Compiling Lazy Pattern Matching

Luc Maranget *

1 Introduction

Pattern matching is a key feature of the ML language. Pattern matching is a way to discriminate between values of structured types and to access their subparts. Pattern matching enhances the clarity and readability of programs. Compare, for instance, the ML function computing the sum of an integers list with its Lisp counterpart (all examples are in CAML [11] syntax).

```
let rec sum xs = match xs with
  []    → 0
| y::ys → y+sum ys
```

```
  (defun sum (l)
    (if (consp l)
      (+ (car l) (sum (cdr l)))
      0))
```

In ML, patterns can be nested arbitrarily. This means that pattern matching has to be compiled into sequences of simple tests: a complicated pattern such as ((1, x), y::[]) cannot be recognized by a single test. Usually, pattern matching compilers attempt to "factorize" tests as much as possible, to avoid testing several times same position in a term.

A pattern matching expression does not specify the order in which tests are performed. When ML is given strict semantics, as in SML [7], all orders are correct and choosing a particular order is only a matter of code size and run-time efficiency. When ML is given lazy semantics, as inz LML [1], all testing orders are not semantically equivalent. Consider for instance the ML definition:

```
let F x y  = match (x,y) with
  (true,true)   → 1
| (_,false)     → 2
| (false,true)  → 3
```

Patterns can be checked from left to right, as it is usually the case (function F_1, below), or from right to left (function F_2)

```
let F1 x y =
  if x then
    if y then 1
    else 2
  else
    if y then 3
    else 2
```

```
let  F2 x y =
  if y then
    if x then 1
    else 3
  else
    2
```

When variable y is bound to false, the test on variable x is useless. This can be avoided by testing y before x, as in F_2.

Worse, consider function application F ⊥ false, where ⊥ is a non-terminating computation. In strict ML, function arguments are reduced before calling the functions, so that both compilations F_1 ⊥ false and F_2 ⊥ false do not terminate. In lazy ML, function arguments are not be evaluated until their values are actually needed. Therefore, function F_1 will loop by trying to evaluate x = ⊥, whereas F_2 will give the answer 2. In the spirit of lazy evaluation, a result should be given whenever possible. Thus, a lazy compiler should compile function F as F_2, not as F_1.

It is essential for a lazy ML compiler to produce a correct compilation of pattern matching whenever there exists one. This problem has first been solved in the case of non-overlapping patterns by Huet and

*INRIA Rocquencourt. This work was partially funded by DRET under grant N°8780814.

Lévy [2]. Given a set of (possibly) overlapping patterns, A. Laville [5] shows how to replace them, when possible, by an equivalent set of non-overlapping patterns, compiled using Huet and Lévy's technique. A. Suárez and L. Puel [8] translate the initial set of overlapping patterns into an equivalent set of "constrained" patterns, which are special patterns encoding the disambiguating rule of pattern matching on overlapping patterns. Then, they compile pattern matching on the constrained patterns with an extension of Huet and Lévy's technique.

In this paper we take a more direct approach: we compile pattern matching on overlapping patterns. We first recall the semantics of lazy pattern matching, as given by A. Laville [5]. Then, we explain our compilation technique as a source to source transformation. Given a set of patterns, several compilations are possible, we prove that they all satisfy a *partial correctness* property. We also give a criterion to characterize *totally correct* compilations and an algorithm to find a totally correct compilation whenever there exists one. We compare our approach to previous ones and we show that effective computation of a correct compilation is more feasible in our framework. Our algorithm may still lead to huge computations, we propose a simple heuristic that solves this problem in practice.

2 Values and patterns

Our intention is to model pattern matching as a function on the set of terms representing the results of lazy ML programs.

2.1 Partial values

A constructor is a functional symbol with an arity. A constructor will often be represented by c and its arity by a. Constructors are defined by data type declarations. Consider for instance the type declaration:

```
type tree α =
  Leaf α | Node (tree α) α (tree α)
```

This declaration defines the type `tree` α of the binary trees of objects of type α. It introduces the two constructors `Leaf` and `Node`, of arities 1 and 3. The set of all constructors in a type is the *signature* of this type. Some types, as pairs, lists, booleans and integers are pre-defined. There is one single binary constructor, written as an infix ",", in the signature of the type of pairs. The type of lists has two constructors, the binary *cons*, written with an infix "::" and the nullary *nil*, written as []. The booleans are the two nullary constructors `true` and `false`. Finally, the signature of the type of integers is infinite; it consists of all the signed integers, viewed as nullary constructors.

The distinguished nullary symbol Ω stands for the unknown parts of a value. The set $\mathcal{V}_\Omega$ of partial values is the set of terms built with constructors and symbol Ω:

$$\textit{Partial values}\quad \mathcal{V}_\Omega:\quad V ::= \Omega \mid c\ V_1\ V_2 \ldots V_a$$

In a partial value $V = c\ V_1\ V_2 \ldots V_a$, constructor c is the *root constructor* of V. We only consider values that are well typed in the standard sense, partial value Ω belonging to all types. For instance, `Node` Ω `1 (Leaf 2)` has type `tree int`.

A lazy language distinguishes between totally unknown values and partially unknown values. Consider, for instance, a list of two unknown values, represented as Ω::Ω::[]. We can refer to the length of such a partial list. In a lazy language, we should even be able to compute it. As to the totally unknow value Ω, it does not carry any information at all. This suggests that partial values may be considered as more or less precise approximations of the results of computations. The definition ordering captures this intuition.

Definition 2.1 (Definition Ordering) *Let U and V be two partial values of the same type. The partial value U is said to be* less defined *than V, written $U \preceq V$, if and only if :*

$$\left\{\begin{array}{l} U = \Omega \\ \quad \textit{or} \\ \left\{\begin{array}{l} U = c\ U_1\ \ldots\ U_a,\ V = c\ V_1\ \ldots\ V_a \\ \quad \textit{and} \\ \textit{for all } i \textit{ in the interval } 1 \ldots a,\ U_i \preceq V_i \end{array}\right. \end{array}\right.$$

Two partial values U and V are said to be *compatible*, written $U \uparrow V$, when they can be refined toward the same partial value, i.e., when there exists a common upper bound of U and V. When this is not the case, values U and V are *incompatible*, written $U \# V$.

We also consider the set $\mathcal{T}_\Omega$ of well typed partial terms built from constructors, symbol Ω and a set of variables.

$$\textit{Partial terms}\quad \mathcal{T}_\Omega:\quad M ::= \Omega \mid v \mid c\ V_1\ V_2 \ldots V_a$$

A *substitution*, written as σ, is a morphism on partial terms, i.e., a function on partial terms such that: $\sigma(c\ V_1\ V_2 \ldots V_a) = c\ \sigma(V_1)\ \sigma(V_2) \ldots \sigma(V_a)$. Sometimes, a substitution will be written as the environment $[v_1 \backslash M_1,\ v_2 \backslash M_2,\ \ldots v_n \backslash M_n]$ binding, for any integer i in the interval $[1 \ldots n]$, the variable v_i to the partial term M_i. Application of such a substitution to a partial term is written as $N[v_1\backslash M_1,\ v_2\backslash M_2,\ \ldots v_n\backslash M_n]$. For any partial term M, the partial value M_Ω is obtained by substituting Ω for all variables in M (i.e., $M_\Omega = M[v_1\backslash\Omega,\ v_2\backslash\Omega,\ \ldots v_n\backslash\Omega]$, where $v_1, v_2, \ldots, v_n$ are the variables of term M).

2.2 Patterns

Patterns are strict linear terms, i.e., partial terms without Ω, such that the same variable does not appear more than once in them. Variables in pattern are written as x.

$$\textit{Patterns}\quad \mathcal{P}:\quad p ::= x \mid c\ p_1\ p_2 \ldots p_a \qquad p \text{ is linear}$$

A pattern can be seen as representing a set of (partial) terms sharing a common "prefix". Additionally, subterms located under this prefix are bound to pattern variables.

Definition 2.2 (Instantiation relation) *Let p be a pattern and M be a partial term belonging to a common type. Term M is an instance of pattern p, written $p \preceq M$, iff there exists a substitution σ such that $\sigma(p) = M$.*

The instantiation relation is closely related to the definition ordering.

Lemma 2.3 *Let p be a pattern and M be a partial term. The following equivalence holds:*

$$p \preceq M \textit{ iff } p_\Omega \preceq M_\Omega$$

Proof: Easy induction on p, it requires the linearity of patterns. □

A pattern p and a partial term M are incompatible, and we write $p \# M$, when M is sufficiently defined to ensure that it is not an instance of p. That is, we state $p \# M$, if and only if

$$\left\{\begin{array}{l} p = c\ p_1 \ldots p_a,\ M = c'\ M_1 \ldots M_{a'} \text{ with } c \neq c' \\ \quad\text{or} \\ \left\{\begin{array}{l} p = c\ p_1 \ldots p_a,\ M = c\ M_1 \ldots M_a \\ \quad\text{and} \\ \text{there exists } i \text{ such that } p_i \# M_i \end{array}\right. \end{array}\right.$$

The compatibility notation $p \uparrow M$ applies when pattern p and partial term M are not incompatible. The following equivalence properties, holding for any pattern p and any partial term M, directly follow from lemma 2.3:

$$p \# M \text{ iff } p_\Omega \# M_\Omega \quad\text{and}\quad p \uparrow M \text{ iff } p_\Omega \uparrow M_\Omega$$

Partial term M may be a pattern q. If patterns p and q are compatible, then they are also said to be *ambiguous* or *overlapping*. As a consequence of lemma 2.3, two patterns are compatible if and only if they admit a common instance.

3 Compilation

Pattern matching is modeled as a function over the set of partial values. This function is compiled into multi-way branches represented by *simple* pattern matching expressions, in the spirit of [1].

3.1 The matching function

Pattern matching is usually formalized as a predicate on partial values [2, 5]. We prefer a representation as a function over partial values, closer to pattern matching in ML.

A *clause* is a three-tuple (i, p, e), where i is an integer, p is a pattern and e is a partial term, such that all variables in term e are variables of pattern p. Integer i is the *number* of the clause, whereas term e is its *result*. To simplify notations, we shall write clauses as $p^i\!: e_i$. We consider *sets of clauses* meeting the following three conditions:

1. All clause numbers are distinct.
2. All patterns belong to a common type.
3. All results belong to a common type.

Sets of clauses are written $E = \{p^i\!: e_i \mid i \in I\}$, where I is a set of numbers. These sets are ordered by the ordering on the clause numbers. The pattern matching function takes this ordering into account to resolve possible ambiguities between patterns. In our view, clause numbers just express the natural textual clause ordering meant by the programmer, when he writes one clause after (under) another.

Definition 3.1 (Matching predicate (Laville)) *Let $E = \{p^1\!: e_1, p^2\!: e_2, \ldots, p^m\!: e_m\}$ be a set of clauses. Let V be a partial value. Value V* matches *clause number i in E and we write $match_i(V, E)$, if and only if the following two conditions are satisfied:*

$$p^i \preceq V \quad \textit{and} \quad \forall j < i,\ p^j \# V$$

Notice that the matching predicates defined by two distinct clause numbers are mutually exclusive, because $p^j \# V$ excludes $p^j \preceq V$.

Definition 3.2 (Pattern matching function) *Let E be a set of clauses. For any partial value V, we define the partial value $match(V, E)$ as follows:*

- *If value V matches clause number i, we take $match(V, E) = \sigma(e_i)$, where σ is the substitution such that $\sigma(p^i) = V$.*

- *Otherwise, V is a* non-matching value *and we take $match(V, E) = \Omega$.*

It is easy to show that, given a set of clauses E, the function $match(V, E)$ is a monotonic function over partial values. Other rules than the textual priority rule (definition 3.1) can be used to resolve ambiguity in patterns: in particular, the specificity rule [4]. We do not consider this alternative, since the textual priority ordering mimics the familiar "*if condition*$_1$ *then result*$_1$ *else if condition*$_2$ *then result*$_2$ ..." construct. Furthermore, both schemes have the same expressive power [5].

Pattern matching expressions can also be written as ML programs. If a pattern variable does not appear in the corresponding result expression, then its name is unimportant and the pattern variable is replaced by the symbol "_". Consider, for instance, the set of clauses $E = \{(x_1, \texttt{true}) : \texttt{true}, (\texttt{true}, x_2) : \texttt{true}, (x_3, x_4) : \texttt{false}\}$ and the function $or(V) = match(V, E)$. In ML syntax we have:

```
or(V) =  match  V with
                 _,true    →  true
               | true,_    →  true
               | _,_       →  false
```

There is a finite number of partial values of type bool × bool. By definition 3.2, we get:

V	$or(V)$
Ω (Ω,Ω) $(\Omega,\texttt{false})$ $(\texttt{true},\Omega)$ $(\texttt{false},\Omega)$	Ω
$(\Omega,\texttt{true})$ $(\texttt{false},\texttt{true})$ $(\texttt{true},\texttt{true})$ $(\texttt{true},\texttt{false})$	true
$(\texttt{false},\texttt{false})$	false

Note that *or* is *not* the "parallel or" function *por*, since $por(\Omega, \texttt{true}) = por(\texttt{true}, \Omega) = \texttt{true}$.

It may seem that the definition of pattern matching might be simplified by replacing condition *2*: $\forall j < i,\ p^j \# V$, by the new and less strict condition: $\forall j < i,\ p^j \not\preceq V$. Such a change is not advisable though, since it would imply loosing monotonicity. Consider, for instance, the pattern matching $match(V, E)$ defined by:

```
match V with 1  →  Ω  |  _  →  2
```

The modified definition of the matching function would give us: $match(\Omega, E) = 2 \not\preceq match(1, E) = \Omega$.

3.2 Pattern matching on vectors

When we examine the compilation of pattern matching in the next section, we shall consider "intermediate" matchings. In these matchings, the value to match and the patterns have a common prefix: the part of the value examined so far. More precisely, let n be a positive integer, let v_1, $v_2 \ldots v_n$ be n variables and let N be a linear partial term whose variables are v_1, $v_2 \ldots v_n$. An intermediate matching is a pattern matching of the format:

$$\begin{array}{l} \texttt{match}\ N[v_1\backslash V_1,\ v_2\backslash V_2, \ldots v_n\backslash V_n]\ \texttt{with} \\ \quad N[v_1\backslash p_1^1,\ v_2\backslash p_2^1, \ldots v_n\backslash p_n^1] \to e_1 \\ \quad \vdots \\ \mid\ N[v_1\backslash p_1^m,\ v_2\backslash p_2^m, \ldots v_n\backslash p_n^m] \to e_m \end{array}$$

Obviously, the result of such a matching does not depend on the prefix N, but only on the partial values V_i and patterns p_i^j that are substituted for the variables v_i.

The n partial values may be seen as a vector $\vec{V} = (V_1\ V_2 \ldots V_n)$, whereas each clause may be seen as a vector clause consisting of a number i, of a vector of n patterns $\vec{p^i}$ and of a result term e_i. The set of clauses is replaced by a *clause matrix* (P) written as:

$$(P) = \begin{pmatrix} p_1^1 & p_2^1 \cdots p_n^1 : e_1 \\ p_1^2 & p_2^2 \cdots p_n^2 : e_2 \\ & \vdots \\ p_1^m & p_2^m \cdots p_n^m : e_m \end{pmatrix}$$

In pattern p_d^i, integer i is the clause or row number, whereas d is the column index.

The instantiation and the incompatibility relation on patterns and values trivially extend to vectors:

$$(p_1\ p_2 \ldots p_n) \preceq (V_1\ V_2 \ldots V_n)$$
if and only if
for all i in $1 \ldots n$, we have $p_i \preceq V_i$

$$(p_1\ p_2 \ldots p_n) \# (V_1\ V_2 \ldots V_n)$$
if and only if
there exists i in $1 \ldots n$ such that $p_i \# V_i$

Substitutions operate on vectors in the natural way: $\sigma(\vec{p}) = (\sigma(p_1)\ \sigma(p_2) \ldots \sigma(p_n))$. It is then straightforward to extend the definition of the matching predicate to vectors of partial values $\vec{V}$ and matrices of clauses (P):

$$match_i(\vec{V}, (P)) \text{ iff } \begin{cases} \vec{p^i} \preceq \vec{V} \\ \quad \text{and} \\ \text{For all } j < i, \text{ we have } \vec{p^j} \# \vec{V} \end{cases}$$

$$\mathcal{C}((v_1\ v_2\ldots v_n), \begin{pmatrix} x_1 & x_2\cdots x_n & : e_1 \\ p_1^2 & p_2^2\cdots p_n^2 & : e_2 \\ & \cdots & \\ p_1^m & p_2^m\cdots p_n^m & : e_m \end{pmatrix}) = e_1[x_1\backslash v_1,\ x_2\backslash v_2,\ \ldots x_n\backslash v_n]$$

Figure 1: Compilation: first row is made of variables

$$\mathcal{C}((v_1\ v_2\ldots v_n), \begin{pmatrix} x^1 & p_2^1\cdots p_n^1 & : e_1 \\ x^2 & p_2^2\cdots p_n^2 & : e_2 \\ & \cdots & \\ x^m & p_2^m\cdots p_n^m & : e_m \end{pmatrix}) = \mathcal{C}((v_2\ldots v_n), \begin{pmatrix} p_2^1\cdots p_n^1 & : e_1[x^1\backslash v_1] \\ p_2^2\cdots p_n^2 & : e_2[x^2\backslash v_1] \\ \cdots & \\ p_2^m\cdots p_n^m & : e_m[x^m\backslash v_1] \end{pmatrix})$$

Figure 2: A column is made of variables

If vector $\vec{V}$ matches clause number i in matrix (P), then there exists a substitution σ such that $\sigma(\vec{p}^i) = \vec{V}$ and take $match(\vec{V},(P)) = \sigma(e_i)$. Otherwise, vector $\vec{V}$ does not match any clause in (P) and take $match(\vec{V},(P)) = \Omega$

3.3 Compilation

Compilation is defined as a function $\mathcal{C}$, from the set of pattern matching expressions to the set of nested simple pattern matching expressions. Simple matchings are a natural presentation of multiway-branches in ML: they are pattern matchings such that the patterns to be matched are non-nested or *simple* patterns.

Simple patterns $p ::= x \mid c\ x_1\ x_2\ldots x_a$ p is linear

Function $\mathcal{C}$ takes two arguments. The first argument is a linear vector of n variables $\vec{v} = (v_1\ v_2\ldots v_n)$ and the second one is a matrix of clauses (P) of width n. A typical call to function $\mathcal{C}$ is thus written as $\mathcal{C}(\vec{v},(P))$. Vector $\vec{v}$ abstracts the input to the pattern matching. To see this, let $\vec{V} = (V_1\ V_2\ldots V_n)$ be any vector of partial values of size n. We define $\mathcal{C}((V_1\ V_2\ldots V_n),(P))$ as $\mathcal{C}((v_1\ v_2\ldots v_n),(P))[v_1\backslash V_1,\ v_2\backslash V_2,\ \ldots v_n\backslash V_n]$. Matrix (P) represents the unchecked subparts of the initial patterns. Given a set of clauses $E = \{p^i : e_i \mid 1 \leq i \leq m\}$, compilation is started by:

$$\mathcal{C}((v), \begin{pmatrix} p^1 & : e_1 \\ & \vdots \\ p^m & : e_m \end{pmatrix})$$

Where v is a fresh variable and function $\mathcal{C}$ is defined as follows:

1. If vector $\vec{v}$ is of length zero, then the matching process is over. Either the clause matrix is empty and matching fails:

$$\mathcal{C}((),()) = \Omega$$

Otherwise, there is one or more clauses in (P) and the result expression of the first clause is the result of the whole matching:

$$\mathcal{C}((), \begin{pmatrix} : e_1 \\ \vdots \\ : e_m \end{pmatrix}) = e_1$$

2. If the first row of patterns only contains variables, then matching successes and returns the first expression as its result (see figure 1).

3. If one column of patterns —the first one, for instance— contains only variables, then the corresponding variable in vector $\vec{v}$ does not need to be examined (see figure 2).

4. If none of the rules above apply, then matching can only progress by examining one of the variables v_i. Until otherwise stated, the choice of this variable is arbitrary and the result of compilation *a priori* depends on this choice. When variable v_i is chosen, we shall say that matching is done by following index i. To be more specific, assume that the first variable, v_1, is chosen. Let $\Sigma = \{c_k \mid 1 \leq k \leq z\}$ be the set of the root constructors of the patterns in the first column of (P). To each constructor c_k, of arity a_k, a new matrix (P_k) is associated. Matrix (P_k) contains the clauses of (P) that may match a vector of values of the format $((c_k\ U_1\ \ldots\ U_{a_k})\ V_2\ldots V_n)$. More precisely, the following table shows how each row of the new matrix is constructed:

p_1^i	row of P_k
x_1	$\_\cdots\_\ p_2^i\cdots p_n^i{:}e_i[x_1\backslash v_1]$
$c_k\ q_1^i\ \ldots\ q_{a_k}^i$	$q_1^i\cdots q_{a_k}^i\ p_2^i\cdots p_n^i{:}e_i$
$c\ q_1^i\ \ldots\ q_a^i (c \neq c_k)$	row deleted in (P_k)

$$\mathcal{C}((v_1\ v_2 \ldots v_n),(P)) = \left\{ \begin{array}{lll} \texttt{match } \texttt{v}_1 \texttt{ with} & & \\ \quad \texttt{c}_1\ \texttt{w}_1 \ldots \texttt{wa}_1 & \rightarrow & \mathcal{C}((w_1\ w_2 \ldots w_{a_1}\ v_2 \ldots v_n),(P_1)) \\ \mid \texttt{c}_2\ \texttt{w}_1 \ldots \texttt{wa}_2 & \rightarrow & \cdots \\ & & \vdots \\ \mid \texttt{c}_z\ \texttt{w}_1 \ldots \texttt{wa}_z & \rightarrow & \mathcal{C}((w_1\ w_2 \ldots w_{a_z}\ v_2 \ldots v_n),(P_z)) \\ \mid \_ & \rightarrow & \mathcal{C}((v_2 \ldots v_n),(P_d)) \end{array} \right.$$

Figure 3: Compilation: the general case

If the set of constructors Σ is not a complete signature, then some clauses in matrix (P) may match a value vector of the format $((c\ U_1 \ldots U_a)\ V_2 \ldots V_n)$, where c is a constructor that does not belong to Σ. To match these cases, a default matrix (P_d) is built:

p_1^i	row of P_d
x_1	$p_2^i \cdots p_n^i : e_i[x_1 \backslash v_1]$
$c\ q_1^i\ q_2^i \cdots q_a^i$	row deleted in (P_d)

The original ordering of the rows is preserved in the new matrices (P_k) and (P_d), so that the clauses are arranged by increasing number. Compilation is then defined by figure 3 (in this figure, the w_j are fresh variables).

Compilation always terminates, since the size of the clause matrix (P) strictly decreases at each recursive call to function $\mathcal{C}$. To see this, consider the lexicographic ordering on the pairs of positive integers $(N_c(P), N_v(P))$, where $N_c(P)$ and $N_v(P)$ are the sums for all the patterns in (P) of the n_c and n_v functions, defined by:

$$\left\{ \begin{array}{rl} n_c(x) &= 0 \\ n_c(c\ p_1 \ldots p_a) &= 1 + n_c(p_1) + \cdots + n_c(p_a) \end{array} \right.$$

$$\left\{ \begin{array}{rl} n_v(x) &= 1 \\ n_v(c\ p_1 \ldots p_a) &= 0 \end{array} \right.$$

Now, consider the *or* function of section 3.2. The initial call to function $\mathcal{C}$ is given at the top of figure 4. Then, there are two possible compilations, depending on whether matching proceeds by examining variable x first or variable y first. The rest of the compilations is easy and we get the two compiled expressions $\mathcal{C}_1(v)$ and $\mathcal{C}_2(v)$ as given in figure 4. These two compilations are syntactically different. They are also semantically different When applied to value $V = (\Omega, \texttt{true})$, the automata generated by compilation schemes $\mathcal{C}_1$ and $\mathcal{C}_2$ give different answers ($\mathcal{C}_1(V) = \Omega$ and $\mathcal{C}_2(V) = \texttt{true} = or(V)$). For any other partial value V, we get $\mathcal{C}_1(V) = \mathcal{C}_2(V) = or(V)$. Therefore, automaton $\mathcal{C}_1(v)$ does not correctly implement function *or*, whereas automaton $\mathcal{C}_2(v)$ does

4 Correct compilation

4.1 Partial correctness

As shown by the $\mathcal{C}_1$ example above, our compilation scheme may not be correct. It satisfies a partial correctness property, though: when the compiled automaton gives a result that is strictly more defined than Ω, this result is correct.

Lemma 4.1 (Partial correctness) *Let E be a set of clauses. Consider any compilation of the matching by E. During this compilation, for any call to function $\mathcal{C}$ and any partial value vector $(V_1\ V_2 \ldots V_n)$, the following equality holds:*

$$\mathcal{C}((V_1\ V_2 \ldots V_n),(P)) \preceq match((V_1\ V_2 \ldots V_n),(P))$$

Proof: By induction on the definition of compilation. We only give the interesting case. In the case of the inductive step *4*, suppose that compilation progresses by a simple matching on variable v_1. Then, value $\mathcal{C}((V_1\ V_2 \ldots V_n),(P))$ is:

$$\begin{array}{lll} \texttt{match } V_1 \texttt{ with} & & \\ \quad \texttt{c}_1\ \texttt{w}_1 \ldots \texttt{wa}_1 & \rightarrow & \mathcal{C}((\texttt{w}_1 \ldots \texttt{wa}_1\ V_2 \ldots V_n),(P_1)) \\ & & \vdots \\ \mid \texttt{c}_z\ \texttt{w}_1 \ldots \texttt{wa}_z & \rightarrow & \mathcal{C}((\texttt{w}_1 \ldots \texttt{wa}_z\ V_2 \ldots V_n),(P_z)) \\ \mid \_ & \rightarrow & \mathcal{C}((V_2 \ldots V_n),(P_d)) \end{array}$$

There are now three cases. In the case where value V_1 equals Ω, then the simple matching on V_1 fails. That is, we get $\mathcal{C}((V_1\ V_2 \ldots V_n),(P)) = \Omega$, where value Ω is always less defined than $match((V_1\ V_2 \ldots V_n),(P))$.

If $V_1 = c\ U_1 \ldots U_a$, where constructor c is not the root constructor of one of the patterns in the first column of (P), then value V_1 matches the default clause of the simple matching and we get:

$$\mathcal{C}((V_1\ V_2 \ldots V_n),(P)) = \mathcal{C}((V_2 \ldots V_n),(P_d))$$

It follows, by induction hypothesis:

$$\mathcal{C}((V_1\ V_2 \ldots V_n),(P)) \preceq match((V_2 \ldots V_n),(P_d)[v_1 \backslash V_1])$$

$$\mathcal{C}((v), \begin{pmatrix} \texttt{\_},\texttt{true} : \texttt{true} \\ \texttt{true},\texttt{\_} : \texttt{true} \\ \texttt{\_},\texttt{\_} \quad : \texttt{false} \end{pmatrix}) = \texttt{match v with x,y} \rightarrow \mathcal{C}((x\ y), \begin{pmatrix} \texttt{\_} & \texttt{true} & : \texttt{true} \\ \texttt{true} & \texttt{\_} & : \texttt{true} \\ \texttt{\_} & \texttt{\_} & : \texttt{false} \end{pmatrix})$$

$$\begin{array}{l|l}
\begin{array}{l}
\texttt{match v with} \\
\quad \texttt{x,y} \rightarrow \texttt{(match x with} \\
\qquad \texttt{true} \rightarrow \mathcal{C}((y), \begin{pmatrix} \texttt{true} : \texttt{true} \\ \texttt{\_} \quad : \texttt{true} \\ \texttt{\_} \quad : \texttt{false} \end{pmatrix}) \\
\quad \texttt{| \_} \quad \rightarrow \mathcal{C}((y), \begin{pmatrix} \texttt{true} : \texttt{true} \\ \texttt{\_} \quad : \texttt{false} \end{pmatrix}))
\end{array}
&
\begin{array}{l}
\texttt{match v with} \\
\quad \texttt{x,y} \rightarrow \texttt{(match y with} \\
\qquad \texttt{true} \rightarrow \mathcal{C}((x), \begin{pmatrix} \texttt{\_} \quad : \texttt{true} \\ \texttt{true} : \texttt{true} \\ \texttt{\_} \quad : \texttt{false} \end{pmatrix}) \\
\quad \texttt{| \_} \quad \rightarrow \mathcal{C}((x), \begin{pmatrix} \texttt{true} : \texttt{true} \\ \texttt{\_} \quad : \texttt{false} \end{pmatrix}))
\end{array}
\\
\qquad \vdots & \qquad \vdots \\
\qquad \vdots & \qquad \vdots \\
\mathcal{C}_1(v) = \begin{cases}
\texttt{match v with} \\
\quad \texttt{x,y} \rightarrow \texttt{(match x with} \\
\qquad \texttt{true} \rightarrow \texttt{(match y with} \\
\qquad\qquad \texttt{true} \rightarrow \texttt{true} \\
\qquad\qquad \texttt{| \_} \quad \rightarrow \texttt{true)} \\
\quad \texttt{| \_} \quad \rightarrow \texttt{(match y with} \\
\qquad\qquad \texttt{true} \rightarrow \texttt{true} \\
\qquad\qquad \texttt{| \_} \quad \rightarrow \texttt{false))}
\end{cases}
&
\mathcal{C}_2(v) = \begin{cases}
\texttt{match v with} \\
\quad \texttt{x,y} \rightarrow \texttt{(match y with} \\
\qquad \texttt{true} \rightarrow \texttt{true} \\
\qquad \texttt{| \_} \quad \rightarrow \texttt{(match x with} \\
\qquad\qquad \texttt{true} \rightarrow \texttt{true} \\
\qquad\qquad \texttt{| \_} \quad \rightarrow \texttt{false))}
\end{cases}
\end{array}$$

Figure 4: The two possible compilations of function *or*

Moreover, for any clause $\vec{p}^i : e_i$ in matrix (P), we have:

$$(p_1^i\ p_2^i \ldots p_n^i) \preceq ((c\ U_1\ \ldots\ U_a)\ V_2 \ldots V_n)$$
$$\text{if and only if}$$
$$p_1^i = x^i \text{ and } (p_2^i \ldots p_n^i) \preceq (V_2 \ldots V_n)$$

$$(p_1^i\ p_2^i \ldots p_n^i) \# ((c\ U_1\ \ldots\ U_a)\ V_2 \ldots V_n)$$
$$\text{if and only if}$$
$$\begin{cases} p_1^i = c'\ q_1^i\ \ldots\ q_{a'}^i \\ \qquad \text{or} \\ p_1^i = x^i \text{ and } (p_1^2 \ldots p_n^i) \# (V_2 \ldots V_n) \end{cases}$$

Thus, for a vector of partial values $\vec{V}$ whose first component is of the format $V_1 = c\ U_1\ \ldots\ U_a$, matchings by matrices (P) and (P_d) are equivalent. In symbols, partial values $match(((c\ U_1\ \ldots\ U_a)\ V_2 \ldots V_n), (P))$ and $match((V_2 \ldots V_n), (P_d)[v_1 \backslash c\ U_1\ \ldots\ U_a])$ are the same. Therefore:

$$\mathcal{C}((V_1\ V_2 \ldots V_n), (P)) \preceq match((V_1\ V_2 \ldots V_n), (P))$$

Finally, if partial value V_1 admits a root constructor c_k, where constructor c_k is the root constructor of a pattern in the first column of (P), then partial value $V_1 = c_k\ U_1\ \ldots\ U_{a_k}$ matches the clause with simple pattern $c_k\ w_1\ \ldots\ w_{a_k}$. That is, value $\mathcal{C}(((c_k\ U_1\ \ldots\ U_{a_k})\ V_2 \ldots V_n), (P))$ reduces to value $\mathcal{C}((U_1 \ldots U_{a_k}\ V_2 \ldots V_n), (P_k))$. By induction hypothesis, we get the inequality $\mathcal{C}((V_1\ V_2 \ldots V_n), (P)) \preceq match((U_1 \ldots U_{a_k}\ V_2 \ldots V_n), (P_k)[v_1 \backslash V_1])$. Whereas, by expanding definitions as we already did above, values $match(((c_k\ U_1\ \ldots\ U_{a_k})\ V_2 \ldots V_n), (P))$ and $match((U_1 \ldots U_{a_k}\ V_2 \ldots V_n), (P_k)[v_1 \backslash V_1])$ are equal. Hence the result:

$$\mathcal{C}((V_1\ V_2 \ldots V_n), (P)) \preceq match((V_1\ V_2 \ldots V_n), (P))$$

□

4.2 Total correctness

We now aim at improving the compilation scheme above, so that it yields a correct compilation, if possible. A compilation is correct, if and only if, for every call to function $\mathcal{C}$ and every partial value vector $(V_1\ V_2 \ldots V_n)$, we have:

$$\mathcal{C}((V_1\ V_2 \ldots V_n), (P)) = match((V_1\ V_2 \ldots V_n), (P))$$

Totally correct automata enjoy optimality properties as defined by other authors. First, the automaton produced by a correct compilation is a "lazy algorithm" in the sense of A. Laville [5]. This means that such an automaton only explores a minimal prefix of the recognized value. Furthermore, a given subpart inside this prefix is never looked at twice. A correct automaton hence enjoy an optimal run-time behavior: it performs a minimal number of tests on matched values. Correct automata also satisfy the other optimality property defined in [8]: they fail to produce a result only on the "minimal" set of the partial values that are not defined enough to match a clause in the initial set E (i.e., a correct automaton gives Ω as a result, if and only if the result of the matching by E is Ω).

The proof of partial correctness carries almost unchanged to total correctness, except for the inductive step *4*. If simple pattern matching is performed on the first value component, V_1, and if V_1 is Ω, then we get: $\mathcal{C}((\Omega\ V_2 \ldots V_n),(P)) = \Omega$. However, it may be the case that $match((\Omega\ V_2 \ldots V_n),(P)) \succ \Omega$, if the whole value vector $(\Omega\ V_2 \ldots V_n)$ matches a clause in matrix (P). A correct compiler must avoid this situation whenever possible.

Definition 4.2 (Directions) *Let (P) be a clause matrix of width n. The column index d such that $1 \leq d \leq n$ is a* direction *for the matching by (P), written $d \in Dir(P)$, iff the following two conditions are met:*

1. *$match((\Omega\ \Omega \ldots \Omega),(P)) = \Omega$.*
2. *There is no vector $\vec{V} = (V_1\ V_2 \ldots V_n)$, such that $\vec{V}$ matches a clause in (P) and $V_d = \Omega$.*

Directions are computable from the matrix (P). Consider the set $Dir_i(P)$ of *directions for the matching by clause number i*, defined as:

$$Dir_i(P) = \{d \in [1..n] \mid match_i(\vec{V},(P)) \Rightarrow V_d \succ \Omega\}$$

Then $Dir(P)$ is the intersection of the $Dir_i(P)$ sets. For instance, at the critical compilation step for function *or*, we have:

$$(P) = \begin{pmatrix} \_ & \texttt{true} & : \texttt{true} \\ \texttt{true} & \_ & : \texttt{true} \\ \_ & \_ & : \texttt{false} \end{pmatrix}$$

and

$match_1(\vec{V},(P)) \Leftrightarrow \texttt{true} \preceq V_2$
$match_2(\vec{V},(P)) \Leftrightarrow \texttt{true} \preceq V_1 \wedge \texttt{true} \# V_2$
$match_3(\vec{V},(P)) \Leftrightarrow \texttt{true} \# V_1 \wedge \texttt{true} \# V_2$

Thus, we get $Dir_1(P) = \{2\}$, $Dir_2(P) = \{1,2\}$, $Dir_3(P) = \{1,2\}$ and $Dir(P) = \{2\}$. See section 5 for a full description of the computation of directions.

Directions gives us a method to test the correctness of a given compilation:

Lemma 4.3 (Correct compilation) *Let E be a set of clauses. A given compilation of the matching by E is correct, if and only if, at each inductive step 4 of the compilation, there exists a direction d in the clause matrix (P) and compilation goes on by a simple matching on variable v_d.*

Proof: See the discussion at the beginning of this section. □

For instance, in the case of the *or* function, compilation $\mathcal{C}_2$ can be stated as correct without testing it on all partial values, since it always performs simple matchings by following directions.

Checking directions only gives a sufficient condition of correctness because this method ignores result expressions. Consider, for instance, the following matching on pairs of booleans:

```
match v with true,true → true | _,_ → Ω
```

After a first trivial inductive step the compilation of such a matching amounts to:

$$\begin{array}{l} \texttt{match v with} \\ \quad \texttt{x,y} \ \rightarrow\ \mathcal{C}((x\ y), \begin{pmatrix} \texttt{true} & \texttt{true} & : \texttt{true} \\ \_ & \_ & : \Omega \end{pmatrix}) \end{array}$$

At this stage, the set of directions $Dir(P)$ is empty. Namely, we have $match_2(\vec{V},(P)) \Leftrightarrow \texttt{true} \# V_1 \vee \texttt{true} \# V_2$ and thus $Dir_2(P) = \emptyset$. However, both possible compilations are correct. Indeed, it is not important whether Ω is associated to any value $\vec{V}$ matching clause number 2 because it is recognized to match clause number 2 —as it is always the case for $\vec{V} = (\texttt{false false})$, for instance—, or because simple pattern matching fails on one of its subcomponents —as it is the case for $\vec{V} = (\Omega\ \texttt{false})$, when left-to-right subcomponent matching order is chosen—.

We deliberately ignore such correct compilations. If the result expressions of the clauses were full ML expressions, and that we identified —a bit quickly— value Ω and the non-termination of a ML program, then it would become undecidable to know whether the value of an expression is Ω or not.

4.3 Finding a correct compilation

Some pattern matching expressions cannot be compiled correctly. Consider a variation on the classical example due to G. Berry, as given in figure 5. Sixteen different compilations are possible. None of them is correct To see this, it is not necessary to completely construct all these automata. The correctness criterion of lemma 4.3 applies at automaton construction time and can be used to prune the search for a correct automaton. Such a limited enumeration is still

```
let G x y z = match (x,(y,z)) with
  (true,(false,_))  →  1
| (false,(_,true))  →  2
| (_,(true,false))  →  3
```

Figure 5: Berry's example

not satisfying, since a pattern matrix with more than one direction may imply some backtracking.

Fortunately, discovering one matrix without a direction during any compilation attempt is sufficient to ensure that there is no correct compilation at all. Due to lack of space, we only sketch the proof of this result.

Proposition 4.4 *Let E be a set of clauses. The following compilation algorithm yields an automaton correctly implementing the matching by E whenever possible.*

Compile pattern matching as described in section 3.3. At the inductive step 4, consider the directions for the matching by matrix (P). Two cases are possible:

1. *If matrix (P) does not have a direction, then fail.*
2. *If matrix (P) has directions, then choose one and continue compilation by a simple matching following this direction.*

Proof: If condition *1* above occurs, then it can be shown that matching by E is not a sequential function in the sense of Kahn-Plotkin (see [3, 2]). Whereas any automaton produced by our method is sequential in this sense. □

5 Implementation

Given a clause matrix (P), a clause number i and a column index d, we want to know whether d belongs to $Dir_i(P)$ or not. This can be expressed as the unused match case detection problem: is the last clause of the matrix $(Q_{(d,i)})$ below satisfiable or not ? That is, does there exist a value vector $\vec{V}$ such that $match_i(\vec{V},(Q_{(d,i)}))$ holds ? The matrix $(Q_{(d,i)})$ is the submatrix of (P) obtained by deleting column d and the clauses after clause i:

$$(Q_{(d,i)}) = \begin{pmatrix} p_1^1 \cdots p_{d-1}^1 & p_{d+1}^1 \cdots p_n^1 : e_1 \\ p_1^2 \cdots p_{d-1}^2 & p_{d+1}^2 \cdots p_n^2 : e_2 \\ & \vdots \\ p_1^i \cdots p_{d-1}^i & p_{d+1}^i \cdots p_n^i : e_i \end{pmatrix}$$

Lemma 5.1 *Let (P) be a clause matrix. Let i be a clause number and d be a column index in matrix (P). Let $(Q_{(d,i)})$ be as described above. Then, index d is not a direction for matrix (P), if and only if pattern p_d^i is a variable and the last clause of matrix $(Q_{(d,i)})$ is satisfiable.*

Proof: In the case where $p_d^i = c\ q_1 \dots q_a$ is not a variable, then any value vector $(V_1\ V_2 \dots V_n)$ matching clause number i in matrix (P) is such that component V_d is an instance of pattern $c\ q_1 \dots q_a$. Therefore, we get $V_d \succ \Omega$. Otherwise, let $V_1, \dots, V_{d-1}, V_{d+1}, \dots, V_n$ be any $n-1$ partial values. The following equality can be shown by expanding definitions:

$$\begin{gathered} match_i((V_1 \dots V_{d-1}\ \Omega\ V_{d+1} \dots V_n),(P)) \\ \| \\ match_i((V_1 \dots V_{d_1}\ V_{d+1} \dots V_n),(Q_{(d,i)})) \end{gathered}$$

□

We now give an algorithm to solve the unused match case detection problem in the general case. Given a pattern matrix (P), of size n by m, the algorithm below computes the truth value of the formula $\mathcal{F}(P) = \exists\, \vec{V}\ match_m(\vec{V},(P))$. This algorithm closely follows the compilation algorithm itself:

1. If the rows of matrix (P) are empty, or if its first row contains only variables, then the value of $\mathcal{F}(P)$ depends on the number m of rows in (P). If $m = 1$, then $\mathcal{F}(P) =$ `true`, since any instance of $\vec{p}^{\,1}$ matches the last (and only) clause of matrix (P). Otherwise, $\mathcal{F}(P) =$ `false`.
2. In all the other cases, let us choose a column index. Suppose that index 1 is chosen. Let Σ be the set of the root constructors of the patterns in the first column of (P). To each constructor c_k in Σ, a new pattern matrix (P_k) is associated as in the compilation of pattern matching (section 3.3). If set Σ is not a complete signature or if Σ is the empty set, then a default matrix (P_d) is also considered. There are two subcases:
   (a) If $p_1^m = x$, then let $\vec{V}$ be a value vector satisfying the last clause of (P). If V_1 has a root constructor, then the matching by matrix (P) is equivalent to the matching by one of the matrices (P_k) or (P_d). Otherwise, if $V_1 = \Omega$, then, because the matching predicate is monotonic, any value vector $\vec{U} = (U_1\ V_2 \dots V_n)$ such that $U_1 \succ \Omega$ matches clause m as $\vec{V}$ does. Therefore, $\mathcal{F}(P)$ is true, if and only at one at least of the formulas $\mathcal{F}(P_1)$, $\mathcal{F}(P_2)$, $\dots \mathcal{F}(P_z)$ or $\mathcal{F}(P_d)$ is.

(b) If $p_1^m \succ x$, then let c be the root constructor of p_1^m. If c belongs to Σ, then there exists a matrix (P_k) such that $\mathcal{F}(P) = \mathcal{F}(P_k)$. Otherwise, we have $\mathcal{F}(P) = \mathcal{F}(P_d)$.

Regarding the efficiency of this algorithm, it can be observed that the number of calls to function $\mathcal{F}$ is bounded by the number of calls to function $\mathcal{C}$, when compilation is done by making the same choices at critical steps. This upper bound is reached when $\mathcal{F}(P)$ is false and when the last row of matrix (P) contains only variables. As shown by the example given in the appendix, the number of calls to function $\mathcal{C}$ can be quite large. Although we do not know whether this upper bound is indeed reached or not in the worst cases, experiments showed us that a naive implementation of function $\mathcal{F}$ may lead to important computations. Fortunately, we were able to avoid this misbehavior by using the following three heuristics:

1. Matrix (P) itself can be reduced. Let $\vec{p}^i$ and $\vec{p}^j$ be two rows inside matrix (P) (i.e. $i < m$ and $j < m$), such that $\vec{p}^i \preceq \vec{p}^j$. For any value vector $\vec{V}$ such that $\vec{p}^i \# \vec{V}$, we necessarily have $\vec{p}^j \# \vec{V}$. That is, pattern vector $\vec{p}^j$ is useless for the computation of $\mathcal{F}(P)$ and matrix (P) can be simplified by only retaining the pattern rows that are minimal for the definition ordering. This simplification of matrix (P) is particularly worthwhile when some pattern row contain a lot of variables.

2. When there is a default matrix (P_d), it is tested first. This amounts to making the assumption that, if there exists a value vector satisfying the last row of (P), then its components are likely not to appear inside matrix (P).

3. We also attempt to minimize the size and number of the matrices (P_1), (P_2) $\ldots(P_z)$, by a good choice of the column to examine at step 2-(a). For each column, characterized by its index i, let $z(i)$ be the number of different root constructors in column i and $v(i)$ be the number of variables in column i. Let then $r(i)$ be the total number of rows in the matrices (P_1), (P_2) $\ldots(P_{z(i)})$, we have $r(i) = z(i)v(i) + m$ or $r(i) = z(i)v(i) + m - v(i)$, depending on whether there is a default matrix (P_d) or not. We select a column with a minimal $r(i)$. If there are several columns such that $r(i)$ is minimal, then we favor one with a minimal number of different root constructors $z(i)$. Other size measures have been tested, including matrices surfaces (number of rows $\times$ number of columns) and the function N_c of section 3.3. Choosing a good measure is not easy and this heuristic is less efficient than the two others.

Regarding the efficiency of the computation of set $Dir(P)$, it is usually not necessary to compute all the $Dir_i(P)$ sets First, if there is a column in matrix (P) which contains no variable, then, by lemma 5.1, the index of this column is a direction. Such a direction is an *obvious direction* and knowing just one direction is enough to apply the compilation algorithm of proposition 4.4. Otherwise, there is no obvious direction in matrix (P) and set $Dir(P)$ has to be tested for emptyness. If index d does not belong to $Dir_i(P)$, then, for any other clause number j, we need not check whether index d belongs to $Dir_j(P)$ or not, since we already know that d is not a direction for the whole matrix (P). Of course, the $Dir_i(P)$ sets are examined following increasing clause numbers i, so that index checkings are avoided when matrices $Q_{(d,i)}$ are large. That is, we compute $\mathcal{D}_m = Dir(P)$, where $\mathcal{D}_1 = Dir_1(P)$ and $\mathcal{D}_{i+1} = \{ d \in \mathcal{D}_i \mid d \in Dir_{i+1}(P) \}$. If matrix (P) has no direction, then there exists a clause number max, such that $\mathcal{D}_{max} \neq \emptyset$ and $\mathcal{D}_{max+1} = \emptyset$. In such case, the column indices in $\mathcal{D}_{max}$ are called *partial directions*.

The other approaches to the compilation of lazy pattern matching [5, 8] involve the explicit computation of the set of value vectors matching the clauses of matrix (P). Let $\mathcal{M}$ be this set. We have $\mathcal{M} = \bigcup_{i=1}^{m} \mathcal{M}_i$, where $\mathcal{M}_i = \{\vec{V} \mid match_i(\vec{V}, (P))\}$. In [5] set $\mathcal{M}$ is described by its *minimal generators*, that is, by the subset of its least defined elements. In [8] each set $\mathcal{M}_i$ is represented by a normalized constrained pattern that can be seen as the disjunctive normal form of the characteristic proposition

$$\mathcal{X}_i(V_1, V_2, \ldots V_n) = (\bigwedge_{j=1}^{i-1} \bigvee_{k=1}^{n} p_k^j \# V_k) \bigwedge (\bigwedge_{k=1}^{n} p_k^i \preceq V_k)$$

Direct implementation of these two representations for set $\mathcal{M}$ leads to data structures whose size grow exponentially with the size of the input matrix (P). Our approach, by directly computing directions, avoids such an exponential space behavior.

6 Conclusion

We have described a compiler for lazy pattern matching that produces a correct automaton whenever there exists one. When there are several correct automata, our compiler attempts to generate one with a reasonable size, using heuristic 3 above. When there is no correct automaton, the compiler issues a warning message and outputs a partially correct automaton (in the sense of section 4.1), still attempting to minimize its size, using partial directions and heuristic 3. Our work resulted in the first integration of the correct compilation of lazy pattern matching in a lazy ML compiler [6]. Furthermore, we developed a simple presentation of the theory of lazy pattern matching.

In some rare occasions, the heuristics we use can be defeated and the size of the automaton gets very large —In fact, as shown in the appendix, some set of clauses defeat any heuristic— Other compilers producing tree-like pattern matching automata, such as SML-NJ or CAML [11], face the same problem. In [1, 10] an alternative technique of compilation is presented: pattern matching expressions are compiled using a backtracking construct. This technique leads to matching automata whose size is linear in the size of the input program. A similar approach may be possible in our case, but it would probably imply loosing the optimal run-time behavior.

In a preliminary version of this work, we suggested the following other direction for further work: analyze the complexity of the unused match case detection problem. We recently learned that this problem is NP-complete [9].

Acknowledgements

I thank X. Leroy for his editorial help and A. Suárez for fruitful discussions.

References

[1] L. Augustsson, "*Compiling pattern matching*". FPCA'85.

[2] G. Huet, J.-J. Lévy, "*Call by Need Computations in Non-Ambiguous Linear Term Rewriting Systems*". INRIA, technical report 359, 1979.

[3] G. Kahn, G. Plotkin, "*Domaines concrets*", Rapport IRIA Laboria 336, 1978.

[4] R. Kennaway, "*The Specificity Rule for Lazy Pattern Matching in Ambiguous Term Rewriting Systems*". ESOP'90.

[5] A. Laville, "*Comparison of Priority Rules in Pattern Matching and Term Rewriting*". Journal of Symbolic Computation (1991) 11, 321–347.

[6] L. Maranget, "*GAML: A Parallel Implementation of Lazy ML*". FPCA'91.

[7] R. Milner, M. Tofte, R. Harper, "*The Definition of Standard ML*". The MIT Press.

[8] L. Puel, A. Suárez, "*Compiling Pattern Matching by Term Decomposition*". LFP'90.

[9] R.C. Sekar, R. Ramesh and I.V. Ramakrishnan, "*Adaptive Pattern Matching*". ICALP'92.

[10] P. Wadler, chapter on the compilation of pattern matching in: S. L. Peyton Jones, "*The Implementation of Functional Programming Languages*". Prentice-Hall, 1987.

[11] P. Weis, "*The CAML Reference manual*" Version 2.6.1, INRIA Technical Report 121 1990.

Appendix

Let n be a strictly positive integer. Let I_n be the identity matrix of size n by n. And let then A_n be the pattern matrix of size $n(n-1)/2$ by n inductively defined as:

$$A_1 = () \qquad A_n = \left(\begin{array}{c|c} 2\ 2\ \ldots 2 & \_\ \_\ \ldots \_ \\ \hline I_{n-1} & A_{n-1} \end{array}\right)$$

For instance, we have:

$$A_5 = \begin{pmatrix} 2 & 2 & 2 & 2 & \_ & \_ & \_ & \_ & \_ & \_ \\ 1 & \_ & \_ & \_ & 2 & 2 & 2 & \_ & \_ & \_ \\ \_ & 1 & \_ & \_ & 1 & \_ & \_ & 2 & 2 & \_ \\ \_ & \_ & 1 & \_ & \_ & 1 & \_ & 1 & \_ & 2 \\ \_ & \_ & \_ & 1 & \_ & \_ & 1 & \_ & 1 & 1 \end{pmatrix}$$

Given any column index i in A_n, there are $n-1$ pattern rows in A_n whose component number i is 1 or _. Thus, as any two pattern rows in A_n are incompatible, any value vector whose component number i is 1 may possibly match $n-1$ pattern rows. This is also true for any value vector whose component number i is 2. Let us call $\mathcal{P}$ this property. As a consequence of property $\mathcal{P}$, whichever column index is chosen, the critical compilation step *4* will yield at least two recursive calls to function $\mathcal{C}$. And the pattern matrix given as an argument in these recursive call will have $n-1$ rows.

Now, apply the $\mathcal{C}$ compilation scheme to matrix A_n, not trying to produce a totally correct automaton (i.e., generate all possible automata). Because all intermediate pattern matrices that arise while compiling A_n enjoy a property similar to property $\mathcal{P}$, it can be shown that the size of the generated automata is greater than 2^n. That is, if $N = n^2(n-1)/2$ is the size of A_n, the size of these automata grows at least as $2^{\sqrt[3]{N}}$.