INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Two Techniques for Compiling Lazy Pattern Matching

Luc Maranget

N ° 2385
Octobre 1994

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel

Rapport de recherche

1994

N 0249-6399

Two Techniques for Compiling Lazy Pattern Matching

Luc Maranget*

Programme 2 — Calcul symbolique, programmation et génie logiciel

Projet Para

Rapport de recherche n°2385 — Octobre 1994 — 37 pages

Abstract: In ML style pattern matching, pattern size is not constrained and ambiguous patterns are allowed. This generality leads to a clear and concise programming style but is challenging in the context of lazy evaluation. A first challenge concerns language designers: in lazy ML, the evaluation order of expressions follows actual data dependencies. That is, only the computations that are needed to produce the final result are performed. Once given a proper (that is, non-ambiguous) semantics, pattern matching should be compiled in a similar spirit: any value matching a given pattern should be recognized by performing only the minimal number of elementary tests needed to do so. This challenge was first met by A. Laville.

A second challenge concerns compiler designers. As it stands, Laville's compilation algorithm cannot be incorporated in an actual lazy ML compiler for efficiency and completeness reasons. As a matter of fact, Laville's original algorithm did not fully treat the case of integers in patterns and can lead to explosions both in compilation time and generated code size. This paper provides a complete solution to that second challenge. In particular, the well-known (and size-efficient) pattern matching compilation technique using backtracking automata is here introduced for the first time into the world of lazy pattern matching.

(Résumé : tsvp)

*Luc.Maranget@inria.fr

Deux techniques de compilation du filtrage paresseux

Résumé : Le langage de programmation ML possède une construction de filtrage très générale, les motifs à reconnaître peuvent être arbitrairement profonds et ambigus entre eux. Cette flexibilité autorise un style de programmation clair et concis, mais elle soulève deux questions importantes dans le contexte de l'évaluation paresseuse. La première question s'adresse aux concepteurs de ML : dans les langages paresseux, l'ordre l'évaluation des expressions se fait selon les dépendances entre données, c'est à dire que seuls les calculs utiles à la production du résultat final sont effectués. Une fois une sémantique convenable (c'est à dire non-ambigüe) donnée au filtrage, il convient de le compiler conformément à cette sémantique et dans un esprit "paresseux" : toute valeur filtrée doit être reconnues en effectuant le minimun possible de tests élémentaires. Cette question a été résolue pour la première fois par A. Laville.

La seconde question est d'ordre plus pratique et concerne les concepteurs de compilateurs. Tel quel, l'algorithme d'A. Laville n'est pas utilisable dans dans un compilateur ML pour des raisons de complétude et d'efficacité. En effet, cet algorithme ne traite pas le cas du filtrage par des motifs de type entier. En outre, il peut, dans certains cas, produire des automates de taille exponentielle en la taille des motifs compilés. Ce travail résoud ces questions. Une approche plus simple que l'approche initiale permet de traiter le cas des motifs de signature infinie. Ensuite, un nouvel algorithme de compilation du filtrage paresseux est proposé. Les automates cibles du nouvel algorithme utilisent une construction d'échappement qui autorise un partage de code suffisant pour garantir que leur taille est linéaire en la taille des motifs compilés.

1 Introduction

Pattern matching is a key feature of the ML language. It provides 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 [Weis, 1990, Leroy *et al.*, 1993] 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 the 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 [Milner *et al.*, 1991], 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 in LML [Augustsson, 1985], 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
```

The patterns can be checked from left to right, as it is usually the case (function $\mathtt{F}_1$, below), or from right to left (function $\mathtt{F}_2$).

```
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 the variable y is bound to false, the test on x is useless. This can be avoided by testing y before x, as in $\mathtt{F}_2$.

Worse, consider the application F $\perp$ false, where $\perp$ is a non-terminating computation. In strict ML, function arguments are reduced before calling the functions, so that both compilations $\mathtt{F}_1$ $\perp$ false and $\mathtt{F}_2$ $\perp$ false do not terminate. In lazy ML, function arguments are not evaluated until their values are actually needed. Therefore, $\mathtt{F}_1$ will loop by trying to evaluate $\mathtt{x} = \perp$, whereas $\mathtt{F}_2$ will give the answer 2. In the spirit of lazy evaluation, a result should be given whenever possible. Thus, a lazy compiler should compile the function F as $\mathtt{F}_2$, not as $\mathtt{F}_1$.

In the previous example, there is a good reason to claim function $\mathtt{F}_2$ to be more "correct" than function $\mathtt{F}_1$. Indeed, both functions output the same result when given the same input, except in one case, where $\mathtt{F}_2$ gives the result 2, whereas $\mathtt{F}_1$ does not terminate. Unfortunately, things are not always that simple and there are pattern matching definitions that cannot be compiled correctly. Consider, for instance, Berry's famous example:

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

One easily checks that there does not exist a correct compilation of `B` and that one cannot order the possible compilation by their termination properties. By "possible" compilation, I mean here some automaton that will examine all or some of the variables `x`, `y` and `z`, one after the other. Let us consider for instance an automaton that examines `x` first. Because such an automaton will always loop by engaging in the non-terminating computation $\perp$, it cannot output the correct value `3`, when given the input `x` $= \perp$, `y` $=$ `true` and `z` $=$ `false`, whereas other automata that start by examining `y` can. Similarly no automata examining `y` first will ever produce the correct result `2` as a compilation of the function call `B false` $\perp$ `true`, whereas some of the automata examining `x` first will.

The task of a lazy ML compiler is thus twofold. When given a pattern matching definition, it must both determine whether this definition can be compiled correctly or not and produce a correct compilation when possible. This problem has first been solved in the case of non-overlapping patterns by Huet and Lévy [Huet and Lévy, 1979]. Given a set of possibly overlapping patterns, Laville [Laville, 1991] shows how to replace them, when possible, by an equivalent set of non-overlapping patterns, compiled using Huet and Lévy's technique. Laville's method is not complete, since it cannot treat the case of datatypes with infinite signatures (such as the type of integers). As a matter of fact, in presence of infinite signatures, Laville's equivalent set of non-overlapping patterns can also be infinite. Following a similar idea, Suárez and Puel [Puel and Suárez, 1990] 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 the pattern matching on the constrained patterns with an extension of Huet and Lévy's technique. Although the Suárez and Puel's approach does consider infinite signatures, its direct application in a compiler remains problematic, since the size of the constrained patterns can be exponential in the size of the initial ambiguous patterns. I experimented such a misbehavior while implementing Suárez and Puel technique, even on very ordinary pattern matching definition.

In this paper, I take a simpler approach: compilation operates directly on overlapping patterns. I first recall the semantics of lazy pattern matching, as given by Laville [Laville, 1991]. I then present two direct compilation techniques that preserve this semantics whenever possible. These technique are presented as source-to-source transformations. They differ by the nature of the target automata they produce. The first technique produces *tree-like automata*. The main advantage of tree-like automata is simplicity, so that the associated compilation scheme is straightforward and its correctness proof is easy. Unfortunately, the price paid for this simplicity is a potential explosion in size of the output automata. The second and more sophisticated technique produces *automata with failures*. These automata possess a static exception construct that enables some code sharing. The sharing introduced in *automata with failures* is sufficient to guarantee that the size of the output automaton is linear in the size of the input pattern matching definition. Both compilation schemes have been integrated into the GAML compiler [Maranget, 1991] for lazy ML. Finally, I compare these schemes one with another and with previous non-lazy approaches, both from the theoretical and practical points of view.

This paper extends on [Maranget, 1992] in many important ways. In particular, the treatment of automata with failures is entirely new (section 4). A thorough presentation of the full compilation algorithm is given (section 5.2), as well as a discussion of the main theoretical an pragmatical aspects of lazy pattern matching (section 6).

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:

$$\texttt{type tree } \alpha \texttt{ = Leaf } \alpha \texttt{ | Node (tree } \alpha\texttt{) } \alpha \texttt{ (tree } \alpha\texttt{)}$$

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 respective arities 1 and 3. The set $\{\texttt{Leaf}, \texttt{Node}\}$ is the *signature* of the type `tree` α. There is at least one predefined type with an infinite signature: the integer type `int`, all integers being seen 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 from constructors and the symbol Ω:

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

Given a partial value $V = c\ V_1\ V_2 \ldots V_a$, we say that constructor c is the *root constructor* of V. We only consider values that are well typed in the standard sense, with the 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 computations results. 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 :*

$$\begin{cases} U = \Omega \\ \quad \textit{or} \\ \begin{cases} U = c\ U_1 \ldots U_a,\ V = c\ V_1 \ldots V_a \\ \quad \textit{and} \\ \textit{for all } i \textit{ in } [1 \ldots a],\ U_i \preceq V_i \end{cases} \end{cases}$$

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

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

$$\textit{Partial terms} \quad \mathcal{T}_\Omega: \quad M ::= \Omega \mid v \mid c\ M_1\ M_2 \ldots M_a$$

A *substitution*, written σ, is a morphism on partial terms, that is, a function on partial terms such that: $\sigma(c\ M_1\ M_2 \ldots M_a) = c\ \sigma(M_1)\ \sigma(M_2) \ldots \sigma(M_a)$. Thus, a substitution is totally defined by its values on variables. The *domain* of a substitution σ is the set of variables v such that we have $\sigma(v) \neq v$. In practice, only substitutions with finite domains are considered. Such a substitution will often 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 $[1 \ldots n]$, the variable v_i to the partial term M_i. Application of an environment to a partial term is written $N[v_1 \backslash M_1,\ v_2 \backslash M_2, \ldots, v_n \backslash M_n]$. For any partial term M, the partial value M_Ω is obtained by substituting Ω for all variables in M (that is, $M_\Omega = M[v_1 \backslash \Omega,\ v_2 \backslash \Omega, \ldots, v_n \backslash \Omega]$, where $v_1, v_2, \ldots, v_n$ are the variables of term M).

Given two substitutions σ and ρ with disjoint domains, the union $\sigma \cup \rho$ is the substitution whose domain is the set union of the domains of σ and ρ and that coincides with σ and ρ on their respective domains.

2.2 Patterns

Patterns are strict linear terms, that is, partial terms without Ω such that the same variable does not appear more than once in them. Pattern variables 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. The term M is an instance of the pattern p, written $p \preceq M$, if and only if there exists a substitution σ such that $\sigma(p) = M$.*

Note that if M is an instance of p, then there exists an unique substitution σ such that $\sigma(p) = M$ and whose domain is minimal. In the following, this unique substitution will be implicitly selected, when I talk about "the" substitution σ such that $\sigma(p) = M$.

The instantiation relation is closely related to the definition ordering.

Lemma 2.1 *Let p be a pattern and M be a partial term. The following equivalence holds:*

$$p \preceq M \textit{ if and only if } p_\Omega \preceq M_\Omega$$

Proof: Easy induction on p, using 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 $p \# M$ holds, if and only if

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

When pattern p and partial term M are not incompatible, we write $p \uparrow M$. The following equivalence properties, holding for any pattern p and any partial term M, directly follow from lemma 2.1:

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

The partial term M can be a pattern q. If p and q are compatible, then they are also said to be *ambiguous* or *overlapping*. Indeed, as a consequence of lemma 2.1, two patterns are compatible if and only if they admit a common instance.

2.3 The matching function

Pattern matching is usually formalized as a predicate on partial values [Huet and Lévy, 1979, Laville, 1991]. We prefer a representation as a function over partial values, closer to pattern matching in ML.

A *clause* is a triple (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 $p^i\colon e_i$. 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 $E = \{p^i\colon e_i \mid i \in I\}$, 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. To simplify, first assume that clause numbers just express the natural textual clause ordering meant by the programmers, when they write one clause after (under) another.

Definition 2.3 (Matching predicate (Laville)) *Let $E = \{p^1\colon e_1, p^2\colon e_2, \ldots, p^m\colon e_m\}$ be a set of clauses. Let V be a partial value. The value V* matches *clause number i in E, written $match_i[E](V)$, 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 2.4 (Matching function) *Let E be a set of clauses. For any partial value V, we define the partial value $match[E](V)$ as follows:*

- *If the value V matches clause number i, we take $match[E](V) = \sigma(e_i)$, where σ is the substitution such that $\sigma(p^i) = V$.*
- *Otherwise, V is a* non-matching value *and we take $match[E](V) = \Omega$.*

Observe that $match[P]$, the traditional matching predicate defined by the indexed set of patterns $P = \{p^1, p^2, \ldots, p^m\}$ can be simulated by the matching function $match[E]$, where E is the set of clauses $\{p^1{:}\,\mathtt{true}, p^2{:}\,\mathtt{true}, \ldots, p^m{:}\,\mathtt{true}\}$. Obviously, any value V matches some pattern in P, if and only if we have $match[E](V) = \mathtt{true}$.

It is easy to show that, given a set of clauses E, the $match[E]$ function is monotonic over partial values. Other rules than the textual priority rule (definition 2.3) can be used to resolve ambiguity in patterns: in particular, the specificity rule [Kennaway, 1990]. 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 [Laville, 1991].

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 $E = \{(x_1, \mathtt{true}) : \mathtt{true}, (\mathtt{true}, x_2) : \mathtt{true}, (x_3, x_4) : \mathtt{false}\}$ and the function $or(V) = match[E](V)$ (Here "`,`"is the pair constructor, the sole constructor of the pair type). In ML syntax, we have:

$$
\begin{array}{lll}
or(V) = & \multicolumn{2}{l}{\mathtt{match}\ V\ \mathtt{with}} \\
 & \mathtt{\_,true} & \rightarrow \mathtt{true} \\
| & \mathtt{true,\_} & \rightarrow \mathtt{true} \\
| & \mathtt{\_,\_} & \rightarrow \mathtt{false}
\end{array}
$$

There is a finite number of partial values of type $\mathtt{bool} \times \mathtt{bool}$. From definition 2.4, we get:

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

Note that *or* is *not* the "parallel or" function *por*, since, by definition, we have $por(\Omega, \mathtt{true}) = por(\mathtt{true}, \Omega) = \mathtt{true}$.

It may seem that the definition of pattern matching might be simplified by replacing the second condition $\forall j < i,\ p^j \# V$ 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 $match[E](V)$ defined by:

$$\mathtt{match}\ \mathrm{V}\ \mathtt{with}\ \mathtt{1}\ \rightarrow\ \Omega\ |\ \mathtt{\_}\ \rightarrow\ \mathtt{2}$$

The above modified definition of the matching function would give us: $match[E](\Omega) = \mathtt{2} \not\preceq match[E](\mathtt{1}) = \Omega$.

2.4 Pattern matching on vectors

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:

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

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 *pattern matrix* (P) and a *term vector* (E) written as:

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

In pattern p_j^i, the index i is the clause or row number, whereas j is the column index.

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

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

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

It is then straightforward to extend the definition of the matching predicate to vectors of partial values $\vec{V}$ and pattern matrices (P):

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

Finally, once substitutions are extended to operate on vectors in the natural way: $\sigma(\vec{p}) = (\sigma(p_1)\,\sigma(p_2) \ldots \sigma(p_n))$, the matching function also extends to vectors: if $\vec{V}$ matches the row number i in matrix (P), then there exists a substitution σ such that $\sigma(\vec{p}^{\,i}) = \vec{V}$ and take $match[(P),(E)](\vec{V}) = \sigma(e_i)$. Otherwise, vector $\vec{V}$ does not match any clause in $(P),(E)$ and take $match[(P),(E)](\vec{V}) = \Omega$.

Another natural use of pattern matching on vectors is getting rid of superfluous tuples. As we just saw, a pattern matching definition can be given by a pattern matrix (P) of size $n \times m$ and a result vector (E) of size m. For instance, the *or* function of the previous section can be simplified, provided the value of $or(\Omega)$ is of no importance.

$$\texttt{or}(x\ y) = match[\begin{pmatrix} \texttt{\_} & \texttt{true} \\ \texttt{true} & \texttt{\_} \\ \texttt{\_} & \texttt{\_} \end{pmatrix}, \begin{pmatrix} \texttt{true} \\ \texttt{true} \\ \texttt{false} \end{pmatrix}](x\ y)$$

Here also, an alternative formulation as a ML definition exists:

```
let or _    true = true
|   or true _    = true
|   or _    _    = false
```

If the names of the arguments of the function *or* are significant, it can equivalently be written as follows:

```
let or x y = match (x,y) with
  (_,true)  →  true
| (true,_)  →  true
| (_,_)     →  false
```

3 First technique

3.1 Tree-like automata

Tree-like automata are the simplest and most natural among sequential automata. Other authors call them decision trees or matching trees. Every state in a tree-like automata is characterized by some position u to examine and some already examined prefix N; both position and prefix are relative to the currently examined term. Every transition coming out of a state is labelled by some constructor symbol c, that can take place at position u. These automata are described here as ML programs. These particular ML programs are nested simple matchings, simple matching being the natural representation of elementary comparison in ML.

Simple matchings:

F ::=

$\texttt{match}\ x\ \texttt{with}\ p \rightarrow A\ \{\texttt{|}\ p \rightarrow A\}^*\ [\texttt{|}\ \texttt{\_} \rightarrow A]$

Simple patterns:

p ::= $c\ x_1\ x_2 \ldots x_a$ and p is linear

Automata:

A	::=	F	nested matching
	|	M	Partial term

A typical simple matching is thus written as follows:

$$F = \left\{ \begin{array}{lll} \texttt{match}\ x\ \texttt{with} & & \\ \quad c_1\ x_1\ x_2 \ldots x_{a_1} & \rightarrow & A_1 \\ \texttt{|} \quad\quad \vdots & & \\ \texttt{|}\ c_z\ x_1\ x_2 \ldots x_{a_z} & \rightarrow & A_z \\ \texttt{|}\ \texttt{\_} & \rightarrow & A_d \end{array} \right.$$

We further enforce that all *recognized constructors* c_1, c_2,..., c_z are different and belong to the same type. When present, the last clause $\_ \rightarrow A_d$ is the *default clause*.

The functions `F1` and `F2` from the introduction are example of tree-like automata, provided the `if ... then ... else ...` construct is translated into the equivalent `match ... with true` $\rightarrow$ `... | false` $\rightarrow$ ... construct.

The "semantics" of tree-like automata is given as a proof of some judgement $\sigma \vdash A \Rightarrow U$, read: "automaton A gives value U as a result, when executed in environment σ". These proofs are built from the following inference rules and axioms:

$$\frac{\sigma \cup [x \backslash c_k\ V_1 \ldots V_{a_k}] \cup [x_1 \backslash V_1, \ldots, x_{a_k} \backslash V_{a_k}] \vdash A_k \Rightarrow U}{\sigma \cup [x \backslash c_k\ V_1 \ldots V_{a_k}] \vdash \texttt{match}\ x\ \texttt{with}\ \ldots\ \mid\ c_k\ x_1 \ldots x_{a_k} \rightarrow A_k \ldots \Rightarrow U}\ \text{(constr)}$$

$$\frac{\sigma \cup [x \backslash c\ V_1 \ldots V_a] \vdash A_d \Rightarrow U, \qquad \text{(where } c \text{ is not a recognized constructor)}}{\sigma \cup [x \backslash c\ V_1 \ldots V_a]\ \vdash \texttt{match}\ x\ \texttt{with} \ldots\ \mid\ \_ \rightarrow A_d \Rightarrow U}\ \text{(default)}$$

$$\sigma \cup [x \backslash \Omega] \vdash \texttt{match}\ x\ \texttt{with} \ldots \Rightarrow \Omega\ \text{(failure)} \qquad\qquad \sigma \vdash M \Rightarrow \sigma(M)\ \text{(success)}$$

Simple matchings examine the value V bound to some variable x. There are three possible cases:

- V has a root constructor c_k, which is recognized (rule (constr)). Analysis of the input values will proceed by using the right-hand side of the matched clause $c_k\ x_1\ x_2 \ldots x_a \rightarrow A_k$, after some appropriate extension of the current environment.
- V has a root constructor, which is not recognized (rule (default)). Analysis of the input values will proceed by using the right-hand side of the default clause. (Note that no default clause is needed when the constructors recognized by a simple matching make up a full signature).
- V is not defined enough to make a decision (rule (failure)).

The last axiom (success) terminates any successful analysis, the running automaton is reduced to a partial term, which is is returned, once its variables have been properly substituted.

A slightly technical point here deserves mention: the rules for evaluating simple matchings assume some implicit correctness constraints. All judgements $\sigma \vdash F \Rightarrow U$ must obey the following constraints:

1. The examined variable x belongs to the domain of σ.
2. Simple pattern variables x_1, x_2,... do not belong to the domain of σ.

3.2 Compilation

Compilation is defined as a function $\mathcal{C}$, mapping general pattern matching expressions to nested simple pattern matching expressions. More precisely, $\mathcal{C}$ takes three arguments and a typical call is written $\mathcal{C}(\vec{x}, (P), (E))$. The first argument $\vec{x}$ is a linear vector of n variables $(x_1\ x_2 \ldots x_n)$, it abstracts the partial value vectors given as arguments to the compiled matching. In other words, the value

$\mathcal{C}((V_1\ V_2 \ldots V_n), (P), (E))$ is defined by the execution of the automaton $\mathcal{C}((x_1\ x_2 \ldots x_n), (P), (E))$ in the environment $[x_1\backslash V_1,\ x_2\backslash V_2, \ldots, x_n\backslash V_n]$:

$$[x_1\backslash V_1, \ldots, x_n\backslash V_n] \vdash \mathcal{C}((x_1 \ldots x_n), (P), (E)) \Rightarrow \mathcal{C}((V_1 \ldots V_n), (P), (E))$$

The second argument (P) is a pattern matrix of size n by m. Initially, (P) is the pattern matrix to be compiled. Later on, it represents the yet unprocessed subparts of the initial patterns. Finally, the third argument (E) is a term vector of size m. Initially, it is made of the right-hand sides of the compiled clauses. Later on, it represents the result terms that still can be reached at some compilation stage.

The function $\mathcal{C}$ is inductively defined as follows:

1. If $\vec{x}$ is of length zero ($n = 0$), then the compilation is finished. Either the pattern matrix is empty ($m = 0$) and matching will always fail:

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

   Otherwise ($m > 0$), there is at least one (empty) row in matrix (P) and the first result expression is the result of the whole matching:

$$\mathcal{C}((), \begin{pmatrix} \\ \\ \vdots \\ \\ \end{pmatrix}, \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{pmatrix}) = e_1$$

2. If the first row of matrix (P) is made of variables only, then matching will always succeed and give the first component of vector (E) as its result:

$$\mathcal{C}((x_1\ x_2 \ldots x_n), \begin{pmatrix} y_1 & y_2 & \cdots & y_n \\ p_1^2 & p_2^2 & \cdots & p_n^2 \\ & & \vdots & \\ p_1^m & p_2^m & \cdots & p_n^m \end{pmatrix}, \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{pmatrix}) = e_1[y_1\backslash x_1,\ y_2\backslash x_2, \ldots, y_n\backslash x_n]$$

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

$$\mathcal{C}((x_1\ x_2 \ldots x_n), \begin{pmatrix} y^1 & p_2^1 & \cdots & p_n^1 \\ y^2 & p_2^2 & \cdots & p_n^2 \\ & & \vdots & \\ y^m & p_2^m & \cdots & p_n^m \end{pmatrix}, \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{pmatrix})$$

$$||$$

$$\mathcal{C}((x_2 \ldots x_n), \begin{pmatrix} p_2^1 \cdots p_n^1 \\ p_2^2 \cdots p_n^2 \\ \vdots \\ p_2^m \cdots p_n^m \end{pmatrix}, \begin{pmatrix} e_1[y^1\backslash x_1] \\ e_2[y^2\backslash x_1] \\ \vdots \\ e_m[y^m\backslash x_1] \end{pmatrix})$$

4. Matching can also progress by examining one of the variables x_i such that the column number i in (P) possesses at least one non-variable pattern. Until otherwise stated, the choice of this variable is arbitrary and the result of compilation depends *a priori* on it. When the variable x_i is chosen, we shall say that compilation progresses by following index i. To be more specific, assume that the first variable x_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 rows of (P) that can match a value vector 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 (P_k) and each component of the new vector (E_k) are built:

p_1^i	row in (P_k)		(E_k)
y	$\_ \cdots \_$	$p_2^i \cdots p_n^i$	$e_i[y \backslash x_1]$
$c_k\ q_1^i \ldots q_{a_k}^i$	$q_1^i \cdots q_{a_k}^i$	$p_2^i \cdots p_n^i$	e_i
$c_{k'}\ q_1^i\ q_2^i \ldots q_{a_{k'}}^i$	No row number i		

If the set of constructors Σ is not a complete signature, that is, if there are other constructors in the signature of the the type of the constructors in Σ, then some rows in matrix (P) can match value vectors 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 treat these cases, a default matrix (P_d) is built:

p_1^i	row in (P_d)	(E_d)
y	$p_2^i \cdots p_n^i$	$e_i[y \backslash x_1]$
$c\ q_1^i\ q_2^i \ldots q_a^i$	No row number i	

Of course, 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 numbers. Compilation then proceeds by emitting an elementary test on variable x_1. At run-time, this test will direct analysis towards the appropriate subproblem.

$$\mathcal{C}((x_1\ x_2 \ldots x_n), (P), (E))$$

$$||$$

$$
\begin{array}{ll}
\texttt{match } x_1 \texttt{ with} & \\
\quad c_1\ y_1 \ldots y_{a_1} & \rightarrow \mathcal{C}((y_1 \ldots y_{a_1}\ x_2 \ldots x_n), (P_1), (E_1)) \\
\texttt{|}\ c_2\ y_1 \ldots y_{a_2} & \rightarrow \cdots \\
 & \vdots \\
\texttt{|}\ c_z\ y_1 \ldots y_{a_z} & \rightarrow \mathcal{C}((y_1 \ldots y_{a_z}\ x_2 \ldots x_n), (P_z), (E_z)) \\
\texttt{|}\ \_ & \rightarrow \mathcal{C}((x_2 \ldots x_n), (P_d), (E_d))
\end{array}
$$

As a first exemple of the application of this inductive step 4, consider the following pattern matrix and result vector:

$$(P) = \begin{pmatrix} \texttt{true} & \texttt{true} \\ \texttt{false} & \texttt{\_} \\ \texttt{\_} & \texttt{\_} \end{pmatrix} \qquad (E) = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

First, assume that compilation progresses by following index number one. The set Σ of the root constructors of the patterns in the first column is $\Sigma = \{\texttt{true}, \texttt{false}\}$. The decomposition procedure described above will therefore yield two matrices (P_1) and (P_2), the matrix (P_1) being made of the rows of (P) that can be matched by a vector whose first component is `true` (that is, the first and third rows of (P)), whereas (P_2) is made of the rows of (P) that can be matched by a vector whose first component is `false` (that is, the second and third rows of (P)). More precisely, we get:

$$(P_1) = \begin{pmatrix} \texttt{true} \\ \texttt{\_} \end{pmatrix}, \quad (E_1) = \begin{pmatrix} 1 \\ 3 \end{pmatrix} \qquad (P_2) = \begin{pmatrix} \texttt{\_} \\ \texttt{\_} \end{pmatrix}, \quad (E_2) = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

Since the signature of the booleans does not comprise any other constructor than `true` and `false`, there is no default matrix here. Thus, the elementary matching emitted here is like the following:

$$\mathcal{C}((\texttt{x y}), (P), (E)) = \begin{cases} \texttt{match x with} \\ \quad \texttt{true} \quad \rightarrow \quad \mathcal{C}((\texttt{y}), \begin{pmatrix} \texttt{true} \\ \texttt{\_} \end{pmatrix}, \begin{pmatrix} 1 \\ 3 \end{pmatrix}) \\ \texttt{| false} \quad \rightarrow \quad \mathcal{C}((\texttt{y}), \begin{pmatrix} \texttt{\_} \\ \texttt{\_} \end{pmatrix}, \begin{pmatrix} 2 \\ 3 \end{pmatrix}) \end{cases}$$

Second, assume that compilation progresses by following index number two. Here we get $\Sigma' = \{\texttt{true}\}$. Thus, only one matrix (P_1') is built to take into account the value vectors $\vec{V} = (V_1\ \texttt{true})$. The set Σ' does not make up a full signature. Thus, a default matrix (P_d') is built, to take into account the value vectors $\vec{V} = (V_1\ (c\ U_1 \ldots U_a))$, where c is not `true`.

$$(P_1') = \begin{pmatrix} \texttt{true} \\ \texttt{false} \\ \texttt{\_} \end{pmatrix}, \quad (E_1') = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \qquad (P_d') = \begin{pmatrix} \texttt{false} \\ \texttt{\_} \end{pmatrix}, \quad (E_d') = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

The emitted elementary matching is as follows:

$$\mathcal{C}((\texttt{x y}), (P), (E)) = \begin{cases} \texttt{match y with} \\ \quad \texttt{true} \quad \rightarrow \quad \mathcal{C}((\texttt{x}), \begin{pmatrix} \texttt{true} \\ \texttt{false} \\ \texttt{\_} \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}) \\ \texttt{| \_} \quad \rightarrow \quad \mathcal{C}((\texttt{x}), \begin{pmatrix} \texttt{false} \\ \texttt{\_} \end{pmatrix}, \begin{pmatrix} 2 \\ 3 \end{pmatrix}) \end{cases}$$

Compilation always terminates, since the size of (P) strictly decreases at each recursive call to $\mathcal{C}$. 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:

$$\begin{cases} n_c(x) = 0 \\ n_c(c\ p_1 \ldots p_a) = 1 + n_c(p_1) + \cdots + n_c(p_a) \end{cases} \qquad \begin{cases} n_v(x) = 1 \\ n_v(c\ p_1 \ldots p_a) = 0 \end{cases}$$

As an example of a full compilation, consider the `or` function from the end of section 2.4. The initial call to function $\mathcal{C}$ is as follows:

$$\mathcal{C}((\mathtt{x\ y}), \begin{pmatrix} \_ & \mathtt{true} \\ \mathtt{true} & \_ \\ \_ & \_ \end{pmatrix}, \begin{pmatrix} \mathtt{true} \\ \mathtt{true} \\ \mathtt{false} \end{pmatrix})$$

There are two possible compilations for the `or` function, depending on which of the variables `x` or `y` is examined first:

$$\begin{array}{l} \mathtt{match\ x\ with} \\ \quad \mathtt{true} \rightarrow \mathcal{C}((\mathtt{y}), \begin{pmatrix} \mathtt{true} \\ \_ \\ \_ \end{pmatrix}, \begin{pmatrix} \mathtt{true} \\ \mathtt{true} \\ \mathtt{false} \end{pmatrix}) \\ \mathtt{|}\ \_ \quad\ \ \rightarrow \mathcal{C}((\mathtt{y}), \begin{pmatrix} \mathtt{true} \\ \_ \end{pmatrix}, \begin{pmatrix} \mathtt{true} \\ \mathtt{false} \end{pmatrix}) \\ \qquad\qquad \vdots \\ \qquad\qquad \vdots \end{array}$$

$$\mathtt{or_1(x\ y)} = \begin{cases} \mathtt{match\ x\ with} \\ \quad \mathtt{true} \rightarrow \mathtt{(match\ y\ with\ true} \rightarrow \mathtt{true\ |\ \_} \rightarrow \mathtt{true)} \\ \mathtt{|}\ \_ \quad \rightarrow \mathtt{(match\ y\ with\ true} \rightarrow \mathtt{true\ |\ \_} \rightarrow \mathtt{false)} \end{cases}$$

$$\begin{array}{l} \mathtt{match\ y\ with} \\ \quad \mathtt{true} \rightarrow \mathcal{C}((\mathtt{x}), \begin{pmatrix} \_ \\ \mathtt{true} \\ \_ \end{pmatrix}, \begin{pmatrix} \mathtt{true} \\ \mathtt{true} \\ \mathtt{false} \end{pmatrix}) \\ \mathtt{|}\ \_ \quad\ \ \rightarrow \mathcal{C}((\mathtt{x}), \begin{pmatrix} \mathtt{true} \\ \_ \end{pmatrix}, \begin{pmatrix} \mathtt{true} \\ \mathtt{false} \end{pmatrix}) \\ \qquad\qquad \vdots \\ \qquad\qquad \vdots \end{array}$$

$$\mathtt{or_2(x\ y)} = \begin{cases} \mathtt{match\ y\ with} \\ \quad \mathtt{true} \rightarrow \mathtt{true} \\ \mathtt{|}\ \_ \quad \rightarrow \mathtt{(match\ x\ with\ true} \rightarrow \mathtt{true\ |\ \_} \rightarrow \mathtt{false)} \end{cases}$$

The $\mathtt{or_1}$ and $\mathtt{or_2}$ automata are syntactically different, they also differ semantically. Indeed, we get here: $\mathtt{or_1}(\Omega\ \mathtt{true}) = \Omega$ —since simple matching of variable x immediately fails—, whereas

$\texttt{or}_2(\Omega\ \texttt{true}) = \texttt{true}$. For all other values of the boolean vector $\vec{V}$, we get $\texttt{or}_1(\vec{V}) = \texttt{or}_2(\vec{V}) = \texttt{or}(\vec{V})$. Thus, $\texttt{or}_2$ is the only automaton that implements correctly the function `or`.

There is some important point about the undefined term Ω: it can be produced by compilation only when (P) is empty (case 1). Such a situation can only be introduced by the non-trivial inductive step 4, which can produce an empty default matrix (P_d), when the considered column of matrix (P) is made of non-variable patterns whose root constructors are not a complete signature. The resulting automaton then has the ability to recognize some values that are incompatible with all patterns.

3.3 Correctness

A compilation is correct when the execution of the output automaton implements exactly the input pattern matching function.

Definition 3.1 *Given a pattern matrix (P) and a result vector (E), compilation $\mathcal{C}(\vec{x}, (P), (E))$ is correct, if and only if the following equality holds for all value vectors $\vec{V}$:*

$$\mathcal{C}(\vec{V}, (P), (E)) = \mathit{match}[(P), (E)](\vec{V})$$

As shown by the $\texttt{or}_1$ automaton from the previous section, the $\mathcal{C}$ compilation scheme may not be correct. In this case, the problem can be traced back to the untimely examination of the first component of $(\Omega\ \texttt{true})$. On the other hand, $(\Omega\ \texttt{true})$ clearly matches the first clause of the function `or`. The idea is to select carefully column indices, in order to avoid such a situation.

Definition 3.2 (Directions) *Let (P) be a pattern matrix of width n. The column index d such that $1 \leq d \leq n$ is a* direction *for the matching by (P), written $d \in \mathit{Dir}(P)$, if and only if the following two conditions are met:*

1. *The vector $\vec{\Omega}$ does not match any row in (P).*
2. *There is no vector $\vec{V} = (V_1\ V_2 \ldots V_n)$, such that $\vec{V}$ matches a pattern row in (P) and $V_d = \Omega$.*

The first condition above is just here to rule out the trivial case where any value vector matches some row of matrix (P).

Directions are computable from the matrix (P). Consider the set $\mathit{Dir}_i(P)$ of *directions towards row number i*, defined as:

$$\mathit{Dir}_i(P) = \{\, d \in [1 \ldots n] \mid \mathit{match}_i[(P)](\vec{V}) \Rightarrow V_d \succ \Omega \,\}$$

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

$$(P) = \begin{pmatrix} \_ & \texttt{true} \\ \texttt{true} & \_ \\ \_ & \_ \end{pmatrix} \qquad \text{and} \qquad \begin{cases} \mathit{match}_1[(P)](\vec{V}) \Leftrightarrow \texttt{true} \preceq V_2 \\ \mathit{match}_2[(P)](\vec{V}) \Leftrightarrow \texttt{true} \preceq V_1 \wedge \texttt{true} \# V_2 \\ \mathit{match}_3[(P)](\vec{V}) \Leftrightarrow \texttt{true} \# V_1 \wedge \texttt{true} \# V_2 \end{cases}$$

Thus, we get $\mathit{Dir}_1(P) = \{2\}$, $\mathit{Dir}_2(P) = \{1, 2\}$, $\mathit{Dir}_3(P) = \{1, 2\}$ and hence $\mathit{Dir}(P) = \{2\}$. In section 5.1, I give a full description of a general and efficient method for computing directions.

It is now time to examine again the arbitrary choice of the variable to be tested at the critical compilation step 4.

Definition 3.3 *Let E be a set of clauses. A given compilation of the matching by E is done by* following directions, *when, at each inductive application of step 4, there exists a direction d in pattern matrix (P) and that compilation goes on by following index d.*

Directions gives a sufficient condition to assert the correctness of a given compilation.

Proposition 3.1 (Correct compilation) *Compilations that follow directions produce correct automata.*

Proof: Proof is by induction on the definition of $\mathcal{C}$. The proof requires some extension of the matching function (end of section 2.4) and of the execution of an automaton (beginning of this section). Both functions now operate on the full set $\mathcal{T}_\Omega$ of partial terms. These extensions are straightforward: any variable x behaves as Ω does.

Thus, given any compilation step $\mathcal{C}(\vec{x}, (P), (E))$ and any term vector $\vec{M} = (M_1\ M_2 \ldots M_n)$, we show the correctness identity:

$$match[(P), (E)](\vec{M}) = \mathcal{C}(\vec{M}, (P), (E))$$

1. If (P) is empty ($n = m = 0$), then there are no pattern rows to match and we get:

   $$match[(), ()]() = \Omega = \mathcal{C}((), (), ())$$

   If (P) has at least one empty row ($n = 0$ and $m > 0$), then the first row always matches, that is, there exists some substitution σ_0 with an empty domain such that $\sigma_0(()) = \sigma_0(\vec{p}^{\,1}) = \vec{M} = ()$. We finally get:

   $$match[\begin{pmatrix} \\ \vdots \\ \\ \end{pmatrix}, \begin{pmatrix} e_1 \\ \vdots \\ e_m \end{pmatrix}](()) = \sigma_0(e_1) = e_1$$

   Whereas, by definition of $\mathcal{C}$ and the (success) rule, we get:

   $$\mathcal{C}((), \begin{pmatrix} \\ \vdots \\ \\ \end{pmatrix}, \begin{pmatrix} e_1 \\ \vdots \\ e_m \end{pmatrix}) = e_1 \qquad \text{and} \qquad \sigma_0 \vdash e_1 \Rightarrow \sigma_0(e_1) = e_1$$

2. If the first row of (P) is made of variables only, that is, if there exists n variables y^1, y^2,$\ldots y^n$, such that $\vec{p}^{\,1} = (y_1\ y_2 \ldots y_n)$, then there exists a substitution $\sigma = [y_1\backslash M_1, \ldots, y_n\backslash M_n]$, with $\sigma(\vec{p^1}\,) = \vec{M}$. Thus we get:

   $$match[(P), (E)](\vec{M}) = e_1[y_1\backslash M_1, \ldots, y_n\backslash M_n]$$

   On the other hand, like in the second subcase above, we get:

   $$\begin{gathered} \mathcal{C}(\vec{x}, (P), (E)) = e_1[y_1\backslash x_1, \ldots, y_n\backslash x_n] \\ \text{and} \\ [x_1\backslash M_1, \ldots, x_n\backslash M_n] \vdash e_1[y_1\backslash x_1, \ldots, y_n\backslash x_n] \Rightarrow e_1[y_1\backslash M_1, \ldots, y_n\backslash M_n] \end{gathered}$$

3. If a pattern column —the first one for instance— is made of variables only, then we get:

   $$\mathcal{C}(\vec{x}, (P), (E)) \quad = \quad \mathcal{C}(\vec{x}\,', (P'), (E')) \qquad (1)$$

   Where I have written $\vec{x}\,'$, (P') and (E') for:

$$\vec{x}\,' = (x_2 \ldots x_n) \qquad (P') = \begin{pmatrix} p_2^1 \cdots p_n^1 \\ p_2^2 \cdots p_n^2 \\ \vdots \\ p_2^m \cdots p_n^m \end{pmatrix} \qquad (E') = \begin{pmatrix} e_1[y^1\backslash x_1] \\ e_2[y^2\backslash x_1] \\ \vdots \\ e_m[y^m\backslash x_1] \end{pmatrix}$$

One easily shows that the equality (1) on automata implies the following equality on automata results.

$$\mathcal{C}(\vec{M}, (P), (E)) = \mathcal{C}((M_2 \ldots M_n), (P'), (E')[x_1\backslash M_1])$$

(Proofs for these two judgements have exactly the same structure).

Moreover, by induction hypothesis applied to matrix (P'), we get:

$$\mathcal{C}(\vec{M}, (P), (E)) = \mathit{match}[(P'), (E'[x_1\backslash M_1])](\vec{M}\,') \tag{2}$$

Given any valid row number i, pattern p_1^i is a variable that does not affect matching and the following equivalences hold:

$$(y^i\ p_2^i \ldots p_n^i) \preceq (M_1\ M_2 \ldots M_n) \quad \Leftrightarrow \quad (p_2^i \ldots p_n^i) \preceq (M_2 \ldots M_n)$$

$$(y^i\ p_2^i \ldots p_n^i) \mathrel{\#} (M_1\ M_2 \ldots M_n) \quad \Leftrightarrow \quad (p_2^i \ldots p_n^i) \mathrel{\#} (M_2 \ldots M_n)$$

That is, $\vec{M}$ matches row number i in (P), if and only if $\vec{M}' = (M_2 \ldots M_n)$ matches row number i in (P'). By making the additional remark that e'_i is $e_i[y^i\backslash x_1]$, we finally get:

$$\mathit{match}[(P'), (E')[x_1\backslash M_1]](\vec{M}'\,) = \mathit{match}[(P), (E)](\vec{M})$$

Hence the correctness equality.

4. We finally consider the case where some elementary test is emitted. We adopt the notations from section 3.2 and assume that column index 1 is a direction of (P). Given some term vector $\vec{M}$ of size n, there are three subcases to consider, depending on the first component of $\vec{M}$.

   If the term M_1 does not have a root constructor (that is, $M_1 = \Omega$ or M_1 is a variable), then the simple matching of M_1 will always fail and we get $\mathcal{C}(\vec{M}, (P), (E)) = \Omega$. On the other hand, since i is a direction of (P), no value vector with an undefined first component matches some row of (P) and we get $\mathit{match}[(P), (E)](\vec{M}) = \Omega$.

   If M_1 is of the form $M_1 = c\ N_1 \ldots N_a$, where the constructor c is not one of the root constructors appearing in the first column of (P), then, by rule (default) of simple matching, we get the following equality:

$$\mathcal{C}(\vec{M}, (P), (E)) = C((M_2 \ldots M_n), (P_d), (E_d)[x_1\backslash M_1]))$$

   Therefore, by induction hypothesis, it follows that:

$$\mathcal{C}(\vec{M}, (P), (E)) = \mathit{match}[(P_d), (E_d)[x_1\backslash M_1]](M_2 \ldots M_n)$$

Moreover, for any valid row number, the following equivalences hold:

$$\vec{p}^{\,i} \preceq ((c\ N_1 \ldots N_a) \ldots M_n) \quad \Leftrightarrow \quad p_1^i = y^i \text{ and } (p_2^i \ldots p_n^i) \preceq (M_2 \ldots M_n)$$

$$\vec{p}^{\,i} \# ((c\ N_1 \ldots N_a) \ldots M_n) \quad \Leftrightarrow \quad \begin{cases} p_1^i = c_k\ q_1^i \ldots q_{a_k}^i, \text{ where } c_k \in \Sigma \\ \qquad \text{or} \\ p_1^i = y^i \text{ and } (p_2^i \ldots p_n^i) \# (M_2 \ldots M_n) \end{cases}$$

Thus, in this case, the matching predicates defined by (P) and (P_d) are equivalent and correctness follows as in the case number 3 above.

Finally, consider the case $M_1 = c_k\ N_1 \ldots N_{a_k}$, where the constructor c_k is one of the root constructors appearing in the first column of (P). Then, by rule (constr) of simple matching and induction hypothesis, we get:

$$\mathcal{C}(\vec{M}, (P), (E)) = match[(P_k), (E_k)[x_1 \backslash c_k\ N_1 \ldots N_{a_k}]](N_1 \ldots N_{a_k}\ M_2 \ldots M_n)$$

Since the root constructor of M_1 is known, the following equivalences hold:

$$\vec{p}^{\,i} \preceq \vec{M} \quad \Leftrightarrow \quad \begin{cases} p_1^i = y^i \text{ and } (p_2^i \ldots p_n^i) \preceq (M_2 \ldots M_n) \\ \qquad \text{or} \\ \begin{cases} p_1^i = c_k\ q_1^i \ldots q_{a_k}^i \\ \qquad \text{and} \\ (q_1^i \ldots q_{a_k}^i\ p_2^i \ldots p_n^i) \preceq (N_1 \ldots N_{a_k}\ M_2 \ldots M_n) \end{cases} \end{cases}$$

$$\vec{p}^{\,i} \# \vec{M} \quad \Leftrightarrow \quad \begin{cases} p_1^i = y^i \text{ and } (p_2^i \ldots p_n^i) \# (M_2 \ldots M_n) \\ \qquad \text{or} \\ p_1^i = c_{k'}\ q_1^i \ldots q_{a_{k'}}^i, \text{ where } k \neq k' \\ \qquad \text{or} \\ \begin{cases} p_1^i = c_k\ q_1^i \ldots q_{a_k}^i \\ \qquad \text{and} \\ (q_1^i \ldots q_{a_k}^i\ p_2^i \ldots p_n^i) \# (N_1 \ldots N_{a_k}\ M_2 \ldots M_n) \end{cases} \end{cases}$$

From here, we then get the equivalence between matchings by (P) and (P_k) and correctness, as in the previous cases.

□

As an application of proposition 3.1, in the case of the *or* function, the automaton $\texttt{or}_2$ can be stated as correct without testing it on all partial values, since it always performs simple matchings by following directions.

4 Second technique

Simplicity is the main advantage of tree-like automata. Unfortunately, these automata have a major drawback: the size of output automata can be exponential in the size of the input programs — see [Maranget, 1992, Sekar *et al.*, 1992]. This drawback is particularly annoying when one thinks of integrating the $\mathcal{C}$ pattern matching compilation scheme in an operational ML compiler. One has to notice though, that compilers as mature as SML/NJ [Appel and Macqueen, 1991] and Caml V3.1 [Weis, 1990] use a similar compilation technique. Nevertheless, there already exists a different kind of automata for implementing pattern matching, such automata are the targets of the LML [Augustsson, 1985] and Caml Light [Leroy *et al.*, 1993, Leroy, 1990] compilers. One salient feature of these automata is a backtracking construct. Standard pattern matching compilation schemes rely on it to guarantee that the size of output automaton is linear in the size of the input program. The main contribution of the present paper is to show how lazy pattern matching can be compiled correctly in this second framework, while preserving the highly desirable linear bound on the size of output automata.

4.1 Automata with failures

From the syntactic point of view, there is little change between tree-like and automata with failures: the right-hand side of a clause can now be a new `default` primitive.

Simple matchings:

$$F ::= \texttt{match}\ x\ \texttt{with}\ p \rightarrow A\ \{\texttt{|}\ p \rightarrow A\}^*\ [\texttt{|}\ \_ \rightarrow A]$$

Simple patterns:

$$p ::= c\ x_1\ x_2 \ldots x_a \quad \text{and } p \text{ is linear}$$

Automata:

$$\begin{array}{lcll} A & ::= & F & \text{nested matching} \\ & \texttt{|} & M & \text{partial term} \\ & \texttt{|} & \texttt{default} & \text{default primitive} \end{array}$$

Semantically, when execution comes across a `default` primitive, control is transferred to the closest enclosing default clause. This behavior is best described by automaton *states* and *one-step transition rules* from one state to another. An automaton state is a triple (ρ, σ, A), where A is an automaton, σ is a value environment and ρ is a control environment. A value environment is simply a substitution, and a control environment is a list of partial automaton states $<\sigma, A>$, where σ is a value environment and A is an automaton. The following one-step transitions describe the run-time behavior of automata with failures.

If $\sigma(x) = c_k\ V_1 \ldots V_{a_k}$,

$$(\rho, \sigma, \texttt{match}\ x\ \texttt{with}\ \ldots \mid c_k\ x_1 \ldots x_{a_k} \rightarrow A_k\ \ldots \mid \_ \rightarrow A_d) \hookrightarrow \ldots$$
$$\ldots\ (<\sigma, A_d>.\rho, \sigma \cup [x_1 \backslash V_1, \ldots, x_{a_k} \backslash V_{a_k}], A_k) \quad \text{(constr)}$$

If $\sigma(x) = c\ V_1 \ldots V_a$ where c is not recognized,

$$(\rho, \sigma, \texttt{match}\ x\ \texttt{with}\ \ldots \mid \_ \rightarrow A_d) \hookrightarrow (\rho, \sigma, A_d) \quad \text{(default)}$$

$$(<\phi, A>.\rho, \sigma, \texttt{default}) \hookrightarrow (\rho, \phi, A) \quad \text{(raise)}$$

Observe that control environments are stacks. That is, the transition (constr) pushes the current default clause on the stack, while the transition (raise) pops the closest enclosing default clause from the stack. Here, as in the case of tree-like automata, some condition on variable names is implicitly assumed, so that we can safely perform the union of substitutions in the right-hand side of rule (constr). Obviously, requiring the pattern variables to be all different is enough.

An automaton *final state* is any state from which there are no transitions. The result V of automaton A in the initial environments σ and ρ is computed as follows. First, repetitively apply one-step transition rules, starting from the initial state (ρ, σ, A), until some final state (ρ', σ', A') is reached, thereby computing a *complete* transition. Second, consider the final automaton A'. If A' is a simple matching, then the execution of A failed and we take $V = \Omega$. Otherwise, A is the right-hand side of a clause and we define $V = \sigma'(A')$. As in the case of tree-like automata, the value V can in fact be a term, when σ operates on terms.

4.2 Compilation

Thanks to the `default` primitive, automata with failures can directly implement pattern matching with priorities: try to match against the first pattern, in case of failure, try to match against the second pattern,... In practice, one seeks to "factorize" matching attempts, by trying to match many patterns at the same time.

The new compilation function $\mathcal{D}$ takes four arguments. A typical call is written $\mathcal{D}(\vec{x}, (P), (E), d)$. The first three arguments are as in section 3.2: $\vec{x}$ is a variable vector of size n, (P) is a pattern matrix of width n and height m and (E) is a term vector of size m. The fourth argument d is itself an automaton and may be seen as the "failure continuation"of the compiled matching. Control is to be transferred to the automaton d in case of matching failure. The initial call to $\mathcal{D}$ is thus written as follows:

$$\mathcal{D}((x), \begin{pmatrix} p^1 \\ p^2 \\ \vdots \\ p^m \end{pmatrix}, \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{pmatrix}, \Omega)$$

The function $\mathcal{D}$ is then inductively defined as follows:

1. base case: $n = 0$. If (P) has no rows $(m = 0)$, then the output automata reduces to the default automaton d.

   $$\mathcal{D}((), (), (), d) = d$$

   Otherwise, there is at least one empty row in (P) and the compilation result is the first component of (E).

   $$\mathcal{D}((), \begin{pmatrix} \\ \\ \vdots \\ \end{pmatrix}, \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{pmatrix}, d) = e_1$$

2. If the matrix (P) has at least one column $(n > 0)$, then select one column of (P) arbitrarily. Let us assume for the moment that the first column of (P) is chosen. There are then two subcases:

   (a) If the pattern p_1^1 is a variable y^1, then let k be highest row number such that all patterns $p_1^1, p_1^2, \ldots, p_1^k$ are variables $y^1, y^2, \ldots, y^k$. The matrix (P) is cut horizontally into two new submatrices (Q) and (R) of respective heights k and $k - m$:

$$(Q) = \begin{pmatrix} y^1 & p_2^1 \cdots p_n^1 \\ y^2 & p_2^2 \cdots p_n^2 \\ & \vdots \\ y^k & p_2^k \cdots p_n^k \end{pmatrix} \qquad (R) = \begin{pmatrix} p_1^{k+1}(\neq y) & p_2^{k+1} & \cdots & p_n^{k+1} \\ p_1^{k+2} & p_2^{k+2} & \cdots & p_n^{k+2} \\ & \vdots & & \\ p_1^m & p_2^m & \cdots & p_n^m \end{pmatrix}$$

Two nested inductive calls to $\mathcal{D}$ are then performed, where the main argument to the outer call is (Q) without its first column and the main argument to the inner call is (R).

$$\mathcal{D}(\vec{x}, (P), (E), d)$$

$$||$$

$$\mathcal{D}((x_2\ x_3 \ldots x_n), \begin{pmatrix} p_2^1 \ldots p_n^1 \\ p_2^2 \ldots p_n^2 \\ \vdots \\ p_2^k \ldots p_n^k \end{pmatrix}, \begin{pmatrix} e_1[y^1 \backslash x_1] \\ e_2[y^2 \backslash x_1] \\ \vdots \\ e_k[y^k \backslash x_1] \end{pmatrix}, \mathcal{D}(\vec{x}, (R), \begin{pmatrix} e_{k+1} \\ e_{k+2} \\ \vdots \\ e_m \end{pmatrix}, d))$$

(b) If the pattern p_1^1 has a root constructor, then let $k+1$ be the lowest row number such that the pattern p_1^{k+1} is a variable. As above, (P) is cut into two new submatrices of respective heights k and $m-k$:

$$(Q) = \begin{pmatrix} p_1^1 & p_2^1 \cdots p_n^1 \\ p_1^2 & p_2^2 \cdots p_n^2 \\ & \vdots \\ p_1^k & p_2^k \cdots p_n^k \end{pmatrix} \qquad (R) = \begin{pmatrix} p_1^{k+1}(= y) & p_2^{k+1} & \cdots & p_n^{k+1} \\ p_1^{k+2} & p_2^{k+2} & \cdots & p_n^{k+2} \\ & \vdots & & \\ p_1^m & p_2^m & \cdots & p_n^m \end{pmatrix}$$

Let c_1, c_2, …, c_z be the root constructors that appear in the first column of (Q). The rows of (Q) are grouped according to the root constructor of their first pattern, yielding the new matrices (Q_1), (Q_2), …, (Q_z). The decomposition process is exactly the same as in the case of tree-like automata —section 3.2, case 4 of the definition of $\mathcal{C}$—, it also produces the result vectors (E_1), (E_2), …, (E_z). An important point is that no row of (Q) will ever be duplicated here, since there are no variables in the first column of (Q). A simple matching on variable x_1 is finally emitted:

$$\mathcal{D}((x_1\ x_2 \ldots x_n), (P), (E), d)$$

$$||$$

$$\begin{array}{l} \texttt{match}\ x_1\ \texttt{with} \\ \quad c_1\ y_1 \ldots y_{a_1} \to \mathcal{D}((y_1 \ldots y_{a_1}\ x_2 \ldots x_n), (Q_1), (E_1), \texttt{default}) \\ \texttt{|}\ c_2\ y_1 \ldots y_{a_2} \to \cdots \\ \qquad \vdots \\ \texttt{|}\ c_z\ y_1 \ldots y_{a_z} \to \mathcal{D}((y_1 \ldots y_{a_z}\ x_2 \ldots x_n), (Q_z), (E_z), \texttt{default}) \\ \texttt{|}\ \_ \qquad\qquad \to \mathcal{D}((x_1\ x_2 \ldots x_n), (R), (F), d) \end{array}$$

At run-time, this simple matching will examine the root constructor of the value of x_1 and will select the appropriate subproblem (Q_1), (Q_2),…(Q_z) or the default subproblem (R), depending on whether this root constructor is one of the recognized constructors c_1, c_2,

$\ldots c_z$ or not. The failure continuation of the inductive calls (Q_1), (Q_2),$\ldots(Q_z)$ is the `default` construct, so that control will be transferred to the default subproblem (R), in case any selected subproblem (Q_i) fails.

Compilation always terminates, since the sum of the sizes of the patterns in (P) strictly decreases at every inductive call. Moreover, since no patterns ever get duplicated, the total number of non-default clauses in the simple matchings produced is bounded by the sum of the sizes of the initial patterns. Taking the default clauses into account, the size of the output automaton is bounded by twice the size of the input program.

As an example, consider the following `or'` function definition:

```
let or'  false false  = false
|   or'  true  _      = true
|   or'  _     true   = true
```

The initial call to function $\mathcal{D}$ is as follows:

$$\mathcal{D}((x\ y), \begin{pmatrix} \texttt{false} & \texttt{false} \\ \texttt{true} & \texttt{\_} \\ \texttt{\_} & \texttt{true} \end{pmatrix}, \begin{pmatrix} \texttt{false} \\ \texttt{true} \\ \texttt{true} \end{pmatrix}, \Omega)$$

There are now two possibilities, depending on whether compilation proceeds left-to-right or right-to-left. In the first case, `x` is examined first:

$$\begin{array}{l} \texttt{match x with} \\ \quad \texttt{false} \rightarrow \mathcal{D}((y), \begin{pmatrix} \texttt{false} \end{pmatrix}, \begin{pmatrix} \texttt{false} \end{pmatrix}, \texttt{default}) \\ \texttt{| true} \;\; \rightarrow \mathcal{D}((y), \begin{pmatrix} \texttt{\_} \end{pmatrix}, \begin{pmatrix} \texttt{true} \end{pmatrix}, \texttt{default}) \\ \texttt{| \_} \qquad \rightarrow \mathcal{D}((x\ y), \begin{pmatrix} \texttt{\_} & \texttt{true} \end{pmatrix}, \begin{pmatrix} \texttt{true} \end{pmatrix}, \Omega) \end{array}$$

This first left-to-right compilation finally yields the automaton `or'`$_1$:

```
match x with
  false → (match y with false → false | _ → default)
  true  → true
| _     → (match y with true → true | _ → Ω)
```

The second compilation, which examines `y` before `x`, produces the following automaton `or'`$_2$:

$$\begin{array}{l} \texttt{match y with} \\ \quad \texttt{false} \rightarrow \mathcal{D}((x), \begin{pmatrix} \texttt{false} \end{pmatrix}, \begin{pmatrix} \texttt{false} \end{pmatrix}, \texttt{default}) \\ \texttt{| \_} \qquad \rightarrow \mathcal{D}((x\ y), \begin{pmatrix} \texttt{true} & \texttt{\_} \\ \texttt{\_} & \texttt{true} \end{pmatrix}, \begin{pmatrix} \texttt{true} \\ \texttt{true} \end{pmatrix}, \Omega) \\ \qquad\qquad\qquad \vdots \end{array}$$

```
match y with
  false → (match x with false → false | _  → default)
| _     → (match x with
              true → true
           | _     → (match y with true  → true | _  → Ω))
```

By explicit computation of all cases, one easily shows that the function `or'`, the automaton `or'`$_1$ and the automaton `or'`$_2$ coincide on all possible value vectors $\vec{V} = $ (`x y`), except for the case $\vec{V} = $ (`true` Ω), where we get `or'`$(\vec{V}) = $ `or'`$_1(\vec{V}) = $ `true` and `or'`$_2(\vec{V}) = \Omega$. That is, automaton `or'`$_2$ implements correctly function `or'`, whereas automaton `or'`$_2$ does not. Observe that $\vec{V} = $ (`true` Ω) matches the second clause of function `or'` and that the non-correctness of automaton `or'`$_1$ can be tracked back to the untimely examination of the second component of $\vec{V}$. In other words, the automaton `or'`$_1$ is not correct because it is not generated by following a direction at the critical initial step. In the following section this remark is generalized. More precisely, I use directions of intermediate pattern matrices to state a sufficient condition for generating correct automata.

4.3 Correctness

Correctness will rely on following directions of some pattern matrices, just like in the case of tree-like automata. But a slight complication arises here: the argument matrix (P) does not fully account for the currently compiled matching. Therefore, the full $\mathcal{D}$ compilation scheme takes three extra arguments and a typical call to $\mathcal{D}$ is now written $\mathcal{D}(\vec{x}, (P), (E), \vec{x}\,', (P'), (P''), d)$. The new arguments are the variable vector $\vec{x}\,'$ of size n', the pattern matrix (P') of size $n' \times m'$ and the pattern matrix (P'') of size $n' \times m''$. At any stage in the compilation scheme, $(P)'$ accounts for the patterns that are already known to be incompatible with the currently examined values, whereas (P'') represents the totality of the patterns that can still be reached at that point. In some sense, made precise in the safety conditions below, (P'') includes (P).

Definition 4.1 *Consider any call $\mathcal{D}(\vec{x}, (P), (E), \vec{x}\,', (P'), (P''), d)$. Two safety conditions are defined as follows:*

1. *The vector $\vec{x}$ is a subvector of the vector $\vec{x}\,'$. That is, there exists an injective mapping $\mathcal{I}$ from the interval $[1 \ldots n]$ into the interval $[1 \ldots n']$ such that, for all integer i in the interval $[1 \ldots n]$, we get $x_i = x'_{\mathcal{I}(i)}$.*

2. *The matrix (P) is a submatrix of the matrix (P''). That is, we have $m \leq m''$ and, for any column index i in the interval $[1 \ldots n]$ and any row number j in the interval $[1 \ldots m]$, we have the equality $p_i^j = p''^{j}_{\mathcal{I}(i)}$. Furthermore, for any row number j in the interval $[1 \ldots m]$ and any column index i'' that is not the image of some integer by the mapping $\mathcal{I}$, the pattern $p''^{j}_{i''}j$ is a variable.*

As we shall see later in correctness proof, the main reason for the safety condition is to guarantee that any vector matching the row number i in (P'') (where $i < m$) also matches the row number i in (P).

The initial call to $\mathcal{D}$ is now written as follows:

$$\mathcal{D}((x), \begin{pmatrix} p^1 \\ p^2 \\ \vdots \\ p^m \end{pmatrix}, \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{pmatrix}, (x), (), \begin{pmatrix} p^1 \\ p^2 \\ \vdots \\ p^m \end{pmatrix}, \Omega)$$

Now return to the inductive cases of the definition of $\mathcal{D}$, in order to make explicit the computation of the new arguments. Still make the arbitrary choice of examining the first column of (P).

2-(a) If p_1^1 is a variable, we then get the following two nested inductive calls (see section 4.2 for the meaning of the $\vec{y}$, (Q), $\vec{z}$ and (R) notations):

$$\mathcal{D}(\vec{y}, (Q), (F), \vec{y}\,', (Q'), (Q''), \mathcal{D}(\vec{z}, (R), (G), \vec{z}\,', (R'), (R''), d))$$

The new arguments for the outermost call are easily defined, we take $\vec{y}\,' = \vec{x}\,'$, $(Q') = (P')$ and $(Q'') = (P'')$. As to the innermost call, we first take $\vec{z}\,' = \vec{x}\,'$. Then, the matrix (R'') is built by removing the k top rows of (P''). Finally, these row are added at the bottom of (P'), yielding the matrix (R') (remember that k is the highest row number such that pattern p_k^1 is a variable).

$$(R') = \begin{pmatrix} {p'}_1^1 & {p'}_2^1 & \cdots & {p'}_{n'}^1 \\ & \vdots & & \\ {p'}_1^{m'} & {p'}_2^{m'} & \cdots & {p'}_{n'}^{m'} \\ {p''}_1^1 & {p''}_2^1 & \cdots & {p''}_{n'}^1 \\ & \vdots & & \\ {p''}_1^k & {p''}_2^k & \cdots & {p''}_{n'}^k \end{pmatrix} \qquad (R'') = \begin{pmatrix} {p''}_1^{k+1} & {p''}_2^{k+1} & \cdots & {p''}_{n'}^{k+1} \\ & \vdots & & \\ {p''}_1^{m''} & {p''}_2^{m''} & \cdots & {p''}_{n'}^{m''} \end{pmatrix}$$

2-(b) If p_1^1 has a root constructor, let $i' = \mathcal{I}(1)$ be the component index in $\vec{x}\,'$ such that $x'_{i'} = x_1$. Let also $c_1, c_2, \ldots, c_z$ be the root constructors appearing in the first row of (P). Without loss of generality, I describe the inductive call $\mathcal{D}(\vec{y}, (Q_1), \vec{y}', (Q'_1), (Q''_1), \texttt{default})$, which is to be performed when constructor c_1 has been recognized (see case 2-(b) in section 4.2). Matrices (P') and (P'') are decomposed according to their column number i', in order to extract pattern rows whose pattern number i' can match values with c_1 as root constructor (see section 3.2, case 4, for a complete description of this decomposition procedure). This process yields the new matrices (Q'_1) and (Q''_1). Finally we take $\vec{y}\,' = (y_1 \ldots y_{a_1}\ {x''}_1 \ldots {x''}_{i'-1}\ x''_{i'+1} \ldots x''_{n''})$.

As to the inductive call $\mathcal{D}(\vec{x}, (R), (F), \vec{x}, (R'), (R''), d)$, which is to be performed in the default case, matrices (R') and (R'') are the same as the ones for the innermost inductive call in case 2-(a) just above. That is, the first k rows of (P') are transferred at the bottom of (P'') (remember that here, integer k is the highest row number such that p_k^1 is not a variable).

One easily checks that the safety conditions are met in the initial call. To see that they are preserved across inductive calls, first notice that, in step 2-(a), whenever some rows are removed from the top of (P), the same number of rows are removed from the top of (P'') and that whenever a column is removed from (P) and not from (P''), this column consists solely of variables. Furthermore, in step 2-(b), (P) and (P'') are decomposed according to the columns number 1 and $i' = \mathcal{I}(1)$ respectively and we know from the safety conditions that p_1^j equals ${p''}_{i'}^j$ for any row number j valid both in (P) and (P''), so that (Q_1), (Q_2), …, (Q_z) are submatrices of (Q''_1), (Q''_2), …, (Q''_z).

The correctness property is simple to formulate: a compilation is correct when it produces an automaton $A = \mathcal{D}(\vec{x}, (P), (E), \ldots, \Omega)$, whose result in environments $\sigma = [x_1 \backslash V_1,\, x_2 \backslash V_2, \ldots, x_n \backslash V_n]$ and ρ equates the value $match[(P), (E)](\vec{V}\,)$. The correctness proof will rely on a stronger property and requires a few new concepts and notations.

Notation 4.1 *Let (P') and (P'') be two pattern matrices of same width n' and of respective heights m' and m''. The new matrix $(P'@P'')$ is defined as the $n' \times (m' + m'')$ matrix build by concatenating the rows of (P') and the rows of (P''). Furthermore a* direction *towards (P'') in $(P'@P'')$ is any column index d, such that there exists no value vector $\vec{V}$ whose component V_d is the undefined term Ω and that matches some row in $(P'@P'')$ whose number is strictly greater than m'.*

At any compilation stage, write $A = \mathcal{D}(\vec{x}, (P), \vec{x}', (P'), (P''), d)$ for the produced automaton and consider some arbitrary term environment σ and control environment ρ. Then, define the following two strong correctness properties:

(α) If there exists some integer i in $[1 \ldots m]$, such that the term vector $\sigma(\vec{x}\,')$ matches the row number $m' + i$ in $(P'@P'')$ —and then we know that there exists some substitution ϕ, such that $\phi(\vec{p''}^{\,i}) = \sigma(\vec{x}\,')$ and $match[(P), (E)](\sigma(\vec{x})) = \phi(e_i)$—, then the automaton state (ρ, σ, A) leads to the final state (ρ', σ', e'_i), where we get the equality $\sigma'(e'_i) = \phi(e_i)$.

(β) If there exists some integer i in $[m + 1 \ldots m'']$, such that the term vector $\sigma(\vec{x}\,')$ matches the row number $m' + i$ in $(P'@P'')$, then the automaton state (ρ, σ, A) leads to the intermediate state (ρ, σ', d).

It should be noticed that the correctness condition (α) uses the safety conditions, when it implicitely asserts that any vector matching the row number $m' + i$ in $(P'@P'')$ with $i \leq m$ also matches the row number i in (P).

Proposition 4.1 *Any call $\mathcal{D}(\vec{x}, (P), \vec{x}', (P'), (P''), d)$ obeys the (α) and (β) correctness properties, provided that compilation follows some direction towards (P'') in $(P'@P'')$ at every critical compilation step 2-(b).*

Proof: Proof is by induction on the definition of $\mathcal{D}$.

1. If (P) is empty ($n = 0$, $m = 0$), then the output automaton A reduces to the failure continuation d and the intermediate state (ρ, σ, d) is already reached, regardless of σ. Therefore, one just has to check that case (α) is not possible here, which is obvious since an empty matrix does not even have a single row to match.

   If (P) has at least one empty row ($n = 0$, $m > 0$), then we get $A = e_1$ and the final state (ρ, σ, e_1) is already reached, regardless of σ. Moreover, the term vector $\sigma(\vec{x}\,')$ cannot match any row in (P'') but the first, because the first row in (P'') is made of variables only (by the safety condition). Therefore, cases (α) with $i > 1$ and (β) are not possible.

2. If (P) possesses at least one pattern, then assume —for instance— that compilation progresses by decomposing (P) along its first row and that we have $\mathcal{I}(1) = 1$ (that is, $x'_1 = x_1$).

   (a) If p^1_1 is a variable y^1, then we have the following nested calls, (see case 2-(a) in the previous descriptions of $\mathcal{D}$):

   $$\mathcal{D}(\vec{y}, (Q), (F), \ldots, \mathcal{D}(\vec{x}, (R), (G), \ldots, d))$$

Let then σ be an environment such that $\sigma(\vec{x'})$ matches row number $m' + i$ $(1 \leq i \leq m'')$ in $(P'@P'')$. Let ϕ be a substitution such that $\phi(\vec{p}''^i) = \sigma(\vec{x}\,')$. There are three possible cases:

If $i \leq k$ —remember that here, k is the smallest row number such that p^1_{k+1} is not a variable—, then the transition $(\rho, \sigma, A) \hookrightarrow^* (\rho', \sigma', f'_i)$ holds, where we have $\sigma'(f'_i) = \phi(f_i)$, by direct application of the hypothesis (α) to the outermost call $\mathcal{D}(\vec{y}, (Q), (F), \ldots)$. Moreover, we have $f_i = e_i[y^i \backslash x_1]$ (par definition of $\mathcal{D}$), $\sigma'(x_1) = \sigma(x_1)$ (environments can only grow during transitions), and $\sigma(x_1) = \phi(y^i)$ (by hypothesis). Therefore, we finally get $\sigma'(f'_i) = \phi(e_i)$.

If i belongs to $[k+1 \ldots m]$, then, by application of the induction hypothesis (β) to the outermost call $\mathcal{D}(\vec{y}, (Q), (F), \ldots)$, followed by an application of the induction hypothesis (α) to the innermost call $\mathcal{D}(\vec{x}, (R), (G), \ldots)$, we get the transitions:

$$(\rho, \sigma, A) \hookrightarrow^* (\rho, \sigma', \mathcal{D}(\vec{x}, (R), (G), \ldots, d)) \hookrightarrow^* (\rho'', \sigma'', g'_i)$$

By induction hypothesis, we also have $\phi(g_i) = \sigma''(g'_i)$. Thus, we directly get: $\phi(e_i) = \sigma''(g'_i)$, since $g_i = e_i$ by construction of the term vector (G).

Finally, if i is strictly greater than m, then, by a double application of the hypothesis (β), we get:

$$(\rho, \sigma, A) \hookrightarrow^* (\rho, \sigma', \mathcal{D}(\vec{x}, (R), (G), \ldots, d)) \hookrightarrow^* (\rho, \sigma'', d)$$

(b) If p^1_1 has a root constructor, then, keeping notations from the definition of $\mathcal{D}$, consider an environment σ, such that $\sigma(\vec{x}\,')$ matches the row number $m' + i$ in $(P'@P'')$. It is important to notice that, by hypothesis, the index 1 is a direction in $(P'@P'')$, so that $\sigma(x_1)$ has a root constructor c.

If i belong to $[1 \ldots k]$, then the constructor c must be recognized by the elementary matching emitted at this compilation stage — let c be c_1 for instance. By definition of one-step transitions and application of the hypothesis (α) to the call $A_1 = \mathcal{D}(\ldots, (Q_1), (E_1), \ldots, \texttt{default})$, we get the transitions:

$$(\rho, \sigma, A) \hookrightarrow (\rho'', \sigma'', A_1) \hookrightarrow^* (\rho', \sigma', e'_i)$$

Where $\phi(e_i) = \sigma'(e'_i)$.

If i belongs to $[k+1 \ldots m]$, then there are two subcases. If c is recognized by the elementary test emitted here —assume $c = c_1$ for instance—, then the correctness result follows by a one-step transition (constr), an application of the hypothesis (β) to the call $\mathcal{D}(\ldots, (Q_1), (E_1), \ldots, \texttt{default})$, a one-step transition (raise) and an application of the hypothesis (α) to $\mathcal{D}(\vec{x}, (R), (F), \ldots, d)$. Otherwise, we get correctness more directly, by a one-step transition (default) and an application of the hypothesis (α) to the call $\mathcal{D}(\vec{x}, (R), (F), \ldots, d)$.

If i is strictly greater than m, then the reasoning is exactly as above, but for the last step which is replaced by an application of the hypothesis (β) to the default inductive call $\mathcal{D}(\vec{x}, (R), (F), \ldots, d)$.

$\square$

5 Building a correct and efficient pattern matching compiler

The ultimate goal of this paper is to introduce a pattern matching compiler that outputs a correct automaton whenever there exists one. Obviously, a first step is to compute directions, the next section shows how to achieve this intermediate goal.

5.1 Efficient computation of directions

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

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

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

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

$$match_i[(P)](V_1 \ldots V_{d-1}\ \Omega\ V_{d+1} \ldots V_n) = match_i[(Q_{(d,i)})](V_1 \ldots V_{d-1}\ V_{d+1} \ldots V_n)$$

□

The following algorithm solves the unused match case detection problem in the general case. Given a pattern matrix (P), of size $n \times m$, it computes the truth value of the formula $\mathcal{F}(P) = \exists \vec{V}\ match_m[(P)](\vec{V})$. This algorithm closely follows the compilation scheme $\mathcal{C}$ itself (see section 3.2):

1. If the rows of (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 $\mathcal{F}(P)$ is true, since any instance of $\vec{p}^{\,1}$ matches the last (and only) row of matrix (P). Otherwise, $\mathcal{F}(P)$ is false.

2. In all the other cases, let us choose a column index. Assume that the index 1 is chosen. Let $\Sigma = \{c_1, c_2, \ldots, c_z\}$ 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, exactly as it is done in the $\mathcal{C}$ scheme. If 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 that matches the last row 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 $\vec{U} = (U_1\ V_2 \ldots V_n)$ such that $U_1 \succ \Omega$ matches row m as $\vec{V}$ does. Therefore, $\mathcal{F}(P)$ is true, if and only at one of the formulas $\mathcal{F}(P_1)$, $\mathcal{F}(P_2)$, ..., $\mathcal{F}(P_z)$ or $\mathcal{F}(P_d)$ is.

(b) If $p_1^m \succ x$, then let c_k be the root constructor of p_1^m. We easily show the equality $\mathcal{F}(P) = \mathcal{F}(P_k)$.

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. As shown by the example given in [Maranget, 1992, Sekar *et al.*, 1992], the number of calls to function $\mathcal{C}$ can be quite large. Although I do not know whether this upper bound is indeed reached or not in the worst cases, experiments showed that a naive implementation of function $\mathcal{F}$ may lead to important computations. Fortunately, this misbehavior can be avoided in practice by using the following three heuristics:

1. The matrix (P) itself can be reduced. Let $\vec{p}^{\,i}$ and $\vec{p}^{\,j}$ be two rows in (P), such that $i < m$, $j < m$ and $\vec{p}^{\,i} \preceq \vec{p}^{\,j}$. For any value vector $\vec{V}$ such that $\vec{p}^{\,i} \# \vec{V}$, we necessarily have $\vec{p}^{\,j} \# \vec{V}$, by definition of the compatibility relation. 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. I 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. I select a column with a minimal $r(i)$. If there are several columns such that $r(i)$ is minimal, then I favor one with a minimal number of different root constructors $z(i)$. Another, simpler, choice is first to minimize $v(i)$ (in a attempt to limit row duplication) and then to minimize $z(i)$ (in an attempt to limit the number of matrices generated). Other size measures have been tested, including the surface of matrices (number of rows $\times$ number of columns) and the function N_c of section 3.2. Choosing a good measure is not easy and this heuristic is less efficient than the two others.

Regarding the efficiency of the computation of the set of directions $Dir(P)$, 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 schemes. Otherwise, there is no obvious direction in (P) and $Dir(P)$ has to be tested for emptyness. If some index d does not belong to $Dir_i(P)$, then, for any other row 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 row 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 row 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*. Partial directions will be used later in section 6, in the process of "not so incorrectly" compiling pattern matching definition that cannot be compiled correctly.

Previous approaches to lazy pattern matching compilation [Laville, 1991, Puel and Suárez, 1990] involve the explicit computation of the set of value vectors matching the rows of matrix (P). Let $\mathcal{M}$ be this set. We have $\mathcal{M} = \bigcup_{i=1}^{m} \mathcal{M}_i$, where $\mathcal{M}_i = \{\vec{V} \mid match_i[(P)](\vec{V})\}$. In [Laville, 1991] the set $\mathcal{M}$ is described by its *minimal generators*, that is, by the subset of its least defined elements. In [Puel and Suárez, 1990] each set $\mathcal{M}_i$ is represented by a normalized constrained pattern that can be seen as the disjunctive normal form of the following 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 the 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.

5.2 The compilation algorithm

Both correctness criteria 3.1 and 4.1 apply at compile-time, so that incorrect automata are ruled out before they are completely built. For instance, when compiling the `G` function given in the introduction, the $\mathcal{C}$ and $\mathcal{D}$ compilation scheme find that this function cannot be compiled correctly as soon as the first call to the compilation scheme, since this first call is performed with a (P) matrix that has no directions.

The existence of these compile-time correctness criteria is not enough, though. If a matrix (P) with many directions is discovered, the search for a correct automaton *a priori* necessitates to try all the compilations that follow each of these possible directions. Namely, if, from that stage, compilation goes on following a first direction and that a (P) matrix without a direction is discovered later, we cannot be sure that such a situation would also have occurred, if another direction had been selected in the first place. Therefore, to detect non-compilable matchings, a straightforward compilation algorithm should check all existing directions. Such a technique can be extremely expensive. The key idea for solving this problem is that backtracking as described above is useless, because discovering a matrix without a direction at any compilation stage characterizes a pattern matching function that cannot be implemented correctly.

More specifically, the matching functions that can be implemented correctly are described by invoking the *sequentiality* theory, as introduced by [Kahn and Plotkin, 1978] and presented in [Curien, 1986] or [Huet and Lévy, 1979][Part II]. I will only sketch this part, considering the case of automata with failures; complete developments and proofs for the case of the tree-like automata are in my thesis [Maranget, 1992]. From here, I assume some familiarity with basic notions on terms such as *occurrences* and *subterms*. The subterm located at occurrence u in some term M is written M/u.

Definition 5.1 ([Kahn and Plotkin, 1978]) *Let $\mathcal{P}$ be a monotonic predicate over partial values. An occurrence u is an* index *of $\mathcal{P}$ in U, if and only if we have $U/u = \Omega$ and for all V such that $V \succeq U$, $F(V)$ holds implies implies $V/u \succ \Omega$. Then, $\mathcal{P}$ is sequential at U, such that $F(U)$ does not hold, if and only if whenever there exists $V \succeq U$ such that $F(V)$ holds, it follows that there exists an index of $\mathcal{P}$ in U. Finally, $\mathcal{P}$ is sequential, if and only if it is sequential at every partial value on which $\mathcal{P}$ does not hold.*

Our first step is to introduce terms into the compilation process. Thus, the $\mathcal{D}$ compilation scheme will now take an extra argument. This new argument N is a linear term and appears in first position. At any compilation stage, it is to be understood as the already *explored prefix* of the examined term. Initially, nothing has been explored yet and the prefix N is a simple variable. The first call to $\mathcal{D}$ is as follows:

$$\mathcal{D}(x, (x), \begin{pmatrix} p^1 \\ p^2 \\ \vdots \\ p^m \end{pmatrix}, \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{pmatrix}, (x), (), \begin{pmatrix} p^1 \\ p^2 \\ \vdots \\ p^m \end{pmatrix}, \Omega)$$

Then, the argument N is inductively computed as follows (see the inductive definition of $\mathcal{D}$, that is, cases the cases 2-(a) in sections 4.2 and 4.3):

2-(a) If p_1^1 is a variable, then we get the following two nested inductive calls:

$$\mathcal{D}(N, (x_2 \ldots x_n), (Q), (F), \vec{x}\,', (P'), (P''), \mathcal{D}(N, \vec{x}, (R), (G), \vec{x}\,', (R'), (R''), d))$$

Observe that N does not change here, since the examination of the input term does not progress at this stage.

2-(b) If p_1^1 has a root constructor, then let us assume that a simple matching of the variable $x_1 = x'_1$ is emitted and let c_1, c_2, …, c_z be the recognized constructors. Let us then consider —for instance— the inductive call to be performed when constructor c_1 has been recognized:

$$\mathcal{D}(N[x'_1 \backslash c_1\ y_1 \ldots y_{a_1}], (y_1 \ldots y_{a_1}\ x_2 \ldots x_n), (Q_1), (y_1 \ldots y_{a_1}\ x'_2 \ldots x'_{n'}), (Q'_1), (Q''_1), \texttt{default})$$

As to the default inductive call, it now yields:

$$\mathcal{D}(N, \vec{x}, (R), (F), \vec{x}\,', (R'), (R''), d)$$

In the following, we consider any call $\mathcal{D}(N, \vec{x}, (P), (E), \vec{x}\,', (P'), (P''), d)$ taken from the compilation of the set of clauses $E = \{q^1\colon e_1, q^2\colon e_2, \ldots, q^k\colon e_k\}$.

Lemma 5.2 *For any substitution σ, the following equivalence holds:*

$$\text{The vector } (\sigma(x'_1)\ \sigma(x'_2) \ldots \sigma(x'_{n'})) \text{ matches some row in } (P'@P'')$$
$$\Updownarrow$$
$$\text{The partial value } \sigma(N) \text{ matches some clause in } E$$

Proof: The proof is by induction on the definition of $\mathcal{D}$. The key point is that the rows of $(P'@P'')$ are exactly the subpatterns q^i_1, q^i_2, $\ldots q^i_{n'}$ of one of the initial patterns q^i that admits N as prefix, (that is, $q^i = N[x'_1\backslash q^i_1, x'_2\backslash q^i_2, \ldots, x'_{n'}\backslash q^i_{n'}]$). □

Lemma 5.3 *At any inductive step during compilation, consider a column index d in $(P'@P'')$. Let u_d be the occurrence of the variable x'_d in the prefix N. Then, we have the following equivalence:*

$$u_d \textit{ is an index for the matching predicate defined by } E \Leftrightarrow d \textit{ is a direction in } (P'@P'')$$

Proof: This lemma is a quite direct consequence of the previous lemma and of the fact that the variables of the linear term N are exactly the components of $\vec{x}\,'$. □

Corollary 5.1 *If some matrix $(P'@P'')$ without a direction is discovered during compilation of the clause set E, then the matching predicate defined by E is not sequential.*

Our second step is to show that the the recognition of matching values by an automaton defines a sequential predicate. Now consider an extended version of the transition rules of automata with failures. An extended automaton state is a four-tuple (N, ρ, σ, A), where N is a linear term. Initially, the term N is the same single variable as the one given as first argument to the compilation function $\mathcal{D}$. Later on, during transitions, the term N records the part of the input term that has already been examined, in the sense made clear by the following extended transition rules:

$$\begin{array}{l}
\text{If } \sigma(x) = c_k\ V_1 \ldots V_{a_k}, \\
(N, \rho, \sigma, \texttt{match}\ x\ \texttt{with}\ \ldots \mid c_k\ x_1 \ldots x_{a_k} \rightarrow A_k\ \ldots \mid _ \rightarrow A_d) \hookrightarrow \quad \ldots \\
\quad \ldots \quad (N[x\backslash c_k\ x_1 \ldots x_{a_k}], <\sigma, A_d>.\rho, \sigma \cup [x_1\backslash V_1, \ldots, x_{a_k}\backslash V_{a_k}], A_k) \quad \text{(constr)}
\end{array}$$

$$\begin{array}{l}
\text{If } \sigma(x) = c\ V_1 \ldots V_a \text{ where } c \text{ is not recognized,} \\
(N, \rho, \sigma, \texttt{match}\ x\ \texttt{with}\ \ldots \mid _ \rightarrow A_d) \hookrightarrow (N[x\backslash c\ \Omega \ldots \Omega], \rho, \sigma, A_d) \quad \text{(default)}
\end{array}$$

$$(N, <\phi, A>.\rho, \sigma, \texttt{default}) \hookrightarrow (N, \rho, \phi, A) \quad \text{(raise)}$$

To compute the result of the execution of the automaton A on the value U, we just compute the complete transition starting from the initial state $(x, (), [x\backslash U], A)$, thereby reaching some final state (N', ρ', σ', A'). Then, we say that U has been recognized, if and only if B is some clause right-hand side. Otherwise, U has not been recognized and B is either a simple matching or the undefined value Ω. Observe that, at any intermediate state $(N'', \rho'', \sigma'', A'')$, we have $U \succeq N''_\Omega$ and $\sigma''(N'') = U$. Also observe that the recognized prefix is increasing at each one-step transition.

Lemma 5.4 *The execution of the automaton A on the value U defines a sequential predicate.*

Proof: Consider some value U which is not reconized by A. There exists a complete transition $(x, (), [x\backslash U], A) \hookrightarrow^* (N', \rho', \sigma', A')$, where A' is not a clause right-hand side. Observe that the execution of A on any value $V \succeq U$ will yield the following complete transition $(x, (), [x\backslash V], A) \hookrightarrow^* (N', \tau', \phi', A') \hookrightarrow^* (N'', \tau'', \phi'', A'')$, because N' is also a prefix of V. We now have two subcases:

1. If A' equals Ω, then we necessarily get $A'' = A'$ and and A cannot recognize V.

2. If A' is some simple matching, then let u be the occurrence in N' of the variable x tested by A'. Since (N', ρ', σ', A') is a final state, with have $\sigma'(x) = \Omega$, that is, $U/u = \Omega$. Since A'' is not a simple matching there is at least one transition of type (constr) or (default) starting from (N', τ', ϕ', A'). Thus, the subterm N''/u has a root constructor and so does $V/u \succeq N''_\Omega/u$. □

Note that the above proof also applies to the case of tree-like automata, provided tree-like automata are given the appropriate (and straightforward) semantics based upon states and transitions.

I can now formulate the main result of this section and present two efficient, correct and complete algorithms for compiling pattern matching.

Proposition 5.1 *Consider the two following compilation algorithms: use the $\mathcal{C}$ or $\mathcal{D}$ scheme following directions of appropriate matrices whenever an elementary test is emitted. When a matrix with no directions is discovered, then fail.*

Given a clause set E, both algorithms produce an automaton that implements correctly the matching by E, if and only if the matching predicate defined by the patterns of E is sequential in the Kahn-Plotkin sense.

Proof: The direct sense of the proof comes from the fact that discovering a directionless matrix during compilation implies the existence of a value U, such that the matching predicate defined by E is not sequential at U (corollary 5.1). Conversely, automata of both kind define sequential predicates (lemma 5.4). Therefore, in case the compilation algorithm succeeds, the matching predicate defined by E is sequential, by correctness property 3.1 or 4.1. □

6 Conclusion

In this paper, I have fully described two pattern matching compilation schemes. Moreover, I showed that both schemes can serve as a basis for a realistic and correct compilation of lazy pattern matching, this result is entirely new for the second scheme. The main advantage of lazy pattern matching is maximization of the termination properties of programs. More precisely, a standard supercombinator-based compiler extended with the correct compilation of lazy pattern matching described here implements a correct reduction strategy for ML considered as a rewriting system [Maranget, 1992]. The implemented strategy is correct in the sense that it terminates and produces a result whenever any other strategy does.

When designing a semantics or at least a precise definition for lazy functional languages such as LML or Haskell [Johnsson, 1987, Hudak *et al.*, 1992], the denotation of pattern matching expressions should, in my view, be based upon a matching predicate very similar to the Laville's definition (definition 2.3). This definition of matching is appealing because of its simplicity and generality. The compilers described in this paper are correct relatively to this definition of matching, but they are not complete, since they only accept sequential pattern matching predicates. First, observe that the sequential functions are widely accepted as the class of functions that can be computed naturally on a sequential computer. On this topic, see, for instance, [Berry and Lévy 1979] and [Plotkin, 1977]. In other words, it would be problematic to design a pattern matching compiler that would be both correct and complete relatively to Laville's semantics, unless the target automata of this new hypothetical compiler run on a parallel computer, real or simulated. Thus, the compilation algorithm 5.1 goes as far as can be expected of any compiler that produces code for ordinary, sequential computers. Also observe that the compilers presented here detect all non-sequential matchings and can thus react appropriately when given such a matching as input. An appropriate reaction is usually to issue a warning message before producing a non-correct automaton.

Published language definitions [Johnsson, 1987, Hudak *et al.*, 1992] present a different semantics for pattern matching: they select one particular sequential semantics (i.e., left-to-right). This approach lacks the generality and elegance of Laville's solution. First, there is no natural correspondence between this left-to-right matching order and terminating rewriting strategies, as there is one in the case of the lazy matching order. I already gave an intuition for this argument in the introduction: the left-to-right semantics for the function `F` does not gives a non-Ω result whenever possible. Therefore, a left-to-right semantics can be seen as incorrect, at least, it is not as expressive as possible.

Moreover, in my opinion, including a particular compilation technique for pattern matching in the language definition is not useful to the programmers and may even prove undesirable. Programmers do not need such a precision and relying on non-obvious features does not usually lead to a very clear programming style. In a slightly different context, observe that the definition of the C language does not specify the evaluation order of the arguments to primitives such as "+".

Furthermore, it is important not to overspecify the language definition. Doing so constraints the compiler designers too much and prevents them from introducing some optimizations. As a matter of fact, the correct compilation of the lazy semantics for pattern matching has some practical benefits. Let us consider a slight generalization of the $\mathcal{C}$-based compilation algorithm as introduced under the name of "adaptive" compilation by [Sekar *et al.*, 1992]: when a pattern matrix (P) with directions is discovered, then select one direction, otherwise select column index 1. Sekar *et al.* showed that such a compilation technique produces automata that are always smaller than the automata compiled using the standard left-to-right technique. Although Sekar *et al.* measure of automata size is not strictly correlated to output code size, experimental comparisons between "adaptive" and left-to-right versions of the $\mathcal{C}$ and $\mathcal{D}$ compilation algorithm show that the adaptive versions of compilation schemes usually produce shorter code.

Sekar *et al.* also showed the semantic correctness of their compiler: it produces automata that detect matching values whenever left-to-right automata do. My view of correctness is different: I only require that the sequential pattern matchings are compiled correctly, so that I leave room for more aggressive optimizations. Consider the following "ultra-adaptive" algorithm: when a pattern matrix (P) with directions is discovered, then select one direction, otherwise select some column index following heuristics aimed at reducing automata size. A first and efficient heuristic, useful both in $\mathcal{C}$ and $\mathcal{D}$-based ultra-adaptive algorithms, consist in selecting a *partial direction* (see the end of section 5.1), a partial direction being some column index that is a direction towards a maximal number of consecutive rows in (P), starting from the top. If there are more than one partial direction, they are further discriminated by using techniques that differ according to the basic compilation scheme used. In the $\mathcal{C}$-based algorithm, column indices are further selected by following some of the heuristics introduced in the case 3 at the end of section 5.1. A simple and reasonably satisfactory choice is here to first minimize the number of variables in the selected columns (in an attempt to avoid row duplication) and then, if necessary, to minimize the number of different root constructors (in an attempt to locally minimize automaton breadth). In the $\mathcal{D}$-based algorithm, the number of transitions between variable and non-variable patterns is minimized (in an attempt to locally minimize control transfer possibilities). The same heuristics apply to select one direction, when matrix (P) admits several directions.

The choice of ultra-adaptive pattern matching has some additional benefits in term of running time performance. When the compiled pattern matching defines a sequential matching predicate, correct tree-like automata perform exactly the tests needed to identify a matching value (otherwise, they would not be correct [Puel and Suárez, 1990]), these tests being performed only once. In this sense, correct tree-like automata have an optimal run-time behavior. As to correct automata with failures, although they also perform exactly the tests needed to identify a matching value, these tests can be performed several times. Nevertheless, correct automata with failures still do not perform useless tests, so that they tend to run faster than incorrect automata. Furthermore, some of the heuristics also tend to lower running time. This is the case of the partial directions in both schemes (some of the matching values are found by performing only the elementary tests that are indeed necessary) and of minimizing control transfer possibilities in the $\mathcal{D}$ scheme (tests are less likely to be performed several times). However, it is fair to say that this kind of optimization lies at the fine tuning level, since performing tests accounts for a small part of the running time of a typical ML program. As a consequence, the gain in code size obtained by using some $\mathcal{D}$-based algorithm outweights the gain in run-time efficiency obtained by using the corresponding $\mathcal{C}$-based algorithm.

As an example, the following table describes the size (in kilo-bytes and percentage) of the GAML compiler [Maranget, 1991] targeted for the sparc architecture (GAML is written in GAML), when compiled using various pattern matching compilation techniques:

	trees ($\mathcal{C}$)	failures ($\mathcal{D}$)
left-to-right	1176 (110 %)	1080 (101 %)
lazy (ultra-adaptive)	1128 (105 %)	1072 (100 %)

Thus, using the traditional left-to-right tree-based technique —as the SML/NJ and CAML V3.1 compilers do— means paying a penalty of 10 % in code size over the lazy dag-based technique. As a final conclusion, these figures show the practical benefits of using automata with failures instead of tree-like automata. These benefits can still be enjoyed when the lazy semantics for pattern matching is preferred.

References

[Appel and Macqueen, 1991] A. W. Appel and D. B. MacQueen, "*Standard ML of New Jersey*". International Symposium on Programming Language Implementation and Logic Programming 1991. LNCS 528.

[Augustsson, 1985] L. Augustsson, "*Compiling pattern matching*". Conference Functional Programming and Computer Architecture 1985.

[Berry and Lévy 1979] G. Berry and J.-J. Lévy. "*A survey of some syntactic results in the λ-calculus*". Conference Mathematical Foundations of Computer Science, 1979. LNCS 74.

[Curien, 1986] P. L. Curien, "*Categorical combinators, sequential algorithms and functional programming*". Pitman, London, and John Wiley and Sons, 1986.

[Hudak *et al.*, 1992] P. Hudak, S. Peyton-Jones and P. Wadler, "*Report on the Programming Language Haskell, Version 1.2*". University of Yale technical report, March 1992.

[Huet and Lévy, 1979] G. Huet, J.-J. Lévy, "*Call by Need Computations in Non-Ambiguous Linear Term Rewriting Systems*". INRIA, technical report 359, 1979. Reprinted in, J.-L. Lassez and G. Plotkin Editors, "*Computational Logic, Essays in Honor of Alan Robinson*". The MIT press, 1991.

[Johnsson, 1987] T. Johnsson, "*Compiling Lazy Functional Languages*". Ph.D. thesis, Chalmers University of Technology, Sweden, 1987.

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

[Kennaway, 1990] R. Kennaway, "*The Specificity Rule for Lazy Pattern Matching in Ambiguous Term Rewriting Systems*". Conference European Symposium On Programming 1990.

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

[Leroy, 1990] X. Leroy, "*The Zinc experiment: an Economical Implementation of the ML Language*", INRIA Technical Report 117, février 1990.

[Leroy *et al.*, 1993] X. Leroy *et al.* "*The Caml Light system, release 0.6*", Software and documentation distributed by anonymous FTP on `ftp.inria.fr`.

[Maranget, 1991] L. Maranget, "*GAML: A Parallel Implementation of Lazy ML*". Conference Functional Programming and Computer Architecture 1991.

[Maranget, 1992] L. Maranget, "*Compiling Lazy Pattern Matching*". Conference Lisp and Functional Programming 1992.

[Maranget, 1992] L. Maranget, "*La stratégie paresseuse*". Thèse de l'université de Paris VII, July 6 1992 (In french).

[Milner *et al.*, 1991] R. Milner, M. Tofte, R. Harper, "*The Definition of Standard ML*". The MIT Press, 1991.

[Plotkin, 1977] G. D. Plotkin, "*LCF Considered as a Programming Language*". Theoretical Computer Science, volume 5, pages 223–255, 1977.

[Puel and Suárez, 1990] L. Puel, A. Suárez, "*Compiling Pattern Matching by Term Decomposition*". Conference Lisp and Functional Programming 1990.

[Sekar *et al.*, 1992] R.C. Sekar, R. Ramesh and I.V. Ramakrishnan,"*Adaptive Pattern Matching*". Conference international Colloquium on Automata Languages and Programming 1992.

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

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

Unité de recherche Inria Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 Villers Lès Nancy
Unité de recherche Inria Rennes, Irisa, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Unité de recherche Inria Rhône-Alpes, 46 avenue Félix Viallet, 38031 Grenoble Cedex 1
Unité de recherche Inria Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Unité de recherche Inria Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex

Éditeur

Inria, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)

ISSN 0249-6399