Functional Back-Ends within the Lambda-Sigma Calculus

Thérèse Hardin, Luc Maranget, Bruno Pagano

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Functional Back-Ends within the Lambda-Sigma Calculus

Thérèse Hardin, Luc Maranget, Bruno Pagano

N ° 3034

Novembre 1996

THÈME 1

ISSN 0249-6399

Functional Back-Ends within the Lambda-Sigma Calculus

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

Thème 1 — Réseaux et systèmes

Projet Para

Rapport de recherche n°3034 — Novembre 1996

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

Key-words: Weak lambda calculus, explicit substitutions, semantics of functional languages, compilation of functional languages

(Résumé : tsvp)

This work was partially supported by the ESPRIT Basic Research Project 6454 - CONFER.

*Inria Rocquencourt et LITP, tour 45-55 2ème étage porte 208, 4 Place Jussieu, 75252 Paris Cedex 05, Therese.Hardin@inria.fr

**Luc.Maranget@inria.fr

***LITP, tour 45-55 2ème étage porte 203 gauche, 4 Place Jussieu, 75252 Paris Cedex 05, pagano@cadillac.ibp.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Compilation et exécution fonctionnelles dans le Lambda-Sigma calcul

Résumé : Nous définissons un λ-calcul faible $\lambda\sigma_w$ comme un sous-système du λ-calcul avec substitutions explicites $\lambda\sigma_{\Uparrow}$. Notre système $\lambda\sigma_w$ est une abstraction du langage de bas niveau produit par les compilateurs de langages fonctionnels, au sens où le λ-calcul est une abstraction des langages fonctionnels eux-mêmes. En outre, le calcul fort des substitutions explicites $\lambda\sigma_{\Uparrow}$ pourrait bien être une bonne théorie syntaxique pour décrire et prouver la correction de compilateurs fonctionnels simplifiés. Dans ce travail, nous donnons quatre exemples de cette démarche en prouvant des versions simplifiées de compilateurs et d'environnements d'exécution. Les compilateurs sont modélisés par des réécritures de $\lambda\sigma$-termes et les environnements d'exécution par quatre machines abstraites : la machine de Krivine, la SECD, la FAM et la CAM. Notre formalisme nous permet de presenter la première preuve connue de la FAM.

Mots-clé : Lambda calcul faible, substitutions explicites, sémantique des langages fonctionnels, compilation des langages fonctionels

Functional Back-Ends within the Lambda-Sigma Calculus

1 Introduction

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

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

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

Closures are naturally expressed in the λ-calculus with explicit substitutions [1, 10] as a term $(\lambda M)[s]$, where M is a term standing for a piece of code, and s is a substitution, that is, an environment, collecting the values of free variables. We now need some rules to compute on closures.

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

In this paper, we first introduce the weak $\lambda\sigma$-calculus and give a unified presentation of abstract machines. Afterwards, we show that the basic operations of abstract machines correspond to certain rewriting steps in the weak $\lambda\sigma$-calculus. More precisely, the deterministic evaluation strategy implemented by an abstract machine is identified as a rewriting strategy in $\lambda\sigma_w$. We make this correspondence fully explicit for the Krivine machine and the FAM. The latter example involves a true compilation of the input λ-term to a $\lambda\sigma$-term. Thereby, we give the first known proof of the correctness of a FAM-based compiler and runtime system. Other execution models are briefly discussed in section 7.

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

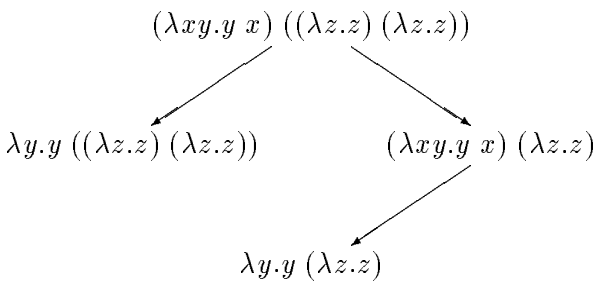
2 Preliminaries

2.1 The lambda-calculus with explicit substitutions

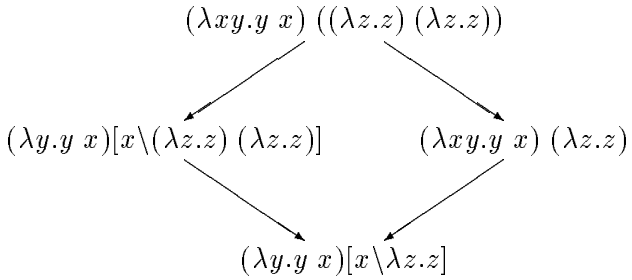
The traditional weak λ-calculus is an attempt to model the execution of machine code within the λ-calculus; it conforms with the basic intuition that functions, once compiled, are code and cannot change (otherwise, there would be no compilation). A tentative definition of weak reduction is thus to *suppress*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

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

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



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



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

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

$$\begin{array}{lrcl} \textsc{terms:} & M & ::= & \mathtt{n} \mid (MM) \mid \lambda M \mid M\,[s], \quad \text{with } n \geq 1 \\ \textsc{substitutions:} & s & ::= & \mathrm{id} \mid \uparrow \mid M \cdot s \mid s \circ s \end{array}$$

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

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

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

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

$$\begin{array}{lrcl} (\mathrm{App}) & (M_1\ M_2)\,[s] & \to & (M_1\,[s]\ M_2\,[s]) \\ (\mathrm{FVar}) & \mathtt{1}\,[M \cdot s] & \to & M \\ (\mathrm{RVar}) & \mathtt{n{+}1}\,[M \cdot s] & \to & \mathtt{n}\,[s] \\ (\mathrm{Clos}) & (M\,[s])\,[t] & \to & M\,[s \circ t] \\ (\mathrm{AssEnv}) & (s \circ t) \circ u & \to & s \circ (t \circ u) \\ (\mathrm{MapEnv}) & (M \cdot s) \circ t & \to & M\,[t] \cdot (s \circ t) \\ (\mathrm{ShiftCons}) & \uparrow \circ (M \cdot s) & \to & s \\ (\mathrm{IdL}) & \mathrm{id} \circ s & \to & s \end{array}$$

There is one rule per term construct in the algebra of $\lambda\sigma$-terms, except for λ. The propagation of substitutions through application nodes and accesses inside list-structured substitutions are handled by the first three rules above in a straightforward manner. The case of functions is more subtle. In the simplest case of the so-called "shared environment machines" (Krivine machine or SECD, for instance), functions are compiled as abstractions — i.e., as code —, and execution will only pair

these abstractions with an environment, producing closures. The "copied environment machines", such as the FAM, are more sophisticated: a function λM is compiled into a closure $(\lambda M')\,[s]$, where s collects the references of M to a global environment, whereas the free variables in $\lambda M'$ refer only to s. At run-time, applying some substitution t to the closure $(\lambda M')\,[s]$ will result in applying t to s, thus composing the two substitutions into one new substitution $s \circ t$ — as illustrated by the rewriting rule (Clos). Hence, we need rules to substitute inside substitutions; in other words, rules to compose substitutions. These rules are the remaining four rules above. Here again, there is one rule per term construct in the sort of substitutions.

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

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

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

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

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

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

Figure 1: 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 σ_w plus the strong substitution

rules of figure 1. With respect to strong substitution, $\lambda\sigma$-closures are not values any more, since any $\lambda\sigma$-closure $(\lambda M)\,[s]$ now immediately reduces to $\lambda(M\,[\Uparrow(s)])$, by the rule (Lambda). Most of the other rules of $\sigma_{\Uparrow}$ explicit the de Bruijn indices adjustments. The remaining two rules —(IdR) and (Id)— define the substitution id as the identity. As it can be expected, $\sigma_{\Uparrow}$ is a terminating and confluent rewriting system [10].

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

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

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

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

$$((\lambda M_1)\,[s]\ M_2) \overset{\text{(Lambda)}}{\longrightarrow} (\lambda(M_1\,[\Uparrow(s)])\ M_2) \overset{\text{(BetaStrong)}}{\longrightarrow} (M_1\,[\Uparrow(s)])\,[M_2 \cdot \mathrm{id}] \overset{\text{(Clos)}}{\longrightarrow}$$
$$M_1\,[\Uparrow(s) \circ (M_2 \cdot \mathrm{id})] \overset{\text{(LiftEnv)}}{\longrightarrow} M_1\,[M_2 \cdot (s \circ \mathrm{id})] \overset{\text{(IdR)}}{\longrightarrow} M_1\,[M_2 \cdot s]$$

Thus, as a subsystem of the strong system $\lambda\sigma_{\Uparrow}$, the weak system $\lambda\sigma_w$ is also correct with respect to the λ-calculus. This correctness is to be understood as follows: given any $\lambda\sigma$-term M, the $\sigma_{\Uparrow}$-normal form $\sigma_{\Uparrow}(M)$ is a λ_{DB}-term. Furthermore, if M reduces to M' by the rule (StrongBeta), then $\sigma_{\Uparrow}(M)$ β-reduces to $\sigma_{\Uparrow}(M')$ in one or more steps. Conversely, β-reduction in λ_{DB} can be simulated by a (StrongBeta) rewriting step followed by $\sigma_{\Uparrow}$-normalization. Again, refer to [10][Section 5] for details and proofs.

In this paper, functional programs are modelized as closed λ-terms. More precisely, given a λ-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:

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

By definition, a λ_{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 $M \overset{\lambda\sigma_{\Uparrow}}{\longrightarrow} M'$, the $\lambda\sigma$-term M' is also closed.*

Proof: The property holds for λ_{DB}-terms and β-reduction. It smoothly extends to $\lambda\sigma$-terms and $\lambda\sigma_{\Uparrow}$ reduction. □

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:

$$\textsc{code} \ ::= \ () \quad | \quad \textsc{instruction};\textsc{code}$$

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

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

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

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.

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

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

At this point, the puzzled reader may have a look at the machine descriptions in the following four sections 3, 4, 5, and 6. keeping 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, ..., x_n, the stack $S = x_n : \cdots : x_2 : x_1 : ()$ is simply written $x_n : \cdots : x_2 : x_1$. A subsequence $x_j : \cdots : x_i$ is written $\overrightarrow{x_{j,i}}$. For instance, $\overrightarrow{x_{n,n-i}} : S$ is a stack with the i elements x_n, x_{n-1}, ..., x_{n-i} standing on top of it.*

2.3 Implementation of a strategy by a machine

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

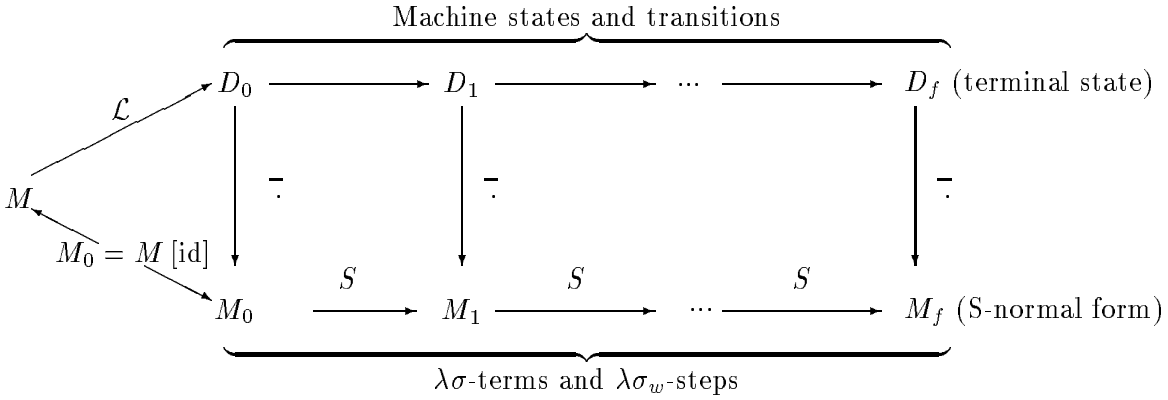
We present a simplified setting, where a bisimulation is given by two partial functions, the compile-and-load function $\mathcal{L}$ that translates $\lambda\sigma$-terms into machine states, and the decompilation function $\overline{\cdot}$ that translates machine states into $\lambda\sigma$-terms.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

1. *Initial condition:* Let M be a $\lambda\sigma$-term such that $\mathcal{L}(M)$ exists. Then $\overline{\mathcal{L}(M)} = M\,[\mathrm{id}]$.
2. *The machine follows the strategy S:* If $D_1 \rightarrow D_2$ and $\overline{D_1}$ exists, then $\overline{D_2}$ exists and we have either $\overline{D_2} = \overline{D_1}$ —and we say the machine performs a *silent transition*— or $\overline{D_1} \xrightarrow{\text{S}} \overline{D_2}$.
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 condition 1 is justified by the strong rule (Id): $M\,[\mathrm{id}] \rightarrow M$ 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 the strong $\lambda\sigma$-calculus to assert a correctness property.

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

The following diagram summarizes the bisimulation conditions:



This diagram illustrates the ideal case when they are no silent transitions.

3 The Krivine Machine

As a gentle introduction to our framework, we describe the Krivine Machine [8]. This machine is very simple:

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

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

A λ_{DB}-term is compiled as follows:

$$\begin{array}{rcl} [\![\, \mathtt{n} \,]\!] & = & \mathsf{Access}(n) \\ [\![\, \lambda N \,]\!] & = & \mathsf{Grab}; [\![\, N \,]\!] \\ [\![\, (N_1\ N_2) \,]\!] & = & \mathsf{Push}([\![\, N_2 \,]\!]); [\![\, N_1 \,]\!] \end{array}$$

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

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

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

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

and environments:

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

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

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

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 $\overline{[\![\, N \,]\!]} = N$, *for any* λ_{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* $D \to D'$. *Then* $\overline{D'}$ *exists and we have the reduction* $\overline{D} \xrightarrow{\lambda\sigma_w *} \overline{D'}$.

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

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

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. □

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

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

$$(N_1\ N_2)\,[s] \xrightarrow{\mathrm{K}} (N_1\,[s]\ N_2\,[s]) \qquad\qquad ((\lambda N)\,[s]\ M) \xrightarrow{\mathrm{K}} N\,[M \cdot s]$$

$$\frac{M_1 \xrightarrow{\mathrm{K}} M_1'}{(M_1\ M_2) \xrightarrow{\mathrm{K}} (M_1'\ M_2)}$$

It remains to show condition 3 on terminal states.

Lemma 4 *Let D be a reachable, terminal state, then $\overline{D}$ is a K-normal form:*

Proof: The Krivine machine may stop for two reasons:

- When $D = (\mathsf{Grab}; C/e)$. Then, we get $\overline{D} = (\lambda\overline{C})\,[\overline{e}]$, which is a $\lambda\sigma$-closure and a K-normal form.
- When $D = (\mathsf{Access}(m)/()) :: D_0$, i.e, when an access fails. Then, $\overline{D} = (\ldots(\mathtt{m}\ \overline{f_{n-1}})\ldots\ \overline{f_1})$, 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 $\sigma_{\Uparrow}(\overline{D})$ admits at least $\mathtt{m}$ as a free variable. Thus, by lemma 1, this case can occur only when the initial program is not a closed λ_{DB}-term. □

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

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

4 The SECD machine

4.1 SECD in the λ_{DB}-calculus

The original SECD machine of [16] used named variables. In our presentation, we consider a slightly modified SECD machine that reduces λ_{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 λ_{DB}-term or a new symbol @.

$$\textsc{instruction} \quad := \quad \lambda_{\mathrm{DB}}\textsc{-term} \mid @$$

An argument stack AS is a list of closures.

$$\textsc{stack} := () \mid \textsc{closure} : \textsc{stack}$$

Finally, a frame of the SECD machine is a (AS, e, C) triple:

$$\textsc{frame} := (\textsc{stack}\ ,\ \textsc{environment}\ ,\ \textsc{code})$$

Thus, a SECD state is written $D = (AS_n, e_n, C_n) :: \cdots :: (AS_2, e_2, C_2) :: (AS_1, e_1, C_1)$.

The transition rules are as follows:

$$\begin{array}{rcl}
(AS, e, (N_1\ N_2); C) :: D & \xrightarrow{\mathbf{app}} & (AS, e, N_2; N_1; @; C) :: D \\
(AS, e, \lambda N; C) :: D & \xrightarrow{\mathbf{lam}} & ((N/e) : AS, e, C) :: D \\
((N_0/e_0) : f : AS, e, @; C) :: D & \xrightarrow{@} & ((), f \cdot e_0, N_0) :: (AS, e, C) :: D \\
(f, e, ()) :: (AS', e', C') :: D & \xrightarrow{\mathbf{dump}} & (f : AS', e', C') :: D \\
(AS, f_1 \cdots f_n \cdot e, \mathtt{n}; C) :: D & \xrightarrow{\mathbf{var}} & (f_n : AS, f_1 \cdots f_n \cdot e, C) :: D
\end{array}$$

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

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.

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

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

$$\text{Stacks: } \overline{f_n : f_{n-1} : \cdots : f_1} \quad = \quad \overline{f_n} : \overline{f_{n-1}} : \cdots : \overline{f_1}$$

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 λ_{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:

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

Observe that a L-value $(\lambda N)\,[e]$ is a $\lambda\sigma_w$-normal form, since N is a λ_{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 Φ decompiles the closures appearing inside environments and stacks.

$$\Phi((AS_n, e_n, C_n) :: \cdots :: (AS_1, e_1, C_1)) = (\overline{AS_n}, \overline{e_n}, C_n) :: \cdots :: (\overline{AS_1}, \overline{e_1}, C_1)$$

Given a state $D = (AS_n, e_n, C_n) :: \cdots :: (AS_1, e_1, C_1)$, we write $\Phi(D) = (S_n, s_n, C_n) :: \cdots :: (S_1, s_1, C_1)$, 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:

$$(M, s, ()) \Downarrow M \ \text{(Res)} \qquad ((), s, N) \Downarrow N\,[s] \ \text{(Code)}$$

$$\frac{(S, s, C) \Downarrow M}{(S, s, C; N; @) \Downarrow (N\,[s]\ M)}\ \text{(AppRight)} \qquad \frac{(S, s, C) \Downarrow M_1 \quad (\text{where } S \neq ())}{(S : M_2, s, C; @) \Downarrow (M_1\ M_2)}\ \text{(AppLeft)}$$

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

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).

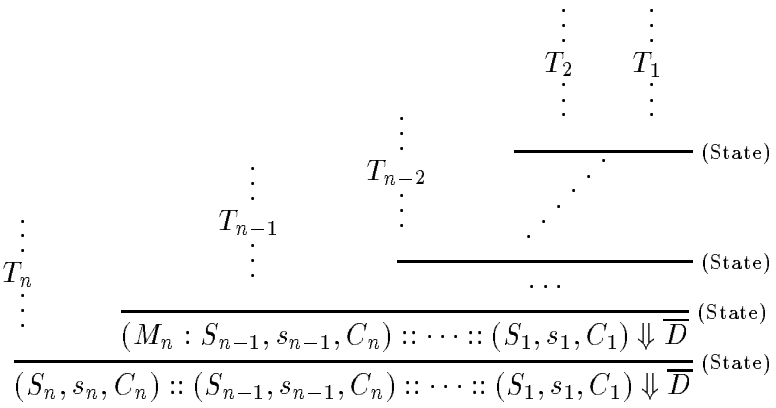


Figure 2: Structure of the proof of $\Phi(D) \Downarrow \overline{D}$.

The basic idea of our decompilation procedure is as follows: in a triple (S, s, C) such that $(S, s, C) \Downarrow M$ 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 (S, s, C) 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 (S, s, C) (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 $D = (AS, e, C; @)$. We get $\Phi(D) = (S, s, C; @)$ 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 λ_{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:*

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

2. *The $\lambda\sigma$-term $\overline{D}$ is unique.*

Proof: The first proposition is easy, once one understands the structure of the proof of $\Phi(D) \Downarrow \overline{D}$. Consider a state $D = (AS_n, e_n, C_n) :: (AS_{n-1}, e_{n-1}, C_{n-1}) :: \cdots :: (AS_1, e_1, C_1)$, we get $\Phi(D) =$

$(S_n, s_n, C_n) :: (S_{n-1}, s_{n-1}, C_{n-1}) :: \cdots :: (S_1, s_1, C_1)$. The proof tree of $\Phi(D) \Downarrow \overline{D}$ is schematized in figure 2. In this figure, T_n stands for a proof tree whose conclusion is $(S_n, s_n, C_n) \Downarrow M_n$. 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 $(S, s, C) \Downarrow M$ 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 $i \in [1 \ldots n-1]$, the proof tree T_i admits the judgment $(M_{i+1} : S_i, s_i, C_i) \Downarrow M_i$ as its conclusion. Thus, given any judgment $(S, s, C) \Downarrow M$ 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 (S, s, C), there exists at most one proof tree of a judgment $(S, s, C) \Downarrow M$. 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 $(S, s, C'; @) = (S' : M'_2, s, C''; N''; @)$, such that the following two proof nodes hold:

$$\frac{(S', s, C''; N'') \Downarrow M'_1}{(S' : M'_2, s, C''; N''; @) \Downarrow (M'_1\ M'_2)}\ (\text{AppLeft}) \qquad \frac{(S' : M'_2, s, C'') \Downarrow M''}{(S' : M'_2, s, C''; N''; @) \Downarrow (N''[s]\ M'')}\ (\text{AppRight})$$

However, the judgment $(S', s, C''; N'') \Downarrow M'_1$ can only be proved by the axiom (Code), because N'' is a λ_{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) $S' \neq ()$. □

In the following, we assume that all the proof trees we consider are proofs of judgments $\Phi(D) \Downarrow \overline{D}$. Thus they have the structure pictured in figure 2, where all the stacks in T_n hold L-values and all the elements except, possibly, the topmost in the stacks in $T_{n-1}, \ldots, T_1$ 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 λ_{DB}-term. Then, we have $\overline{\mathcal{L}(N)} = N\,[id]$*

Proof: Quite straightforward, since we have $\mathcal{L}(M) = ((), (), N)$ and thus we get $((), \overline{()}, N) \Downarrow N\,[\mathrm{id}]$ by the decompilation rule (Code). □

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 $D \to D'$. First consider the state $D = ((), e, (N_1\ N_2))$, by the decompilation rule (Code), we get $\overline{D} = (N_1\ N_2)\,[\overline{e}]$. Moreover, we have the SECD transition $((), e, (N_1\ N_2)) \xrightarrow{\mathbf{app}} ((), e, N_2; N_1; @)$. The $\lambda\sigma$-term $\overline{D'} = (N_1[\overline{e}]\ N_2[\overline{e}])$ is computed by the following proof tree:

$$\frac{((), \overline{e}, N_2) \Downarrow N_2\,[\overline{e}]\ (\text{Code})}{((), \overline{e}, N_2; N_1; @) \Downarrow (N_1\,[\overline{e}]\ N_2\,[\overline{e}])}\ (\text{AppRight})$$

Hence we get the strategy axiom

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

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

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

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

Notice that the remaining two transitions **lam** and **dump** do not suggest any new axiom, since, for these transitions $D \to D'$, we get $\overline{D} = \overline{D'}$. 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 $D = (AS, e, C; N; @)$ such that $D \to D'$ with $D' = (AS', e, C'; N; @)$ (i.e, the code C is not empty and we do not consider the case of a transition @). Thus we get:

$$\frac{(\overline{AS}, \overline{e}, C) \Downarrow M}{(\overline{AS}, \overline{e}, C; N; @) \Downarrow (N\,[\overline{e}]\ M)}\ \text{(AppRight)} \qquad \text{and} \qquad \frac{(\overline{AS'}, \overline{e}, C') \Downarrow M'}{(\overline{AS'}, \overline{e}, C'; N; @) \Downarrow (N\,[\overline{e}]\ M')}\ \text{(AppRight)}$$

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

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

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:

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

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

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

Lemma 7 *Consider any provable judgment $(P : S, s, C) :: D \Downarrow M$ and any $\lambda\sigma$-term P', such that $P \xrightarrow{\mathrm{L}} P'$. Then, there exists a $\lambda\sigma$-term M' such that the judgment $(P' : S, s, C) :: D \Downarrow M'$ holds and we have $M \xrightarrow{\mathrm{L}} M'$.*

Proof: Simple induction on the proof of $(P : S, s, C) :: D \Downarrow M$.

1. The base case of rule (Res) is obvious.

2. In the case of the rule (AppRight), we have:

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

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

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

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:

$$\frac{(P : S, s, C) \Downarrow M_1}{(P : S : M_2, s, C; @) \Downarrow (M_1\ M_2)} \quad \text{and} \quad \frac{(P' : S, s, C) \Downarrow M_1'}{(P : S : M_2, s, C; @) \Downarrow (M_1'\ M_2)}$$

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 $D = (S', s', C') :: D'$, we have:

$$\frac{(P : S, s, C) \Downarrow M_1 \quad (M_1 : S', s', C') :: D' \Downarrow M}{(P : S, s, C) :: D \Downarrow M}$$

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

Then, our idea is to interpret the instruction to be executed next as a $\lambda\sigma$-term to be plugged in the same context $(X : S, s, C) :: D$ 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 λ_{DB}-term.

Lemma 8 *Let N be any λ_{DB}-term, such that the judgment $(S, s, N; C) :: D \Downarrow M$ holds. Then the judgment $(N\,[s] : S, s, C) :: D \Downarrow M$ holds.*

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

$$((), s, N) \Downarrow N\,[s]\ \text{(Code)} \qquad\qquad (N\,[s], s, ()) \Downarrow N\,[s]\ \text{(Res)}$$

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

$$\frac{(P, s, ()) \Downarrow P}{(P, s, N; @) \Downarrow (N\,[s]\ P)}\ \text{(AppRight)} \qquad\qquad \frac{(N\,[s], s, ()) \Downarrow N\,[s]}{(N\,[s] : P, s, @) \Downarrow (N\,[s]\ P)}\ \text{(AppLeft)}$$

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

- If we have the proof node:

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

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

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

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

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

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

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

□

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

Lemma 9 *If $(M_1 : M_2 : S, s, @; C) :: D \Downarrow M$ holds, then $((M_1\ M_2) : S, s, C) :: D \Downarrow M$ 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 $\overline{D}$ exists and $D \to D'$ by one transition step. Then, $\overline{D'}$ exists and we have two possibilities:*

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

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

$$\frac{(\overline{f}, \overline{e}, ()) \Downarrow \overline{f} \qquad (\overline{f} : \overline{AS'}, \overline{e'}, C') :: \Phi(D_0) \Downarrow M}{(\overline{f}, \overline{e}, ()) :: (\overline{AS'}, \overline{e'}, C') :: \Phi(D_0) \Downarrow M}\ (\text{State})$$

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

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

$$(((\lambda N_0)\,[\overline{e_0}]\ \overline{f}) \xrightarrow{\text{L}} N_0\,[\overline{f} \cdot \overline{e_0}]$$

Hence the result, by lemma 7.

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

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

$$(\mathtt{n}\,[\overline{e}] : \overline{AS}, \overline{e}, C) :: \Phi(D_0) \Downarrow \overline{D}$$

On the other hand, we get:

$$(\overline{f_n} : \overline{AS}, \overline{e}, C) :: \Phi(D_0) \Downarrow \overline{D'}$$

Hence the result, by the strategy axiom (Var^n) and lemma 7.

2. If N is an abstraction λN_0, then, on the one hand, by lemma 8, we get:

$$((\lambda N_0)\,[\overline{e}] : \overline{AS}, \overline{e}, C) :: \Phi(D_0) \Downarrow \overline{D}$$

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

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

$$((N_1\ N_2)\,[\overline{e}] : \overline{AS}, \overline{e}, C) :: \Phi(D_0) \Downarrow \overline{D}$$

Here, we have $D' = (AS, e, N_2; N_1; @; C) :: D_0$. 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:

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

Hence, the result, by the strategy axiom (App) and by lemma 7. □

Now, we prove the final condition 3.

Lemma 11 *Let N be λ_{DB}-term, and D be a terminal state, with $\mathcal{L}(N) \rightarrow^* D$. Then, $\overline{D}$ is a L-normal form.*

Proof: Let us state $D = (AS_n, e_n, C_n) :: \cdots :: (AS_1, e_1, C_1)$. First observe that, by lemma 10, $\overline{D}$ 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 $\overline{D} = \overline{(f, (), ())} = \overline{f}$, 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 $(\overline{AS_n}, \overline{e_n}, @; C'_n) \Downarrow M_n$, which in turn must include a proof of $(S, \overline{e_n}, @) \Downarrow P$, 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. $C_n = \mathtt{m}; C'_n$ and $e_n = f_1 \cdots f_k \cdot \mathrm{id}$, with $k < m$. Then, $\overline{D}$ is a L-failure term W, where L-failure terms are defined as follows:

$$\begin{array}{rcll} W & ::= & \mathtt{m}\,[\overline{f_1} \cdots \overline{f_k} \cdot \mathrm{id}] & \text{with } k < m \\ & | & M\ W \ | \ W\ \overline{f} & \end{array}$$

where M stands for a $\lambda\sigma$-term and $\overline{f}$ for a L-value. The subterm $\mathtt{m}\,[\overline{f_1} \cdots \overline{f_k} \cdot \mathrm{id}]$ is a L-normal form which is not a L-value. It stands in L-redex position inside $\overline{D}$. Thus, $\overline{D}$ is a L-normal form which is not a value. Moreover, the $\sigma_{\Uparrow}$-normal form $\sigma_{\Uparrow}(\overline{D})$ is not a closed λ_{DB}-term, since it admits at least one free variable $\mathtt{m{-}k} = \sigma_{\Uparrow}(\mathtt{m}\,[\overline{f_1} \cdots \overline{f_k} \cdot \mathrm{id}])$. Therefore, by lemma 1, this case may only occur when the initial program is not a closed λ_{DB}-term. □

Lemma 12 *The SECD cannot perform infinitely many consecutive silent transitions.*

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

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

Now, given a transition $D \to D'$ 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. □

Finally, we conclude:

Theorem 2 *The SECD machine implements the L-strategy.*

s

5 The Functional Abstract Machine

5.1 Basics

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

The FAM has four instructions:

$$\begin{array}{lcll}
\textsc{instruction} & ::= & \mathsf{Local} & \\
& | & \mathsf{Global}(n) & (n \geq 1) \\
& | & \mathsf{Apply} & \\
& | & \mathsf{Fun}(n, \textsc{code}) & (n \geq 0)
\end{array}$$

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

$$\begin{array}{lcl}
\textsc{stack} & ::= & () \quad | \quad \textsc{closure} : \textsc{stack} \\
\textsc{frame} & ::= & (\textsc{stack}, \textsc{environment}, \textsc{code})
\end{array}$$

The transitions rules of the FAM are defined as follows:

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

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

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

$$\begin{aligned}(N\,[\mathrm{id}]\ P) &\overset{(\mathrm{Beta})}{\longrightarrow} (\mathtt{1}\ (\mathtt{5}\ \mathtt{7}))\,[P\cdot\mathrm{id}] \overset{\sigma_w}{\longrightarrow}{}^{*}\ P\ (\mathtt{4}\ \mathtt{6})\\ M\ P &\overset{(\mathrm{Beta})}{\longrightarrow} (\mathtt{1}\ (\mathtt{2}\ \mathtt{3}))\,[P\cdot\mathtt{4}\cdot\mathtt{6}\cdot\mathrm{id}] \overset{\sigma_w}{\longrightarrow}{}^{*}\ P\ (\mathtt{4}\ \mathtt{6})\end{aligned}$$

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

$$\begin{aligned}\mathcal{C}(\overrightarrow{n_{1,m}}, \mathtt{n_i}) &= \mathtt{i}\\ \mathcal{C}(\overrightarrow{n_{1,m}}, (N_1\ N_2)) &= (\mathcal{C}(\overrightarrow{n_{1,m}}, N_1)\,\mathcal{C}(\overrightarrow{n_{1,m}}, N_2))\\ \mathcal{C}(\overrightarrow{n_{1,m}}, \lambda N) &= \begin{cases}(\lambda\mathcal{C}(1 : p_1{+}1 : p_2{+}1 : \cdots : p_k{+}1, N))\,[\mathcal{C}(\overrightarrow{n_{1,m}}, \mathtt{p_1}\cdot\mathtt{p_2}\cdots\mathtt{p_k}\cdot\mathrm{id})]\\ \text{where } p_1 : p_2 : \cdots : p_k = \mathcal{F}_0(\lambda N)\end{cases}\\ \mathcal{C}(\overrightarrow{n_{1,m}}, N\cdot s) &= \mathcal{C}(\overrightarrow{n_{1,m}}, N)\cdot\mathcal{C}(\overrightarrow{n_{1,m}}, s)\\ \mathcal{C}(\overrightarrow{n_{1,m}}, \mathrm{id}) &= \uparrow^m\end{aligned}$$

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 $s = M_1 \cdots M_i \cdots M_m \cdot \mathrm{id}$, where M_i is the run-time value of the variable $\mathtt{n_i}$ of the input term.

The translation $(\lambda M)\,[t]$ of a λ-abstraction λN 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 λN. Thus, t is the list of the positions in s of these free variables. The substitution t ends with the new special substitution $\uparrow^m$, which ultimately discards s. This discarding operator is an ordinary $\lambda\sigma$-substitution:

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

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

$$\begin{aligned}\uparrow^m \circ (M_1 \cdot M_2 \cdots M_m \cdot \mathrm{id}) &\overset{(\mathrm{AssEnv})}{\longrightarrow}\\ \uparrow \circ (\uparrow^{m-1} \circ (M_1 \cdot M_2 \cdots M_m \cdot \mathrm{id})) &\overset{\sigma_w}{\longrightarrow}{}^{*} \quad \ldots\\ \uparrow \circ (M_m \cdot \mathrm{id}) &\overset{(\mathrm{ShiftCons})}{\longrightarrow} \mathrm{id}\end{aligned}$$

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 λ_{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 $N = \mathtt{n}$, 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 $n \in \mathcal{F}_{d+1}(N_0) \quad \Rightarrow \quad n = 1$ or $n-1 \in \mathcal{F}_{d+2}(N_0)$ by induction hypothesis. □

Lemma 14 *Let N be a λ_{DB}term. The following $\sigma_{\Uparrow}$ equality holds, for any substitution s and integer d:*

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

(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 the sections 3.2 and 4.1 of [10], $\Uparrow^d(s)$ and $\mathtt{1}\cdot\mathtt{2}\cdots\mathtt{d}\cdot(s\circ\uparrow^d)$ perform the same replacements on terms. Operationally, both substitutions behave as s after it went through d nested levels of λ's. □

Proposition 1 (Correctness of compilation) *The $\mathcal{C}$ compilation scheme never fails and is correct. More precisely, given a λ_{DB}-term N, let $\overrightarrow{n_{1,m}}$ such that $\mathcal{F}_0(N) \subseteq \overrightarrow{n_{1,m}}$. Then $\mathcal{C}(\overrightarrow{n_{1,m}}, N)$ is a $\lambda\sigma$-term M, such that $M\,[\mathtt{n_1}\cdots\mathtt{n_m}\cdot id]$ and N are $\sigma_{\Uparrow}$-equivalent.*

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

$$M\,[\mathtt{n_1}\cdots\mathtt{n_m}\cdot s] \xrightarrow{\sigma_{\Uparrow}}{}^{*}\ N$$

The proof is by induction on N. In the proof, we state $t(\overrightarrow{n_{1,m}}, s) = \mathtt{n_1}\cdots\mathtt{n_m}\cdot s$.

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

$$\mathtt{i}\,[t(\overrightarrow{n_{1,m}}, s)] \xrightarrow{\sigma_w}{}^{*}\ \mathtt{n_i}$$

- 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 $\mathcal{F}_0(N_1) \subseteq \overrightarrow{n_{1,m}}$ and $\mathcal{F}_0(N_2) \subseteq \overrightarrow{n_{1,m}}$. Thus, we can apply the induction hypothesis and both $M_1 = \mathcal{C}(\overrightarrow{n_{1,m}}, N_1)$ and $M_2 = \mathcal{C}(\overrightarrow{n_{1,m}}, N_2)$ exist. So does $M = \mathcal{C}(\overrightarrow{n_{1,m}}, N) = (M_1\ M_2)$. Furthermore, we have

$$M\,[t(\overrightarrow{n_{1,m}}, s)] \xrightarrow{\sigma_{\Uparrow}}{}^{*}\ N$$

by the σ_w-rule (App) and by induction hypothesis.

- If $N = \lambda N_0$, then let $\overrightarrow{p_{1,k}}$ 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 $\overrightarrow{p_{1,k}} = \mathcal{F}_1(N_0)$ and thus, by lemma 13, we have $\mathcal{F}_0(N_0) \subseteq 1 : p_1{+}1 : \cdots : p_k{+}1$. Therefore, the $\lambda\sigma$-term $M_0 = \mathcal{C}(1 : p_1{+}1 : \cdots : p_k{+}1\,, N_0)$ exists, by induction hypothesis. Now, all the variables in the substitution $u = \mathtt{p_1} \cdots \mathtt{p_k} \cdot \mathrm{id}$ belong to $\mathcal{F}_0(N)$. Thus they can be compiled in the compile-time environment $\overrightarrow{n_{1,m}} \supseteq \mathcal{F}_0(N)$ and so does the substitution u. Let u_0 be $\mathcal{C}(\overrightarrow{n_{1,m}}\,, u)$. Finally, the $\lambda\sigma$-term $M = (\lambda N_0)\,[u_0]$ exists.

  Furthermore, since we have $\mathcal{C}(\overrightarrow{n_{1,m}}\,, \mathtt{p_i})\,[t(\overrightarrow{n_{1,m}}, s)] \xrightarrow{\sigma_{\Uparrow}}{}^{*}\ \mathtt{p_i}$ (by induction) and $\uparrow^m \circ\, t(\overrightarrow{n_{1,m}}, s) = s$ (by σ_w-reduction), we get:

  $$u_0 \circ t(\overrightarrow{n_{1,m}}, s) \xrightarrow{\sigma_{\Uparrow}}{}^{*}\ \mathtt{p_1} \cdots \mathtt{p_k} \cdot s$$

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

  $$M\,[t(\overrightarrow{n_{1,m}}, s)] \xrightarrow{(\mathrm{Clos})} (\lambda M_0)\,[u_0 \circ t(\overrightarrow{n_{1,m}}, s)] \xrightarrow{\sigma_{\Uparrow}}{}^{*}\ (\lambda M_0)\,[\mathtt{p_1} \cdots \mathtt{p_k} \cdot s]$$

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

  $$M_0\,[t(\overrightarrow{n_{1,m}}, s)] \xrightarrow{\sigma_{\Uparrow}}{}^{*}\ \lambda(M_0\,[\Uparrow (\mathtt{p_1} \cdots \mathtt{p_k} \cdot s)])$$

  Let us consider N_0', the $\sigma_{\Uparrow}$-normal form of M_0, which is a λ_{DB}-term ([10][Lemma 4.8]). By definition of reduction, we get:

  $$M_0\,[t(\overrightarrow{n_{1,m}}, s)] \xrightarrow{\sigma_{\Uparrow}}{}^{*}\ \lambda(N_0'\,[\Uparrow (\mathtt{p_1} \cdots \mathtt{p_k} \cdot s)])$$

  Our lemma 14 applies here and we get:

  $$M_0\,[t(\overrightarrow{n_{1,m}}, s)] =_{\sigma_{\Uparrow}} \lambda(N_0'\,[1 \cdot ((\mathtt{p_1} \cdots \mathtt{p_k} \cdot s) \circ \uparrow)])$$

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

  $$M_0\,[t(\overrightarrow{n_{1,m}}, s)] =_{\sigma_{\Uparrow}} \lambda(N_0'\,[1 \cdot \mathtt{p_1}{+}1 \cdots \mathtt{p_k}{+}1 \cdot (s \circ \uparrow)]) \tag{1}$$

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

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

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

  $$N_0'\,[1 \cdot \mathtt{p_1}{+}1 \cdots \mathtt{p_k}{+}1 \cdot (s \circ \uparrow)] \xrightarrow{\sigma_{\Uparrow}}{}^{*}\ N_0 \tag{2}$$

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

  $$M\,[t(\overrightarrow{n_{1,m}}, s)] \xrightarrow{\sigma_{\Uparrow}}{}^{*}\ \lambda N_0 = N$$

  □

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

Corollary 15 *Let N be a closed λ_{DB}-term (i.e., a program) and M be its compilation $\mathcal{C}(\emptyset\,, N)$. The initial condition $M\,[id] =_{\sigma_{\Uparrow}} N$ 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, $M = \mathcal{C}(\overrightarrow{n_{1,k}}, N)$ itself can be characterized by a predicate $\mathcal{P}_k$ that is defined as follows:

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

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 λ_{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 $[\![\ ]\!]$ generates FAM code.

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

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

5.2 The decompilation

First, we inverse the compilation procedure $[\![\ ]\!]$. 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".

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

$$\frac{C_2 \Downarrow^m M_2 \quad C_1 \Downarrow^m M_1}{C_2; C_1; \mathsf{Apply} \Downarrow^m (M_1\ M_2)}\ \text{(Apply)}$$

$$\frac{C_1 \Downarrow^m M_1 \quad \cdots \quad C_n \Downarrow^m M_n \quad C_0 \Downarrow^{n+1} M_0}{C_1; \cdots; C_n; \mathsf{Fun}(n, C_0) \Downarrow^m (\lambda M_0)\,[M_1 \cdots M_n \cdot \uparrow^m]}\ \text{(Fun}(n)\text{)}$$

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

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

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:

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

Lemma 16 *Let M be a $\lambda\sigma$-term such that $\mathcal{P}_m(M)$ holds. Then we have $[\![\, M \,]\!] \Downarrow^m M$.*

Proof: Obvious induction on M. □

The decompilation of machine states is best understood as a two-stage process. In a first step, we translate the frames (AS, e, C) into triples (S, s, C). Roughly, the stack S is the translation of the argument stack AS and the substitution s is the translation of the environment e. The translation of the bottom element of AS is incorporated either in S or in s. In the latter case, this bottom element is the argument of a pending function call. The following procedure Φ operates this first transformation:

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

In a second step, the value of $\overline{D}$ is computed by proving a judgment $\Phi(D) \Downarrow \overline{D}$, with the following axioms and inference rules:

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

$$\frac{(S, s, C_2) \Downarrow M_2 \quad C_1 \Downarrow^m M_1}{(S,\ s,\ C_2; C_1; \mathsf{Apply}) \Downarrow (M_1\,[s]\ M_2)}\ \text{(AppRight)} \qquad \frac{(S, s, C_1) \Downarrow M_1}{(S{:}M_2,\ s,\ C_1; \mathsf{Apply}) \Downarrow (M_1\ M_2)}\ \text{(AppLeft)}$$

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

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

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

Our decompilation procedure is a partial function from syntactic FAM states to $\lambda\sigma$-terms. The whole purpose of this section is to show that decompilation is total on accessible FAM states.

At first, our state decompilation procedure may seem a bit complicated. However, it is a simple extension of the code decompilation procedure. In a triple (S, s, C), like in the case of the SECD, the stack S is the list of the subterms that have already been computed by the machine, whereas the code segment C encodes the subterms that are still to be computed. Decompilation rules combine these two sets of subterms. The rules (Res), (Code), (AppLeft), (AppRight) and (State) are basically the same as the homonymous decompilation rules of the SECD.

The new rule (Fun(n,i)) performs the decompilation of a triple (S, s, C'; $\mathsf{Fun}(n, C_0)$). This operation resembles the decompilation of a triple (S, s, C'; Apply). More specifically, the $\lambda\sigma$-substitution $M_1 \cdots M_i \cdot ((M_{i+1} \cdots M_n \cdot \uparrow^m) \circ s)$ stands for a closure environment that is not fully computed yet, where the term M_i is the subterm being currently computed, while the subterms $M_1, \ldots, M_{i-1}$ are fully reduced and the computation of the subterms $M_{i+1}, \ldots, M_n$ is yet to be started.

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

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 $(S, s, C) \Downarrow M$, where S is a non-empty stack, are strict suffixes.*
2. *Strict suffixes cannot be decompiled.*

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

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

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., $\Phi(D) = (S, s, C)$.

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

Otherwise, we have two subcases, either C ends by an Apply or by a $\mathsf{Fun}(n, C_0)$ instruction. As far as ambiguity is concerned, these subcases are the same. Thus, for instance, we assume $\Phi(D) = (S, s, C'; \mathsf{Fun}(n, C_0))$. If the stack S is empty, then only the rule (Code) may apply 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 $C_i; C_{i+1}; \cdots; C_n$, where C_i is a strict suffix and $C_{i+1}, \ldots, C_n$ are decompilable code segments. In other words, the rule (Fun(n,i)) is applied unambiguously. □

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 $(S, s, C) \Downarrow M$,

such that all the $\lambda\sigma$-terms in S are C-values, except, possibly, the topmost one. Furthermore, let $D = (AS, e, C_0) :: D_0$ be a state and let be a judgment $(S, s, C) \Downarrow M$ that occurs inside the proof tree of $(\overline{AS}, \overline{e}, C_0) \Downarrow P$. Then *all* the $\lambda\sigma$-terms in S are C-values (see figure 2). 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 λ_{DB}-term. Let M be $\mathcal{C}(\emptyset, N)$, Then, we have the equality,*

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

Proof: By lemma 16, we have $[\![\, M \,]\!] \Downarrow^0 M$. Hence the result, by the decompilation rule (Code).

$$\frac{[\![\, M \,]\!] \Downarrow^m M}{((), \mathrm{id}, [\![\, M \,]\!]) \Downarrow M\,[\mathrm{id}]}\ (\mathrm{Code})$$

□

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, in order 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 $(P : S, s, C) :: D \Downarrow M$. 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 $(P : S, s, C) :: D \Downarrow M$ holds for some $\lambda\sigma$-term M, then there exists a $\lambda\sigma$-term M', such that judgment $(P' : S, s, C) :: D \Downarrow M'$ 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 $(P : S, s, C) :: D \Downarrow M$. First consider the case when D is empty:

1. The base case of rule (Res) is obvious.

2. Consider the rule (AppRight). We have

$$\frac{(P : S, s, C_2) \Downarrow M_2 \quad C_1 \Downarrow^m M_1}{(P : S, s, C_2; C_1; \mathsf{Apply}) \Downarrow (M_1\ M_2)}\ (\mathrm{AppRight})$$

By induction hypothesis, there exists M_2' such that $(P' : S, s, C_2) \Downarrow M_2'$. Therefore, we get:

$$\frac{(P' : S, s, C_2) \Downarrow M_2' \quad C_1 \Downarrow^m M_1}{(P' : S, s, C_2; C_1; \mathsf{Apply}) \Downarrow (M_1\ M_2')}\ (\text{AppRight})$$

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

$$\frac{(P : S, s, C_1) \Downarrow M_1}{(P : S : M_2, s, C_1; \mathsf{Apply}) \Downarrow (M_1\ M_2)}\ (\text{AppLeft})$$

Thus, by direct induction, there exists M_1' such that:

$$\frac{(P' : S, s, C_1) \Downarrow M_1'}{(P' : S : M_2, s, C_1; \mathsf{Apply}) \Downarrow (M_1\ M_2')}\ (\text{AppLeft})$$

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:

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

By induction there exists M_i', such that:

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

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

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

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

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

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

□

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 λ_{DB}-terms.

Lemma 22 *Let $D = (AS, e, I; C) :: D_0$ be a FAM state such that $\overline{D}$ exists and the execution of I is enabled. Then, $\Phi(D)$ can be written $(S_I : S, s, I; C) :: \Phi(D_0)$ and there exists a $\lambda\sigma$-term M_I such that $(S_I, s, I) \Downarrow M_I$.*

Proof: More precisely let us state $D = (AS_n, e_n, I; C_n) :: \cdots :: (AS_1, e_1, C_1)$. By hypothesis, $\Phi(D)$ exists and we have $\Phi(D) = (S_n, s_n, I; C_n) :: \cdots :: (S_1, s_1, C_1)$. Then, consider, for instance, the case of the instruction $\mathsf{Fun}(n_0, C_0)$. Since D can be decompiled, the proof tree T_n exists (i.e., the proof of $(S_n, s_n, I; C_n) \Downarrow M_n$, see figure 2). There is no other choice for T_n than to terminate by the following proof:

$$\frac{(\overline{f_{n_0}}, s_n, ()) \Downarrow \overline{f_{n_0}}\ (\text{Res}) \quad C_0 \Downarrow^{n_0+1} M_0}{(\overline{f_{n_0}} : \cdots : \overline{f_1}, s_n, \mathsf{Fun}(n_0, C_0)) \Downarrow (\lambda M_0)\,[\overline{f_1} \cdots \overline{f_n} \cdot (\uparrow^m \circ s_n)]}\ (\text{Fun}(n_0, n_0))$$

Let us state $S_I = \overline{f_{n_0}} : \cdots : \overline{f_1}$ and $M_I = (\lambda M_0)\,[\overline{f_1} \cdots \overline{f_n} \cdot (\uparrow^m \circ s_n)]$. By construction of proofs, S_I is a prefix of S_n . In other words, we get $S_n = S_I : S$.

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	$()$	$\mathtt{1}\,[s]$
Global(i)	$()$	$\mathtt{i{+}1}\,[s]$
Apply	$\overline{f_1} : \overline{f_2}$	$(\overline{f_1}\ \overline{f_2})$
Fun(n, C_0)	$\overline{f_{n_0}} : \cdots : \overline{f_1}$	$(\lambda M_0)\,[\overline{f_1} \cdots \overline{f_{n_0}} \cdot (\uparrow^m \circ s)]$

□

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

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

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

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 $(X : S, s, C) :: D$ sometimes 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 $(S_I, s, I) \Downarrow M_I$ and $(S_I : S, s, I; C) :: D \Downarrow M$ hold, where the $\lambda\sigma$-terms in S_I and S are C-values. Then, there exists a $\lambda\sigma$-term M', such that $(M_I : S, s, C) :: D \Downarrow M'$ holds. Furthermore, we have $M \triangleright\triangleright M'$, where $\triangleright\triangleright$ is a deterministic relation between $\lambda\sigma$-terms defined in figure 3.*

$$\mathtt{n}\,[s] \triangleright\triangleright \mathtt{n}\,[s] \qquad \frac{M_1 \text{ is a C-value} \quad M_2 \text{ is a C-value}}{(M_1\ M_2) \triangleright\triangleright (M_1\ M_2)}$$

$$\frac{M_1 \text{ is a C-value} \quad \ldots \quad M_n \text{ is a C-value}}{(\lambda M_0)\,[M_1 \cdots M_n \cdot(\uparrow^m \circ\, s)] \triangleright\triangleright (\lambda M_0)\,[M_1 \cdots M_n \cdot(\uparrow^m \circ\, s)]}$$

$$\frac{M_2\,[s] \triangleright\triangleright M_2'}{(M_1\ M_2)\,[s] \triangleright\triangleright (M_1\,[s]\ M_2')} \qquad \frac{M_2 \triangleright\triangleright M_2'}{(M_1\ M_2) \triangleright\triangleright (M_1\ M_2')} \qquad \frac{M_2 \text{ is a C-value} \quad M_1 \triangleright\triangleright M_1'}{(M_1\ M_2) \triangleright\triangleright (M_1'\ M_2)}$$

$$\frac{M_1\,[s] \triangleright\triangleright M_1'}{((\lambda M_0)\,[M_1 \cdot t])\,[s] \triangleright\triangleright (\lambda M_0)\,[M_1' \cdot(t \circ s)]}$$

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

$$\frac{M_1 \text{ is a C-value} \quad \cdots \quad M_i \text{ is a C-value} \quad M_{i+1}\,[s] \triangleright\triangleright M_{i+1}'}{(\lambda M_0)\,[M_1 \cdots M_i \cdot((M_{i+1} \cdot t) \circ s)] \triangleright\triangleright (\lambda M_0)\,[M_1 \cdots M_i \cdot M_{i+1}' \cdot(t \circ s)]}$$

Figure 3: Relation $\triangleright\triangleright$

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).

$$(M_I, s, ()) \Downarrow M_I \ \text{(Res)}$$

   By the previous lemma 22, we get the first three rules that define the relation $\triangleright\triangleright$.

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

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

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

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

$$\frac{\dfrac{I; C_2 \Downarrow^m M_2 \quad C_1 \Downarrow^m M_1}{I; C_2; C_1; \mathsf{Apply} \Downarrow^m (M_1\ M_2)}\ \text{(Apply)}}{((), s, I; C_2; C_1; \mathsf{Apply}) \Downarrow (M_1\ M_2)\,[s]}\ \text{(Code)}$$

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

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

Thus, by induction hypothesis, there exists M_2' such that the judgment $(M_I, s, C_2) \Downarrow M_2'$ holds. Furthermore, we have $M_2\,[s] \triangleright\triangleright M_2'$. Hence, we get:

$$\frac{(M_I, s, C_2) \Downarrow M_2' \quad C_1 \Downarrow^m M_1}{(M_I, s, C_2; C_1; \mathsf{Apply}) \Downarrow (M_1\,[s]\ M_2')}\ (\text{AppRight})$$

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

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

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

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

3. If $(S_I : S, s, I; C) \Downarrow M$ 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,

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

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

$$\frac{(M_I : S, s, C_2) \Downarrow M_2' \quad C_1 \Downarrow^m M_1}{(M_I : S, s, C_2; C_1; \mathsf{Apply}) \Downarrow (M_1\,[s]\ M_2')}\ (\text{AppRight})$$

Hence the result.

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

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

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

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

$$\frac{(M_I, s, C_1) \Downarrow M_1'}{(M_I : M_2, s, C_1) \Downarrow (M_1'\ M_2)}\ (\text{AppLeft})$$

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

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

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

$$\frac{(M_I : S, s, C_1) \Downarrow M_1'}{(M_I : S : M_2, s, C_1; \mathsf{Apply}) \Downarrow (M_1'\ M_2)}\ (\text{AppLeft})$$

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

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

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

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

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

$$\frac{(M_I : S, s, C_i) \Downarrow M_i' \quad C_{i+1} \Downarrow^m M_{i+1} \quad \cdots \quad C_n \Downarrow^m M_n \quad C_0 \Downarrow^{n+1} M_0}{\begin{array}{l}(M_I : S : M_{i-1} : \cdots : M_1, s, C_i; C_{i+1}; \cdots; C_n; \mathsf{Fun}(n, C_0)) \ \Downarrow \\ \qquad (\lambda M_0)\,[M_1 \cdots M_i' \cdot ((M_{i+1} \cdots M_n \cdot \uparrow^m) \circ s)]\end{array}}$$

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

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

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

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

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

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

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

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

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

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

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

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 $\overset{\text{C}}{\sim}$ by considering the three reduction axioms and the inference rules of $\sim$. That is, we state:

$$\frac{(\lambda M_0)\,[s] \text{ and } M \text{ are C-values}}{((\lambda M_0)\,[s]\ M) \overset{\mathrm{C}}{\sim} M_0\,[M \cdot s]}\ (\text{Beta}) \qquad \mathtt{n}\,[M_1 \cdots M_n \cdot s] \overset{\mathrm{C}}{\sim} M_n\ (\text{Var}^n)$$

$$\frac{M_1 \text{ is a C-value} \quad \cdots \quad M_n \text{ is a C-value}}{(\lambda M_0)\,[M_1 \cdots M_n \cdot (\uparrow^m \circ (P_1 \cdots P_m \cdot \mathrm{id})] \overset{\mathrm{C}}{\sim} (\lambda M_0)\,[M_1 \cdot \cdots \cdot M_n \cdot \mathrm{id}]}\ (\text{Shift}^m)$$

$$\frac{M_2 \overset{\mathrm{C}}{\sim} M_2'}{(M_1\ M_2) \overset{\mathrm{C}}{\sim} (M_1\ M_2')}\ (\text{AppLeft}) \qquad \frac{M_2 \text{ is a C-value} \quad M_1 \overset{\mathrm{C}}{\sim} M_1'}{(M_1\ M_2) \overset{\mathrm{C}}{\sim} (M_1'\ M_2)}\ (\text{AppRight})$$

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

Then, we define a step in the C-strategy as a $\triangleright\triangleright$ step followed by a $\overset{\mathrm{C}}{\sim}$ step. Thus, for any two $\lambda\sigma$-terms M and M', we have $M \xrightarrow{\mathrm{C}} M'$ if and only if there exists M'' such that $M \triangleright\triangleright M''$ and $M'' \overset{\mathrm{C}}{\sim} M'$.

Lemma 24 *Let D be a FAM state such that $\overline{D}$ 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 $\overline{D} = \overline{D'}$.*
2. *Otherwise, D reduces to D' by the execution of one instruction I and we have $\overline{D} \xrightarrow{\mathrm{C}} \overline{D'}$.*

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 $D = (AS, e, I; C) :: D_0$. By lemma 22, $\Phi(D)$ can be written as $\Phi(D) = (S_I : S, s, I; C) :: \Phi(D_0)$, where S_I is a stack such that $(S_I, s, I) \Downarrow M_I$. Therefore, by lemma 23, there exists M such that:

$$(M_I : S, s, C) :: \Phi(D_0) \Downarrow M \quad \text{and} \quad \overline{D} \triangleright\triangleright M$$

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 $D = ((C_0/e_0) : f : AS_0, e, \mathsf{Apply}; C) :: D_0$ and thus $S_I = (\lambda M_0)\,[\overline{e_0}] : \overline{f}$, with $m_0 = \text{length}(e_0)$ and $C_0 \Downarrow^{m_0+1} M_0$. On the $\lambda\sigma$-term side, we get $M_I = ((\lambda M_0)\,[\overline{e_0}]\ \overline{f})$.

   On the other hand, we have $D' = (f, e_0, C_0) :: (AS_0, e, C) :: D_0$. Thus, if $\overline{D'}$ exists, we have:

$$\frac{\dfrac{C_0 \Downarrow^{m_0+1} M_0}{((), \overline{f} \cdot \overline{e_0}, C_0) \Downarrow M_0\,[\overline{f} \cdot \overline{e_0}]}\ (\text{Code}) \qquad (M_0\,[\overline{f} \cdot \overline{e_0}] : S, s, C) :: \Phi(D_0) \Downarrow \overline{D'}}{\Phi(D') \Downarrow \overline{D'}}\ (\text{State})$$

   Now, let us state $M_I' = M_0\,[\overline{f} \cdot \overline{e_0}]$. Notice that we have $M_I \overset{\mathrm{C}}{\sim} M_I'$, by the axiom (Beta). Therefore, by lemma 21, there exists M' such that $(M_I' : S, s, C) :: \Phi(D_0) \Downarrow M'$, with $M \overset{\mathrm{C}}{\sim} M'$.

   In other words, $\overline{D'}$ exists and we have $M \overset{\mathrm{C}}{\sim} \overline{D'}$.

$$\frac{(\lambda M_0)\,[s] \text{ and } M \text{ are C-values}}{((\lambda M_0)\,[s]\ M) \xrightarrow{C} M_0\,[M \cdot s]}\ (\text{Beta})$$

$$\mathtt{n}\,[M_1 \cdots M_n \cdot s] \xrightarrow{C} M_n\ (\text{Var}^n) \qquad \uparrow^n \circ (M_1 \cdots M_n \cdot \text{id}) \xrightarrow{C} \text{id}\ (\text{Shift}^n)$$

$$\frac{M_2\,[s] \xrightarrow{C} M_2'}{(M_1\ M_2)\,[s] \xrightarrow{C} (M_1\,[s]\ M_2')}\ (\text{MapApp})$$

$$\frac{M_2 \xrightarrow{C} M_2'}{(M_1\ M_2) \xrightarrow{C} (M_1\ M_2')}\ (\text{AppRight}) \qquad \frac{M_1 \xrightarrow{C} M_1' \quad M_2 \text{ is a C-value}}{(M_1\ M_2) \xrightarrow{C} (M_1'\ M_2)}\ (\text{AppLeft})$$

$$\frac{s \circ t \xrightarrow{C} s'}{((\lambda M)\,[s])\,[t] \xrightarrow{C} (\lambda M)\,[s']}\ (\text{MapClos}) \qquad \frac{s \xrightarrow{C} s'}{(\lambda M_0)\,[s] \xrightarrow{C} (\lambda M_0)\,[s']}\ (\text{ClosRight})$$

$$\frac{M\,[s] \xrightarrow{C} M'}{(M \cdot t) \circ s \xrightarrow{C} M' \cdot (t \circ s)}\ (\text{MapEnv}) \quad \frac{M \xrightarrow{C} M'}{M \cdot s \xrightarrow{C} M' \cdot s}\ (\text{ConsLeft}) \quad \frac{M \text{ is a C-value} \quad s \xrightarrow{C} s'}{M \cdot s \xrightarrow{C} M \cdot s'}\ (\text{ConsRight})$$

Figure 4: The C-strategy

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

Finally, since $\overline{D} \triangleright\triangleright M$ and $M \overset{C}{\sim} \overline{D'}$, we get $\overline{D} \xrightarrow{C} \overline{D'}$, by definition of $\xrightarrow{C}$. □

Figure 4 exposes the C-strategy in the same small step formalism we used for other strategies. Axioms are somehow simplified and inference rules are classified per term construct in the algebra of $\lambda\sigma$-terms.

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. First, reduction is now possible inside the environment part s of a closure $(\lambda M_0)\,[s]$ (rule (ClosRight)). This reduction operates from the left to the right (rules (ConsLeft) and (ConsRight)). By the nature of the closure environments that the $\mathcal{C}$ compilation scheme produces, this reduction of environments ultimately amounts to replacing variables by their values (axiom (Var^n)) and then discarding this current substitution (axiom (Shift^n)). The C-strategy can cope with alternative and more sophisticated compilation schemes. In such schemes, complete sub-expressions would be abstracted out of function body. Then, the premise $M \xrightarrow{C} M'$ of rule (ConsLeft) could be any C-reduction. The second innovation lies in the propagation of substitutions inside terms (rules (MapApp)), the C-strategy combines several σ_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.

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

$$\begin{aligned}
\mathcal{Q}_k(\mathtt{n}) &= (n \leq k) \\
\mathcal{Q}_k(M_1\ M_2) &= \mathcal{Q}_k(M_1) \wedge \mathcal{Q}_k(M_2) \\
\mathcal{Q}_k(\lambda M) &= \mathcal{Q}_{k+1}(M) \\
\mathcal{Q}_k(M\,[s]) &= \mathcal{Q}_{l_s}(M) \wedge \mathcal{Q}_k(s), \text{ where } l_s = \mathrm{length}(s) \\
\\
\mathcal{Q}_k(M \cdot s) &= \mathcal{Q}_k(M) \wedge \mathcal{Q}_k(s) \\
\mathcal{Q}_k(s \circ t) &= \mathcal{Q}_{l_t}(s) \wedge \mathcal{Q}_k(t), \text{ where } l_t = \mathrm{length}(t) \\
\mathcal{Q}_k(s) &= \mathit{false} \quad \text{otherwise}
\end{aligned}$$

$$\begin{aligned}
\mathrm{length}(\uparrow^n) &= 0 \\
\mathrm{length}(M \cdot s) &= 1 + \mathrm{length}(s) \\
\mathrm{length}(s \circ t) &= \mathrm{length}(s), \text{ when } \mathcal{Q}_{l_t}(s) \text{ where } l_t = \mathrm{length}(t) \\
\mathrm{length}(s) &\text{ is undefined otherwise}
\end{aligned}$$

One easily checks that the new predicate $\mathcal{Q}_k$ generalizes $\mathcal{P}_k$. That is, the following implication holds:

$$\mathcal{P}_k(M) \quad \Rightarrow \quad \mathcal{Q}_k(M)$$

Thus, given any closed λ_{DB}-term N, let M be $\mathcal{C}(\emptyset, N)$. Then, the predicate $\mathcal{Q}_0(M\,[\mathrm{id}])$ 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 $M \xrightarrow{\mathrm{C}} M'$ 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 $s \xrightarrow{\mathrm{C}} s'$, we have $\mathrm{length}(s) = \mathrm{length}(s')$. □

Lemma 26 (Final state condition) *Le N be a closed λ_{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, $\overline{D}$ is a C-normal form.*

Proof: First observe that, by lemma 24, $\overline{D}$ exists and that we have $M\,[\mathrm{id}] \xrightarrow{\mathrm{C}}{}^* \overline{D}$. Let us state $D = (AS_n, e_n, C_n) :: \cdots :: (AS_1, e_1, C_1)$. 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 $\overline{D} = \overline{(f, (), ())} = \overline{f}$, which is a C-value and a C-normal form.

2. If the head instruction of C_n is $\mathsf{Fun}(n_0, C_0)$ or 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 $D = (AS, (\,), \mathsf{Local}; C)$, $D = ((), e, \mathsf{Local}; C) :: D_0$ or $D = (AS, \overrightarrow{f_{1,m}}, \mathsf{Global}(k); C) :: D'$ with $k > n$. Then, $\overline{D}$ is a C-failure term W, defined as follows:

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

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 $W' = \mathtt{n}\,[\overline{f_1} \cdots \overline{f_k} \cdot \mathrm{id}]$ 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 $\mathcal{Q}_0(M\,[\mathrm{id}])$, by lemmas 24 and 25. □

The transition **dump** 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) [7] 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.

$$\text{INSTRUCTION} \ ::= \ \mathsf{Fst} \mid \mathsf{Snd} \mid < \mid , \mid > \mid \mathsf{App} \mid \Lambda(\text{CODE})$$

CAM environments are structured as trees, they are written f (when they are closures) or e (when they are pairs or nil).

$$\text{ENVIRONMENT} \ ::= \ () \mid \text{CLOSURE} \mid (\text{ENVIRONMENT}, \text{ENVIRONMENT})$$

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

$$\begin{array}{rcl} \text{STACK} & ::= & () \mid \text{ENVIRONMENT} : \text{STACK} \\ \text{FRAME} & ::= & (\text{STACK} \bullet \text{CODE}) \\ \text{STATE} & ::= & \text{FRAME} \end{array}$$

The transitions of the CAM are as follows:

$$\begin{array}{rcl}
((e,f) : S \bullet \mathsf{Fst}; C) & \xrightarrow{\textbf{car}} & (e : S \bullet C) \\
((e,f) : S \bullet \mathsf{Snd}; C) & \xrightarrow{\textbf{cdr}} & (f : S \bullet C) \\
(e : S \bullet \Lambda(C); C') & \xrightarrow{\textbf{cur}} & ((C/e) : S \bullet C') \\
(e : S \bullet <; C) & \xrightarrow{\textbf{push}} & (e : e : S \bullet C) \\
(f : e : S \bullet \,,; C) & \xrightarrow{\textbf{swap}} & (e : f : S \bullet C) \\
(f : g : S \bullet >; C) & \xrightarrow{\textbf{cons}} & ((g,f) : S \bullet C) \\
(((C/e), f) : S \bullet \mathsf{App}; C') & \xrightarrow{\textbf{app}} & ((e,f) : S \bullet C; C')
\end{array}$$

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 $,$, 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 λ_{DB}-terms in CAM code follows the usual translation from λ-terms to terms of categorical cartesian logic (CCL).

$$\begin{array}{rcl}
[\![\, \mathtt{1} \,]\!] & = & \mathsf{Snd} \\
[\![\, \mathtt{n+1} \,]\!] & = & \mathsf{Fst}[\![\, \mathtt{n} \,]\!] \\
[\![\, (N_1\ N_2) \,]\!] & = & <[\![\, N_1 \,]\!] \,,\, [\![\, N_2 \,]\!] > \mathsf{App} \\
[\![\, \lambda N \,]\!] & = & \Lambda([\![\, N \,]\!])
\end{array}$$

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

Finally, a code $C = [\![\, N \,]\!]$ is loaded into the CAM as $\mathcal{L}(N) = (() \bullet C)$.

6.2 The decompilation

First, we inverse the compilation procedure $[\![\ ]\!]$. We do so by proving judgments $C \Downarrow N$, which read "the code segment C stands for the λ_{DB}-term N".

$$\mathsf{Fst}^n; \mathsf{Snd} \Downarrow \mathtt{n+1} \qquad \frac{C_1 \Downarrow N_1 \qquad C_2 \Downarrow N_2}{<C_1 \,,\, C_2> \mathsf{App} \Downarrow (N_1\ N_2)} \qquad \frac{C \Downarrow N}{\Lambda(C) \Downarrow \lambda N}$$

For any λ_{DB}-term N and categorical code C, the equivalence $[\![\, N \,]\!] = C \Leftrightarrow C \Downarrow N$ is easily shown.

The decompilation procedure extends naturally to environments and closures,

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

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

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

$$(f \bullet ()) \Downarrow \overline{f} \ \text{(Res)} \qquad \frac{C \Downarrow N}{(e \bullet C) \Downarrow N\,[\overline{e}]}\ \text{(Code)}$$

$$\frac{(S \bullet C) \Downarrow M \quad C' \Downarrow N}{(S : e \bullet C \,;\, C' > \mathsf{App}) \Downarrow (M\ N\ [\overline{e}])}\ \text{(AppLeft)} \qquad \frac{(S \bullet C) \Downarrow M}{(S : f \bullet C > \mathsf{App}) \Downarrow (\overline{f}\ M)}\ \text{(AppRight)}$$

$$((f_1, f_2) \bullet \mathsf{App}) \Downarrow (\overline{f_1}\ \overline{f_2})\ \text{(AppCons)}$$

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 The strategy

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

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 $(S' \bullet C')$ from a state $(S \bullet C)$ by removing some elements from the bottom of the stack S and some instructions from the end of code C. In other words, $(S \bullet C)$ is decomposed as $(S' : S'' \bullet C'; C'')$, where $(S' : C')$ 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 $\overline{D}$ exists, then D can be written $(S_I : S \bullet C_I; C)$ and there exists a $\lambda\sigma$-term M_I such that $(S_I \bullet C_I) \Downarrow M_I$.*

Additionally, D' can be written $(S'_I : S \bullet C'_I; C)$ and there exists a $\lambda\sigma$-term M'_I such that $(S'_I \bullet C'_I) \Downarrow M'_I$.

Proof: A very constructive one: first assume that I is Snd, Fst, $<$ or App. We get:

I	Snd	Fst	$<$	App
C_I	Snd	$\mathsf{Fst}^n;\mathsf{Snd}$	$<[\![N_1]\!] \, , [\![N_2]\!] > \mathsf{App}$	App
S_I	(e,f)	(e,f)	e	$((\![N_1]\!]/e_1), f_2)$
M_I	$\mathtt{1}\,[\overline{f}\cdot\overline{e}]$	$\mathtt{n{+}1}\,[\overline{f}\cdot\overline{e}]$	$(N_1\ N_2)\,[\overline{e}]$	$((\lambda N_1)\,[\overline{e_1}]\ \overline{f_2})$
C'_I	$()$	$\mathsf{Fst}^{n-1};\mathsf{Snd}$	$[\![N_1]\!] \, , [\![N_2]\!] > \mathsf{App}$	$[\![N_1]\!]$
S'_I	f	e	$e : e$	(e_1, f_2)
M'_I	$\overline{f}$	$\mathtt{n}\,[\overline{e}]$	$(N_1\,[\overline{e}]\ N_2\,[\overline{e}])$	$N_1\,[\overline{f_2}\cdot\overline{e_1}]$

Then assume I is one of the remaining three instructions Λ, , or $>$:

I	$\Lambda([\![N_0]\!])$	,	$>$
C_I	$\Lambda([\![N_0]\!])$	$, [\![N_2]\!] > \mathsf{App}$	$> \mathsf{App}$
S_I	e	$f : e$	$f_2 : f_1$
M_I	$(\lambda N_0)\,[\overline{e}]$	$(\overline{f}\ N_2\,[\overline{e}])$	$(\overline{f_1}\ \overline{f_2})$
C'_I	$()$	$[\![N_2]\!] > \mathsf{App}$	App
S'_I	$([\![N_0]\!]/e)$	$e : f$	(f_1, f_2)
M'_I	$(\lambda N_0)\,[\overline{e}]$	$(\overline{f}\ N_2\,[\overline{e}])$	$(\overline{f_1}\ \overline{f_2})$

□

The axioms of the X-strategy derive from the lemma above. These axioms are the four rules $M_I \xrightarrow{\mathrm{X}} M'_I$, 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 $D = (S_I : S \bullet C_I; C)$ be a CAM state such that both judgments $D \Downarrow M$ and $(S_I \bullet C_I) \Downarrow M_I$ hold. Let $(S'_I \bullet C'_I)$ be any CAM state such that the judgment $(S'_I \bullet C'_I) \Downarrow M'_I$ holds. Let D' be the state $(S'_I : S \bullet C'_I; C)$. 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 $M = M_I$. □

First notice that, when the instruction I is Λ, , 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:

$$\mathtt{1}\,[M \cdot s] \xrightarrow{X} M \text{ (FVar)} \qquad \mathtt{n{+}1}\,[M \cdot s] \xrightarrow{X} \mathtt{n}\,[s] \text{ (RVar)}$$

$$(N_1\ N_2)\,[s] \xrightarrow{X} (N_1\,[s]\ N_2\,[s]) \text{ (App)} \qquad \frac{(\lambda N)\,[s] \text{ is a X-value} \quad M \text{ is a X-value}}{((\lambda N)\,[s]\ M) \xrightarrow{X} N\,[M \cdot s]} \text{ (Beta)}$$

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

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

Lemma 29 (Final state condition) *Let N be a closed λ_{DB}-term and let D be a terminal state computed by the CAM starting from $\mathcal{L}(N)$. Then, $\overline{D}$ is a X-normal form.*

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

1. The code C is empty. Then, $\overline{D}$ must be computed by the decompilation rule (Res). Thus, S holds a single element which *must* be a closure f. Hence $\overline{D}$ is the X-value $\overline{f}$. This is the normal case.

2. A variable access fails. That is, $D = ((\,) \bullet \mathsf{Fst}; C_0)$ or $D = ((\,) \bullet \mathsf{Snd}; C_0)$ Then, $\overline{D}$ is a X-failure term W:

$$\begin{array}{rcll} W & ::= & \mathtt{n}\,[\mathrm{id}] & \text{with } n \geq 1 \\ & | & W\ M & \\ & | & M\ W & \text{where } M \text{ is a X-value} \end{array}$$

   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 $N\,[\mathrm{id}]$ 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 $\overline{D}$ exists. The proof tree of $(S \bullet C) \Downarrow M$ must include the proof of a judgment $(S_I \bullet C_I) \Downarrow M_I$ 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 "$,$", we have $C_I = \ , C'_I > \mathsf{App}$ and

$$\frac{(f \bullet (\,)) \Downarrow \overline{f} \text{ (Res)} \quad C'_I \Downarrow N'_I}{(f : e \bullet \ , C'_I > \mathsf{App}) \Downarrow (\overline{f}\ N'_I\,[\overline{e}])} \text{ (AppLeft)}$$

   Thus, we get $S_I = f : e$. Hence, since $S = S_I : S'$, the transition **swap** 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 [18], we also made explicit the strategy of the ZAM. Briefly, the ZAM implements the L-strategy, that is, right-to-left call-by-value.

The G-machine and the TIM [13] can also be understood in the $\lambda\sigma$-calculus, although these machines look quite different from environment machines. In order to simplify the management of variables at run-time, compilers for these machines translate input λ-terms into supercombinators, by the so-called λ-lifting operation [23]. Supercombinators are n-ary functions without free variables, that is, in terms of $\lambda\sigma_w$, closures $(\lambda\lambda\ldots\lambda M)\,[\mathrm{id}]$, where M is a $\lambda\sigma$-term whose variables are all λ-bound. We call such closures $\lambda\sigma$-supercombinators. In the $\lambda\sigma$-framework, λ-lifting is actually quite similar to the first phase $\mathcal{C}$ of the FAM compilation. Where, from a function, the transformation $\mathcal{C}$ produces a closure, the λ-lifting will produce the partial application of a $\lambda\sigma$-supercombinator. For instance, the abstraction, $\lambda(\mathtt{1}\ (\mathtt{5}\ \mathtt{7}))$ is translated by the scheme $\mathcal{C}$ into the $\lambda\sigma$-closure $(\lambda(\mathtt{1}\ (\mathtt{2}\ \mathtt{3})))\,[\mathtt{4}\cdot\mathtt{6}\cdot\mathrm{id}]$, whereas it lambdalifts to the $\lambda\sigma$-term $((\lambda\lambda\lambda(\mathtt{1}\ (\mathtt{2}\ \mathtt{3})))\,[\mathrm{id}])\ \mathtt{6}\ \mathtt{4}$.

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

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

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

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

8 Related works and conclusion

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

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 [9], the Krivine machine and the CAM, two shared

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

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

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

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

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

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy, "*Explicit Substitutions*", Journal of Functional Programming, 6(2):299-327 March 1996.

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

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

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

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

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

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

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

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

[10] P.-L. Curien, T. Hardin and J.-J. Lévy, "*Confluence Properties of Weak and Strong Calculi of Explict Substitutions*", JACM, Vol. 43, No 2, pp 362–397, 1996.

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

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

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

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

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

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

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

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

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

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

[21] B. Pagano, "*Bi-simulation de machines abstraites en lambda-sigma-calcul*", Technique et science informatique, volume 15, numéro 7/1996, éditions Hermès (In french).

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

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

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

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS
Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399