Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 237 (2009) 57–73
www.elsevier.com/locate/entcs

New Developments in Environment Machines

Maribel Fernández[1] and Nikolaos Siafakas[2]

Department of Computer Science, King's College London, Strand, London WC2R 2LS, U.K.

Abstract

In this paper we discuss and compare abstract machines for the lambda-calculus, implementing various evaluation strategies. Starting from the well-known Categorical abstract machine (CAM) and Krivine's abstract machine (KAM), we develop two families of machines that differ in the way they treat environments. The first family is inspired by the work on closed reduction strategies, whereas the second is built in the spirit of the jumping machines based on the work done on Linear Logic

Keywords: λ-calculus, environment machines, explicit substitutions, Linear Logic.

1 Introduction

The λ-calculus is regarded as the theoretical foundation of functional programming languages. It can be thought of as a simple, statically scoped programming language. Abstract machines are one of the tools one can use to provide a formal operational semantics to a programming language. Abstract machines are transition systems that bridge the gap between the specification of a dynamic semantics of a language and a concrete implementation. They may be considered as rewriting systems, where the rewriting rules have no superpositions.

There is no measure of the abstractness of an abstract machine, however, we will adopt some terminology: following [2] we refer to *abstract machines* as transition systems that accept abstract syntax-trees (in our case, λ-terms) as source-syntax, while machines that work with an instruction set will be called *virtual machines*, or *concrete machines* if there exists a hardware implementation that offers the particular instruction set.

We are interested in a particular kind of abstract machine, called *environment machine*. The components that are common to all environment machines are: the expression being evaluated, a control stack and the environment which provides

[1] Email: maribel.fernandez@kcl.ac.uk
[2] Email: nikolaos.siafakas@kcl.ac.uk

doi:10.1016/j.entcs.2009.03.035

the bindings for the free variables in the expression. Functions are represented via *closures*, that is, a piece of syntax coupled with an environment containing the mappings that provide values for the free variables. The underlying λ-calculi that help to specify and understand environment machines are calculi of explicit substitutions [1,30,27], which internalise syntactically and semantically the otherwise external substitution operation.

Many of the environment machines that we encounter in the literature are resource unconscious: the memory model is often unspecified, yet referential transparency is guaranteed; some external machinery is always assumed (a garbage collector, a cloning device) or is given if the memory-layout is more concrete. This level of abstraction, although convenient when analyzing properties of the machines, is not sufficiently detailed from an implementation point of view. Linear Logic [21] addresses resource management issues in proofs, and the same techniques can be applied to the λ-calculus. Several abstract machines for the λ-calculus have been developed following this approach (see for example [3,19,11,29,13]), and also, at a more abstract level, several reduction strategies have been defined to take into account the management of resources in the β-reduction process (or more specifically, in the propagation of substitutions), see for instance work on the λ_{xgc}-calculus in [30], or work motivated by Linear Logic in [14,15,20,19].

In this paper, starting from (variants of) two well-known abstract machines — Krivine's call-by-name machine [24], which we will call KAM, and the Categorical abstract machine [8], called CAM — we develop two families of machines that exploit recent work on evaluation strategies in the λ-calculus. Our first contribution is a family of machines, which we call λ_c-machines, based on the work on closed reduction [14,15,31]. It includes call-by-value and call-by-name machines that can be seen as a refinement of the CAM and KAM, respectively. More interestingly, this family includes machines that perform reductions under abstractions, following the closed-reduction strategy discussed in [15,31]. The second contribution is a family of machines that operate on global environments and are designed to facilitate compilation: the motivation is to have memory management included in the definition. The notion that we want to make significant here is that of a "virtual copy". The machines operate without the need for a garbage collector and each step may be accomplished in amortised $O(1)$ time.

All the machines described in this paper have been implemented; the prototypes are available from `http://www.dcs.kcl.ac.uk/pg/siafakas/`.

2 Background

The first *environment machine*, the SECD-machine [25], dates back to the 1960s. It is a call-by-value machine, where arguments are evaluated from right-to-left. The Categorical Abstract Machine [8], also a call-by-value machine, is equipped with an instruction set and may be regarded as a virtual machine.

Krivine's call-by-name environment machine [24] is simple and influential. A lazy (call-by-need) version of the KAM can be found in [16]; the ZINC abstract

machine [26] may be thought of as a call-by-value version of Krivine's abstract machine.

There are indeed many machines available in the literature that can be seen as variants of the KAM or the CAM (see the discussion in [2]). Most of these machines aim at reducing programs (i.e., closed λ-terms) to weak head normal form; they do not perform reductions under abstractions. A *strong reduction* machine was studied in [23]. In terms of programming languages, one can think of reduction under an abstraction as a "specialisation" of a function definition. The standard approach that is taken to achieve such a specialisation is to define a weak abstract machine which then calls new instances of itself inside the bodies of abstractions, using the normalisation by evaluation technique (see e.g. [7,15]).

Thus, environment machines can be classified according to the strategy of evaluation they implement (e.g., call-by-name, call-by-value), according to the kind of reduction performed (e.g., weak reduction, strong reduction), and also according to the way they associate environments to terms. For instance, in the CAM and the KAM, environments are associated to sub-terms of the term being evaluated, and provide information about the terms that should be substituted for the free variables in the sub-term. Later we will describe machines that will use only one environment instead of associating environments to sub-terms. To distinguish these two classes of machines, we say that the first class of machines use *local environments* whereas the latter use *global environments*.

3 Abstract machines with local environments

We start by presenting two weak machines with local environments that use names for variables, instead of the standard presentation using De Bruijn indices [12]. The machines are variants of the KAM and the CAM with Lisp-like associative lists to represent environments; usually the KAM machine is presented using De Bruijn indices [9], however names and maps have also been used (see for instance [16,3]). Since we use names, we may have duplicate entries for keys in the list, however, keys that arise earlier in the list shadow keys that arise later in the list. In this way, one can implement maps in an non-destructive manner.

After presenting these two machines, we show how the management of resources (i.e., the environment) can be improved, using techniques inspired by the work on closed reduction [14,15].

3.1 A call-by-name environment machine with names

We specify the machine as a transition system, with a set of transition rules on configurations. A *configuration* consists of a λ-term and an environment (i.e., a closure), and a stack of closures. Environments are presented as lists of substitutions. We use the infix, right associative (:) operator and the empty list is denoted by []. Substitutions and closures are represented as pairs, with a tag c (for closure) or s (for substitution). The machine is loaded with a term, an empty environment (when no other domain is provided in advance), and an empty stack; a successful

Term	Env	Stack		Term	Env	Stack	Cond	Rule
tu	e	s	$\rightarrow$	t	e	$(u,e)^c : s$		Push
$\lambda x.t$	e	$(u,e')^c : s$	$\rightarrow$	t	$((u,e')^c,x)^s : e$	s		Pass
y	$((u,e')^c,x)^s : e$	s	$\rightarrow$	y	e	s	$x \not\equiv y$	EntF
y	$((u,e')^c,x)^s : e$	s	$\rightarrow$	u	e'	s	$x \equiv y$	EntT
$\lambda x.t$	e	$[]$	$\not\rightarrow$					Halt

Table 1
Call-by-name environment machine

run yields nothing but a closure. The transition rules for the machine are given in Table 1. To prove the correctness of the machine we define, following [9], a calculus of closures with a call-by-name strategy presented as a big-step operational semantics [17,18]. Terms are written $t[s]$ where t is a pure λ-term and s is a list of substitutions of the form (u,x) where x is a variable and u a term in the calculus, that is, each term $t[s]$ is a closure.

We can obtain the pure λ-term represented by a closure $t[s]$ by using s as a substitution in t; we denote the result as $Subst(t,s)$. If $Subst(t,s)$ is a closed λ-term, we say that the closure $t[s]$ is closed. The *values* of closed closures are abstractions $(\lambda x.t)[s]$. The relation $t[s] \rightarrow_{CBN} v$ defines the value v of a closed closure $t[s]$ in a call-by-name strategy.

$$\frac{v \ value}{v \rightarrow_{CBN} v} \qquad \frac{t \rightarrow_{CBN} v}{x[(t,x):s] \rightarrow_{CBN} v} \qquad \frac{x[s] \rightarrow_{CBN} v}{x[(t,y):s] \rightarrow_{CBN} v}$$

$$\frac{t[s] \rightarrow_{CBN} (\lambda x.t')[s'] \quad t'[(u[s],x):s'] \rightarrow_{CBN} v}{(tu)[s] \rightarrow_{CBN} v}$$

Definition 3.1 (Compilation and De-compilation) *The de-compilation (or read-back) of a machine configuration $(t,e,[s_1 \ldots s_n])$ $(n \geq 0)$ is a term obtained by*

$$decomp\,(t,e,[(u_1,e'_1)^c : \ldots : (u_n,e'_n)^c]) = t[e]\ \ u_1[e'_1]\ \ldots\ u_n[e'_n]$$

The function comp compiles terms from the calculus of closures into machine states, using two auxiliary functions (compc compiles closures and comps compiles substi-

tutions; they are mutually recursive): comp $t[s] = (t, (comps\ s), [])$ *where*

$$\begin{aligned} comps\ [] &= [] \\ comps\ [(u,x):s] &= [(compc\ u, x)^s : (comps\ s)] \\ compc\ u[s] &= (u, comps\ s)^c \end{aligned}$$

We say that a configuration in the abstract machine is closed *when* $decomp(t, e, s)$ *is a closed term.*

The following standard properties are used in the proof of correctness of the machine with respect to the call-by-name strategy:

Proposition 3.2 (i) *An irreducible, closed configuration in the call-by-name machine has the form* $(\lambda x.t, e, [])$.

(ii) *If* $(t, e, s) \to (t', e', s')$ *then* $(t, e, s \circ s'') \to (t', e', s' \circ s'')$, *where* $\circ$ *denotes list concatenation. Therefore, if* $(t, e, []) \to^* (t', e', [])$ *also* $(t, e, s) \to^* (t', e', s)$ *for any* s.

Theorem 3.3 (Correctness of the Call-by-name Environment Machine) *Let* t *be a closed* λ*-term.*

(i) *If* $t[] \to_{CBN} v$ *using the call-by-name strategy then comp* $t[] = (t, [], []) \to^* (u, e, [])$ *final, and* $(u, e, []) = comp\ v$.

(ii) *If comp* $t[] = (t, [], []) \to^* (u, e, s)$, *where* (u, e, s) *is irreducible, then* $t \to_{CBN} decomp(u, e, s)$.

3.2 A call-by-value environment machine with names

We now describe a machine isomorphic to the eager machine given in [13] and closely related to the Categorical Abstract Machine. The configurations of the machine consist of a λ-term, the environment (a list of substitutions mapping variables to closures; closures are terms with environments), and a stack that contains closures tagged either with Q or P. As before, closures and substitutions are represented as pairs. We attach a tag s to a pair of a variable and a closure to indicate that we are using it as a substitution; we attach a tag c to the closure part of the substitution. In the stack, closures are tagged with Q or P. We use the tag Q for arguments, and mark with a P functions stored in the stack while their arguments are evaluated. The transition rules for the abstract machine are given in Table 2.

It is easy to see that this machine is exactly the call-by-value machine defined in [13], but using names of variables instead of De Bruijn indices. The proof of correctness can be easily adapted from the one in [13].

3.3 Improving the management of resources: λ_c-machines

In the machines given above, environments carry a lot of useless information during computation due to the uncontrolled distribution of substitutions. A good repre-

Term	Env	Stack		Term	Env	Stack	Cond
tu	e	s	$\to$	t	e	$(u,e)^Q : s$	
$\lambda x.t$	e	$(u,e')^Q : s$	$\to$	u	e'	$(\lambda x.t, e)^P : s$	
$\lambda x.t$	e	$(\lambda x'.t', e')^P : s$	$\to$	t'	$((\lambda x.t, e)^c, x') : e'$	s	
y	$((u,e')^c, x)^s : e$	s	$\to$	y	e	s	$x \not\equiv y$
y	$((u,e')^c, x)^s : e$	s	$\to$	u	e'	s	$x \equiv y$
$\lambda x.t$	e	$[]$	$\not\to$				

Table 2
Call-by-value environment machine

sentation of environments is essential in order to get efficient implementations. It is well known that most of the closures that these machines build are useless and various approaches have been taken to record in the environments just the relevant bindings (e.g. environment splitting in [6]). Explicit substitution calculi provide a theoretical framework to explain how to deal with the environments. Linear Logic [21] adds explicit control over the assumptions in proofs, and as mentioned in the introduction explicit substitution calculi exploit these ideas to improve the management of substitutions. Thus, an explicit substitution calculus that controls the distribution of substitutions is a good candidate to specify environment machines in a resource conscious way. We will use a calculus of explicit substitutions that we call λ_c, inspired by [14,28].

Definition 3.4 (Terms in the λ_c-calculus) *We use x, y, z to denote variables, t, u, v to denote terms, and $\overline{o}$ to denote a sequence of elements $o_1, \ldots, o_n$. Table 3 shows the term constructions, together with the variable constraints that must be satisfied for each construction, and the associated free variables.*

The variable constraints imply that each variable occurs free in a term exactly once. Note that in closures, i.e., λ_c-terms of the form $t[s]$, s contains substitutions for variables that must occur free in t; it may contain several pairs (u, x), or none, in the latter case we write $t[id]$. In all the pairs (u, x) occurring in s the term u is closed. Also, unlike in the previous system, t may also contain closures.

A pure λ-term will be compiled into a λ_c-term by the function defined below.

Definition 3.5 (Compilation) *Let t be a λ-term. Its compilation $[\![t]\!]$ into λ_c is defined as: $[x_1] \ldots [x_n]\langle t \rangle$ where $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$, $n \geq 0$, we assume w.l.o.g. that the variables are processed in lexicographic order, and $\langle \cdot \rangle$ is defined by: $\langle x \rangle = x$, $\langle tu \rangle = \langle t \rangle \langle u \rangle$, and $\langle \lambda x.t \rangle = \lambda x.[x]\langle t \rangle$ if $x \in \mathsf{fv}(t)$, otherwise $\langle \lambda x.t \rangle = \lambda x.\epsilon_x.\langle t \rangle$. We*

Name	Term	Variable Constraint	Free Variables
Variable	x	$-$	$\{x\}$
Abstraction	$\lambda x.t$	$x \in \mathsf{fv}(t)$	$\mathsf{fv}(t) - \{x\}$
Application	tu	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \emptyset$	$\mathsf{fv}(t) \cup \mathsf{fv}(u)$
Erase	$\epsilon_x.t$	$x \notin \mathsf{fv}(t)$	$\mathsf{fv}(t) \cup \{x\}$
Copy	$\delta_x^{y,z}.t$	$x \notin \mathsf{fv}(t), y \neq z, \{y,z\} \subseteq \mathsf{fv}(t)$	$(\mathsf{fv}(t) - \{y,z\}) \cup \{x\}$
Closure	$t[\overline{(u,x)}]$	$\overline{x} \in \mathsf{fv}(t),\ \forall i \neq j, x_i \neq x_j, \mathsf{fv}(u) = \emptyset$	$\mathsf{fv}(t) - \overline{x}$

Table 3
λ_c-terms and variable constraints

define $[\cdot]\cdot$ *below, using* $t[x := u]$ *to denote the usual (implicit) notion of substitution.*

$$\begin{aligned} [x]x &= x & [x](\lambda y.t) &= \lambda y.[x]t \\ [x](\epsilon_y.t) &= \epsilon_y.[x]t & [x](\delta_y^{y',y''}.t) &= \delta_y^{y',y''}.[x]t \end{aligned}$$

$$\begin{aligned} [x](tu) &= \delta_x^{x',x''}.[x'](t[x := x'])[x''](u[x := x'']) && x \in \mathsf{fv}(t), x \in \mathsf{fv}(u), x', x''\ fresh \\ &= ([x]t)u && x \in \mathsf{fv}(t), x \notin \mathsf{fv}(u) \\ &= t([x]u) && x \in \mathsf{fv}(u), x \notin \mathsf{fv}(t) \end{aligned}$$

For example:

$$[\![(xy)(xy)]\!] = [x][y]\langle\!\langle (xy)(xy) \rangle\!\rangle = \delta_y^{y',y''}.\delta_x^{x',x''}.(x'y')(x''y'') \neq [y][x]\langle\!\langle (xy)(xy) \rangle\!\rangle$$

The compilation function returns a λ_c-term without closures. However, to load the abstract machine below, we assume there is a further step in the compilation which adds an empty closure $[id]$ to each sub-term. Although the compilation is defined on open terms, we demand that sufficient substitutions are provided in advance to obtain a closed term.

We will now investigate how call-by-name and call-by-value versions of λ_c-calculi may improve the KAM and CAM respectively. We start with a traditional operational semantics and derive the corresponding abstract machines. Next, we exploit the copying and erasing constructs to mix features from call-by-name and call-by-value. Finally, we discuss a more powerful strategy inspired by [14].

To define the strategies, we use an external function $\bullet$ to extend environments that we define via concatenation: $(t[s'])\bullet[s] = t[s' \circ s]$. Since substitutions are closed, and each variable occurs at most once in s, the concatenation is commutative. One of the main contributions of the paper is the definition abstract machines based on λ_c-terms which we present next.

$$\frac{t \to^c_{CBN} v}{x[(t,x)] \to^c_{CBN} v} \qquad \frac{}{(\lambda y.t)[id] \to^c_{CBN} (\lambda y.t)[id]}\, Ax \qquad \frac{(\lambda y.t \bullet [(u,x)])[s] \to^c_{CBN} v}{(\lambda y.t)[(u,x):s] \to^c_{CBN} v}$$

$$\frac{x \in fv(t) \quad ((t \bullet [(u',x)])u)[s] \to^c_{CBN} v}{(tu)[(u',x):s] \to^c_{CBN} v} \qquad \frac{x \in fv(u) \quad (t(u \bullet [(u',x)]))[s] \to^c_{CBN} v}{(tu)[(u',x):s] \to^c_{CBN} v}$$

$$\frac{t \to^c_{CBN} (\lambda x.r)[id] \quad r \bullet [(u,x)] \to^c_{CBN} v \quad fv(u) = \emptyset}{(tu)[id] \to^c_{CBN} v}\, Beta$$

$$\frac{(\delta^{y'y''}_y.t \bullet [(u,x)])[s] \to^c_{CBN} v}{(\delta^{y'y''}_y.t)[(u,x):s] \to^c_{CBN} v} \qquad \frac{(t \bullet [(u,x'),(u,x'')]) \bullet [s] \to^c_{CBN} v}{(\delta^{x'x''}_x.t)[(u,x):s] \to^c_{CBN} v}\, Delta$$

$$\frac{(\epsilon_y.t \bullet [(u,x)])[s] \to^c_{CBN} v}{(\epsilon_y.t)[(u,x):s] \to^c_{CBN} v} \qquad \frac{t \bullet [s] \to^c_{CBN} v}{(\epsilon_x.t)[(u,x):s] \to^c_{CBN} v}$$

Fig. 1. Call-by-name evaluation of λ_c-terms

Call-by-name λ_c-machine

We define first a call-by-name evaluation strategy for λ_c-terms in Figure 1. The operational semantics is given by a set of axioms and rules, defining a relation $t[s] \to^c_{CBN} v$ between closed closures and values. Values are λ_c-terms of the form $(\lambda y.t)[id]$, that is, the substitutions will be pushed inside abstractions to compute values. Usually, this operation requires α-conversion to avoid capture of variables, but we rely on a closed reduction strategy, which is α-conversion free [14].

The corresponding abstract machine is defined by the transition rules in Table 4 on configurations containing a λ_c-term and a stack of λ_c-terms. The correctness of the machine with respect to the operational semantics is easy to prove, the proof follows the same lines as the correctness proofs mentioned earlier in this section. We state the main results below:

Proposition 3.6 (i) *An irreducible, closed configuration in the call-by-name λ_c machine has the form $((\lambda x.t)[id], [])$.*

(ii) *If $(t,s) \to (t',s')$ then $(t, s \circ s'') \to (t', s' \circ s'')$, where $\circ$ denotes list concatenation. Therefore, if $(t,[]) \to^* (t',[])$ also $(t,s) \to^* (t',s)$ for any s.*

Theorem 3.7 (Correctness of the Call-by-name λ_c Environment Machine) *Let t be a closed λ_c-term obtained by compiling a closed λ-term u.*

(i) *If $t[] \to^c_{CBN} v$ then $(t,[]) \to^* (v,[])$ final.*

(ii) *If $(t,[]) \to^* (v,[])$ final, then $t \to^c_{CBN} v$.*

In order to define a call-by-value strategy, $\to^c_{CBV}$, for λ_c, we just need to replace

Term	Stack		Term	Stack	Cond
$(tu)[(u',x):e]$	s	$\to^c_{CBN}$	$((t \bullet [(u',x)])u)[e]$	s	$x \in fv(t)$
$(tu)[(u',x):e]$	s	$\to^c_{CBN}$	$(t(u \bullet [(u',x)]))[e]$	s	$x \in fv(u)$
$x[(u,x)]$	s	$\to^c_{CBN}$	u	s	
$(\lambda x.t)[(u,y):e]$	s	$\to^c_{CBN}$	$(\lambda x.t \bullet [(u,y)])[e]$	s	$x \not\equiv y$
$(\epsilon_y.t)[(u,x):e]$	s	$\to^c_{CBN}$	$(\epsilon_y.t \bullet [(u,x)])[e]$	s	$x \not\equiv y$
$(\epsilon_x.t)[(u,x):e]$	s	$\to^c_{CBN}$	$t \bullet [e]$	s	
$(\delta_y^{y'y''}.t)[(u,x):e]$	s	$\to^c_{CBN}$	$(\delta_y^{y'y''}.t \bullet [(u,x)])[e]$	s	$x \not\equiv y$
$(\delta_x^{x'x''}.t)[(u,x):e]$	s	$\to^c_{CBN}$	$(t \bullet [(u,x'),(u,x'')]) \bullet [e]$	s	
$(tu)[]$	s	$\to^c_{CBN}$	t	$u:s$	
$(\lambda x.t)[]$	$u:s$	$\to^c_{CBN}$	$t \bullet [(u,x)]$	s	
$(\lambda x.t)[]$	$[]$	$\not\to^c_{CBN}$			

Table 4
Call-by-name λ_c abstract machine

the rule *Beta* in the definition of $\to^c_{CBN}$ by the following rule:

$$\frac{t \to^c_{CBV} (\lambda x.r)[id] \quad u \to^c_{CBV} v' \quad r \bullet [(v',x)] \to^c_{CBV} v \quad fv(u) = \emptyset}{(tu)[id] \to^c_{CBV} v}\ Beta$$

We omit the corresponding call-by-value environment machine.

The right time for evaluating the argument: the λ_{CBVN}-machine

The relations $\to^c_{CBV}$ and $\to^c_{CBN}$ differ in the way the argument is evaluated. We can take profit of the explicit copy constructs in the λ_c syntax, and trigger the evaluation of the argument just before copying. This gives us a reduction strategy that is in-between call-by-name and call-by-value (but it does not correspond exactly to a call-by-need strategy, since the existence of a copy construct does not imply neededness; see [14,15] for a more detailed discussion).

To define a strategy that evaluates substitutions before copying, we use the rules in Table 5 instead of the corresponding rules of $\to^c_{CBN}$ (see Figure 1). We also put some laziness in the way substitutions are propagated (we will avoid pushing substitutions through abstractions). In the corresponding abstract machine, we use tagged terms, in the same way as in the categorical abstract machine. We do reduce the substitution just before copying: in the operational semantics, it is easy to see that a substitution can evaluate only to an abstraction. We use Q to tag arguments as before, however, P now tags copying constructs instead of functions. The transition rules for the abstract machine are the rules in rows 1-7 in Table 4 together with the rules in Table 6.

$$\frac{}{(\lambda y.t)[s] \to^{c}_{CBVN} (\lambda y.t)[s]}\ Ax' \qquad \frac{t \to^{c}_{CBVN} (\lambda x.r)[s] \quad r \bullet [(u,x) \circ s] \to^{c}_{CBVN} v}{(tu)[id] \to^{c}_{CBVN} v}\ Beta'$$

$$\frac{u \to^{c}_{CBVN} u' \quad t \bullet ([(u',x'),(u',x'')] \circ s) \to^{c}_{CBVN} v}{(\delta^{x'x''}_{x}.t)[(u,x) : s] \to^{c}_{CBVN} v}\ Delta'$$

Table 5
$\to^{c}_{CBVN}$-evaluation of λ_c-terms

Term	Stack		Term	Stack
$(tu)[id]$	s	$\to^{c}_{CBVN}$	t	$(u)^{Q} : s$
$(\delta^{x'x''}_{x}.t)[(u,x) : e]$	s	$\to^{c}_{CBVN}$	u	$((\delta^{x'x''}_{x}.t)[e])^{P} : s$
$(\lambda x.t)[e]$	$(u)^{Q} : s$	$\to^{c}_{CBVN}$	$t \bullet ([(u,x)] \circ e)$	s
$\underbrace{(\lambda\_.\_)[\_]}_{B}$	$((\delta^{x'x''}_{x}.t)[e])^{P} : s$	$\to^{c}_{CBVN}$	$t \bullet ([(B,x'),(B,x'')] \circ e)$	s
$(\lambda x.t)[e]$	$[]$	$\not\to^{c}_{CBVN}$		

Table 6
The Cbvn-machine

Reduction under abstraction - improving the strategy

It is known [4] that no usual strategy based on environments can achieve Lévy's optimality. An efficient operational semantics that relaxes some of the demands of optimal reduction has been given in [14]. The idea is that we can take advantage of our ability to reduce under abstractions (since we are in an α-conversion free calculus), but not at the top-level, since we want to stop on a weak-head normal form. In particular, it is only useful to perform these extra reductions on a term which will be copied, in order to share these reductions. We refer the reader to [14] for the definition of an evaluation strategy for λ_c that interleaves a weak strategy with a stronger one, called only before an application of the δ rules.

4 Abstract machines with global environments

In the latter machines, each environment is coupled with a term whereas the KAM and CAM machines explicitly define an environment pointer in their transitions. Yet another way to couple terms with environments is to define a global lookup-table, an array to be precise, that contains the bindings. Intuitively, we will use variable names as array-indices, and reductions will side-effect the array structure. Below we present two machines based on global environments. First we define a machine that works with linear λ-terms and then we develop techniques that will

allow us to move to the full case.

The linear J-Machine

A closed λ-term is linear if every abstraction in a term binds exactly one variable occurrence. Such terms are a subset of pure λ_c-terms, that is terms without substitutions, where no sub-term contains instances of δ and ϵ-terms.

Definition 4.1 We define linear closure terms as tuples (t, a) where

- t is a linear λ-term and we assume all variable names are pairwise distinct, and
- a is an array of size equal to the number of distinct variables in t, representing the *global environment* (GEnv for short).

We write $Array\ (l, u)\ [\overline{(n, t)}]$ to denote an array with lower bound l and upper bound u that associates to the index n the term t. We write a_n to access the element at index n in the array a, and $a \mid [(n, u)]$ to destructively update the element at index n with u.

We proceed with the definition of the operational semantics. To evaluate a closed, linear term, we start with a linear closure (t, a) where a is empty, and use the array to store the substitutions as they are created. Values are closures where the term is an abstraction.

$$\frac{v\ value}{v \to v} \qquad \frac{t = a_x \quad t \to v}{(x, a) \to v} \qquad \frac{(t, a) \to (\lambda x.t', a') \quad (t', a' \mid [(x, u)]) \to v}{(tu, a) \to v}$$

The configurations of the corresponding abstract machine consist of a closure term and a stack (s) of pure λ-terms. Notice that no closure terms are build in the stack (cf. KAM and CAM). The transitions of the machine are given below:

Term	GEnv	Stack		Term	GEnv	Stack
tu	a	s	$\to^g_{LIN}$	t	a	$u : s$
$\lambda n.t$	a	$u : s$	$\to^g_{LIN}$	t	$a \mid [(n, u)]$	s
n	a	s	$\to^g_{LIN}$	a_n	a	s
$\lambda n.t$	a	$[]$	$\not\to^g_{LIN}$			

The work-flow of the machine is simple: we traverse the term, updating the global environment (a similar machine was defined in [13]). We do not overwrite any previously stored arguments in the array, since the machine does not copy any environments. Note that this machine does not require any external machinery and each transition takes $O(1)$ time. We obtain the result via the following read-back

function, where the overlined substitution is built using the associations in a.

$$\begin{aligned} r(t, a, [u_1 \ \ldots \ u_n]) &= r'(tu_1 \ldots u_n, a) \\ r'(t, a) &= t && \text{if } \mathrm{fv}(t) = \emptyset \\ r'(t, a) &= r'(t\overline{[u/n]}, a) && \text{if } \mathrm{fv}(t) = \overline{n} \end{aligned}$$

The δ-machine

The problem is the following: how do we use the previous array design in a non-linear way? Clearly, we cannot expect to fit all bindings in an array whose size is bound by the size of the initial term!

We follow the idea developed in the previous section and work with closure terms, but now using the set of pure λ_c-terms to build our closures, that is, terms in closures will also contain ϵ and δ. The design of the environment part is more involved since copying will take place and thus, each array index cannot identify a single binding. The solution is to work with an array of lists. Due to the unique variable name constraint and the linearity of λ_c-terms, we can still use names as indexes in the array. The array will be initialised with empty list structures, and as the computation progresses, the list at the element x in the array will contain the different terms that need to be substituted for x in different copies of the term.

The problem is then how to discriminate between all of the elements in such a list. Our solution is the following: the presence of δ-terms in the syntax gives a clear indication of which arguments are copied. All we have to do is to "tag" arguments with an identifier that indicates to which copy these belong; we then need an appropriate semantics that accesses the right copy at the right time. Summarising, each element in the array will contain a list of triples of the form (c, u, c') where the first and last elements are strings — $c := R \circ c \mid S \circ c \mid \epsilon$ — which we call *copy addresses*, used to identify copies of terms. The middle element is reserved for arguments which are pure λ_c-terms. More precisely, an element (c, u, c') in the list a_n indicates that the c copy of the variable n is associated to the c' copy of u. We use the auxiliary function $(a_n)_c^{find}$, which yields a triple from the list stored at array index n. The triple is found based on its first element c. We also use the auxiliary function $(a_n)_c^{del}$, which yields a list, where the triple identified by c is removed. Updating a list element of an array is done as before; the difference is that we may have multiple updates at several lists of the array: $a \mid \overline{[(n, l)]}$ where now $\overline{o}$ denotes a sequence of several index-element pairs. We now summarise the definition of closure terms.

Definition 4.2 A closure term is a triple (t, a, c) where t is a λ_c-term, c is a copy address, and $a = Array\ (l, u)\ \overline{[(n, b)]}$ is an array representing the global environment $GEnv$, where the elements b are lists of triples of the form (c, t, c') consisting of a copy address, a λ_c-term and another copy address.

The big-step operational semantics is provided next; followed by the definition

of the abstract machine which implements the semantics.

$$\frac{v \ value}{v \to v} \qquad \frac{(c,t,c') = (a_x)_c^{find} \quad (t, a \parallel [(x,(a_x)_c^{del})], c') \to v}{(x,a,c) \to v}$$

$$\frac{(t,a,c) \to (\lambda x.t', a', c') \quad (u,a',c) \to (u',a'',c'') \quad (t', a'' \parallel [(x,(c',u',c'') : (a''_x))], c') \to v}{(tu,a,c) \to v}$$

$$\frac{(c,u,c') = (a_x)_c^{find} \quad a' = a \parallel \begin{bmatrix} (x,(a_x)_c^{del}), \\ (y,(c,u,R \circ c') : a_y), \\ (z,(c,u,S \circ c') : a_z) \end{bmatrix} \quad (t, \mathrm{cpy}((c,u,c'), \mathrm{fv}(u), a'), c) \to v}{(\delta_x^{y,z}.t, a, c) \to v}$$

$$\frac{(c,u,c') = (a_x)_c^{find} \quad a' = a \parallel [(x,(a_x)_c^{del})] \quad (t, \mathrm{clr}((c,u,c'), \mathrm{fv}(u), a'), c) \to v}{(\epsilon_x.t, a, c) \to v}$$

The strings that we use as copy addresses come from the Geometry of Interaction interpretation for Linear Logic [22] and the path based realisation in [29]. There, sharing is maintained via contraction nodes (fan nodes) whose traversal builds $(R \mid S)^*$-strings to discriminate between shared contexts. In our case, sharable information is represented via δ-nodes, where we create copy addresses to discriminate between different copies. From a different point of view, one may argue that we implement on the fly α-conversion: our variable names consist of a fixed part (given at initialisation) and a volatile part (the copy address). Notice that the bindings of each copy are tagged by distinct addresses.

It is easy to see from the rules above that we only store evaluated expressions in the array structure and hence, these rules define a call-by-value strategy. We have used in the rules two auxiliary functions (cpy and clr) which facilitate copying and cleaning up elements in the environment. Copying is carried out by the definition in Table 7, which we prefer to present as a transition relation so that it can be directly used in the definition of the abstract machine; we use a stack (r) to keep track of recursive calls. Cleaning up behaves in a similar fashion: all we need to do is to modify the first rule in Table 7 such that the copies tr' and tr'' are omitted in the definition (i.e. nothing is concatenated). The effect of this is that we simply delete (or use up) unwanted triples.

The configurations of the corresponding abstract machine consist of a closure term and a stack: $t_c^Q : s \mid t_c^P : s \mid []$ where each stack cell is tagged with the usual constants P, Q and a copy-address c.

The machine is loaded with a λ_c-term, the array of lists where each list is initialised to the empty one and we set the *current-copy-address* to ϵ. The latter indicates that initially, we do not work inside a copy. The transitions are defined in

triple	fvs	GEnv	stack	triple	fvs	GEnv	stack
(c,u,to)	$f':f$	a	r	(c,u,to)	f	$a \mid [(f', tr' : tr'' : (a_{f'})^{del}_{to})]$	$(to,u',to'):r$
				where $(to,u',to') = (a_{f'})^{find}_{to}$ $tr' = ((S \circ to),u',(S \circ to'))$ $tr'' = ((R \circ to),u',(R \circ to'))$			
_	[]	a	$(c,u,to):r$	(c,u,to)	$fv(u)$	a	r
_	[]	a	[]				

Table 7
Copying in the δ-machine

Table 8. Here we use again the two external functions, namely *cpy* and *clr* but it is

Term	GEnv	CurC	Stack	Term	GEnv	CurC	Stack
tu	a	c	s	t	a	c	$(u)^Q_c : s$
$\lambda n.t$	a	c	$(u)^Q_{c'} : s$	u	a	c'	$(\lambda n.t)^P_c : s$
$\lambda n.t$	a	c	$(\lambda n'.t')^P_{c'} : s$	t'	$a \mid [(n', (c', \lambda n.t, c) : a_{n'})]$	c'	s
n	a	c	s	u	$a \mid [(n, (a_n)^{del}_c)]$	c'	s
				where $(c,u,c') = (a_n)^{find}_c$			
$\delta^{n'n''}_n.t$	a	c	s	t	$\mathrm{cpy}((c,u,c'),\mathrm{fv}(u),a')$	c	s
				where $(c,u,c') = (a_n)^{find}_c$ $a' = a \mid \begin{bmatrix} (n,(a_n)^{del}_c), \\ (n',(c,u,R \circ c') : a_{n'}), \\ (n'',(c,u,S \circ c') : a_{n''}) \end{bmatrix}$			
$\epsilon_n.t$	a	c	s	t	$\mathrm{clr}((c,u.c'),\mathrm{fv}(u),a')$	c	s
				where $(c,u,c') = (a_n)^{find}_c$ $a' = a \mid [\,(n,(a_n)^{del}_c)\,]$			
$\lambda n.t$	a	c	[]				

Table 8
δ-machine transitions

not too difficult to internalise these using a system of marks on the stack.

Stack	Term	c	k	y	x''	x'	x
$[]$	$(\lambda x.\delta_x^{x'\,x''}.x'x'')((\lambda y.y)(\lambda k.k))$	ϵ	$[]$	$[]$	$[]$	$[]$	$[]$
$((\lambda y.y)(\lambda k.k))_\epsilon^Q : []$	$\lambda x.\delta_x^{x'\,x''}.x'x''$	ϵ	$[]$	$[]$	$[]$	$[]$	$[]$
$(\lambda x.\delta_x^{x'\,x''}.x'x'')_\epsilon^P : []$	$(\lambda y.y)(\lambda k.k)$	ϵ	$[]$	$[]$	$[]$	$[]$	$[]$
$(\lambda k.k)_\epsilon^Q : (\lambda x.\delta_x^{x'\,x''}.x'x'')_\epsilon^P : []$	$\lambda y.y$	ϵ	$[]$	$[]$	$[]$	$[]$	$[]$
$(\lambda y.y)_\epsilon^P : (\lambda x.\delta_x^{x'\,x''}.x'x'')_\epsilon^P : []$	$\lambda k.k$	ϵ	$[]$	$[]$	$[]$	$[]$	$[]$
$(\lambda x.\delta_x^{x'\,x''}.x'x'')_\epsilon^P : []$	y	ϵ	$[]$	$(\lambda k.k)_\epsilon^\epsilon : []$	$[]$	$[]$	$[]$
$(\lambda x.\delta_x^{x'\,x''}.x'x'')_\epsilon^P : []$	$\lambda k.k$	ϵ	$[]$	$[]$	$[]$	$[]$	$[]$
$[]$	$\delta_x^{x'\,x''}.x'x''$	ϵ	$[]$	$[]$	$[]$	$[]$	$(\lambda k.k)_\epsilon^\epsilon : []$
$[]$	$x'x''$	ϵ	$[]$	$[]$	$(\lambda k.k)_S^\epsilon : []$	$(\lambda k.k)_R^\epsilon : []$	$[]$
$(x'')_\epsilon^Q : []$	x'	ϵ	$[]$	$[]$	$(\lambda k.k)_S^\epsilon : []$	$(\lambda k.k)_R^\epsilon : []$	$[]$
$(x'')_\epsilon^Q : []$	$(\lambda k.k)$	R	$[]$	$[]$	$(\lambda k.k)_S^\epsilon : []$	$[]$	$[]$
$(\lambda k.k)_R^P : []$	x''	ϵ	$[]$	$[]$	$(\lambda k.k)_S^\epsilon : []$	$[]$	$[]$
$(\lambda k.k)_R^P : []$	$(\lambda k.k)$	S	$[]$	$[]$	$[]$	$[]$	$[]$
$[]$	k	R	$(\lambda k.k)_S^R : []$	$[]$	$[]$	$[]$	$[]$
$[]$	$(\lambda k.k)$	S	$[]$	$[]$	$[]$	$[]$	$[]$

Table 9
δ-machine transitions with input $[\![(\lambda x.xx)((\lambda x.x)(\lambda x.x))]\!] \overset{\alpha}{\equiv} (\lambda x.\delta_x^{x'\,x''}.x'x'')((\lambda y.y)(\lambda k.k))$

Theorem 4.3 (Correctness) *The δ-machine implements the call-by-value operational semantics defined above on closure terms.*

Example 4.4 We compile the term $(\lambda x.xx)((\lambda x.x)(\lambda x.x))$ into a λ_c-term and α-convert as appropriate. For the sake of clarity, we use names instead of natural numbers, obtaining: $(\lambda x.\delta_x^{x'\,x''}.x'x'')((\lambda y.y)(\lambda k.k))$. The transitions are given in Table 9 on page 16; we write $u_{c'}^{c}$ instead of (c, u, c'). Note that δ-terms generate a *copy-address* for each copy and if we move inside a copy, we update the *current-copy-address*. The last line in the example shows the final state of the machine: the computation yields the identity function and the *current-copy-address* indicates that we jumped into the S-copy of the two copies that the δ-term generates. The external functions are not utilised, because the terms that we copy are closed.

5 Conclusions

Applying techniques inspired by the work in Linear Logic, we have derived two families of abstract machines for the λ-calculus. In the first one, environments associate to each sub-term, locally, the information needed about the bindings of the free variables. A syntax for terms with explicit copying and erasing constructs allows us to optimise the machines. The second family of machines also benefits from the use of a syntax with explicit copying and erasing, but uses a global environment, well-suited for compilation. The copy addresses that we generate can get quite big; a more compact representation would be more efficient, for instance, one could draw fresh addresses from a free-list. Finally, notice that there is no garbage collector involved. There is however a space leak if we do not make full copies of the copy address.

References

[1] M. Abadi, L. Cardelli, and P. l. Curien. Explicit substitutions. *Journal of Functional Programming*, 1:31–46, 1991.

[2] M.S. Ager, D. Biernacki, O. Danvy and J. Midtgaard. A Functional Correspondence between Evaluators and Abstract Machines. In *Proceedings of PPDP 2003*, ACM Press.

[3] F. Alberti. An abstract machine based on linear logic and explicit substitutions. MSc thesis, University of Birmingham, 1997.

[4] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[5] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, second, revised edition, 1984.

[6] Z. Benaissa and P. Lescanne. Super-Closures. In *Proc. of WPAM98, as Technical Report of the University of SaarBruck, number A 02/98*, 1998.

[7] U. Berger, M. Eberl, and H. Schwichtenberg. Normalization by evaluation, 1998.

[8] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8:173–202, 1987.

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

[10] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Progress in Theoretical Computer Science. Birkhauser, 1993.

[11] V. Danos and L. Regnier. Reversible, irreversible and optimal λ-machines. *Theoretical Computer Science*, 227(1–2):79–97, 1999.

[12] N. G. de Bruijn. Lambda calculus notation with nameless dummies. *Indagationes Mathematicae*, 34:381–392, 1972.

[13] M. Fernández and I. Mackie. Call by value lambda-graph rewriting — without rewriting. In *Proceedings of the Int. Conference on Graph Transformations (ICGT'02)*, Barcelona, 2002. LNCS 2505, Springer, 2002.

[14] M. Fernández, I. Mackie and F-R. Sinot. Closed Reduction: Explicit Substitutions without alpha-conversion. In *Mathematical Structures in Computer Science*, 15(2), 2005.

[15] M. Fernández, I. Mackie and F-R. Sinot. Lambda-Calculus with Director Strings. In *Applicable Algebra in Engineering, Communication and Computing*, 15(6), pages 393–437, Springer, 2005.

[16] D. P. Friedman, A. Ghuloum, J. G. Siek and L. Winebarger. Improving the Lazy Krivine Machine. In *Higher-Order and Symbolic Computation*, 2003.

[17] G. Kahn. Natural semantics. *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87 table of contents*, pages 22–39, 1987.

[18] G. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19. *Computer Science Department, Aarhus University*, 1981.

[19] N. Ghani, V. de Paiva and E. Ritter. Linear Explicit Substitutions. Journal of the IGPL, Vol. 8, No. 1, 2000.

[20] D. Kesner. The Theory of Explicit Substitutions Revisited. *proceedings of the 16th EACSL Annual Conference on Computer Science and Logic (CSL) LNCS 4646, pages 238-252, Lausanne, Switzerland*, 2007.

[21] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[22] J.-Y. Girard. Geometry of interaction 1: Interpretation of System F. In R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo, editors, *Logic Colloquium 88*, volume 127 of *Studies in Logic and the Foundations of Mathematics*, pages 221–260. North Holland Publishing Company, Amsterdam, 1989.

[23] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Proceedings of ICFP'02, Pittsburgh, Pennsylvania, USA*, 2002.

[24] J-L. Krivine. Un interprète du λ-calcul. Available online: `www.logique.jussieu.fr/~krivine`, 1985.

[25] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[26] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report, INRIA Rocquencourt, 1990.

[27] P. Lescanne. From $\lambda\sigma$ to λv a journey through calculi of explicit substitutions. In *Conference Record of POPL '94: 21ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon*, pages 60–69, New York, NY, 1994.

[28] I. Mackie. *The Geometry of Implementation*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, September 1994.

[29] I. Mackie. The geometry of interaction machine. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 198–208. ACM Press, January 1995.

[30] K. H. Rose. Explicit substitution – tutorial & survey. Technical Report LS-96-3, Sept. 1996. BRICS, Dept. of Comp. Sci., University of Aarhus, Denmark.

[31] F-R. Sinot. Stratégies efficaces et modèles d'implantation pour les langages fonctionnels. Thèse de doctorat, École Polytechnique, Palaiseau, France, 2006.