Thus they can be compiled in the compile-time environment  $\vec{n}_{1,m} \supseteq \mathcal{F}_0(N)$  and so does the substitution  $u$ . Let  $u_0$  be  $\mathcal{C}(\vec{n}_{1,m}, u)$ . Finally, the  $\lambda\sigma$ -term  $M = (\lambda N_0)[u_0]$  exists.

Furthermore, since we have  $\mathcal{C}(\vec{n}_{1,m}, \mathbf{p}_i)[t(\vec{n}_{1,m}, s)] \xrightarrow{\sigma_{\uparrow}^*} \mathbf{p}_i$  (by induction) and  $\uparrow^m \circ t(\vec{n}_{1,m}, s) = s$  (by  $\sigma_w$ -reduction), we get:

$$u_0 \circ t(\vec{n}_{1,m}, s) \xrightarrow{\sigma_{\uparrow}^*} \mathbf{p}_1 \cdots \mathbf{p}_k \cdot s$$

Thus, we get the following  $\sigma_{\uparrow}$ -conversions:

$$M[t(\vec{n}_{1,m}, s)] \xrightarrow{(\text{Clos})} (\lambda M_0)[u_0 \circ t(\vec{n}_{1,m}, s)] \xrightarrow{\sigma_{\uparrow}^*} (\lambda M_0)[\mathbf{p}_1 \cdots \mathbf{p}_k \cdot s]$$

Now, by the strong substitution rule (Lambda), we get

$$M_0 [t(\vec{n}_{1,m}, s)] \xrightarrow{\sigma_{\uparrow}}^* \lambda(M_0 [\uparrow (p_1 \cdots p_k \cdot s)])$$

Let us consider  $N'_0$ , the  $\sigma_{\uparrow}$ -normal form of  $M_0$ , which is a  $\lambda_{\text{DB}}$ -term (Curien *et al.*, 1996)[Lemma 4.8]. By definition of reduction, we get:

$$M_0 [t(\vec{n}_{1,m}, s)] \xrightarrow{\sigma_{\uparrow}}^* \lambda(N'_0 [\uparrow (p_1 \cdots p_k \cdot s)])$$

Our Lemma 14 applies here and we get:

$$M_0 [t(\vec{n}_{1,m}, s)] =_{\sigma_{\uparrow}} \lambda(N'_0 [1 \cdot ((p_1 \cdots p_k \cdot s) \circ \uparrow)])$$

Hence, by the  $\sigma_{\uparrow}$ -rules (MapEnv) and (VarShift1), on the one hand we finally get:

$$M_0 [t(\vec{n}_{1,m}, s)] =_{\sigma_{\uparrow}} \lambda(N'_0 [1 \cdot p_1 + 1 \cdots p_k + 1 \cdot (s \circ \uparrow)]) \quad (1)$$

On the other hand, by application of the induction hypothesis to  $N_0$  we get:

$$M_0 [1 \cdot p_1 + 1 \cdots p_k + 1 \cdot (s \circ \uparrow)] \xrightarrow{\sigma_{\uparrow}}^* N_0$$

Therefore, by the Church–Rosser property and since  $N_0$  is a  $\sigma_{\uparrow}$  normal form, we get:

$$N'_0 [1 \cdot p_1 + 1 \cdots p_k + 1 \cdot (s \circ \uparrow)] \xrightarrow{\sigma_{\uparrow}}^* N_0 \quad (2)$$

Finally, still by the Church–Rosser property, from (1) and (2) above, we conclude:

$$M [t(\vec{n}_{1,m}, s)] \xrightarrow{\sigma_{\uparrow}}^* \lambda N_0 = N$$

□

It is important to notice that the  $\lambda\sigma$ -terms  $N$  and  $M [\text{id}]$  only differ by *substitution steps*. As a consequence,  $M [\text{id}]$  and  $N$  are more than just  $\beta$ -equivalent  $\lambda$ -terms, they are the same  $\lambda$ -term.

#### Corollary 15

Let  $N$  be a closed  $\lambda_{\text{DB}}$ -term (i.e., a program) and  $M$  be its compilation  $\mathcal{C}(\emptyset, N)$ . The initial condition  $M [\text{id}] =_{\sigma_{\uparrow}} N$  holds.

The output  $M$  of  $\mathcal{C}$  is not just any  $\lambda\sigma$ -term. First, all substitutions in  $M$  are of the general form  $s = M_1 \cdot M_2 \cdots M_m \cdot \uparrow^k$ . The integer  $m$  is the length of substitution  $s$  and we write  $m = \text{length}(s)$ . Second,  $M = \mathcal{C}(\vec{n}_{1,k}, N)$  itself can be characterized by a predicate  $\mathcal{P}_k$  that is defined as follows:

$$\begin{aligned} \mathcal{P}_k(\mathbf{n}) &= (n \leq k) \\ \mathcal{P}_k(M_1 M_2) &= \mathcal{P}_k(M_1) \wedge \mathcal{P}_k(M_2) \\ \mathcal{P}_k((\lambda M) [s]) &= \mathcal{P}_{l_s+1}(M) \wedge \mathcal{P}_k(s), \text{ where } l_s = \text{length}(s) \\ \mathcal{P}_k(M) &= \text{false} \text{ otherwise} \\ \mathcal{P}_k(M \cdot s) &= \mathcal{P}_k(M) \wedge \mathcal{P}_k(s) \\ \mathcal{P}_k(\uparrow^n) &= (n = k) \\ \mathcal{P}_k(s) &= \text{false} \text{ otherwise} \end{aligned}$$

Intuitively,  $\mathcal{P}_k(M)$  holds, when  $M$  is to be evaluated with respect to an environment of size  $k$ . If  $N$  is a closed  $\lambda_{\text{DB}}$ -term, then we get  $\mathcal{P}_0(\mathcal{C}(\emptyset, N))$ .

Let  $M$  be a  $\lambda\sigma$ -term that is a result of the first compilation procedure  $\mathcal{C}$ . Giving  $M$  as input to the second compilation procedure  $\llbracket \cdot \rrbracket$  generates FAM code.

$$\begin{aligned} \llbracket 1 \rrbracket &= \text{Local} \\ \llbracket n + 1 \rrbracket &= \text{Global}(n) \\ \llbracket (M_1 M_2) \rrbracket &= \llbracket M_2 \rrbracket; \llbracket M_1 \rrbracket; \text{Apply} \\ \llbracket (\lambda M_0) [M_1 \cdots M_n \cdot \uparrow^m] \rrbracket &= \llbracket M_1 \rrbracket; \dots; \llbracket M_n \rrbracket; \text{Fun}(n, \llbracket M_0 \rrbracket) \end{aligned}$$

Finally, a closed  $\lambda_{\text{DB}}$ -term  $N$  is compiled first to the term  $M = \mathcal{C}(\emptyset, N)$  and then to the code  $C = \llbracket M \rrbracket$ . Execution starts from the initial state  $\mathcal{L}(M) = \langle () \cdot () \cdot C \rangle$ .

## 5.2 Decompileation

First, we inverse the compilation procedure  $\llbracket \cdot \rrbracket$ . We do so by proving judgments  $C \Downarrow^m M$ , which read “the code segment  $C$  stands for the  $\lambda\sigma$ -term  $M$  in an environment of size  $m$ ”.

$$\begin{array}{c} \text{Local } \Downarrow^m 1 \text{ (Local)} \qquad \text{Global}(i) \Downarrow^m i+1 \text{ (Global)} \\ \frac{C_2 \Downarrow^m M_2 \quad C_1 \Downarrow^m M_1}{C_2; C_1; \text{Apply } \Downarrow^m (M_1 M_2)} \text{ (Apply)} \\ \frac{C_1 \Downarrow^m M_1 \quad \cdots \quad C_n \Downarrow^m M_n \quad C_0 \Downarrow^{n+1} M_0}{C_1; \cdots; C_n; \text{Fun}(n, C_0) \Downarrow^m (\lambda M_0) [M_1 \cdots M_n \cdot \uparrow^m]} \text{ (Fun}(n)) \end{array}$$

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

$$\begin{array}{l} \text{Closures:} \quad \overline{(C/f_1 \cdot f_2 \cdots f_n)} = (\lambda M) \overline{[f_1 \cdot f_2 \cdots f_n]}, \text{ where } C \Downarrow^{n+1} M \\ \text{Environments:} \quad \overline{f_1 \cdot f_2 \cdots f_n} = \overline{f_1} \cdot \overline{f_2} \cdots \overline{f_n} \cdot \text{id} \\ \text{Stacks:} \quad \overline{f_n : \dots : f_2 : f_1} = \overline{f_n} : \dots : \overline{f_2} : \overline{f_1} \end{array}$$

Closures are the values of the FAM: they are expected as final results. Terms produced by decompiling closures are the counterparts of these results in  $\lambda\sigma$ . We call them C-values, they can be defined structurally:

$$\begin{array}{l} \text{C-VALUES:} \quad V ::= (\lambda M) [e] \quad \text{where } m = \text{length}(e) \text{ and } \mathcal{P}_{m+1}(M) \\ \text{C-ENVIRONMENTS:} \quad e ::= \text{id} \mid V \cdot e \end{array}$$

*Lemma 16*

Let  $M$  be a  $\lambda\sigma$ -term such that  $\mathcal{P}_m(M)$  holds. Then we have  $\llbracket M \rrbracket \Downarrow^m M$ .

*Proof*

Obvious induction on  $M$ .  $\square$

The decompilation of machine states is best understood as a two-stage process. In a first step, we translate the frames  $\langle AS \cdot e \cdot C \rangle$  into triples  $\langle S \cdot s \cdot C \rangle$ . Roughly,



*Lemma 17*

A decompilable code is any code segment  $C$ , such that there exists an integer  $m$  and a  $\lambda\sigma$ -term  $M$ , with  $C \Downarrow^m M$ . A strict suffix is any code segment  $C$ , 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  $\langle S \cdot s \cdot C \rangle \Downarrow M$ , 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.  $\square$

*Corollary 18*

Code decompilation is non-ambiguous.

*Proof*

Let  $m$  be an integer and  $C$  be a code. We show by induction on  $C$  that there does not exist two different terms  $M$  and  $M'$  such that  $C \Downarrow^m M$  and  $C \Downarrow^m M'$  hold.

- The base case  $C = \text{Local}$  or  $C = \text{Global}(i + 1)$  is straightforward.
- The code segment  $C$  ends by the instruction `Apply`. Assume there were two different decompositions  $C = C_2;C_1;\text{Apply}$  and  $C = C'_2;C'_1;\text{Apply}$ . Then, for instance,  $C'_1$  would be a decompilable suffix of  $C_1$ .
- A similar reasoning applies when  $C$  ends by the instruction `Fun`( $n, C_0$ ).  $\square$

*Lemma 19*

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.  $\Phi(D) = \langle S \cdot s \cdot C \rangle$ .

If the code  $C$  is empty or ends by a `Local` or `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 `Fun`( $n, C_0$ ) instruction. As far as ambiguity is concerned, these subcases are the same. Thus, for instance, we assume  $\Phi(D) = \langle S \cdot s \cdot C'; \text{Fun}(n, C_0) \rangle$ . If the stack  $S$  is empty, then only the rule (Code) may apply unambiguously (cf. Corollary 18). If  $S$  contains at least one element, then we must apply the decompilation rule (Fun( $n, i$ )). By the previous Lemma 17-2, there is at most one way to cut  $C'$  into  $C_i;C_{i+1};\dots;C_n$ , where  $C_i$  is a strict suffix and  $C_{i+1}, \dots, C_n$  are decompilable code segments. In other words, the rule (Fun( $n, i$ )) is applied unambiguously.  $\square$

Then, as we did in the case of the SECD machine and by the same easy argument, we see that all proof trees produced by decompiling real FAM states only contain judgments  $\langle S \cdot s \cdot C \rangle \Downarrow M$ , such that all the  $\lambda\sigma$ -terms in  $S$  are C-values, except, possibly, the topmost one. Furthermore, let  $D = \langle AS \cdot e \cdot C_0 \rangle :: D_0$  be a state and let be a judgment  $\langle S \cdot s \cdot C \rangle \Downarrow M$  that occurs inside the proof tree of  $\langle \overline{AS} \cdot \bar{e} \cdot C_0 \rangle \Downarrow P$ . Then *all* the  $\lambda\sigma$ -terms in  $S$  are C-values (see figure 4). From now on, we only consider proof trees that meet this constraint.

### 5.3 Strategy and correctness

In this section we show that our decompilation scheme meets the conditions of section 2.3.

*Lemma 20 (Initial state condition)*

Let  $N$  be a closed  $\lambda_{DB}$ -term. Let  $M$  be  $\mathcal{C}(\emptyset, N)$ . Then, we have the equality,

$$\overline{\mathcal{L}(M)} = M \text{ [id]}$$

*Proof*

By lemma 16, we have  $\llbracket M \rrbracket \Downarrow^0 M$ . Hence the result, by the decompilation rule (Code).

$$\frac{\llbracket M \rrbracket \Downarrow^m M}{\langle () \cdot \text{id} \cdot \llbracket M \rrbracket \rangle \Downarrow M \text{ [id]}} \text{ (Code)}$$

□

The rest of this section is devoted to the C-strategy that the FAM implements. By contrast with the previous section on the L-strategy and the SECD, we introduce the C-strategy gradually, to demonstrate how strategy rules are inferred from the correctness proof of the FAM. However, the pattern of the proof for the FAM and the C-strategy is the same as the one for the SECD and the L-strategy. Only our point of view changes, since we now infer the strategy instead of just checking it.

First, we examine proofs of judgments  $\langle P : S \cdot s \cdot C \rangle :: D \Downarrow M$ . Such judgments are introduced by the rule (State). Doing so, we infer some structural rule of the C-strategy from the structure of these proof trees. (The similar lemma for the SECD is Lemma 7.)

*Lemma 21*

Consider any two  $\lambda\sigma$ -terms  $P$  and  $P'$ . If the judgment  $\langle P : S \cdot s \cdot C \rangle :: D \Downarrow M$  holds for some  $\lambda\sigma$ -term  $M$ , then there exists a  $\lambda\sigma$ -term  $M'$ , such that judgment  $\langle P' : S \cdot s \cdot C \rangle :: D \Downarrow M'$  holds. Furthermore, given any relation  $\sim$ , such that  $P \sim P'$ , we have  $M \sim M'$ , provided  $\sim$  obeys the following structural rules:

$$\frac{M_2 \sim M'_2}{(M_1 M_2) \sim (M_1 M'_2)} \qquad \frac{M_2 \text{ is a C-value} \quad M_1 \sim M'_1}{(M_1 M_2) \sim (M_1 M'_2)}$$

$$\frac{M_1 \text{ is a C-value} \quad \cdots \quad M_{i-1} \text{ is a C-value} \quad M_i \sim M'_i}{(\lambda M_0) [M_1 \cdots M_{i-1} \cdot M_i \cdot s] \sim (\lambda M_0) [M_1 \cdots M_{i-1} \cdot M'_i \cdot s]}$$

*Proof*

Simple induction over the proof of  $\langle P : S \cdot s \cdot C \rangle :: D \Downarrow M$ . First consider the case when  $D$  is empty:

1. The base case of rule (Res) is obvious.
2. Consider the rule (AppRight). We have

$$\frac{\langle P : S \cdot s \cdot C_2 \rangle \Downarrow M_2 \quad C_1 \Downarrow^m M_1}{\langle P : S \cdot s \cdot C_2; C_1; \text{Apply} \rangle \Downarrow (M_1 \ M_2)} \text{(AppRight)}$$

By induction hypothesis, there exists  $M'_2$  such that  $\langle P' : S \cdot s \cdot C_2 \rangle \Downarrow M'_2$ . Therefore, we get:

$$\frac{\langle P' : S \cdot s \cdot C_2 \rangle \Downarrow M'_2 \quad C_1 \Downarrow^m M_1}{\langle P' : S \cdot s \cdot C_2; C_1; \text{Apply} \rangle \Downarrow (M_1 \ M'_2)} \text{(AppRight)}$$

3. In the case of the rule (AppLeft), we have

$$\frac{\langle P : S \cdot s \cdot C_1 \rangle \Downarrow M_1}{\langle P : S : M_2 \cdot s \cdot C_1; \text{Apply} \rangle \Downarrow (M_1 \ M_2)} \text{(AppLeft)}$$

Thus, by direct induction, there exists  $M'_1$  such that:

$$\frac{\langle P' : S \cdot s \cdot C_1 \rangle \Downarrow M'_1}{\langle P' : S : M_2 \cdot s \cdot C_1; \text{Apply} \rangle \Downarrow (M_1 \ M'_2)} \text{(AppLeft)}$$

Furthermore, observe that  $M_2$  cannot be the topmost element of the stack  $P : S : M_2$ . Thus,  $M_2$  is a C-value.

4. In the case of the rule (Fun( $n,i$ )), we have:

$$\frac{\langle P : S \cdot s \cdot C_i \rangle \Downarrow M_i \quad C_{i+1} \Downarrow^m M_{i+1} \quad \cdots \quad C_n \Downarrow^m M_n \quad C_0 \Downarrow^{n+1} M_0}{\langle P : S : M_{i-1} : \cdots : M_1 \cdot s \cdot C_i; \cdots; C_n; \text{Fun}(n, C_0) \rangle \Downarrow (\lambda M_0) [M_1 \cdots M_i \cdot ((M_{i+1} \cdots M_n \cdot \uparrow^m) \circ s)]}$$

By induction there exists  $M'_i$ , such that:

$$\frac{\langle P' : S \cdot s \cdot C_i \rangle \Downarrow M'_i \quad C_{i+1} \Downarrow^m M_{i+1} \quad \cdots \quad C_n \Downarrow^m M_n \quad C_0 \Downarrow^{n+1} M_0}{\langle P' : S : M_{i-1} : \cdots : M_1 \cdot s \cdot C_i; \cdots; C_n; \text{Fun}(n, C_0) \rangle \Downarrow (\lambda M_0) [M_1 \cdots M'_i \cdot ((M_{i+1} \cdots M_n \cdot \uparrow^m) \circ s)]}$$

Furthermore, the  $\lambda\sigma$ -terms  $M_1, \dots, M_{i-1}$  are C-values.

Finally, let us assume  $D = \langle S_n \cdot s_n \cdot C_n \rangle :: D_{n-1}$ . We have the following proof tree:

$$\frac{\langle P : S \cdot s \cdot C \rangle \Downarrow M_n \quad \langle M_n : S_n \cdot s_n \cdot C_n \rangle :: D_{n-1} \Downarrow M}{\langle P : S \cdot s \cdot C \rangle :: D \Downarrow M} \text{(State)}$$

By induction there exists  $M'_n$ , such that  $\langle P' : S \cdot s \cdot C \rangle \Downarrow M'_n$  holds, with  $M_n \sim M'_n$ . Therefore, by a second application of the induction hypothesis, there exists  $M'$ , with  $\langle M'_n : S_n \cdot s_n \cdot C_n \rangle :: D_{n-1} \Downarrow M'$  and  $M \sim M'$ . Hence, we get:

$$\frac{\langle P' : S \cdot s \cdot C \rangle \Downarrow M'_n \quad \langle M'_n : S_n \cdot s_n \cdot C_n \rangle :: D_{n-1} \Downarrow M'}{\langle P' : S \cdot s \cdot C \rangle :: D \Downarrow M'} \text{(State)}$$

□

Obviously, the C-strategy should be a deterministic subrelation of  $\sim$ .

Then, as we did for the SECD (cf. Lemma 8), our idea is to interpret the instruction to be executed next as a  $\lambda\sigma$ -term to be plugged in a context. However, we run across a first difficulty here: unlike SECD instructions, FAM instructions are not  $\lambda_{DB}$ -terms.

*Lemma 22*

Let  $D = \langle AS \cdot e \cdot I ; C \rangle :: D_0$  be a FAM state such that  $\bar{D}$  exists and the execution of  $I$  is enabled. Then,  $\Phi(D)$  can be written  $\langle S_I : S \cdot s \cdot I ; C \rangle :: \Phi(D_0)$  and there exists a  $\lambda\sigma$ -term  $M_I$  such that  $\langle S_I \cdot s \cdot I \rangle \Downarrow M_I$ .

*Proof*

More precisely, let us state  $D = \langle AS_n \cdot e_n \cdot I ; C_n \rangle :: \dots :: \langle AS_1 \cdot e_1 \cdot C_1 \rangle$ . By hypothesis,  $\Phi(D)$  exists and we have  $\Phi(D) = \langle S_n \cdot s_n \cdot I ; C_n \rangle :: \dots :: \langle S_1 \cdot s_1 \cdot C_1 \rangle$ . Then, consider, for instance, the case of the instruction  $\text{Fun}(n_0, C_0)$ . Since  $D$  can be decompiled, the proof tree  $T_n$  exists (i.e., the proof of  $\langle S_n \cdot s_n \cdot I ; C_n \rangle \Downarrow M_n$ , see figure 4). There is no other choice for  $T_n$  than to terminate by the following proof:

$$\frac{\langle \bar{f}_{n_0} \cdot s_n \cdot () \rangle \Downarrow \bar{f}_{n_0} \text{ (Res)} \quad C_0 \Downarrow^{n_0+1} M_0}{\langle \bar{f}_{n_0} : \dots : \bar{f}_1 \cdot s_n \cdot \text{Fun}(n_0, C_0) \rangle \Downarrow (\lambda M_0) [\bar{f}_1 \dots \bar{f}_n \cdot (\uparrow^m \circ s_n)]} \text{ (Fun}(n_0, n_0))$$

Let us state  $S_I = \bar{f}_{n_0} : \dots : \bar{f}_1$  and  $M_I = (\lambda M_0) [\bar{f}_1 \dots \bar{f}_n \cdot (\uparrow^m \circ s_n)]$ . By construction of proofs,  $S_I$  is a prefix of  $S_n$ . In other words, we get  $S_n = S_I \cdot S$ .

The remaining three instructions are treated similarly. Results are summarized hereafter (we state  $s = s_n$  and  $m = \text{length}(s)$ ):

$I$	$S_I$	$M_I$
Local	$()$	$1 [s]$
Global( $i$ )	$()$	$i+1 [s]$
Apply	$\bar{f}_1 : \bar{f}_2$	$(\bar{f}_1 \bar{f}_2)$
$\text{Fun}(n, C_0)$	$\bar{f}_{n_0} : \dots : \bar{f}_1$	$(\lambda M_0) [\bar{f}_1 \dots \bar{f}_n \cdot (\uparrow^m \circ s)]$

□

From the proof of the lemma above we easily infer the axioms of the C-strategy. To do so, we consider states  $D_I = \langle AS \cdot e \cdot I \rangle$ . The execution of  $I$  yields a new state  $D'_I$ , a new  $\lambda\sigma$ -term  $\bar{D}'_I$  and an axiom  $\bar{D}_I \xrightarrow{c} \bar{D}'_I$ .

$$\frac{(\lambda M_0) [s] \text{ is a C-value} \quad M \text{ is a C-value}}{((\lambda M_0) [s] M) \xrightarrow{c} M_0 [M \cdot s]} \quad n [M_1 \dots M_n \cdot s] \xrightarrow{c} M_n$$

$$\frac{M_1 \text{ is a C-value} \quad \dots \quad M_n \text{ is a C-value}}{(\lambda M_0) [M_1 \dots M_n \cdot (\uparrow^m \circ (P_1 \dots P_m \cdot \text{id}))] \xrightarrow{c} (\lambda M_0) [M_1 \dots M_n \cdot \text{id}]}$$

Notice that the transition **return** is the only silent transition.

Designing an equivalent to Lemma 8 for the FAM rises a second and more serious difficulty. The ‘plugging’ of a term  $X$  in a context  $\langle X : S \cdot s \cdot C \rangle :: D$  sometimes

$$\begin{array}{c}
 \frac{n [s] \triangleright\triangleright n [s]}{n [s] \triangleright\triangleright n [s]} \quad \frac{M_1 \text{ is a C-value} \quad M_2 \text{ is a C-value}}{(M_1 M_2) \triangleright\triangleright (M_1 M_2)} \\
 \\
 \frac{M_1 \text{ is a C-value} \quad \dots \quad M_n \text{ is a C-value}}{(\lambda M_0) [M_1 \cdots M_n \cdot (\uparrow^m \circ s)] \triangleright\triangleright (\lambda M_0) [M_1 \cdots M_n \cdot (\uparrow^m \circ s)]} \\
 \\
 \frac{M_2 [s] \triangleright\triangleright M'_2}{(M_1 M_2) [s] \triangleright\triangleright (M_1 [s] M'_2)} \quad \frac{M_2 \triangleright\triangleright M'_2}{(M_1 M_2) \triangleright\triangleright (M_1 M'_2)} \quad \frac{M_2 \text{ is a C-value} \quad M_1 \triangleright\triangleright M'_1}{(M_1 M_2) \triangleright\triangleright (M'_1 M_2)} \\
 \\
 \frac{M_1 [s] \triangleright\triangleright M'_1}{((\lambda M_0) [M_1 \cdot t]) [s] \triangleright\triangleright (\lambda M_0) [M'_1 \cdot (t \circ s)]} \\
 \\
 \frac{M_1 \text{ is a C-value} \quad \dots \quad M_{i-1} \text{ is a C-value} \quad M_i \triangleright\triangleright M'_i}{(\lambda M_0) [M_1 \cdots M_{i-1} \cdot M_i \cdot s] \triangleright\triangleright (\lambda M_0) [M_1 \cdots M_{i-1} \cdot M'_i \cdot s]} \\
 \\
 \frac{M_1 \text{ is a C-value} \quad \dots \quad M_i \text{ is a C-value} \quad M_{i+1} [s] \triangleright\triangleright M'_{i+1}}{(\lambda M_0) [M_1 \cdots M_i \cdot ((M_{i+1} \cdot t) \circ s)] \triangleright\triangleright (\lambda M_0) [M_1 \cdots M_i \cdot M'_{i+1} \cdot (t \circ s)]}
 \end{array}$$

 Fig. 6. Relation  $\triangleright\triangleright$ .

initiates a few substitution steps. We encode these steps using a new relation  $\triangleright\triangleright$ , which is a deterministic subrelation of  $\sim$ .

*Lemma 23*

Consider an instruction  $I$ . Further assume that the judgments  $\langle S_I \cdot s \cdot I \rangle \Downarrow M_I$  and  $\langle S_I : S \cdot s \cdot I ; C \rangle :: D \Downarrow M$  hold, where the  $\lambda\sigma$ -terms in  $S_I$  and  $S$  are C-values. Then, there exists a  $\lambda\sigma$ -term  $M'$ , such that  $\langle M_I : S \cdot s \cdot C \rangle :: D \Downarrow M'$  holds. Furthermore, we have  $M \triangleright\triangleright M'$ , where  $\triangleright\triangleright$  is a deterministic relation between  $\lambda\sigma$ -terms defined in figure 6.

*Proof*

We first consider the cases where  $D$  is empty.

1. If the code  $C$  is empty, then we must have  $S = ()$  and thus  $M = M_I$ . Hence, we get  $M' = M$  by the decompilation rule (Res).

$$\langle M_I \cdot s \cdot () \rangle \Downarrow M_I \text{ (Res)}$$

By the previous Lemma 22, we get the first three rules of figure 6.

2. If  $\langle S_I : S \cdot s \cdot I ; C \rangle \Downarrow M$  is proved using the rule (Code), then both stacks  $S_I$  and  $S$  are empty and we get:

$$\frac{I ; C \Downarrow^m P}{\langle () \cdot s \cdot I ; C \rangle \Downarrow P [s]} \text{ (Code)}$$

Then, there are subcases, depending upon the structure of  $C$ .

(a) If  $C = C_2; C_1; \text{Apply}$ , then we get:

$$\frac{\frac{I; C_2 \Downarrow^m M_2 \quad C_1 \Downarrow^m M_1}{I; C_2; C_1; \text{Apply} \Downarrow^m (M_1 M_2)} (\text{Apply})}{\langle () \cdot s \cdot I; C_2; C_1; \text{Apply} \rangle \Downarrow (M_1 M_2) [s]} (\text{Code})$$

Moreover, by the decompilation rule (Code), we have:

$$\frac{I; C_2 \Downarrow^m M_2}{\langle () \cdot s \cdot I; C_2 \rangle \Downarrow M_2 [s]} (\text{Code})$$

Thus, by induction hypothesis, there exists  $M'_2$  such that  $\langle M_I \cdot s \cdot C_2 \rangle \Downarrow M'_2$  holds. Furthermore, we have  $M_2 [s] \triangleright M'_2$ . Hence, we get:

$$\frac{\langle M_I \cdot s \cdot C_2 \rangle \Downarrow M'_2 \quad C_1 \Downarrow^m M_1}{\langle M_I \cdot s \cdot C_2; C_1; \text{Apply} \rangle \Downarrow (M_1 [s] M'_2)} (\text{AppRight})$$

With  $(M_1 M_2) [s] \triangleright (M_1 [s] M'_2)$ .

(b) If  $C = I; C_1; C_2; \dots; C_n; \text{Fun}(n, C_0)$ , then we get:

$$\frac{\frac{I; C_1 \Downarrow^m M_1 \quad C_2 \Downarrow^m M_2 \quad \dots \quad C_n \Downarrow^m M_n \quad C_0 \Downarrow^{n+1} M_0}{I; C_1; C_2; \dots; C_n; \text{Fun}(n, C_0) \Downarrow^m (\lambda M_0) [M_1 \cdot M_2 \cdot \dots \cdot M_n \cdot \uparrow^m]} (\text{Fun}(n))}{\langle () \cdot s \cdot I; C_1; C_2; \dots; C_n; \text{Fun}(n, C_0) \rangle \Downarrow ((\lambda M_0) [M_1 \cdot M_2 \cdot \dots \cdot M_n \cdot \uparrow^m]) [s]} (\text{Code})$$

Thus we have  $M = ((\lambda M_0) [M_1 \cdot M_2 \cdot \dots \cdot M_n \cdot \uparrow^m]) [s]$ . Then, by an argument similar to the one used above, there exists a  $\lambda\sigma$ -term  $M'$ , such that  $M \triangleright M'$  and  $\langle M_I \cdot s \cdot C_1; \dots; C_n; \text{Fun}(n, C_0) \rangle \Downarrow M'$  hold. More precisely, we have  $M' = (\lambda M_0) [M'_1 \cdot ((M_2 \cdot \dots \cdot M_n \cdot \uparrow^m) \circ s)]$ , with  $M_1 [s] \triangleright M'_1$ .

3. If  $\langle S_I : S \cdot s \cdot I; C \rangle \Downarrow M$  is proved using the rule (AppRight), then we have two subcases, depending on the position of  $I$  with respect to the premises of the rule (AppRight)

(a) If  $I$  is the first instruction of the left premise,

$$\frac{\langle S_I : S \cdot s \cdot I; C_2 \rangle \Downarrow M_2 \quad C_1 \Downarrow^m M_1}{\langle S_I : S \cdot s \cdot I; C_2; C_1; \text{Apply} \rangle \Downarrow (M_1 [s] M_2)} (\text{AppRight})$$

Then, by induction there exists a  $\lambda\sigma$ -term  $M'_2$ , such that  $M_2 \triangleright M'_2$  and  $\langle M_I : S \cdot s \cdot C_2 \rangle \Downarrow M'_2$ . Observing that the stack  $M_I : S$  cannot be empty, we get:

$$\frac{\langle M_I : S \cdot s \cdot C_2 \rangle \Downarrow M'_2 \quad C_1 \Downarrow^m M_1}{\langle M_I : S \cdot s \cdot C_2; C_1; \text{Apply} \rangle \Downarrow (M_1 [s] M'_2)} (\text{AppRight})$$

Hence the result.

(b) Otherwise,  $C_2$  is empty and  $I$  is the first instruction of the right premise. We have:

$$\frac{\langle M_2 \cdot s \cdot () \rangle \Downarrow M_2 (\text{Res}) \quad I; C_1 \Downarrow^m M_1}{\langle M_2 \cdot s \cdot I; C_1; \text{Apply} \rangle \Downarrow (M_1 [s] M_2)} (\text{AppRight})$$

Observe that, by hypothesis,  $M_2$  is a C-value, as we have  $S = M_2$  here.

Then, by the rule (Code), we have  $\langle () \cdot s \cdot I; C_1 \rangle \Downarrow M_1 [s]$ . Thus, by induction, there exists a  $\lambda\sigma$ -term  $M'_1$ , such that  $\langle M_I \cdot s \cdot C_1 \rangle \Downarrow M'_1$  and  $M_1 [s] \triangleright M'_1$ . Hence the result, since, by the decompilation rule (AppLeft), we get:

$$\frac{\langle M_I \cdot s \cdot C_1 \rangle \Downarrow M'_1}{\langle M_I : M_2 \cdot s \cdot C_1 \rangle \Downarrow (M'_1 M_2)} (\text{AppLeft})$$

4. If  $\langle S_I : S \cdot s \cdot I ; C \rangle \Downarrow M$  is proved using the rule (AppLeft), that is, if we have:

$$\frac{\langle S_I : S \cdot s \cdot I ; C_1 \rangle \Downarrow M_1}{\langle S_I : S : M_2 \cdot s \cdot I ; C_1 ; \text{Apply} \rangle \Downarrow (M_1 M_2)} \text{ (AppLeft)}$$

Then, by a straightforward application of the induction hypothesis, there exists  $M'_1$ , such that  $M_1 \triangleright\triangleright M'_1$  and

$$\frac{\langle M_I : S \cdot s \cdot C_1 \rangle \Downarrow M'_1}{\langle M_I : S : M_2 \cdot s \cdot C_1 ; \text{Apply} \rangle \Downarrow (M'_1 M_2)} \text{ (AppLeft)}$$

5. If  $\langle S_I : S \cdot s \cdot I ; C \rangle \Downarrow M$  is proved using the decompilation rule (Fun( $n,i$ )), then we have two subcases.

(a) If induction is straightforward, that is, if we have:

$$\frac{\langle S_I : S \cdot s \cdot I ; C_i \rangle \Downarrow M_i \quad C_{i+1} \Downarrow^m M_{i+1} \quad \cdots \quad C_n \Downarrow^m M_n \quad C_0 \Downarrow^{n+1} M_0}{\langle S_I : S : M_{i-1} : \cdots : M_1 \cdot s \cdot I ; C_i ; \cdots ; C_n ; \text{Fun}(n, C_0) \rangle \Downarrow (\lambda M_0) [M_1 \cdots M_i \cdot ((M_{i+1} \cdots M_n \cdot \uparrow^m) \circ s)]}$$

where, by hypothesis,  $S_I : S$  is non-empty and  $M_1, \dots, M_{i-1}$  are C-values.

Then, there exists  $M'_i$  such that  $M_i \triangleright\triangleright M'_i$  and

$$\frac{\langle M_I : S \cdot s \cdot C_i \rangle \Downarrow M'_i \quad C_{i+1} \Downarrow^m M_{i+1} \quad \cdots \quad C_n \Downarrow^m M_n \quad C_0 \Downarrow^{n+1} M_0}{\langle M_I : S : M_{i-1} : \cdots : M_1 \cdot s \cdot C_i ; C_{i+1} ; \cdots ; C_n ; \text{Fun}(n, C_0) \rangle \Downarrow (\lambda M_0) [M_1 \cdots M'_i \cdot ((M_{i+1} \cdots M_n \cdot \uparrow^m) \circ s)]}$$

(b) Otherwise,  $C_i$  is empty (or  $I$  is the first instruction of  $C_{i+1}$ ) and we have:

$$\frac{\langle M_i \cdot s \cdot () \rangle \Downarrow M_i \quad I ; C_{i+1} \Downarrow^m M_{i+1} \quad \cdots \quad C_n \Downarrow^m M_n \quad C_0 \Downarrow^{n+1} M_0}{\langle M_i : \cdots : M_1 \cdot s \cdot I ; C_{i+1} ; \cdots ; C_n ; \text{Fun}(n, C_0) \rangle \Downarrow (\lambda M_0) [M_1 \cdots M_{i-1} \cdot M_i \cdot ((M_{i+1} \cdots M_n \cdot \uparrow^m) \circ s)]}$$

(Observe that  $S_I$  and  $S$  must be empty here.)

By the decompilation rule (Code), we have  $\langle () \cdot s \cdot I ; C_{i+1} \rangle \Downarrow M_{i+1} [s]$ . Therefore, by induction, there exists  $M'_{i+1}$ , such that  $\langle M_i \cdot s \cdot C_{i+1} \rangle \Downarrow M'_{i+1}$  and  $M_{i+1} [s] \triangleright\triangleright M'_{i+1}$ . Therefore, by the decompilation rule (Fun( $i+1,n$ )), we get:

$$\frac{\langle M_i \cdot s \cdot C_{i+1} \rangle \Downarrow M'_{i+1} \quad C_{i+2} \Downarrow^m M_{i+2} \quad \cdots \quad C_n \Downarrow^m M_n \quad C_0 \Downarrow^{n+1} M_0}{\langle M_I : M_i : \cdots : M_1 \cdot s \cdot C_{i+1} ; \cdots ; C_n ; \text{Fun}(n, C_0) \rangle \Downarrow (\lambda M_0) [M_1 \cdots M_i \cdot M'_{i+1} \cdot ((M_{i+2} \cdots M_n \cdot \uparrow^m) \circ s)]}$$

Hence the result, since  $M_i, \dots, M_1$  are C-values by hypothesis.

Now, assume that  $D$  is not empty. That is, we have:

$$\frac{\langle S_I : S \cdot s \cdot I ; C \rangle \Downarrow P \quad \langle P : S' \cdot s' \cdot C' \rangle :: D' \Downarrow M}{\langle S_I : S \cdot s \cdot I ; C \rangle :: \langle S' \cdot s' \cdot C' \rangle :: D' \Downarrow M} \text{ (State)}$$

Then, by induction, there exists  $P'$ , with  $P \triangleright\triangleright P'$  and  $\langle M_I : S \cdot s \cdot C \rangle \Downarrow P'$ . Moreover, by our lemma 21, there exists  $M'$ , such that  $\langle P' : S' \cdot s' \cdot C' \rangle :: D' \Downarrow M'$ . Finally, we get:

$$\frac{\langle M_I : S \cdot s \cdot C \rangle \Downarrow P' \quad \langle P' : S' \cdot s' \cdot C' \rangle :: D' \Downarrow M'}{\langle M_I : S \cdot s \cdot C \rangle :: \langle S' \cdot s' \cdot C' \rangle :: D' \Downarrow M'} \text{ (State)}$$

Furthermore, we get  $M \triangleright\triangleright M'$ , since  $\triangleright\triangleright$  is included in  $\sim$ .  $\square$

The C-strategy is a combination of the three axioms of Lemma 22, of the relation  $\sim$  (cf. Lemma 21) and of the relation  $\triangleright\triangleright$  (cf. Lemma 23). The exact combination is given by the proof that the FAM implements a deterministic strategy.

More precisely, we first define a new relation  $\overset{c}{\sim}$  by considering the three reduction axioms and the inference rules of  $\sim$ . That is, we state:

$$\begin{array}{c}
\frac{(\lambda M_0) [s] \text{ and } M \text{ are C-values}}{((\lambda M_0) [s] M) \overset{c}{\sim} M_0 [M \cdot s]} \text{ (Beta)} \quad \mathfrak{n} [M_1 \cdots M_n \cdot s] \overset{c}{\sim} M_n \text{ (Var}^n) \\
\frac{M_1 \text{ is a C-value} \quad \cdots \quad M_n \text{ is a C-value}}{(\lambda M_0) [M_1 \cdots M_n \cdot (\uparrow^m \circ (P_1 \cdots P_m \cdot \text{id}))] \overset{c}{\sim} (\lambda M_0) [M_1 \cdots M_n \cdot \text{id}]} \text{ (Shift}^m) \\
\frac{M_2 \overset{c}{\sim} M'_2}{(M_1 M_2) \overset{c}{\sim} (M_1 M'_2)} \text{ (AppLeft)} \quad \frac{M_2 \text{ is a C-value} \quad M_1 \overset{c}{\sim} M'_1}{(M_1 M_2) \overset{c}{\sim} (M'_1 M_2)} \text{ (AppRight)} \\
\frac{M_1 \text{ is a C-value} \quad \cdots \quad M_{i-1} \text{ is a C-value} \quad M_i \overset{c}{\sim} M'_i}{(\lambda M_0) [M_1 \cdots M_{i-1} \cdot M_i \cdot s] \overset{c}{\sim} (\lambda M_0) [M_1 \cdots M_{i-1} \cdot M'_i \cdot s]} \text{ (Fun}(n,i))
\end{array}$$

Then, we define a step in the C-strategy as a  $\triangleright\triangleright$  step followed by a  $\overset{c}{\sim}$  step. Thus, for any two  $\lambda\sigma$ -terms  $M$  and  $M'$ , we have  $M \overset{c}{\rightarrow} M'$  if and only if there exists  $M''$  such that  $M \triangleright\triangleright M''$  and  $M'' \overset{c}{\sim} M'$ . Decomposing the rules on closures of  $\triangleright\triangleright$  and  $\overset{c}{\sim}$  into many smaller rules according to the structure of substitutions yields the definition of the C-strategy given in figure 7. The C-strategy is given in the same small step formalism we used for other strategies. Axioms and inference rules are classified per term construct in the algebra of  $\lambda\sigma$ -term.

#### Lemma 24

Let  $D$  be a FAM state such that  $\bar{D}$  exists. Let  $D'$  be a state such that  $D$  reduces to  $D'$  in one step. We have the following two cases:

1. If  $D \xrightarrow{\text{Return}} D'$  then  $\bar{D} = \bar{D}'$ .
2. Otherwise,  $D$  reduces to  $D'$  by the execution of one instruction  $I$  and we have  $\bar{D} \overset{c}{\rightarrow} \bar{D}'$ .

#### Proof

The first proposition is a direct corollary of definitions (cf. the decompilation rule (State)).

Let  $D$  be a state that evolves into  $D'$  by the execution of one instruction  $I$ . Let us state  $D = \langle AS \cdot e \cdot I; C \rangle :: D_0$ . By lemma 22,  $\Phi(D)$  can be written as  $\Phi(D) = \langle S_I : S \cdot s \cdot I; C \rangle :: \Phi(D_0)$ , where  $S_I$  is a stack such that  $\langle S_I \cdot s \cdot I \rangle \Downarrow M_I$ . Therefore, by Lemma 23, there exists  $M$  such that:

$$\langle M_I : S \cdot s \cdot C \rangle :: \Phi(D_0) \Downarrow M \quad \text{and} \quad \bar{D} \triangleright\triangleright M$$

Then, there are two cases, depending on whether a new frame is created or not:

1. First consider the case where  $I$  is Apply. Then, on the one hand, we have  $D = \langle (C_0/e_0) : f : AS_0 \cdot e \cdot \text{Apply}; C \rangle :: D_0$  and thus  $S_I = (\lambda M_0) [\bar{e}_0] : \bar{f}$ , with  $m_0 = \text{length}(e_0)$  and  $C_0 \Downarrow^{m_0+1} M_0$ . On the  $\lambda\sigma$ -term side, we get  $M_I = ((\lambda M_0) [\bar{e}_0] \bar{f})$ . On the other hand, we have  $D' = \langle f \cdot e_0 \cdot C_0 \rangle :: \langle AS_0 \cdot e \cdot C \rangle :: D_0$ . Thus, if  $\bar{D}'$  exists, we have:

$$\frac{\frac{C_0 \Downarrow^{m_0+1} M_0}{\langle () \cdot \bar{f} \cdot \bar{e}_0 \cdot C_0 \rangle \Downarrow M_0 [\bar{f} \cdot \bar{e}_0]} \text{(Code)} \quad \langle M_0 [\bar{f} \cdot \bar{e}_0] : S \cdot s \cdot C \rangle :: \Phi(D_0) \Downarrow \bar{D}'}{\Phi(D') \Downarrow \bar{D}'} \text{(State)}$$

Now, let us state  $M'_I = M_0 [\bar{f} \cdot \bar{e}_0]$ . Notice that we have  $M_I \overset{c}{\sim} M'_I$ , by the axiom (Beta). Therefore, by lemma 21, there exists a  $\lambda\sigma$ -term  $M'$  such that  $\langle M'_I : S \cdot s \cdot C \rangle :: \Phi(D_0) \Downarrow M'$ , with  $M \overset{c}{\sim} M'$ .

In other words,  $\bar{D}'$  exists and we have  $M \overset{c}{\sim} \bar{D}'$ .

2. Otherwise  $I$  is Global( $i$ ), Local or Fun( $n, C_0$ ). In these cases, there exists  $M'_I$  such that  $\Phi(D') = \langle M'_I : S \cdot s \cdot C \rangle :: \Phi(D_0)$ . The  $\lambda\sigma$ -term  $M'_I$  is  $\bar{g}$ , where  $g$  is either a newly created closure (when  $I$  is Fun( $n, C_0$ )) or a closure retrieved from the current environment or from the stack (when  $I$  is a variable access). Thus, by Lemma 21,  $\bar{D}'$  exists and we have  $\langle M'_I : S \cdot s \cdot C \rangle :: \Phi(D_0) \Downarrow \bar{D}'$ . Naturally, by our choice of axioms, we have  $M_I \overset{c}{\sim} M'_I$  and thus  $M \overset{c}{\sim} \bar{D}'$ .

Finally, since  $\bar{D} \triangleright M$  and  $M \overset{c}{\sim} \bar{D}'$ , we get  $\bar{D} \xrightarrow{c} \bar{D}'$ , by definition of  $\xrightarrow{c}$ .  $\square$

As illustrated by the rules (AppRight), (AppLeft) and (Beta), the FAM follows a right-to-left call-by-value strategy. With respect to the simpler strategies we already saw, the C-strategy presents two innovative features.

The first innovation lies in the propagation of substitutions inside terms (rules (MapApp)), the C-strategy combines several  $\sigma_w$ -reduction rules in one step. Finally, the rules (MapClos) and (MapEnv) ensures a similar propagation mechanism inside the environment component of closures and inside environments themselves.

Second, reduction is now possible inside the environment part  $s$  of a closure  $(\lambda M_0) [s]$  (rules (ClosRight)). This reduction operates from the left to the right (rules (ConsLeft) and (ConsRight)). It is important to notice that our simple compilation scheme does not fully exploit the reduction capabilities of the FAM. Here, the  $\mathcal{C}$  compilation scheme abstracts only the free variables out of function bodies. As a consequence, when the FAM executes code produced by this simple scheme, it performs only variable fetching while reducing substitutions. That is, the term  $M$  in the rule (MapEnv) is always a variable and the rule (ConsLeft) will never be used. The C-strategy can cope with alternative and more sophisticated compilation schemes. In such schemes, complete sub-expressions would be abstracted out of function bodies and the premise of rule (ConsLeft) could be any C-reduction.

$$\begin{array}{c}
\frac{(\lambda M_0) [s] \text{ and } M \text{ are C-values}}{((\lambda M_0) [s] M) \xrightarrow{c} M_0 [M \cdot s]} \text{ (Beta)} \\
n [M_1 \cdots M_n \cdot s] \xrightarrow{c} M_n (\text{Var}^n) \qquad \uparrow^n \circ (M_1 \cdots M_n \cdot \text{id}) \xrightarrow{c} \text{id} \text{ (Shift}^n\text{)} \\
\frac{M_2 [s] \xrightarrow{c} M'_2}{(M_1 M_2) [s] \xrightarrow{c} (M_1 [s] M'_2)} \text{ (MapApp)} \\
\frac{M_2 \xrightarrow{c} M'_2}{(M_1 M_2) \xrightarrow{c} (M_1 M'_2)} \text{ (AppRight)} \qquad \frac{M_1 \xrightarrow{c} M'_1 \quad M_2 \text{ is a C-value}}{(M_1 M_2) \xrightarrow{c} (M'_1 M_2)} \text{ (AppLeft)} \\
\frac{s \circ t \xrightarrow{c} s'}{((\lambda M) [s]) [t] \xrightarrow{c} (\lambda M) [s']} \text{ (MapClos)} \qquad \frac{s \xrightarrow{c} s'}{(\lambda M_0) [s] \xrightarrow{c} (\lambda M_0) [s']} \text{ (ClosRight)} \\
\frac{M [s] \xrightarrow{c} M'}{(M \cdot t) \circ s \xrightarrow{c} M' \cdot (t \circ s)} \text{ (MapEnv)} \qquad \frac{M \xrightarrow{c} M'}{M \cdot s \xrightarrow{c} M' \cdot s} \text{ (ConsLeft)} \\
\frac{M \text{ is a C-value} \quad s \xrightarrow{c} s'}{M \cdot s \xrightarrow{c} M \cdot s'} \text{ (ConsRight)}
\end{array}$$

Fig. 7. The C-strategy.

Remember that the starting terms of the C-strategy are particular: they are terms  $M [\text{id}]$  such that the predicate  $\mathcal{P}_0(M)$  holds. Thus, the terms produced by the C-strategy are also particular. They satisfy a predicate  $\mathcal{Q}_k$ :

$$\begin{aligned}
\mathcal{Q}_k(\mathbf{n}) &= (n \leq k) \\
\mathcal{Q}_k(M_1 M_2) &= \mathcal{Q}_k(M_1) \wedge \mathcal{Q}_k(M_2) \\
\mathcal{Q}_k(\lambda M) &= \mathcal{Q}_{k+1}(M) \\
\mathcal{Q}_k(M [s]) &= \mathcal{Q}_{l_s}(M) \wedge \mathcal{Q}_k(s), \text{ where } l_s = \text{length}(s) \\
\mathcal{Q}_k(M \cdot s) &= \mathcal{Q}_k(M) \wedge \mathcal{Q}_k(s) \\
\mathcal{Q}_k(s \circ t) &= \mathcal{Q}_{l_t}(s) \wedge \mathcal{Q}_k(t), \text{ where } l_t = \text{length}(t) \\
\mathcal{Q}_k(s) &= \text{false} \text{ otherwise} \\
\text{length}(\uparrow^n) &= 0 \\
\text{length}(M \cdot s) &= 1 + \text{length}(s) \\
\text{length}(s \circ t) &= \text{length}(s), \text{ when } \mathcal{Q}_{l_t}(s) \text{ where } l_t = \text{length}(t) \\
\text{length}(s) &\text{ is undefined otherwise}
\end{aligned}$$

One easily checks that the new predicate  $\mathcal{Q}_k$  generalizes  $\mathcal{P}_k$ . That is, the implication  $\mathcal{P}_k(M) \Rightarrow \mathcal{Q}_k(M)$  holds. Thus, given any closed  $\lambda_{\text{DB}}$ -term  $N$ , let  $M$  be  $\mathcal{C}(\emptyset, N)$ . Then, the predicate  $\mathcal{Q}_0(M [\text{id}])$  holds, since we have  $\mathcal{P}_0(M)$ .

Intuitively, given a term  $M$ , the predicate  $\mathcal{Q}_k(M)$  holds when  $M$  is being evaluated in an environment of size  $k$ . As expected, this condition is preserved by the C-strategy:

*Lemma 25*

Let  $M$  and  $M'$  be two  $\lambda\sigma$ -terms, such that  $M \xrightarrow{c} M'$  and  $\mathcal{Q}_k(M)$ . Then, we have  $\mathcal{Q}_k(M')$ .

*Proof*

Tedious. A key point is that, given two substitutions  $s$  and  $s'$  such that  $s \xrightarrow{c} s'$ , we have  $\text{length}(s) = \text{length}(s')$ .  $\square$

*Lemma 26 (Final state condition)*

Let  $N$  be a closed  $\lambda_{\text{DB}}$ -term and let  $M$  be  $\mathcal{C}(\emptyset, N)$ . Let  $D$  be a terminal state computed by the FAM from  $\mathcal{L}(M)$ . Then,  $\bar{D}$  is a C-normal form.

*Proof*

First observe that, by Lemma 24,  $\bar{D}$  exists and that we have  $M[\text{id}] \xrightarrow{c}^* \bar{D}$ . Let us state  $D = \langle AS_n \cdot e_n \cdot C_n \rangle :: \dots :: \langle AS_1 \cdot e_1 \cdot C_1 \rangle$ . There are three kinds of states from which no transition is enabled. The first two cases are the same as for the SECD (cf. Lemma 11), as concerns both statement and proof:

1.  $n = 1$  and  $C_1 = ()$ . then, the stack  $AS_1$  holds exactly one closure  $f$  (otherwise it cannot be decompiled) and we get  $\bar{D} = \overline{\langle f \cdot () \cdot () \rangle} = \bar{f}$ , which is a C-value and a C-normal form.
2. If the head instruction of  $C_n$  is  $\text{Fun}(n_0, C_0)$  or  $\text{Apply}$  and if there are not enough arguments on the state  $AS_n$  for it to execute, then it can be shown that  $D$  cannot be decompiled either. Therefore, this case cannot occur.
3. If an environment access fails, that is, if  $D = \langle AS \cdot () \cdot \text{Local}; C \rangle$ ,  $D = \langle () \cdot e \cdot \text{Local}; C \rangle :: D_0$  or  $D = \langle AS \cdot \overrightarrow{f_{1,m}} \cdot \text{Global}(k); C \rangle :: D'$  with  $k > n$ . Then,  $\bar{D}$  is a C-failure term  $W$ , defined as follows:

$$\begin{array}{ll}
 W ::= & m [\overline{f_1} \cdots \overline{f_k} \cdot \text{id}] & \text{with } k < m \\
 & | \quad M W \\
 & | \quad W M & \text{where } M \text{ is a C-value} \\
 & | \quad (\lambda M_0) [M_1 \cdots M_{i-1} \cdot W \cdot s] & \text{where } M_1, \dots, M_{i-1} \text{ are C-values}
 \end{array}$$

A C-failure term is a C-normal form, since a C-normal form which is not a value stands in C-redex position.

In fact, such a case cannot occur. Any C-failure term  $W$  has a subterm  $W' = n [\overline{f_1} \cdots \overline{f_k} \cdot \text{id}]$  with  $k < n$ , such that  $\mathcal{Q}_0(W')$  does not hold. As a result,  $\mathcal{Q}_0(W)$  cannot hold, which contradicts the initial condition  $\mathcal{Q}_0(M[\text{id}])$ , by Lemmas 24 and 25.  $\square$

The transition **return** is the only silent transition of the FAM. Obviously, it cannot be performed infinitely many times in a row. We conclude:

*Theorem 3*

The FAM implements the C-strategy.

## 6 The Categorical Abstract Machine

In this section, we prove that the categorical abstract machine (CAM) (Cousineau *et al.*, 1985) implements a strategy in  $\lambda\sigma_w$ . For brevity, some proofs, which are very similar to previous ones, are only sketched.

### 6.1 Basics

The CAM has seven instructions.

INSTRUCTION ::= Fst | Snd | < |  $\mathfrak{S}$  | > | App |  $\Lambda$ (CODE)

CAM environments belong to two different sorts: they are either closures (written  $f$ ) or trees (written  $e$ ).

ENVIRONMENT ::= () | CLOSURE | (ENVIRONMENT, ENVIRONMENT)

The states of the CAM (written  $D = \langle S \cdot C \rangle$ ) are just pairs of a stack with a code.

STACK ::= () | ENVIRONMENT : STACK

FRAME ::=  $\langle$  STACK  $\cdot$  CODE  $\rangle$

STATE ::= FRAME

The transitions of the CAM are as follows:

$$\begin{aligned} \langle (e, f) : S \cdot \text{Fst}; C \rangle &\xrightarrow{\text{car}} \langle e : S \cdot C \rangle \\ \langle (e, f) : S \cdot \text{Snd}; C \rangle &\xrightarrow{\text{cdr}} \langle f : S \cdot C \rangle \\ \langle e : S \cdot \Lambda(C); C' \rangle &\xrightarrow{\text{cur}} \langle (C/e) : S \cdot C' \rangle \\ \langle e : S \cdot <; C \rangle &\xrightarrow{\text{push}} \langle e : e : S \cdot C \rangle \\ \langle f : e : S \cdot \mathfrak{S}; C \rangle &\xrightarrow{\text{swap}} \langle e : f : S \cdot C \rangle \\ \langle f : g : S \cdot >; C \rangle &\xrightarrow{\text{cons}} \langle (g, f) : S \cdot C \rangle \\ \langle (C/e), f : S \cdot \text{App}; C' \rangle &\xrightarrow{\text{app}} \langle (e, f) : S \cdot C; C' \rangle \end{aligned}$$

We have adopted a slightly unusual presentation of CAM transitions: for an instruction to execute, not only must the proper number of arguments stand on top of the stack, but these arguments must also be of the proper sort, either closure or tree node. Consider for instance the instruction  $\mathfrak{S}$ , i.e. the transition **swap**. This instruction swaps the two topmost elements of the stack, provided the topmost one is a closure and the other one is a tree node. Doing so, we make explicit the sort discipline that is usually left implicit in standard categorical code.

Compilation of  $\lambda_{\text{DB}}$ -terms in CAM code follows the usual translation from  $\lambda$ -terms to terms of Categorical Cartesian Logic (CCL).

$$\begin{aligned} \llbracket 1 \rrbracket &= \text{Snd} \\ \llbracket n + 1 \rrbracket &= \text{Fst} \llbracket n \rrbracket \\ \llbracket (N_1 N_2) \rrbracket &= \langle \llbracket N_1 \rrbracket \mathfrak{S} \llbracket N_2 \rrbracket \rangle \text{App} \\ \llbracket \lambda N \rrbracket &= \Lambda(\llbracket N \rrbracket) \end{aligned}$$

In the rules above we omit the separator ‘;’ in code segments.

Finally, a code  $C = \llbracket N \rrbracket$  is loaded into the CAM as  $\mathcal{L}(N) = \langle () \cdot C \rangle$ .

### 6.2 Decompilation

First, we inverse the compilation procedure  $\llbracket \cdot \rrbracket$ . We do so by proving judgments  $C \Downarrow N$ , which read “the code segment  $C$  stands for the  $\lambda_{\text{DB}}$ -term  $N$ ”.

$$\text{Fst}''; \text{Snd} \Downarrow n+1 \quad \frac{C_1 \Downarrow N_1 \quad C_2 \Downarrow N_2}{\langle C_1 \succ C_2 \rangle \text{App} \Downarrow (N_1 N_2)} \quad \frac{C \Downarrow N}{\Lambda(C) \Downarrow \lambda N}$$

For any  $\lambda_{\text{DB}}$ -term  $N$  and categorical code  $C$ , the equivalence  $\llbracket N \rrbracket = C \Leftrightarrow C \Downarrow N$  is easily shown.

The decompilation procedure extends naturally to environments and closures,

$$\begin{aligned} \overline{(C/e)} &= (\lambda N) [\bar{e}], \text{ where } C \Downarrow N \\ \overline{(e, f)} &= \bar{f} \cdot \bar{e} \\ \overline{()} &= \text{id} \end{aligned}$$

A X-value is the decompilation of a closure. Thus, X-values are written  $\bar{f}$ .

State decompilation is performed by proving a judgment  $D \Downarrow \bar{D}$ , using the following axioms and inference rules:

$$\begin{aligned} \langle f \cdot () \rangle \Downarrow \bar{f} \text{ (Res)} & \quad \frac{C \Downarrow N}{\langle e \cdot C \rangle \Downarrow N [\bar{e}]} \text{ (Code)} \\ \frac{\langle S \cdot C \rangle \Downarrow M \quad C' \Downarrow N}{\langle S : e \cdot C \succ C' \rangle \text{App} \Downarrow (M N [\bar{e}])} \text{ (AppLeft)} & \quad \frac{\langle S \cdot C \rangle \Downarrow M}{\langle S : f \cdot C \rangle \text{App} \Downarrow (\bar{f} M)} \text{ (AppRight)} \\ \langle (f_1, f_2) \cdot \text{App} \rangle \Downarrow (\bar{f}_1 \bar{f}_2) \text{ (AppCons)} & \end{aligned}$$

Decompilation rules follow a sort discipline, just as transition rules do. For instance, the rule (Res) applies only when the stack  $S$  holds a single closure. Besides, decompilation rules are analog to other machines rules and bear the same names. The only slight novelty is the rule (AppCons), which derives from the transition **cons**, a transition that could easily be merged with the transition **app**.

The non-ambiguity of state decompilation follows quite easily from the rich structure of categorical code. Basically, proof trees are unique because the ‘<’ and ‘>’ instructions act as well-balanced parenthesis in code segments.

### 6.3 Strategy

As we did for the FAM, we introduce gradually the strategy implemented by the CAM. We call this strategy the X-strategy (written  $\xrightarrow{X}$ ).

In the absence of multi-frame CAM states, the proof that the CAM implements the X-strategy differs slightly from the corresponding proofs for the SECD and the

FAM. Specifically, induction is now performed differently, by directly considering ‘sub-states’, instead of plugging a  $\lambda\sigma$ -term on top of the current active frame as we did for the previous two machines.

More specifically the two inductive decompilation rules (AppLeft) and (AppRight) both extract a sub-state  $\langle S' \cdot C' \rangle$  from a state  $\langle S \cdot C \rangle$  by removing some elements from the bottom of the stack  $S$  and some instructions from the end of code  $C$ . In other words,  $\langle S \cdot C \rangle$  is decomposed as  $\langle S' : S'' \cdot C' ; C'' \rangle$ , where  $\langle S' \cdot C' \rangle$  is a valid CAM state. The following lemma exposes such a state decomposition, designed for identifying strategy axioms.

*Lemma 27*

Let  $D$  be a CAM state such that the execution of any instruction is enabled, yielding a new state  $D'$ . If  $D$  exists, then  $D$  can be written  $\langle S_I : S \cdot C_I ; C \rangle$  and there exists a  $\lambda\sigma$ -term  $M_I$  such that  $\langle S_I \cdot C_I \rangle \Downarrow M_I$ . Additionally,  $D'$  can be written  $\langle S'_I : S \cdot C'_I ; C \rangle$  and there exists a  $\lambda\sigma$ -term  $M'_I$  such that  $\langle S'_I \cdot C'_I \rangle \Downarrow M'_I$ .

*Proof*

A very constructive one:

$I$	Snd	Fst	<	App
$C_I$	Snd	Fst <sup>n</sup> ; Snd	< $\llbracket N_1 \rrbracket \mathbin{\text{\textcircled{>}}} \llbracket N_2 \rrbracket$ > App	App
$S_I$	$(e, f)$	$(e, f)$	$e$	$((\llbracket N_1 \rrbracket / e_1), f_2)$
$M_I$	$1 \bar{f} \cdot \bar{e}$	$n+1 \bar{f} \cdot \bar{e}$	$(N_1 N_2) [\bar{e}]$	$((\lambda N_1) [\bar{e}_1] \bar{f}_2)$
$C'_I$	()	Fst <sup>n-1</sup> ; Snd	$\llbracket N_1 \rrbracket \mathbin{\text{\textcircled{>}}} \llbracket N_2 \rrbracket$ > App	$\llbracket N_1 \rrbracket$
$S'_I$	$f$	$e$	$e : e$	$(e_1, f_2)$
$M'_I$	$\bar{f}$	$n [\bar{e}]$	$(N_1 [\bar{e}] N_2 [\bar{e}])$	$N_1 [\bar{f}_2 \cdot \bar{e}_1]$

$I$	$\Lambda(\llbracket N_0 \rrbracket)$	,	>
$C_I$	$\Lambda(\llbracket N_0 \rrbracket)$	$\mathbin{\text{\textcircled{>}}} \llbracket N_2 \rrbracket$ > App	> App
$S_I$	$e$	$f : e$	$f_2 : f_1$
$M_I$	$(\lambda N_0) [\bar{e}]$	$(\bar{f} N_2 [\bar{e}])$	$(\bar{f}_1 \bar{f}_2)$
$C'_I$	()	$\llbracket N_2 \rrbracket$ > App	App
$S'_I$	$(\llbracket N_0 \rrbracket / e)$	$e : f$	$(f_1, f_2)$
$M'_I$	$(\lambda N_0) [\bar{e}]$	$(\bar{f} N_2 [\bar{e}])$	$(\bar{f}_1 \bar{f}_2)$

□

The axioms of the X-strategy derive from the lemma above. These axioms are the four rules  $M_I \xrightarrow{X} M'_I$ , where  $I$  is any of the instructions **Fst**, **Snd**, **<** or **App**.

In the case of the transitions **cur**, **swap** and **cons**, we have  $M_I = M'_I$ . Thus, these transitions are good candidates for being silent.

Now, thanks to our induction technique on sub-states, we make explicit the inductive rules of the X-strategy:

*Lemma 28*

Let  $D = \langle S_I : S \cdot C_I ; C \rangle$  be a CAM state such that both judgments  $D \Downarrow M$  and  $\langle S_I \cdot C_I \rangle \Downarrow M_I$  hold. Let  $\langle S'_I \cdot C'_I \rangle$  be any CAM state such that the judgment  $\langle S'_I \cdot C'_I \rangle \Downarrow M'_I$  holds. Let  $D'$  be the state  $\langle S'_I : S \cdot C'_I ; C \rangle$ . Then, there exists a  $\lambda\sigma$ -term  $M'$  such that  $D' \Downarrow M'$ . Furthermore, given any relation  $\sim$ , such that  $M_I \sim M'_I$ , we have  $M \sim M'$ , provided  $\sim$  obeys the following structural rules:

$$\frac{M_1 \sim M'_1}{(M_1 M_2) \sim (M'_1 M_2)} \qquad \frac{M_1 \text{ is a X-value} \quad M_2 \sim M'_2}{(M_1 M_2) \sim (M_1 M'_2)}$$

*Proof*

By induction on the length of  $C$ . Let us consider, for instance, the base case and assume that  $C$  is empty. Then, one shows by induction on the length of  $C_I$  that  $S$  must be empty and thus  $M = M_I$ .  $\square$

First, notice that, when the instruction  $I$  is  $\Lambda$ ,  $\mathfrak{S}$  or  $>$  (and thus when  $M_I = M'_I$ ), we just proved that the corresponding transitions **cur**, **swap** and **cons** are silent.

Then, we get the X-strategy, by combining the four axioms  $M_I \xrightarrow{X} M'_I$  (where  $I$  is **Fst**, **Snd**, **<** or **App**) with the inductive rules of lemma 28:

$$\begin{array}{c} 1 [M \cdot s] \xrightarrow{X} M \text{ (FVar)} \qquad \mathfrak{n}+1 [M \cdot s] \xrightarrow{X} \mathfrak{n} [s] \text{ (RVar)} \\ (N_1 N_2) [s] \xrightarrow{X} (N_1 [s] N_2 [s]) \text{ (App)} \qquad \frac{(\lambda N) [s] \text{ is a X-value} \quad M \text{ is a X-value}}{((\lambda N) [s] M) \xrightarrow{X} N [M \cdot s]} \text{ (Beta)} \\ \frac{M_1 \xrightarrow{X} M'_1}{(M_1 M_2) \xrightarrow{X} (M'_1 M_2)} \text{ (AppLeft)} \qquad \frac{M_1 \text{ is a X-value} \quad M_2 \xrightarrow{X} M'_2}{(M_1 M_2) \xrightarrow{X} (M_1 M'_2)} \text{ (AppRight)} \end{array}$$

Finally, the X-strategy is simple left-to-right call-by-value.

*Lemma 29 (Final state condition)*

Let  $N$  be a closed  $\lambda_{DB}$ -term and let  $D$  be a terminal state computed by the CAM starting from  $\mathcal{L}(N)$ . Then,  $\bar{D}$  is a X-normal form.

*Proof*

First observe that  $\bar{D}$  exists and is computed by iterating the X-strategy starting from  $N [id]$ . Let us then state  $D = \langle S \cdot C \rangle$ . The CAM may stop for many reasons, which fall into three classes:

1. The code  $C$  is empty. Then,  $\bar{D}$  must be computed by the decompilation rule (Res). Thus,  $S$  holds a single element which *must* be a closure  $f$ . Hence  $\bar{D}$  is the X-value  $\bar{f}$ . This is the normal case.
2. A variable access fails. That is,  $D = \langle () \cdot \text{Fst}; C_0 \rangle$  or  $D = \langle () \cdot \text{Snd}; C_0 \rangle$ . Then,  $\bar{D}$  is a X-failure term  $W$ :

$$\begin{array}{l}
 W ::= n [\text{id}] \quad \text{with } n \geq 1 \\
 \quad | \quad W M \\
 \quad | \quad M W \quad \text{where } M \text{ is a X-value}
 \end{array}$$

One easily sees that X-failure terms are X-normal forms. Furthermore, they are not closed  $\lambda\sigma$ -terms. Hence, by Lemma 1 and since  $N [\text{id}]$  is closed, this case cannot occur.

3. The code  $C$  is not empty (i.e.  $C = I; C_0$ ), but the execution of the instruction  $I$  is not enabled, because  $S$  holds too few arguments or because the sort discipline on transitions is violated. In fact, such cases cannot occur here, precisely because  $\bar{D}$  exists. The proof tree of  $\langle S \cdot C \rangle \Downarrow M$  must include the proof of a judgment  $\langle S_I \cdot C_I \rangle \Downarrow M_I$  that ‘consumes’ the instruction  $I$ , i.e. a proof whose premises do not include  $I$  anymore. Moreover, by the inductive structure of proofs, we have  $S = S_I : S'$ . When, for instance,  $I$  is ‘ $\mathfrak{g}$ ’, we have  $C_I = \mathfrak{g} C'_I > \text{App}$  and

$$\frac{\langle f \cdot () \rangle \Downarrow \bar{f} \text{ (Res)} \quad C'_I \Downarrow N'_I}{\langle f : e \cdot \mathfrak{g} C'_I > \text{App} \rangle \Downarrow (\bar{f} N'_I [\bar{e}])} \text{ (AppLeft)}$$

Thus, we get  $S_I = f : e$ . Hence, since  $S = S_I : S'$ , the transition **swapp** is enabled.

□

Finally, all CAM transitions but the transition **app** consume one instruction. Therefore, any computation of the CAM that does not include the transition **app** is finite. Thus, since the transition **app** is not silent, there cannot be infinitely many silent CAM transitions successively. We conclude:

*Theorem 4*

The CAM implements the X-strategy.

## 7 Other execution models

In the previous sections, we have exhibited the  $\lambda\sigma_w$ -calculus strategies hidden inside four machines. By a simple improvement on (Leroy, 1990), we also made explicit the strategy of the ZAM. Briefly, the ZAM implements the L-strategy, i.e. right-to-left call-by-value.

The G-machine and the TIM (Fairbairn and Wray, 1987) can also be understood in the  $\lambda\sigma$ -calculus, although these machines look quite different from environment machines. In order to simplify the management of variables at run-time, compilers for these machines translate input  $\lambda$ -terms into supercombinators, by the so-called  $\lambda$ -lifting operation (Peyton-Jones, 1987). Supercombinators are n-ary functions without free variables, that is, in terms of  $\lambda\sigma_w$ , closures  $(\lambda\lambda \dots \lambda M) [\text{id}]$ , where  $M$  is a  $\lambda\sigma$ -term whose variables are all  $\lambda$ -bound. We call such closures  $\lambda\sigma$ -supercombinators. In the

$\lambda\sigma$ -framework,  $\lambda$ -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  $\lambda$ -lifting will produce the partial application of a  $\lambda\sigma$ -supercombinator. For instance, the abstraction,  $\lambda(1 (5 7))$  is translated by the scheme  $\mathcal{C}$  into the  $\lambda\sigma$ -closure  $(\lambda(1 (2 3))) [4 \cdot 6 \cdot \text{id}]$ , whereas it lambda-lifts to the  $\lambda\sigma$ -term  $((\lambda\lambda\lambda(1 (2 3))) [\text{id}]) 6 4$ .

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:

$$(\text{Beta}_n) \quad ((\lambda \dots \lambda M) [\text{id}]) N_1 \dots N_n \rightarrow M [N_n \dots N_1 \cdot \text{id}]$$

The G-machine and the TIM implement a very similar strategy that basically amounts to contracting the leftmost-outermost  $(\text{Beta}_n)$  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 (Appel, 1992). Roughly speaking, a schematic SML/NJ compiler would first translate a source  $\lambda$ -term into a  $\lambda$ -term in continuation-passing style (CPS). Then, this  $\lambda$ -term in CPS would be further transformed by the so-called closure conversion. The original SML/NJ closure conversion transforms functions into record data structures, which encode closures. In our our closure calculus, such records are of course not needed and our closure conversion would output  $\lambda\sigma$ -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 Appel (1992) does, using enriched  $\lambda$ -calculus (in CPS) as a formal language.

## 8 Related work and conclusion

The main contribution of this paper resides in the introduction the  $\lambda\sigma_w$ -calculus as ‘the’ weak  $\lambda$ -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.

Rather than providing an ‘automatic’ procedure for proving abstract machines, we introduced a method to do such proofs. This method consists in extracting strategy axioms from machine transitions and strategy structural rules from the machine structure. Doing so, we abstract on implementation issues, such as stack management or closure format, focusing on semantics.

Our work is to be compared first with similar attempts to prove, formalize or derive several functional back-ends in an unified formalism. In Curien (1991), 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 Maranget (1991)), and can be seen as a predecessor of our standard term rewriting system  $\lambda\sigma_w$ . A recent

publication (Douence and Fradet, 1995) resembles our work, since it models many compilers and abstract machines, using the  $\lambda$ -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 systems have in common, to the point of proposing a definition for compiled functionality in a relatively well established formalism, whereas (Douence and Fradet, 1995) focuses more on modelling the exact structure of abstract machines, to establish their ‘taxonomy’ of functional languages implementation.

Then, our work is to be compared also with others 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  $\lambda$ -calculus. As a consequence, our correctness proofs appear to be quite simple. By contrast, the correctness proofs of the CAM in Asperti (1992) and of the SECD machine in Rittri (1988) are complicated. This is no surprise from the  $\lambda\sigma$  point of view, since these proofs basically include a correctness proof of  $\lambda\sigma_w$  with respect to the  $\lambda$ -calculus. Our bisimulation technique is a simplification of Rittri’s work. Our main simplification lies in adopting a term language for closures, where Rittri used ordinary  $\lambda$ -calculus. In the end, this yields simpler proofs, as well as more numerous and detailed machine descriptions. 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 Hannan and Miller (1992), which relied upon natural or ‘big-step’ semantics and from Crégut (1990) and Leroy (1990), 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  $\lambda$ -calculus (Maranget, 1991). As a second example, given the same  $\lambda$ -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 modelled in a simple extension to term-graphs of the weak  $\lambda$ -calculus with explicit substitutions. To render sharing while preserving the desirable simplicity of terms, several techniques already exist, such as subterms labelling (Maranget, 1991), explicit recursive equations (Ariola *et al.*, 1995), or specialized bindings (Lauchbury, 1993).

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. Moreover, our approach should be relevant and easy to consider when compilation is modelled by type-preserving transformations of a typed language, since there exist typed  $\lambda$ -calculi with explicit substitutions (Abadi *et al.*, 1996). For instance, in Abadi *et al.* (1996), closures are typed using existential polymorphism, and type-safe closure conversions are presented. However, closures are naturally typed in any typed  $\lambda\sigma$ -calculus, using standard type systems. It would be interesting to compare the two approaches.

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, etc. A first step we are taking is to extend  $\lambda\sigma$  with arbitrary term rewriting systems.

### References

- Abadi, M., Cardelli, L., Curien, P.-L. and Lévy, J.-J. (1996) Explicit substitutions. *J. Functional Program.* **6**(2), 299–327.
- Appel, A. W. (1992) *Compiling with Continuations*. Cambridge University Press.
- Ariola, Z., Felleisen, M., Maraist, J., Odersky, M. and Wadler, P. (1995) The call-by-need lambda-calculus. *22nd Symposium on Principles of Programming Languages*.
- Asperti, A. (1992) A categorical understanding of environment machines. *J. Functional Program.* **2**(1), 23–59.
- Augustsson, L. (1984) A compiler for Lazy ML. *1984 Symposium on Lisp and Functional Programming*.
- Cardelli, L. (1984) Compiling a functional language. *1984 Symposium on Lisp and Functional Programming*.
- Cousineau, G., Curien, P.-L. and Mauny, M. (1985) The Categorical Abstract Machine. *Functional Programming and Computer Architecture*.
- Crégut, P. (1990) An abstract machine for the normalization of lambda-terms. *1990 Symposium on Lisp and Functional Programming*.
- Curien, P.-L. (1991) An abstract framework for environment machines. *Theor. Comput. Sci.* **82**(2), 389–402.
- Curien, P.-L., Hardin, T. and Lévy, J.-J. (1996) Confluence properties of weak and strong calculi of explicit substitutions. *J. ACM*, **43**(2), 362–397.
- Douence, R. and Fradet, P. (1995) Towards a taxonomy of functional language implementations. *Symposium on Programming Languages Implementations and Logic Programming*.
- Dowek, G., Hardin, T. and Kirchner, C. (1995) Higher-order unification via explicit substitutions. *IEEE Symposium on Logics in Computer Science*. IEEE Press.
- Fairbairn, J. and Wray, S. (1987) Tim: A simple, lazy abstract machine to execute supercombinators. *Functional Programming and Computer Architecture: Lecture Notes in Computer Science 274*. Springer-Verlag.
- Hannan, J. and Miller, D. (1992) From operational semantics to abstract machines. *J. Mathematical Structures in Comput. Sci.*, **2**(4), 415–459.
- Herbelin, H. (1994) A lambda-calculus structure isomorphic to sequent calculus structure. *Computer Science and Logics: Lecture Notes in Computer Science 933*. Springer-Verlag.
- Landin, P. J. (1964) The mechanical evaluation of expressions. *Computer J.*, **6**(4), 308–320.
- Launchbury, J. (1993) A natural semantics for lazy evaluation. *20th Symposium on Principles of Programming Languages*.

- Leroy, X. (1990) The ZINC Experiment: An Economical Implementation of the ML Language. *Technical Report 117*, INRIA, France.
- Lescanne, P. (1994) From lambda-sigma to lambda-epsilon: a journey through calculi of explicit substitutions. *21st Symposium on Principles of Programming Languages*.
- Maranget, L. (1991) Optimal derivation in orthogonal rewriting systems and in weak lambda calculi. *18th Symposium on Principles of Programming Languages*.
- Minamide, Y., Morriset, G. and Harper, R. (1996) Typed closure conversion. *23rd Symposium on Principles of Programming Languages*.
- Pagano, B. (1996) Bi-simulation de machines abstraites en lambda-sigma-calcul. *Technique et science informatique*, **15**(7), 953–975. (Éditions Hermès (In French).)
- Plotkin, G. D. (1977) LCF Considered as a programming language. *Theor. Comput. Sci.*, **5**, 225–255.
- Peyton Jones, S. L. (1987) *The implementation of Functional Programming Languages*. Prentice-Hall.
- Rittri, M. (1988) Proving the Correctness of a Virtual Machine by a Bisimulation. *PhD thesis*, University of Göteborg and Chalmers University of Technology.