Implementing Functional Languages in the Categorical Abstract Machine

Michel Mauny *Ascánder Suárez*

INRIA
Domaine de Voluceau
Rocquencourt
B.P. 105
F-78153 Le Chesnay Cedex
France

ABSTRACT

We present an implementation of the Categorical Abstract Machine (CAM) ([CouCurMa85]) leading to efficient implementations of functional languages. We define eager and lazy semantics for a functional programming language and give for each semantics a compilation to CAM code. Several significant optimizations of CAM code are described. This approach has been used to implement the ML Language.

1. Introduction

A formal approach to the compilation of typed λ-calculus led ([CouCurMa85]) to an abstract machine called Categorical Abstract Machine (CAM for short).

The ML language ([GorMilWa79]) is typechecked and statically scoped. As remarked by Milner ([Milner78]), its code is free of type tests. Hence, CAM code appears to be a good target language for compiling ML programs.

Two implementations of the ML language are developed at INRIA. An implementation of a strict ML has been realized by the second author. That implementation has been used to implement a lazy variant of ML by the first author. A remarkable point is about these two implementations is that each of them uses features of the other one.

We describe the implementation of the kernel of the ML language through the CAM. Section 2 presents its syntax and gives two different semantics to it: call-by-value (or strict evaluation) and lazy evaluation. We recall the definition of the CAM in section 3 and give in sections 4 and 5 the compilation of the languages obtained. Some optimizations are then described. Section 6 describes the expansion phase from CAM code to machine code.

2. The language

We describe the kernel of a functional language for which we will give the compilations in the following sections. This kernel is essentially composed of a typed λ-calculus with constants and a fix-point operator. Identifiers of this language are statically scoped.

The language can be seen as the functional kernel of the ML language. The different semantics presented here and their compilation have been effectively extended to a complete version of ML ([Suarez86]).

We first give a definition of the syntax of the language, and then two semantics: strict evaluation and lazy evaluation. Semantics are given by inference systems following Plotkin's formalism ([Plotkin81]).

2.1. Syntax

Its definition is given in a BNF style. We assume we have a set *Var* of variable names, and a set C of constants, being the disjoint union of sets B of basic constants (containing at least the boolean values *true* and *false*) and F of functional constants.

The set *Exp* of expressions of the language is defined by.

Exp ::= C	
| *Var*	
| (*Exp* , *Exp*) | *fst*(*Exp*) | *snd*(*Exp*)	couples and projections
| (***fun*** *Var* → *Exp*)	abstractions
| (*Exp* *Exp*)	applications
| (***let*** *Var* = *Exp* ***in*** *Exp*)	
| (***let rec*** *Var* = *Exp* ***in*** *Exp*)	
| (***if*** *Exp* ***then*** *Exp* ***else*** *Exp*)	

One recognize here a λ-calculus with explicit product and projections augmented with constants, fix-point operator and conditional.

The expression

$$\textbf{let } x = e_1 \textbf{ in } e_2$$

is equivalent to

$$(\textbf{fun } x \rightarrow e_2)\ e_1$$

Moreover, we will write

$$\textbf{let } f\ x = e_1 \textbf{ in } e_2$$

for

$$\textbf{let } f = \textbf{fun } x \rightarrow e_1 \textbf{ in } e_2$$

We will take care only of expressions member of the subset of *Exp* containing *closed* and *typable* expressions, i.e.

- containing no free variable occurrence (each occurrence of a variable x is bound to a ***fun*** x in the abstract syntax tree)
- there is a type that can be assigned to (using Milner's algorithm, for example)

Examples

$$\textbf{\textit{let rec}}\ fact\ n = \textbf{\textit{if}}\ n=0\ \textbf{\textit{then}}\ 1 \quad \textbf{\textit{else}}\ n*fact(n-1)\ \textbf{\textit{in}}\ fact$$

2.2. Semantics

We give to *Exp* expressions two different semantics: strict evaluation and lazy evaluation. Each one is defined by an inference system.

Each system is composed of a set of rules $\frac{P_1, \cdots, P_n}{C}$ where P_i and C are instances of the scheme $e_1 \longrightarrow e_2$ (e_1 and e_2 denote expressions) which must be read "e_1 *reduces to* e_2". P_i are called *premisses* and C the *conclusion*. A rule without premisses is called an *axiom*. The inference rule above must be read as "*if* P_1 *and* ... *and* P_n, *then* C".

We do not recall the definition of the substitution operation in the λ-calculus.

2.2.1. Strict evaluation

Given an application $(e_1\ e_2)$ to evaluate, strict evaluation or call-by-value essentially consists in evaluation of e_1, giving an abstraction ($\textbf{\textit{fun}}\ x \rightarrow e_1'$) (type-checking insures that we get either a functional constant or an abstraction), evaluation of e_2 and then evaluation of e_1' in which the result of evaluation of e_2 has been substituted to all free occurrences of the formal parameter x.

For the formal definition of strict evaluation we introduce a new syntactical category Val_S corresponding to the set of *strict values* of the language (the set of expressions in normal form for the inference system below).

Val_S is defined by:

$$\begin{aligned} Val_S ::= & \ C \\ & \mid (\textbf{\textit{fun}}\ Var \rightarrow Exp) \\ & \mid (Val_S, Val_S) \end{aligned}$$

We will use v, v_1,.. as variables ranging over elements of Val_S.

The axioms are:

$f\ v_1 \longrightarrow v$ if $v = f\ v_1$ (where $f \in F$) exists.
$fst(v_1,v_2) \longrightarrow v_1$
$snd(v_1,v_2) \longrightarrow v_2$
$(\textbf{\textit{fun}}\ x \rightarrow e)\ v \longrightarrow e[x \leftarrow v]$
$\textbf{\textit{let}}\ x = v\ \textbf{\textit{in}}\ e \longrightarrow e[x \leftarrow v]$
$\textbf{\textit{let rec}}\ x = v\ \textbf{\textit{in}}\ e \longrightarrow e[x \leftarrow (t = v[x \leftarrow t])]$
$\textbf{\textit{if true then}}\ e_1\ \textbf{\textit{else}}\ e_2 \longrightarrow e_1$

$$\textit{if false then } e_1 \textit{ else } e_2 \rightarrow e_2$$

The substitution operation realized while applying sixth axiom is in fact the substitution of an infinite tree to free occurrences of x. We can see this tree as a cyclic graph, the cycle being defined by the recursive equation contained in the right member of the axiom.

The reduction relation $\rightarrow_{Exp_S}$ is the smallest relation containing the above set of axioms and such that:

$$\frac{e \rightarrow_{Exp_S} e'}{fst(e) \rightarrow_{Exp_S} fst(e')} \qquad \frac{e \rightarrow_{Exp_S} e'}{snd(e) \rightarrow_{Exp_S} snd(e')}$$

$$\frac{e_1 \rightarrow_{Exp_S} e_1'}{(e_1,e_2) \rightarrow_{Exp_S} (e_1',e_2)} \qquad \frac{e_2 \rightarrow_{Exp_S} e_2'}{(v_1,e_2) \rightarrow_{Exp_S} (v_1,e_2')}$$

$$\frac{e_1 \rightarrow_{Exp_S} e_1'}{(e_1\ e_2) \rightarrow_{Exp_S} (e_1'\ e_2)} \qquad \frac{e_2 \rightarrow_{Exp_S} e_2'}{(v_1\ e_2) \rightarrow_{Exp_S} (v_1\ e_2')}$$

$$\frac{e_1 \rightarrow_{Exp_S} e_1'}{\textit{let } x = e_1 \textit{ in } e_2 \rightarrow_{Exp_S} \textit{let } x = e_1' \textit{ in } e_2}$$

$$\frac{e_1 \rightarrow_{Exp_S} e_1'}{\textit{let rec } x = e_1 \textit{ in } e_2 \rightarrow_{Exp_S} \textit{let rec } x = e_1' \textit{ in } e_2}$$

$$\frac{e \rightarrow_{Exp_S} e'}{\textit{if } e \textit{ then } e_1 \textit{ else } e_2 \rightarrow_{Exp_S} \textit{if } e' \textit{ then } e_1 \textit{ else } e_2}$$

The relation $\rightarrow_{Exp_S}$ defines a partial function. Strict evaluation of an expression e is given by the unreducible expression (if it exists) ending a chain of reductions of e.

Axioms indicate how to reduce an expression and inference rules specify in which context a reduction is possible. Axioms impose some reductions being realized only when some parts of the expression are values. Furthermore, inference rules specify the order of evaluation of applications and couples components.

2.2.2. Lazy evaluation

Unlike strict evaluation, lazy evaluation impose (β-) reducing an application without having evaluated the argument of the function. Moreover, components of structures (couples here) are evaluated only if we access them. So the result of an evaluation is (if it exists) a constant, an abstraction or a couple composed of possibly unevaluated expressions. This feature makes possible the manipulation of infinite objects, and one can get them as a result in a finite time, assuming we do not try to access all their components.

The set Val_L of *lazy values* is defined by:

$$\begin{aligned} Val_L ::= & \ C \\ & \mid (\textit{fun } Var \rightarrow Exp) \\ & \mid (Exp, Exp) \end{aligned}$$

We will use v, v_1,.. as variables ranging over elements of Val_L.

For simplicity's sake, we will consider every primitive function to be strict and its domain to be a basic type, we otherwise would need special reduction and inference rules to treat correctly each primitive.

The axioms are:

$f\ b \longrightarrow v$ if $v = f\ b$ (where $f \in F, b \in B$) exists.
$fst(e_1,e_2) \longrightarrow e_1$
$snd(e_1,e_2) \longrightarrow e_2$
$(\mathit{fun}\ x \rightarrow e_1)\ e_2 \longrightarrow e_1[x \leftarrow e_2]$
$\mathit{let}\ x = e_1\ \mathit{in}\ e_2 \longrightarrow e_2[x \leftarrow e_1]$
$\mathit{let\ rec}\ x = e_1\ \mathit{in}\ e_2 \longrightarrow e_1[x \leftarrow (t \equiv e_1[x \leftarrow t])]$
$\mathit{if\ true\ then}\ e_1\ \mathit{else}\ e_2 \longrightarrow e_1$
$\mathit{if\ false\ then}\ e_1\ \mathit{else}\ e_2 \longrightarrow e_2$

The reduction relation $\longrightarrow_{Exp_L}$ is the smallest relation containing the above set of axioms and such that:

$$\frac{e \longrightarrow_{Exp_L} e',\ f \in F}{f\ e \longrightarrow_{Exp_L} f\ e'}$$

$$\frac{e \longrightarrow_{Exp_L} e'}{fst(e) \longrightarrow_{Exp_L} fst(e')} \qquad \frac{e \longrightarrow_{Exp_L} e'}{snd(e) \longrightarrow_{Exp_L} snd(e')}$$

$$\frac{e_1 \longrightarrow_{Exp_L} e_1'}{(e_1\ e_2) \longrightarrow_{Exp_L} (e_1'\ e_2)}$$

$$\frac{e \longrightarrow_{Exp_L} e'}{\mathit{if}\ e\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \longrightarrow_{Exp_L} \mathit{if}\ e'\ \mathit{then}\ e_1\ \mathit{else}\ e_2}$$

The relation $\longrightarrow_{Exp_L}$ defines a partial function. Lazy evaluation of an expression e is given by the unreducible expression (if it exists) ending a chain of reductions of e.

In this case, axioms indicate that projections and applications are reducible without having reduced neither components of couples nor sub-expressions of applications which are in argument position. Inference rules give sub-expressions of applications that are in function position, arguments of projections and tests of conditional expressions as sole available contexts of reduction.

3. The Categorical Abstract Machine

The CAM is a virtual machine based on the translation of typed λ-calculus terms into Categorical Combinators. It can also be seen as a successor of the SECD machine ([Landin65]) in the sense that it is a "closure machine".

A first version of the CAM will be used for the strict language and some extensions will be done when compiling the lazy language.

The CAM is composed of three registers: the accumulator, the code and the stack. The set *Ins* of instructions of the machine is defined by:

$$\begin{aligned} Ins ::= & \ Ins_F \text{ (instructions associated to the primitives in } F) \\ & \mid push \mid swap \mid cons \mid car \mid cdr \mid app \\ & \mid cur(Code) \mid quote(Val) \mid branch(Code\ .\ Code) \end{aligned}$$

where *Code* denotes lists of elements of *Ins*, and *Val* is the set of values defined by:

$$\begin{aligned} Val ::= & \ C && \text{constants} \\ & \mid (Ins\ list : Env) && \text{closures} \\ & \mid (Val . Val) \end{aligned}$$

The *states* of the machine belong to the set:

$$\{Val \ \# \ Ins\ list \ \# \ (Val \cup Ins\ list)\ list\}.$$

The transitions of the machine are given by:

$\{s\ ,\ ins_f :: C\ ,\ S\} \rightarrow_{CAM} \{t\ ,\ C\ ,S\}$ where ins_f is the instruction associated to the primitive $f \in F$, s is the representation of an expected value (say v_1) for this primitive and t is the representation of $v = f\ v_1$.

$\{s\ ,\ push :: C\ ,\ S\} \rightarrow_{CAM} \{s\ ,\ C\ ,\ s :: S\}$

$\{t\ ,\ swap :: C\ ,\ s :: S\} \rightarrow_{CAM} \{s\ ,\ C\ ,\ t :: S\}$

$\{t\ ,\ cons :: C\ ,\ s :: S\} \rightarrow_{CAM} \{(s,t)\ ,\ C\ ,\ S\}$

$\{(s,t)\ ,\ car :: C\ ,\ S\} \rightarrow_{CAM} \{s\ ,\ C\ ,\ S\}$

$\{(s,t)\ ,\ cdr :: C\ ,\ S\} \rightarrow_{CAM} \{t\ ,\ C\ ,\ S\}$

$\{s\ ,\ cur(C_1) :: C\ ,\ S\} \rightarrow_{CAM} \{(C_1 : s)\ ,\ C\ ,\ S\}$

$\{((C_1 : s), t)\ ,\ app :: C\ ,\ S\} \rightarrow_{CAM} \{(s,t)\ ,\ C_1\ ,\ C :: S\}$

$\{s\ ,\ quote(t) :: C\ ,\ S\} \rightarrow_{CAM} \{t\ ,\ C\ ,\ S\}$

$\{true\ ,\ branch(C_1, C_2) :: C\ ,\ s :: S\} \rightarrow_{CAM} \{s\ ,\ C_1\ ,\ C :: S\}$

$\{false\ ,\ branch(C_1, C_2) :: C\ ,\ s :: S\} \rightarrow_{CAM} \{s\ ,\ C_2\ ,\ C :: S\}$

$\{s\ ,\ [\]\ ,\ C :: S\} \rightarrow_{CAM} \{s\ ,\ C\ ,S\}$

The final state of the machine is $\{s\ ,\ [\]\ ,\ [\]\}$ and the result of the evaluation is s.

4. Compilation of the strict language

We define a first version of the compiler for the strict language, it takes as arguments an expression and a formal environment ρ and produces CAM code. The formal environment is used to compile accesses to local values in the environment. The main difference between this compilation and the original one (presented in [CouCurMa85]) is the treatment of recursion.

Following λ-calculus, we can define recursion using the fix-point combinator Y defined as $YM = M(YM)$. Using this combinator we can rephrase every recursive declaration.

$$\textit{let rec } f\ x = e_0 \textit{ in } e \qquad (*)$$

into

$$\textit{let } f = Y(\textit{fun } f \rightarrow (\textit{fun } x \rightarrow e_0)) \textit{ in } e \qquad (**)$$

As shown in the semantic description of the languages, it is useful to represent terms

with (eventually cyclic) graphs to allow sharing when using the fix-point equation seen as a rewriting rule. When reducing the expression to which f is bound in (**), one then may use only once the fix-point rule, binding f to $(fun\ x \rightarrow e_0)$ in which free occurrences of f are bound to this expression itself. The fix-point combinator Y produces a cycle in the environment to allow, from inside an expression, the access to the expression itself. A technique inspired by the fix-point combinator is used in the lazy language to compile recursive non functional values.

In the categorical framework, environments and code may be distinguished, and loops in the environments may be transported in the code. We use here this technique: instead of having *cyclic environments*, we have *cyclic programs*. Local function calls are compiled appending the code of the function after the code of the argument. The code of local functions is stored in the formal environment as *annotations*. In the case of recursive functions, the compiler uses a recursive data structure as environment, and the generated code is represented by a cyclic graph.

The formal environments have the following structure.

$P ::= ()$	the empty environment
$\mid Var$	a variable name
$\mid (P\ ,\ Var)$	environment constructor
$\mid P\{Var = Ins\ list\}$	function annotations

The empty environment is used only to denote meaningless environments, as the initial environment of the compilation. An environment can also be a variable, but this case is not used in this version of the compiler. The environment is augmented with a new variable using the environment constructor. In $\rho_{\{f\ =\ C\}}$, the annotation $\{f\ =\ C\}$ of the formal environment ρ associates the name f to the code C. The compilation of an expression e, given by $[\![e]\!]_{()}$, is recursively defined on the structure of expressions:

$[\![f]\!]_\rho = cur(cdr\ ins_f)$ where $f \in F$
$[\![b]\!]_\rho = quote(b)$ where $b \in B$
$[\![(f\ e)]\!]_\rho = [\![e]\!]_\rho\ ins_f$ where $f \in F$
$[\![x]\!]_{\rho\{x\ =\ C\}} = C$
$[\![x]\!]_{\rho\{y\ =\ C\}} = [\![x]\!]_\rho$ if $x \neq y$
$[\![x]\!]_{(\rho,x)} = cdr$
$[\![x]\!]_{(\rho,y)} = car\ [\![x]\!]_\rho$ if $x \neq y$
$[\![(e_1\ ,\ e_2)]\!]_\rho = push\ [\![e_1]\!]_\rho\ swap\ [\![e_2]\!]_\rho\ cons$
$[\![fst(e)]\!]_\rho = [\![e]\!]_\rho\ car$
$[\![snd(e)]\!]_\rho = [\![e]\!]_\rho\ cdr$
$[\![(fun\ x \rightarrow e)]\!]_\rho = cur([\![e]\!]_{(\rho,x)})$
$[\![(e_1\ e_2)]\!]_\rho = [\![(e_1\ ,\ e_2)]\!]_\rho\ app$
$[\![let\ f\ x = e_1\ in\ e_2]\!]_\rho = [\![e_2]\!]_{\rho'}$ where $\rho' = \rho\{f = [\![(fun\ x \rightarrow e_1)]\!]_\rho\}$
$[\![let\ rec\ f\ x = e_1\ in\ e_2]\!]_\rho = [\![e_2]\!]_{\rho'}$ where $\rho' = \rho\{f = [\![(fun\ x \rightarrow e_1)]\!]_{\rho'}\}$
$[\![let\ x = e_1\ in\ e_2]\!]_\rho = push\ [\![e_1]\!]_\rho\ cons\ [\![e_2]\!]_{(\rho,x)}$
$[\![if\ e\ then\ e_1\ else\ e_2]\!]_\rho = push\ [\![e]\!]_\rho\ branch([\![e_1]\!]_\rho, [\![e_2]\!]_\rho)$

In the compilation of the *"let rec"* construct of the language, we use a cyclic formal environment: $\rho' = \rho\{f = [\![\cdots]\!]_{\rho'}\}$. For the implementation of its compiler, the strict language lacks for recursive data structures. In the real implementation of the ML language, we have extended the strict language to deal with cyclic data structures. The implementation uses some kind of *local laziness* inspired by the compiler of the lazy language described in section 5 of this paper.

An Optimizing Compiler

In the compilation presented in the last section, there is an optimization based on categorical equations (cf. [CouCurMa85]). The expression ($\mathit{let}\ x = e_1\ \mathit{in}\ e_2$) can be seen as an abbreviation for ($\mathit{fun}\ x \rightarrow e_2)\ e_1$. Knowing that the abstraction is immediately applied to an expression, a β-reduction is done at compile time We save the construction of the closure and the couple needed by the β-reduction realized when executing the *app* instruction.

Another optimization is based on the recognition of closed expressions: each time an expression is compiled, the code of each sub-expressions is surrounded by some environment saving/restoring instructions. These instructions are useful only when the last sub-expression (to be evaluated) has free variables, otherwise they can be suppressed as the execution environment will not be used during the execution of the last sub-expression. Some closed expressions are for example:

$$1+2 \qquad \mathit{let}\ x=1\ \mathit{in}\ x+1 \qquad \mathit{fun}\ x \rightarrow x*x$$

Closed abstractions or combinators play a crucial role in some optimizations proposed later in this section.

The compiler can be improved to recognize closed sub-expressions and to produce a better code in these special cases.

$[\![x]\!]_x = []$

$[\![(e_1,e_2)]\!]_\rho =$ if e_2 is closed
then $[\![e_1]\!]_\rho\ \mathit{push}\ [\![e_2]\!]_{()}\ \mathit{cons}$
else $\mathit{push}\ [\![e_1]\!]_\rho\ \mathit{swap}\ [\![e_2]\!]_\rho\ \mathit{cons}$

$[\![(\mathit{fun}\ x \rightarrow e)]\!]_\rho =$ if $(\mathit{fun}\ x \rightarrow e)$ is a combinator
then $\mathit{cur}(\mathit{cdr}\ [\![e]\!]_x)$
else $\mathit{cur}([\![e]\!]_{(\rho.x)})$

$[\![(e_1\ e_2)]\!]_\rho =$ if e_2 is closed
then $[\![e_1]\!]_\rho\ \mathit{push}\ [\![e_2]\!]_{()}\ \mathit{cons}\ \mathit{app}$
else $\mathit{push}\ [\![e_1]\!]_\rho\ \mathit{swap}\ [\![e_2]\!]_\rho\ \mathit{cons}\ \mathit{app}$

$[\![\mathit{let}\ x = e_1\ \mathit{in}\ e_2]\!]_\rho =$ if $(\mathit{fun}\ x \rightarrow e_2)$ is a combinator
then $[\![e_1]\!]_\rho\ [\![e_2]\!]_x$
else $\mathit{push}\ [\![e_1]\!]_\rho\ \mathit{cons}\ [\![e_2]\!]_{(\rho.x)}$

Each of these rules replaces its corresponding rule in the previous definition of $[\![\_]\!]$. Other rules stay unchanged.

Local Functions and Combinators

The next step is the optimization of locally defined functions (as no global environment is proposed for this mini-language all functions are in this case). The annotations of the formal environment give all the information to realize the β-

reduction of the application of a variable (annotated as a function) to an expression. For example, if we have a formal environment.

$$\rho = (..(\rho'\{f = cur(C)\}, x_{i-1}),...,x_0)$$

the compilation of the expression *f e* gives:

$$[\![f\ e]\!]_\rho = push\ car^i\ cur(C)\ swap\ [\![e]\!]_\rho\ cons\ app$$

It is easy to see in the categorical formalism that this program scheme is equivalent to:

$$[\![f\ e]\!]_\rho = push\ car^i\ swap\ [\![e]\!]_{\rho'}\ cons\ C$$

In the case of recursive functions the cycle in the formal environment produces cyclic graphs as programs as will be shown in the example at the end of this section.

When a local function is a combinator, its annotation is of the form $\{f = cur(cdr\ C')\}$, which means that the access to the environment and also the couple construction made just before the code *C* are unnecessary and the following special case can also be included:

$$[\![f\ e]\!]_\rho = [\![e]\!]_\rho\ C' \text{ if } \rho = (..(\rho'\{f = cur(cdr\ C')\}, x_i),...,x_0)$$

Example

$$[\![let\ rec\ zero\ x = if\ x=0\ then\ 0\ else\ zero(pred\ x)\ in\ zero\ 100000]\!]_{()}$$
$$= quote(100000)\ C$$
where $C = push\ [\![x=0]\!]_x\ branch(quote(0)\ .\ pred\ C)$

Other optimizations not discussed in this paper include:

- Local Variables: this optimization works once again with the "*let .. in ..*" construct of the language. Each time a local variable is declared the formal environment is augmented and accesses to all other local variables is affected. If we place the whole environment in a flat structure (instead of a list as in our machine), we obtain constant access time for local values, but we lose sharing of the execution environments between local functional values (see [Cardelli 84]). When compiling "*let* $x = e_1$ *in* e_2", building a new environment with the local value of x can be saved if there is no free occurrence of x in any abstraction sub-expression of e_2. In this case, the value of x can be pushed on the stack, giving constant access time to that value during the execution of $[\![e_2]\!]$. If x appears free in an abstraction inside e_2, it must be possible to build a closure containing its value in the environment part, hence that value must be present in the execution environment of $[\![e_2]\!]$.
- Curry Bind: the code of the application of a curryed function to all its arguments can have no overhead compared to the application of an equivalent non curryed version of this function with only one (structured) argument. This can be outlined with the following rule of compilation in which the local function has n levels of curryfication. (There is also a similar rule for the local combinators)

$$[\![f\ e_1 \cdots e_n]\!]_\rho = push\ [\![e_1]\!]_\rho\ cons\ swap\ push\ [\![e_2]\!]_\rho\ cons\ swap\ \cdots\ push\ [\![e_n]\!]_\rho\ cons\ C$$

if $\rho = (..(\rho'\{f = cur(cur\ (cur\ \cdots\ (cur\ C)\ \cdots\ ))\}, x_i)...,x_0)$

5. Compilation of the lazy language

The CAM can be easily extended to support lazy evaluation ([CouCurMa85], [Mauny85]), using the classical technique ([Plotkin75], [Henderson80]) of building new closures to represent unevaluated expressions. These new closures are composed of the code of the expression and the environment to use when executing this code. The new instruction *fre C* (freeze) builds this frozen object and the instruction *unf* (unfreeze) "thaws" the execution of the frozen code in its own environment. The instruction *upd* (update) allows the sharing of "thawed" values.
The instruction *wind* is used to build cyclic data structures as those used by the strict compiler (for the formal environments).

Frozen objects are noted as $(C.s)$ (code C, environment s). We must extend the set of values of the machine with frozen values. The new definition is:

$Val ::= ()$	empty environment
$\mid C$	constants
$\mid (ins\ list.Env)$	frozen values
$\mid (ins\ list:Env)$	closures
$\mid (Val, Val)$	

The transitions associated with these new instructions are:

$$\{s\ .\ fre(C_1)::C\ .\ S\} \rightarrow_{CAM} \{(C_1.s)\ .\ C\ .\ S\}$$
$$\{(C_1.s)\ .\ unf::C\ .\ S\} \rightarrow_{CAM} \{s\ .\ C_1\ .\ (C_1.s)::C::S\}$$
$$\{s\ .\ unf::C\ .\ S\} \rightarrow_{CAM} \{s\ .\ C\ .\ S\}$$
$$\{t\ .\ upd::C\ .\ (C_1.s)::S\} \rightarrow_{CAM} \{(C_1.s)[(C_1.s)\leftarrow t\ .\ C\ .\ S\}$$
$$\{s\ .\ wind::C\ .\ (t,())::S\} \rightarrow_{CAM} \{s[(t,())\leftarrow(t,s)]\ .\ C\ .S\}$$

The rules concerning *upd* and *wind* instructions are implemented using a physical modification of objects.

For the strict primitives some CAM code (particular to each primitive) is prefixed to prepare the arguments. Those sequences of instructions are noted $lins_f$.

The compilation of the lazy language is given by the following function:

$[\![f]\!]_\rho = cur(cdr\ unf\ lins_f)$ where $f \in F$
$[\![b]\!]_\rho = quote(b)$ where $b \in B$
$[\![(f\ e)]\!]_\rho = [\![e]\!]_\rho\ lins_f$ where $f \in F$
$[\![x]\!]_{(\rho,x)} = cdr\ unf$
$[\![x]\!]_{(\rho,y)} = car\ [\![x]\!]_\rho$ if $x \neq y$
$[\![(e_1,e_2)]\!]_\rho = push\ fre([\![e_1]\!]_\rho\ upd)\ swap\ fre([\![e_2]\!]_\rho\ upd)\ cons$
$[\![fst(e)]\!]_\rho = [\![e]\!]_\rho\ car\ unf$
$[\![snd(e)]\!]_\rho = [\![e]\!]_\rho\ cdr\ unf$
$[\![(fun\ x \rightarrow e)]\!]_\rho = cur([\![e]\!]_{(\rho,x)})$
$[\![(e_1\ e_2)]\!]_\rho = push\ [\![e_1]\!]_\rho\ swap\ fre([\![e_2]\!]_\rho\ upd)\ cons\ app$
$[\![let\ f\ x = e_1\ in\ e_2]\!]_\rho = [\![e_2]\!]_{\rho'}$ where $\rho' = \rho\{f = [\![(fun\ x \rightarrow e_1)]\!]_\rho\}$
$[\![let\ rec\ f\ x = e_1\ in\ e_2]\!]_\rho = [\![e_2]\!]_{\rho'}$ where $\rho' = \rho\{f = [\![(fun\ x \rightarrow e_1)]\!]_\rho\}$
$[\![(let\ x = e_1\ in\ e_2)]\!]_\rho = push\ fre([\![e_1]\!]_\rho\ upd)\ cons\ [\![e_2]\!]_{(\rho,x)}$
$[\![(let\ rec\ x = e_1\ in\ e_2)]\!]_\rho = push\ push\ (quote\ ())\ cons\ push$
$fre([\![e_1]\!]_{(\rho,x)}\ upd)\ wind\ cons\ [\![e_2]\!]_{(\rho,x)}$
$[\![if\ e\ then\ e_1\ else\ e_2]\!]_\rho = push\ [\![e]\!]_\rho branch([\![e_1]\!]_\rho,[\![e_2]\!]_\rho)$

The correctness proof of this compilation (without sharing) can be found in [Mauny85]. All the optimizations proposed for the strict language can be applied to this compilation.

It is possible in the lazy language to build recursive data structures; for example, the infinite list of natural numbers can be defined as:

$$\textit{let rec } Nat = 0 :: \textit{map succ } Nat$$

It is also interesting to see that the expression:

$$\textit{let rec } x = 1::x \textit{ in } x$$

produces an infinite term with a finite representation:

6. Implementation

In a real implementation of the ML language, the CAM code can be seen as an intermediate language which should be rewritten into a host machine code. An environment should also be available for the problems of communication, memory allocation/retrieving, etc.

We have decided to use the virtual machine of the LE_LISP system ([Chailloux84]): the LLM3. The LLM3 provides the possibility to easily write memory allocators/garbage collectors, in-line machine code expansion and the advantage of being machine-independent.

The toplevel of the ML system consists in evaluating global declarations. Global identifiers will be treated as constants by the compiler. The toplevel loop contains the following steps:

- parsing producing the shallow syntax of the language,
- type-checking the current ML phrase synthetising its most general type (in the sense of [Milner78]). During this step, are performed some kind of analysis such that combinators detection, etc...
- compiling the current expression into CAM code,
- rewriting the CAM code into LLM3 code. At this step, some usual optimizations are realized. They consist in detecting CAM instructions sequences which are known as being rewritten in LLM3 instructions sequences which are more compact and efficient than the ones produced by a naive rewriting process,
- execution of the program,
- updating the global environment.

The implemented version of the ML Language contains abstract/concrete types, call-by-pattern, streams and typed exceptions as in Standard ML ([Milner85]). An interface with the Yacc ([Johnson75]) parser generator is also available.

About efficiency

We give an estimation of the number of function calls per second compared whith others languages and with other implementations of ML available at INRIA. As a target machine, we use a VAX-780. For this estimation, we use in ML the following definition:

let rec fcps = fun 1 → 1 | 2 → 1 | *n* → 1 + *fcps*(*n*−1) + *fcps*(*n*−2)

SYSTEM	Function Calls / Second
Le_ML (Our version of ML)	94000
C	45200
Pascal	36600
Franz ML (ML V6.2 Compiled in Franz LISP) *	15000
LE_LISP	12500
Poly/SML	12300
Le_LML (Our lazy ML) *	7600

The estimation for the lazy version of ML is identical to that of the strict version when the optimizations are realized after strictness analysis. This is a future improvement of the lazy ML.

7. Conclusion

Traditionally closure-based implementations of functional languages have been considered inefficient because of the environments but with the optimisation of local functions and local variables, the environments contains only what is absolutely necessary.
In the implementation of ML due to Luca Cardelli ([Cardelli84]), the access time for local variables is constant but each construction of a new closure implies allocation of space proportional to the number of elements of the environment.
In our compilation the operation of closure construction takes constant time and space (and only when the applied function is not a combinator), so we encourage partial application and in particular multiple use of partially applied functions.

The theoretical bases of the Categorical Abstract Machine allow a very formal approach to the compilation of functional programming languages and to their code optimization.

References

[Cardelli84] Cardelli L., *Compiling a functional language*, Proc. ACM Conf on LISP and Functional Programming, Austin (1984).

[Chailloux85] Chailloux J., *LE_LISP Version 15, le manuel de référence*, INRIA (Feb. 85).

[Chailloux86] Chailloux J., *La machine LLM3*, INRIA internal report (Jan. 86).

[CouCurMa85] Cousineau G., Curien P.L., Mauny M., *The Categorical Abstract Machine*, IFIP Conference on Functional Programming Languages and Computer Architecture, Nancy (Sept. 1985).

[Henderson80] Henderson P., *Functional programming application and implementation*, Prentice Hall International (1980).

[GorMilWa79] Gordon M., Milner R., Wadsworth C., *Edimburgh LCF*, Lecture Notes in Computer Science, No 78, Springer-Verlag (1979).

*Garbage collecting time not included.

[Johnson75] Johnson S. C., *YACC· Yet another compiler-compiler*, CSTR 32, AT&T Bell Laboratories (1975).

[Mauny85] Mauny M., *Compilation des langages fonctionnels dans les combinateurs catégoriques. Application au langage ML*, Thèse de troisième cycle, Université Paris VII, (Sept. 85).

[Milner78] Milner R., *A theory of type polymorphism in programming*, Journal of Computer and System Science, Vol. 17,3, pp. 348-375 (Dec. 1978)

[Milner 85] Milner R., *A proposal for Standart ML*, Proc. ACM Conf on LISP and Functional Programming, Austin (1984).

[Plotkin75] Plotkin G.D., *Call-by-name, call-by-value and the lambda-calculus*, Theoretical Computer Science, Vol. 1, pp. 125-159 (1975).

[Plotkin81] Plotkin G., *A structural approach to operational semantics*, University of Aarhus, DAIMI FN-19 (1981).

[Suarez86] Suarez A., *Une implémentation de ML en ML*, Thèse à paraître.