Implementation of Lazy Pattern Matching Algorithms

Alain Laville
I.N.R.I.A. (Projet FORMEL)
B.P. 105 78150 Le Chesnay CEDEX France
and Université de Reims
B.P. 347 51062 Reims CEDEX France

1 Introduction

Several of the recently developped functional programming languages include a function definition capability that uses a "pattern matching" mechanism. These languages handle structured values which may be given as argument to the functions. The calculation to perform in order to get the result of the function call is choosen according to the structure of the argument. One may find this feature in such languages as HOPE ([3]), MIRANDA ([13]) or ML ([1], [10], [12]). Pattern matching, and algorithms to perform it, has been widely studied in the theory of "Term Rewriting Systems". It is here used in a more or less restricted way (linearity of patterns, patterns without function symbols ... for example in ML).

Although pattern matching comes from "Term Rewriting Systems" theory, languages using it often add a complementary mechanism that does not belong to this theory. One generally asks computations to be deterministic, which is not the case if a value may match several patterns (this is called ambiguity). Such a trouble may be avoided using unambiguous sets of patterns. But, since such a constraint leads to tedious work from the programmer, ambiguity is allowed between the patterns, and a "meta-rule" is added to choose between ambiguous patterns. Various priority rules have been suggested, the two most frequently used are the following :

- when a value matches several patterns, choose the first one in the list (ordered as given by the programmer).
- when a value matches several patterns, choose the most defined one.

We only shall address in this paper the first case of priority rule, which is the one used in ML.

There is also a growing interest in "lazy" evaluation (see [11] for precise definition). This essentially means that a value is effectively computed only when it is needed to produce the result, and, for a structured value, that only the needed parts are evaluated. This ensures that the language is *safe*, i.e. if computation fails there was no way to avoid this failure. This moreover gives to the language the ability of handling infinite data structures as long as only finite parts of them are used in calculations. MIRANDA and at least two implementations of ML include this feature : Lazy CAML at INRIA (see [8] or [9]) and LML the implementation of Gōteborg (see [1]).

However there are problems when using pattern matching in a lazy language. The question is : "How to find which pattern is matched by a given value, without doing useless computations ?". It is connected with other troubles such as expression evaluations which succeed or fail depending on the order of the arguments in the function definition.

Example Define the two functions :

```
let f1 = function (true, false) -> true
                | (false, x)    -> false;;

let f2 = function (false, true) -> true
                | (x, false)    -> false;;
```

and denote by $\perp$ an infinite computation. One would expect, in a lazy system that both `f1 (false, ⊥)` and `f2 (⊥, false)` evaluate to `false`. This is not the case in existing functional languages : since they use explicit top-down and left-to-right scanning of the patterns they succeed evaluating `f1 (false, ⊥)` to `false` and loop evaluating `f2 (⊥, false)` (see for example the pattern matching compilations described in [2] or [14]). The algorithms we give here provide a compilation of pattern matching with which the two function calls return `false`.

This leads to the question of the existence of a lazy pattern matching algorithm (and of its effective building). The answer was known when no ambiguity exists between patterns since the work of G. Huet and J.J. Lévy (see [5]). In [6], we extended this result to the case of ambiguity with priority meta-rule. However, the results established in this work are essentially theoretical ones and do not give a practical implementation tool (for example, compilation of real pattern matching definitions, using algorithms of [6] may need up to several hours of CPU time).

We present here improved algorithms of practical use, together with an efficiency study of a compiler which incorporates them. This study was done with the CAML system (see [12]), a version of ML using the Categorical Abstract Machine (CAM, see [4]) currently implemented at I.N.R.I.A.

A very detailed version of both the theoretical and the implementation aspects may be found in the author's thesis ([7]).

2 Theoretical Results

We first recall those of the theoretical results of [6] which are useful to state our practical algorithms.

2.1 Notations and definitions

Definition 1 A *pattern* is a term built from the pairing operator, some constructors of (already defined) concrete data types and the special symbol Ω. The meaning of Ω is that one doesn't care, during the pattern matching process, about what may appear at the place where it is used. It replaces both the variables and the "don't care" symbol.

A value of CAML is said to be an *instance* of a pattern if it can be obtained from the pattern by replacing all the Ω's by any values.

With a list of patterns [$p_1, \ldots, p_n$], we shall say that a value v of CAML *matches* the pattern p_i if p_i is the *first* pattern in the list, of which v is an instance.

We shall say that a function is *defined by pattern* if

1. Its definition consists of an ordered list of pairs (pattern, expression)
2. Its value when applied to an argument v is obtained in the following way : first find the first pattern, say p, in the list such that v is an instance of p and then evaluate the result of the corresponding expression.

Assume now that we want to compile the definition by pattern of a function :

$$\Pi = \left\{ \begin{array}{cccc} \mathit{function} & p_1 & \rightarrow & exp_1 \\ | & p_2 & \rightarrow & exp_2 \\ \cdots & \cdots & \cdots & \cdots \\ | & p_n & \rightarrow & exp_n \end{array} \right.$$

Here the p_i's are the patterns to be matched against the argument of the function call (if the function has arity greater than 1, the patterns p_i's are t-uples).

We define a signature Σ containing all the constructors of the types used in the patterns $p_1, \ldots, p_n$ and two other symbols : Ω, which will denote the "unknown" or "undefined", and `otherwise` which will be used to group many cases into a single one (see section 4).

Definition 2 Since such terms will often denote partially known, or partially evaluated values, we shall call *partial term* every term built over Σ. We shall only use *term* to denote a partial term in which there is no symbol Ω (but we do not forbid to use "partial term" even in this case).

Partial terms provide us with a formalism suited for patterns (which are partially undefined terms) as well as for lazy values (which may be thought of as partially unknown since they are not completely evaluated).

Definition 3 We define a partial ordering (denoted by $\leq$) over the set of all partial terms as follows :

- For each partial term M : $\Omega \leq M$
- $F(M_1, \ldots, M_n) \leq F(N_1, \ldots, N_n)$ if and only if $M_i \leq N_i$ $(1 \leq i \leq n)$

The ordering $\leq$ is a kind of prefix ordering with the meaning that a partial term is less than another if it is less defined (or less known).

We shall use the following notations :

- $M \uparrow N$ means that M and N have a common upper bound (and we shall say that M and N are compatible)
- $M \sharp N$ means that they don't have one (and we shall say that M and N are incompatible)
- $\vee$ and $\wedge$ will respectively denote the l.u.b. (when it exists) and the g.l.b. of two partial terms
- $[M, N[$ will denote the set of partiel terms t verifying $M \leq t$ and $t < N$.

Definition 4 When seeing partial terms as trees, we shall say that M_i is the i^{th} *son* of the partial term $F(M_1, \ldots, M_n)$. We call *occurrence* an integer list which designates a subterm of a given partial term. For example, the occurrence [2; 3] points to the third son of the second son of the full partial term. The prefix ordering of occurrences will be denoted by $\leq$. For a given partial term M, we shall denote $\mathcal{O}(M)$ the set of all occurrences in M, $\overline{\mathcal{O}}(M)$ the set of occurrences in M where the symbol is not Ω and $\mathcal{O}_\Omega(M)$ the set of occurrences in M where the symbol is Ω. The symbol in M at occurrence u will be denoted by $M(u)$.

We shall now define some predicates over the set of partial terms (i.e. functions with values in the set {tt, ff} of the truth values).

Definition 5 For each $i \in \{1, \ldots, n\}$, the predicate $match_i$ is defined by $match_i(M) = \text{tt}$ if and only if the following two conditions hold :

1. $p_i \leq M$
2. $\forall j < i \quad p_j \sharp M$

We then define the predicate $match_\Pi$:
$match_\Pi(M) = \text{tt}$ if and only if $match_i(M) = \text{tt}$ for some $i \in \{1, \ldots, n\}$.

The meaning of these predicates is the following :
$match_i(M) = \text{tt}$ iff M is sufficiently defined to know that every value better defined than M is an instance of p_i and not one of a pattern with higher priority than p_i (i.e. a pattern p_j with $j < i$).
$match_\Pi(M) = \text{tt}$ iff M is sufficiently defined to decide which pattern will be matched by any value better defined than M.

Lemma 1 *If we order the set of truth values by defining ff < tt, all these predicates are monotonically increasing functions from the partial terms into the truth values.*

2.2 Lazyness Results

Definition 6 We shall call *minimally extended pattern* (associated with Π) any partial term t verifying the following two properties :

1. $match_\Pi(t) = \text{tt}$
2. $\forall t' < t,\ match_\Pi(t') = \text{ff}$

We shall denote the set of all minimally extended patterns by MEP_Π.

Definition 7 Let *match* be one of the predicates $match_i$ or $match_\Pi$ and M be a given partial term. Let $u \in \mathcal{O}_\Omega(M)$ be such that $\forall N \geq M\ match(N) = \text{tt}$ implies $N(u) \neq \Omega$. We shall say that such an occurrence is an *index of match in M.*

We shall say that *match* is *sequential at M* if and only if the two conditions $match(M) = \text{ff}$ and there exists $N \geq M$ such that $match(N) = \text{tt}$ imply together that there exists an index of *match* in M.

Definition 8 We call *pattern matching algorithm* any deterministic algorithm which will match any partial term against Π (i.e. which finds the first $p_i \in \Pi$ of which the partial term is an instance).

We say that a pattern matching algorithm is *lazy* if it never does useless work (as partial terms are trees and a pattern is a prefix of any partial term that matches it, this process has to work in a top-down way). We may express this constraint in the following way : Assume we want to match a value v against Π and let U be the set of all occurrences in v where the symbol was evaluated during the pattern matching process. Denote v_Ω the partial term which coincides with v along U and is completed with Ω's according to the arities of the symbols used. Then we ask v_Ω to be less than or equal to (for the ordering of partial terms) every prefix of the full value v which is sufficient to choose the right hand side.

Example Consider the two classical function definitions :

```
let AND = function (true, true)  -> true
                 | (x, y)        -> false;;

let XOR = function (true, false) -> true
                 | (false, true) -> true
                 | (x, y)        -> false;;
```

it is easy to see that any pattern matching algorithm is lazy in the case of XOR (both parts of the pair are always needed), and none is lazy in the case of AND (each part of the pair is useless if the other is `false`).

We may now present the main results that were established in [6].

Theorem 1 *1. If the signature Σ is finite the set* MEP_Π *is finite and computable from the list of patterns* Π.

2. Given a function defined by pattern, there exists an associated lazy pattern matching algorithm if and only if the predicate $match_\Pi$ is sequential at any partial term.

3. If two members of MEP_Π *are compatible then there exists no lazy pattern matching algorithm.*

4. If all the elements of MEP_Π *are pairwise incompatible, the existence of a lazy pattern matching algorithm is decidable. Moreover, if such an algorithm exists, one may mechanically be built from the initial list of patterns.*

We only look at these parts of the proof that are useful to present the algorithms of the next sections, and particularly to the effective building of the lazy pattern matching algorithm.

When all the minimally extended patterns are pairwise incompatible, the predicate $match_\Pi$ is the ordinary matching predicate against the set of patterns MEP_Π. Hence its sequentiality is easily decided using already known methods. One only has to check if this predicate is sequential at every partial term which is a prefix of an element of MEP_Π. Moreover, one can exhibit a lazy pattern matching algorithm when the checking succeeds. These two goals are achieved by trying to build a "matching tree" (see for details Huet and Lévy [5] where this method is introduced). A matching tree is a tree of which each node contains a partial term M with an index u of $match_\Pi$ in M, and the branches issued from a node $< M, u >$ are labelled with the symbols that may be placed at u in M in order to get a match. Leaves contain elements of MEP_Π which mean a success in the matching process. The root of the tree contains the partial term Ω with the trivial index of $match_\Pi$ at this partial term.

Given the matching tree and a value v to be matched, the lazy pattern matching algorithm is as follows : start at the root of the tree and when reaching a node take the symbol in v at the occurrence contained in the node ; if there is a branch labelled with this symbol starting from the node then follow it, else the matching process fails ; when reaching a leave one ensures that v is greater than the element, say p, of MEP_Π contained in the leave. Hence the value matches the unique initial pattern p_i such that $match_i(p) = $ tt.

The proof of computability of MEP_Π only consists of the following remark : The set of occurrences in any member of MEP_Π may be bounded by $\bigcup_{i=1}^{n} \overline{\mathcal{O}}(p_i)$ and the signature Σ is finite. Such a characterisation leads to practically useless algorithms because of the exponential growth of the number of partial terms to compute, when the number of patterns or the number of constructors grow. We shall now give pratical algorithms.

3 Building of the Matching Tree

We shall address now the question of efficiently building the matching tree defined in the preceding section[1]. This process asks for finding (at least) an index in each partial term one has to place in the tree. If one has previously generated the set of all the minimally extended patterns, it is easy to find an index in a partial term M less defined than some of the extended patterns. One only has to check if there exists an occurrence $u \in \mathcal{O}_\Omega(M)$ such that u belongs to $\overline{\mathcal{O}}(p)$ for all minimally extended pattern p greater than M.

However, generating all the minimally extended patterns is a very costly process. Hence we shall try to build the tree without using the minimally extended patterns. We shall show that in many cases one can find some index in a partial term without knowing the minimally extended patterns that are greater (see section 3.1). Unfortunately this method may fail to give any index in partial terms where such indexes exist. In such cases we shall have to compute the minimally extended patterns, but only from a restricted set of initial patterns. Once we have generated these patterns we may use them to find indexes in subsequent partial terms.

In the following we only shall deal with partial terms that appear in the matching tree. Hence, we state here two properties of such partial terms which will be used in some of the proofs of the following sections.

Notations : We shall denote $M[u \leftarrow N]$ the partial term obtained by replacing in M the subterm at occurrence u by the partial term N. $F(\vec{\Omega})$ will denote the partial term whose top symbol is F and all the sons (according with arity of F) are Ω.

Proposition 1 *Let M be a partial term appearing in some non terminal node of the matching tree. One has the two following properties :*

1. *There exists a minimally extended pattern p such that $M < p$.*
2. *M has been built by filling indexes, i.e. there exists a sequence $M_0, \ldots, M_q$ of partial terms and a sequence $u_0, \ldots, u_{q-1}$ of occurrences such that :*
   - *(a)* $M_0 = \Omega$
   - *(b)* $M_q = M$
   - *(c) for all $i \in \{0, \ldots, q-1\}$, u_i is an index of $match_\Pi$ in M_i*
   - *(d)* $\forall i \in \{0, \ldots, q-1\}\ \exists F_i \in \Sigma \quad (M_{i+1} = M_i[u_i \leftarrow F_i(\vec{\Omega})])$

Proof : Both properties are obvious consequences of the definition of a matching tree. ∎

Definition 9 We shall call *accessible from* a partial term M, each initial pattern p_i such that $match_i(N) = tt$ for some partial term $N \geq M$.

[1]I would like to thanks one of the referees of this paper for his careful reading and very useful comments. They led to great improvement in the presentation of this section.

3.1 Easy indexes

We shall give here a way of finding an index in a partial term M that is already present in some non terminal node of the matching tree without using the set of all the minimally extended patterns. Looking for efficiency in the compilation process, we do not try to generate *all* the indexes, only to give one as soon as possible.

We first remark that the set of indexes of $match_\Pi$ in M is the intersection of the sets of indexes in M of all the predicates $match_i$ for those i such that p_i is accessible from M. This fact follows obviously from the definition of $match_\Pi$.

Let p_i be accessible from M. What are the indexes of $match_i$ in M ?

First, it is clear that all the occurrences in $\overline{\mathcal{O}}(p_i) \cap \mathcal{O}_\Omega(M)$ are such indexes. Moreover, if we assume that p_i is incompatible with any pattern p_j with $j < i$, then we got here all the indexes of $match_i$ in M.

Assume now that p_i is ambiguous with some patterns of higher priority, denoted here $q_1, \dots, q_n$, and that $M \uparrow q_j$ for all $j \in \{1, \dots, n\}$. Since we assumed that there exists a partial term on which $match_i$ returns tt, the set of occurrences $\overline{\mathcal{O}}(q_j) \setminus \overline{\mathcal{O}}(p_i)$ is nonempty for all $j \in \{1, \dots, n\}$ (it contains the occurrences where one may exclude q_j in order to recognize p_i). We shall denote by U_j the set of occurrences in $\mathcal{O}_\Omega(M)$ which are a prefix of at least one element of $\overline{\mathcal{O}}(q_j) \setminus \overline{\mathcal{O}}(p_i)$.

Consider a partial term N such that $N \geq M$ and $match_i(N) = $ tt. There must exist an occurrence $u_j \in U_j$ such that $N(u_j) \neq \Omega$. If the set U_j only has one element, we may thus ensure that this unique element is an index of $match_i$ in M. Repeating this for all j, we show that the union of those U_j which are singletons is contained in the set of the indexes of $match_i$ in M.

Remark 1 The condition on U_j to be a singleton, although sufficient, is not a necessary one. Suppose, for example, that we deal with this definition of an "Exclusive Or" function :

```
let XOR = function (true, false) -> true
                 | (false, true) -> true
                 | (x, y)        -> false;;
```

It is associated with the list of initial patterns $\Pi = [(\text{true,false}) \; ; \; (\text{false,true}) \; ; \; (\Omega,\Omega)]$. Here pattern p_3 is ambiguous with both p_1 and p_2. If we deal with the partial term $M = (\Omega,\Omega)$ and search for indexes of $match_3$ in M, we get $U_1 = U_2 = \{[1],[2]\}$. Hence the preceding method does not give any index for $match_3$ in M.

On the other hand, the set of all the minimally extended patterns is here :

$$\{(true, false), (false, true), (true, true), (false, false)\}$$

so that, in M, both occurrences [1] and [2] are indexes of $match_\Pi$ (and of course of $match_3$ too).

Remark 2 It is not easy to check if a given pattern p_i is accessible from a partial term M. However one may ensure that if p_i is accessible from M then one has $M \uparrow p_i$. Hence we shall use the preceding result with *all* the initial patterns compatible with M rather than with the accessible ones. It is obvious that this does not produce wrong indexes.

3.2 Restricting the Set of Initial Patterns

We are now faced with the problem of finding an index (if it is possible) in a partial term M where the method of the preceding section has failed. From the properties of the partial terms in the matching tree, we know that it suffices to compute the minimally extended patterns greater than M. Moreover these patterns will give us an easy way to find indexes in all the partial terms appearing in nodes of the matching tree below the node of M.

Using the fact that M was built by filling indexes (see proposition 1), we may give the following characterisation of those minimally extended patterns which are greater than M. It essentially means that one may built them starting from M and discarding all the initial patterns not accessible from M. Denoting Π' the list of patterns accessible from M, the sets $\{t \geq M \ ; \ match_{\Pi}(t) = \mathrm{tt}\ \}$ and $\{t \geq M \ ; \ match_{\Pi'}(t) = \mathrm{tt}\ \}$ have the same minimal elements.

Proposition 2 *Let $\{p_{i_1}, \ldots, p_{i_n}\}$ be a subset of the initial patterns list, containing all the initial patterns accessible from M (we assume that $i_j < i_k$ whenever $j < k$). Let t be a partial term greater than M. Then t is a minimally extended pattern if and only if it satisfies the two following conditions :*

1. $\exists k \ (t \geq p_{i_k}$ *and* $\forall j < k \ (t \sharp p_{i_j}))$
2. $\forall t' \in [M, t[\ \forall k \ (t' \wedge p_{i_k} < p_{i_k}$ *or* $\exists j < k \ (t' \uparrow p_{i_j}))$

<u>Proof</u> : Assume that t is a minimally extended pattern. Since $t \geq M$ and $match_{\Pi}(t) = \mathrm{tt}$, using definition of the subscripts i_k, we can find some k such that $match_{i_k}(t) = \mathrm{tt}$. This means that $t \geq p_{i_k}$ and for all $j < i_k$, we get $t \sharp p_j$. This implies that if $j < k$ (and hence $i_j < i_k$) we have $t \sharp p_{i_j}$: t satisfies the first condition.

In order to get the second condition, let t' be a partial term such that $t' < t$. Since t is a minimally extended pattern, one has $match_{\Pi}(t') = \mathrm{ff}$, and then for all k, either $t' \wedge p_{i_k} < p_{i_k}$, or there exists $j < i_k$ such that $t' \uparrow p_j$. If for all k, $t' \wedge p_{i_k} < p_{i_k}$ holds, we get the desired result. Assume now that for some k, this condition does not hold, then we have to show that there exists $l < k$ such that $t' \uparrow p_{i_l}$. Due to assumption above we know that there exists $j < i_k$ such that $t' \uparrow p_j$. Thus it suffices to show that this subscript j is one of the subscripts i_m. Since moreover t' is greater than or equal to M we have $t' \vee p_j \geq M$ and $match_j(t' \vee p_j) = \mathrm{tt}$. Hence p_j is one of the patterns p_{i_m}, we proved that t satisfies the second condition.

Conversely, let t be a partial term greater than M and satisfying the two conditions given above. Since $t \geq M$, if $j \notin \{i_1, \ldots, i_n\}$ we have $match_j(t) = \mathrm{ff}$. Combining this result with the first condition we get :

$$\exists k \ (t \geq p_{i_k} \text{ and } \forall j < i_k \ (match_j(t) = \mathrm{ff}))$$

which implies that $match_{i_k}(t) = \text{tt}$, hence $match_\Pi(t) = \text{tt}$.

It remains to show that such a t is minimal. To get this result, assume that there exists a partial term $t' < t$ such that $match_\Pi(t') = \text{tt}$. We shall prove that this leads to a contradiction. From the monotonicity of the predicates $match_i$ (see lemma 1), one has $match_{i_k}(t') = \text{tt}$ for the same k as t. If we assumed that t' is greater than or equal to M (i.e. $t' \in [M, t[$) then the second condition would imply that $match_{i_k}(t') = \text{ff}$. This contradiction ensures that t' is not greater than or equal to M. Hence we can find an occurrence v such that $t'(v) = \Omega$ and $M(v) \neq \Omega$. There exists a subscript j_0 such that $M_{j_0+1} = M_{j_0}[v \leftarrow F_{j_0}(\vec{\Omega})]$ and v is an index of $match_\Pi$ in M_{j_0}. This implies $M_{j_0}(v) = \Omega$. Now look at the partial term $t' \vee M_{j_0}$: it is greater than or equal to M_{j_0}, by monotonicity it satisfies $match_\Pi(t' \vee M_{j_0}) = \text{tt}$. Since it has a symbol Ω at occurrence v this contradicts the assumption that v is an index of $match_\Pi$ in M_{j_0}.

In all cases we get a contradiction when we assume that $match_\Pi(t') = \text{tt}$. Hence, t is a minimally extended pattern associated with the initial list of patterns. ∎

Remark 3 We shall use this proposition with the list of patterns *compatible* with M since we do not have an easy way to compute the list of patterns *accessible* from M.

3.3 Generating the Useful Extended Patterns

According to the results of section 3.2, we shall give an algorithm generating all the minimally extended patterns, greater than a given partial term. This set will be built incrementally using two different steps.

In the following we assume given a partial term M and we shall denote by $\{p_1, \ldots, p_n\}$ a subset of the initial patterns list containing at least all those initial patterns which are accessible from M. We assume moreover that the ordering on the patterns has not been removed, i.e. if $i < j$ then p_i has higher priority than p_j. We shall give a construct of the set of all the minimally extended patterns that are greater than M. According to proposition 2, this set may be defined as :

$$
\begin{aligned}
EM = \{\, t;\ & 1)\ \ t \geq M \\
& 2)\ \ \exists k\ (t \geq p_k \text{ and } \forall j < k\ (t \nparallel p_j)) \\
& 3)\ \ \forall t' \in [M,\ t[\ \ \forall i\ (t' \wedge p_i < p_i \text{ or } \exists j < i\ (t' \uparrow p_j))\ \}
\end{aligned}
$$

EM is the set of minimal elements of $\{t \geq M\ ;\ match_\Pi(t) = \text{tt}\ \}$ (the third condition meaning $match_i(t) = \text{ff}$).

Proposition 3 *The set EM is the union, for all the subscripts $k \in \{1, \ldots, n\}$, of the disjoint sets, denoted by EM_k, of the minimal partial terms greater than M verifying $match_k$. EM_k may be defined as follows :*

$$\begin{aligned} EM_k = \{ t;\ & 1)\ \ t \geq M \vee p_k \\ & 2)\ \ \forall j < k\ (t \sharp p_j) \\ & 3)\ \ \forall t' \in [M \vee p_k,\ t[\ \forall i \leq k \\ & \quad\ (t' \wedge p_i < p_i \text{ or } \exists j < i\ (t' \uparrow p_j))\ \} \end{aligned}$$

Proof : We first remark that each $t \in EM_k$ satisfies the two conditions : $t \uparrow p_k$ and $\forall j < k\ (t \sharp p_j)$. This ensures that t does not belong to EM_i for $i \neq k$.

Now let $t \in EM$, and let k be the subscript for which t satisfies the second condition in the definition of EM. Then $t \in EM_k$.

Conversely, let $t \in EM_k$ for some k. It is obvious that t satisfies the first two conditions in the definition of EM. To establish the third one, let t' be a partial term in the interval $[M,\ t[$. Since $t' < t$ and $t \uparrow p_k$, we get $t' \uparrow p_k$. Hence t' is compatible with a pattern (p_k) with higher priority than each p_i for $i > k$: for these i the third condition is satisfied.

Assume now that $i < k$ and denote by t'' the partial term $t' \vee p_k$. Since $t \geq p_k$, one has $t'' \in [M \vee p_k,\ t[$. It follows that :

- either $t'' \wedge p_i < p_i$ and hence (since $t' \leq t''$) $t' \wedge p_i < p_i$,
- or one can find $j < i$ such that $t'' \uparrow p_j$ and $t' \uparrow p_j$. ■

We shall give a method to build stepwise the sets EM_k. Each step of the building will let grow the partial term p_k in order to make it incompatible with one of the patterns with higher priority.

Proposition 4 *Let $k \in \{1, \ldots, n\}$. We recursively define the sets $EM_k^1, \ldots, EM_k^k$, in the following way :*

1. *EM_k^1 is the singleton $\{M \vee p_k\}$*
2. *We assume EM_k^i to be built, for some $i \in \{1, \ldots, k-1\}$. We define EM_k^{i+1} as the union, for all $t \in EM_k^i$, of the following sets E_t :*
   - *if $t \sharp p_{k-i}$ then $E_t = \{t\}$*
   - *else $E_t = \{t' \geq t$; there exists a unique occurrence where the symbols in t' and p_{k-i} are distinct (and are not Ω) and $\overline{\mathcal{O}}(t') \subseteq \overline{\mathcal{O}}(t) \cup \overline{\mathcal{O}}(p_{k-i})\}$.*

Then EM_k is the set of all the elements of EM_k^k that are minimal according to the ordering over the partial terms.

Proof :

We shall establish, using induction on i, the following three properties of the sets EM_k^i :

1. $\forall t \in EM_k^i \;\; t \geq M \vee p_k$

2. $\forall t \in EM_k^i \;\; t \sharp p_j \;\; (j = k-i+1, \ldots, k-1)$

3. If a partial term t satisfies

$$t \;\geq\; M \vee p_k \text{ et } t \sharp p_j \;\; (j = k-i+1, \ldots, k-1)$$

then there exists $t_0 \in EM_k^i$ such that $t \geq t_0$

The three properties hold obviously for $i = 1$. Assume they hold for some i and look at EM_k^{i+1}.

1. Since the elements of EM_k^{i+1} are greater than or equal to those of EM_k^i the first property remains true.

2. Let $t \in EM_k^{i+1}$. There exists $t_0 \in EM_k^i$ such that $t \in E_{t_0}$. Since $t \geq t_0$ and $t_0 \sharp p_j \;\; (j = k-i+1, \ldots, k-1)$ (from the induction hypothesis), we have the following property : $t \sharp p_j \;\; (j = k-i+1, \ldots, k-1)$. From the definition of E_{t_0}, we know that t is incompatible with all the partial terms p_{k-i}. We get so the second property.

3. Let $t \geq M \vee p_k$ satisfying $t \sharp p_j \;\; (j = k-i, \ldots, k-1)$, we shall exhibit an element t_0 of EM_k^{i+1} less than or equal to t. From the induction hypothesis, we know that there exists an element t_1 of EM_k^i less than or equal to t. If this element is not compatible with p_{k-i}, it is a member of EM_k^{i+1} and we may take $t_0 = t_1$. If it is compatible, let u_0 be an occurrence, minimal for the prefix ordering on occurrences, where the symbols in t and p_{k-i} are different and not equal to Ω. Such an occurrence exists from the hypothesis on t. We may then define t_0 by the conditions $\overline{\mathcal{O}}(t_0) = \overline{\mathcal{O}}(t_1) \cup \{u; u \leq u_0\}$ and $t_0 \leq t$. These two conditions may hold together since $\overline{\mathcal{O}}(t_1) \cup \{u; u \leq u_0\} \subseteq \overline{\mathcal{O}}(t)$.

   Moreover, t_0 and p_{k-i} only differ at occurrence u_0 and $t_1 < t_0$. We get $t_0 \in E_{t_1}$, and hence the result. ∎

Remark 4 In the preceding construct, we may replace each set E_t (see proposition 4) by the set of its minimal elements (for the ordering over the partial terms). The first two properties of the sets EM_k remain since we only reduce the size of these sets. The third one remains since we keep all the minimal elements.

We have now to give a method to build the sets which we called E_t in the proposition 4. This building may be done in an obvious way : given the partial terms t and p_j, E_t is the set of all the partial terms we get by replacing in $t \vee p_j$ the subterm at an occurrence u in $\overline{\mathcal{O}}(p_j) \setminus \overline{\mathcal{O}}(t)$ by $F(\vec{\Omega})$, where F is any constructor other than $p_j(u)$.

There are at least two reasons for inefficiency in this method. First, it does not use the fact that one is not allowed to place any constructor at a given occurrence in a partial term : ML's typing constraints only allow a constructor of the same type than $p_j(u)$ to appear at occurrence u in p_j. The second reason is that it does not use remark 4, building many partial terms that are to be discarded. We shall not address here the question of using only useful symbols when generating the set of all the minimally extended patterns. It essentially relies on looking at which symbols appear in still accessible patterns when one tries to extend a partial term toward a minimally extended pattern ; we shall discuss it in a section dealing with properties which depend on ML specificities (see below section 4).

In order to use remark 4 in the construct given in proposition 4, replacing E_t by its minimal elements, we have to characterize these minimal elements of a set E_t. This is done in the following lemma.

Lemma 2 *With the notations of the proposition 4, assume given $t \in EM_k^i$ such that $t \uparrow p_{k-i+1}$. Let $t' \in E_t$ and denote by u_0 the occurrence where t' is incompatible with p_{k-i+1}. Then t' is minimal in E_t if and only if $\overline{\mathcal{O}}(t') = \overline{\mathcal{O}}(t) \cup \{u; u \leq u_0\}$.*

Proof : To get a partial term less than t', we have to replace in t the subterm at an occurrence $u \in \overline{\mathcal{O}}(t')$ by Ω. If we want that the result remains greater than or equal to t, we must choose $u \notin \overline{\mathcal{O}}(t)$. If u is a prefix of u_0, the resulting partial term is compatible with p_{k-i+1}. If u is not a prefix of u_0, the result is still an element of E_t. ■

4 Restricting the Set of Constructors

We present in this section a way of improving the pattern matching compilation by restricting the set of symbols one has to place in an extended pattern at a given occurrence.

As was previously stated, only the symbols that appear in the initial patterns accessible from a partial term M, are useful when building the minimally extended patterns greater than M : if we find, during the pattern matching process, another constructor in the value to be matched, it only excludes from the accessible patterns all those which have not an Ω at the occurrence we just looked at. This does not depend on the symbol we found, only on the fact that it does not appear at this occurrence in the patterns accessible from the prefix we have already scanned.

This point relies on using infinite data types in patterns, such as `Integer` or `String`. In such a case one cannot assume that the signature is finite, an assumption that we used when proving that the set EMP_Π is computable. However, the preceding remark gives the answer : When dealing with infinite data types we collect all the values that appear in the initial patterns and group the other values as a single one (denoted "`otherwise`" in Σ). In such a way we get a finite signature. The same method gives us a restriction on the set of useful constructors : Rather than using all the constructors in the types that appear in the initial patterns, we only deal with the constructors that are present in the patterns which are accessible from the already scanned prefix. Of course we have to add the "`otherwise`" cases each time that not all the constructors of a type appear in the patterns list : the other ones may be present in the values which will be given to the pattern matching algorithm.

The way to restrict the set of constructors we just described has a disadvantage : one has to handle some bookkeeping of the accessible patterns all along the building of the minimally extended patterns. This may be avoided, using weaker restrictions over the sets of constructors to be tested. We give here two restrictions which may be set only looking at the set of initial patterns. They may be combined, but the resulting constraint remains weaker than it could be done.

Using typing information

The type constraints of the ML system require that the patterns of a pattern matching definition have a common most general (polymorphic) type. This implies that every prefix of one of these patterns has a type more general than this one. Moreover, the typechecker of the ML system ensures that when we have looked at a prefix M of some ML value v, we may find the type of each constructor that may appear in v at an occurrence u belonging to $\mathcal{O}_\Omega(M)$. It is the (common) type of all the constructors that appear at occurrence u in the initial patterns accessible from M. Hence, we don't have to worry about constructors belonging to other types.

Using occurrences information

A simple way to restrict the set of constructors to deal with, is building an A-list which associates each occurrence with the set of all the constructors appearing at that occurrence in at least one of the initial patterns. There is of course no reason to consider any other constructor to fill an occurrence when building the extended patterns (assuming the use of an "`otherwise`" case which groups all other constructors).

Remark 5 Even combining the last two restrictions does not eliminate all useless extended pattern as shows the following example. Of course these useless patterns do not affect lazyness ; moreover, cases where there arise, are rather uncommon. One has here to make a choice between compile-time and run-time efficiency.

Example The following pattern matching is part of a function (see page 15) which performs some optimizations over the CAM code generated by the CAML compiler ("::" is the infix notation CAML uses for the CONS function) :

```
function (Push :: Car :: Swap :: _)     -> 1
       | (Push :: Quote _ :: App :: _) -> 2
       | c                             -> 3;;
```

If we try to generate the minimally extended patterns greater than `(Push :: Car :: Ω :: Ω)` we have to replace the first Ω. Typing constraints give here no restriction, since all the patterns are lists of instructions. Using occurrence's information restricts the set of constructors to {Swap, App} (and "otherwise"). It is a real improvement (the type "instruction" has about 15 constructors, 5 of them appearing in the patterns list). However it could be sufficient to consider Swap and "otherwise".

5 Efficiency results

5.1 Compile time

Efficiency's assessment has been done by using various examples of pattern matching definition. These examples have been taken from the CAML system itself (this system is written in CAML and bootstrapped, see [12]). Using definitions that were written before our implementation work, we ensure that these examples are not biased toward or against our algorithms[2].

The bigger example is the following one (there are 74 cases but we do not explicitly state all of them) :

```
function []                                      ->  1
       | (Push::Quote 0::Swap::CC)               ->  2
       | (Push::Quote <<'()>>::Branch(_,Pop::F)::CC) ->  3
       | (Push::Quote _::Branch(Pop::T,_)::CC)   ->  4
       | (Push::Quote 0::App::CC)                ->  5
       | (Push::Quote 0::Call(2,fn)::CC)         ->  6
       | (Push::Quote <<(quote ^00 . ^Z)>>::CC0) ->  7
       | (Push::Car::Swap::CC)                   ->  8
       | (Push::Cdr::Swap::CC)                   ->  9
       | (Push::Acc 0::Swap::CC)                 -> 10
       | (Push::Rest 1::Swap::CC)                -> 11
       | (Push::Swap::CC)                        -> 12
       | (Push::Cons::CC)                        -> 13
       | (Push::Rplac1 0::Pop::CC)               -> 14
       | (Push::Pop::CC)                         -> 15
       | (Push::CC)                              -> 16
.......................................................
       | (Push_trap(xl,C1)::_ as CC)             -> 74;;
```

It is a function that performs some optimizations over the CAM code generated by the CAML compiler. Hence, the patterns are lists of CAM instructions representing various code structures. In order to only study the pattern-matching process, we replaced the right hand side expressions by integer constants representing the rule number.

[2]at least not intentionnally

We did not try to evaluate the improvement due to the restrictions over the set of constructors to deal with : it may be very important, but is easy to implement and is obviously an improvement.

However this improvement is not sufficient to get an effective implementation. For example, we generated the set of minimally extended patterns, using only the theoretical existence algorithm and the restrictions on the constructors, for the above function definition. The generation used more than two hours of C.P.U. time on a Vax 11-780 !!

In order to study the effect of the other improvements given in this paper, we compare compilation time of some parts of this same function.

First, we give results of compiling the complete function. If we generate all the minimally extended patterns, using the incremental building method given in section 3.3 but without the restriction of the set of initial patterns given in section 3.2, the compilation took 373 seconds on a Vax 11-780 (of which about 230 seconds are G.C. time). Including the restriction, the compilation took only 160 seconds (with about 90 seconds of G.C. time).

In order to see the effect of the restriction method, we compiled with and without it, some parts of the function above. Keeping only rules 3, 4 and 7 we get an highly ambiguous pattern matching in which the method of section 3.2 does not reject any rule before looking at a set of extended patterns. In fact the two compilations need comparable time (40 seconds vs. 37 seconds), with a slight gain since in the second case the generation process does not start from Ω but from a more precise partial term (corresponding to `(Push :: Quote _ :: _)`).

Keeping now the following pattern matching :

```
function []                                          -> 1
       | (Push::Quote 0::Swap::CC)                   -> 2
       | (Push::Quote <<'()>>::Branch(_,Pop::F)::CC) -> 3
       | (Push::Quote _::Branch(Pop::T,_)::CC)       -> 4
       | (Push::Quote 0::Call(2,fn)::CC)             -> 6
       | (Push::Quote <<(quote ^00 . ^Z)>>::CC0)     -> 7
       | (Push::Car::Swap::CC)                       -> 8
       | (Push::Acc 0::Swap::CC)                     -> 10
       | (Push::Swap::CC)                            -> 12
       | (Push::Rplac1 0::Pop::CC)                   -> 14
       | (Push::CC)                                  -> 16;;
```

the difference between the two compilation times is 101 seconds vs. 49 seconds. The second method needs generating six sets of minimally extended patterns, one from four of the initial patterns, the other five from only two of the original patterns.

The experiments we made with other pattern matching definitions lead to results of the same order of magnitude.

We tried to compare the compilation times needed by our algorithms with those needed by the compiler of the CAML system. These comparisons are made using various pattern matching definitions from the CAML system itself. They are not perfectly precise since the work of the CAML compiler has to be distinguished from the typechecking, and these two processes are somewhat interleaved in the present system.

Depending on the kind of pattern matching (numerous ambiguity cases or not, number of constructors, size of the patterns ...), our compilation needs from 2/3 of the time used by the present compiler up to three times this time. It is not clear which are the most important features in these differences between the various cases. However, the comparisons with the present CAML compiler are essentially intended to ensure that our algorithms run fast enough to be incorporated in a future version of the system, which is now the case.

5.2 Run time

The same comparisons as in the testing of compile time efficiency, have been run. We looked at two kinds of measurement : size of the generated code, and the time needed to recognize which rule has to be applied.

Concerning the second point, precise comparison would need to define some kind of *average* value to be matched. We did not try to do such a work. However, with various attempts, the two codes do the pattern matching using about the same amount of time.

Concerning the size of the generated code, we estimate it by the number of instructions in the generated CAM code. This is a machine independent value, and it does not depend on other parts of the CAML system. The number of CAM instructions, in the code given by our algorithms, is about two third of the number of CAM instructions in the code generated by the present compiler. The gain comes from more efficient access to parts of the value, and from a less number of testing instructions to provide. The gain on the number of testing instructions is about 20 % in all the cases we tried. For example, in the case of the function given page 15 our compiler gives a code with 1011 CAM instructions and 203 test versus 1490 instructions and 242 tests with the present system.

However, the values we give here are only approximate ones. One may build specific ML pattern matching and values which do not fit with these estimates. This is the main reason why we used pattern matching of the CAML system in order to make our comparisons. Thus, since we did not use ad hoc examples, we mean that the values we give here are a rather correct estimate of the efficiency one may expect from our algorithms.

6 Conclusion

The compiler we used all along this paper is not actually implemented in the CAML system. There are two main reasons for this fact. The first one is that we want to do more experiments on some features of this compiler (it may be modified to use its data structures in the binding of the patterns' variables, it could be useful to implement some heuristics to deal with cases where no index exists in a partial term ...). The second one is that we have to provide an interface with the whole compiler and the typechecker and it is likely that these processes will have to be somewhat adapted according to which features we eventually retain in our pattern matching compiler.

However, the work we present in this paper shows that it is effectively possible to use lazy pattern matching in a functional language. This is a needed feature in a lazy system. We show that this may be a useful one even in a strict version : it gives a more compact object code without loss of run-time performance and with a quite acceptable compile-time performance. This is an incitement to further study the possible uses of this kind of mechanism.

References

[1] L. Augustsson "A Compiler for Lazy ML", A.C.M. Conference on Lisp and Functional Programming, Austin 1984, pp 218-225

[2] L. Augustsson "A Pattern Matching Compiler", Conf. on Functional Programming Languages and Computer Architecture, Nancy, 1985 (LNCS 217)

[3] R. Burstall D. MacQueen D. Sannella "HOPE : An Experimental Applicative Language", A.C.M. Conference on Lisp and Functional Programming, Stanford 1980, pp 136-143

[4] G. Cousineau P.L. Curien M. Mauny "The Categorical Abstract Machine", in J.P. Jouannaud ed. Functional Programming Languages and Computer Architecture, L.N.C.S. 201, Springer Verlag 1985

[5] G. Huet J.J. Lévy "Call by Need Computations in Non Ambiguous Linear Term Rewriting Systems", Rapport IRIA Laboria 359, August 1979

[6] A. Laville "Lazy Pattern Matching in the ML Language", Proceedings of the 7th Conference on Foundations of Software Technology and Theoretical Computer Science, Pune (India), December 1987, L.N.C.S.

[7] A. Laville "Filtrage et Evaluation paresseuse", Thèse de Doctorat, Université Paris 7, to appear

[8] M. Mauny "Compilation des Langages Fonctionnels dans les Combinateurs Catégoriques, Application au langage ML", Thèse de 3ème cycle, Université Paris 7, 1985

[9] M. Mauny A. Suarez "Implementing Functional Languages in the Categorical Abstract Machine", A.C.M. Conference on Lisp and Functional Programming, Cambridge 1986, pp 266-278

[10] R. Milner "A Proposal for Standard ML", A.C.M. Conference on Lisp and Functional Programming, Austin 1984, pp 184-197

[11] G. Plotkin "Call-by-need, Call-by-value and the Lambda Calculus", T.C.S. Vol 1, pp 125-159, 1975

[12] A. Suarez "Une Implémentation de ML en ML", Thèse, Université Paris 7, to appear

[13] D. Turner "Miranda a Non Strict Functional Language with Polymorphic Types", in J.P. Jouannaud ed. Functional Programming Languages and Computer Architecture, L.N.C.S. 201, Springer Verlag 1985

[14] P. Wadler, "Efficient Compilation of Pattern Matching", in S. Peyton Jones *The Implementation of Functional Programming Languages*, Prentice-Hall Series in Computer Science, 1987