UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1)39 63 55 11

Rapports Techniques

N°137

Programme 2

Calcul symbolique, Programmation
et Génie logiciel

INTEGRATING LAZY EVALUATION IN STRICT ML

Michel MAUNY

Juillet 1992

Intégration de l'évaluation paresseuse en ML strict

Michel Mauny

Résumé

Nous présentons trois extensions de ML à l'évaluation paresseuse, ouvrant ainsi le langage à l'écriture naturelle de certains algorithmes difficilement codables dans le langage strict, tels que ceux manipulant des structures de données potentiellement infinies. Ces extensions reposent sur l'hypothèse selon laquelle une valeur suspendue n'est référencée directement qu'au plus une seule fois. L'impact en termes d'efficacité de ces extensions sur la partie stricte du langage est prise en considération, et se trouve minimisée par la troisième extension, dans laquelle l'évaluation paresseuse ne coûte qu'aux programmes qui l'utilisent effectivement, et seulement en des occurrences précises. Ces extensions ont tour à tour été incorporées au langage Caml durant sa conception. La dernière de ces extensions est intégrée à la version distribuée du langage (V2-6.1). Enfin, nous donnons une technique permettant de compiler de façon sûre des définitions récursives arbitraires, en utilisant l'évaluation paresseuse dans les cas où la compilation en appel par valeur peut être incorrecte.

Integrating lazy evaluation in strict ML

Michel Mauny

Abstract

We present ways of extending ML with lazy evaluation, thus providing natural ways of writing algorithms which are painful to encode in classical versions of ML, such as algorithms dealing with potentially infinite data structures. These extensions are based on the assumption that any suspended value is directly referenced at most once. The runtime penalty on the strict part of the language is taken into account, and is minimized in the third extension, in which the price of lazy evaluation has to be paid only by programs that actually use lazyness, and only at specific occurrences. These extensions have been incorporated into the Caml language during its design. The last extension has been integrated into the current version of the language (V2-6.1). Finally, we give a technique allowing to compile recursive definitions in a secure way, using lazy evaluation when call-by-value compilation can be incorrect.

1

1 Introduction

The "pragmatic" approach to functional programming uses strict evaluation (call-by-value) leaving control of the evaluation order to the programmer. When evaluating an application, the expressions in function and argument positions are both evaluated before the value of the argument is passed to the function (even if this value is never used by the function). This approach enables the usage of side-effects and other non-purely functional extensions (conventional input/output, exception handling, etc).

The "purely functional" approach uses lazy evaluation, in which the argument is passed unevaluated to the function, its evaluation occurring only if the value of the argument is needed by the function. Lazy evaluation permits to execute correctly programs that would produce errors or would never terminate when submitted to strict evaluation: for instance, programs making use of infinite data structures. Non-purely functional extensions are usually incompatible with lazy evaluation because the order of evaluation may be counter-intuitive and unpredictable.

So far, very few languages have been able to handle both evaluation strategies, because of the following problems:

- at the level of language design, the primitives or syntactic constructs allowing to switch from an evaluation mode should remain usable;
- the execution model must be able to handle efficiently both evaluation strategies.

The latter condition is met by all classical execution models, even those based on graph reduction, which is usually implemented, at a lower level, following a more traditional scheme (abstract machine), e.g. the G-machine [6].

The former aspect has been discussed in [11], where lazy evaluation is the dominant evaluation strategy. As an example of language with strict semantics allowing lazy evaluation, the best known is the Scheme language [13], that provides two primitives (`force` and `delay`) to encode explicitely delayed evaluation with sharing.

In this paper, we report on different experiences which have been implemented in the Caml programming language [3, 14]. Since Caml is a strict language, we will only present ways of extending strict languages to handle lazy evaluation. The extension of lazy languages to strict evaluation is less interesting, since it implies the loss of some nice properties of lazy evaluation, and it is usually realized automatically by strictness analysis during the compilation process.

The remainder of the paper is organized as follows. We first recall the Categorical Abstract Machine [2] (CAM, for short) execution model and the compilation of a simple functional language into CAM code (section 2). The CAM execution model is presented here just to be more specific. The techniques presented in the following sections do not depend in an essential way of the CAM execution model: any other execution model could be used instead, provided it satisfies the criteria given in section 2.4.

In sections 3 and 4 we present two ways of mixing evaluation strategies: "lazy and strict modes" and "explicit delays". These two extensions rely on the hypothesis that values held in environments and in data structures can only be accessed through one, unique path. Both extensions have one important drawback: efficiency is penalized even when lazy evaluation is not used.

This disadvantage does not appear in the third extension presented in section 5: "lazy data constructors". This extension makes an hypothesis on data structure components; all existing ML

implementations meet this condition. In this extension, the strict part of the language does not incur any runtime penalty, thanks to user-defined types and strong static typing. On the other hand, the extension may seem less easy to use. We will show by examples that this extension provides laziness sufficient to program many interesting examples.

In section 6, as an application of mixing both evaluation regimes, we present the compilation of arbitrary recursive definitions in a strict language.

The author has implemented a version of the Caml language extended as presented in section 3. The current version of Caml (V2-6.1) uses the third extension (section 5) which has been implemented by P. Weis.

2 Compilation of lazy evaluation in a stack-based abstract machine

```
<Type definition>   ::=  type <Type params> <Type name> = <Type expression>
                         ...
                         and <Type params> <Type name> = <Type expression>

<Type params>   ::=                                          No parameter
                  |  <Type variable>                         A single parameter
                  |  (<Type variable>,<Type variable>,...)

<Type name>   ::=  <Identifier>

<Type expression>
              ::=  [ <Constructor> of <Type> | <Constructor> of <Type> | ... ]
                                                             Sum type expressions
                  |  {}                                      Empty product type expression
                  |  { <Label> : <Type> ; ... }              Product type expressions

<Constructor>   ::=  [A-Z][A-Za-z]*

<Label>   ::=  [A-Z][A-Za-z]*

<Identifier>   ::=  [a-z]+

<Type>   ::=  <Type variable>
            |  <Type> -> <Type>
            |  <Type arguments> <Type name>

<Type arguments>   ::=                                       No argument
                     |  <Type>                               A single argument
                     |  (<Type>,<Type>,...)

<Type variable>   ::=  'a | 'b | ...
```

Figure 1: The syntax of types and type definitions

We recall in this section the CAM execution model, and its extensions to handle lazy evaluation.

This has been presented in earlier papers [4, 2, 10, 9] where further information concerning the theoretical origins and optimizations of the CAM can be found.

In the following subsections, we present the syntax of the language: λ-calculus with constants, structured data types, pattern-matching and recursion. The CAM instructions are then given, and the compilation techniques for call-by-value and lazy semantics are presented.

2.1 The object language

We consider a subset of ML with type definitions. A type definition is of the form given in figure 1. Sum data constructors and record labels are capitalized in order to distinguish them from usual identifiers. A typical type definition is:

```
type α list = [Nil | Cons of (α, α list) pair]
and (α,β) pair = {Fst:α; Snd:β}
```

We introduced in that example the parameterized types of lists and pairs. The data types definable in the language are product types: records with 0, 1 or more fields, and sum types with two or more summands. We do not loose generality in forbidding the definition of a sum type with only one summand, since such a type can be defined as a record type with only one field. Moreover, to keep things simple, we want to distinguish easily between product and sum types.
A data structure is:

- either a (possibly empty) labeled product,
- or the application of a unary sum constructor to a value (constant data constructors may be simulated by application to the empty product).

The syntax of the expressions of the object language is given in figure 2. Expressions are built up from a set of basic constants which are assumed to be integers. Functions use pattern-matching to access the components of their argument.

2.2 CAM code and its execution

2.2.1 Instructions

The instructions will be part of a *state* of the CAM. A state possesses 3 components: the *register*, the *program counter* and the *stack pointer*. Each of them holds a pointer in a real implementation; however, for simplicity, we will describe them as containing respectively a value, a list of instructions (code), and a list of stack elements (values or code).

2.2.2 Code, values and states

Code, environments, values and stacks are described by the following productions:

<*Code*>	::=	[ ]	Empty code sequence
	|	<*Instruction*>;<*Code*>	
<*Environment*>	::=	()	Empty environment
	|	(<*Environment*>,<*Value*>)	Non-empty environment (paren. associate to the left)

```
<Expr>    ::=  <Basic constant>
             |  <Identifier>
             |  <Functional constants>
             |  {}                                   Empty product
             |  { <Label> = <Expr> ; ... }           Non empty products
             |  <Constructor> <Expr>                 Summands
             |  <Expr> <Expr>                        Applications
             |  function  <Pattern> →<Expr>          Pattern-matching abstractions
                        ...
                    |  <Pattern> →<Expr>
             |  rec <Pattern> → <Expr>               Recursive expressions

<Pattern>    ::=  <Basic constant>
                |  <Identifier>
                |  _                                 Wildcard
                |  {}                                Empty product pattern
                |  { <Label>= <Pattern> ; ... }      Non-empty product patterns
                |  <Constructor> <Pattern>           Summand patterns

<Basic constant>    ::=  <Integer>
```

Figure 2: The syntax of expressions

```
<Value>    ::=  <Basic constant>
             |  <Environment>
             |  (<Value>,...,<Value>)          Tuples
             |  <Constructor>(<Value>)         Summands
             |  <Code>:<Environment>           Functional closures
             |  <Code>*<Environment>           Suspensions

<Stack>    ::=  [ ]                           Empty stack
             |  <Value>;<Stack>
             |  <Code>;<Stack>
```

2.2.3 Operational semantics

We now give the operational semantics of the CAM. We define a function:

$$\Rightarrow : (<Value> \times <Code> \times <Stack>) \rightarrow (<Value> \times <Code> \times <Stack>)$$

The $\Rightarrow$ function is described by transition diagrams involving instructions. Transitions are divided into small sets, and an informal presentation of the instructions is given before each set of transitions.

Basic operations and stack manipulation

<*Instruction*>

::=	`Plus` | ...	Basic instructions
|	`Quote(addr)`	Loads address `addr` (used for constants and data constr.)
|	`Push`	Pushes contents of register onto the stack
|	`Pop`	Pops the top of the stack into the register
|	`Swap`	Exchanges contents of register and top of stack

The empty code acts as a `Return` instruction, and only the transition of the `Plus` instruction is given: other basic instructions behave in a similar way.

Value	Code	Stack	⇒	Value	Code	Stack
v	[ ]	$K; S$	⇒	v	K	S
m	`Plus`; K	$n; S$	⇒	$n+m$	K	S
E	`Quote`(c); K	S	⇒	c	K	S
x	`Push`; K	S	⇒	x	K	$x; S$
x	`Pop`; K	y;S	⇒	y	K	S
v	`Swap`; K	$E; S$	⇒	E	K	$v; S$

Sum data structures

<*Instruction*>

::=	...	
|	`Tag(T)`	Tags the value held in the register as being applied to the T data constructor
|	`Untag`	Untags the value held in the register
|	`UntagUnf`	Same as `UnTag`; then evaluates the result in case of suspension
|	`If(c,addr1,addr2)`	Tests if the content of the register is tagged with T (when `c` is a tag T) or checks basic constant equality when `c` is a basic constant, then gives control to `addr1` if so or to `addr2` otherwise

Value	Code	Stack	⇒	Value	Code	Stack
v	`Tag`(T); K	S	⇒	T(v)	K	S
T(v)	`Untag`; K	S	⇒	v	K	S
$V = \mathrm{T}(K_1 * E)$	`UntagUnf`; K	S	⇒	E	K_1	$V; K; S$
T(v)	`UntagUnf`; K	S	⇒	v	K	S
T(v)	`If`(T, K_1, K_2); K	$E; S$	⇒	E	K_1	$K; S$
$\mathrm{T}_1(v)$ where $\mathrm{T}_1 \neq \mathrm{T}$	`If`(T, K_1, K_2); K	$E; S$	⇒	E	K_2	$K; S$
c	`If`(c, K_1, K_2); K	$E; S$	⇒	E	K_1	$K; S$
c_1 where $c_1 \neq c$	`If`(c, K_1, K_2); K	$E; S$	⇒	E	K_2	$K; S$

Tuples

<*Instruction*>
::= ...
| `Pack(n)` Builds a `n`-uple with the content of the register and the `n`-1 topmost stack elements
| `Nth(n)` Loads the contents of the `n`th component of the value held in the register
| `NthUnf(n)` Loads the contents of the `n`th component of the value held in the register, then evaluates it if it is a suspension

Value	Code	Stack	⇒	Value	Code	Stack
v_n	$\texttt{Pack}(n);K$	$v_{n-1};\ldots;v_1;S$	⇒	$(v_1,\ldots,v_n)$	K	S
$(v_1,\ldots,v_n,\ldots)$	$\texttt{Nth}(n);K$	S	⇒	v_n	K	S
$V=(v_1,\ldots,K_1*E,\ldots)$	$\texttt{NthUnf}(n);K$	S	⇒	E	K_1	$V;K;S$
$(v_1,\ldots,v_n,\ldots)$	$\texttt{NthUnf}(n);K$	S	⇒	v_n	K	S

Function closures

<*Instruction*>
::= ...
| `Cur(addr)` Builds a function closure composed of the code address `addr` and the environment held in the register
| `App` Applies a closure (top of stack) to an arg. (register)
| `Mkenv` Builds a new environment from the current one (register) and the value held on top of stack (popped)

Value	Code	Stack	⇒	Value	Code	Stack
E	$\texttt{Cur}(K_1);K$	S	⇒	$K_1:E$	K	S
v	$\texttt{App};K_1$	$(K:E);S$	⇒	(E,v)	$K;K_1$	S
v	$\texttt{Mkenv};K_1$	$E;S$	⇒	(E,v)	K_1	S

Suspensions

Suspensions are created by the `Freeze` instruction. They are evaluated by access instructions (into data structures or environments). Suspensions are written $K*E$, where K is a code address and E denotes an environment.

<*Instruction*>
::= ...
| `Freeze(addr)` Builds a suspension closure composed of `addr` and the current environment

Value	Code	Stack	⇒	Value	Code	Stack
E	$\texttt{Freeze}(K_1);K$	S	⇒	K_1*E	K	S

Environment fetching

<Instruction>

::=	...	
|	`Env(n)`	Loads the `n`th sub-environment of the current environment
|	`Val`	Loads the value in first position in the current environment
|	`ValUnf`	Loads the value in first position in the current environment and evaluates it in case of a suspension

Value	Code	Stack	$\Rightarrow$	Value	Code	Stack
$E' = (E, v)$	$\texttt{Env}(1); K$	S	$\Rightarrow$	E'	K	S
(E, v)	$\texttt{Env}(n+2); K$	S	$\Rightarrow$	E	$\texttt{Env}(n+1); K$	S
(E, v)	$\texttt{Val}; K$	S	$\Rightarrow$	v	K	S
$E'' = (E', (K_1 * E))$	$\texttt{ValUnf}; K$	S	$\Rightarrow$	E	K_1	$E''; K_1; S$
(E, v)	$\texttt{ValUnf}; K$	S	$\Rightarrow$	v	K	S

Updating

Updating instructions perform a side-effect on the value held on the top of the stack. We express such side-effects by an annotation on the arrow denoting a change of state. These annotations are written:

- either "nth$(v)(n) := v_1$", meaning that the nth component of the record v is overwritten by v_1;
- or "$v := v_1$", indicating that the values v and v_1 are physically identified (componentwise); this notation is used for updating data constructors arguments and environments.

<Instruction>

::=	...	
|	`UpdSum`	Updates the argument of the value (from a sum type) top of stack (popped) with the value in the register
|	`UpdProd(n)`	Updates the `n`th component of the data structure on top of stack (popped) with the value in the register
|	`UpdEnv`	Updates the value part of the environment held on top of stack (popped) with the value in the register
|	`Wind`	Builds a cyclic environment for recursive values: cf figure 3

Value	Code	Stack	side-effect $\Longrightarrow$	Value	Code	Stack
v	$\texttt{UpdProd}(n); K$	$v_1; S$	$\xRightarrow{\text{nth}(v_1)(n) := v}$	v	K	S
v	$\texttt{UpdSum}; K$	$v_1 = \text{T}(K_1 * E); S$	$\xRightarrow{v_1 := \text{T}(v)}$	v	K	S
v	$\texttt{UpdEnv}; K$	$E_2 = (E_1, K_1 * E); S$	$\xRightarrow{E_2 := (E_1, v)}$	v	K	S
v	$\texttt{Wind}; K$	$E_1 = (E, \Omega); S$	$\xRightarrow{E_1 := (E, v)}$	E_1	K	S

(where Ω is a special constant)

2.3 Compiling the strict language

We assume in this section call-by-value semantics for our language, and we give its compilation to CAM code under this assumption. We take the evaluation order "from left to right" for applications and data structure components. The compilation of pattern-matching is fairly naive. The reader is referred to [11], [8] and [12] for more clever compilation algorithms. We also assume that the programs being compiled:

- do not contain any free variable,
- do not contain non linear patterns: a variable cannot occur twice in the same pattern,
- contain only total functions (in particular, all matches are total: pattern-matching cannot fail),
- have been previously passed through a type-checker that has rearranged record expressions with a canonical order for labels.

We define a compilation function taking as argument a syntactic construct and a *formal environment* from which accesses to values of local variables are computed. Formal environments are defined by:

$$\begin{array}{lcll} <\textit{Formal environment}> & ::= & () & \text{Empty environment} \\ & | & (<\textit{Formal environment}>, <\textit{Pattern}>) & \end{array}$$

The syntactic category $<\textit{Pattern}>$ is defined in figure 2. The compilation of the strict language is given by the following function:

Comp: $<\textit{Expr}> \rightarrow <\textit{Formal environment}> \rightarrow <\textit{Code}>$

2.3.1 Identifiers (Access)

In the general case, the code for an identifier is an access in the environment ($\texttt{Env}(n)$), followed by an access to its value in a pattern:

$$\begin{array}{lcll} \text{Comp}\,[\![x]\!]\,((\ldots(\rho, p_n), \ldots), p_0) & = & \texttt{Val}; \text{Access}\,[\![x]\!]\,p_0 & \text{if } x \text{ occurs in } p_0 \\ & = & \texttt{Env}(n); \texttt{Val}; \text{Access}\,[\![x]\!]\,p_n & \text{if } x \text{ occurs in } p_n, \\ & & & \text{but not in } p_i, \text{ for } 0 \leq i < n \end{array}$$

The Access $[\![\,]\!]$ auxiliary function returns the code necessary to access a value in a data structure.

Access:$<\textit{Expr}> \rightarrow <\textit{Pattern}> \rightarrow <\textit{Code}>$

$$\begin{array}{ll} \text{Access}\,[\![x]\!]\,y & = [\,] \text{ if } x = y \\ \text{Access}\,[\![x]\!]\,(\text{T}(p)) & = \texttt{Untag}; \text{Access}\,[\![x]\!]\,p \\ \text{Access}\,[\![x]\!]\,\{L_1 = p_1; \ldots; L_n = p_n\} & = \texttt{Nth}(i); \text{Access}\,[\![x]\!]\,p_i \text{ where } x \text{ occurs in } p_i \end{array}$$

2.3.2 Basic constants (BaseConst)

$$\text{Comp}\,[\![b]\!]\,\rho = \texttt{Quote}(b)$$

2.3.3 Functional constants (FunConst)

The code for functional constants uses basic instructions (e.g. `Plus`), and accesses depending on how they take their arguments (curried or not):

$$\text{Comp}\,[\![f]\!]\,\rho = \texttt{Cur}(K)$$

where K is the code for f. For example, the code for the (non-curried) addition is:

$$\texttt{Cur}(\texttt{Val};\texttt{Push};\texttt{Nth}(1);\texttt{Swap};\texttt{Nth}(2);\texttt{Plus})$$

where `Plus` is the basic instruction computing the sum of the register and the topmost element of the stack.

2.3.4 Product data structures (Prod)

All components of a record are evaluated in succession, with intermediate saves and restores of the current environment onto the stack. Then a `Pack` instruction builds the record.

$$\begin{aligned}
&\text{Comp}\,[\![\{\}]\!]\,\rho = \texttt{Quote}(\{\})\\
&\text{Comp}\,[\![\{L_1 = e_1; \ldots; L_n = e_n\}]\!]\,\rho \;=\; \texttt{Push};\\
&\qquad\qquad\qquad\qquad\quad \text{Comp}\,[\![e_1]\!]\,\rho;\texttt{Swap};\texttt{Push};\\
&\qquad\qquad\qquad\qquad\quad \ldots;\\
&\qquad\qquad\qquad\qquad\quad \text{Comp}\,[\![e_{n-1}]\!]\,\rho;\texttt{Swap};\\
&\qquad\qquad\qquad\qquad\quad \text{Comp}\,[\![e_n]\!]\,\rho;\texttt{Pack}(n)
\end{aligned}$$

2.3.5 Sum data constructor applications (Sum)

A sum data constructor evaluates its argument, and then gives it the appropriate tag.

$$\text{Comp}\,[\![\text{T}(e)]\!]\,\rho = \text{Comp}\,[\![e]\!]\,\rho;\texttt{Tag}(\text{T})$$

2.3.6 Function applications (Appl)

The code for an application saves the current environment, evaluates the function part, restores the environment, evaluates the argument and then performs the application by calling the closure in a new environment.

$$\text{Comp}\,[\![e_1\;e_2]\!]\,\rho = \texttt{Push};\text{Comp}\,[\![e_1]\!]\,\rho;\texttt{Swap};\text{Comp}\,[\![e_2]\!]\,\rho;\texttt{App}$$

2.3.7 Functional abstractions (Abstr)

The code for a function builds a closure with the code for the function body and the current environment. The code for the function body is generated in a new formal environment built from the old one and the pattern of the argument. The compilation of functions uses auxiliary compilation schemes MatchCases $[\![\,]\!]$ and Match $[\![\,]\!]$ dedicated respectively to match rules and patterns.

$$\text{Comp}\,[\![\texttt{function}\;\textit{cases}]\!]\,\rho = \texttt{Cur}(\texttt{Push};\text{MatchCases}\,[\![\textit{cases}]\!]\,\rho)$$

Since pattern-matching will have to realize tests on the actual argument, the current environment (containg that argument in its first position) must be saved before testing, and restored for further tests or final execution when a match rule is chosen. This is why we `Push` the current environment before executing the function body.

2.3.8 Pattern-matching (Match)

The MatchCases $[\![\,]\!]$ scheme is defined by:

$$\text{MatchCase: } <\textit{Match Cases}> \rightarrow <\textit{Formal environment}> \rightarrow <\textit{Code}>$$

$$\begin{aligned}
&\text{MatchCases } [\![\,]\!]\,\rho = [\,] \\
&\text{MatchCases } [\![p \rightarrow e \mid \textit{cases}]\!]\,\rho = \text{Match } [\![p]\!]\,(\texttt{Val})(\texttt{Pop};\text{Comp } [\![e]\!]\,(\rho, p), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{Pop};\texttt{Push};\text{MatchCases } [\![\textit{cases}]\!]\,\rho)
\end{aligned}$$

The Match $[\![\,]\!]$ compiling function has the following signature:

$$\text{Match: } <\textit{Pattern}> \rightarrow <\textit{Code}> \rightarrow (<\textit{Code}> \times <\textit{Code}>) \rightarrow <\textit{Code}>$$

In Match $[\![P]\!]\ K\,(K_1, K_2)$, K is a piece of code (representing the accesses to data structure components) to be executed when tests have to be done. K_1 (resp. K_2) is the code to which control will be passed in case of success (resp. failure) of the matching tests. Match $[\![\,]\!]$ is defined by:

$$\begin{aligned}
&\text{Match } [\![_]\!]\,K(K_1, K_2) = K_1 \\
&\text{Match } [\![x]\!]\,K(K_1, K_2) = K_1 \\
&\text{Match } [\![\{\}]\!]\,K(K_1, K_2) = K_1 \\
&\text{Match } [\![\{L_1 = P_1; \ldots; L_n = P_n\}]\!]\,K(K_1, K_2) = \\
&\qquad \text{Match } [\![P_1]\!]\,(K;\texttt{Nth}(1)) \\
&\qquad\qquad (\texttt{Pop};\texttt{Push};\text{Match } [\![P_2]\!]\,(K;\texttt{Nth}(2)) \\
&\qquad\qquad\qquad (\quad \ldots, \\
&\qquad\qquad\qquad\ K_2), \\
&\qquad\qquad K_2) \\
&\text{Match } [\![\mathrm{T}(p)]\!]\,K(K_1, K_2) = \texttt{Push}; K; \texttt{If}(\mathrm{T}, \text{Match } [\![p]\!]\,(K;\texttt{Untag})(K_1, K_2), K_2) \\
&\text{Match } [\![b]\!]\,K(K_1, K_2) = \texttt{Push}; K; \texttt{If}(b, K_1, K_2)
\end{aligned}$$

2.3.9 Recursion in the strict language (Rec)

The code for a recursive expression puts the special value Ω in the current environment (`Mkenv`) and evaluates the expression in this new environment. Once the expression is evaluated, the value Ω is physically replaced by the value of the expression (`Wind`). Finally, this value is returned (`Val`). The code associated to the recursive construct `rec` $p \rightarrow e$ is:

$$\text{Comp } [\![\texttt{rec } p \rightarrow e]\!]\,\rho = \texttt{Push};\texttt{Quote}(\Omega);\texttt{Mkenv};\text{Comp } [\![e]\!]\,(\rho, p);\texttt{Wind};\texttt{Val}$$

The effect of the `Wind` instruction can be seen more precisely on figure 3. Recursive expressions must be limited in order to avoid execution errors. Given a recursive expression `rec` $p \rightarrow e$, during evaluation of e, the pattern p is bound to the value Ω. Once e has been evaluated to a value v, then the occurrence of Ω gets replaced (in the environment) by v.

Therefore, during the evaluation of e, if an access to Ω is attempted, an execution error should occur. This can be the case if, for example, some variable occurring in p occurs in e outside of a functional abstraction.

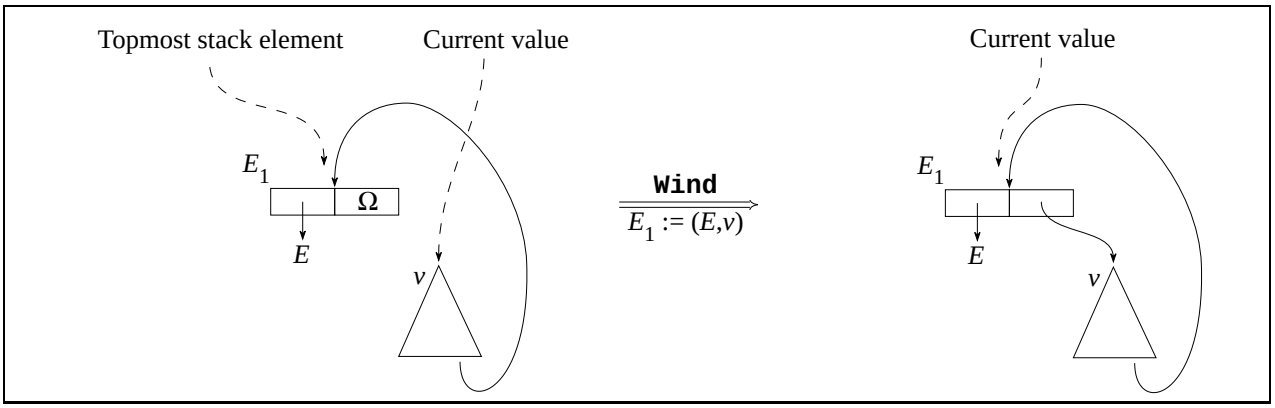

Figure 3: The effect of the `Wind` instruction

The classical way of avoiding this problem, is to restrict the body of a recursive definition to be an immediate abstraction. More precisely, such definitions are restricted to have the following shape:

`rec` $p \rightarrow e$

where p matches syntactically e, and the subexpressions of e corresponding to a variable of p are functional abstractions (recall that matching with p cannot fail, since we supposed all matches to be complete). For example:

```
rec {Fst=even; Snd=odd} → {Fst = function 0 → True
                                  | n → odd(- n 1);
                           Snd = function 0 → False
                                  | n → even(- n 1)}
```

which would written with syntactic sugar as:

```
let rec even = function 0 → True | n → odd(n-1)
and      odd = function 0 → False | n → even(n-1)
```

Of course, less severe restrictions are possible, at the price of an analysis of the recursive definition. However, some programs need to be rejected, since any access of the recursively defined value is dangerous before the updating.

In fact, these restrictions consist in allowing recursive variables only at "secure" occurrences: i.e. occurrences that statically guarantee that the value of recursive variables will not be needed before creating the cyclic structure. In a purely strict language, only abstractions protect such occurrences, and according to the analysis done by the compiler, more or less programs are accepted.

The resulting restrictions may be alleviated (and even completely avoided) by using lazy evaluation, which can protect occurrences of recursive variables. This will be considered in sections 5 and 6.

2.4 Compiling the lazy language

The experiences related in this paper do not rely on the particular execution model represented by the CAM, but only on the following assumption:

There exists at most one pointer to a suspension.

The reason is that we update suspensions by changing the "father" pointer (from the suspension to its value). Having several pointers to a suspension would result in a loss of sharing, since only

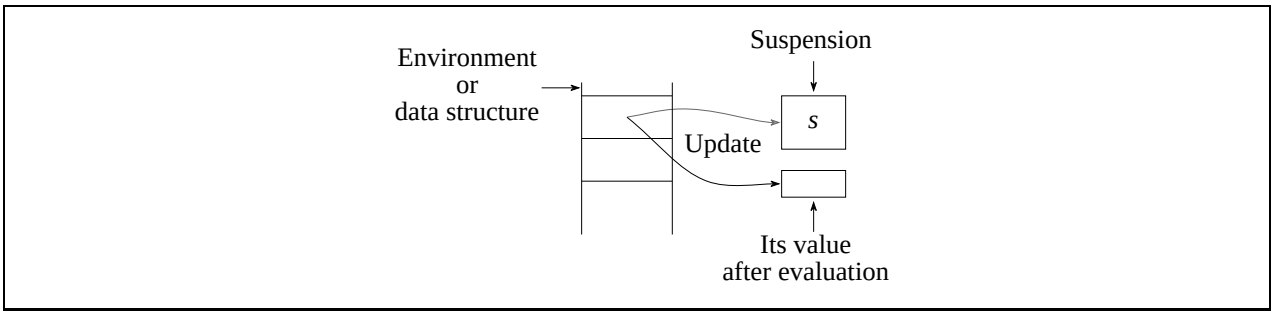

Figure 4: Updating a suspension

one of these pointers would be updated. In figure 4, we can see that updating the father pointer of a suspension preserves sharing of evaluations if the suspension s is pointed to only by this father. In our execution model, this hypothesis holds for values in environments (linked lists) and data structure components. We obtain the following rules:

Delaying is only permitted on expressions that are:

- *either in argument position of an application,*
- *or (textually) as an immediate sub-component of a data structure.*

The CAM representation of environments and the representation of functional and lazy closures (with shared environments) allow the presence of suspensions inside the environments, because all accesses to a suspension will follow the same pointer.

For the same reason, given a value component of a data structure, all accesses to this value will go through the data structure itself.

The compilation scheme of the lazy language shares some rules with the one of the strict language. We have to introduce special compilation rules only for:

- variable and data structure accesses,
- function application,
- construction of data structures,
- recursion.

2.4.1 Identifiers (AccessUnf)

`Unf` instructions are added anywhere suspensions can appear.

$$\begin{array}{lcll} \text{Comp}\,[\![x]\!]\,((\ldots(\rho,p_n),\ldots),p_0) & = & \texttt{ValUnf};\text{Access}\,[\![x]\!]\,p_0 & \text{if } x \text{ occurs in } p_0 \\ & = & \texttt{Env}(n);\texttt{ValUnf};\text{Access}\,[\![x]\!]\,p_n & \text{if } x \text{ occurs in } p_n, \\ & & & \text{but not in } p_i, \text{ for } 0 \leq i < n \end{array}$$

$$\begin{array}{ll} \text{Access}\,[\![x]\!]\,y & = [\,] \text{ if } x = y \\ \text{Access}\,[\![x]\!]\,(\text{T}(p)) & = \texttt{UntagUnf};\text{Access}\,[\![x]\!]\,p \\ \text{Access}\,[\![x]\!]\,\{L_1 = p_1;\ldots;L_n = p_n\} & = \texttt{NthUnf}(i);\text{Access}\,[\![x]\!]\,p_i \text{ if } x \text{ occurs in } p_i \end{array}$$

2.4.2 Functional constants (FunConstUnf)

All accesses into the environments and structured arguments are followed by an evaluation. The code for the addition is now:

$$\texttt{Cur}(\texttt{ValUnf}; \texttt{Push}; \texttt{NthUnf}(1); \texttt{Swap}; \texttt{NthUnf}(2); \texttt{Plus})$$

2.4.3 Product data structures (LazyProd)

A product data structure is composed of suspensions created by the `Freeze` instruction. The code argument of `Freeze` always end with an instruction updating the suspension when evaluated. Updating instructions are $\texttt{UpdProd}(n)$ for the nth component of a record, `UpdEnv` for a value in the environment and `UpdSum` for sum data constructor arguments.

$$\begin{aligned}
&\text{Comp}\,[\![\{\}]\!]\,\rho = \texttt{Quote}(\{\}) \\
&\text{Comp}\,[\![\{L_1 = e_1; \ldots; L_n = e_n\}]\!]\,\rho \;=\; \texttt{Push}; \\
&\qquad\qquad \texttt{Freeze}(\text{Comp}\,[\![e_1]\!]\,\rho; \texttt{UpdProd}(1)); \texttt{Swap}; \texttt{Push} \\
&\qquad\qquad \ldots; \\
&\qquad\qquad \texttt{Freeze}(\text{Comp}\,[\![e_{n-1}]\!]\,\rho; \texttt{UpdProd}(n-1)); \texttt{Swap}; \\
&\qquad\qquad \texttt{Freeze}(\text{Comp}\,[\![e_n]\!]\,\rho; \texttt{UpdProd}(n)); \texttt{Pack}(n)
\end{aligned}$$

2.4.4 Sum data constructor applications (LazySum)

Sum injections are lazy:

$$\text{Comp}\,[\![\text{T}(e)]\!]\,\rho = \texttt{Freeze}(\text{Comp}\,[\![e]\!]\,\rho; \texttt{UpdSum}); \texttt{Tag}(\text{T})$$

2.4.5 Expression applications (LazyAppl)

Arguments are suspended. If the function needs the value of its argument, the argument will be evaluated when it is accessed for the first time.

$$\text{Comp}\,[\![e_1\ e_2]\!]\,\rho = \texttt{Push}; \text{Comp}\,[\![e_1]\!]\,\rho; \texttt{Swap}; \texttt{Freeze}(\text{Comp}\,[\![e_2]\!]\,\rho; \texttt{UpdEnv}); \texttt{App}$$

2.4.6 Pattern-matching (MatchUnf)

Pattern-matching is modified in order to evaluate suspensions as they are accessed:

$$\begin{aligned}
&\text{MatchCases}\,[\![p \to e \mid \mathit{cases}]\!]\,\rho = \text{Match}\,[\![p]\!]\,(\texttt{ValUnf})(\texttt{Pop}; \text{Comp}\,[\![e]\!]\,(\rho, p), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{Pop}; \texttt{Push}; \text{MatchCases}\,[\![\mathit{cases}]\!]\,\rho)
\end{aligned}$$

$$\begin{aligned}
&\text{Match}\,[\![\{L_1 = P_1; \ldots; L_n = P_n\}]\!]\,K(K_1, K_2) = \\
&\qquad \text{Match}\,[\![P_1]\!]\,(K; \texttt{NthUnf}(1)) \\
&\qquad\qquad\qquad (\texttt{Pop}; \texttt{Push}; \text{Match}\,[\![P_2]\!]\,(K; \texttt{NthUnf}(2)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\qquad \ldots, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\ K_2), \\
&\qquad\qquad\qquad K_2) \\
&\text{Match}\,[\![\text{T}(p)]\!]\,K(K_1, K_2) = \texttt{Push}; K; \texttt{If}(\text{T}, \text{Match}\,[\![p]\!]\,(K; \texttt{UntagUnf})(K_1, K_2), K_2)
\end{aligned}$$

2.4.7 Recursion in the lazy language (LazyRec)

The code associated to the recursive construct `rec` $p \to e$ is:

$$\text{Comp}\,[\![\texttt{rec}\ \ p \to e]\!]\,\rho = \texttt{Push}; \texttt{Quote}(\Omega); \texttt{Mkenv}; \texttt{Freeze}(\text{Comp}\,[\![e]\!]\,(\rho, p)); \texttt{Wind}; \texttt{ValUnf}$$

The evaluation of e is delayed until the last instruction `ValUnf`, which will evaluate (lazily) the recursive value. We are sure that all accesses to values of recursive variables will be realized after the updating. We thus have a secure compilation of recursively defined values: laziness generalizes the effect of functional abstraction.

2.4.8 Loop detection

It is possible to detect at runtime some loops in recursive programs using laziness. When the evaluation of a recursively defined value requires its own value *before* the updating occurs, we are in presence of an unavoidable loop. The detection of such programs is done by changing the effect of `Unf` instructions.

Before pushing the suspension on top of the stack, `Unf` instructions can update it with an empty environment and a code address always failing. The original code and environment are no longer needed since they are respectively in the register and the program counter. If the evaluation of the suspension terminates, then the reference to the suspension will be updated, freeing the "always failing" suspension.

This simple modification has shown to be quite useful, particularly in recursive definitions involving pattern-matching, since matching has a strict semantics due to runtime tests.

3 Lazy and strict modes

The first extension presented below permits to arbitrarily mix lazy avaluation and strict evaluation. The syntax of the language is extended with two keywords, `lazy` and `strict`, which indicate changes of "compilation mode". Expressions under each of these keywords will be compiled with lazy semantics (in the case of `lazy`) or strict semantics (in the case of `strict`).

3.1 Syntax extension

We extend the syntax of the language given in figure 2 by the two following rules:

<*Expr*>	::=	...	
	|	`lazy` <*Expr*>	Submitted to lazy compilation
	|	`strict` <*Expr*>	Submitted to strict compilation

These extensions can be seen as explicit switches to the lazy (resp. strict) compilation scheme.

3.2 Compilation

The compilation of this extension uses the set of rules given for the lazy language, but suppresses the `Freeze` instructions for subexpressions of a `strict` construct. In other words, the strict part of the language uses `Unf` instructions and no `Freeze`, while the lazy part uses both.

Two cases have to be considered in order to describe the compilation of this extended language. We will say that a subexpression is:

- in *strict mode* when the closest annotation (going from the root of the subexpression upwards to the root of the whole expression) is `strict`;
- in *lazy mode* when the closest annotation is `lazy`.

We also assume given a default mode, for parts of programs that are not in the scope of an annotation.

3.2.1 Strict mode

The strict mode makes use of the following set of rules:

- (AccessUnf) [2.4.1]: access to the value of a variable provokes its evaluation;
- (BaseConst) [2.3.2];
- (FunConstUnf) [2.4.2]: functional constants also have to evaluate their strict arguments;
- (Prod) [2.3.4]: products are strict;
- (Sum) [2.3.5]: sums are strict;
- (Appl) [2.3.6]: the argument of an application is immediately computed;
- (Abstr) [2.3.7];
- (MatchUnf) [2.4.6]: matching provokes evaluation when tests are performed;
- (LazyRec) [2.4.7]: we use the lazy recursion scheme since it is more powerful than (Rec) [2.3.9]. However, (Rec) [2.3.9] is also acceptable.

3.2.2 Lazy mode

The lazy mode is managed with the "lazy" rules, as in a lazy language: values of variables are evaluated as they are accessed (i.e. when needed), and the evaluation of arguments and data structures components is delayed.

- (AccessUnf) [2.4.1]
- (BaseConst) [2.3.2]
- (FunConstUnf) [2.4.2]
- (LazyProd) [2.4.3]
- (LazySum) [2.4.4]
- (LazyAppl) [2.4.5]
- (Abstr) [2.3.7]
- (MatchUnf) [2.4.6]
- (LazyRec) [2.4.7]

3.3 Examples

As an example, we give the code for building the infinite stream of Fibonacci numbers. We use a sugared version of the language. Its translation into the core language should be obvious.

First, we introduce the type of infinite streams:

```
type α stream = {Head:α; Tail: α stream}
```

Although this is unimportant here, the default annotation is supposed to be `strict`. Let us recall that these annotations have no effect on type definitions.

The definition of the `fibs` infinite stream uses a recursive auxiliary function `f`. Its body is protected from looping by a `lazy` annotation.

```
let fibs =
    let rec f = function a → function b → lazy {Head=a; Tail=f a (a+b)}
    in f 0 1
```

The laziness necessary for this example to work concerns the stream data structure and more specifically the tail of streams.

Another interesting example involves laziness of function application. The following example is borrowed from [7]. The goal is to compute in one pass the minimum of the leaves of binary trees of integers. The type of pairs could be defined as:

```
type (α,β) pair = {Fst:α; Snd:β}
```

We add more syntactic sugar, writing the `pair` type constructor with an infix `*`, pairs with parentheses and `fst`, `snd` the two projections.

```
type α tree = [Leaf of α | Node of (α tree * α tree)]
```

We suppose the existence of a function `min: int → int → int` returning the minimum of its arguments.

```
let minimum = function t →
    let rec traverse m =
                function Node(t1,t2) →
                            let min_tree1 = lazy(traverse m) t1
                            and min_tree2 = lazy(traverse m) t2
                            in lazy (min (fst min_tree1) (fst min_tree2),
                                     Node(snd min_tree1, snd min_tree2))
                        | Leaf n → lazy (n, Leaf m) in
    let rec min_tree = lazy(traverse (fst min_tree) t
    in min_tree;;
```

Here is an example of execution:

```
minimum (Node (Node (Leaf 5, Leaf 3), Node (Leaf 3, Leaf 4)))
=> (3, Node (Node (Leaf 3, Leaf 3), Node ( Leaf 3, Leaf 3)))
```

Here, both forms of laziness are crucial. The recursive definition of `min_tree` evaluates correctly only if `traverse` is lazy in its first argument, and if pairs in which results are returned are lazy in their components. For `traverse` to be lazy in its first argument, we have to enclose all calls to `traverse` by a `lazy` annotation (remember that this makes argument passing to be lazy, and thus the first argument of `traverse` to remain unevaluated). We also have to annotate pairs by `lazy` for their components to remain unevaluated until needed.

3.4 Discussion

This extension has the disadvantage of not being intuitive: the programmer's intuition is expressed more in terms of *delay as much as possible the evaluation of this expression* than in terms of compiling

modes. This can be seen in the `fibs` example where, in order to delay the evaluation of `f a (a+b)`, we have to enclose the whole record expression in a `lazy` annotation. If we write:

```
{Head=a; Tail=lazy(f a (a+b))}
```

instead of:

```
lazy {Head=a; Tail=f a (a+b)}
```

then only the evaluation of `a` and `a+b` would be delayed: the recursive call to `f` would still be needed, since the `Tail` field is strict. However, writing:

```
lazy {Head=a; Tail=strict(f a (a+b))}
```

would still produce a terminating program: the recursive call to `f` would be delayed, and the values of `a` and `a+b` would be computed before `f` is called, but only when the `Tail` field is needed.

The extension proposed in the following section is closer to the programmer's intuition.

4 Explicit delays

This extension is basically the same as the one realized in the Scheme language where lazy evaluation can be encoded by using special functions `delay` and `force`. The idea is to rely on the fact that there is no evaluation of function bodies unless they are applied: this is a kind of laziness present even in a strict language (cf. sections 2.3.9 and 6 where this laziness is used in presence of recursion). A delayed expression becomes the body of an abstraction with a useless argument (usually the empty record), and the effect of the first `force` is to apply this abstraction to the empty record, to update the "suspension" with its value, and to return this value. Subsequent `force` will only return this value.

This extension is easily encoded in ML with even more security, thanks to the type system. However, the weakness of ML in typechecking polymorphic references prohibits polymorphic suspended values, and makes `delay` non-definable. It has to be a primitive operation. Moreover, the abstractions encoding delayed values are unnatural.

In the extension that we propose here, we need only to extend the language with a `delay` keyword. The effect of `force` is obtained automatically by the compiler (however, too many `force` are produced and clever static analyses are necessary to remedy this problem).

In order to agree with the execution model, valid occurrences of explicit delays are:

- delay of the argument part of an application;
- delay of a data structure component (sum or product).

For example, we cannot delay the evaluation of the expression in function position. We cannot delay an already delayed expression.

4.1 Syntax extension

We replace the syntactic rules for applications, data constructor application and product building given in figure 2 by the following three rules:

```
<Expr>   ::=   ...
             |   { <Label> = <DExpr> ; ... }
             |   <Expr> <DExpr>
             |   <Constructor> <Dexpr>
```

The syntactic category <*DExpr*> (possibly delayed expressions) is defined by:

$<DExpr>$ ::= `delay` $<Expr>$
| $<Expr>$

Intuitively, the effect of the `delay` $<Expr>$ construct is to delay as much as possible the evaluation of $<Expr>$. The evaluation of delayed expressions will occur as usual, i.e. when accessing them.

4.2 Compilation

The compilation of this extended language is the same as for the strict part of the previous extension (the set of rules given in section 3.2.1). In other words we force evaluation when accessing the environment or data structure components. The code for a `delay` expression is:

$$\text{Comp}\,[\![\texttt{delay}\ e]\!]\,\rho = \texttt{Freeze}(\text{Comp}\,[\![e]\!]\,\rho; \mathit{update})$$

where *update* is an update instruction chosen in accordance with the context: `UpdEnv` when e is in argument position in an application, `UpdSum` in case of a data constructor argument, or `UpdProd`(n) if e is the nth component of a record expression.

4.3 Examples

We translate here the previous two examples from section 3.3. The infinite stream of Fibonacci numbers is easily translated to:

```
let fibs =
    let rec f = function a → function b → {Head=delay a; Tail=delay(f a (a+b))}
    in f 0 1
```

The occurrences of `delay` protect the stream components from being evaluated unless they are needed. The example could be simplified into:

```
let fibs =
    let rec f = function a → function b → {Head=a; Tail=delay(f a (a+b))}
    in f 0 1
```

Now, the stream is lazy only in its tail, which is exactly what is needed.

The example of computing the minimum value in a tree translates to:

```
let minimum = function t →
    let rec traverse m =
                function Node(t1,t2) →
                            let min_tree1 = traverse (delay m) t1
                            and min_tree2 = traverse (delay m) t2
                            in  (delay(min (fst min_tree1) (snd min_tree2)),
                                 delay(Node(snd min_tree1, snd min_tree2)))
                        | Leaf n → delay (n, Leaf m) in
    let rec min_tree = traverse (delay (fst min_tree)) t
    in min_tree;;
```

Delay annotations have been inserted where laziness is required.

4.4 Discussion

This extension seems much more practical than the first one. However, a serious runtime penalty remains: any access to a value in the environment or to a component of a data structure has to check whether it is or not a suspension. This penalty is unacceptable when doing, say, arithmetic

operations where no laziness is involved. We would need an analysis detecting useless occurrences of these tests. Instead of studying further this extension to try to reduce this penalty, we go to the next extension which takes advantage of the type system.

5 Lazy data constructors

The last extension is the latest one integrated to the Caml language. We believe that this extension is minimal in the sense that it preserves the efficiency of the strict sublanguage. The ML type system enables to know exactly when accesses to possibly delayed expressions occur, and the type-checker can transmit these informations to the compiler.

We only take advantage here of the fact that the "single pointer assumption" is correct for data structures components. No assumption is made on environments. This extension is thus well-adapted to many languages: this assumption is valid in all current implementations of ML; while the representation of environments differs greatly from one implementation to another.

This extension consists in laziness annotations in type definitions. We annotate some sum data constructors or records labels as delaying the evaluation of their argument. Its main advantage is that we know at compile-time where the price of lazy evaluation has to be paid (accesses to annotated fields/constructors argument). Because of our compilation of pattern-matching, evaluation of suspended components will occur only if they are effectively used. We thus have more laziness than some other execution models, where, during a pattern-matching, data structure components are pushed onto the stack, invalidating the "single pointer assumption". In such cases, evaluation should occur *before* pushing the components onto the stack.

The other advantage of this extension is that annotations appear only in the type definition, keeping the source code readable.

5.1 Syntax extensions

We only extend the syntax of type definitions given in figure 1. We replace the <*Type expression*> syntactic category by the following one:

```
<Type expression>
                ::= [ <AConstructor> of <Type> | <AConstructor> of <Type> | ...]
                  | {}
                  | { <ALabel> : <Type> ; ... }

<ALabel>    ::=   lazy <Label>
                | <Label>

<AConstructor>   ::=   lazy <Constructor>
                     | <Constructor>
```

The category <*AConstructor*> (resp. <*ALabel*>) should be read as <*AnnotatedConstructor*> (resp. <*AnnotatedLabel*>).

5.2 Compilation

In order to compile this extended language, we will assume that the typechecker has annotated the abstract syntax trees in the following way:

- all record labels and sum data constructors occurring in the program are annotated with one of the symbols *strict* or *lazy*.

We will write these annotations as superscripts. We only present here the compilation rules that are concerned by these annotations. The rules that are not shown are taken from the rules for the strict language (section 2.3).

5.2.1 Identifiers (AnnotAccess)

$$\begin{array}{ll}
\text{Access}\,[\![x]\!]\,y & = [\,] \text{ if } x = y \\
\text{Access}\,[\![x]\!]\,(\mathrm{T}^{\texttt{strict}}(p)) & = \texttt{Untag};\text{Access}\,[\![x]\!]\,p \\
\text{Access}\,[\![x]\!]\,(\mathrm{T}^{\texttt{lazy}}(p)) & = \texttt{UntagUnf};\text{Access}\,[\![x]\!]\,p \\
\text{Access}\,[\![x]\!]\,\{L_1^{a_1} = p_1; \ldots; L_i^{\texttt{strict}} = p_n; \ldots; L_n^{a_n} = p_n\} & = \texttt{Nth}(i);\text{Access}\,[\![x]\!]\,p_i \\
\quad \text{where } x \text{ occurs in } p_i & \\
\text{Access}\,[\![x]\!]\,\{L_1^{a_1} = p_1; \ldots; L_i^{\texttt{lazy}} = p_n; \ldots; L_n^{a_n} = p_n\} & = \texttt{NthUnf}(i);\text{Access}\,[\![x]\!]\,p_i \\
\quad \text{where } x \text{ occurs in } p_i &
\end{array}$$

5.2.2 Product data structures (AnnotProd)

$$\begin{array}{lll}
\text{Comp}\,[\![\{\}]\!]\,\rho = \texttt{Quote}(\{\}) & & \\
\text{Comp}\,[\![\{L_1^{a_1} = e_1; \ldots; L_n^{a_n} = e_n\}]\!]\,\rho & = & \texttt{Push}; \\
 & & \text{CompAnnot}\,[\![e_1]\!]\,a_1\ 1\ \rho; \texttt{Swap}; \texttt{Push} \\
 & & \ldots; \\
 & & \text{CompAnnot}\,[\![e_{n-1}]\!]\,a_{n-1}\ (n-1)\ \rho; \texttt{Swap}; \\
 & & \text{CompAnnot}\,[\![e_n]\!]\,a_n\ n\ \rho; \texttt{Pack}(n)
\end{array}$$

Here, CompAnnot $[\![\ ]\!]$ is the function choosing to delay the evaluation of its expression argument or not according to the annotation a_i. It takes as extra arguments the annotation, an integer and the formal environment.

$$\begin{array}{l}
\text{CompAnnot}\,[\![e]\!]\,\texttt{strict}\ n\ \rho = \text{Comp}\,[\![e]\!]\,\rho \\
\text{CompAnnot}\,[\![e]\!]\,\texttt{lazy}\ n\ \rho = \texttt{Freeze}(\text{Comp}\,[\![e]\!]\,\rho; \texttt{UpdProd}(n))
\end{array}$$

5.2.3 Sum data constructor applications (AnnotSum)

$$\begin{array}{l}
\text{Comp}\,[\![\mathrm{T}^{\texttt{strict}}(e)]\!]\,\rho = \text{Comp}\,[\![e]\!]\,\rho; \texttt{Tag}(\mathrm{T}) \\
\text{Comp}\,[\![\mathrm{T}^{\texttt{lazy}}(e)]\!]\,\rho = \texttt{Freeze}(\text{Comp}\,[\![e]\!]\,\rho; \texttt{UpdSum}); \texttt{Tag}(\mathrm{T})
\end{array}$$

5.2.4 Pattern-matching (AnnotMatch)

$$\begin{array}{l}
\text{MatchCases}\,[\![\ ]\!]\,\rho = [\,] \\
\text{MatchCases}\,[\![p \rightarrow\ e \mid \mathit{cases}]\!]\,\rho = \text{Match}\,[\![p]\!]\,(\texttt{Val})(\texttt{Pop}; \text{Comp}\,[\![e]\!]\,(\rho, p), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{Pop}; \texttt{Push}; \text{MatchCases}\,[\![\mathit{cases}]\!]\,\rho)
\end{array}$$

$$
\begin{array}{l}
\text{Match } [\![\_]\!]\, K(K_1,K_2) = K_1 \\
\text{Match } [\![x]\!]\, K(K_1,K_2) = K_1 \\
\text{Match } [\![\{\}]\!]\, K(K_1,K_2) = K_1 \\
\text{Match } [\![\{L_1^{a_1} = P_1; \ldots; L_n^{a_n} = P_n\}]\!]\, K(K_1,K_2) = \\
\qquad \text{Match } [\![P_1]\!]\, (K; \text{AnnotNth}(a_1)(1)) \\
\qquad\qquad (\texttt{Pop}; \texttt{Push}; \text{Match } [\![P_2]\!]\, (K; \text{AnnotNth}(a_2)(2)) \\
\qquad\qquad\qquad (\quad \ldots, \\
\qquad\qquad\qquad\quad K_2), \\
\qquad\qquad K_2) \\
\qquad\qquad\qquad\qquad \text{where AnnotNth}(\texttt{strict})(m) = \texttt{Nth}(m) \\
\qquad\qquad\qquad\qquad\quad \text{AnnotNth}(\texttt{lazy})(m) = \texttt{NthUnf}(m) \\
\text{Match } [\![\text{T}^{\texttt{strict}}(p)]\!]\, K(K_1,K_2) = \texttt{Push}; K; \texttt{If}(\text{T}, \text{Match } [\![p]\!]\, (K; \texttt{Untag})(K_1,K_2), K_2) \\
\text{Match } [\![\text{T}^{\texttt{lazy}}(p)]\!]\, K(K_1,K_2) = \texttt{Push}; K; \texttt{If}(\text{T}, \text{Match } [\![p]\!]\, (K; \texttt{UntagUnf})(K_1,K_2), K_2) \\
\text{Match } [\![b]\!]\, K(K_1,K_2) = \texttt{Push}; K; \texttt{If}(b, K_1, K_2)
\end{array}
$$

5.3 Examples

We translate here the previous examples from sections 3.3 and 4.3. The translation of the Fibonacci examples involves a new definition of the type `stream`

```
type α stream = {Head: α; lazy Tail: α stream}
let fibs =
    let f = function a → function b → {Head=a; Tail= f a (a+b)}
    in f 0 1
```

The example of computing minimum values of a tree involves two type definitions in order to introduce laziness at the desired places.

```
type α delay = {lazy Delay: α}
```

This type would be sufficient to encode lazy evaluation. However, in order to preserve readability, we introduce a type for lazy pairs.

```
type (α,β) lazy_pair = {lazy Fst: α; lazy Snd: β}
```

We are now ready to give the definition of the `minimum` function:

```
let minimum = function t →
    let rec traverse {Delay=m} =
                function Node(t1,t2) →
                            let {Fst=min1;Snd=tree1} = traverse {Delay=m} t1
                            and {Fst=min2;Snd=tree2} = traverse {Delay=m} t2
                            in  {Fst=min min1 min2; Snd=Node(tree1,tree2)}
                        | Leaf n →  {Fst=n; Snd=Leaf m} in
    let rec {Fst=min;Snd=tree} = traverse {Delay=min} t
    in (min,tree);;
```

5.4 Discussion

This extension only requires new type definitions for data structures that are to be evaluated lazily. Its advantage is that the strict language does not suffer from any runtime penalty. Lazy data structures are easy to design and to use, and many algorithms make use of them in a natural way. When laziness on arguments is crucial, we have to use an artificial lazy data structure (e.g. the type

`α delay` in the last example). This shows that the mechanism is powerful enough to encode a lazy ML in a strict one with lazy data structures.

6 Application to the compilation of recursion

We use the "single pointer assumption" to provide a secure compilation of recursive definitions of non-functions. In ML, recursive definitions are written as:

`let rec` p_1 `=` e_1
`and` ...
`and` p_n `=` e_n
`in` E

We translate such definitions into:

`let {#1=`p_1`; ...; #n=`p_n`} = (rec {#1=`p_1`; ...; #n=`p_n`} → {#1=`e_1`; ...; #n=`e_n`})`
`in` E

or, with no syntactic sugar at all:

`(function {#1=`p_1`; ...; #n=`p_n`} → `E`) (rec {#1=`p_1`; ...; #n=`p_n`} → {#1=`e_1`; ...; #n=`e_n`}))`

We use a family of record types in order to describe simultaneous definitions (mutually recursive, for example). Labels are consecutive numbers preceded by a "`#`" sign (e.g. `#1`). These records allow to translate any multiple definition in ML into a single definition.

The compilation of such definitions proceeds in three steps:

Step 1: simplification. We match symbolically the expressions $(e_i)_{i=1,n}$ against the patterns $(p_i)_{i=1,n}$, obtaining a set of pairs $(p'_j, e'_j)_{j=1,m}$.

Step 2: safety detection. We examine in turn each pair (p'_j, e'_j) and determine whether the evaluation of e'_j needs to be delayed or not.

Step 3: reconstruct an expression equivalent to the original one, and compile it, taking into account the annotations produced in step 2.

We now describe more precisely each step. We suppose given the expression:

`rec {#1=`p_1`; ...; #n=`p_n`} → {#1=`e_1`; ...; #n=`e_n`}`

Step 1: simplification

From the list $(p_i, e_i)_{i=1,n}$ we produce a new list using the *flatten* function defined by:

$$
\begin{array}{l}
\mathit{flatten}\ [\,] = [\,] \\
\mathit{flatten}\ (\mathtt{x}, e); L = (\mathtt{x}, e); (\mathit{flatten}\ L) \\
\mathit{flatten}\ (\_, e); L = (\_, e); (\mathit{flatten}\ L) \\
\mathit{flatten}\ (\{\}, e); L = (\{\}, e); (\mathit{flatten}\ L) \\
\mathit{flatten}\ (\{\mathtt{lab}_1{=}p_1; \ldots; \mathtt{lab}_n{=}p_n\}, \{\mathtt{lab}_1{=}e_1; \ldots; \mathtt{lab}_n = e_n\}); L \\
\qquad\qquad = \mathit{flatten}\ ((p_1, e_1); \ldots; (p_n, e_n); L) \\
\mathit{flatten}\ (\{\mathtt{lab}_1{=}p_1; \ldots; \mathtt{lab}_n{=}p_n\}, e); L = (\{\mathtt{lab}_1{=}p_1; \ldots; \mathtt{lab}_n{=}p_n\}, e); \mathit{flatten}\ L)
\end{array}
$$

Here, [] stands for the empty list, and $x; L$ represents the list whose head is x and tail is L. Each case is tried in turn, first cases having priority on last cases. Patterns are assumed to be linear (i.e. there is no multiple occurrence of the same variable). The *flatten* function is undefined for patterns like T(p). The reason is that all matches are supposed to be complete, and the `rec` construct has

only one alternative. Since sum data types must have at least two summands, an occurrence of such a pattern would contradict these hypotheses.

The result of *flatten* (e, p) is a list $(p'_j, e'_j)_{j=1,m}$. This list will be examined in order to detect non-abstractions. A dependency analysis such as the one described in [11] should take place at the end of step 1, partitioning the list of pattern-expression pairs into an ordered list of sets of properly mutually recursive patterns-expressions.

Step 2: safety detection

The detection proposed here is the simplest one: we distinguish between textual abstractions and other expressions. This analysis annotates the expressions (and the corresponding patterns) by `lazy` or `strict`. The `strict` annotation is for textual abstractions, and `lazy` for other expressions. These annotations will then be used by the compiler.

At the end of this step, we get an expression of the form:

`rec {#1`^{a_1}`=`p'_1`; ...; #m`^{a_m}`=`p'_m`} → {#1`^{a_1}`=`e'_1`; ...; #m`^{a_m}`=`e'_m`}`

where a_j = `strict` if e_j is a textual abstraction, `lazy` otherwise.

Step 3: reconstruction and compilation

In this step, we use the compilation scheme given in section 5, with rule (Rec) [2.3.9] dealing with recursion for the strict language. This compilation scheme will "`Freeze`" the expressions annotated as being `lazy` and produce access instructions terminated by "`Unf`" to access their values.

One disadvantage of this scheme is that "`Unf`" instructions can remain in the code of recursive values. However, such instructions only remain in the recursive values themselves: they can be eliminated from the enclosing expression. We illustrate this fact on a simple example (borrowed from [11]):

```
let rec x = fac z
and   fac = function 0 → 1
                   | n → n*fac(n-1)
and     z = 4
and   sum = function 0 → (function y → y)
                   | x → (function y → sum (x-1) (y+1))
in sum x z
```

which could be rewritten without syntactic sugar as:

```
(function {#1=x; #2=fac; #3=z; #4=sum} → sum x z)
        (rec {#1=x; #2=fac; #3=z; #4=sum}
             → {#1=fac z;
                #2=function 0 → 1 | n → * n (fac (- n 1));
                #3=4;
                #4=function 0 → (function y → y)
                          | x → (function y → sum (- x 1) (+ y 1))})
```

Let us look at the results of steps 1 and 2 without doing dependency analysis. We obtain the following annotations:

```
(function {#1lazy=x; #2strict=fac; #3lazy=z; #4strict=sum} → sum x z)
        (rec {#1lazy=x; #2strict=fac; #3=lazyz; #4strict=sum}
             → {#1lazy  =fac z;
                #2strict=function 0 → 1 | n → * n (fac (- n 1));
                #3lazy  =4;
```

```
                #4^strict=function 0 → (function y → y)
                                     | x → (function y → sum (- x 1) (+ y 1))})
```

Fields `#1` and `#3` are annotated as being `lazy` and fields `#2` and `#4` are annotated as `strict`. Note that field `#3` could have been annotated as being `strict` if our criterion was slightly more sophisticated.

Now, we give the previous expression to the compiler which will take into account these annotations. If we are in the context of a strict ML, with `let` construct, we may improve this result by forcing the evaluation of lazy fields of the record before giving control to the expression `sum x z`. In other words, we could transform:

```
(function {#1^lazy=x; #2^strict=fac; #3=^lazy z; #4^strict=sum} → sum x z)
```

into:

```
(function {#1=x; #2=fac; #3=z; #4=sum} → unf x; unf z; sum x z)
```

where "`;`" denotes sequential evaluation and "`unf`" is a primitive fetching the (possibly unevaluated) value of its variable argument, evaluating it, and updating the corresponding field of the record argument. In that case, we are sure that no "`Unf`" instruction remains in the body of the "`let rec`" (`sum x z` in our example). Such instructions could be present in the body of recursively defined values, and flow analysis would be necessary in order to remove them.

This treatment of recursion is orthogonal to other transformations such as dependency analysis, and can be improved by a more sophisticated detection of safe call-by-value recursion. As an example, given the following commonly used scheme of recursive definition:

```
let rec (f,g) =
        let h = e1 in (e2,e3)
```

it is safe to "extract" the binding of `h` when e_1 does not contain any free occurrence of `f` and `g` outside of the recursive definition, making the declaration of `h` local to the declaration of `f` and `g`, producing:

```
local h = e1 in
let rec (f,g) = (e2,e3)
```

allowing to simplify further the recursive construct.

In a language without user-accessible lazy data structures, it is still possible to compile correctly arbitrary recursive definitions, with laziness used as an implementation tool invisible to the programmer.

7 Conclusion

The "single pointer assumption" allows the presence of suspensions in the following occurrences (in the representation of runtime data):

- values held in environments,
- components of data structures (sum and products).

The first two extensions presented in this paper use both possibilities. The price to pay is one test for each access. The third extension rules out the first possibility and allows laziness to occur only in components of data structures. This laziness being introduced in the type definitions, programs remain readable, and the runtime penalty is minimum in the sense that the strict part of the language remains compiled as usual.

The compilation of arbitrary recursive definitions has been given as an interesting application of this work. We used a special record type in order to store mutually recursive definitions, adding `lazy` annotations to some fields in order to delay accesses to the values of recursive variables until a cyclic structure is effectively built.

8 Related work

The LCS language [1] includes a special parameterized type `α lift`, with a single data constructor `up`. Since other data constructors are strict, `up` has to be a keyword in order to receive a special treatment by the compiler. In LCS, the `lift` type admits recursion but does not admit equality. Pattern-matching with the `up` constructor is slightly restricted: since pattern-matching does not evaluate suspensions, the compiler produces a warning when a refutable pattern (such as constants or sum patterns with more that one summand) occurs under an `up` constructor.

This mechanism has the same power as our third extension: we can define the `lift` type in our extended language as a record type with a single lazy field (type `delay`, section 5.3), and the `lift` type can make lazy any sum data constructor of record field. The `lift` solution has the advantage of not adding a new mechanism to type definitions, and the disadvantage of needing explicit delays and tests (`up` and `down`). Moreover, the presence of lifted values in record fields (for example) produce an extra (allocated) indirection to values.

9 Acknowledgements

Thanks are due to A. Suárez who first noticed that there was only one pointer to a suspension in the lazy version of Caml. I am also grateful to N.K. Anil and X. Leroy who carefully read a previous version of this paper.

References

[1] B. Berthomieu. LCS Users Manual. Technical Report 91226, LAAS, Toulouse, France, 1991.

[2] G. Cousineau, P.-L. Curien, and M. Mauny. *The Categorical Abstract Machine*, pages 50–64. Number 201 in Lecture Notes in Computer Science. Springer Verlag, 1985.

[3] G. Cousineau and G. Huet. The CAML primer. Technical Report 122, INRIA, 1990.

[4] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Research Notes in Theorical Computer Science. Pitman, 1986.

[5] D.S. Freidman and D.S. Wise. CONS should not evaluate its arguments. In *Proc. of Intl. Conference on Automata, Languages and Programming*, 1976.

[6] T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of ACM Conference on Compiler Construction*, pages 58–69, 1984.

[7] T. Johnsson. Attribute grammars as a functional programming paradigm. Report 42, University of Göteborg and Chalmers University of Technology, 1987.

[8] A. Laville. *Evaluation Paresseuse des Filtrages avec Priorités.* PhD thesis, Université de Paris VII, 1988.

[9] M. Mauny. *Compilation des Langages Fonctionnels dans les Combinateurs Catégoriques — Application au langage ML.* PhD thesis, Université de Paris VII, 1985.

[10] M. Mauny and A. Suárez. Implementing functional languages in the categorical abstract machine. In *Proceedings of the ACM International Conference on Lisp and Functional Programming*, pages 266–278, 1986.

[11] S.L. Peyton-Jones. *The Implementation of Functional Programming Languages.* Series in Computer Science. Prentice-Hall International, 1987.

[12] L. Puel and A. Suárez. Compiling pattern matching by term decomposition. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, 1990.

[13] J. Rees and W. Clinger. The revised^3 report on the algorithmic language Scheme. In *SIGPLAN Notices*, volume 21, 1987.

[14] P. Weis, M.V. Aponte, A. Laville, M. Mauny, and A. Suárez. The CAML reference manual. Technical Report 121, INRIA, 1990.