View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Elsevier - Publisher Connector

Science of Computer Programming 8 (1987) 173-202
North-Holland

THE CATEGORICAL ABSTRACT MACHINE

G. COUSINEAU, P.-L. CURIEN and M. MAUNY
Université Paris VII, LITP, 75251 Paris Cedex 05, France

Communicated by G. Berry
Received January 1985
Revised July 1986

Abstract. The Cartesian closed categories have been shown by several authors to provide the right framework of the model theory of λ-calculus. The second author developed this as a syntactic equivalence between two calculi, giving rise to a new kind of combinatory logic: the categorical combinatory logic, where computations can be done through simple rewrite rules, and, as usual with combinators, avoiding problems with variable name clashes. This paper goes further (though not requiring a previous knowledge of categorical combinatory logic) and describes a very simple machine where categorical terms can be considered as code acting on a graph of values (the essential actions are LISP's "cons", "car" and "cdr", as well as "rplacd" to implement recursion). The only saving mechanism is a stack containing pointers on code or on the graph. Abstractions are handled in the very same way as in P. Landin's SECD machine, using closures. The machine is called *categorical abstract machine* or CAM. The CAM is easier to grasp and prove than the SECD machine. The paper discusses the implementation of a real functional programming language, ML, through the CAM. A basic acquaintance with λ-calculus is required.

1. Introduction

We use the categorical framework developed in a syntactic and computational fashion in [8] to define an abstract machine for implementing functional programming languages with static binding of variables, i.e. λ-calculus based languages. Our machine, called *categorical abstract machine* or CAM, may be viewed as a synthesis of three different approaches to the implementation of functional programming languages:
- De Bruijn's formalism for eliminating problems caused by α-conversions [3],
- Turner's SK-reduction machine [21],
- Landin's SECD machine [12, 14].

As recalled in more detail below, De Bruijn's trick was to replace variable names by numbers recording their binding heights, allowing to get an automatic treatment of α-conversions while performing β-reductions. The point is that De Bruijn's notation may be considered as a combinatory logic which is nothing but the second author's categorical combinatory logic [12], endowed with rewrite rules of the very same kind as the SK-rules of combinatory logic involved in the SK-reduction machine. But while the SK-reduction machine sticks to the rewriting mechanism, the CAM is a very basic machine: combinatory terms are nothing but sequences of instructions acting on a register and a stack. The CAM handles abstractions as closures in the same way as the SECD machine. However, the save-restore mechanisms are quite different in the two machines.

The paper is composed as follows: the rest of the introduction is devoted to give to the reader some feeling about categorical combinators and their 'SK-like' rewrite rules. Then in Section 2 we show that our combinatory rules naturally suggest machine instructions, and define the core of the CAM. Section 3 sketches correctness proofs, which are fully developed in [15]. Section 4 describes extensions to handle arithmetic, conditionals and recursion, as well as lazy evaluation which is naturally handled in the call-by-value framework by explicitly introducing delaying or '*freezing*' mechanisms. Section 5 describes an ML implementation of the CAM, hence giving rise to an ML interpreter written in ML. Section 6 is a discussion.

We shortly illustrate the different combinatory approaches to evaluation. Throughout the paper we shall use an ML-like syntax (GorLCF). Our example is

$$\textbf{let } x = \text{plus } \textbf{in } x\ (4,(x \textbf{ where } x = 3));$$

We apologize for the poor interest of the program, but a more involved example would obscure the discussion. Expressed as a λ-expression, our program looks as follows

$$M = (\lambda x.x(4,(\lambda x.x)3)) +$$

By doing so we loose some optimization (see Section 2, but we do not care here).

First we take a look at the classical combinator approach. The expression M is compiled into a combinatory term. But combinatory logic does not know about products and handles only curry-ed functions. So we have to consider a curry-ed addition and write

$$N = (\lambda x.x4((\lambda x.x)3)) +$$

The translation into combinatory logic is (without using Turner's optimizations)

$$S(SI(K4))(S(KI)(K3)) +$$

Now using one or another computation strategy based on the well-known rules

$$Sxyz = xz(yz), \qquad Kxy = x$$

we get 7. Here in an innermost-leftmost sequence:

$$\begin{aligned} &S(SI(K4))(S(KI)(K3)) + \to (SI(K4)+)(S(KI)(K3)+) \\ &\quad \to (I+)(K4+)(S(KI)(K3)+) \\ &\quad \to +(K4+)(S(KI)(K3)+) \to +4(S(KI)(K3)+) \to +4((KI+)(K3+)) \\ &\quad \to +4(I(K3+)) \to +43 \to 7. \end{aligned}$$

Now we turn to the De Bruijn's style. We could have computed M by performing β-reductions, yielding

$$M = (\lambda x.x(4,(\lambda x.x)3)) + \to +(4,(\lambda x.x)3) \to +(4,3) \to 7.$$

But it is well known that β-conversions give rise to boring problems of name clashes (not in our example, which is too simple). For instance $(\lambda xy.x)y$ by no way reduces to $\lambda y.y$. De Bruijn's idea is to avoid the name clashes by getting rid of the names themselves. The only important information about a variable in a closed term (i.e. having no free variables) is its *binding height*, i.e. the number of λ's between it and (not including) the binding one. Then the variable names are replaced by this number (not to be confused with an integer constant), and De Bruijn showed that this gives rise to an elegant treatment of β-reductions. For instance

$$P = \lambda y.(\lambda xy.x)y \quad \text{becomes} \quad \lambda.(\lambda\lambda.1)0.$$

A suitable rephrasing of the β-rule (see Section 3) yields $\lambda\lambda.1$, and one never has to change explicitly $\lambda xy.x$ into, say $\lambda xz.x$, to avoid the name clash.

Actually it was shown in [8] that the De Bruijn's notation not only yields an elegant treatment of β-conversion, but also may be considered as a combinatory logic with rules of the very same kind as the SK-rules.

Since we would like the paper to be self-contained, we try to introduce these ideas briefly below (more details may be found in the cited references). We shall introduce a semantic setting to motivate our rules.

First we come back to the normal λ-calculus, and try to describe the meanings of expressions. They depend on associations of values with identifiers, i.e. on environments. Thus M has as meaning a function $[\![M]\!]$ associating a value with an environment. We get the well-known semantic equations, where *applying* a function to its argument is denoted by simple juxtaposition:

$$[\![x]\!]\rho = \rho(x),$$

$$[\![c]\!]\rho = c,$$

$$[\![MN]\!]\rho = [\![M]\!]\rho([\![N]\!]\rho),$$

$$[\![\lambda x.M]\!]\rho d = [\![M]\!]\rho[x \leftarrow d]$$

where $\rho(x)$ is the value associated with x in ρ, c is a constant denoting a value, also called c following usual practice, $\rho[x \leftarrow d]$ is ρ where x has been updated with value d.

How should these equations be rephrased when replacing classical λ-expressions by expressions in De Bruijn's notation? We assume that ρ has the shape $(\ldots((), v_n) \ldots, v_0)$ where v_i is associated with i. We leave the reader convince himself that the following is obtained:

$$[\![0]\!](\rho, d) = d, \qquad [\![n+1]\!](\rho, d) = [\![n]\!]\rho,$$

$$[\![c]\!]\rho = c,$$

$$[\![MN]\!]\rho = [\![M]\!]\rho([\![N]\!]\rho),$$

$$[\![\lambda.M]\!]\rho d = [\![M]\!](\rho, d).$$

But we are not interested in meanings for themselves, we want meanings to suggest computations. Our combinatorial approach stresses that the meaning of, say MN, is a *combination* of the meanings of M and N. We introduce three combinators: S of arity 2, Λ, $'$, of arity 1, and infinitely many combinators $n!$ with the intention that

$$[\![n]\!]=n!,\quad [\![c]\!]={}'c,\quad [\![MN]\!]=S([\![M]\!],[\![N]\!]),\quad [\![\lambda.M]\!]=\Lambda([\![M]\!]).$$

This allows us to transform our semantic equations into purely syntactic ones:

$$0!(x,y)=y,\qquad (n+1)!(x,y)=n!x,$$

$$('x)y=x,$$

$$S(x,y)z=xz(yz),$$

$$\Lambda(x)yz=x(y,z).$$

This is not so far from the SK-rules: the first three rules forget about an argument just as K does, while the fourth rule is an uncurried form of the S-rule; finally the last rule precisely describes *currying*, i.e. transforming a function with two arguments (more precisely one argument which is a couple, see below for a discussion of coupling) into a function of its first argument yielding a function of its second argument. Hence, roughly speaking, categorical combinatory logic is something like "*combinatory logic with explicit products*".

This remark directly leads to considering explicit products in the λ-calculus, as in the subterm $(4,(\lambda x.x)3)$ of our example. So we introduce a pairing combinator $\langle\,\rangle$, intending

$$[\![(M,N)]\!]=\langle[\![M]\!],[\![N]\!]\rangle.$$

It is also very natural to associate the destructors corresponding to this new combinator, i.e. the projections *Fst* and *Snd*. This gives rise to the following equations:

$$Fst(x,y)=x,$$

$$Snd(x,y)=y,$$

$$\langle x,y\rangle z=(xz,yz).$$

The last equation relates pairing with *coupling*: in a set theoretical setting, the pair of two functions $f:D\mapsto E$, $g:D\mapsto F$ is the function $h:D\mapsto E\times F$ which has as output the couple of their outputs.

Notice that what we call here couples are often called pairs (in particular in ML). But here we have to distinguish (the couple of f, g above is not a function, it is an element in $(D\Rightarrow E)\times(D\Rightarrow F)$), and we have no better name for pairing functions.

Actually we have introduced coupling earlier in our exposition, when discussing the 'shape' of the environment: we have taken the view of a tree (or graph) representation of environments, the nodes being coupling nodes. A final remark about coupling: we stress that *Fst*, *Snd* have arity 0; hence $Fst(x, y)$ has to be read *Fst* applied to the couple of x, y.

Clearly there is some redundancy in all these rules: the rules about *Fst* and $n+1!$, about *Snd* and $0!$, about $\langle\, ,\rangle$ and S are quite similar: duplication may be avoided by introducing a new binary operator, the ***composition***, i.e. the ***basic tool of category theory***, and a new constant *App*.

Now consider $S(\, ,)$ and $n!$ as shorthands for $App \circ \langle\, ,\rangle$ and $Snd \circ Fst^n$ (setting $Fst^{n+1} = Fst \circ Fst^n$). Then we can replace the two tables of combinatory equations by the following:

$$
\begin{array}{ll}
(ass) & (x \circ y)z = x(yz), \\
(fst) & Fst\,(x, y) = x, \\
(snd) & Snd\,(x, y) = y, \\
(dpair) & \langle x, y\rangle z = (xz, yz), \\
(ac) & App(\Lambda(x)y, z) = x(y, z), \\
(quote) & ('x)y = x
\end{array}
$$

(notice that *ass* relates composition to application as *dpair* does with pairing and coupling).

We have chosen to make one rule out of the above rules on S, Λ (notice that the rule on S becomes $S(x, y)z = App(xz, yz)$), to get a more homogeneous treatment of our three operators of arity 0: *Fst*, *Snd* and *App*. Notice also that we get as a derived rule

$$\Lambda(x \circ Snd)yz = (x \circ Snd)(y, z) = xz \quad \text{(without using } quote\text{)}$$

which is of course implied by *quote*. We shall use this coding for functional constants such as +. The reason of the difference between codings of basic and functional constants will appear clearly after the description of the CAM instructions.

Now it is time to apply our general discussion to an actual computation. Our example M becomes $M' = S(\Lambda(S(0!, \langle '4, S(\Lambda(0!), '3)\rangle)), \Lambda(+ \circ Snd))$.

Here is where our discussion gets connected with the SECD machine approach: M' is going to be evaluated by ***applying it to an environment***, namely the empty environment since our term is closed.

We evaluate $M'()$, by an innermost-leftmost strategy. We set

$$A = S(0!, \langle '4, B\rangle) \quad \text{and} \quad B = S(\Lambda(0!), '3).$$

Here is the reduction sequence:

$$
\begin{aligned}
&S(\Lambda(A), \Lambda(+\circ Snd))() \to App(\Lambda(A)(), \Lambda(+\circ Snd)())\\
&\quad\to A\rho \quad (\text{setting } \rho = ((), \Lambda(+\circ Snd)()))\\
&\quad\to App(0!\rho, \langle '4, B\rangle\rho) \to App(\Lambda(+\circ Snd)(), \langle '4, B\rangle\rho)\\
&\quad\to App(\Lambda(+\circ Snd)(), ('4\rho, B\rho))\\
&\quad\to App(\Lambda(+\circ Snd)(), (4, B\rho))\\
&\quad\to App(\Lambda(+\circ Snd)(), (4, App(\Lambda(0!)\rho, '3\rho)))\\
&\quad\to App(\Lambda(+\circ Snd)(), (4, App(\Lambda(0!)\rho, 3)))\\
&\quad\to App(\Lambda(+\circ Snd)(), (4, 0!(\rho, 3)))\\
&\quad\to App(\Lambda(+\circ Snd)(), (4, 3)) \to (+\circ Snd)((), (4, 3))\\
&\quad\to +(Snd((), (4, 3))) \to +(4, 3) \to 7.
\end{aligned}
$$

The last step is to explain how all these computations can be carried out by an abstract machine, which we do in the next section.

2. The core of the categorical abstract machine

Before going further, it is time to justify the attribute 'categorical'. Categories are a game with *composition* and *identity* (the identity will arise to perform an optimization, see below), Cartesian categories add rules about *products* to the game, involving $\langle\,,\rangle$ (*pairing*), *Fst*, *Snd*. Cartesian closed categories add to the game Λ (*currying*) and *App*, allowing to talk about *exponentials*, i.e. function spaces. More detail would lead us out of our scope and we refer to [8].

Our rules evaluate categorical terms like M' above, built with those combinators, by *applying* them to an environment, which is built by *coupling* its different components. Notice that these two last operators are not present in M'; applying is introduced when starting with $M'()$, while new couples are created each time *dpair*, *ac* are used. So we could call them *dynamic* combinators, while the others, which appear at compile time, are *static*.

Now we want to suggest that the static operators naturally give rise to very basic machine intructions. First we remark that along the rules applied in reducing $M'()$ above, the redexes had all the form Mv where v is a value, i.e. a term in normal form w.r.t. our rules, and M is (the transformation of) a term in De Bruijn's notation. Now consider M as code acting on v. M is made of elementary pieces of code.

Fst and *Snd* are easily viewed as instructions: *Fst* acts on a value (v_1, v_2) by accessing to its first son (supposing the value is represented as a binary tree). So this can explain how a variable is going to be evaluated. For the couples, we are concerned with the action of $\langle M, N\rangle$ on v. The equation about $\langle\,,\rangle$ tells us what to do:

$$\langle M, N\rangle v = (Mv, Nv).$$

So the actions of M, N on v should be carried out independently and then put together by building a tree whose root is a couple, and whose sons are the values v_1, v_2 obtained. In a sequential machine we have to choose to evaluate, say M, first. This yields v_1. But we should have stored v before working on it, in order to restore it when tackling N, yielding v_2. Finally we put together v_1 and v_2, but this supposes again that we have stored v_1, which should have been done exactly when we have restored v.

Now we have the structure of our machine: a *term* (rather a graph in an actual implementation), which is a structured value, a *code*, and a *stack* (or dump). So a state of the machine is a triplet with respective components T, C, S. The discussion above suggests to define the code for $\langle M, N\rangle$ as the following sequence of instructions: '⟨' followed by the sequence of instructions corresponding to the code of M, followed by ',', followed by the sequence of instructions corresponding to the code of N, followed by '⟩', where the effects of '⟨', "," and '⟩' are respectively:

⟨: push the term onto the top of the stack,
,: swap the term and the top of the stack,
⟩: make a couple out of the top of the stack and the term, replace the term by the couple just built, and pop the stack.

Notice that we have not done any compiling at all: The concrete syntax used for pairing corresponds exactly to where control instructions have to be inserted to combine the evaluations of M, N into the evaluation of $\langle M, N\rangle$.

Now that we have the structure of the machine we can describe *Fst* and *Snd* in a more precise way:

Fst: expects a term (s, t) and replaces it by s,
Snd: expects a term (s, t) and replaces it by t.

The code for $n!$ is made of n '*Fst*' instructions followed by an '*Snd*' instruction. Again to get the code, we would have had nothing to do if we had taken $x|y$ (or even $x\,y$) as the concrete syntax of the composition which we denote by $y \circ x$ according to the most usual mathematical practice.

For currying, the code of $\Lambda(M)$ is just $\Lambda(C)$ where C is the code of M (actually its address in a realistic implementation, see Section 5), and the action of 'Λ' is

Λ: replace the term s by $C{:}s$ where C is in the code encapsulated in Λ.

$C{:}s$ is only a shorthand notation for '$\Lambda(M)s$'. From the rewrite rule point of view, the action of 'Λ' is none, since $\Lambda(M)v$ is a value as soon as v is a value, hence may not be rewritten. In terms of actions, this can be rephrased as: the action of $\Lambda(M)$ on v is $\Lambda(M)v$, whence the description of the command 'Λ', which is nothing but *building a closure as in the SECD machine.* Indeed, as stressed by our notation $C{:}s$, we handle as a value a couple made of the code corresponding to the body of a λ-expression, and a value, which is nothing but the declaration environment of the function described by the abstraction (see the second example below).

We continue with λ-calculus application, rephrased as $App \circ \langle M, N\rangle$. The underlying rule is now $(App \circ \langle M, N\rangle)v = App(Mv, Nv)$.

Suppose (Mv, Nv) has been evaluated to (v_1, v_2), which is the job of the code associated with $\langle M, N\rangle$. What remains to be done is the instruction '*App*' which expects $v_1 = \Lambda(P)v_1'$ and will perform $App(\Lambda(P)v_1', v_2) = P(v_1', v_2)$. In terms of our machine the code corresponding to $App \circ \langle M, N\rangle$ is the code for $\langle M, N\rangle$ followed by '*App*', with the following effect:

App: expects a term $(C{:}S, t)$, replaces it by (s, t) and prefixes the rest of the code by C.

We still have the constants to deal with: for basic constants like integers the code for $'c$ is $'(c)$, with the following action:

$'$: replaces the term by the encapsulated constant.

For others, which are functions, like the addition, we use the coding suggested in the previous section: the code for $'c$ is $\Lambda(C)$ where C is *Snd* followed by 'c'.

The description of the core of our machine is now complete. We summarize it by Table 1. We give to the instructions a name in relation with their behaviours, so that

Fst *Snd* $\langle$, $\rangle$ *App* Λ $'$

become

fst snd push swap cons app cur quote

and we use semicolons to separate instructions.

Table 1
CAM: the λ-calculus with explicit products

Configuration			Configuration		
Term	Code	Stack	Term	Code	Stack
(s, t)	fst;C	S	s	C	S
(s, t)	snd;C	S	t	C	S
s	(quote c);C	S	c	C	S
s	(cur C);$C1$	S	$(C{:}s)$	$C1$	S
s	push;C	S	s	C	$s.S$
t	swap;C	$s.S$	s	C	$t.S$
t	cons;C	$s.S$	(s, t)	C	S
$(C{:}s, t)$	app;$C1$	S	(s, t)	C;$C1$	S
(m, n)	plus;C	S	$m+n$	C	S

Notice that the sequences ‘cons; app’, ‘cons; plus’ should be obviously contracted, avoiding a useless ‘cons’, so that ‘app’, ‘plus’, would act as true binary instructions, taking as arguments the top of the stack and the term. However, we keep hereafter the decomposition as it is, because this allows a simpler explanation of a number of optimizations. In our real implementation, the optimization is performed when expanding CAM code to host machine code.

We run our example, using the mathematical symbols to stress the relation with the rewrite rules used at the end of the previous section (A, B denote the codes corresponding to previous section’s A, B).

$()$	$\langle \Lambda(A), \Lambda(Snd+)\rangle App$	$[]$
$()$	$\Lambda(A), \Lambda(Snd+)\rangle App$	$[()]$
$A{:}()$	$, \Lambda(Snd+)\rangle App$	$[()]$
$()$	$\Lambda(Snd+)\rangle App$	$[A{:}()]$

Now ‘+’ stands as an abbreviation for $(Snd+){:}()$

$+$	$\rangle App$	$[A{:}()]$
$(A{:}(), +)$	App	$[]$
$((), +)$	$\langle Snd, \langle '4, B\rangle\rangle App$	$[]$
$((), +)$	$Snd, \langle '4, B\rangle\rangle App$	$[((), +)]$
$+$	$, \langle '4, B\rangle\rangle App$	$[((), +)]$
$((), +)$	$\langle '4, B\rangle\rangle App$	$[+]$
$((), +)$	$'4, B\rangle\rangle App$	$[((), +); +]$
4	$, B\rangle\rangle App$	$[((), +); +]$
$((), +)$	$B\rangle\rangle App$	$[4; +]$
$((), +)$	$\Lambda(Snd), '3\rangle App\rangle\rangle App$	$[((), +); 4; +]$
$Snd{:}((), +)$	$, '3\rangle App\rangle\rangle App$	$[((), +); 4; +]$
$((), +)$	$'3\rangle App\rangle\rangle App$	$[Snd{:}((), +); 4; +]$
3	$\rangle App\rangle\rangle App$	$[Snd{:}((), +); 4; +]$
$(Snd{:}((), +), 3)$	$App\rangle\rangle App$	$[4; +]$
$(((), +), 3)$	$Snd\rangle\rangle App$	$[4; +]$
3	$\rangle\rangle App$	$[4; +]$
$(4, 3)$	$\rangle App$	$[+]$
$(+, (4, 3))$	App	$[]$

We expand back our abbreviation:

$((), (4, 3))$	$Snd+$	$[]$
$(4, 3)$	$+$	$[]$
7	$[]$	$[]$

The above session demonstrates clearly the interest of our coding of functional constants. If we had taken the code ‘(quote +)’ for the constant +, we would have been constrained to change the behaviour of ‘app’, which would have to test whether the first component of the term is a closure, or any of the functional constants. The chosen coding avoids performing such tests.

Now we run a more involved example, where the closure mechanism appears clearly, in connection with static binding:

$$\textbf{let } x=5 \textbf{ in let } z\,y=y+x \textbf{ in let } x=1 \textbf{ in } (zx)*2;;$$

The corresponding λ-expression is

$$P=(\lambda x.(\lambda z.(\lambda x.(zx)*2)1)(\lambda y.y+x))5.$$

But actually the first notation is better and leads to a simple compile time optimization, as we explain now. The point is that the code corresponding to $(\lambda.M)N$ is 'push', followed by 'cur C' (where C is the code of M), followed by 'swap', followed by the code $C1$ of N followed by 'cons' and 'app'. Supposing that evaluating $C1$ on term s yields value v, the important steps are:

s	push;(cur C);swap;$C1$;cons; app	S
s	(cur C);swap;$C1$;cons; app	$s.S$
$C{:}s$	swap;$C1$;cons; app	$s.S$
s	$C1$;cons; app	$(C{:}s).S$
v	cons; app	$(C{:}s).S$
$(C{:}s, v)$	app	S
(s, v)	C	S

Now the same result is achieved by the following optimized code: 'push' followed by the code of N, followed by 'cons', followed by the code of M, as shown below:

s	push;$C1$;cons;C	S
s	$C1$;cons;C	$s.S$
v	cons;C	$s.S$
(s, v)	C	S

This optimization has to do with the identity combinator, without which our combinatory logic would hardly be called 'categorical'! This leads us to a short digression; besides being able to evaluate expressions relative to an environment as developed here, categorical combinatory logic is able to simulate the β-reductions in a natural way, using a set of rules distinct from the one used here, involving only the 'pure' categorical combinators, i.e. excluding coupling and applying. The start-point is the following rule:

$$(\mathit{Beta})\quad \mathit{App}\circ\langle\Lambda(x),y\rangle=x\circ\langle \mathit{Id},y\rangle$$

which allows then to distribute $\langle \mathit{Id}, y\rangle$ (or whatever it becomes) along the structure of x up to the leaves, which are $n!$ for some n. Then we need the projection rules, very similar to the ones involving the couple operator:

$$\mathit{Fst}\circ\langle x,y\rangle=x,\qquad \mathit{Snd}\circ\langle x,y\rangle=y.$$

As an illustration we simulate

$$\lambda x.(\lambda y.y)x\rightarrow\lambda x.x$$

which cannot be simulated neither by the *SK*-rules, nor by the rules on which our machine is based, but rather either by the Curry axioms [13] or by rules of the kind of the three rules we have just listed (in contrast to Curry axioms, the categorical setting is very intuitive). Here is the simulation:

$$\Lambda(App \circ \langle \Lambda(Snd), Snd \rangle) \to \Lambda(Snd \circ \langle Id, Snd \rangle) \to \Lambda(Snd).$$

Now coming back to our optimization, *Beta* suggests the following code and reduction for **let** $x = N$ **in** M:

s	push;skip;swap;$C1$;cons;C	S
s	skip;swap;$C1$;cons;C	$s.S$
s	swap;$C1$;cons;C	$s.S$
s	$C1$;cons;C	$s.S$
v	cons;C	$s.S$
(s, v)	C	S

where ‘skip’ stands for *Id*, with the nonaction effect. But the effect of ‘push: skip; swap’ is clearly the effect of ‘push’, and we have obtained a ‘theoretical foundation’ for the optimization proposed above.

Another useful optimization is to compile $M + N$ into: code of $\langle M, N \rangle$, followed by ‘plus’. This is justified by

$$App \circ \langle \Lambda(+ \circ Snd), \langle M, N \rangle \rangle = + \circ Snd \circ \langle Id, \langle M, N \rangle \rangle = + \circ \langle M, N \rangle.$$

Here is the evaluation of our term *P*. We use the notation $x | y = y \circ x$ to make compiling easier to follow; we take as abbreviations

$$B = \langle C, '2 \rangle | *, \qquad C = \langle Fst \,|\, Snd, Snd \rangle | App, \qquad D = \langle Snd, Fst \,|\, Snd \rangle | +$$

which correspond to the subterms $(zx) * 2$, zx and $y + x$. We also contract some steps:

()	⟨'5⟩⟨*Λ*(*D*)⟩⟨'1⟩*B*	[]
((), 5)	⟨*Λ*(*D*)⟩⟨'1⟩*B*	[]
(((), 5, *D*:((), 5))	⟨'1⟩*B*	[]
((((), 5), *D*:((), 5)), 1)	*B*	[]
((((), 5), *D*:((), 5)), 1)	*C*, '2⟩*	[((((), 5), *D*:((), 5)), 1)]
((((), 5), *D*:((), 5)), 1)	1!, 0!⟩*App*, '2⟩*	[((((), 5), *D*:((), 5)), 1); ((((), 5), *D*:((), 5)), 1)]
((((), 5), *D*:(((), 5)), 1)	0!⟩*App*, '2⟩*	[*D*:((), 5); ((((), 5), *D*:((), 5)), 1)]
(*D*:((), 5), 1)	*App*, '2⟩*	[((((), 5), *D*:((), 5)), 1)]
(((), 5), 1)	0!, 1!⟩+, '2⟩*	[(((), 5), 1);((((), 5), *D*:((), 5)), 1)]
(((), 5), 1)	1!⟩+, '2⟩*	[1; ((((), 5), *D*:((), 5)), 1)]
(1, 5)	+, '2⟩*	[((((), 5), *D*:((), 5)), 1)]
((((), 5), *D*:((), 5)), 1)	'2⟩*	[6]
(6, 2)	*	[]
12	[]	[]

We come to the promised discussion of static binding. Static binding means that the free occurrence of x in the definition of z is bound to 5, the value of x when z is declared. Dynamic binding would instead amount to bind the same x to whatever its value is when the body $y + x$ is evaluated, which would be 1 in our example.

To avoid this dynamic capture, the SECD machine had a closure mechanism, which is nothing but assigning as value to an abstraction the couple of the abstraction and its declaration environment. And this is *exactly* what our rule about 'cur' does. And the reader may check that the instance of that rule creating D:((), 5) expresses Landin's closure feature. The complementary feature is Landin's 'apply' instruction, which evaluates the body D w.r.t. the environment obtained from the closure and from the value last obtained (the argument of the function which the abstraction represents). This is exactly what our rule 'app' does. But Landin's 'apply' also entails saving mechanisms, which in our setting are carried out by 'push' and 'swap'. Actually this is the *only* significant difference between the CAM and the SECD machine.

For the sake of completeness, we try to make the similarity clear, starting from the classical description of the SECD machine, as found in the original paper [14], or in tutorial presentations as [12], with the difference that, consistently with the CAM compiler, we assume evaluation from left to right in applications. So the code for λ-calculus application MN is the code of M followed by the code of N followed by 'apply'. The code of a number n! is 'access n', and the code for constants and abstractions is as in the CAM (the code encapsulated in a closure ending with a 'return', see Section 5). Products are avoided by currying $*$ and $+$. Table 2 gives the rules of the machine, which has a stack of values, an environment component, a code component, and a stack where environments are saved (the dump in the SECD terminology).

At this stage, nothing is said about the way of representing environments. The notation is not only vague, but even rather contradictory: 'access n' suggests a vector, but the operation '$v.E1$' is practically unfeasible as such: only a fixed size of memory can be allocated for environments as vectors. Thus we are free to interpret the SECD machine in a context where environments are represented as in the CAM. Then 'access n' has to be changed into a sequence 'access; fst;...; fst; snd', where the role of 'access' is to copy the top of the environment stack to the value stack. We reformulate the machine, also simplifying it by considering the E component as being the top of the D component, see Table 3. This looks not so different from the CAM! To stress even more the similarity we present in Table 4 a two-component version of the CAM, where the value component is now the top of the stack.

It should now be clear that the difference between the two last tables lies only in the way of saving environments. The CAM has the conceptually simplest approach (which is reflected by the simplicity of the correctness proof, as compared with the proof of correctness of the SECD machine [18]). The other approach seems to save some stack manipulations (think of expressions $MN_1N_2\ldots N_n$, or $\langle\ldots\langle M_1, M_2\rangle,\ldots M_n\rangle$), but the price to pay is a more complex machine structure (three components against two): a true implementation of two stacks is not so easy. Let us mention also that some optimizations of the CAM tend to minimize the number of stack manipulations by recognizing when expressions really need their environment.

Table 2
The SECD machine

Configuration			
Stack	Environment	Code	Dump
S	E	(access n);C	D
S	E	(quote c);C	D
S	E	(cur C);$C1$	D
$v.(C:E1).S$	E	apply;$C1$	D
$n.m.S$	E	plus;C	D
S	$E1$	return;C	$E.D$
$v^{(a)}.S$	E	C	D
$c.S$	E	C	D
$(C:E).S$	E	$C1$	D
S	$v.E1$	$C;C1$	$E.D$
$m+n.S$	E	C	D
S	E	C	D

[a] v is the nth element of E.

Table 3
The SECD machine, revisited

Stack	Code	Dump	Stack	Code	Dump
S	access;C	$s.D$	$s.S$	C	$s.D$
$(s, t).S$	fst;C	D	$s.S$	C	D
$(s, t).S$	snd;C	D	$t.S$	C	D
S	(quote c);C	D	$c.S$	C	D
S	(cur C);$C1$	$s.D$	$(C:s).S$	$C1$	$s.D$
$t.(C:s).S$	apply;$C1$	D	S	$C;C1$	$(s, t).D$
$n.m.s$	plus;C	D	$m+n.S$	C	D
S	return C	$t.D$	S	C	D

Table 4
CAM revisited

Stack	Code	Stack	Code
$(s, t).S$	fst;C	$s.S$	C
$(s, t).S$	snd;C	$t.S$	C
$s.S$	(quote c);C	$c.S$	C
$s.S$	(cur C);$C1$	$(C:s).S$	$C1$
$s.S$	push;C	$s.s.S$	C
$t.s.S$	swap;C	$s.t.S$	C
$t.s.S$	cons;C	$(s, t).S$	C
$t.(C:s).S$	app;$C1$	$(s, t).S$	$C;C1$
$n.m.S$	plus;C	$m+n.S$	C

There exists another interpretation of the abstract description of the SECD adopted for example in Cardelli's Functional Abstract Machine [5]. He keeps the environment-as-vector point of view; his solution in the 'apply' rule is to keep v in the stack, and to create environments-as-vectors only when a closure has to be built. This entails a distinction between *local* and *global* variables, which are accessed in the stack, in the vector respectively. The efficiency of his method as compared with ours is clear for access times and function application, while closure building is his most expensive operation. This cost becomes really apparent when running highly functional programs, or implementing lazyness. On the other hand, our actual implementation represents the top level environment with a symbol table, so that the access time problem concerns only the local environments, which in practice are of small size.

3. Proofs (for the core of the CAM)

Establishing the correctness of our machine amounts to formally justify that it is both like a reduction machine, and like a 'De Bruijn' machine, i.e. a device performing β-reductions in the De Bruijn's notation. More precisely we want to prove

(1) that the CAM stops with empty code and stack if and only if the evaluated term reduces in innermost strategy to the term of the final configuration, using the rules of Section 1;

(2) that the innermost combinatory evaluation of a term stops if and only if its call-by-value evaluation by De Bruijn's β-reduction stops, and that the final De Bruijn's term realizes the final combinatory term in a sense which we define below.

Since we tackle formal proofs, we need a formal definition of our different calculi. We shall call *De Bruijn's calculus* the set DBC of terms defined as follows (using grammar notation):

$$DBC ::= n!\,|\,\langle DBC, DBC\rangle\,|\,fst(DBC)\,|\,snd(DBC)\,|\,S(DBC, DBC)\,|\,\Lambda(DBC).$$

Notice that unlike ML (see Section 5) we consider here *fst*, *snd* as operators of arity 1: this allows for an easier compilation. De Bruijn's expressions are denoted by $M, N, M_1, \ldots$.

The De Bruijn's expressions can be considered as categorical terms if $n!$ (called *variable*), $fst(DBC)$, $snd(DBC)$) and $S(DBC, DBC)$ are taken as short-hands for $Snd \circ Fst^n$, $Fst \circ DBC$, $Snd \circ DBC$ and $App \circ \langle DBC, DBC\rangle$, resp.

We propose to describe the rewrite strategy in the form of a deductive system in the style of [19]. The system is defined on a set of *questions* and *answers*. Questions have the form $M?v$ where M is a De Bruijn's expression and v is a value; values are certain categorical terms (we shall be more precise later, but for our first proposition we do not care). Answers are values. A *state* is either a question or an answer (the rules apply if no one before may apply):

$$v \xrightarrow[C]{*} v$$

$$\Lambda(M)?v \xrightarrow[C]{*} \Lambda(M)v$$

$$(n+1)!?(v_1, v_2) \xrightarrow[C]{*} n!?v_1$$

$$0!?(v_1, v_2) \xrightarrow[C]{*} v_2$$

$$\frac{M?v \xrightarrow[C]{*} \Lambda(M_1)v_1 \quad N?v \xrightarrow[C]{*} v_2}{S(M, N)?v \xrightarrow[C]{*} M_1?(v_1, v_2)}$$

$$\frac{M?v \xrightarrow[C]{*} v_1 \quad N?v \xrightarrow[C]{*} v_2}{S(M, N)?v \xrightarrow[C]{*} App(v_1, v_2)}$$

$$\frac{M?v \xrightarrow[C]{*} v_1 \quad N?v \xrightarrow[C]{*} v_2}{\langle M, N\rangle?v \xrightarrow[C]{*} (v_1, v_2)}$$

$$\frac{M?v \xrightarrow[C]{*} (v_1, v_2)}{fst(M)?v \xrightarrow[C]{*} v_1}$$

$$\frac{M?v \xrightarrow[C]{*} v_1}{fst(M)?v \xrightarrow[C]{*} Fst(v_1)}$$

$$\frac{M?v \xrightarrow[C]{*} (v_1, v_2)}{snd(M)?v \xrightarrow[C]{*} v_2}$$

$$\frac{M?v \xrightarrow[C]{*} v_1}{snd(M)?v \xrightarrow[C]{*} Snd(v_1)}$$

$$\frac{s_1 \xrightarrow[C]{*} s_2 \quad s_2 \xrightarrow[C]{*} s_3}{s_1 \xrightarrow[C]{*} s_3}$$

The second rule accounts for the rule 'cur' of the CAM: $\Lambda(M)$ as a De Bruijn's expression acts on v and yields $\Lambda(M)v$ which is now a value.

Now we formalize the compilation of Section 2. We call *CAM* the compiling function:

$$CAM(n+1!) = \text{fst};CAM(n!),$$

$$CAM(0!) = \text{snd},$$

$$CAM(\Lambda(M)) = (\text{cur } CAM(M)),$$

$$CAM(S(M, N)) = \text{push } CAM(M);\text{swap};CAM(N);\text{cons; app},$$

$$CAM(\langle M, N\rangle) = \text{push } CAM(M);\text{swap};CAM(N);\text{cons},$$

$$CAM(fst(M)) = CAM(M);\text{fst},$$

$$CAM(snd(M)) = CAM(M);\text{snd}.$$

Clearly CAM is injective, so that we may write

$$M = CAM^{-1}(C) \quad \text{if } CAM(M) = C.$$

We shall need to add rules to the CAM to be able to build the normal form in the term part when the machine stops, since it may stop with nonempty code, as it stands. For example, when executing $fst(A(Snd))$ (our untyped setting allows this), the machine stops with code 'fst' and term (cur snd). So we duplicate the rules for 'fst', 'snd' and 'cons', with the convention that the new ones apply only when the corresponding three 'active' rules defined in Section 2 cannot apply. We slightly rephrase the machine in order to avoid heaviness in the correctness statement, see Table 5. Also we shall write $\{s, C, S\} \to \{s_1, C1, S_1\}$ when the machine moves from a configuration to another by one of its rules. Now the correctness of the CAM relies on the following.

Proposition 3.1. *For any De Bruijn's expression M and any value v, the following are equivalent*:

(1) $M?v \to_C^* v_1$;

(2) $\{v, CAM(M), []\} \xrightarrow{*} \{v_1, [], []\}$.

Proof. In both directions the following remark is useful: if

$$\{s, C, S\} \xrightarrow{*} \{s_1, C1, S_1\},$$

then for any $C2$ and S_2

$$\{s, C@C2, S@S_2\} \xrightarrow{*} \{s_1, C1@C2, S_1@S_2\}.$$

To show (1)⇒(2) we also show (1a)⇒(2a) where

(1a) $M?v \to_C^* M_1?v_1$;

(2a) $\{v, CAM(M), []\} \xrightarrow{*} \{v_1, CAM(M_1), []\}$.

Table 5

(s, t)	fst;C	S	s	C	S
s	fst;C	S	$Fst\, s$	C	S
(s, t)	snd;C	S	t	C	S
s	snd;C	S	$Snd\, s$	C	S
s	(cur C);$C1$	S	$A(CAM^{-1}(C))s$	$C1$	S
s	push;C	S	s	C	$s.S$
t	swap;C	$s.S$	s	C	$t.S$
t	cons;C	$s.S$	(s, t)	C	S
$(A(M)s, t)$	app;$C1$	S	(s, t)	$CAM(M);C1$	S
s	app;C	S	$App\, s$	C	S

The proof is a routine inspection of the different cases of the deduction system. In the other direction the property to be proved is that if the CAM started as described in the statement stops after n steps, then the final configuration has the shape shown in (2), and (1) is true, which is again a routine induction on n. $\square$

The second proof is more involved, and requires entering in the detail of states.

The *environment order* of a De Bruijn's expression M is the maximal difference, when nonnegative, $n+1-m$ where $n!$ ranges over variable occurrences of M and m is the number of Λ's above the concerned occurrence in the term. A De Bruijn's expression is *closed* when its environment order is 0.

We restrict the set of *values* to be the set of terms defined as follows:

- abstraction: $\Lambda(M)((), v_{n-1}, \ldots, v_0)$ is a value (more specifically an *abstraction value*) if M is a De Bruijn's expression of environment order at most n and $v_0, \ldots, v_{n-1}$ are values,
- couple: (v_1, v_2) is a value if v_1, v_2 are values,
- first projection: *Fst* v if v is a value which is not a couple,
- second projection: *Snd* v if v is a value which is not a couple,
- application: $App(v_1, v_2)$ if v_1, v_2 are values and v_1 is not an abstraction value.

We restrict the set of questions by allowing only $M?((), v_{n-1}, \ldots, v_0)$ where M is a De Bruijn's expression of environment order at most n and $v_{n-1}, \ldots, v_0$ are values.

These definitions are justified by the following facts (left to the reader):

- values are normal forms w.r.t. $\rightarrow_C^*$ (whence the terminology at the beginning of the section),
- if $M?v$ is a state and $M?v \rightarrow_C^* s$ for some s, then s is a state.

To be able to compare combinatory computations with computations by β-reductions, we must be able to recover De Bruijn's expressions from states. This is performed by the function *REAL* defined as follows:

$$REAL(M?((), v_{n-1}, \ldots, v_0)) = M[REAL(v_0); REAL(v_1), \ldots : REAL(v_{n-1})],$$

$$REAL(\Lambda(M)((), v_{n-1}, \ldots, v_0)) = REAL(\Lambda(M)?((), v_{n-1}, \ldots, v_0)),$$

$$REAL((v_1, v_2)) = \langle REAL(v_1), REAL(v_2)\rangle,$$

$$REAL(Fst\, v) = fst(REAL(v)),$$

$$REAL(Snd\, v) = snd(REAL(v)),$$

$$REAL(App(v_1, v_2)) = S(REAL(v_1), REAL(v_2)).$$

The notation in the first line denotes the substitution: for M of environment order at most n, and closed $N_0, N_1, \ldots, N_{n-1}$ (notice that $REAL(s)$ is closed), $M[N_0; \ldots : N_{n-1}]$ is defined by

$$i![N_0: \ldots ; N_{n-1}] = N_i \quad \text{if } 0 \leqslant i \leqslant n-1,$$

$$(\lambda.M)[N_0; \ldots ; N_{n-1}) = \lambda.(M[0!; N_0; \ldots ,; N_{n-1}]),$$

the other cases are mere distribution.

Notice that if M is closed, then $REAL(M?()) = M$.

Now we have to specify the call-by-value evaluation strategy for the λ-calculus in De Bruijn's notation. We only care for closed expressions. The normal forms w.r.t. the strategy are the abstractions, the couples of normal forms, the projections of normal forms which are not couples, and the λ-calculus applications of normal forms the first of which is not an abstraction. They are called λ-calculus values, and denoted by $V, V_1 \dots$

The strategy is specified much like the combinatory strategy, by the following deductive system $\rightarrow_B^*$ ('B' stands for De Bruijn), having a unique axiom:

$$V \xrightarrow[B]{*} V$$

$$\frac{M \xrightarrow[B]{*} V_1 \quad N \xrightarrow[B]{*} V_2}{\langle M, N\rangle \xrightarrow[B]{*} \langle V_1, V_2\rangle}$$

$$\frac{M \xrightarrow[B]{*} \Lambda(M_1) \quad N \xrightarrow[B]{*} V_2}{S(M, N) \xrightarrow[B]{*} M_1[V_2]}$$

$$\frac{M \xrightarrow[B]{*} V_1 \quad N \xrightarrow[B]{*} V_2}{S(M, N) \xrightarrow[B]{*} S(V_1, V_2)}$$

$$\frac{M \xrightarrow[B]{*} \langle V_1, V_2\rangle}{fst(M) \xrightarrow[B]{*} V_1}$$

$$\frac{M \xrightarrow[B]{*} V}{fst(M) \xrightarrow[B]{*} fst(V)}$$

$$\frac{M \xrightarrow[B]{*} \langle V_1, V_2\rangle}{snd(M) \xrightarrow[B]{*} V_2}$$

$$\frac{M \xrightarrow[B]{*} V}{snd(M) \xrightarrow[B]{*} snd(V)}$$

Now we can state the second proposition of the section.

Proposition 3.2. *The following are equivalent, for any closed M:*

(1) $M \rightarrow_B^* V$,

(2) $M ?() \rightarrow_C^* v$ *and* $REAL(v) = V$.

Proof. To prove (2)⇒(1) we establish (2a)⇒(1a) where

(2a) $s \rightarrow_C^* t$,

(1a) $REAL(s) \rightarrow_B^* REAL(t)$.

The only interesting case is the first rule with S. We may suppose

$$REAL(M ?v) \xrightarrow[B]{*} REAL(\Lambda(M_1) ?((), v_1^{n-1}, \dots, v_1^0))$$

and

$$REAL(Nv) \xrightarrow[B]{*} REAL(v_2).$$

Now setting V decorated with the corresponding subscript (and superscript) for the realization of v, we have

$$REAL(M?v) \xrightarrow[B]{*} \Lambda(M_1[0!;\ V_1^0;\dots;\ V_1^{n-1}]) \quad \text{and} \quad REAL(N?v) \xrightarrow[B]{*} V_2.$$

Hence we know from the definition of $\xrightarrow[B]{*}$

$$S(REAL(M?v), REAL(N?v))$$
$$= REAL(S(M,N)?v) \xrightarrow[B]{*} (M_1[0!;\ V_1^0;\dots;\ V_1^{n-1}])[V_2]$$
$$= M_1[V_2;\ V_1^0;\dots;\ V_1^{n-1}] = REAL(M_1?((), v_1^{n-1},\dots, v_1^0, v_2))$$

(the last but one equality is easily checked by induction on M_1).

To get (2)$\Rightarrow$(1) from (2a)$\Rightarrow$(1a) we only have to check that for any v $REAL(v)$ is a value, which is obvious by definition.

To show (1)$\Rightarrow$(2) first we establish that if $REAL(s) = V$, then $s \to_C^* v$ for some v. This is non trivial only if $s = M?v$ and one proceeds by induction on M, using (2a)$\Rightarrow$(1a) for the application, while for $M = n!$ we notice that obviously $n!?((), v_n, \dots, v_0) \to_C^* v_n$. Then showing (1)$\Rightarrow$(2) reduces to (1b)$\Rightarrow$(2b) where

(1b) $M \to_B^* N$,

(2b) if $REAL(s) = M$, then $s \to_C^* t$ for some t s.t. $REAL(t) = N$ which is proved by induction on the cases of $\to_B^*$ much like (2a)$\Rightarrow$(1a). $\square$

This puts an end to our proof theoretical incursion.

4. Extending the machine to handle conditionals, recursion and lazy evaluation

We turn our attention to recursive function definitions. Here is the unavoidable factorial function:

$$\textbf{letrec}\ \text{fact}\ n = \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n * \text{fact}(n-1)\ \textbf{in}\ \text{fact}\ 1;;$$

We turn it into a λ-expression using a fixed-point functional Y (which we suppose to be a constant rather than defined, because we seek an efficient evaluation):

$$(\lambda g.g1)(Y(\lambda fn.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n * f(n-1))).$$

Or rather, to benefit from the optimization in Section 2,

$$\textbf{let}\ g = Y(\lambda fn.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n * f(n-1))\ \textbf{in}\ g1.$$

So there are two new constructions to examine.

The code for **if** M **then** N **else** P is 'push' followed by the code of M, followed by 'branch $(C1, C2)$', where $C1$, $C2$ are bodies for N, P, with the following effect:

branch: replace the term by the top of the stack, and, according to whether the term is 'true' or 'false', execute $C1$ or $C2$.

For explaining our implementation of recursion we shall start from the fixed-point operator Y of the λ-calculus, obeying the well-known rule

$$YM = M(YM) \quad \text{or} \quad [\![YM]\!] = App \circ \langle [\![M]\!], [\![YM]\!] \rangle.$$

This suggests to introduce a new unary combinator *Fix*, where $Fix(C)$ is the abbreviation of $App \circ \langle [\![Y]\!], C \rangle$. Using one or another coding of the Y combinator in the pure λ-calculus, we derive

$$(Fix) \quad Fix(C) = App \circ \langle C, Fix(C) \rangle.$$

We first concentrate on recursive functional definitions, so we can assume that the argument of *Fix* has the form $\Lambda(\Lambda(M))$. We get

$$Fix(\Lambda(\Lambda(M))) = App \circ \langle \Lambda(\Lambda(M)), Fix(\Lambda(\Lambda(M))) \rangle$$
$$= \Lambda(M) \circ \langle Id, Fix(\Lambda(\Lambda(M))) \rangle$$

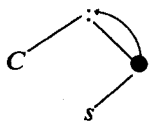
and, abbreviating $Fix(\Lambda(\Lambda(M)))$ to $F(M)$,

$$F(M) = \Lambda(M) \circ \langle Id, F(M) \rangle.$$

What should the corresponding instruction 'F' of the machine look like? If C is the code of M, the effect of $F(C)$ should be the same as executing the sequence 'push; $F(C)$; cons; (cur C)'. Let t be the result of the action of $F(C)$ on the term s; t should also be obtained using the alternative sequence of instructions, forcing

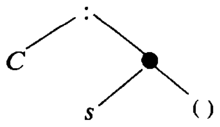
$$t = C{:}(s, t)$$

or in figure:



But such an instruction does not quite look like a real machine instruction. The usual way of winding structures is to destructively replace in a couple one of its elements by whatever you want (cf. LISP's 'rplac' instruction). Moreover this treatment is not easily extendable to simultaneous recursive definitions nor to non-functional recursive definitions.

So we decompose the effect of $F(C)$ into: first construct an object



where () is a dummy value, second construct the previous looping structure through a new instruction 'wind' which has the following effect:

wind: physically replace the right part of the top of the stack (supposed to be a couple) by the value, and remove the top of the stack.

$F(C)$ is now compiled into

push; quote (); cons; push; (cur C); wind

Summarizing, we have augmented the machine with the conditionals and recursion given in Table 6.

We run our example ('*w*','*b*' stand for 'wind' and 'branch'. *B* and *C* are the codes corresponding to **if** $n=0$ **then** 1 **else** $n*f(n-1)$ and $n*f(n-1)$).

$()$	$\langle\langle'()\rangle\langle\Lambda(B)w\rangle\langle 0!,'1\rangle App$	$[]$
$((),())$	$\Lambda(B)w\rangle\langle 0!,'1\rangle App$	$[((),());()]$
$B:((),())$	$w\rangle\langle 0!,'1\rangle App$	$[((),());()]$
$B:(s=((),B:s))$	$\rangle\langle 0!,'1\rangle App$	$[()]$
$((),B:(s=((),B:s)))$	$0!,'1\rangle App$	$[((),B:(s=((),B:s)))]$
$B:(s=((),B:s))$	$,'1\rangle App$	$[((),B:(s=((),B:s)))]$
1	$\rangle App$	$[B:(s=((),B:s))]$
$(B:(s=((),B:s)),1)$	App	$[]$
$(s=((),B:s),1)$	$\langle\langle 0!,'0\rangle=b('1,C)$	$[]$
$(s=((),B:s),1)$	$'0\rangle=b('1,C)$	$[1;(s=((),B:s),1)]$
false	$b('1,C)$	$[(s=((),B:s),1)]$
$(s=((),B:s),1)$	$\langle 0!,\langle 1!,\langle 0!,'1\rangle-\rangle App\rangle*$	$[]$
$(s=((),B:s),1)$	$\langle 1!,\langle 0!,'1\rangle-\rangle App\rangle*$	$[1]$
$(s=((),B:s),1)$	$\langle 0!,'1\rangle-\rangle App\rangle*$	$[B:(s=((),B:s));1]$
$(s=((),B:s),1)$	$'1\rangle-\rangle App\rangle*$	$[1;B:(s=((),B:s));1]$
0	$\rangle App\rangle*$	$[B:(s=((),B:s));1]$
$(s=((),B:s),0)$	$B\rangle*$	$[1]$
$(s=((),B:s),0)$	$\langle 0!,'0\rangle=b('1,C)\rangle*$	$[(s=((),B:s),0);1]$

Table 6
Conditionals and recursion

true	(branch $(C1,C2)$);C	$s.S$	s	$C1;C$	S
false	(branch $(C1,C2)$);C	$s.S$	s	$C2;C$	S
s	wind;C	$(t,()).S$	$s[(t,())\leftarrow(t,s)]$	C	S

$(0, 0)$	$= b('1, C)\rangle *$	$[(s = ((), B{:}s), 0); 1]$
$(s = ((), B{:}s), 0)$	$'1\rangle *$	$[1]$
$(1, 1)$	$*$	$[]$
1	$[]$	$[]$

(notice that $s = ((), B{:}s)$ and $((), B{:}(s = ((), B{:}s)))$ denote the same graph).

So far we have described only call by value, eager evaluation mechanisms: arguments of functions are evaluated completely, components of couples also. Call by name, lazy evaluation, as is well known, may avoid useless computations and allow the manipulation of infinite structures.

At the language level, the laziness may be
- either implicit (guaranteed by the compiler),
- or achieved by the explicit use of a 'freeze' primitive.

We follow the second choice here although ML has no such primitive presently.

As was already remarked by Plotkin [18], the most natural way of implementing call by name for functional expressions seems to introduce explicit delay, or *freeze* instructions in the call by value framework, the action of which is quite similar to the action of 'cur'.

So we add to our framework a 'freeze' instruction, acting as follows:

freeze: replace the term s by the structure $(C.s)$, where C is the code encapsulated in the freeze instruction.

$(C.s)$, which refers to Ms if M has C as code, is called a *laze*. The introduction of lazes modifies the nature of the term component of the machine: the term is not necessarily a value, but possibly contains unevaluated expressions, which have to be forced to evaluation. Thus the compiler has to insert 'unfreeze' instructions at appropriate places. A possible strategy, which we adopt here for simplicity, is to insert those instructions before the 'strict' instructions, i.e. those which cannot be executed on a laze: 'car', 'cdr', 'app', 'plus'. In this approach we have to take care of the possible need of repetitively executing 'unfreeze', yielding the following description:

unfreeze: performs no action (like 'skip' discussed above) unless the term is a laze $C.s$, in which case C is prefixed to the code (including 'unfreeze') and the term becomes s.

The repetitive nature of 'unfreeze' can be avoided by restricting appropriately the places where 'freeze' instructions are inserted (specifically in the explicit approach, 'freeze' can only appear as a component of a couple or an argument of an application). Also the non-repetitive 'unfreeze' has to be inserted differently (specifically *after* 'fst' or 'snd'). This is discussed in detail in [15, 16] (cf. also [9]).

Table 7 gives a summary for our chosen variant.

Table 7
Lazy evaluation

s	(freeze C);$C1$	S	$C.s$	$C1$	S
$C.s$	unfreeze;$C1$	S	s	C;unfreeze;$C1$	S
s	unfreeze;C	S	s	C	S

The third rule is applied if the second cannot be applied.

As an example we execute

$$\textbf{let } z=2 \textbf{ in } (\lambda x.z)(\mathit{freeze}((\lambda y.y)1))$$

(f) stands for 'freeze', and B is the code for $(\lambda y.y)1$

$()$	$\langle '2\rangle\langle \Lambda(1!), f(B)\rangle App$	$[]$
$((),2)$	$\langle \Lambda(1!), f(B)\rangle App$	$[]$
$((),2)$	$f(B)\rangle App$	$[1!:((),2)]$
$B.((),2)$	$\rangle App$	$[1!:((),2)]$
$(((),2), B.((),2)$	$1!$	$[]$
2	$[]$	$[]$

(as expected the useless computation of B has been avoided).

Our last example shows how recursion and laziness may be combined to handle recursively defined lists. We evaluate the following expression, expressed in an ML style:

$$\textbf{letrec } x=(1, \text{freeze } x) \textbf{ in } \mathit{fst}(\mathit{snd}(x))$$

The compiler translates $\mathit{fst}(M)$ by: code of M followed by 'unfreeze', followed by 'fst', thus preparing for forcing delayed evaluations. D stands for '[push; quote (); cons, push]'. We abbreviate 'freeze', 'unfreeze' into 'fr', 'unfr', and use '⟨', ',', '⟩', ''' instead of 'push', 'swap', 'cons', 'quote', to save space:

$()$	$\langle\langle '()\rangle\langle\langle '1, f(\mathit{snd})\rangle w\rangle \mathit{snd\ unf\ snd\ unf\ fst}$	$[]$
$((),())$	$\langle\langle '1, f(\mathit{snd})\rangle w\rangle \mathit{snd\ unf\ snd\ unf\ fst}$	$[()]$
$((),())$	$f(\mathit{snd})\rangle w\rangle \mathit{snd\ unf\ snd\ unf\ fst}$	$[1; ((),()); ()]$
$(1, \mathit{snd}.((),()))$	$w\rangle \mathit{snd\ unf\ snd\ unf\ fst}$	$[((),()); ()]$
$s=(1, \mathit{snd}.((), s))$	$\rangle \mathit{snd\ unf\ snd\ unf\ fst}$	$[()]$
$s=(1, \mathit{snd}.((), s))$	$\mathit{unf\ snd\ unf\ fst}$	$[]$
$s=(1, \mathit{snd}.((), s))$	$\mathit{snd\ unf\ fst}$	$[]$
$s=\mathit{snd}.((), (1, s)))$	$\mathit{unf\ fst}$	$[]$
$s=((), (1, \mathit{snd}.s))$	$\mathit{snd\ unf\ fst}$	$[]$
$s=(1, \mathit{snd}.((), s))$	fst	$[]$
1	$[]$	

5. Using the CAM to compile ML

We present here a compiler for a subset of ML [10] producing CAM code. We have decided to make this presentation completely effective by writing it in ML. ML programs and CAM programs are made into ML objects and the compiling process is made into an ML function. Moreover we also describe the execution of the CAM as an ML function.

This ML description of an ML to CAM compiler happens to be as clear and concise as it could be in any other formalism thanks to the concrete types feature that has been recently proposed by Milner [17] and added to existing implementations of ML [4, 11]. This feature enables one to describe and manipulate object languages in ML by means of their abstract syntax (concrete syntax can also be used through an interface between ML and Yacc [11]).

The ML subset we have chosen to take into account in the following description is of course the result of a compromise between significance and space. It seemed reasonable to omit concrete and abstract type declaration since they mainly concern the type-checker and not the compiler. Our most serious omissions are references and assignments and also exceptions. What remains is
- integers and booleans together with arithmetic, equality tests and conditionals,
- λ-calculus together with let and letrec constructions (abstractions and let's are allowed w.r.t. patterns).

Here is out ML subset:

```
type rec MLexp =
      mlplus| ... |mlequal|mlfst|mlsnd
      |mlint of int
      |mlbool of bool
      |mlvar of string
      |mlcond of MLexp # MLexp # MLexp
      |mlpair of MLexp # MLexp
      |mlin of MLdec # MLexp
      |mlabstr of MLpat # MLexp
      |mlapp of MLexp # MLexp
   and MLdec =
      mllet of MLpat # MLexp
      |mlletrec of MLpat # MLexp
   and MLpat =
      nullpat
      |varpat of string
      |pairpat of MLpat # MLpat;;
```

The three concrete types MLexp, MLdec and MLpat have been defined. They correspond respectively to ML expressions, ML declarations and ML patterns. Now we have to give a similar description of what CAM instructions are. It is given below together with the definition of CAM values:

```
type rec instruction = plus| . . . |eq
        |quote of value
        |fst|snd|cons|wind
        |push|swap|return|app
        |cur of code
        |branch of code # code
   and value = nullvalue
      |int of int
      |bool of bool
      |pair of value # value
      |closure of code # value
   and code = = instruction list;;
```

To make our machine more realistic compared to the description given in previous sections, we shall save return addresses (pointers to code) on the stack. So the stack elements will be either values or addresses. Also a 'return' instruction is added. Table 8 gives the modification of the CAM.

```
type stackelem = val of value
      |cod of code;;
type stack = = stackelem list;;
type config = = value # code # stack;;
```

Now we describe the CAM as an ML function Exec with type 'config → config'; defined by pattern matching on configurations:

```
let rec Exec = fun
   (pair(x, y), (fst::C), D) → Exec(x, C, D)
  |(pair(x, y), (snd::C), D) → Exec(y, C, D)
  |(x, (cons::C), (val y)::D)) → Exec(pair(y, x), C, D)
  |(x, (wind::C), ((val(pair(y, z)) as u)::D)) → (z := x); Exec(u, C, D)
  |(x, (push::C), D) → Exec(x, C, (val x)::D)
  |(x, (swap::C), ((val y)::D)) → Exec(y, C, (val x)::D)
  |(T, ((quote v)::C), D) → Exec(v, C, D)
  |(pair(closure(x, y), z), (app::C), D → Exec(pair(y, z), x, (cod C)::D)
  |((boo b), ((branch(C1, C2))::C), ((val x)::D))
    → Exec(x, (if b then C1 else C2), (cod C)::D)
```

Table 8
Saving code on the stack

$(C1:s,t)$	app.C	S	(s, t)	$C1$	$C.S$
s	return	$C.S$	s	C	S

```
|((pair(int m, int n)), (plus::C), D) → Exec(int(m+n), C, D)
|...
|((pair(int m, int n)), (eq::C), D) → Exec(bool(m = n), C, D)
|(x, ((cur C1)::C), D) → Exec(closure(C1, x), C, D)
|(x, (return::C), ((cod C')::D)) → Exec(x, C', D)
|config → config;;
```

We owe the reader an explanation for the use of an assignment in the 'wind' case. Since components of concrete objects are non-assignable in ML, we should have made value into a reference type. But this would have complicated the description of all the other cases where no destructive operations are used. So we cheated a bit.

Finally we give the compiling function. Since ML uses real identifiers and not De Bruijn codings for these, our compiling function will have to deal with the translation of a variable to some access code that will find at run time its value in the environment. Thus the compiling function has an extra parameter which is a pattern giving the position of the free variables of the expression to be compiled in the environment. The translation of a variable will be taken into account by the auxiliary function 'access' defined by:

```
let rec access id = fun
     nullpat → fail
  |(varpat x) → if x = id then [] else fail
  |(pairpat (x1, x2)) → snd::(access id x2)?(fst::(access id x1));;
```

and the compiling function is the following:

```
let rec Compile pat = fun
    (mlint n) → [quote (int n)]
  |(mlbool b) → [quote (bool b)]
  |(mlvar v) → access v pat
  |(mlcond (E1, E2, E3)) → [push]@(Compile pat E1)
                           @[branch ((Compile pat E2)@[return]
                                     (Compile pat E3)@[return])]
  |(mlpair (E1, E2)) → [push]@(Compile pat E1)@[swap]
                       @(Compile pat E2)@[cons]
  |(mlin (mllet(P, E1), E2)) → [push]@(Compile pat E1)@[cons]
                               @(Compile pat' E2)
    where pat' = pairpat (pat, P)
  |(mlin (mlletrec(P, E1), E2)) → [push; quote nullvalue; cons; push]
                                  @(Compile pat' E1)@[swap; wind]
                                  @(Compile pat' E2)
    where pat' = pairpat (pat, P)
  |(mlabstr (P, E)) → [cur ((Compile pat' E)@[return])]
    where pat' = pairpat (pat, P)
    |(mlapp (E1, E2) → if is_constant E1
                       then (Compile pat E2)@(trans_constant E1)
                       else [push]@(Compile pat E1)@[swap]
                           @(Compile pat E2)@[cons; app]
    |E → if is_constant E
         then [cur (snd::(trans_constant E))]
         else fail;;
```

where the functions 'is_constant' and 'trans_constant' are defined in the following way:

```
    let is_constant e = mem e [mlplus; mlequal; mlfst; mlsnd];;
    let trans_constant =
      fun mlplus → [plus]|mlequal → [eq]|mlfst → [fst]|mlsnd → [snd];;
```

If we want to implement a lazy variant of ML, the changes we have to make to this implementation are minor ones. Let us allow laziness by introducting in ML an explicit 'freeze' operator as well as a freeze and an unfreeze instruction in the CAM:

```
    type rec MLexp =
         . . .
        |mlfreeze of MLexp
        |. . .
    type rec instruction =
         . . .
        |freeze of instruction list
        |unfreeze
        |. . .
      and value =
         . . .
        |frozen of instruction list # value
        |. . .
```

The behaviour of the CAM for these new features is described by the following rules:

```
    let rec Exec = fun
       . . .
      |(T, ((freeze C1)::C), D) → Exec (frozen(C1, T), C, D)
      |(frozen(C1, T1), (unfreeze::C), D)
         → Exec (T1, C1, (cod (unfreeze::C))::D)
      |(T, (unfreeze::C), D) → Exec(T, C, D)
      |. . .
```

Now the Compile function must translate 'mlfreeze *E*' to 'freeze *C*' where *C* is the translation of *E* and also put unfreeze instructions where needed. These unfreeze instructions are needed when an arithmetic operation or a selector function or 'app' is to be executed but the part of the value that has to be unfrozen is not the same in the different cases. To apply 'fst' or 'snd' we must have a pair but its components can be frozen. To apply 'app' we must have a pair the first component of which is a closure. Finally, to apply an arithmetic operation, we must have a pair of integers. So the Compile and trans_constant functions are modified into:

```
let rec Compile pat E = fun
  ...
  |(mlfreeze E) → [freeze ((Compile pat E)@[return])]
  |...
  |(mlapp (E1, E2)) → if is_constant E1
                        then (Compil E2)@(trans_constant E1)
                        else [push]@(Compil E1)@[swap]
                           @(Compile pat E2)@[cons]@unfa@[app]
    where unfa = [push;snd;swap;fst;unfreeze;cons]
  |...
let trans_constant =
let unfb = [unfreeze;push;snd;unfreeze;swap;fst;unfreeze;cons]
in
  fun mplus → unfb@[plus]|mlminus → unfb@[minus]
  |mltimes → unfB@[times]|mlequal → unfb@[eq]
  |mlfst → [unfreeze;fst]|mlsnd → [unfreeze;snd];;
```

Notice that we have put 'unfreeze' instructions only where they are really needed. For example we know that the fst's and snd's that have been produced by the function 'access' to access the environment will never have to operate on frozen values and therefore we have not to accompany them with 'unfreeze'. Similarly, the instruction 'app' will never have to operate on a frozen value but always on a pair and so the only requirement is to unfreeze the left part of this pair.

6. Conclusion

The Categorical Abstract Machine arises very naturally from the semantic description of functional programming languages. It can be seen as a kind of SECD machine where the management of environments is made explicit through graph structures. It is simpler and easier to prove correct than the traditional presentations of the SECD machine. It can easily incorporate lazy evaluation. Moreover we have shown that a functional programming language such as ML compiles very naturally to CAM code.

Besides its 'theoretical' qualities, we think that the CAM can be used to build conceptually simple and reasonably fast implementations of functional languages on conventional architectures. Of course, as already mentioned in our discussion at the end of Section 2, the global environment has to be suitably represented. The simplicity of the compiler, and the existence of intuitively clear rewrite rules allow a formal approach to code optimization, including the application of curried functions to their successive arguments, the detection of closed functional expressions, for which no closures are needed, the detection of more situations where the *Beta* optimization could be used (like in "**let** $fx = M$ **in** ... fa ..."). A combination of the last optimizations allows a very efficient implementation of recursive function definitions as loops in the code rather than by creating a looping environment. These optimizations are described in [6, 14], and even more details will be found in [20].

Acknowledgment

The first ML compiler due to Lockwood Morris which was written in Lisp was much in the spirit of our approach. Indeed the striking similarity between the practical organization of the ML to Lisp compiler, explicited in [7], and the more formal categorical approach was the startpoint for the research reported here.

We would also like to mention that before the design of the CAM, Cartesian closed categories have been used for the design and the implementation of the functional programming language CDS [1, 2, 8], a language where the denotational semantics coincides with the operational one and where (the denotation of) functional expressions is evaluated piecewise on demand at the terminal, just like for expressions of basic type. We would like to thank Gérard Berry (who had the initial intuitions about CDS) and Gérard Huet: they stimulated this work by many helpful discussions. We also thank Ascander Suarez for a tremendous work on implementation and clever optimizations, as well as all the members of the Formel project at Inria and University Paris 7.

References

[1] G. Berry, Programming with concrete data structures and sequential algorithms, *Proc. ACM Conference on Functional Programming Languages and Computer Architecture 81* (1981) 49-57.

[2] G. Berry and P.-L. Curien, Theory and practice of sequential algorithms: the kernel of the applicative language CDS, in: M. Nivat and J. Reynolds, Eds., *Algebraic Methods in Semantics* (Cambridge University Press, London, 1985).

[3] N.G. De Bruijn, Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, *Indag. Math.* **34** (1972) 381-392.

[4] L. Cardelli, ML under Unix, *Polymorphism* I(3) (1983).

[5] L. Cardelli, Compiling a functional language, *Proc. ACM Conference on Lisp and Functional Programming* (1984).

[6] G. Cousineau, P-L. Curien, M. Mauny and A. Suarez, Combinateurs catégoriques et implémentation des langages fonctionnels, *Proc. Spring School on Combinators and Functional Programming Languages*, Lecture Notes in Computer Science **242** (Springer, Berlin, 1987).

[7] G. Cousineau, L'implémentation de ML en Lisp, unpublished notes.

[8] P.-L. Curien, *Categorical Combinators, Sequential Algorithms and Functional Programming* (Pitman, London, 1986).

[9] D.Friedman and D. Wise, Cons should not evaluate its arguments, *Proc. ICALP '76* (Edinburgh University Press, Edinburgh, 1976).
[10] M. Gordon, R. Milner and C. Wadsworth, *Edinburgh LCF*, Lecture Notes in Computer Science **78** (Springer, Berlin, 1979).
[11] The ML handbook, Inria Technical Report, 1984.
[12] P. Henderson, *Functional Programming: Application and Implementation* (Prentice-Hall, Englewood Cliffs, NJ, 1980).
[13] J.R. Hindley and J.P. Seldin, *Introduction to Combinators and λ-Calculus* (Cambridge University Press, London, 1986).
[14] P.J. Landin, The mechanical evaluation of expressions, *Comput. J.* **6** (1964) 308-320.
[15] M. Mauny, Compilation des langages fonctionnels dans les combinators catégoriques; Application au langage ML, Thèse de troisième cycle, Université Paris VII, 1985.
[16] M. Mauny and A. Suarez, Implementing functional languages in the categorical abstract machine, *Proc. Symposium on Lisp and Functional Programming* (1986).
[17] R. Milner, A proposal for Standard ML, *Proc. ACM Conference on Lisp and Functional Programming* (1984).
[18] G.D. Plotkin, Call-by-name, call-by-value and the λ-calculus, *Theoret. comput. Sci.* **1** (1975) 125-159.
[19] G.D. Plotkin, A structural approach to operational semantics, DAIMI FN-19, University of Aarhus, 1981.
[20] A. Suarez, Thesis, in preparation.
[21] D.A. Turner, A new implementation technique for applicative languages, *Software Practice and Experience* **9** (1979) 31-49.