Journal of Functional Programming 2 (1): 23–59, January 1992

A categorical understanding of environment machines

ANDREA ASPERTI
INRIA, Rocquencourt, 78153 Le Chesnay, France
(Asperti@margaux.inria.fr)

Abstract

In the last two decades, category theory has become one of the main tools for the denotational investigation of programming languages. Taking advantage of the algebraic nature of the categorical semantics, and of the rewriting systems it suggests, it is possible to use these denotational descriptions as a base for research into more operational aspects of programming languages.

This approach proves to be particularly interesting in the study and the definition of environment machines for functional languages. The reason is that category theory offers a simple and uniform language for handling terms and environments (substitutions), and for studying their interaction (through application).

Several examples of known machines are discussed, among which the Categorical Abstract Machine of Cousineau *et al.* (1987) and Krivine's machine. Moreover, as an example of the power and fruitfulness of this approach, we define two original categorical machines. The first one is a variant of the CAM implementing a λ-calculus with both call-by-value and call-by-name as parameters passing modes. The second one is a variant of Krivine's machine performing complete reduction of λ-terms.

Capsule review

This paper significantly extends the original work of Cousineau *et al.* (1987) in using the framework of a cartesian closed category to define and reason about the architecture of an abstract machine (the CAM) capable of reducing terms of a lambda-calculus. The paper provides a formal, yet easily accessible account of Mauny's extension of the CAM to perform lazy evaluation, and of an elegant, previously unpublished abstract machine first described by Krivine.

In a more technical second part, intended primarily for specialists, the paper describes two further extensions to the CAM. One is a machine able to reduce a calculus with mixed strict and lazy evaluation rules; the other is a machine that implements strong reduction (reduction inside the bodies of abstraction terms). The mechanism for realizing substitutions is treated in detail. Correctness and termination proofs are given for this machine.

This paper demonstrates the applicability of a categorical framework to express detailed reasoning about rather complex computational structures, yet to do so in a manageable way. It should be of interest to those interested in applying formal methods to computer architecture, as well as those interested in abstract machines as models for compiler design.

0 Introduction

In 1985, Cousineau *et al.* (1987) presented, under the name of Categorical Abstract Machine (CAM), a new environment machine computing weak head normal forms of functional expressions (Mauny, 1985; Cousineau *et al.*, 1987). This machine was directly inspired by the categorical semantics of the λ-calculus, passing through an intermediate system of categorical combinators (Curien, 1986). The CAM was the first relevant application of the philosophy we mentioned in the abstract, and its interest is not merely theoretical: a complete ML-implementation based on this abstract machine (CAML) has been developed at INRIA-Rocquencourt, under the direction of G. Huet (Cousineau and Huet, 1989).

In this paper we pursue this approach, trying to show that the denotational description of programming languages, when properly formalized in a sufficiently algebraic framework (in our case, category theory), can provide a direct hint towards the implementation. The relevance for computer science is evident, since by reducing the gap between denotational and operational semantics, we greatly simplify the correctness proof of the implementation.

This approach proves to be particularly interesting in the study and definition of environment machines for functional languages. Indeed, category theory offers a simple and uniform language for handling terms and environments (substitutions), and for studying their interaction (through application).

The theoretical foundation of the CAM is provided by Curien's Categorical Combinators (Curien, 1986), that is, roughly, a rewriting system inspired by the algebraic formalization of a Cartesian closed category (an abstract notion of model for the simple typed λ-calculus). We shall not pursue this direction in our paper (i.e. we shall not investigate any new combinatorial system), but shall proceed directly to the definition of abstract machines. Our motivation is that a combinatorial system eventually reflects a *fixed* notion of model, which in turn is related to specific features of the programming language (as, for instance, the evaluation strategy). Since a model of β-*v* (call by value) is different from a model of β-*n* (call by name), we should derive different combinatorial systems. In the same way, if we are just interested in the computation of weak head normal forms, we can presumably define a simpler combinatorial system than in the case of strong reduction.

There is an important point to be clarified here. A combinatorial system is directed, that is, dynamic. On the contrary, we shall stick to the static (equational) theory. The main consequence is that we shall be unable to investigate all dynamic properties of an implementation, and in particular termination.

We shall thus restrict our attention to what we call 'the categorical soundness' of the abstract machines, i.e. a mapping $\mathscr{T}$ from machine states to categorical terms that defines an invariant during the computation. If $S \to S'$, then $\mathscr{T}(S) = \mathscr{T}(S')$ w.r.t. the equatorial theory defined by the suitable categorical abstract notion of model for the calculus under investigation.

Our point is that categorical soundness is a useful guide to implementation. Having a simple but informative invariant during the computation is a good way of improving our confidence in the machine, and an easy tool for checking errors.

The paper is structured in three parts. In the first part we revisit some known devices (the CAM, its lazy version, Krivine's machine). Our aim is to stress the relevance of the categorical formal approach to the λ-calculus, that provides an accurate framework for describing the computation, and in particular the substitution process. Indeed, in real implementations, substitutions are never coupled with the β-rule, but they are delayed and explicitly recorded. Moreover, substitution steps are interleaved with β-reductions, preventing us from looking at the substitution phase as a subroutine call. The need for a decomposition of the substitution process has been pointed out by several authors, and many different formalisms have been proposed with this aim. Most of them have been inspired by the λ-calculus (see Revesz, 1985; Kennaway, 1984; Sleep, 1985), maybe adopting De Bruijn notation (Balsters, 1986). The main idea of all these works is that the β-reduction rule and the rules for handling substitutions have the same logical status, requiring a formal language in which substitutions have a proper syntactical representation. Category theory provides such a language (Curien, 1986 was the first to put in evidence the operational relevance of this aspect of category theory.) The most recent formalisms aiming to the integration of code and environment (Curien, 1988; Abadi *et al.*, 1990; Hardin and Lévy, 1990) have actually been developed under the direct influence of the categorical approach (see also Hardin, 1987, for a comparison between categorical combinators and previous investigations).

The main original contribution of this part is a semantic explanation of the implementation of lazy evaluation on the top of the CAM due to M. Mauny (the general idea going back to Plotkin, 1975).

The idea is to interpret laziness by means of a retraction $A < A^*$, where A^* intuitively represents the collection of 'lazy values' of A. The retraction arrows freeze: $A \to A^*$ and unfreeze: $A^* \to A$ define the main categorical combinators for lazy evaluation. This new 'semantical' understanding of the lazy CAM allows us to make a more direct comparison with Krivine's machine (see Curien, 1988). As a matter of fact, Krivine's 'combinators' can be naturally understood as the result of integrating Curien's with the two combinators for lazy evaluation 'freeze' and 'unfreeze' (see section 3).

The second part of the paper is devoted to the definition of an abstract machine implementing a variant of the λ-calculus, where the user can choose between call-by-value and call-by-name for passing parameters to functional expressions (λ_{gl}-calculus, see Asperti, 1990). Our approach (close to the Algol 60 parameters passing mode) is essentially based on the existence of two different kinds of variables: the strict and the lazy ones. By abstracting over a strict variable we get call by value (strict abstraction); in the other case, we have call by name (lazy abstraction). The implementation requires a run time test, which is performed when an actual parameter M is passed to a procedure $\lambda x.P$ in some environment ξ: if x has been declared 'lazy', the evaluation of M is frozen, saving on the environment a closure $(\xi; \text{freeze}(M))$; otherwise we start the reduction of M in ξ. Handling this test correctly is not completely evident, and passing through the categorical denotational semantics is the best way for getting the right implementation (or, at least, for improving our confidence in its correctness).

Finally, Part III is devoted to the problem of strong reduction, i.e. of pursuing evaluation up to the computation of the normal form. In this case, the need of a clear theoretical integration between code and environment becomes even stronger, since with typical safe reduction strategies based on the computation of head normal forms we must be able to handle closures as part of the code (in order to share reductions). This requires a uniform syntactical setting for all these notions, which is naturally provided by category theory.

The main problem of strong reduction is due to the propagation of substitutions across λ-abstractions, since the environment must be suitably 'shifted'. Our abstract machine is based on the introduction of explicit shift combinators, in the spirit of Abadi *et al.* (1990), in order to support environment sharing. This machine seems to have a close relation to Crégut's machine (Crégut, 1990), but we have so far been unable to formally clarify this correspondence.

The paper is essentially self-contained and requires only very elementary notions of category theory.

PART I

This part of the work is devoted to a discussion of some existing environment machines for the evaluation of functional expressions. In particular, in section 1 we shall consider the Categorical Abstract Machine of Cousineau *et al.* (1987); in section 2 we discuss its lazy version (Mauny, 1985); sections 3 and 4 are devoted to Krivine's machine (never published by the author). For all of them we will stress the strong relation between their operational behaviour and category theory. This is well known in the case of the CAM, but not for the two other machines (some preliminary results in this direction are in Asperti, 1990). The relation between the previous machines and category theory is essentially based on the definition of a map from states to categorical terms (in an abstract categorical notion of λ-model). The main property of this map is that it defines an invariant during the computation, that is, if we pass from a state S to a new state S', then their categorical interpretation is equal. Henceforth, we shall refer to this property as the (categorical) *soundness* of the machine.

1 The Categorical Abstract Machine

It is well known that it is possible to give semantics to the type free lambda calculus over a reflexive object A in a Cartesian Closed Category (CCC) **C** (see, for instance, Asperti and Longo, 1991). Remember that an object V is reflexive if there exists a retraction pair $(\psi: A^A \to A, \phi: A \to A^A)$. In case this retraction is an isomorphism, we get an extensional λ-model.

Let M be a term of $\lambda\beta\eta$ with $FV(M) \subseteq \Delta = \{x_1, \ldots, x_n\}$. Let t be the terminal object in the CCC **C**. The interpretation $[M]_\Delta \in \mathbf{C}[A^n, A]$, where $A^n = (\ldots(t \times A) \times \ldots) \times A$ with n copies of A, is defined as follows (we use the two projections fst and snd in a 'polymorphic' fashion, omitting their indexes):

$$[x_i]_\Delta = \mathrm{snd} \circ \mathrm{fst}^{n-i} = \mathrm{pr}^n_i,$$

$$[MN]_\Delta = \mathrm{eval} \circ \langle \phi \circ [M]_\Delta, [N]_\Delta \rangle$$

$$[\lambda x.M]_\Delta = \psi \circ \Lambda([M]_{\Delta \cup \{x\}}).$$

This categorical interpretation suggests a very simple and yet efficient implementation of the lambda calculus (Mauny, 1985; Cousineau *et al.*, 1987). The implementation is based on a call-by-value, leftmost strategy of evaluation. The first step toward the implementation is the compilation of lambda calculus in a language of **categorical combinators**.

Note that $[MN]_\Delta = \text{eval} \circ \langle \phi \circ [M]_\Delta, [N]_\Delta \rangle = \Lambda^{-1}(\phi) \circ \langle [M]_\Delta, [N]_\Delta \rangle$, where $\Lambda^{-1}(\phi)$: $A \times A \to A$ is just the application u of the underlying combinatory algebra. We shall write **app** instead of $\Lambda^{-1}(\phi)$. Moreover let $\mathbf{cur}(f) = \psi \circ \Lambda(f)$, and write $f;g$ instead of $g \circ f$. Then, the equations which define the semantic interpretation of the lambda calculus are rewritten as follows:

$$[x_i]_\Delta = \text{fst}; \ldots \text{fst}; \text{snd} \quad \text{where fst appears } n-i \text{ times}$$
$$[MN]_\Delta = \langle [M]_\Delta, [N]_\Delta \rangle; \text{app}$$
$$[\lambda x.M]_\Delta = \text{cur}([M]_{\Delta \cup \{x\}}).$$

This provides a 'compilation' of the lambda calculus in a language where all the variables have been replaced with 'access paths' to the information they refer to (note the analogy with De Bruijn notation).

One of the main characteristics of the categorical approach is that we can use essentially the same language for representing both the code and the environment. An evaluation of the code C in an environment ξ is then the process of reduction of the term $\xi; C$. The reduction is defined by a set of rewriting rules. The general idea is that the environment should correspond to a categorical term in some normal form (typically, a weak head normal form). The reductions preserve this property of the environment, executing one instruction (i.e. one categorical combinator) of the code, and updating at the same time the program pointer to the following instruction. The computation starts with an empty environment (i.e. the identity).

For fst and snd we have the following rules, whose meaning is clear:

$$\langle \alpha, \beta \rangle; (\text{fst}; C_1) \Rightarrow \alpha; C_1,$$
$$\langle \alpha, \beta \rangle; (\text{snd}; C_1) \Rightarrow \beta; C_1.$$

In the left hand side of the previous rules, $\langle \alpha, \beta \rangle$ is the environment and the rest is the code. We shall use parentheses in such a way that the main semicolon in the expression will distinguish between the environment at its left, and the code at its right.

For $\text{cur}(C_1)$ we use the associative law of composition and delay the evaluation to another time:

$$\xi; (\text{cur}(C_1); C_2) \Rightarrow (\xi; \text{cur}(C_1)); C_2.$$

The structure $(\xi; \text{cur}(C_1))$ corresponds to what is usually called a **closure**.

The right time for evaluating a term of the kind $\text{cur}(C)$ is when it is applied to an actual parameter α. Then, we have:

$$\langle (\xi; \text{cur}(C_1)), \alpha \rangle; (\text{app}; C_2) \Rightarrow \langle \xi, \alpha \rangle; (C_1; C_2).$$

The previous rule is just a rewriting of the equation

$$\Lambda^{-1}(\phi)\circ\langle\psi\circ\Lambda(C_1)\circ\xi,\alpha\rangle = \mathrm{eval}\circ\langle\Lambda(C_1)\circ\xi,\alpha\rangle = C_1\circ\langle\xi,\alpha\rangle$$

that proves the semantical soundness of the previous rule.

Finally, we must consider the evaluation of a term of the kind $\langle C_1, C_2\rangle; C_3$. We have the formal equation:

$$\xi;(\langle C_1, C_2\rangle; C_3) = \langle\xi; C_1, \xi; C_2\rangle; C_3$$

but we cannot simply use it for defining a reduction, since we want also to reduce $\xi; C_1$ and $\xi; C_2$. We must first carry out independently the reductions of $\xi; C_1$ and $\xi; C_2$, and then put them together again building the new environment.

A simple solution on a sequential machine may be given by using a stack and working as follows: first save the current environment ξ by a *push* operation, then evaluate $\xi; c_1$ (that yields a new environment ξ_1); next *swap* the environment ξ_1 with the head of the stack (i.e. with ξ); now we can evaluate $\xi_1; C_2$ obtaining ξ_2; finally, build a pair $\langle\xi_1, \xi_2\rangle$ with the head of the stack ξ_1 and the current environment ξ_2 (that is a *cons* operation). An interesting and elegant property is that if we just write at compile time $\langle C_1, C_2\rangle$ as 'push; C_1; *swap*; C_2; cons', then the above behaviour is obtained by a sequential execution of this code.

Remark The compilation of λ-terms in linear code creates some problems in proving the soundness of the machine, since during evaluation the current code cannot be any longer interpreted as a categorical term (we loose the well balancing of pairing). However, the soundness is clear if we express the rule governing pairing by means of a conditional rewriting rule of the kind:

$$\frac{\xi; C_1 \Rightarrow M \qquad \xi; C_2 \Rightarrow N}{\xi;(\langle C_1, C_2\rangle; C_3) \Rightarrow \langle M, N\rangle; C_3}$$

with M and N in normal form.

The CAM is nothing else than a particular implementation, based on a stack, of this rule.

Definition 1.1

The ***compilation*** *by means of categorical combinators of a λ-term M in a 'dummy' environment $\Delta = (\ldots(nil, x_1), \ldots), x_n)$ is inductively defined as follows:*

$$\mathscr{C}am(x)_{(\Delta,x)} = snd$$

$$\mathscr{C}am(y)_{(\Delta,x)} = fst;\mathscr{C}am(y)_\Delta$$

$$\mathscr{C}am(MN)_\Delta = push;\mathscr{C}am(M)_\Delta;\ swap;\ \mathscr{C}am(N)_\Delta;\ cons;\ app$$

$$\mathscr{C}am(\lambda x.M)_\Delta = cur(\mathscr{C}am(M)_{(\Delta,x)})$$

Examples

1. The closed term $M = \mathrm{I}x.xx$ has the following compilation:

$$\begin{aligned}\mathscr{C}am(\lambda x.xx)_{\mathrm{nil}} &= \mathrm{cur}(\mathscr{C}am(xx)_{(\mathrm{nil},x)})\\ &= \mathrm{cur}(\mathrm{push};\mathscr{C}am(x)_{(\mathrm{nil},x)};\mathrm{swap};\mathscr{C}am(x)_{(\mathrm{nil},x)};\mathrm{cons};\mathrm{app})\\ &= \mathrm{cur}(\mathrm{push};\mathrm{snd};\mathrm{swap};\mathrm{snd};\mathrm{cons};\mathrm{app}).\end{aligned}$$

2. The term $(\lambda x.x)(\lambda x.x)$ is so compiled:

$$\begin{aligned}\mathscr{C}am((\lambda x.x)(\lambda x.x))_{\mathrm{nil}} &= \mathrm{push}; \mathscr{C}am(\lambda x.x)_{\mathrm{nil}}; \mathrm{swap}; \mathscr{C}am(\lambda x.x)_{\mathrm{nil}}; \mathrm{cons}; \mathrm{app} \\ &= \mathrm{push}; \mathrm{cur}(\mathscr{C}am(x)_{(\mathrm{nil},x)}); \mathrm{swap}; \mathrm{cur}(\mathscr{C}am(x)_{(\mathrm{nil},x)}); \mathrm{cons}; \\ &\qquad \mathrm{app} \\ &= \mathrm{push}; \mathrm{cur}(\mathrm{snd}); \mathrm{swap}; \mathrm{cur}(\mathrm{snd}); \mathrm{cons}; \mathrm{app}.\end{aligned}$$

Definition 1.2
The ***reduction*** *of the compiled code is summarized in table* 1:

Table 1

Before			After		
Environment	Code	Stack	Environment	Code	Stack
$\langle\alpha,\beta\rangle$	fst; C	S	α	C	S
$\langle\alpha,\beta\rangle$	snd; C	S	β	C	S
ξ	cur(C_1); C_2	S	ξ; cur(C_1)	C_2	S
$\langle\xi;\mathrm{cur}(C_1),\alpha\rangle$	app; C_2	S	$\langle\xi,\alpha\rangle$	C_1; C_2	S
ξ	push; C	S	ξ	C	$\xi.S$
ξ_1	swap; C	$\xi_2.S$	ξ_2	C	$\xi_1.S$
ξ_1	cons; C	$\xi_2.S$	$\langle\xi_2,\xi_1\rangle$	C	S

Example
The code 'push; cur(snd); swap; cur(snd); cons; app' corresponding to the λ-term $(\lambda x.x)(\lambda x.x)$ gives rise to the following computation:

ENV. = id
CODE = push; cur(snd); swap; cur(snd); cons; app
STACK = ∅

ENV. = id
CODE = cur(snd); swap; cur(snd); cons; app
STACK = id

ENV. = id; cur(snd)
CODE = swap; cur(snd); cons; app
STACK = id

ENV. = id
CODE = cur(snd); cons; app
STACK = id; cur(snd)

ENV. = id; cur(snd)
CODE = cons; app
STACK = id; cur(snd)

ENV. $= \langle \mathrm{id}; \mathrm{cur}(\mathrm{snd}), \mathrm{id}; \mathrm{cur}(\mathrm{snd}) \rangle$
CODE $= \mathrm{app}$
STACK $= \varnothing$

ENV. $= \langle \mathrm{id}, \mathrm{id}; \mathrm{cur}(\mathrm{snd}) \rangle$
CODE $= \mathrm{snd}$
STACK $= \varnothing$

ENV. $= \mathrm{id}; \mathrm{cur}(\mathrm{snd})$
CODE $=$
STACK $= \varnothing$.

Note that 'cur(snd)' is the compilation of $\lambda x.x$.

2 Lazy evaluation

The relevance, in actual programming, of a lazy evaluation mechanism (call-by-name + sharing) is well known. In particular, it may avoid useless computations and, if properly combined with recursion, it allows the manipulation of infinite structures.

The aim of the next two sections is to attempt a categorical explanation of these 'intentional' aspects of the computation, providing in this way a semantical justification for the completely lazy CAM described in Maung (1985). In particular, in this section we face the problem of giving a semantical interpretation to two combinators for 'freezing' and 'unfreezing. These two combinators are the basic operations for dealing with lazy evaluation; even abstract machines as Krivine's, where freezing and unfreezing operations do not explicitly appear, can be profitably understood in terms of these combinators (see Section 3).

The main operational property of 'freeze' and 'unfreeze' is that unfreeze $\circ$ freeze $=$ id, or, in categorical terms, that they define a retraction. Since in this paper we are mainly interested in semantics as a tool for guiding the implementation, we adopt a semantical approach nas naïve as close to the operational intuition, 'explaining' laziness by means of retraction $A < A^*$, where A^* intuitively represents the collection of 'lazy values' of A.

A more accurate approach, suggested by P. L. Curien, will be briefly discussed at the end of this section.

Remark
A canonical, natural, choice for A^* is the exponent A^t, where t is the terminal object of the category. As the reader can easily by himself, this choice suggests a simple simulation of Mauny's lazy abstract machine in the CAM, that is essentially the old idea of embedding call-by-name in a call-by-value paradigm by protecting the parameters with a λ-abstraction (see Plotkin, 1975).

Suppose we have an object A^*, such that $A < A^*$, and let (freeze: $A \to A^*$, unfreeze: $A^* \to A$) be the retraction pair.

An elementary but interesting property of retractions is that they are inherited in higher order objects.

Lemma 2.1
If $A < A^*$ *via* (*freeze, unfreeze*), *then* $A^A < A^{(A^*)}$ *via*

$$\Theta = \Lambda(eval \circ id \times unfreeze) : A^A \to A^{(A^*)},$$
$$\Psi = \Lambda(eval \circ id \times freeze) : A^{(A^*)} \to A^A.$$

Proof

$$\begin{aligned}
\Psi \circ \Theta &= \Lambda(\mathrm{eval} \circ \mathrm{id} \times \mathrm{freeze}) \circ \Lambda(\mathrm{eval} \circ \mathrm{id} \times \mathrm{unfreeze}) \\
&= \Lambda(\mathrm{eval} \circ \mathrm{id} \times \mathrm{freeze} \circ \Lambda(\mathrm{eval} \circ \mathrm{id} \times \mathrm{unfreeze}) \times \mathrm{id}) \\
&= \Lambda(\mathrm{eval} \circ \Lambda(\mathrm{eval} \circ \mathrm{id} \times \mathrm{unfreeze}) \times \mathrm{id} \circ \mathrm{id} \times \mathrm{freeze}) \\
&= \Lambda(\mathrm{eval} \circ \mathrm{id} \times \mathrm{unfreeze} \circ \mathrm{id} \times \mathrm{freeze}) \\
&= \Lambda(\mathrm{eval}) \\
&= \mathrm{id}. \quad \square
\end{aligned}$$

Lemma 2.2
$eval \circ \Theta \times freeze = eval : A^A \times A \to A.$

Proof

$$\begin{aligned}
\mathrm{eval} \circ \Theta \times \mathrm{freeze} &= \mathrm{eval} \circ \Lambda(\mathrm{eval} \circ \mathrm{id} \times \mathrm{unfreeze}) \times \mathrm{freeze} \\
&= \mathrm{eval} \circ \Lambda(\mathrm{eval} \circ \mathrm{id} \times \mathrm{unfreeze}) \times \mathrm{id} \circ \mathrm{id} \times \mathrm{freeze} \\
&= \mathrm{eval} \circ \mathrm{id} \times \mathrm{unfreeze} \circ \mathrm{id} \times \mathrm{freeze} \\
&= \mathrm{eval} \circ \mathrm{id} \times (\mathrm{unfreeze} \circ \mathrm{freeze}) \\
&= \mathrm{eval} \circ \mathrm{id} \times \mathrm{id} \\
&= \mathrm{eval}. \quad \square
\end{aligned}$$

Corollary 2.3
Let M, N *be two* λ*-terms with free variables in* Δ, *and let* $[\_]$ *be the categorical interpretation defined in the previous section. Then* $[MN]_\Delta = eval \circ \langle \Theta \circ \phi \circ [M]_\Delta, freeze \circ [N]_\Delta \rangle$.

Proof

$$\begin{aligned}
[MN]_\Delta &= \mathrm{eval} \circ \langle \phi \circ [M]_\Delta, [M]_\Delta \rangle \\
&= \mathrm{eval} \circ \Theta \times \mathrm{freeze} \circ \langle \phi \circ [M]_\Delta, [N]_\Delta \rangle \\
&= \mathrm{eval} \circ \langle \Theta \circ \phi \circ [M]_\Delta, \mathrm{freeze} \circ [N]_\Delta \rangle. \quad \square
\end{aligned}$$

We define now a new interpretation $[\![\_]\!]$ for the λ-calculus. By this semantics, a term M of $\lambda\beta\eta$ such that $FV(M) \subseteq \Delta = \{x_1, \ldots, x_n\}$ will not be interpreted as an arrow from A^n to A, but as an arrow from $(A^*)^n$ to A. The intuition should be clear: since we wish to give semantics to a lazy calculus, a term M must take values in a lazy environment $(A^*)^n$.

Definition 2.4
Let A and A^* as before. The ***lazy interpretation*** *of a λ-term M is inductively defined as follows*:

$$[\![x_i]\!]_\Delta = \mathit{unfreeze} \circ \mathit{snd} \circ \mathit{fst} \circ \ldots \circ \mathit{fst} \quad \text{where } \mathit{fst} \text{ appear } n-i \text{ times}$$
$$[\![MN]\!]_\Delta = \mathit{eval} \circ \#\Theta \circ \phi \circ \langle [\![M]\!]_\Delta, \mathit{freeze} \circ [\![N]\!]_\Delta \rangle$$
$$[\![\lambda x.M]\!]_\Delta = \psi \circ \Psi \circ \Lambda([\![M]\!]_{\Delta \cup \{x\}}).$$

The relation between the two interpretations is expressed by the following theorem:

Theorem 2.5
Let M be a term of $\lambda\beta\eta$ such that $FV(M) \subseteq \Delta = \{x_1, \ldots, x_n\}$. Then $[M]_\Delta = [\![M]\!]_\Delta \circ \mathit{id}_t \times \mathit{freeze} \times \ldots \times \mathit{freeze}$.

Proof
By induction on the structure of M.
— case $M = x$.

$[\![x_i]\!]_\Delta \circ \mathit{id}_t \times \mathit{freeze} \times \ldots \times \mathit{freeze} =$
$= \mathrm{unfreeze} \circ \mathrm{snd} \circ \mathrm{fst} \circ \ldots \circ \mathrm{fst} \circ \mathrm{id}_t \times \mathrm{freeze} \times \ldots \times \mathrm{freeze}$
$= \mathrm{unfreeze} \circ \mathrm{freeze} \circ \mathrm{snd} \circ \mathrm{fst} \circ \ldots \circ \mathrm{fst}$
$= \mathrm{snd} \circ \mathrm{fst} \circ \ldots \circ \mathrm{fst}$
$= [x_i]_\Delta.$

— case $M = \lambda x.N$

$[\![\lambda x.M]\!]_\Delta \circ \mathrm{id}_t \times \mathrm{freeze} \times \ldots \times \mathrm{freeze}$
$= \psi \circ \Psi \circ \Lambda([\![M]\!]_{\Delta \cup \{x\}}) \circ \mathrm{id}_t \times \mathrm{freeze} \times \ldots \times \mathrm{freeze}$
$= \Psi \circ \Lambda(\mathrm{eval} \circ \mathrm{id} \times \mathrm{freeze}) \circ \Lambda([\![M]\!]_{\Delta \cup \{x\}}) \circ \mathrm{id}_t \times \mathrm{freeze} \times \ldots \times \mathrm{freeze}$
$= \Psi \circ \Lambda(\mathrm{eval} \circ \mathrm{id} \times \mathrm{freeze} \circ \Lambda([\![M]\!]_{\Delta \cup \{x\}}) \times \mathrm{id}) \circ \mathrm{id}_t \times \mathrm{freeze} \times \ldots \times \mathrm{freeze}$
$= \Psi \circ \Lambda(\mathrm{eval} \circ \Lambda([\![M]\!]_{\Delta \cup \{x\}}) \times \mathrm{id} \circ \mathrm{id} \times \mathrm{freeze}) \circ \mathrm{id}_t \times \mathrm{freeze} \times \ldots \times \mathrm{freeze}$
$= \Psi \circ \Lambda([\![M]\!]_{\Delta \cup \{x\}} \circ \mathrm{id} \times \mathrm{freeze}) \circ \mathrm{id}_t \times \mathrm{freeze} \times \ldots \times \mathrm{freeze}$
$= \Psi \circ \Lambda([\![M]\!]_{\Delta \cup \{x\}} \circ \mathrm{id} \times \mathrm{freeze} \circ (\mathrm{id}_t \times \mathrm{freeze} \times \ldots \times \mathrm{freeze}) \times \mathrm{id})$
$= \Psi \circ \Lambda([\![M]\!]_{\Delta \cup \{x\}} \circ (\mathrm{id}_t \times \mathrm{freeze} \times \ldots \times \mathrm{freeze} \times \mathrm{freeze}))$
$= \Psi \circ \Lambda([M]_{\Delta \cup \{x\}})$
$= [\lambda x.M]_\Delta.$

— case $M = PQ$

$[\![PQ]\!]_\Delta \circ \mathrm{id}_t \times \mathrm{freeze} \times \ldots \times \mathrm{freeze} =$
$= \mathrm{eval} \circ \langle \Theta \circ \phi \circ [\![P]\!]_\Delta, \mathrm{freeze} \circ [\![Q]\!]_\Delta \rangle \circ \mathrm{id}_t \times \mathrm{freeze} \times \ldots \times \mathrm{freeze}$
$= \mathrm{eval} \circ \langle \Theta \circ \phi \circ [\![P]\!]_\Delta \circ \mathrm{id}_t \times \mathrm{freeze} \times \ldots \times \mathrm{freeze}, \mathrm{freeze} \circ [\![Q]\!]_\Delta \circ$
$\mathrm{id}_t \times \mathrm{freeze} \times \ldots \times \mathrm{freeze} \rangle$
$= \mathrm{eval} \circ \langle \Theta \circ \phi \circ [P]_\Delta, \mathrm{freeze} \circ [Q]_\Delta \rangle$
$= [PQ]_\Delta.$ □

Corollary 2.6
Let M be a closed term of $\lambda\beta\eta$. *Then* $[M] = [\![M]\!]$.

Let us come to the implementation. We define a new set of categorical combinators for lazy evaluation. One of our combinators will be **unfreeze**: $A^* \to A$; for freezing, we introduce a combinator **freeze**(_), with the following intended interpretation:

$$\textbf{freeze}(f) = \text{freeze} \circ f : (A^*)^n \to A^*, \quad \text{where} \quad f : (A^*)^n \to A.$$

Moreover, in the same way the isomorphism $\phi : A \to A^A$ and $\psi : A^A \to A$ are 'incorporated' in the intended semantics of app and cur, we can avoid an explicit handling of $\Theta : A^A \to A^{(A^*)}, \Psi : A^{(A^*)} \to A^A$ (remember that $\text{app} = \text{eval} \circ \phi \times \text{id}$, and $\text{cur}(f) = \psi \circ \Lambda(f)$). Then, we have two combinators **lazy_app** and **lazy_cur**(_) such that:

$$\text{lazy\_app} = \text{eval} \circ (\Theta \circ \phi) \times \text{id} : A \times A^t \to A$$

$$\text{lazy\_cur}(f) = \psi \circ \Psi \circ \Lambda(f) : (A^*)^n \to A. \quad \text{where} \quad f : (A^*)^{n+1} \to A.$$

The lazy interpretation in 2.4 suggests now the following compilation:

Definition 2.7
The (completely) ***lazy compilation*** *of a* λ*-term* M *in a 'dummy' environment* $\Delta = (\ldots(nil, x_1), \ldots), x_n)$ *is inductively defined as follows*:

$$\mathscr{C}_\ell(x)_{(\Delta, x)} = snd; unfreeze$$

$$\mathscr{C}_\ell(y)_{(\Delta, x)} = fst; \mathscr{C}_\ell(y)_\Delta$$

$$\mathscr{C}_\ell(MN)_\Delta = push; \mathscr{C}_\ell(M)_\Delta; swap; freeze\ (\mathscr{C}_\ell(N)_\Delta); cons; lazy\_app$$

$$\mathscr{C}_\ell(\lambda x.M)_\Delta = lazy\_cur(\mathscr{C}_\ell(M)_{\Delta, x)}.$$

We now come to the evaluation of the generated code C in an environment ξ. Note that since we are working lazily, the environment in this case will not be an arrow from the terminal object to a suitable power of A, but an arrow of the kind $\xi : t \to t \times (A^*)^n$. The reduction is defined by the following set of rewriting rules.

For fst and snd we have the usual rules:

$$\langle \alpha, \beta \rangle ; (\text{fst}; C_1) \Rightarrow \alpha; C_1$$

$$\langle \alpha, \beta \rangle ; (\text{snd}; C_1) \Rightarrow \beta; C_1.$$

For lazy_cur(C_1), we work as for cur, using the associative law of composition and delaying the evaluation to another time:

$$\xi ; (\text{lazy\_cur}(C_1); C_2) \Rightarrow (\xi ; \text{lazy\_cur}(C_1)); C_2.$$

The closure $(\xi ; \text{lazy\_cur}(C_1))$ is opened when it is applied to an actual parameter α. We then have:

$$\langle (\xi ; \text{lazy\_cur}(C_1)), \alpha \rangle ; (\text{lazy\_app}; C_2) \Rightarrow \langle \xi, \alpha \rangle ; (C_1 ; C_2).$$

The previous rule is justified by the following semantical equation:

$$\mathrm{eval} \circ (\Theta \circ \phi) \times \mathrm{id} \circ \langle \psi \circ \Psi \circ \Lambda(C_1) \circ \xi, \alpha \rangle = \mathrm{eval} \circ \langle \Lambda(C_1) \circ \xi, \alpha \rangle = C_1 \circ \langle \xi, \alpha \rangle.$$

Also the rule for freeze is essentially analogous to the one for cur: we do not proceed in the evaluation, but we delay it until an unfreeze operation is met:

$$\xi; (\mathrm{freeze}(C_1); C_2) \Rightarrow (\xi; \mathrm{freeze}(C_1)); C_2$$
$$(\xi; \mathrm{freeze}(C_1)); (\mathrm{unfreeze}; C_2) \Rightarrow \xi; (C_1; C_2).$$

The first rule does not need to be semantically justified: it is just an application of the associative law. As for the soundness of the second rule, it is an obvious consequence of the fact that freeze and unfreeze are inverse of each other.

The rules for push, pop and swap are the usual ones.

Since lazy_cur and lazy_app behave exactly like cur and app, we shall omit in the following the prefix 'lazy'.

The previous rules are summarized in the following table:

Definition 2.8
The ***completely lazy abstract machine*** *is described by table* 2:

This machine was described for the first time in Mauny (1985), *where it was derived from operational intuition.*

Remark
(For Categoricians) P. L. Curien suggested that, instead of focusing on a retraction, a *comonad* could provide a more appropriate account of the relation between lazy and strict evaluation (Theorem 2.5 is not very informative). The idea of using a comonad is essentially suggested by the following two facts:

1. **freeze**(_) is an unary combinator, and it seems unnatural to rely on a 0-ary combinator freeze.
2. We do not actually need a retraction unfreeze $\circ$ freeze = id; it suffices to have the weaker equation

$$(\ddagger) \quad \mathrm{unfreeze} \circ \mathrm{freeze}(f) = f$$

Suppose that the functor $(\_)^*$ taking an object A to the collection A^* of lazy values of type A is a comonad $((\_)^*, \delta, \varepsilon)$. Then unfreeze $= \varepsilon_A : A^* \to A$, $\mathrm{freeze}(f) = \delta \circ (f)^*$, and $(\ddagger)$ results from the comonad equation $\delta_A \circ \varepsilon_{A^*} = \mathrm{id}$.

Consider now a monoidal, cartesian category $(\mathbf{C}, \otimes, \times)$ closed w.r.t. the monoidal product $\otimes$ (a simple example is the category **pSet** of sets with partial functions). Let us call e and t the identity-objects for $\otimes$ and $\times$, respectively. By the categorical semantics of linear logic (Girard, 1986; Seely, 1987), if T is a comonad over $\mathbf{C}$, and

1. $T(t) = e$
2. $T(A \times B) = T(A) \otimes T(B)$

then the co-Kleisli category associated with T is cartesian closed (with $\otimes$ as cartesian

Table 2

Before			After		
Environment	Code	Stack	Environment	Code	Stack
$\langle\alpha,\beta\rangle$	fst; C	S	α	C	S
$\langle\alpha,\beta\rangle$	snd; C	S	β	C	S
ξ	cur(C_1); C_2	S	ξ; cur(C_1)	C_2	S
$\langle\xi;\text{cur}(C_1),\alpha\rangle$	app; C_2	S	$\langle\xi,\alpha\rangle$	$C_1;C_2$	S
ξ	freeze(C_1); C_2	S	ξ; freeze(C_1)	C_2	S
ξ; freeze(C_1)	unfreeze; C_2	S	ξ	$C_1;C_2$	S
ξ	push; C	S	ξ	C	$\xi.S$
ξ_1	swap; C	$\xi_2.S$	ξ_2	C	$\xi_1.S$
ξ_1	cons; C	$\xi_2.S$	$\langle\xi_2,\xi_1\rangle$	C	S

product), i.e. a model of $\beta-n$ (call by name). By adding 'sufficient information' on the monoidal product $\otimes$, we may turn **C** from a model of a linear calculus into a model of $\beta-v$ (call by value). The idea would be to use some well known formalization of categories with partial maps (see, for instance, Robinson & Rosolini, 1988; Curien and Obtulowicz, 1989). However, by adding this information about **C**, we may presumably relax some of the previous assumptions. Moreover, it is not clear yet if a lifting would not provide a more intuitive description than a simple comonad. For instance, a lifting, being left adjoint to an inclusion functor, would preserve limits, and eqns. (1) and (2) above would be automatically satisfied. Note that as soon as we have a lifting, we also have a retraction $A < A^*$, but the naturality condition of the comonad (or the lifting) seems to be essential to prove more informative results than Theorem 2.5.

We plan to provide a more convincing and complete account of the semantic explanation of the relations between $\beta-n$ and $\beta-v$ in a forthcoming joint work by P. L. Curien.

3 comparison with Krivine's machine

The aim of this section is to provide a comparison between the completely lazy CAM and Krivine's abstract machine for lazy evaluation (unpublished, but implemented; see Curien, 1988, for a detailed presentation, and a comparison with the TIM machine of Fairbairn and Wray, 1987). Krivine's machine is a very simple environment machine for implementing lazy evaluation. Both the term M to be evaluated and the environment ξ are represented by graphs; we have two points A and B for accessing these graphs. The first graph remains fixed during the computation, while the second one keeps growing. As usual, the recursive calls to the evaluation are implemented by a stack, where we accumulate closures, that is, pairs of pointers of the form (B, A).

The machine is so simple as to be self-explanatory, so we start straight away with its formal definition. We suppose the term is represented in De Bruijn notation,

adopting the syntax cur(M) for representing the abstraction, in order to make the comparison with the completely lazy machine more explicit.

Definition 3.1
***Krivine's lazy abstract machine** is described by table 3*

Table 3

Before			After		
Environment	Code	Stack	Environment	Code	Stack
$\langle\alpha,\beta\rangle$	$n+1$	S	α	n	S
$\alpha,(\xi;N)\rangle$	0	S	ξ	N	S
$\langle\alpha,\beta\rangle$	0	S	β	END	$\varnothing$
ξ	cur(M)	$\alpha.S$	$\langle\xi,\alpha\rangle$	M	S
ξ	MN	S	ξ	M	$(\xi;N).S$

The first striking difference with respect to the completely lazy abstract machine is the lack of explicit freezing and unfreezing operations. As for the unfreeze combinator, it is easily seen that in the code generated by the lazy compilation in Definition 2.7 it will be always preceded by snd, and conversely every snd is followed by an unfreeze instruction. Thus, we can combine these two combinators in a new combinator snd′ = snd; unfreeze whose behaviour is then described by the following rule:

Environment	Code	Stack	Environment	Code	Stack
$\langle\alpha,\xi;\text{freeze}(C_1)\rangle$	$\text{snd}';C_2$	S	ξ	$C_1;C_2$	S

This rule corresponds to the second rule in Krivine's machine.

As for the freeze(_) combinator, its existence is justified by the fact that the CAM works on a linear code, while Krivine's machine operates on a graph representing the term. Indeed, the freeze combinator is nothing but 'pair of brackets' limited the extent of the linear code under consideration. The actual implementation of the Krivine's machine, which is still based on a compilation into linear code, restores the use of the explicit freeze(_) (as it was also suggested by the notation $(N)M$ for MN, which Krivine used for a while).

The second difference between the two abstract machines is in the evaluation order for the terms in an application MN. We can easily define a new set of categorical combinators which implements this kind of evaluation strategy, based on the isomorphism $A\times B\cong B\times A$. We encapsulate this isomorphism in cons, obtaining a new combinator cons′.

The compilation of a term MN becomes now

$$\mathscr{C}_\ell(MN)_\Delta = \text{push};\text{freeze}(\mathscr{C}_\ell(N)_\Delta);\text{swap};\mathscr{C}_\ell(M)_\Delta;\text{cons}';\text{app}$$

and the behaviour of cons′ is described by the rule

Environment	Code	Stack	Environment	Code	Stack
ξ_1	$\text{cons}';C$	$\xi_2.S$	$\langle\xi_1,\xi_2\rangle$	C	$\xi_1.S$

Now, Krivine's rule

Environment	Code	Stack	Environment	Code	Stack
ξ	MN	S	ξ	M	$(\xi;N).S$

is the composition of three more elementary steps, namely the creation of a new pointer to the current environment ξ, the definition of the closure $(\xi;N)$ and storing it on top of the stack. But this is exactly analogous to the sequential execution of the free combinators push;freeze(C);swap. Indeed:

ENV. $= \xi$
CODE $=$ push; freeze(C_1); swap; C_2
STACK $= S$

ENV. $= \xi$
CODE $=$ freeze(C_1); swap; C_2
STACK $= \xi.S$

ENV. $= \xi$;freeze(C_1)
CODE $=$ swap; C_2
STACK $= \xi.S$

ENV. $= \xi$
CODE $=$ swap; C_2
STACK $= \xi$;freeze(C_1).S

Gluing together these three elementary steps in a single operation we can obviously save some machine instructions in an actual implementation, but up to now Krivine's machine does not provide any real theoretical improvement with respect to the completely lazy CAM. The big improvement is in the way the abstraction is handled. Consider the evaluation of the term MN. M must reduce to something of the form cur(P). The CAM builds a closure of this code with the current environment, but this closure is immediately opened by an app operation! More explicitly, we have the following typical sequence of reductions:

ENV. $= \xi$
CODE $=$ cur(C_1); cons′; app; C_2
STACK $= \alpha.S$.

ENV. $= \xi$;cur(C_1)
CODE $=$ cons′; app; C_2
STACK $= \alpha.S$.

ENV. $= \langle \xi;$ cur(C_1), $\alpha \rangle$
CODE $=$ app; C_2
STACK $= S$.

ENV. $= \langle \xi, \alpha \rangle$
CODE $= C_1; C_2$
STACK $= S$.

The idea is to encapsulate the three combinators cur(_); cons′; app; in a single new combinator cur′(_), whose behaviour must be equivalent to the sequential execution of the previous ones. The compilation of abstraction and application with this new combinator is:

$$\mathscr{C}_\ell(MN)_\Delta = \text{push}; \text{freeze}(\mathscr{C}_\ell(N)_\Delta); \text{swap}; \mathscr{C}_\ell(M)_\Delta,$$
$$\mathscr{C}_\ell(\lambda x.M)_\Delta = \text{cur}'(\mathscr{C}_\ell(M)_{(\Delta,x)}).$$

Moreover, it is not difficult to prove that at run time we cannot have any code following an instruction of the form cur′(C). Then we have the following rule, which is exactly the equivalent of Krivine's rule for cur(_).

Environment	Code	Stack	Environment	Code	Stack
ξ	cur′(C)	$\alpha.S$	$\langle \xi, \alpha \rangle$	C	S

4 The categorical nature of Krivine's machine

In the previous section we have provided a comparison between Mauny's lazy CAM and Krivine's machine. However, this is not the best way to understand the 'categorical' features of the latter. As we have observed, Mauny's machine works on linear code, while Krivine's machine operates on a graph representing the term. It is thus unnatural to pass through CAM instructions when the purely categorical description is much closer. We now define a very simple map $\mathscr{T}$ from machine states to categorical terms. The main property of this map is that if $S \to S'$, then $\mathscr{T}(S) = \mathscr{T}(S')$ as categorical terms. This property is what we call the 'categorical soundness' of the machine.

Recall that a state in Krivine's machine is a triple (ξ, C, S), where ξ is the current environment, C is the current code, and S is the stack, containing closures.

We consider an integer n as an abbreviation for fst;...;fst;snd, with n occurrences of fst, and (MN) as an abbreviation for $\langle M, N \rangle$;app. Note that in this way the environment, the code and the items on the stack (the closures) can be all understood as categorical terms.

Definition 4.1
The map $\mathscr{T}(S)$ from machine states to categorical terms in a categorical model **C**, *is defined as follows*:

$$\mathscr{T}(\xi, END, \varnothing) = \xi$$
$$\mathscr{T}(\xi, C, S) = \mathscr{T}'(\xi; C, S),$$

where:

$$\mathscr{T}'(M, \varnothing) = M$$
$$\mathscr{T}'(M, N.S) = \mathscr{T}'((MN), S).$$

By the definition of $\mathscr{T}$, it is evident that if $\sigma = (\xi, C, \chi_1, \ldots, \chi_r)$ and $\sigma' = (\xi', C', \chi_1', \ldots, \chi_r')$ are two states such that, $\xi = \xi'$, $C = C'$ and, for any i, $\chi_i = \chi_i'$, then $\mathscr{T}(\sigma) = \mathscr{T}(\sigma')$ (where all the previous equalities are intended between categorical terms). The same holds for $\mathscr{T}'$.

Theorem 4.2 (soundness)
If $(\xi, C, S) \to (\xi', C', S')$, *then* $\mathscr{T}(\xi, C, S) = \mathscr{T}(\xi', C', S')$ *as categorical terms.*

Proof
We only consider one step derivations. The proof is by case analysis of the operational rules.

1. $(\langle \alpha, \beta \rangle, n+1, S) \to (\alpha, n, S)$.

$$\begin{aligned}\mathscr{T}(\langle \alpha, \beta \rangle, n+1, S) &= \mathscr{T}'(\langle \alpha, \beta \rangle; n+1, S) \\ &= \mathscr{T}'(\alpha; n, S) \\ &= \mathscr{T}(\alpha, n, S).\end{aligned}$$

2. $(\langle \alpha, (\xi; N) \rangle, 0, S) \to (\xi, N, S)$.

$$\begin{aligned}\mathscr{T}(\langle \alpha, (\xi; N) \rangle, 0, S) &= \mathscr{T}'(\langle \alpha, (\xi; N) \rangle; 0, S) \\ &= \mathscr{T}'(\xi; N, S) \\ &\mathscr{T}(\xi, N, S).\end{aligned}$$

3. $(\langle \alpha, \beta \rangle, 0, \varnothing) \to (\beta, \mathrm{END}, \varnothing)$.

$$\begin{aligned}\mathscr{T}(\langle \alpha, \beta \rangle, 0, \varnothing) &= \mathscr{T}'(\langle \alpha, \beta \rangle; 0, \varnothing) \\ &= \mathscr{T}'(\beta, \varnothing) \\ &= \beta \\ &= \mathscr{T}(\beta, \mathrm{END}, \varnothing).\end{aligned}$$

4. $(\xi, \mathrm{cur}(M), \alpha . S) \to (\langle \xi, \alpha \rangle, M, S)$.

$$\begin{aligned}\mathscr{T}(\xi, \mathrm{cur}(M), \alpha . S) &= \mathscr{T}'(\xi; \mathrm{cur}(M), \alpha . S) \\ &= \mathscr{T}'(((\xi; \mathrm{cur}(M))\, \alpha), S) \\ &= \mathscr{T}'(\langle \xi; \mathrm{cur}(M), \alpha \rangle; \mathrm{app}, S) \\ &= \mathscr{T}'(\langle \xi, \alpha \rangle; M, S) \\ &= \mathscr{T}(\langle \xi, \alpha \rangle, M, S).\end{aligned}$$

5. $(\xi, (MN), S) \to (\xi, M, (\xi; N) . S)$.

$$\begin{aligned}\mathscr{T}(\xi, (MN), S) &= \mathscr{T}'(\xi; (MN), S) \\ &= \mathscr{T}'(\xi; \langle M, N \rangle; \mathrm{app}, S) \\ &= \mathscr{T}'(\langle \xi; M, \xi; N \rangle; \mathrm{app}, S) \\ &= \mathscr{T}'((\xi; M)\, (\xi; N), S) \\ &= \mathscr{T}'(\xi; M, (\xi; N) . S) \\ &= \mathscr{T}(\xi, M, (\xi; N) . S). \quad \square\end{aligned}$$

What is the relevance of the previous, easy theorem? The translation of λ-expressions into categorical terms is complete. However, in general, the result of the evaluation that is a closure $(\xi; P)$ is not syntactically equal to the interpretation of any

λ-term. By unwinding the closure, that is, by resolving the code P against its environment ξ, we can find such a term (this is essentially the 'Real' function in Cousineau *et al.*, 1987, which can be rephrased in a purely categorical setting). In this way, we obtain a λ-term in weak head normal form which is βη equivalent to the term we started with. However, this simple approach based on the categorical equational theory is of no help either in proving termination, nor more generally in proving that the machine actually implements the desired evaluation strategy. If we are also interested in these aspects of the machine, it is necessary to establish a more direct correspondence between its operational rules and a *directed* categorical term rewriting system (such as the theory of categorical combinators) instead of relying on the equational theory.

There is, nevertheless, a particular but important case, where Theorem 4.2 easily applies. Namely, in case the result of the computation is a constant. We have not dealt with constants so far for the sake of simplicity, but in every real language they play a central role. As a matter of fact, they are usually the only 'outputs' we are really interested in. Adding constants to all previous machines is a straightforward task (rule 3 in Krivine's machine is already meant with this purpose). Clearly, when we get a constant c as a result of the evaluation, this is just the categorical interpretation of the corresponding λ-term. We have thus the following simple corollary:

Corollary 4.3
If $(id, C, \varnothing) \to (c, END, \varnothing)$ *where* c *is a constant, then* $C =_{\beta\eta} c$.

Proof
By Theorem 4.2, if $(\mathrm{id}, C, \varnothing) \to (c, \mathrm{END}, \varnothing)$, then $\mathscr{T}(\mathrm{id}, C, \varnothing) = \mathscr{T}(c, \mathrm{END}, \varnothing)$. Since $\mathscr{T}(\mathrm{id}, C, \varnothing) = \mathscr{T}'(\mathrm{id}; C, \varnothing) = \mathrm{id}; C = C$, and $\mathscr{T}(c, \mathrm{END}, \varnothing) = c$, it follows that $C = c$ as categorical terms. But C and c are (the categorical interpretations of the corresponding) λ-terms. Since the categorical interpretation is complete, $C =_{\beta\eta} c$. □

In the final section of this paper we shall see a stronger version of Theorem 4.2. Indeed, we shall consider there a machine performing full reduction, and every result of the machine will be the categorical interpretation of a λ-term.

PART II

This second part of the paper contains the definition of some new environment machines directly inspired by the categorical semantics. Our main aim is to show in this way the fruitfulness of the categorical approach as a simple guide to the definition of new abstract machines. We start with defining a variant of the CAM implementing a λ-calculus with both call-by-value and call-by-name as parameter passing modes (see Asperti, 1990).

5 Integrating lazy and strict evaluation

In the previous sections we have investigated two completely lazy implementations of the λ-calculus. In a real language, we may be interested in having the ability to express both lazy and strict operations. The integration of strict and lazy evaluation becomes particularly relevant in functional languages extended with algebraic or inductive types (such as are found in Miranda, Haskell, or in some dialects of LISP). Functions that are inductive over the structure of an argument are strict in that argument, while all the other functions may be lazy.

Our approach to the problem of integrating strict and lazy evaluation is close to the Algol 60 parameters passing mode. It is essentially based on the existence of two different kinds of variables: the strict and the lazy ones. By abstracting over a strict variable we get call by value (strict abstraction); in the other case we have call by name (lazy abstraction). The fact of considering two different kinds of variables instead of two kinds of lambda abstraction allows a better handling of terms with free variables. Moreover, it justifies the 'uniform' behaviour of the two forms of abstraction with respect to the application.

The implementation requires a run time test, which is performed when an actual parameter M is passed to a procedure $\lambda x.P$ in some environment ξ: if x has been declared 'lazy', the evaluation of M is frozen, saving on the environment a closure $(\xi; \text{freeze}(M))$; otherwise we start the reduction of M in ξ.

We now present the formal operational semantics of the strict-lazy-λ-calculus (λ_{sl}-calculus) in the form of an inference relation.

We have two disjoint sets of variables V_s and V_l. We 'mark' the variables with a subscript s or l according to their type. The set Λ_{sl} of all terms of the λ_{sl}-calculus over the previous sets of variables is inductively defined by the following rules:

if $x_s \in V_s$ **then** $x_s \in \Lambda_{sl}$
if $x_l \in V_l$ **then** $x_l \in \Lambda_{sl}$;
if $x_s \in V_s$ **and** $M \in \Lambda_{sl}$ **then** $\lambda x_s . M \in \Lambda_{sl}$;
if $x_l \in V_l$ **and** $M \in \Lambda_{sl}$ **then** $\lambda_{xl} . M \in \Lambda_{sl}$;
if $M \in \Lambda_{sl}$ **and** $N \in \Lambda_{sl}$ **then** $MN \in \Lambda_{sl}$.

Closed terms and substitutions are defined in the usual way.

We define now the **values** of the calculus, that is, intuitively, the possible outputs of the reduction process.

Definition 5.1
The set Val_s *of* ***strict values*** *in* Λ_{sl} *is the set of all terms of the form,* $\lambda x_s . M$, *with* $M \in \Lambda_{sl}$. *The set* Val_l *of* ***lazy values*** *in* Λ_{sl} *is the set of all terms of the form,* $\lambda x_l . M$, *with* $M \in \Lambda$. *The set* ***Val*** *of values in* Λ_{sl} *is* $Val_s \cup Val_l$.

In the following, we shall use the symbols V, V_s and V_l to range over values, strict values and lazy values, respectively. The capital letters M, N, P, Q... will represent arbitrary terms in Λ.

Consider now the following β-rules:

β-strict. $(\lambda x_s.M)V \to M[V/x_s]$;
β-lazy. $(\lambda x_l.M)N \to M[N/x_l]$.

The **reduction relation** $\to_{sl}$ is the smallest relation over Λ that contains the two β-rules above, and such that

$$\frac{M \to_{sl} M'}{MN \to_{sl} M'N},$$

$$\frac{N \to_{sl} N'}{V_s N \to_{sl} V_s N'}.$$

It is readily seen that this reduction is deterministic.

6 Denotational semantics of the λ_{sl}-calculus

The aim of this section is to provide a denotational semantics for the λ_{sl}-calculus. As in section 2, we are not interested in the semantic itself, but only as a guide to the implementation.

Let us look more closely at the problem. Our starting point is the denotational semantics of the lazy calculus in section 2. Thus, we still assume a domain A^* representing our 'lazy values', such that $A < A^*$.

Let M be an open λ_{sl}-term with free marked variables in $\Delta = \{x_1, \ldots, x_n\}$. Its categorical interpretation will be an arrow $[M]$: $t \times \mathcal{f}_1(A) \times \ldots \times \mathcal{f}_n(A) \to A$, where $\mathcal{f}_i(A) = A$ if the variable x_i is marked 'strict', and $\mathcal{f}_i(A) = A^*$ if the variable x_i is marked 'lazy'.

The only problem is in interpreting the application. Recall that the 'canonical' interpretation of section 1 is

$$[MN]_\Delta = \mathrm{eval} \circ \langle \phi \circ [M]_\Delta, [N]_\Delta \rangle, \quad \text{where eval}: A^A \times A \to A,$$

while the 'lazy interpretation' of Definition 2.4 gives

$$[\![MN]\!]_\Delta = \mathrm{eval}^* \circ \langle \Phi \circ \phi \circ [\![M]\!]_\Delta, \mathrm{freeze} \circ [\![N]\!]_\Delta \rangle, \quad \text{where eval}^*: A^{(A^*)} \times A^* \to A.$$

In both these cases, $[M]$ and $[\![M]\!]$ have type (are elements in) A, but since we know a priori the intended use of the parameter N with respect to the term M, we can decide the proper transformation in order to apply the eval map. In particular, in the first case we transform $[M]$ in $\phi \circ [M]$ of type A^A; in the second case we transform it in $\Theta \circ \phi \circ [\![M]\!]_\Delta$ of type $A^{(A^*)}$.

In case of a mixed evaluation mechanism, the previous approach does not work: we do not know if the parameter will be passed by value or by name until we look at the function, but since we have 'flattened' the function in A, we have no way to recover this information.

In order to solve this problem, we can use the notion of weak categorical sum a#b of two objects a and b (strong sums would create some problems, see below). What we need is a retraction of the form $A^A \# S^{(A^*)} < A$ (via γ and δ, say). Indeed, in this case we can respectively interpret a term of the form $\lambda x_s.M$ or $\lambda x_l.M$ in its 'proper' space, that is A^A in the first case and $A^{(A^*)}$ in the second one. By an injection and an application of γ, we can still reduce it to an element in A, but when we wish to look

at it as function, that is when we compose it with the δ arrow, we find an element in $A^A \# A^{(A^*)}$, that is, roughly, either an element in A^A or an element in $A^{(A^*)}$. Now, according to the previous cases, we can either leave the argument of this function unchanged, or freeze it.

The hypothesis $A^A \# A^{(A^*)} < A$ is not very nice. It would be more elegant to reduce it to the simpler hypothesis $A \# A < A$. Now if $A^A < A$ and $A < A^*$ we have already proved that also $A^{(A^*)} < A$. Unfortunately, retractions are not preserved by weak categorical sums, but this is 'almost' true, as is expressed by Lemma 6.2 below. It will turn out that this is enough for our purposes. We recall now the definition of weak categorical sum.

Definition 6.1

Let $\mathbf{C}$ *be a category, and* $a, b \in Ob_{\mathbf{C}}$. *The* ***weak sum*** (*weak coproduct*) *of* a *and* b *is an object* $a\#b$ *together with two morphisms* $in_a : a \to a\#b$, $in_b : b \to a\#b$ (*called injections*) *such that, for any* $f \in \mathbf{C}[a, c]$ *and* $g \in \mathbf{C}[b, c]$, *there exists a morphism* $[f, g] \in \mathbf{C}[a\#b, c]$ *such that*

(i) $[f, g] \circ in_a = f$;
(ii) $[f, g] \circ in_b = g$;
(iii) $h \circ [f, g] = [h \circ f, h \circ g]$, *for every* $h : c \to c'$.

The difference with the notion of (strong) categorical sum is that no uniqueness is demanded for the arrow $[f, g]$. In particular, we cannot prove that $[in_a, in_b] = id$ (see Hayashi; 1985, and Martini, 1988 for weak categorical concepts).

We have two reasons for working with the weak notion instead of the strong one. The first one is that weak sums are enough for our purposes. The second one, that is much more important, is that the notion of strong sum must be used very carefully when dealing with λ-models. Indeed, it is well-known that a CCC with coproducts and fixpoints is inconsistent. Now, every reflexive object A in a CCC has fixpoints, and moreover fixpoints are 'inherited' by all the objects that are retractions of A (see Asperti and Longo, 1990). Since we deal with exactly this kind of objects, we could not escape the inconsistency.

Lemma 6.2

Let $\mathbf{C}$ *be a category with weak sums and let* a, b, c, d *be objects in* $\mathbf{C}$. *If* $a < c$ (*via* f, g) *and* $b < d$ (*via* h, k), *then there exists a pair of arrows* $l : a\#b \to c\#d$ *and* $m : c\#d \to a\#b$ *such that* $m \circ l = [in_a, in_b]$. (*We will say that* $a\#b$ *is a* ***weak retract*** *of* $c\#d$).

Proof

Just take $1 = (in_c \circ f)\#(in_d \circ h)$, and $m = (in_a \circ g)\#(in_b \circ k)$. Then we have

$$\begin{aligned} m \circ 1 &= [in_a \circ g, in_b \circ k] \circ [in_c \circ f, in_d \circ h] \\ &= [[in_a \circ g, in_b \circ k] \circ in_c \circ f, [in_a \circ g, in_b \circ k] \circ in_d \circ h] \\ &= [in_a \circ g \circ f, in_b \circ k \circ h] \\ &= [in_a, in_b]. \quad \square \end{aligned}$$

Fortunately, for our model definition it is enough to have a weak retraction between $A^A \# A^{(A^*)}$ and A. This means that we just need a CCC with weak coproducts and an

object A such that $A^A < A, A < A^*$ and $A\#A < A$. Moreover, since $A\#A \cong (A \times t)\#(A \times t)$ is a weak retract of $A \times (t\#t)$, using the 'transitivity' of weak retractions and the well-known fact that in a CCC $A^A < A$ implies $A \times A < A$ (see Asperti and Longo, 1990), we can further reduce the latter assumption to $(t\#t) < A$. We have plenty of examples which satisfy this condition. The simplest one is probably the category of CPO with coalesced sum. If A is a nontrivial reflexive object of this category, then obviously $(t\#t) < A$.

It is well known that in a Cartesian Closed Category with (strong) coproducts **C**, there is an isomorphism between $(a \times c)\#(b \times c)$ and $(a\#b) \times c$ for all the objects a, b, c in **C**. Explicitly, this isomorphism is given by the following pair of arrows

$$i_1 = [in_a \times id_c, in_b \times id_c]: (a \times c)\#(b \times c) \to (a\#b) \times c \quad \text{and}$$
$$i_2 = \text{eval} \circ [\Lambda(in_{a\times c}), \Lambda(in_{b\times c})] \times id_c: (a\#b) \times c \to (a \times c)\#(b \times c)$$

If we only have weak coproducts the previous functions do not define an isomorphism any more. As a matter of fact, we can only prove that

$$i_2 \circ i_1 = [in_{a\times c}, in_{b\times c}],$$
$$i_1 \circ i_2 = [in_a, in_b] \times id_c.$$

However, this loss of information is not a big drawback. We call i_2 a **distribution morphism**, and we will use it in the following interpretation of the λ_{sl}-calculus.

Let $in_s: A^A \to A^A\#A^{(A^*)}$ and $in_1: A^{(A^*)} \to A^A\#A^{(A^*)}$ be the injections. In the following we assume that $A^A\#A^{(A^*)}$ is a weak retract of A via $\gamma: A^A\#A^{(A^*)} \to A$, $\delta: A \to A^A\#A^{(A^*)}$, i.e. $\delta \circ \gamma = [in_s, in_1]$.

Let lazy?: $A \times A^* \to (A^A \times A)\#(A^{(A^*)} \times A^*)$ be the arrow defined as follows:

$$\text{lazy?} = [in_1 \circ id \times \text{unfreeze}, in_2] \circ i_2 \circ \delta \times id$$

where $i_2: A^A\#A^{(A^*)} \times A^* \to (A^A \times A^*)\#(A^{(A^*)} \times A^*)$ is the distribution morphism, and $in_1: A^A \times A \to (A^A \times A)\#(A^{(A^*)} \times A^*), in_2: (A^{(A^*)} \times A^*) \to (A^A \times A)\#(A^{(A^*)} \times A^*)$ are the injections.

Definition 6.3
Let A and A^* as before. The ***interpretation*** *of a λ_{sl}-term M is inductively defined as follows:*

$$[\![x_i]\!]_\Delta = unfreeze \circ snd \circ fst \circ \ldots \circ fst \quad with\ n-i\ occurrences\ of\ fst,\ if\ x \in V_i$$
$$[\![x_i]\!]_\Delta = snd \circ fst \circ \ldots \circ fst \quad with\ n-i\ occurrences\ of\ fst,\ if\ x \in V_s$$
$$[\![MN]\!]_\Delta = [eval, eval^*] \circ lazy? \circ \langle [\![M]\!]_\Delta, freeze \circ [\![N]\!]_\Delta \rangle$$
$$[\![\lambda x_s . M]\!]_\Delta = \gamma \circ in_s \circ \Lambda([\![M]\!]_{\Delta \cup \{x_s\}})$$
$$[\![\lambda x_i . M]\!]_\Delta = \gamma \circ in_i \circ \Lambda([\![M]\!]_{\Delta \cup \{x_i\}}).$$

Lemma 6.4
lazy? $\circ (\gamma \circ in_s) \times id = in_1 \circ id \times unfreeze: A^A \times A^* \to (A^A \times A)\#(A^{(A^*)} \times A^*)$;
lazy? $\circ (\gamma \circ in_i) \times id = in_2: A^{(A^*)} \times A^* \to (A^A \times A)\#(A^{(A^*)} \times A^*)$.

Proof

$$\begin{aligned}
\text{lazy?} \circ (\gamma \circ in_s) \times \text{id} &= [\text{in}_1 \circ \text{id} \times \text{unfreeze}, in_2] \circ i_2 \circ \delta \times \text{id} \circ (\gamma \circ in_s) \times \text{id} \\
&= [in_1 \circ \text{id} \times \text{unfreeze}, in_2] \circ i_2 \circ (\delta \circ \gamma \circ in_s) \times \text{id} \\
&= [in_1 \circ \text{id} \times \text{unfreeze}, in_2] \circ i_2 \circ ([in_s, in_1] \circ in_s) \times \text{id} \\
&= [in_1 \circ \text{id} \times \text{unfreeze}, in_2] \circ i_2 \circ in_s \times \text{id} \\
&= [in_1 \circ \text{id} \times \text{unfreeze}, in_2] \circ \text{eval} \circ \Lambda(in_1') \times \text{id} \\
&\qquad \text{by def of } i_2, \text{ where } in_1' : A^A \times A \to (A^A \times A) \# (A^{(A^*)} \times A) \\
&= [in_1 \circ \text{id} \times \text{unfreeze}, in_2] \circ in_1' \\
&= in_1 \circ \text{id} \times \text{unfreeze}.
\end{aligned}$$

and analogously for the other one. □

Theorem 6.5

$[eval, eval^*] \circ lazy? \circ (\gamma \circ in_s) \times id = eval \circ id \times unfreeze : A^A \times A^* \to A$;

$[eval, eval^*] \circ lazy? \circ (\gamma \circ in_1) \times id = eval^* : A^{(A^*)} \times A^* \to A$.

Proof

Immediate by the previous lemma. □

In order to prove the soundness of the interpretation in Definition 6.3, we need a few substitution lemmas.

Substitution Lemma 6.6

(i) *If* $y \notin FV(N)$, *then* $[\![N]\!]_{\Delta \cup \{y\}} = [\![N]\!]_\Delta \circ fst$

(ii) $[\![M[N/x_s]]\!]_\Delta = [\![M]\!]_{\Delta \cup \{x_s\}} \circ \langle id, [\![N]\!]_\Delta \rangle$

(iii) $[\![M[N/x_i]]\!]_\Delta = [\![M]\!]_{\Delta \cup \{x_i\}} \circ \langle id, freeze \circ [\![N]\!]_\Delta \rangle$.

Proof

(i) By induction on M. The following is a typical case:

$$\begin{aligned}
[\![\lambda x_s . P]\!]_{\Delta \cup \{y\}} &= \gamma \circ in_s \circ \Lambda([\![P]\!]_{\Delta \cup \{y, x_s\}}) \\
&= \gamma \circ in_s \circ \Lambda([\![P]\!]_{\Delta \cup \{x_s, y\}} \circ \mu) \\
&\qquad \text{where } \mu \text{ is the obvious isomorphism } (A \times B) \times C \cong (A \times C) \times B \\
&= \gamma \circ in_s \circ \Lambda([\![P]\!]_{\Delta \cup \{x_s\}} \circ \text{fst} \circ \mu) \quad \text{by induction} \\
&= \gamma \circ in_s \circ \Lambda([\![P]\!]_{\Delta \cup \{x_s\}} \circ \text{fst} \times \text{id}) \quad \text{as fst} \circ \mu = \text{fst} \times \text{id} : (A \times B) \times C \to A \times C \\
&= \gamma \circ in_s \circ \Lambda([\![P]\!]_{\Delta \cup \{x_s\}}) \circ \text{fst} \quad \text{by the naturality of } \Lambda.
\end{aligned}$$

(ii) By induction on M.

$M \equiv x_s$

$$[\![x_s]\!]_{\Delta \cup \{x_s\}} \circ \langle \text{id}, [\![N]\!]_\Delta \rangle = \text{snd} \circ \langle \text{id}, [\![N]\!]_\Delta \rangle$$

$= [\![N]\!]_\Delta$

$= [\![x_s[N/x_s]]\!]_\Delta$

$M \equiv y_t$ where y_t is marked 'lazy'.

$[\![y_t]\!]_{\Delta \cup \{x_s\}} \circ \langle \mathrm{id}, [\![N]\!]_\Delta \rangle = \mathrm{unfreeze} \circ \mathrm{snd} \circ \mathrm{fst}^{n-t+1} \circ \langle \mathrm{id}, [\![N]\!]_\Delta \rangle$

$= \mathrm{unfreeze} \circ \mathrm{snd} \circ \mathrm{fst}^{n-t}$

$= [\![y_t]\!]_\Delta$

$= [\![y_t[N/x_s]]\!]_\Delta$

$M \equiv y_t$ where y_t is marked 'strict'. Analogous.

$M \equiv PQ$.

$[\![PQ]\!]_{\Delta \cup \{x_s\}} \circ \langle \mathrm{id}, [\![N]\!]_\Delta \rangle$

$= [\mathrm{eval}, \mathrm{eval}^*] \circ \mathrm{lazy}? \circ \langle [\![P]\!]_{\Delta \cup \{x_s\}}, \mathrm{freeze} \circ [\![Q]\!]_{\Delta \cup \{x_s\}} \rangle \circ \langle \mathrm{id}, [\![N]\!]_\Delta \rangle$

$= [\mathrm{eval}, \mathrm{eval}^*] \circ \mathrm{lazy}? \circ \langle [\![P]\!]_{\Delta \cup \{x_s\}} \circ \langle \mathrm{id}, [\![N]\!]_\Delta \rangle, \mathrm{freeze} \circ \langle Q \rangle\!]_{\Delta \cup \{x_s\}} \circ \langle \mathrm{id}, [\![N]\!]_\Delta \rangle \rangle$

$= [\mathrm{eval}, \mathrm{eval}^*] \circ \mathrm{lazy}? \circ \langle [\![P[N/x_s]]\!]_\Delta, \mathrm{freeze} \circ [\![Q[N/x_s]]\!]_\Delta \rangle$

$= [\![(PQ)[N/x_s]]\!]_{\Delta \cup \{x_s\}}$.

$M \equiv \lambda y_s . P$

$[\![(\lambda y_s . P)[N/x_s]]\!]_\Delta = \gamma \circ in_s \circ \Lambda([\![P[N/x_s]]\!]_\Delta \cup \{y_s\})$

$= \gamma \circ in_s \circ \Lambda([\![P]\!]_{\Delta \cup \{y_s, x_s\}} \circ \langle \mathrm{id}, [\![N]\!]_{\Delta \cup \{y_s\}} \rangle)$

$= \gamma \circ in_s \circ \Lambda([\![P]\!]_{\Delta \cup \{y_s, x_s\}} \circ \langle \mathrm{id}, [\![N]\!]_\Delta \circ \mathrm{fst} \rangle)$ by (i)

$= \gamma \circ in_s \circ \Lambda([\![P]\!]_{\Delta \cup \{x_s, y_s\}}) \circ \langle \mathrm{id}, [\![N]\!]_\Delta \rangle$ by naturality of Λ

$= [\![(\lambda y_s . P)]\!]_{\Delta \cup \{x_s\}} \circ \langle \mathrm{id}, [\![N]\!]_\Delta \rangle$.

$M \equiv \lambda y_t . P$. Analogous.

(iii) By induction on M.

$M \equiv x_t$

$[\![x_t]\!]_{\Delta \cup \{x_t\}} \circ \langle \mathrm{id}, \mathrm{freeze} \circ [\![N]\!]_\Delta \rangle = \mathrm{unfreeze} \circ \mathrm{snd} \circ \langle \mathrm{id}, \mathrm{freeze} \circ [\![N]\!]_\Delta \rangle$

$= \mathrm{unfreeze} \circ \mathrm{freeze} \circ [\![N]\!]_\Delta$

$= [\![N]\!]_\Delta$

$= [\![x_s[N/x_s]]\!]_\Delta$

$M \equiv x_t$ where x_t is marked 'lazy'

$[\![x_t]\!]_{\Delta \cup \{x_s\}} \circ \langle \mathrm{id}, \mathrm{freeze} \circ [\![N]\!]_\Delta \rangle = \mathrm{unfreeze} \circ \mathrm{snd} \circ \mathrm{fst}^{n-t+1} \circ \langle \mathrm{id}, \mathrm{freeze} \circ [\![N]\!]_\Delta \rangle$

$= \mathrm{unfreeze} \circ \mathrm{snd} \circ \mathrm{fst}^{n-t}$

$= [\![x_t]\!]_\Delta$

$= [\![x_t[N/x_s]]\!]_\Delta$

$M \equiv x_t$ where x_t is marked 'strict'. Analogous.

$M \equiv PQ$.

$[\![PQ]\!]_{\Delta \cup \{x_t\}} \circ \langle \mathrm{id}, \mathrm{freeze} \circ [\![N]\!]_\Delta \rangle$

$= [\mathrm{eval}, \mathrm{eval}^*] \circ \mathrm{lazy}? \circ \langle [\![P]\!]_{\Delta \cup \{x_t\}}, \mathrm{freeze} \circ [\![Q]\!]_{\Delta \cup \{x_t\}} \rangle \circ \langle \mathrm{id}, \mathrm{freeze} \circ [\![N]\!]_\Delta \rangle$

$= [\text{eval},\text{eval}^*] \circ \text{lazy}? \circ \langle [\![P]\!]_{\Delta \cup \{x_l\}} \circ \langle \text{id}, \text{freeze} \circ [\![N]\!]_\Delta,$

$\text{freeze} \circ [\![Q]\!]_{\Delta \cup \{x_l\}} \circ \langle \text{id}, \text{freeze} \circ [\![N]\!]_\Delta \rangle$

$= [\text{eval},\text{eval}^*] \circ \text{lazy}? \circ \langle [\![P[N/x_l]]\!]_\Delta, \text{freeze} \circ [\![Q[N/x_l]]\!]_\Delta \rangle$

$= [\![(PQ)[N/x_l]]\!]_{\Delta \cup \{x_l\}}$.

$M \equiv \lambda y_s . P$

$[\![(\lambda y_s . P)[N/x_l]]\!]_\Delta = \gamma \circ in_s \circ \Lambda([\![P[N/x_s]]\!]_{\Delta \cup \{y_s\}})$

$= \gamma \circ in_s \circ \Lambda([\![P]\!]_{\Delta \cup \{y_s, x\}} \circ \langle \text{id}, \text{freeze} \circ [\![N]\!]_{\Delta \cup \{y_s\}} \rangle)$

$= \gamma \circ in_s \circ \Lambda([\![P]\!]_{\Delta \cup \{y_s, x_l\}} \circ \langle \text{id}, \text{freeze} \circ [\![N]\!]_\Delta \circ \text{fst} \rangle)$ by (i)

$= \gamma \circ in_s \circ \Lambda([\![P]\!]_{\Delta \cup \{x_l, y_s\}}) \circ \langle \text{id}, \text{freeze} \circ [\![N]\!]_\Delta \rangle$ by naturality of Λ

$= [\![(\lambda y_s . P)]\!]_{\Delta \cup \{x_s\}} \circ \langle \text{id}, \text{freeze} \circ [\![N]\!]_\Delta \rangle$.

$M \equiv \lambda y_l . P$. Analogous. □

The previous substitution lemma combines with Theorem 6.5 to give semantics to the β-rule.

Theorem 6.7

(i) $[\![(\lambda x_s . M)V]\!]_\Delta = [\![M[V/x_s]]\!]_\Delta$;

(ii) $[\![(\lambda x_l . M)N]\!]_\Delta = [\![M[N/x_s]]\!]_\Delta$;

(iii) $[\![M]\!]_\Delta = [\![M']\!]_\Delta$ *implies* $[\![MN]\!]_\Delta = [\![M'N]\!]_\Delta$;

(iv) $[\![N]\!]_\Delta = [\![N']\!]_\Delta$ *implies* $[\![V_s N]\!]_\Delta = [\![V_s N']\!]_\Delta$

where Δ *is a set of variables that contains all the free variables of the involved terms.*

Proof

(i) $[\![(\lambda x_s . M)V]\!]_\Delta = [\text{eval},\text{eval}^*] \circ \text{lazy}? \circ \langle [\![\lambda x_s . M]\!]_\Delta, \text{freeze} \circ [\![V]\!]_\Delta \rangle$

$= [\text{eval},\text{eval}^*] \circ \text{lazy}? \circ \langle \gamma \circ in_s \circ \Lambda([\![M]\!]_{\Delta \cup \{x_s\}}), \text{freeze} \circ [\![V]\!]_\Delta \rangle$

$= [\text{eval},\text{eval}^*] \circ \text{lazy}? \circ (\gamma \circ in_s) \times \text{id} \circ \text{id} \times \text{freeze} \circ \langle \Lambda([\![M]\!]_{\Delta \cup \{x_s\}}), [\![V]\!]_\Delta \rangle$

$= \text{eval} \circ \text{id} \times \text{unfreeze} \circ \text{id} \times \text{freeze} \circ \langle \Lambda([\![M]\!]_{\Delta \cup \{x_s\}}), [\![V]\!]_\Delta \rangle$ by Theorem 6.5

$= \text{eval} \circ \langle \Lambda([\![M]\!]_{\Delta \cup \{x_s\}}), [\![V]\!]_\Delta \rangle$

$= \text{eval} \circ \Lambda([\![M]\!]_{\Delta \cup \{x_s\}}) \times \text{id} \circ \langle \text{id}, [\![V]\!]_\Delta \rangle$

$= [\![M]\!]_{\Delta \cup \{x_s\}} \circ \langle \text{id}, [\![V]\!]_\Delta \rangle$

$= [\![M[V/x_s]]\!]_\Delta$ by the substitution lemma.

(ii) $[\![(\lambda x_l . M)N]\!]_\Delta = [\text{eval},\text{eval}^*] \circ \text{lazy}? \circ \langle [\![\lambda x_l . M]\!]_\Delta, \text{freeze} \circ [\![N]\!]_\Delta \rangle$

$= [\text{eval},\text{eval}^*] \circ \text{lazy}? \circ \langle \gamma \circ in_l \circ \Lambda([\![M]\!]_{\Delta \cup \{x_l\}}), \text{freeze} \circ [\![N]\!]_\Delta \rangle$

$= [\text{eval},\text{eval}^*] \circ \text{lazy}? \circ (\gamma \circ in_l) \times \text{id} \circ \langle \Lambda([\![M]\!]_{\Delta \cup \{x_l\}}), \text{freeze} \circ [\![N]\!]_\Delta \rangle$

$= \text{eval}^* \circ \langle \Lambda([\![M]\!]_{\Delta \cup \{x_l\}}), \text{freeze} \circ [\![N]\!]_\Delta \rangle$ by Theorem 6.5

$= \text{eval}^* \circ \Lambda([\![M]\!]_{\Delta \cup \{x_l\}}) \times \text{id} \circ \langle \text{id}, \text{freeze} \circ [\![N]\!]_\Delta \rangle$

$= [\![M]\!]_{\Delta \cup \{x_l\}} \circ \langle \text{id}, \text{freeze} \circ [\![N]\!]_\Delta \rangle$

$= [\![M[N/x_l]]\!]_\Delta$ by the substitution lemma.

(iii) and (iv) are immediate, by the interpretation of the application. □

Corollary 6.8

Let P *be a* λ_{sl}*-term with free variables in* Δ. *If* $P \to_{sl} Q$ *in one step, then* $[\![P]\!]_\Delta = [\![Q]\!]_\Delta$.

Proof
By induction on the number k of the inference rules

$$\frac{M \to_{ls} M'}{MN \to_{ls} M'N} \qquad \frac{N \to_{ls} N'}{V_s N \to_{ls} V_s N'}.$$

used in the one step derivation $P \to_{sl} Q$, using the previous theorem. □

Corollary 6.9 (soundness)
Let P be a λ_{sl}-term with free variables in Δ. If $P \to_{sl} Q$, then $[\![P]\!]_\Delta = [\![Q]\!]_\Delta$.

Proof
By induction on the length of the derivation, using the previous corollary. □

7 The CAM with lazy abstraction

The CAM_l is a variation of the Categorical Abstract Machine implementing the λ_{sl}-calculus. We shall derive this original machine from the denotational semantics of Section 6. The following exposition may look a bit tricky, but the reader should keep in mind three important facts, namely that the categorical semantics of the λ_{sl}-calculus is much more entangled than that of the traditional λ-calculus, that the compilation in linear code, by itself, compromises to some extent a neat explanation of the categorical 'soundness' of the machine, and finally that we wish to get, insofar as it seems possible, an efficient implementation.

For ease of reference we recall here the interpretation of a λ_{sl}-term M given in Definition 6.3:

$[\![x_i]\!]_\Delta = \text{unfreeze} \circ \text{snd} \circ \text{fst} \circ \ldots \circ \text{fst}$ with $n-1$ occurrences of fst, if $x \in V_l$
$[\![x_i]\!]_\Delta = \text{snd} \circ \text{fst} \circ \ldots \circ \text{fst}$ with $n-1$ occurrences of fst, if $x \in V_s$
$[\![MN]\!]_\Delta = [\text{eval},\text{eval}^*] \circ \text{lazy}? \circ \langle [\![M]\!]_{\downarrow\Delta}, \text{freeze} \circ [\![N]\!]_\Delta \rangle$
$[\![\lambda x_s . M]\!]_\Delta = \gamma \circ in_s \circ \Lambda([\![M]\!]_{\Delta \cup \{x_s\}})$
$[\![\lambda x_l . M]\!]_\Delta = \gamma \circ in_l \circ \Lambda([\![M]\!]_{\Delta \cup \{x_l\}})$.

As in Section 2, we take the combinators unfreeze and freeze(_). The other combinators are $\text{cur}_s(\_)$, $\text{cur}_l(\_)$, and lazy_app?, with the following intended semantics:

$\text{cur}_s(f) = \gamma \circ in_s \circ \Lambda(f) : B \to A$, where $f : B \times A \to A$
$\text{cur}_l(g) = \gamma \circ in_l \circ \Lambda(g) : B \to A$ where $g : B \times A^* \to A$
lazy_app? = [eval,eval*] ∘ lazy?

With these combinators, and the usual combinators for handling the stack, the previous interpretation becomes:

$[\![x_i]\!]_\Delta = \text{fst}; \ldots; \text{fst}; \text{sand}; \text{unfreeze}$ with $n-1$ occurrences of fst, if $x \in V_l$
$[\![x_i]\!]_\Delta = \text{fst}; \ldots; \text{fst}; \text{snd}$ with $n-1$ occurrences of fst, if $x \in V_s$

$[\![MN]\!]_\Delta$ = push; $[\![M]\!]_\Delta$; swap; freeze($[\![N]\!]_\Delta$); cons; lazy_app?

$[\![\lambda x_s . M]\!]_\Delta = \mathrm{cur}_s([\![M]\!]_{\Delta \cup \{x_s\}})$

$[\![\lambda x_l . M]\!]_\Delta = \mathrm{cur}_l([\![M]\!]_{\Delta \cup \{x_l\}})$.

The only combinator requiring attention is 'lazy_app?' (all the other combinators have the usual behaviour; in particular for the two cur(_) combinators we use the associative law of composition). Now, when we meet a lazy_app? as next instruction, the current environment can be either in the state $\langle \xi_1; \mathrm{cur}_s(C_1), \xi_2; \mathrm{freeze}(C_2)\rangle$ or in the state $\langle \xi_1; \mathrm{cur}_l(C_1), \xi_2; \mathrm{freeze}(C_2)\rangle$. In both cases we have a pair formed by a closure and a frozen expression; the only difference is in the kind of the cur combinator in the closure.

Category theory gives the following equations

$$\begin{aligned}
&\langle \xi_1; \mathrm{cur}_s(C_1), \xi_2; \mathrm{freeze}(C_2)\rangle; \mathrm{lazy_app?} \\
&\quad = [\mathrm{eval}, \mathrm{eval}^*] \circ \mathrm{lazy?} \circ \langle \xi_1; \mathrm{cur}_s(C_1), \xi_2; \mathrm{freeze}(C_2)\rangle \\
&\quad = [\mathrm{eval}, \mathrm{eval}^*] \circ \mathrm{lazy?} \circ \langle \gamma \circ in_s \circ \Lambda(C_1) \circ \xi_1, \mathrm{freeze} \circ C_2 \circ \xi_2\rangle \\
&\quad = [\mathrm{eval}, \mathrm{eval}^*] \circ \mathrm{lazy?} \circ (\gamma \circ in_s) \times \mathrm{freeze} \circ \langle \Lambda(C_1) \circ \xi_1, C_2 \circ \xi_2\rangle \\
&\quad = \mathrm{eval} \circ \langle \Lambda(C_1) \circ \xi_1, C_2 \circ \xi_2\rangle \quad \text{by Theorem 6.5} \\
&\quad = C_1 \circ \langle \xi_1, C_2 \circ \xi_2\rangle \\
&\quad = \langle \xi_1, C_2 \circ \xi_2\rangle; C_1
\end{aligned}$$

and analogously,

$$\langle \xi_1; \mathrm{cur}_l(C_1), \xi_2; \mathrm{freeze}(C_2)\rangle; \mathrm{lazy\_app?} = \langle \xi_1, \xi_2; \mathrm{freeze}(C_2)\rangle; C_1.$$

The last equation can be immediately adopted as a rewriting rule, but the previous one is not, since we must first independently perform the reduction of the expression $C_2 \circ \xi_2$. In other words, we have the conditional rule

$$\frac{\xi_2; C_2 \Rightarrow M}{\langle \xi_1; \mathrm{cur}_s(C_1), \xi_2; \mathrm{freeze}(C_2)\rangle; (\mathrm{lazy\_app?}; C_3) \Rightarrow \langle \xi_1, M\rangle; C_3},$$

where M is in normal form.

Working with a stack we should save the closure $\xi_1; \mathrm{cur}_s(C_1)$, start the evaluation of $\xi_2; C_2$ and then build the new pair between the current environment (i.e. the result of the evaluation) and the head of the stack. Note now that $\xi_1; \mathrm{cur}_s(C_1)$ was already on the stack just before executing the last cons operation which eventually preceded lazy_app. This suggests that we can anticipate the test before the cons. As a matter of fact, the test can be encapsulate in freeze(_), yielding the new combinator freeze?(_).

Definition 7.1

The ***compilation*** *by means of categorical combinators of a λ_{sl}-term M in a 'dummy' environment Δ of 'marked' variables is inductively defined as follows:*

$$\mathscr{C}am_t(x_s)_{(\Delta x_s)} = snd$$

$$\mathscr{C}am_t(x_l)_{(\Delta x_l)} = snd; unfreeze$$
$$\mathscr{C}am_t(y)_{(\Delta, x)} = fst; \mathscr{C}am_t(y)_\Delta$$
$$\mathscr{C}am_t(\lambda x_s . M)_\Delta = cur_s(\mathscr{C}am_t(M)_{(\Delta, x_s)})$$
$$\mathscr{C}am_t(\lambda_l x . M)_\Delta = cur_l(\mathscr{C}am_t(M)_{(\Delta, x_l)})$$
$$\mathscr{C}am_t(MN)_\Delta = push; \mathscr{C}am_t(M)_\Delta; swap; freeze?\ (\mathscr{C}am_t(N)_\Delta); cons; app.$$

Definition 7.2
FCI The ***reduction*** *of the compiled code is summarized in table* 4:

Table 4

Before			After		
Environment	Code	Stack	Environment	Code	Stack
$\langle \alpha, \beta \rangle$	fst; C	S	α	C	S
$\langle \alpha, \beta \rangle$	snd; C	S	β	C	S
ξ	$cur_s(C_1)$; C_2	S	ξ; $cur_s(C_1)$	C_2	S
ξ	$cur_l(C_1)$; C_2	S	ξ; $cur_l(C_1)$	C_2	S
$\langle \xi$; cur$(C_1), \alpha \rangle$	app; C_2	S	$\langle \xi, \alpha \rangle$	C_1; C_2	S
ξ	push; C	S	ξ	C	$\xi . S$
ξ_1	swap; C	$\xi_2 . S$	ξ_2	C	$\xi_1 . S$
ξ_1	cons; C	$\xi_2 . S$	$\langle \xi_2, \xi_1 \rangle$	C	S
ξ_1	freeze?(C_1); C_2	ξ_2; $cur_s(C_1) . S$	ξ_1	C_1; C_2	ξ_2; cur$(C_1) . S$
ξ_1	freeze?(C_1); C_2	ξ_2; $cur_l(C_1) . S$	ξ_1; freeze(C_1)	C_2	ξ_2; cur$(C_1) . S$
ξ; freeze(C_1)	unfreeze; C_2	S	ξ	C_1; C_2	S

Thus, the freeze? operation requires a test on the top of the stack. Note, moreover, that after this test we can actually forget whether the cur operation was lazy or strict.

The correctness of the previous machine has been proved by Asperti (1990). Note, however, that our proof is not based on the categorical semantics of the λ_{sl}-calculus, but it relies on the correctness of the CAM, passing through an embedding of the λ_{sl}-calculus into the call-by-value λ-calculus, and an operational comparison of the two abstract machines.

PART III

The final sections of this paper are devoted to the presentation of two original categorical machines for strong reduction of λ-terms (i.e. up to the normal form). Strong reduction strategies have recently been the object of several studies (see Crégut, 1990; Abadi *et al.*, 1990; Field, 1990). Their need arises in different fields of computer science, from optimization theory (partial evaluation), to type checking in higher order functional languages, and higher order unification in logic programming.

The categorical nature of the following machines is even more evident than in the cases we have discussed so far. In particular, since the result of the evaluation is a λ-term, and not just a closure, the categorical 'soundness' of the machine immediately yields a 'correctness' result, showing the full power of this approach.

8 Passing environments across a λ

The result of evaluating a weak head normal form is a closure of the kind $(\xi; \mathrm{cur}(C))$. The main problem in reducing the term to its full normal form is that of passing the environment (the substitution) ξ across the λ-abstraction. Category theory suggests the following rewriting:

$$(*) \quad (\xi; \mathrm{cur}(C)) \to \mathrm{cur}(\xi \times \mathrm{id}; C) = \mathrm{cur}(\langle \mathrm{fst}; \xi, \mathrm{snd} \rangle; C).$$

In other words, when we try to reduce a term of the form $(\xi; \mathrm{cur}(C))$, we can immediately return a 'cur' as a partial result of the evaluation, and start a subcomputation of C in the environment $\langle \mathrm{fst}; \xi, \mathrm{snd} \rangle$. Unfortunately, things are not so simple: as a matter of fact, an essential feature of the categorical abstract machines considered so far is that an environment is a term in a suitable normal form, while, in general, $\langle \mathrm{fst}; \xi, \mathrm{snd} \rangle$ is not. Normalizing $\mathrm{fst}; \xi$ is very expensive, since it would essentially imply a complete physical replication of ξ; moreover, this work can be completely useless; take for instance the case C is a closed term. There is, nevertheless, a simple solution to this problem, that is to proceed in the reduction of $\langle \mathrm{fst}; \xi, \mathrm{snd} \rangle$ in a sort of 'lazy' way. To explain our approach, it is convenient to introduce some terminology. In particular, adopting a notation close to that of Abadi *et al.* (1990); we shall write $\uparrow^1(\xi)$ instead of $\mathrm{fst}; \xi$, and, more generally, $\uparrow^n(\xi)$ for $\mathrm{fst}; \ldots \mathrm{fst}; \xi$, with n occurrences of fst. $\uparrow^0$ should be thus understood as an identity. We call $\uparrow^n$ a **shift combinator**, and $\uparrow^n(\xi)$ is a shifted environment. Shifted environments are recursively defined by the following grammar:

$$\xi ::= \uparrow^n(\mathrm{id}), \quad \uparrow^n \langle \xi, \uparrow^m(0) \rangle, \quad \uparrow^n \langle \xi, (\xi'; C) \rangle.$$

where n and m can be any integer. 0 is a De Bruijn index (it stands for the categorical combinator 'snd'). $(\xi'; C)$ is a closure. C is a piece of code; its syntactical category usually depends on the particular machine. For instance, in our first machine it will just be a λ-term, but in the second machine we consider, based on the computation of head normal forms, we will have closures as instructions, requiring a recursive definition between code and environment.

In a real implementation, a shifted environment $\uparrow^n(\xi)$ can be simply represented by an integer n and a pointer to ξ.

The main difference between our approach and that of Abadi *et al.* (1990) can be roughly explained as follows: while they were interested in the definition of a general framework for studying the computational process, we are more concerned with concrete abstract machines. For this reason, we shall try to avoid all the rules concerning a mere symbolic manipulation of environments, since they never occur in practice (or better, they are always guided by the operations on the code). Nevertheless, in this way we shall get a formal system whose interest is not merely practical, but that can be considered as an interesting surrogate for Curien's combinatorial system, and even for the calculus of explicit substitutions (Abadi *et al.*, 1990).

Working with shifted environments, the rewriting rule (*) is now translated as follows:

$$(\uparrow^n(\xi); \mathrm{cur}(C)) \to \mathrm{cur}(\uparrow^0 \langle \uparrow^{n+1}(\xi), \uparrow^0(0) \rangle; C).$$

'Laziness' comes in when we must evaluate a De Bruijn index m in a shifted environment $\uparrow^n(\xi)$. In the case $m > 0$, we have the following rule, whose semantical soundness is obvious:

$$(\uparrow^n \langle \uparrow^p(\xi'), \chi \rangle ; m+1) \to (\uparrow^{n+p}(\xi'); m).$$

If $m = 0$, two cases are possible: either we are pointing to a closure $(\uparrow^q(\xi''); C)$, and we open it with the rule

$$(\uparrow^n \langle \uparrow^p(\xi'), (\uparrow^q(\xi''); C) \rangle ; 0) \to (\uparrow^{n+q}(\xi''); C)$$

or we are pointing to a 'global variable' $\uparrow^q(0)$, that stops the evaluation:

$$(\uparrow^n \langle \uparrow^p(\xi'), \uparrow^q(0) \rangle ; 0) \to n+q.$$

Starting from the previous considerations and following the main structure of Krivine's machine, we define now a first abstract machine that can be considered as the categorical counterpart of Crégut's KN machine (Crégut, 1990). The state of the machine is a triple $(\uparrow^n(\xi), C, S)$, where ξ is a shifted environment, C is the current code, and S is a stack. The machine is defined by the following (conditional) rewriting rules:

1. $(\uparrow^n(\xi), (C'C''), S \to (\uparrow^n(\xi), C', (\uparrow^n(\xi); C'').S)$
2. $(\uparrow^n(\xi'), \mathrm{cur}(C'), \chi.S) \to (\uparrow^0 \langle \uparrow^n(\xi'), \chi \rangle, C', S)$
3. $(\uparrow^n \langle \uparrow^p(\xi'), \chi \rangle, m+1, S) \to (\uparrow^{n+p}(\xi'), m, S)$
4. $(\uparrow^n \langle \uparrow^p(\xi'), (\uparrow^q(\xi''); C) \rangle, 0, S) \to (\uparrow^{n+q}(\xi''), C, S)$
5. $(\uparrow^n \langle \uparrow^p(\xi), \uparrow^q(0) \rangle, 0, \varnothing) \to (n+q), \mathrm{end}, \varnothing)$
6. $$\frac{(\uparrow^{n_t}(\xi_t), C_t, \varnothing) \to (N_t, \mathrm{end}, \varnothing)}{(\uparrow^n \langle \uparrow^p(\xi'), \uparrow^q(0) \rangle, 0, (\uparrow^{n_1}(\xi_1), C_1). \ldots .(\uparrow^{n_r}(\xi_r), C_r)) \to (n+q\, N_1 \ldots N_r, \mathrm{end}, \varnothing)}$$
7. $$\frac{(\uparrow^0 \langle \uparrow^{n+1}(\xi), \uparrow^0(0) \rangle, C, \varnothing) \to (N, \mathrm{end}, \varnothing)}{(\uparrow^n(\xi), \mathrm{cur}(C),) \to (\mathrm{cur}(N), \mathrm{end}, \varnothing)}$$

Rule 5 can be regarded as a particular case of rule 6. The normalization of C starts in the configuration $(\uparrow^0(\mathrm{id}), C, \varnothing)$.

9 Computing head normal forms

The previous machine was based on a leftmost-redex strategy. Our next machine implements normal parallel reduction with left call by value (see Lévy, 1978, p. 193). The main advantage of this machine is that it allows us to share the reduction of different instances of a same variable.

Our first goal is the definition of a machine computing the head normal form of a term, if it exists. For this purpose, we must define a suitable representation of the unevaluated part of the head normal form. Now, in a categorical framework, closures have the same status as terms, and thus we can soundly consider them as part of the code. That is, closures will not only represent (substitution for) variables in the environment, but also unreduced parts of the result. From this point of view, there is still a point that must be clarified: in the previous machine we have an evident

notational redundancy, since a De Bruijn index n is exactly the same as $\uparrow^n(0)$. This was not of great relevance so far, since we still had a clear distinction between code and environment, but now that we are going to mix the two notions it is convenient to adopt the latter as the unique, canonical representation.

The main differences with the previous machine are the following ones:

(a) when we evaluate $\uparrow^0(0)$ in an environment $\uparrow^n\langle\uparrow^p(\xi),\chi\rangle$, we no longer have to make a distinction between the case in which χ is a closure or it is a (global) variable. Since we can now handle closures as code instructions, we just proceed to evaluate χ in an environment $\uparrow^n(\mathrm{id})$. Again, note the 'local' soundness of this rule; the categorical term $\uparrow^n\langle\uparrow^p(\xi),\chi\rangle;\ \uparrow^0(0)$ is equal to $\uparrow^n(\mathrm{id});\chi$. Note also that we are using id in a polymorphic fashion: its arity actually depends on the number of free variables accessible from χ (that is, on the arity of the domain of χ).

(b) the distinction between local and global variables depends on the structure of the environment. When we evaluate $\uparrow^m(0)$ with $m>0$ in an environment $\uparrow^n\langle\uparrow^p(\xi),\chi\rangle$ the variable should be understood as local, and we compute $\uparrow^{m-1}(0)$ in the environment $\uparrow^{n+p}(\xi)$; if, on the contrary, the environment was of the kind $\uparrow^n(\mathrm{id})$, the variable is global, at this stage of the computation; in particular, it is the head variable of our term, and we return as result $\uparrow^{n+m}(0)$ followed by the list of its unevaluated arguments, that are just the closures in the stack.

(c) when we meet a closure $\uparrow^m(\xi);C$ as a code instruction, we start a subcomputation for reducing the closure to its head normal form N. On returning from this subroutine call we restart the previous computation with the same environment and stack, and with N as new code. This execution mode allows us to share the reduction of $\uparrow^m(\xi);C$ by introducing a level of indirection on closures.

Here is the abstract machine:

1. $(\uparrow^n(\xi),(C'C''),S)\to(\uparrow^n(\xi),C',(\uparrow^n(\xi);C'').S)$
2. $$\frac{(\uparrow^n(\xi),\mathrm{cur}(C),\varnothing)\to(\mathrm{cur}(C),\mathrm{end},\varnothing)}{(\uparrow^n(\xi),\mathrm{cur}(C),\chi.S)\to(\uparrow^0\langle\uparrow^0(\mathrm{id}),\chi\rangle,M,S)}$$
3. $(\uparrow^n(\mathrm{id}),\uparrow^m(0),\chi_1.\ \cdots\ .\chi_r)\to(\uparrow^{m+n}(0)\chi_1\cdots\chi_r,\mathrm{end},\varnothing)$
4. $(\uparrow^n\langle\uparrow^p(\xi),\chi\rangle,\uparrow^{m+1}(0),S)\to(\uparrow^{n+p}(\xi),\uparrow^m(0),S)$
5. $(\uparrow^n\langle\uparrow^p(\xi'),\chi\rangle,\uparrow^0(0),S)\to(\uparrow^n(\mathrm{id}),\chi,S)$
6. $$\frac{(\uparrow^m(\xi''),C,\varnothing)\to(N,\mathrm{end},\varnothing)}{(\uparrow^n(\xi'),(\uparrow^m(\xi'');C),S)\to(\uparrow^n(\xi'),N,S)}$$
7. $$\frac{(\uparrow^0\langle\uparrow^{n+1}(\xi),\uparrow^0(0)\rangle,C,\varnothing)\to(N,\mathrm{end},\varnothing)}{(\to^n(\xi),\mathrm{cur}(C),\varnothing)\to(\mathrm{cur}(N),\mathrm{end},\varnothing)}$$

The initial configuration is still $(\uparrow^0(\mathrm{id}),C,\varnothing)$.

Following Crégut (1990), the second rule above may be replaced by the simpler, and more practical rule

2'. $(\uparrow^n(\xi),\mathrm{cur}(C'),\chi.S)\to(\uparrow^0\langle\uparrow^n(\xi),\chi\rangle,C',S)$.

Note that this machine no longer implements normal parallel reduction with left call by value, as Crégut affirms, but is an interesting compromise between that strategy and normal parallel reduction.

The next step is that of introducing a new mechanism to get the full normal form of the term, by unwinding closures. As is discussed by Crégut (1990), we can add strategies at this level to decide whether to pursue the reduction or not. This part of the machine is thus a sort of controller, recursively calling the previous machine on each closure. Moreover, since it is completely independent from the way we compute the head normal form, every controller for Crégut's machine works for our machine as well. Here is a simple example of a controller, borrowed from Crégut (1990):

- NF is a function that takes a partially evaluated term and unwinds closures;
- N takes a term and gives back its normal form, if it exists:

$$NF(\uparrow^m(0)) \to \uparrow^m(0),$$

$$\frac{NF(C) \to C'}{NF(\mathrm{cur}(C)) \to \mathrm{cur}(C')},$$

$$\frac{NF(A) \to A' \quad NF(B) \to B'}{NF(AB) \to A'B'},$$

$$\frac{(\uparrow^m(\xi), C, \varnothing) \to (A, \mathrm{end}, \varnothing) \quad NF(A) \to A'}{NF(\uparrow^m(\xi); C) \to A'},$$

$$\frac{NF(\uparrow^0(\mathrm{id}); C) \to A}{N(C) \to A}.$$

10 Correctness of the machine

In this section we shall study the correctness of our second machine (the following approach applies as well to the machine in section 8).

We start by defining a translation function $\mathscr{T}$ from machine states to categorical terms that, for the categorical nature of the abstract machine, will be particularly simple. In particular, the formal definition of $\mathscr{T}$ is just the same as for Krivine's machine in section 4.

We consider $\uparrow^m(\xi)$ as an abbreviation for $\mathrm{fst}; \ldots; \mathrm{fst}; \xi$ with m occurrences of fst, and (MN) as an abbreviation for $\langle M, N\rangle; \mathrm{app}$, where $\mathrm{app} = \Lambda^{-1}(\phi): A \times A \to A$ as in section 1. Note that the previous categorical 'expansion' of a λ-term M (in De Bruijn notation) is nothing but the categorical interpretation of M. In the following, we shall make no further distinction between a λ-term and its categorical expansion (interpretation); in other words, we shall designate as λ-terms all the categorical terms that are *syntactically* equal to the interpretation of λ-terms.

$\mathscr{T}$ is then defined as follows

$$\mathscr{T}(\xi, \mathrm{end}, \varnothing) = \xi$$
$$\mathscr{T}(\xi, C, S) = \mathscr{T}'((\xi; C), S),$$

where

$$\mathscr{T}'(M, \varnothing) = M,$$
$$\mathscr{T}'(M, N.S) = \mathscr{T}'((MN), S).$$

By the definition of $\mathscr{T}$, it is evident that if $\sigma = (\xi, C, \chi_1, \ldots, \chi_r)$ and $\sigma' = (\xi', C', \chi_1', \ldots, \chi_r')$ are two states such that , $\xi = \xi'$, $C = C'$ and, for any i, $\chi_i = \chi_i'$, then $\mathscr{T}(\sigma) = \mathscr{T}(\sigma')$ (where the previous equalities are meant between categorical terms). A similar result holds for $\mathscr{T}'$.

Theorem 10.1 (soundness)
If $(\chi, C, S) \to (\xi', C', S')$, *then* $\mathscr{T}(\xi, C, S) = \mathscr{T}(\xi', C', S')$.

Proof

1. $(\uparrow^n(\xi), (C'C''), S) \to (\uparrow^n(\xi), C', (\uparrow^n(\xi); C'').S)$.

$$\begin{aligned}\mathscr{T}(\uparrow^n(\xi), (C'C''), S) &= \mathscr{T}'((\uparrow^n(\xi); (C'C'')), S)\\ &= \mathscr{T}'(((\uparrow^n(\xi); C')(\uparrow^n(\xi); C'')), S) \quad \text{(naturality of } \langle , \rangle\\ &= \mathscr{T}'((\uparrow^n(\xi); C'), (\uparrow^n(\xi); C'').S)\\ &= \mathscr{T}(\uparrow^n(\xi), C', (\uparrow^n(\xi); C'').S).\end{aligned}$$

2. $$\frac{(\uparrow^n(\xi), \mathrm{cur}(C), \varnothing) \to (\mathrm{cur}(M), \mathrm{end}, \varnothing)}{(\uparrow^n(\xi), \mathrm{cur}(C), \chi.S) \to (\uparrow^0\langle\uparrow^0(\mathrm{id}), \chi\rangle, M, S)}.$$

By hypothesis, $\mathscr{T}(\uparrow^n(\xi),\ \mathrm{cur}(C),\ \varnothing) = \mathscr{T}'(\uparrow^n(\xi);\ \mathrm{cur}(C),\ \varnothing) = \uparrow^n(\xi);\ \mathrm{cur}(C) = \mathscr{T}(\mathrm{cur}(M),\ \mathrm{end},\ \varnothing) = \mathrm{cur}(M)$.

Then

$$\begin{aligned}\mathscr{T}(\uparrow^n(\xi), \mathrm{cur}(C), \chi.S) &= \mathscr{T}'((\uparrow^n(\xi); \mathrm{cur}(C)), \chi.S)\\ &= \mathscr{T}'(\mathrm{cur}(M), \chi.S)\\ &= \mathscr{T}'((\mathrm{cur}(M)\,\chi), S)\\ &= \mathscr{T}'((\langle\uparrow^0(\mathrm{id}), \chi\rangle : M), S)\\ &= \mathscr{T}'((\uparrow^0\langle\uparrow^0(\mathrm{id}), \chi\rangle; M), S)\\ &= \mathscr{T}(\uparrow^0\langle\uparrow^n(\xi), \chi\rangle, M, S).\end{aligned}$$

3. $(\uparrow^n(\mathrm{id}),\ \uparrow^m(0),\ \chi_1, \ldots, \chi_r) \to (\uparrow^{m+n}(0)\,\chi_1, \ldots, \chi_r,\ \mathrm{end},\ \varnothing)$.
Obvious.

4. $(\uparrow^n\langle\uparrow^p(\xi), \chi\rangle, \uparrow^{m+1}(0), S) \to (\uparrow^{n+p}(\xi), \uparrow^m(0), S)$.

$$\begin{aligned}\mathscr{T}(\uparrow^n\langle\uparrow^p(\xi), \chi\rangle, \uparrow^{m+1}(0), S) &= \mathscr{T}'((\uparrow^n\langle\uparrow^p(\xi), \chi\rangle; \uparrow^{m+1}(0)), S)\\ &= \mathscr{T}'((\langle\uparrow^{n+p}(\xi), \uparrow^n(\chi)\rangle; \mathrm{fst}; \uparrow^m(0)), S)\\ &= \mathscr{T}'((\uparrow^{n+p}(\xi); \uparrow^m(0)), S)\\ &= \mathscr{T}(\uparrow^{n+p}(\xi), \uparrow^m(0), S).\end{aligned}$$

5. $(\uparrow^n\langle\uparrow^p(\xi), \chi\rangle, \uparrow^0(0), S) \to (\uparrow^n(\mathrm{id}), \xi, S)$.

$$\begin{aligned}\mathscr{T}(\uparrow^n\langle\uparrow^p\xi), \chi\rangle, \uparrow^0(0), S) &= \mathscr{T}'((\uparrow^n\langle\uparrow^p(\xi'), \chi\rangle; \uparrow^0(0)), S)\\ &= \mathscr{T}'(\uparrow^n(\mathrm{id}); \langle\uparrow^p(\xi'), \chi\rangle; \mathrm{snd}), S)\\ &= \mathscr{T}'((\uparrow^n(\mathrm{id}); \chi), S)\\ &= \mathscr{T}(\uparrow^n(\mathrm{id}), \chi, S).\end{aligned}$$

6. $$\frac{(\uparrow^m(\xi''), C, \varnothing) \to (N, \mathrm{end}, \varnothing)}{(\uparrow^n(\xi'), \uparrow^m(\xi''; C), S) \to (\uparrow^n(\xi'), N, S)}.$$

By hypothesis, $\mathscr{T}(\uparrow^m(\xi''), C, \varnothing) = \mathscr{T}'((\uparrow^m(\xi''); C), \varnothing) = \uparrow^m(\xi''); C = \mathscr{T}(N, \mathrm{end}, \varnothing) = N$.

Then

$$\begin{aligned}\mathscr{T}(\uparrow^n(\xi'), \uparrow^m(\xi''; C), S) &= \mathscr{T}'((\uparrow^n(\xi'); \uparrow^m(\xi''; C)), S)\\ &= \mathscr{T}'((\uparrow^n(\xi'); (\uparrow^m(\xi''); C)), S)\\ &= \mathscr{T}'((\uparrow^n(\xi'); N), S)\\ &= \mathscr{T}(\uparrow^n(\xi'), N, S).\end{aligned}$$

7. $$\frac{(\uparrow^0\langle\uparrow^{n+1}(\xi), \uparrow^0(0)\rangle, C, \varnothing) \to (N, \text{end}, \varnothing)}{(\uparrow^n(\xi), \text{cur}(C), \varnothing) \to (\text{cur}(N), \text{end}, \varnothing)}.$$

By hypothesis, $\mathscr{T}(\uparrow^0\langle\uparrow^{n+1}(\xi), \uparrow^0(0)\rangle, C, \varnothing) = \mathscr{T}'((\uparrow^0\langle\uparrow^{n+1}(\xi), \uparrow^0(0)\rangle; C), \varnothing)) = \uparrow^0\langle\uparrow^{n+1}(\xi), \uparrow^0(0)\rangle; C = \langle\uparrow^{n+1}(\xi), \uparrow^0(0)\rangle; C = \mathscr{T}(N, \text{end}, \varnothing) = N$.

Then

$$\begin{aligned}\mathscr{T}(\uparrow^n(\xi), \text{cur}(C), \varnothing) &= \mathscr{T}'((\uparrow^n(\xi); \text{cur}(C)), \varnothing)\\ &= \uparrow^n(\xi); \text{cur}(C)\\ &= \text{cur}(\langle \text{fst}; \uparrow^n(\xi), \text{snd}\rangle; C)\\ &= \text{cur}(\langle\uparrow^{n+1}(\xi), \uparrow^0(0)\rangle; C)\\ &= \text{cur}(N)\\ &= \mathscr{T}(\text{cur}(N), \text{end}, \varnothing). \quad \square\end{aligned}$$

The soundness of the controller is obvious. Moreover, it is also evident that the machine returns as result a λ-term in normal form. Thus, by the previous simple theorem and the completeness of the categorical interpretation, we may conclude:

Corollary 10.2 (correctness)
If $N(P) \to Q$, then $P \to \beta Q$, and Q is in normal formal.

What we cannot prove so easily is that if the term P has a head normal form, then the machine will eventually stop.

Roughly speaking, every implementation of the λ-calculus is essentially composed of two distinct parts, one dealing with β-reductions, and the other one realizing the substitution algorithm. For instance, in our second machine, everything but rule 2 can be essentially seen as a substitution algorithm. One could be tempted to say that if the machine is based on a safe strategy in the choice of β-redexes, the termination of the computation, in case the λ-term has a h.n.f., only depends on the termination of the substitution algorithm (that, usually, is not so difficult to prove). However, substitution and β-reduction rules are interleaved in a very complex way, and the previous short cut is not comforted by any theoretical result.

In particular, the substitution algorithm implemented by our second machine is described by the following rewriting system (which can be considered as our counterpart of the system SUBST for Curien's categorical combinators (Hardin, 1987; Hardin and Laville, 1986)).

Definition 10.3 ($SUBST_I$)

Nat_pair.	$\xi; (MN) \to (\xi; M)(\xi; N)$
Free.	$\uparrow^n(\text{id}); \uparrow^m(0) \to \uparrow^{m+n}(0)$
Fst.	$\uparrow^n\langle\uparrow^p(\xi), \chi\rangle; \uparrow^{m+1}(0) \to \uparrow^{n+p}(\xi); \uparrow^m(0)$
Snd.	$\uparrow^n\langle\uparrow^p(\xi), \chi\rangle; \uparrow^0(0) \to \uparrow^n(\text{id}); \chi$
Nat_cur.	$\uparrow^n(\xi); \text{cur}(M) \to \text{cur}(\uparrow^0\langle\uparrow^{n+1}(\xi), \uparrow^0(0)\rangle; M)$.

The previous system does not have any critical pair, thus it is locally confluent (since it is left linear, it is also confluent). We have proved the termination of $SUBST_1$ by relying on the termination of SUBST (we skip the proof, since it is quite technical and not very informative).

However, there is no trivial way to relate the termination of $SUBST_1$ to the termination of the abstract machine (on terms with h.n.f).

11 Conclusions

We have discussed in this paper some examples of (both known and original) environment machines in the unified mathematical framework provided by category theory. The general idea is that the abstract, denotational notion of model for a given calculus, once it is formalized in the categorical language, usually provides a linguistic framework that is sufficiently accurate to serve as a basis for operational investigations. In particular, the categorical language provides a simple formalism for handling terms and environments (substitutions) in a uniform way, and thus for studying their operational interaction at the implementation level.

We have particularly stressed the relevance of the categorical approach as a guide to the definition of new machines, by emphasizing the role of what we called the 'categorical soundness', that defines an invariant over states of the machine during a computation.

In proving that an abstract environment machine actually implements a given functional calculus, we tried to distinguish between two different aspects, namely between termination and correctness (i.e. that fact that, *if the computation stops*, the result is correct). Although we have pointed out some problems of the categorical approach in deriving a 'correctness' result in case the output is a closure (typically with weak reduction strategies), this result can be got by relying only on a purely *equational* theory. On the contrary, it does not seem possible to treat termination in this way. This is the reason why we particularly stressed the relevance of the categorical approach in the creative phase of definition of the machine, rather than in proving properties of it.

One of the aims of the various directed calculi of explicit substitutions which have been developed in recent years is that of handling these two problems at the same time. If we know that our calculus enjoys good properties, by defining a suitable embedding from machines states to terms of the calculus we can derive a lot of information about the machine.

On the other side, directed combinatorial systems, being inspired by given computational models, usually lack flexibility.

Acknowledgements

I would like to thank P. L. Curien, J. J. Lévy, M. Mauny and T. Hardin for the interesting discussions on the topics of this paper. In particular, after finishing my work, I became aware of some unpublished notes of Curien in which he defined some

machines for complete reduction of λ-terms very close to those presented in section 8–9. I am glad to share with him any merit of these results. I would also thank the anonymous referees who helped me so much to improve the early draft of this paper.

References

Abramsky, S. 1989. The lazy λ-calculus. In D. Turner, Ed., *Declarative Programming*, Addison-Wesley.

Abadi, M., Cardelli, L., Curien, P. L. and Lévy, J. J. 1990. Explicit Substitutions. In *Proc. of the 17th Annual Symposium on Principles of Programming Languages*, (*POPL90*). San Francisco. (An extended version is available as Rapport de Recherche INRIA-Rocquencourt no 1176).

Asperti, A. and Longo, G. 1991. *Categories, Types and Structures: An introduction to category theory for the working computer scientist*. MIT Press.

Asperti, A. 1989. *Categorical Topics in Computer Science*. PhD thesis, Università degli Studi di Pisa, Dipartimento di Informatica.

Asperti, A. 1990. Integrating strict and lazy evaluation: the λ_{sl}-calculus. In *Proc. of the 2nd International Workshop on Programming Language Implementation and Logic Programming, PLILP'90*, Linköpink, Sweden, Volume 456 of *Lecture Notes in Computer Science*, Springer-Verlag.

Balsters, H. 1986. *Lambda Calculus extended with Segments*. PhD thesis, University of Eindhoven.

Cardelli, L. 1986. The Amber Machine. In G. Cousineau, P. L. Curien and B. Robinet, Eds., *Combinators and Functional Programming Languages. Volume 242 of Lecture Notes in Computer Science*, Springer-Verlag.

Cousineau, G., Curien, P. L. and Mauny, M. 1987. The Categorical Abstract Machine. *Sci. of Computer Programming*, **8**.

Cousineau, G. and Huet, G. 1986. The CAML Primer. Rapport de Recherche Project Formel, INRIA-Rocquencourt&ENS.

Curien, P. L. and Obtulowicz, A. 1989. Partiality, Cartesian Closedness and Toposes. *Inf. and Computation*, **80** (1).

Crégut, P. 1990. An abstract machine for the normalization of λ-terms. In *Proc. of the ACM Conf. on Lisp and Functional Programming*, Nice, France.

Curien, P. L. 1986. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Pitman.

Curien, P. L. 1988. The λρ-calculus: an Abstract Framework for Environment Machines. Rapport de Recherche du LIENS 88–10.

De Bruijn, N. G. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.* **34**.

Field, J. 1990. On laziness and Optimality in Lambda Interpreters: Tools for Specification and Analysis. In *Proc. of the 17th Annual Symposium on Principles of Programming Languages* (*POPL90*). San Francisco.

Fairbairn, J. and Wray, S. 1987. Tim: a simple, lazy, abstract machine to execute supercombinators. Volume 274 of *Lecture Notes in Computer Science*, Springer-Verlag.

Girard, J. Y. Linear Logic. *Theor. Computer Sci.*, **50**.

Hardin, T. and Laville, A. 1986. Proof of Termination of the Rewriting System SUBST on CCL. *Theor. Computer Sci.*, **46**.

Hardin, T. and Lévy, J. J. 1990. *A Confluent Calculus of Substitutions*. Draft.

Hardin, T. 1987. *Résultats de Confluence pour les Règles Fortes de la Logique Combinatoire Catégorique et Liens avec les Lambda-Calculs*. Thèse de Doctorat, Université Paris VII.

Hardin, T. 1989. Confluence results for the Pure Strong Categorical Combinatory Logic CCL: λ-calculi as Subsystems of CCL. *Theor. Computer Sci.*, **65**.

Hayashi, S. 1985. Adjunctions of Semifunctors: Categorical Structures in Nonextensional Lambda Calculus. *Theor. Computer Sci.*, **41**.

Kennaway, J. 1984. An outline of some results of Staples on optimal reduction orders in Replacement Systems. Internal Report CSA/19/84. University of East Anglia.

Lévy, J. J. 1978. *Réductions correctes et optimales dans le lambda-calcul.* Thèse de doctorat. Université Paris VII.

Mauny, M. 1985. *Compilation des langages fonctionnels dans le combinateurs catégoriques; application au langage ML.* Thèse de Troisième Cycle, Université Paris VII.

Martini, S. 1988. *Modelli non estensionali del polimorfismo in programmazione funzionale.* PhD thesis, Università degli Studi di Pisa, Dipartimento di Infomatica.

Plotkin, G. 1975. Call-by-name, Call-by-value and the λ-calculus. *Theor. Computer Sci.*, **1**.

Revesz, G. 1984. An extension of Lambda Calculus for Functional programming. *J. Logic Computation*, **3**.

Robinson, E. and Rosolini, G. 1988. Categories of Partial Maps. *Infor. and Computation*, **79** (2).

Seely, R. A. G. 1987. Linear Logic, *-autonomous categories and cofree coalgebras. In *Conf. in Category Theory, Computer Science, and Logic*, Boulder, CO (AMS notes).

Sleep, M. 1985. Issues for implementing lambda languages. Notes for Ustica Workshop.