Warnings for pattern matching

LUC MARANGET
Inria Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France
(*e-mail:* Luc.Maranget@inria.fr)

Abstract

We examine the ML pattern-matching anomalies of useless clauses and non-exhaustive matches. We state the definition of these anomalies, building upon pattern matching semantics, and propose a simple algorithm to detect them. We have integrated the algorithm in the Objective Caml compiler, but we show that the same algorithm is also usable in a non-strict language such as Haskell. Or-patterns are considered for both strict and non-strict languages.

1 Introduction

Pattern matching is one of the key features of the ML family of programing languages. Pattern matching favors reasoning (and programming) on a case by case basis. This style of reasoning calls for two basic checks: Are all cases considered? And, is any case subsumed by some others? In pattern matching terms, one wishes to write exhaustive pattern matching expressions whose clauses all are useful. ML compilers should normally flag pattern matching expressions that do not comply with those two basic assumptions or, in other words, expressions that exhibit such *anomalies*. By doing so, compilers provide an important help to programmers in detecting errors.

Techniques for compiling pattern matching fall into two classes, depending whether they target decision tress or backtracking automata. If we compare the two in their ability to provide diagnostics, the decision tree technique has one advantage: checks can be carried on the decision trees, since decision trees are complete and include no dead code. By contrast, backtracking automata cannot be easily analyzed. However, for reasons beyond the scope of this paper, some compiler designers choose the backtracking technique. This is, for instance, the case of Objective Caml (Leroy, X., Doligez, D. *et al.*, 2003) and our initial motivation was to carry out the pattern matching checks for this compiler. Our initial idea was of course to use a simplified version of compilation to decision trees. However, it appears that checking pattern matching requires much less work than compiling pattern matching, up to the point that our final algorithm can be considered as more than just a stripped down version of compilation to decision trees. Furthermore, studying checks *per se*, independently from compilation, finally yields a very general solution: our pattern matching analyzer gives valid answers for both ML and Haskell, whose semantics are quite different.

We divide our study into two parts. In part 1, we define pattern matching anomalies, introduce an algorithm that detects both anomalies, and prove the correctness of our algorithm with respect to the strict, lazy, and Haskell semantics of pattern matching. Part 2 describes the implementation of our algorithm. That is, we show how to refine and adapt our algorithm to the initial, practical, question of providing precise warnings to users. More specifically, in Section 5 we show how to strengthen a diagnostic of non-exhaustivity by supplying examples of non-matching values. Then, in Section 6, we examine *useless pattern detection*, an important refinement of simple useless clause detection. This refinement naturally arises in the presence of or-patterns, a convenient feature to group clauses with identical actions. Finally, we analyze the efficiency of our implementation and conclude.

PART ONE
An algorithm for detecting the anomalies

2 Patterns, values, etc.

Most ML values can be defined as ground terms over certain signatures. Signatures are introduced by *data type* definitions. For instance[1]:

```
type mylist = Nil | One of int | Cons of int * mylist
```

Values of type `mylist` are built from three *constructors* `Nil` (of zero arity, *i.e.* constant constructor), `One` (unary) and `Cons` (binary). Most of ML values can be expressed in that setting. Booleans are a two (constant) constructor type, the integer type is defined as possessing infinitely many (or 2^{31}) constant constructors, pairs are a type with a sole binary constructor (written with the infix operator "`,`"), etc.

More generally, our values are defined as (ground) terms over the constructor signatures. We make them explicit as follows:

$$\begin{array}{lll} v ::= & & \textbf{Values} \\ & c(v_1, v_2, \ldots, v_a) & a \geq 0 \end{array}$$

In examples, we systematically omit () after constants constructors, so as to match Objective Caml syntax (we write `Nil`, `true`, `0`, etc.). This simple definition of values as terms suffices to our purpose of studying pattern matching anomalies in call-by-value ML. However,

- A proper semantics for pattern matching in a lazy language should define partial values, which we do in section 4.
- Not all values can be seen as terms built over known signatures, for instance, consider functions or values of a type that exhibits parametric polymorphism. However, we are not interested in the exact nature of all the types and values

[1] In this paper, we use Objective Caml syntax

of ML. Instead, we shall rely on the following informal axiom: given any type t, we assume the existence of at least one value that possesses t as a type.

Patterns are used to discriminate amongst values. More precisely a pattern describes a set of values with a common prefix. That is, patterns are terms with variables and a given pattern p describes its instances $\sigma(p)$ where σ ranges over substitutions. However, we wish to stay close to programming practice and define patterns as follows:

$p ::=$	**Patterns**
$\_$	wildcard
$c(p_1, p_2, \ldots, p_a)$	constructed pattern $a \geq 0$
$(p_1 \mid p_2)$	or-pattern

Variables in fact do not appear in our definition of patterns. For our purpose, they can be replaced by the wildcard symbol "$\_$". One can see wildcards as variables whose exact names are irrelevant. Additionally, our patterns feature "or-patterns" as offered by modern implementations of the ML language.

Furthermore, it is important to remark that patterns are type correct, that is, we assume that patterns follow the sorting discipline enforced by some declarations of data types. In practice, the Objective Caml compiler performs pattern matching analysis after the typing phase, so that patterns do hold type annotations. In our formal treatment we avoid making those annotation explicit everywhere, this would be quite cumbersome and of little explanatory value. However, when appropriate, we sometime show type annotation as $(p : t)$.

In the usual theory of terms, a term v (of type t) is an *instance* of a pattern p (of type t) when the pattern describes the prefix of the term. That is, when there exists a substitution σ such that $\sigma(p) = v$. In the case of linear patterns, where no variable appears more than once in a given pattern, the instance relation can be defined inductively and the exact names of variables are irrelevant.

Definition 1 (*Instance relation*)
Given any pattern p and a value v, such that p and v are of a common type, the instance relation $p \preceq v$ is defined as:

$$\begin{array}{rcll} \_ & \preceq & v & \\ (p_1 \mid p_2) & \preceq & v & \text{iff } p_1 \preceq v \text{ or } p_2 \preceq v \\ c(p_1, \ldots, p_a) & \preceq & c(v_1, \ldots, v_a) & \text{iff } (p_1 \cdots p_a) \preceq (v_1 \cdots v_a) \\ (p_1 \cdots p_a) & \preceq & (v_1 \cdots v_a) & \text{iff } p_i \preceq v_i, \text{ for all } i \in [1..a] \end{array}$$

It is important to notice again that pattern matching is defined in a typed context. In particular $\_ \preceq v$ does not holds for any value v, but only for value v of the specified type t, which can be made explicit with the notation $(\_ : t)$. Moreover, as a consequence of our informal axiom "types are not empty", any pattern p admits at least one instance. This can be shown by a simple induction on pattern structure, given that the axiom "types are not empty" implies that, for any type t, there exists some value v such that $(\_ : t) \preceq v$.

In definition 1 above we used the very convenient notations $\vec{p} = (p_1 \cdots p_a)$ and $\vec{v} = (v_1 \cdots v_a)$ in the last two lines of the definition of the instance relation. Notations $\vec{p}$ and $\vec{v}$ stand for (row) vectors of patterns and values respectively. Observe that we just defined the instance relation on vectors. We shall also consider matrices of patterns $P = (p^i_j)$, of size $m \times n$ where m is P *height* (number of rows) and n is P *width* (number of columns). Boundary cases deserve a few notations: matrices with no row ($m = 0$) are written $\emptyset$; while non-empty matrices of empty rows ($m > 0$ and $n = 0$) are written $()$. Finally, we sometime denote row number i of matrix P as $\vec{p}^{\,i}$.

We recall the definition of ML pattern matching in this convenient framework of matrices and vectors.

Definition 2 (*ML pattern matching*)
Let P be a pattern matrix and $\vec{v} = (v_1 \cdots v_n)$ be a value vector, where n is equal to the width of P. Row number i in P *filters* $\vec{v}$, if and only if the following two conditions hold.

1. $(p^i_1 \cdots p^i_n) \preceq (v_1 \cdots v_n)$
2. $\forall j < i,\ (p^j_1 \cdots p^j_n) \npreceq (v_1 \cdots v_n)$

We shall also say that $\vec{v}$ matches row number i in P.

In other words, vector $\vec{v}$ matches the first row it is an instance of, starting from the top of matrix P. Again, we leave typing details implicit: all rows in P and $\vec{v}$ must be of a common type.

Example 1
Consider the following matrix P of size 5×2, and whose patterns are of type `mylist`.

$$P = \begin{pmatrix} \texttt{Nil} & \_ \\ \_ & \texttt{Nil} \\ \texttt{One(0)} & \_ \\ \_ & \texttt{One(0)} \\ \_ & \_ \end{pmatrix}$$

Then, for instance we have:

1. Vector $\vec{v} = (\texttt{Nil}\ \texttt{Nil})$ matches the first row of P, since $(\texttt{Nil}\ \_) \preceq (\texttt{Nil}\ \texttt{Nil})$ (because $p^1_1 = \texttt{Nil} \preceq \texttt{Nil} = v_1$) and $p^1_2 = \_ \preceq \texttt{Nil} = v_2$).
2. Vector $\vec{w} = (\texttt{One(0)}\ \texttt{Nil})$ matches the second row of P, since we have
   (a) $(\_\ \texttt{Nil}) \preceq (\texttt{One(0)}\ \texttt{Nil})$, that is $\vec{w}$ is an instance of the second row of matrix P.
   (b) $(\texttt{Nil}\ \_) \npreceq (\texttt{One(0)}\ \texttt{Nil})$ (because $p^1_1 = \texttt{Nil} \npreceq \texttt{One(0)} = w_1$), that is, $\vec{w}$ is not an instance of the first row of matrix P.
3. Vector $\vec{z} = (\texttt{One(1)}\ \texttt{One(1)})$ matches the fifth row of matrix P, since we have:
   (a) Vector $\vec{z}$ is an instance if $\vec{p}^{\,5} = (\_\,\_)$ (as any value vector of the appropriate type is).
   (b) Additionally, vector $\vec{z}$ is not an instance of any of the first four rows of P. For instance, $\texttt{0} \npreceq \texttt{1}$ implies $\texttt{One(0)} \npreceq \texttt{One(1)}$, which in turn implies that vector $\vec{z}$ cannot be an instance of rows number 3 and 4 of matrix P.

As we have already noticed, patterns can be interpreted as the set of their instances. Similarly matrices can be interpreted as the union of the instances of their rows.

Definition 3 (*Instance relation for matrices*)
Let P be a pattern matrix with n columns and m rows, and let $\vec{v} = (v_1 \cdots v_n)$ be a value vector. Vector $\vec{v}$ is an instance of matrix P, written $P \preceq \vec{v}$, if and only if there exists an row number i ($i \in [1 \ldots m]$) such that:

$$(p_1^i \cdots p_n^i) \preceq (v_1 \cdots v_n)$$

ML pattern matching can be reformulated with this new definition as: vector $\vec{v}$ matches row number i in matrix P, if and only if $P^{[1\ldots i)} \npreceq \vec{v}$ and $\vec{p}^{\,i} \preceq \vec{v}$, where matrix $P^{[1\ldots i)}$ is the $(i-1) \times n$ matrix consisting of the rows of P that precede row number i.

3 The useful clause problem

We express pattern matching anomalies in the matrix framework.

Definition 4 (*Exhaustiveness*)
Let P be a pattern matrix. Matrix P is exhaustive, if and only if, for all value vectors $\vec{v}$ of the appropriate type, there exists a row in P that filters $\vec{v}$ in the sense of definition 2.

Definition 5 (*Useless clause*)
Let P be a pattern matrix. Row number i in P is useless, if and only if there does not exists a value vector $\vec{v}$ that matches row number i in the sense of definition 2.

Example 2
Let P and Q be the following two pattern matrices

$$P = \begin{pmatrix} \texttt{Nil} & \texttt{\_} \\ \texttt{\_} & \texttt{Nil} \end{pmatrix} \qquad Q = \begin{pmatrix} \texttt{Nil} & \texttt{\_} \\ \texttt{\_} & \texttt{Nil} \\ \texttt{One(\_)} & \texttt{\_} \\ \texttt{\_} & \texttt{One(\_)} \\ \texttt{Cons (\_,\_)} & \texttt{\_} \\ \texttt{\_} & \texttt{Cons(\_,\_)} \end{pmatrix}$$

Matrix P is not exhaustive, since, for instance, vector $\vec{v} = ($`One (0) One (0)`$)$ does not match any row of P.

By contrast, matrix Q is exhaustive. Let us consider any value vector $\vec{v}$ of the appropriate type. Then, v_1 and v_2 are instances of the patterns `Nil`, `One(_)`, or `Cons(_,_)`. That is, we may partition values into nine sets denoted by nine different pattern vectors. It turns out that this partition is precise enough to apply definition 2.

If $\vec{v}$ is an instance of...	then, $\vec{v}$ matches row number...
`(Nil Nil)` `(Nil One(_))` `(Nil Cons(_,_))`	1
`(One(_) Nil)` `(Cons(_,_) Nil)`	2
`(One(_) One(_))` `(One(_) Cons(_,_))`	3
`(Cons(_,_) One(_))`	4
`(Cons(_,_) Cons(_,_))`	5

As another consequence, one may observe that row number 6 of matrix Q is useless.

Because we use the ML definition of pattern matching (definition 2) we claim that the two definitions above express what is generally understood by "an exhaustive pattern" and "an useless clause", However it is intuitively clear that the two questions are quite similar, and in fact they can be expressed using the following definition.

Definition 6 (*Useful clause*)
Let P be a pattern matrix of size $m \times n$ and let $\vec{q}$ be a pattern vector of size n. Vector $\vec{q}$ is useful with respect to matrix P, if and only if

$$\exists \vec{v}, P \not\preceq \vec{v} \wedge \vec{q} \preceq \vec{v}.$$

We write $\mathcal{U}(P, \vec{q})$ for the formula above We also note $\mathcal{M}(P, \vec{q})$ the following set of *matching value vectors*:

$$\mathcal{M}(P, \vec{q}) = \{\vec{v} \mid P \not\preceq \vec{v} \wedge \vec{q} \preceq \vec{v}\}.$$

Thus, $\mathcal{U}(P, \vec{q})$ simply means that $\mathcal{M}(P, \vec{q})$ is not empty.

Proposition 1

1. Matrix P is exhaustive, if and only if $\mathcal{U}(P, (\_\ \_ \cdots \_))$ is false.
2. Row number i in matrix P is useless, if and only if $\mathcal{U}(P^{[1\ldots i)}, \vec{p^i})$ is false.

Proof

1. If P is exhaustive then any value vector $\vec{v}$ is an instance of some row in P. That is, we have $P \preceq \vec{v}$ for all vectors $\vec{v}$, and thus $\mathcal{U}(P, \vec{q})$ is false for all pattern vectors $\vec{q}$. Conversely, if $\mathcal{U}(P, (\_\ \_ \cdots \_))$ is false, then, as $(\_\ \_ \cdots \_) \preceq \vec{v}$ for all $\vec{v}$, any $\vec{v}$ must be an instance of P (in the sense of definition 3). That is, given $\vec{v}$, there exists a non-empty set of integers I such that $\vec{p^i} \preceq \vec{v}$ for all i in I. Then, vector $\vec{v}$ matches (in the sense of definition 2) row number i_0 in P, where i_0 is the minimal element of I. Thus, P is exhaustive.
2. Corollary of the reformulation of definition 2 as $P^{[1\ldots i)} \not\preceq \vec{v}$ and $\vec{p^i} \preceq \vec{v}$.

$\square$

Our framework of two separate definitions 2 and 3 exposes that, as far as pattern matching anomalies are concerned, the matching predicate can be simplified. In fact, a potential problem was possible only in the case of exhaustiveness, and we solve it essentially by showing that $\vec{v}$ matches some row in P (definition 2) is the same thing as $P \preceq \vec{v}$ (definition 3).

3.1 Solving the useful clause problem

In this section we compute $\mathcal{U}$ recursively. Throughout this inductive definition we shall argue about correctness with respect to definition 6. The computation of $\mathcal{U}$ owes much to the traditional compilation of ML pattern matching to decision trees ((Pettersson, 1992) gives a modern presentation of this quite ancient compilation scheme).

Let thus P be a pattern matrix of size $m \times n$ and $\vec{q}$ be a pattern vector. Induction proceeds by decomposing P and $\vec{q}$ along some column.

Base case If there is no column (*i.e.* $n = 0$), then $\vec{q}$ is the empty pattern vector written $()$. The set of $\vec{q}$ instances consists of the one empty value vector, also written $()$. Then, the value of $\mathcal{U}(P, ())$ depends upon the number of rows m of matrix P.

1. If P has some rows (*i.e.* $m > 0$), then the first row of P admits the empty vector as an instance and $\mathcal{U}(\left(\,\right), ())$ is false.
2. If m is zero, then P is a very empty matrix (no rows, no columns). Since P has no rows, it does not have any instance either. Thus, $\mathcal{U}(\emptyset, ())$ is true. More generally, $\mathcal{U}(\emptyset, \vec{q})$ is true for any vector $\vec{q}$ of any size n.

Base cases are summarized as follows:

$$\mathcal{U}(\left(\,\right), ()) = \mathit{False} \qquad \mathcal{U}(\emptyset, \vec{q}) = \mathit{True}$$

$$\mathcal{M}(\left(\,\right), ()) = \emptyset \qquad \mathcal{M}(\emptyset, \vec{q}) = \{\, \vec{v} \mid \vec{q} \preceq \vec{v} \,\}.$$

Induction If there are columns ($n > 0$), then there are three subcases depending upon the nature of pattern q_1 (in fact any column would work, we choose the first column of P and $\vec{q}$ for simplicity).

1. Pattern q_1 is a constructed pattern, that is $q_1 = c(r_1, \ldots, r_a)$. Then, the first component of any instance of $\vec{q}$ necessarily is $v_1 = c(w_1, \ldots, w_a)$. From matrix P we extract the new *specialized* matrix $\mathcal{S}(c, P)$, such that predicates $P \preceq (c(w_1, \ldots, w_a) \cdots v_n)$ and $\mathcal{S}(c, P) \preceq (w_1 \cdots w_a\ v_2 \cdots v_n)$ are equivalent. The new matrix $\mathcal{S}(c, P)$ is of width $a + n - 1$ and its rows are defined from the rows of P, according to the first component of these rows:

p_1^i	$\mathcal{S}(c, P)$
$c(t_1, \ldots, t_a)$	$t_1 \cdots t_a \quad p_2^i \cdots \quad p_n^i$
$c'(t_1, \ldots, t_{a'})$ $(c' \neq c)$	No row
$\_$	$\_ \cdots \_ \quad p_2^i \cdots \quad p_n^i$
$(t_1 \mid t_2)$	$\mathcal{S}(c, \begin{pmatrix} t_1 & p_2^i \cdots p_n^i \\ t_2 & p_2^i \cdots p_n^i \end{pmatrix})$

Notice that a given row $\vec{p}^{\,i}$, may induce one, none or several rows in $S(c, P)$.

Then, we state:

$$\mathcal{U}(P, (c(r_1, \ldots, r_a)\ q_2 \cdots q_n)) = \mathcal{U}(\mathcal{S}(c, P), (r_1 \cdots r_a\ q_2 \cdots q_n))$$

This formula is a direct consequence of $\mathcal{S}(c, P)$ definition (the exact proof is quite similar to the proof of the next case, which is detailed).

2. Pattern q_1 is a wildcard $\_$. Let $\Sigma = \{c_1, c_2, \ldots, c_z\}$ be the constructors that appear as root constructors of the patterns of P first column (and also as root constructors of their arguments when they are or-patterns).

   (a) We further assume that Σ constitutes a full constructor signature and write a_k for the arity of constructor c_k. Then, we state:

$$\mathcal{U}(P, ((\_ : t)\ q_2 \cdots q_n)) = \bigvee_{k=1}^{z} \mathcal{U}(\mathcal{S}(c_k, P)(\overbrace{\_ \cdots \_}^{a_k \text{ times}}\ q_2 \cdots q_n))$$

   We now show the correctness of the formula above, which basically holds because of typing. Let us first state a key property of matrix specialization. For all value vectors $(w_1 \cdots w_a\ v_2 \cdots v_n)$ and constructor c (of type $t_1 \times \cdots \times t_a \rightarrow t$) we have:

$$P \preceq (c(w_1, \ldots, w_a)\ v_2 \cdots v_n) \iff \mathcal{S}(c, P) \preceq (w_1 \cdots w_a\ v_2 \cdots v_n)$$

   The proof of this key property is mechanical, from the definition of the instance relation. It is worth noticing that the equivalence still holds when Σ is not a complete signature, or when constructor c is not in Σ. Thus, the equivalence above suffices to prove previous case 1.
   Now, let us assume there exists a value vector $\vec{v}$ in $\mathcal{M}(P, \vec{q})$. By typing and because Σ is a complete constructor signature, there exists constructor c_k, such that $v_1 = c_k(w_1, \ldots, w_{a_k})$. Then, by definition of $\mathcal{M}$ we have $P \not\preceq \vec{v}$ and $\vec{q} \preceq \vec{v}$. Hence, we have $\mathcal{S}(c_k, P) \not\preceq (w_1 \cdots w_{a_k}\ v_2 \cdots v_n)$ (key property of specialization) and $(\_ \cdots \_\ q_2 \cdots q_n) \preceq (w_1 \cdots w_{a_k}\ v_2 \cdots v_n)$ (definition of $\preceq$). Thus, value vector $(w_1 \cdots w_{a_k}\ v_2 \cdots v_n)$ belongs to $\mathcal{M}(\mathcal{S}(c_k, P), \ldots)$ and thus $\mathcal{U}(\mathcal{S}(c_k, P), \ldots)$ holds. Conversely, if any of the $\mathcal{U}(\mathcal{S}(c_k, P), \ldots)$ holds, then there exists some $(w_1 \cdots w_{a_k}\ v_2 \cdots v_n)$ in $\mathcal{M}(\mathcal{S}(c_k, P), \ldots)$. Thus, there exists $\vec{v} = (c_k(w_1, \ldots, w_{a_k})\ v_2 \cdots v_n)$ acting as a witness of $\mathcal{U}(P, \ldots)$ validity (by the key property of specialization).

   (b) We then consider the case where Σ is not a complete constructor signature, including the case where Σ is empty (*i.e.* the first column of P is made of wildcards and of or-patterns thereof). We define the following *default* matrix $\mathcal{D}(P)$ of width $n - 1$.

p_1^i	$\mathcal{D}(P)$
$c_k(t_1, \ldots, t_{a_k})$	No row
$\_$	$p_2^i \cdots p_n^i$
$(t_1 \mid t_2)$	$\mathcal{D}\left(\begin{pmatrix} t_1 & p_2^i \cdots p_n^i \\ t_2 & p_2^i \cdots p_n^i \end{pmatrix}\right)$

In the case where Σ is empty, computing $\mathcal{D}(P)$ essentially amounts to erasing P first column (some rows may get duplicated because of or-patterns). We state:

$$\mathcal{U}(P, ((\_ : t)\ q_2 \cdots q_n)) = \mathcal{U}(\mathcal{D}(P), (q_2 \cdots q_n))$$

It is important to notice that we significantly depart from compilation to decision trees, which requires additional recursive calls on specialized matrices when Σ is non-empty.
To prove the formula, it is convenient to state some "key properties" of the default matrix. For all value vectors $\vec{v}$, we have:

$$P \not\preceq (v_1\ v_2 \cdots v_n) \Longrightarrow \mathcal{D}(P) \not\preceq (v_2 \cdots v_n)$$

For, given some vector $\vec{v}$ such that $P \not\preceq \vec{v}$, we have $\vec{p}^{\,i} \not\preceq \vec{v}$ for all rows $\vec{p}^{\,i}$ of P. Now consider rows $\vec{p}^{\,i}$ such that $p_1^i = \_$, then, from $(\_\ p_2^i \cdots p_n^i) \not\preceq \vec{v}$ and $\_ \preceq v_1$, we get $(p_2^i \cdots p_n^i) \not\preceq (v_2 \cdots v_n)$. Furthermore, consider rows such that $p_1^i = (\cdots|\_|\cdots)$ (p_1^i is a or-pattern whose one alternative is a wildcard), then, from $(r_1 \,|\, r_2) \not\preceq v \Longrightarrow r_1 \not\preceq v \wedge r_2 \not\preceq v$, we also get $(p_2^i \cdots p_n^i) \not\preceq (v_2 \cdots v_n)$. That is, we have $\mathcal{D}(P) \not\preceq (v_2 \cdots v_n)$.
Conversely, for all constructors c (of the appropriate type) not in Σ and value vectors $(w_1 \cdots w_a\ v_2 \cdots v_n)$, we have:

$$\mathcal{D}(P) \not\preceq (v_2 \cdots v_n) \Longrightarrow P \not\preceq (c(w_1, \ldots, w_a)\ v_2 \cdots v_n)$$

And, if Σ is empty, for all value vectors $(v_1\ v_2 \cdots v_n)$, we have the more direct converse implication:

$$\mathcal{D}(P) \not\preceq (v_2 \cdots v_n) \Longrightarrow P \not\preceq (v_1\ v_2 \cdots v_n)$$

(Proofs from the definition of $\not\preceq$ are easy, but tedious.)
We now prove $\mathcal{U}(P, ((\_ : t)\ q_2 \cdots q_n)) = \mathcal{U}(\mathcal{D}(P), (q_2 \cdots q_n))$ Let us consider $\vec{v}$ in $\mathcal{M}(P, \vec{q})$. we immediately have $(v_2 \cdots v_n) \in \mathcal{M}(\mathcal{D}(P), (q_2 \cdots q_n))$, by our first key property of the default matrix. Conversely, if t is a ML data type (which is granted if Σ is not empty) and that there exists $(v_2 \cdots v_n)$ in $\mathcal{M}(\mathcal{D}(P), \ldots)$. Then, by the hypothesis "Σ does not hold all the constructors of type t" we know that there exists some constructor c of type $t_1 \times \cdots \times t_a \to t$ such that $c \notin \Sigma$. Furthermore, by our informal axiom "types are not empty", there exist values $w_1, \ldots, w_a$ of respective types $t_1, \ldots t_a$. Then, vector $\vec{v} = (c(w_1, \ldots, w_a)\ v_2 \cdots v_n)$ is a witness of $\mathcal{U}(P, \vec{q})$ validity, by the first converse key property of $\mathcal{D}(P)$. The case $\Sigma = \emptyset$ is similar and more direct, since there exists some value v_1 of type t (which is not necessarily a ML data type).

3. Pattern q_1 is a or-pattern $(r_1 \,|\, r_2)$. Then we state:

$$\mathcal{U}(P, ((r_1 \,|\, r_2)\ q_2 \cdots q_n)) = \mathcal{U}(P, (r_1\ q_2 \cdots q_n)) \vee \mathcal{U}(P, (r_2\ q_2 \cdots q_n))$$

Correctness is a corollary of the definition of the instance relation on or-patterns: $(r_1 \,|\, r_2) \preceq v$ means $r_1 \preceq v$ or $r_2 \preceq v$.

Now that we know how to compute $\mathcal{U}$, we can detect pattern matching anomalies. Given some pattern matching expression:

match ... with p_1 -> e_1 | p_2 -> e_2 | ... | p_m -> e_m

We check exhaustiveness by:

$$\mathcal{U}(\begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_m \end{pmatrix}, (\_))$$

While the usefulness of clause number i is checked by computing:

$$\mathcal{U}(\begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_{i-1} \end{pmatrix}, (p_i))$$

4 Lazy pattern matching

4.1 Lazy pattern matching in theory

In previous sections we only considered strict ML. In a strict language we can define ML pattern matching as a predicate operating on terms, which also are the values of program expressions. In other word, pattern matching apply to completely evaluated expressions, or normal forms.

Studying pattern matching in a lazy language such as Haskell requires a more sophisticated semantical setting. Essentially, lazy language manipulate values that are known partially and, more significant to our study, pattern matching operates on such incomplete values.

$v ::=$		**Partial Values**
	Ω	Undefined value
	$c(v_1, v_2, \ldots, v_a)$	(constructor) head-normal form.

Definition 1 of the instance relation for patterns and values apply unchanged to lazy values (we have $\_ \preceq \Omega$). Thus we keep the same notation $\preceq$, and maintain definition 3 of the instance relation for matrices.

However we cannot keep definition 2 of ML pattern matching.

Example 3
Let us consider a simple example[2].

```
case e of True -> 1 | _ -> 2
```

[2] In this section, we use Haskell syntax

If the root symbol of expression e is not a constructor, its partial value is Ω. Then, since `True` $\not\preceq \Omega$ and $\_ \preceq \Omega$, the value of the whole expression is `2`. But, if we compute e further, its value may become `True` and then the value of the whole expression becomes `1`. Something is wrong, since the value of the whole expression changed from `1` to `2`.

More generally, partial values and computation interact. Let us consider some expression e. If the root symbol of expression e is a constructor, this root symbol cannot be reduced any further and we can consider that e is a head-normal form and express the current value of e as $c(v_1, v_2, \ldots, v_a)$ (see (Huet & Lévy, 1991) for a more precise treatment in the context of term rewriting systems). Otherwise, the current value of expression e is Ω. Then, we consider various "current values" along the evaluation of e. As constructors cannot be reduced, those values are increasing according to the following precision ordering.

Definition 7 (*Precision ordering*)
Relation $\preceq_\Omega$ is defined on pairs of values (v, w) as follows:

$$\begin{array}{rcll} \Omega & \preceq_\Omega & w & \\ c(v_1, \ldots, v_a) & \preceq_\Omega & c(w_1, \ldots, w_a) & \text{iff } (v_1 \cdots v_a) \preceq_\Omega (w_1 \cdots w_a) \\ (v_1 \cdots v_n) & \preceq_\Omega & (w_1 \cdots w_n) & \text{iff } v_i \preceq_\Omega w_i, \text{ for all } i \in [1 \ldots n] \end{array}$$

To be of practical use, a predicate $\mathcal{P}$ that defines pattern matching must be monotonic, That is, $\mathcal{P}(w)$ holds for all w such $v \preceq_\Omega w$ when $\mathcal{P}(v)$ holds. With monotonic predicates, matching decisions do not change along computations. One should notice that, given any pattern p, predicate $p \preceq v$ is monotonic in v. Example 3 shows that the predicate $P \not\preceq \vec{v}$ is not monotonic in general.

We thus need a new definition of pattern matching. For the moment, we leave most of lazy pattern matching unspecified.

Definition 8 (*General (lazy) pattern matching*)
Let $\mathcal{P}(P, \vec{v})$ be a predicate defined over pattern matrices P and value vectors $\vec{v}$, where the size n of $\vec{v}$ is equal to the width of P. Row number i in P *filters* $\vec{v}$, if and only if the following condition holds.

$$\mathcal{P}(P^{[1\ldots i)}, \vec{v}) \wedge \vec{p}^{\,i} \preceq \vec{v}$$

We call $\mathcal{P}$ the *disambiguating* predicate and now look for sufficient conditions on $\mathcal{P}$ that account for our intuition of pattern matching in a lazy language.

1. Pattern matching is deterministic, in the sense that at most one clause is matched. Hence, for all P and $\vec{v}$, we assume:
$$\mathcal{P}(P, \vec{v}) \Longrightarrow P \not\preceq \vec{v}$$
2. Matching the first row of a matrix reduces to the instance relation. Hence, for all $\vec{v}$, we assume:
$$\mathcal{P}(\emptyset, \vec{v}) = \mathit{True}$$
3. We require predicate $\mathcal{P}$ to be monotonic in its value component. That is,

given any matrix P, for all value vectors $\vec{v}$ and $\vec{w}$, we assume:

$$\mathcal{P}(P, \vec{v}) \wedge \vec{v} \preceq_\Omega \vec{w} \Longrightarrow \mathcal{P}(P, \vec{w})$$

The three conditions above are our basic assumptions on $\mathcal{P}$. We further define $\mathcal{U}_\mathcal{P}$ and $\mathcal{M}_\mathcal{P}$ as $\mathcal{U}$ and $\mathcal{M}$ (definition 6) parameterized by $\mathcal{P}$.

Now, given a definition of pattern matching over lazy values, we face the temptation to assume that the computation of $\mathcal{U}$ described in section 3.1 still works for $\mathcal{U}_\mathcal{P}$. More precisely, by finding additional sufficient conditions on predicate $\mathcal{P}$ we aim at solving the useful clause problem as already done. Thus, we review the proof that the inductive computation of $\mathcal{U}$ is correct.

Base cases follow from basic assumptions.

- By our first basic assumption and since $\left(\,\right) \preceq ()$, we have $\mathcal{P}(\left(\,\right), ()) = \mathit{False}$. Thus we have $\mathcal{U}_\mathcal{P}(\left(\,\right), ()) = \mathit{False}$.
- By our second basic assumption, we directly get $\mathcal{U}_\mathcal{P}(\emptyset, \vec{q}) = \mathit{True}$.

To prove the inductive cases, it suffices to restate the "key" properties of Section 3.1, replacing $P \npreceq \vec{v}$ by $\mathcal{P}(P, \vec{v})$. First, for all matrices P, value vectors $(w_1 \cdots w_a\ v_2 \cdots v_n)$ and constructors c (of the appropriate type) we assume:

$$\mathcal{P}(P, (c(w_1, \ldots, w_a)\ v_2 \cdots v_n)) \iff \mathcal{P}(\mathcal{S}(c, P), (w_1 \cdots w_a\ v_2 \cdots v_n)) \tag{1}$$

Additionally, for all value vectors $\vec{v}$ we assume:

$$\mathcal{P}(P, (v_1\ v_2 \cdots v_n)) \Longrightarrow \mathcal{P}(\mathcal{D}(P), (v_2 \cdots v_n)) \tag{2}$$

Furthermore, given any matrix P, let Σ be set of the root constructors of P first column. If Σ is not empty, for all constructors c not in Σ and all values vectors $(w_1 \cdots w_a\ v_2 \cdots v_n)$ we assume:

$$\mathcal{P}(\mathcal{D}(P), (v_2 \cdots v_n)) \Longrightarrow \mathcal{P}(P, (c(w_1, \ldots, w_a)\ v_2 \cdots v_n)) \tag{3}$$

If Σ is empty, and for all value vectors $(v_1\ v_2 \cdots v_n)$ we assume instead:

$$\mathcal{P}(\mathcal{D}(P), (v_2 \cdots v_n)) \Longrightarrow \mathcal{P}(P, (v_1\ v_2 \cdots v_n)) \tag{4}$$

It is not obvious at all that those properties are sufficient to prove the validity of the inductive definition of $\mathcal{U}_\mathcal{P}$ in the lazy case, since Ω does not show in section 3.1. Indeed $\mathcal{P}$ monotonicity plays some part here.

Let us reconsider case 2-(a). We want to prove the formula:

$$\mathcal{U}_\mathcal{P}(P, ((\_ : t)\ q_2 \cdots q_n)) = \bigvee_{k=1}^{z} \mathcal{U}_\mathcal{P}(\mathcal{S}(c_k, P), (\overbrace{\_ \cdots \_}^{a_k \text{ times}}\ q_2 \cdots q_n))$$

Given some value $\vec{v}$ in $\mathcal{M}_\mathcal{P}(P, \ldots)$ the case when v_1 is not Ω is proved just as we did in the strict case, using key property (1). However, the case when $v_1 = \Omega$ is still pending. Then, by monotonicity of $\mathcal{P}$ (and $\preceq$), the non-empty type axiom, and for any constructor c (in the signature of type t), there exists $(w_1 \cdots w_a)$ such that $(c(w_1, \ldots, w_a)\ v_2 \cdots v_n) \in \mathcal{M}_\mathcal{P}(P, \ldots)$. Thus, we can conclude that $\mathcal{U}_\mathcal{P}(\mathcal{S}(c_k, P), \ldots)$ holds for all c_k in Σ.

The proof of all remaining inductive cases does not change. Case 2-(b) is proved by using properties (2), (3) and (4), property (1) suffices to prove case 1, and case 3 stems directly from the definition of $\preceq$ for or-patterns.

As an immediate consequence of the correctness of algorithm $\mathcal{U}$, the useless clause problem is now solved in the lazy case (see second item in proposition 1). However, the exact formulation of exhaustiveness needs a slight change. Reconsider Example 3:

`case` e `of True -> 1 | _ -> 2`

By definition of $\preceq$, $\texttt{True} \preceq \Omega$ does not hold, hence Ω cannot match first clause. However, $\mathcal{P}((\ \texttt{True}\ ), \Omega)$ does not hold either, by monotonicity. Hence, there is no row that filters value Ω and definition 4 would flag this matching as non-exhaustive, a clear contradiction with our intuition of exhaustiveness. Thus, we now directly define an exhaustive matrix P from the condition $\mathcal{U}_\mathcal{P}(P, (\_ \cdots \_)) = \mathit{False}$. That is, P is exhaustive, if and only if for all vectors $\vec{v}$, $\mathcal{P}(P, \vec{v})$ does not hold. By this new definition, the example is exhaustive. For all value v of type `bool`, we have:

$$\begin{pmatrix} \texttt{True} \\ \_ \end{pmatrix} \preceq v$$

Hence, by our first basic assumption, for all v, $\mathcal{P}(P, v)$ does not hold.

4.2 Lazy pattern matching, à la Laville

Laville's definition of lazy pattern matching (Laville, 1991) stems directly from the need of a monotonic $\mathcal{P}$: if we decide that some term is evaluated enough not to match a pattern, we want this to remain true when the term is evaluated further. By definition, matrix P and value $\vec{v}$ are incompatible, written $P \mathbin{\#} \vec{v}$, when making $\vec{v}$ more precise cannot produce an instance of P. That is, $P \mathbin{\#} \vec{v}$ means:

$$\forall \vec{w}, \vec{v} \preceq_\Omega \vec{w} \Longrightarrow P \npreceq \vec{w}$$

Definition 9 (*Lazy pattern matching (Laville)*)
State $\mathcal{P}(P, \vec{v}) = P \mathbin{\#} \vec{v}$ in the generic definition 8.

Our first basic assumption follows by stating $\vec{w}$ to be $\vec{v}$ in definition, the second assumption follows from $\emptyset \npreceq \vec{v}$ for all $\vec{v}$, and monotonicity is a consequence of $\preceq_\Omega$ transitivity.

Incompatibility is the most general $\mathcal{P}$ in the following sense: for all predicate $\mathcal{P}$, all matrices and value vectors $\vec{v}$, we have.

$$\mathcal{P}(P, \vec{v}) \Longrightarrow P \mathbin{\#} \vec{v}$$

For, if P and $\vec{v}$ are compatible (*i.e.* not incompatible), then there exists $\vec{w}$, with $\vec{v} \preceq_\Omega \vec{w}$ and $P \preceq \vec{w}$. Thus, by the first basic assumption, $\mathcal{P}(P, \vec{w})$ does not hold, and, by $\mathcal{P}$ monotonicity, $\mathcal{P}(P, \vec{v})$ does not hold either.

Incompatibility is easily computed by the following rules:

$$
\begin{array}{rcll}
c(p_1,\ldots,p_a) & \# & c'(v_1,\ldots,v_{a'}) & \\
c(p_1,\ldots,p_a) & \# & c(v_1,\ldots,v_a) & \text{iff } (p_1\cdots p_a)\,\#\,(v_1\cdots v_a) \\
(p_1\cdots p_n) & \# & (v_1\cdots v_n) & \text{iff there exists } i\in[1\ldots n], p_i\,\#\,v_i \\
(p_1\,|\,p_2) & \# & v & \text{iff } p_1\,\#\,v \text{ and } p_2\,\#\,v \\
\\
P & \# & \vec{v} & \text{iff for all } i\in[1\ldots n], \vec{p}^{\,i}\,\#\,\vec{v}
\end{array}
$$

It is then routine to show that incompatibility meets key properties.

Laville's definition is quite appealing as a good, implementation independent, definition of lazy pattern matching. However, there is a slight difficulty: predicate $P\,\#\,\vec{v}$ is not sequential in the sense of (Kahn & Plotkin, 1978) in $\vec{v}$ for any matrix P. This means that its compilation on an ordinary, sequential, computer is problematic (Maranget, 1992; Sekar *et al.*, 1992). As a consequence of this difficulty, the Haskell committee adopted another semantics for pattern matching. Their definition is aware of the presence of Ω and solves the difficulty by specifying a left-to-right testing order of patterns arguments.

4.3 Pattern matching in Haskell

By interpreting the Haskell report (Hudak, P., Peyton Jones, S. L. *et al.*, 1998) we can formulate a pattern matching predicate for this language. Matching can yield three different results: it may either succeed, fail or diverge. Furthermore, matching of arguments is performed left-to-right. We encode "success", "failure" and "divergence" by the three values T, F and $\bot$, and define the following $\mathcal{H}$ function.

$$
\begin{array}{rcl}
\mathcal{H}(\_, v) & = & T \\
\mathcal{H}(c(p_1,\ldots,p_a),\Omega) & = & \bot \\
\mathcal{H}(c(p_1,\ldots,p_a),c'(v_1,\ldots,v_{a'})) & = & F \\
\mathcal{H}(c(p_1,\ldots,p_a),c(v_1,\ldots,v_a)) & = & \mathcal{H}((p_1\cdots p_a),(v_1\cdots v_a)) \\
\mathcal{H}((p_1\ p_2\cdots p_n),(v_1\ v_2\cdots v_n)) & = & \mathcal{H}(p_1,v_1)\wedge_\bot \mathcal{H}((p_2\cdots p_n),(v_2\cdots v_n)) \\
\mathcal{H}((),()) & = & T
\end{array}
$$

Where the extended (left-to-right) boolean connectors are defined as:

$$
\begin{array}{rclcrcl}
T\wedge_\bot x & = & x & \quad & T\vee_\bot x & = & T \\
F\wedge_\bot x & = & F & & F\vee_\bot x & = & x \\
\bot\wedge_\bot x & = & \bot & & \bot\vee_\bot x & = & \bot
\end{array}
$$

We ignore some of Haskell patterns such as irrefutable patterns. We also ignore or-patterns at the moment. From this definition one easily shows the following two properties on vectors:

$$
\mathcal{H}(\vec{p},\vec{v}) = T \iff \vec{p}\preceq\vec{v} \qquad\qquad \mathcal{H}(\vec{p},\vec{v}) = F \Longrightarrow \vec{p}\,\#\,\vec{v} \Longrightarrow \vec{p}\not\preceq\vec{v}
$$

We then interpret the many program equivalences of section 3.17.3 in the Haskell report as expressing a downward search for a pattern of which $\vec{v}$ is an instance of:

$$
\mathcal{H}(\emptyset,\vec{v}) = F, \qquad\qquad \mathcal{H}(P,\vec{v}) = \mathcal{H}(\vec{p}^{\,1},\vec{v})\vee_\bot \mathcal{H}(P^{[2\ldots m]},\vec{v})
$$

That is, $\mathcal{H}(P, \vec{v}) = T$ means "$\vec{v}$ is found to match some row in P in the Haskell way", $\mathcal{H}(P, \vec{v}) = F$ means "no row of P is found to be matched", and $\mathcal{H}(P, \vec{v}) = \bot$ means "$\vec{v}$ is not precise enough to make a clear decision".

We can now formulate the left-to-right Haskell way of pattern matching in our setting.

Definition 10 (*Haskell pattern matching*)
State $\mathcal{P}(P, \vec{v})$ to be $\mathcal{H}(P, \vec{v}) = F$ in the generic definition 8.

One easily checks that predicate $\mathcal{H}(P, \vec{v}) = F$ meets all basic assumptions and key properties. Additionally, to prove the correctness of $\mathcal{U}(P, \vec{q})$, we now temporarily assume that matrix P and vector $\vec{q}$ are destructured along their first column. This simple restriction leads to a mechanical proof of key properties (1) to (4).

4.4 Or-patterns in Haskell

As this work is partly dedicated to specific warnings for or-patterns, we wish to enrich Haskell matching with or-patterns. The $\mathcal{H}$ function is extended to consider or-patterns, sticking to left-to-right bias.

$$\mathcal{H}((p_1 \,|\, p_2), v) \quad = \quad \mathcal{H}(p_1, v) \vee_\bot \mathcal{H}(p_2, v)$$

Semantical consequences are non-negligible, since the equivalence $\mathcal{H}(\vec{p}, \vec{v}) = T \iff \vec{p} \preceq \vec{v}$ does not hold any more, as can be seen by considering $\mathcal{H}((\texttt{True} \,|\, \_), \Omega) = \bot$. By contrast, most properties of the instance relation still hold for the relation $\mathcal{H}(\vec{p}, \vec{v}) = T$; in particular, monotonicity and the existence of at least one value $\vec{v}$ such that $\mathcal{H}(\vec{p}, \vec{v}) = T$.

However, as an important consequence, the definition of extended Haskell pattern matching must change: it does not fit the pattern of the generic definition anymore.

Definition 11 (*Haskell matching with or-patterns*)
Let P be a pattern matrix and $\vec{v}$ be a value vector. Vector v matches row i in P, if and only if the following proposition hold:

$$\mathcal{H}(P^{[1\ldots i)}, \vec{v}) = F \qquad \wedge \qquad \mathcal{H}(\vec{p}^{\,i}, \vec{v}) = T$$

It is not difficult to show that the inductive computation of $\mathcal{U}$ is still correct with the new definition of Haskell matching. Only inductive case 3 (on or-patterns) deserves a specific proof. The previous proof of inductive case 3 stemmed from the semantical property:

$$\mathcal{M}(P, ((r_1 \,|\, r_2)\ q_2 \cdots q_n)) = \mathcal{M}(P, (r_1\ q_2 \cdots q_n)) \cup \mathcal{M}(P, (r_2\ q_2 \cdots q_n))$$

A property that is no longer valid with the new definition of Haskell matching.

Fortunately, our interest is in deciding whether $\mathcal{M}(P, \vec{q})$ is empty or not. To that aim, we first get rid of the annoying case $\mathcal{H}(p, v) = \bot$.

Lemma 1
Let p be a pattern and v be a value, such that $\mathcal{H}(p, v) = \bot$. There exists value w, such that $v \preceq_\Omega w$ and $\mathcal{H}(p, w) \neq \bot$

Proof
By induction on p.

- Case $p = \_$ is excluded, since $\mathcal{H}(\_, w) = T$ for all values w.
- If $p = c(p_1, \ldots, p_a)$, then, by hypothesis $\mathcal{H}(p, v) = \bot$, we have two subcases.
  - If $v = c(v_1, \ldots, v_a)$ with $\mathcal{H}(p_i, v_i) = \bot$ for i in some (non-empty) index set I. Applying induction hypothesis to all such i yields values v'_i such that $\mathcal{H}(p_i, v'_i) \neq \bot$. Then we state $w = c(w_1, \ldots, w_a)$ where $w_i = v'_i$ for $i \in I$, and $w_i = v_i$ otherwise. Now $\mathcal{H}(p_i, w_i) \neq \bot$ for all i, and thus we have $\mathcal{H}(c(p_1, \ldots, p_a), w) = \mathcal{H}(\vec{p}, \vec{w}) \neq \bot$.
  - Otherwise $v = \Omega$, then construct $w = c(w_1, \ldots, w_a)$ as in the previous case, taking $v_1, \ldots, v_a$ to be Ω.
- If $p = (q_1 \,|\, q_2)$, we have two subcases.
  - If $\mathcal{H}(q_1, v) = F$ and $\mathcal{H}(q_2, v) = \bot$, then (by induction) there exists value w such that $\mathcal{H}(q_2, w) \neq \bot$ and, by definition of $\mathcal{H}$, we have $\mathcal{H}((q_1 \,|\, q_2), w) = \mathcal{H}(q_2, w)$.
  - If $\mathcal{H}(q_1, v) = \bot$, then (by induction) there exists value w, such that $v \preceq_\Omega w$ and $\mathcal{H}(q_1, w) \neq \bot$ If $\mathcal{H}(q_1, w) = T$, then we conclude $\mathcal{H}((q_1 \,|\, q_2), w) = T$. Otherwise, $\mathcal{H}(q_1, w) = F$ and thus $\mathcal{H}((q_1 \,|\, q_2) = \mathcal{H}(q_2, w)$. And we conclude either directly or by induction in the case where $\mathcal{H}(q_2, w) = \bot$.

□

We can now terminate the proof of algorithm $\mathcal{U}$.

Lemma 2
For definition 11 of extended Haskell pattern matching, we have:

$$\mathcal{U}(P, ((r_1 \,|\, r_2)\ q_2 \cdots q_n)) = \mathcal{U}(P, (r_1\ q_2 \cdots q_n)) \vee \mathcal{U}(P, (r_2\ q_2 \cdots q_n))$$

Proof
Let us assume that $\mathcal{U}(P, ((r_1 \,|\, r_2)\ q_2 \cdots q_n))$ holds and take $\vec{v}$ in the set of matching values $\mathcal{M}(P, ((r_1 \,|\, r_2)\ q_2 \cdots q_n))$. By hypothesis we have $\mathcal{H}(P, \vec{v}) = F$ and $\mathcal{H}((q_2 \cdots q_n)) = T$. Furthermore at least one of $\mathcal{H}(r_1, v_1) = T$ or $\mathcal{H}(r_2, v_1) = T$ holds. We can conclude from definitions. Conversely, any $\vec{v}$ in $\mathcal{M}(P, (r_1\ q_2 \cdots q_n))$, $\vec{v}$ is an element of $\mathcal{M}(P, ((r_1 \,|\, r_2)\ q_2 \cdots q_n))$. Thus, consider $\vec{v}$ in $\mathcal{M}(P, (r_2\ q_2 \cdots q_n))$ and further assume that $\vec{v} \notin \mathcal{M}(P, (r_1\ q_2 \cdots q_n))$. Then, we have $\mathcal{H}(r_2, v_2) = T$, $\mathcal{H}((q_2 \cdots q_n), (v_2 \cdots v_n)) = T$, and we further either have $\mathcal{H}(r_1, v_1) = F$ or $\mathcal{H}(r_1, v_1) = \bot$. In the former case, we conclude immediately, since $\vec{v}$ belongs to $\mathcal{M}(P, ((r_1 \,|\, r_2)\ q_2 \cdots q_n))$. In the latter case, by lemma 1 there exists w such that $v_1 \preceq_\Omega w$ and $\mathcal{H}(r_1, w) \neq \bot$, hence we have $\mathcal{H}((r_1 \,|\, r_2), w) = T$. Let us state $\vec{w} = (w\, v_2 \cdots v_n)$, we have $\mathcal{H}(P, \vec{w}) = F$ (by monotonicity) and $\mathcal{H}(((r_1 \,|\, r_2)\, q_2 \cdots q_n), \vec{w}) = T$ (by $\mathcal{H}((r_1 \,|\, r_2), w) = T$ and $H((q_2 \cdots q_n), (v_2 \cdots v_n)) = T$). □

It is time to clearly stress on some important consequence of the correctness of the algorithm of section 3.1 for computing $\mathcal{U}$, with respect to our three semantics for pattern matching. If we prove some sufficiently abstract property of $\mathcal{U}$, by using one of our semantics, then the property automatically holds for the other semantics.

We now take advantage of this trick. We used first column in the description of algorithm $\mathcal{U}$, informally arguing that "any other column would work". This informal remark is obviously true for strict pattern matching, since the definition of the predicates $P \npreceq \vec{v}$ and $\vec{q} \preceq \vec{v}$ are independent from column order.

Lemma 3 (*Irrelevance of column order*)
Let P be a pattern matrix and $\vec{q}$ be a pattern vector. By permuting the same columns in both P and $\vec{q}$ we get matrix P' and vector $\vec{q}\,'$. Then we have $\mathcal{U}(P, \vec{q}) = \mathcal{U}(P', \vec{q}\,')$.

Proof
As already noticed, the property obviously holds for strict matching. Then, consider any of the other two semantics, temporarily writing $\mathcal{U}_{\mathcal{P}}(P, \vec{q})$ for the associated utility predicate and $\mathcal{U}_{\npreceq}(P, \vec{q})$ for the strict utility predicate. For any matrix P and vector $\vec{q}$, we have $\mathcal{U}_{\npreceq}(P, \vec{q}) = \mathcal{U}_{\mathcal{P}}(P, \vec{q})$. Thus, we trivially have:

$$\mathcal{U}_{\mathcal{P}}(P, \vec{q}) = \mathcal{U}_{\npreceq}(P, \vec{q}) = \mathcal{U}_{\npreceq}(P', \vec{q}\,') = \mathcal{U}_{\mathcal{P}}(P', \vec{q}\,')$$

□

This "semantical" proof technique is quite powerful and we believe it to be quite elegant. Proving the irrelevance of column order more directly by induction on matrix and pattern structure would be quite cumbersome. Basically, this simplicity owes much to our focus on determining whether sets of matching values are empty or not, avoiding to describes those sets precisely. More generally, we enjoy the benefits of considering pattern matching anomalies directly, not as a by-product of compilation. As a matter of fact, observe that, with Haskell semantics, the set of matching values depends on column order.

PART TWO
Implementation

5 Specializing $\mathcal{U}$ for exhaustiveness check

Programmers sometimes are quite upset in front of "non-exhaustive match" warnings, especially when their matchings are complex. An example of a "non-matching value" helps a lot not only in convincing them that they indeed wrote a non-exhaustive matching, but also in correcting the matching.

Such an example (or counter-example) is best expressed as a pattern representing a set of non-matching instances. Then, programmers can add this "counter-example" at the end of their matchings, hoping this will make it exhaustive. To consider an easy example:

```
let nilp = function [] -> true
Warning: this pattern-matching is not exhaustive.
Here is an example of a non-matching value:
_::_
```

The given pattern matching is not exhaustive and all instances of the pattern `_::_` (a list cell) are non-matching. Here, by adding a clause with pattern `_::_` one achieves exhaustive matching.

Examples of non-matching values are easily computed by a slight extension of algorithm $\mathcal{U}$. Indeed, algorithm $\mathcal{U}$ shows that $\mathcal{M}(P, \vec{q})$ is not empty by implicitly computing a witness of that fact.

The new algorithm $\mathcal{I}$ takes a matrix P and an integer n as arguments, since $\mathcal{I}$ is used in a context where the pattern vector $\vec{q}$ of section 3.1 is a vector of n wildcards. Algorithm $\mathcal{I}$ returns a pattern vector $\vec{p}$ of size n such that all the instances of $\vec{p}$ are non-matching values. Or, if no such vector exists (*i.e.* if P is exhaustive), $\mathcal{I}$ returns the distinguished constant $\perp$.

Base case If $n = 0$, then we state:

$$\mathcal{I}(\begin{pmatrix} \end{pmatrix}, 0) = \perp \qquad\qquad \mathcal{I}(\emptyset, 0) = ()$$

More generally, one can observe that $\mathcal{I}(\emptyset, n)$ is a vector consisting of n wildcards.

Induction If $n > 0$, we define Σ as the set of constructors appearing at the root of the patterns (and of or-pattern alternatives) in the first column of P, as we did in section 3.1.

1. We first assume that Σ is a complete constructor signature. Then, we should perform the recursive calls $\mathcal{I}(\mathcal{S}(c_k, P), a_k + n - 1)$, for c_k taken from Σ.
   If *all* those computations return $\perp$, then $\mathcal{I}(P, n)$ also is $\perp$. Otherwise, if one of the calls $\mathcal{I}(\mathcal{S}(c_k, P), a_k + n - 1)$ returns pattern vector $(r_1 \cdots r_{a_k}\ p_2 \cdots p_n)$, we can state $\mathcal{I}(P, n) = (c_k(r_1, \ldots, r_{a_k})\ p_2 \cdots p_n)$. Of course, in practice, we stop performing recursive calls as soon as one such call is discovered. As there can be others $c_{k'}$ such that $\mathcal{I}(\mathcal{S}(c_{k'}, P), \ldots)$ returns a pattern vector, algorithm $\mathcal{I}$ is non-deterministic.
2. If Σ is not a complete constructor signature, we perform the only recursive call $\mathcal{I}(\mathcal{D}(P), n - 1)$ and we state $\mathcal{I}(P, n) = \perp$ if the recursive call returns $\perp$. Otherwise, $\mathcal{I}(\mathcal{D}(P), n - 1)$ returns the vector $(p_2 \cdots p_n)$, and the result of $\mathcal{I}(P, n)$ depends on whether Σ is empty or not. If Σ is empty, then we state $\mathcal{I}(P, n) = (\_\ p_2 \cdots p_n)$. If Σ is not empty, then we state $\mathcal{I}(P, n) = (c(\_, \ldots, \_)\ p_2 \cdots p_n)$, where c is a constructor from the signature of the c_k's without being a c_k. If the the signature of the c_k's is finite and not too big one can even use a or-pattern that regroups all the extra constructors.

It should be clear that $\mathcal{I}(P, n) = \perp$, if and only if P is exhaustive (*i.e.* $\mathcal{U}(P, (\_ \cdots \_))$ does not hold). Otherwise, $\mathcal{I}(P, n)$ is some pattern vector $\vec{p}$ and all the instances of $\vec{p}$ are non-matching values.

A simple example will demonstrate algorithm $\mathcal{I}$ at work. Let us check the exhaustiveness of the following matching that acts on values of type `mylist` (Section 2):

```
match ... with  One 1 -> ···
```

We thus compute $\mathcal{I}((\texttt{One 1}), 1)$.

$$\begin{array}{rcll} \mathcal{I}(\emptyset, 0) & = & () & \text{By } \textbf{base}\text{-2.} \\ \mathcal{I}((\texttt{ One 1 }), 1) & = & \texttt{(Nil|Cons (\_, \_))} & \text{By } \textbf{induction}\text{-2}, \Sigma = \{\texttt{One}\} \end{array}$$

In the case of non-exhaustive matchings, one may think that algorithm $\mathcal{I}$ should make an additional effort to provide more non-matching values, by systematically computing recursive calls on specialized matrices when possible, and by returning a list of all pattern vectors returned by recursive calls. We can first observe that it is not possible in general to supply the users with all non-matching values, since the signature of integers is (potentially) infinite. Furthermore, we claim that supplying one of the non-matching patterns is enough. Correcting the source that triggers the warning is a programmer's job, and we intentionally limit the task of the compiler to supplying a precise (and not too costly) warning, justified by a concrete example. In our example, the answer `(Nil|Cons (_,_))` points out the most obvious forgotten pattern. Furthermore, if the programmers write a clause for the flagged pattern and recompiles the "corrected" program, then the compiler will flag other non-matching patterns such as `One 0`. Hence, the whole information is thus available to programmers, if they want it.

6 Specializing $\mathcal{U}$ for the useless clause problem

At first sight, it seems that plain algorithm $\mathcal{U}$ suffices in flagging useless clauses. Indeed, one hardly sees what additional concise and useful information could be given to programmers, whose expected reaction is to suppress the useless clause before recompiling. However or-patterns introduce their specific anomaly, which is related to the useless clause anomaly but does not reduce to it.

6.1 Useless clause is (almost) enough

Let us assume that we write a function to detect lists of type `mylist` (Section 2) whose first element is `1`.

```
let f = function
| One 1 | Cons (1,_) -> true
| _ -> false
```

Looking back, we see that we are not protected against a future extension of type `mylist`. We rewrite:

```
let f = function
| One 1 | Cons (1,_) -> true
| Nil | One _ | Cons (_,_) -> false
```

(Now, if the definition of type `mylist` is changed by supplying an additional constructor, recompilation will produce a "non-exhaustive matching" warning.)

Later, we discover that variables can occur inside or-patterns. We rewrite:

```
let f = function
| One x | Cons (x,_) -> x=1
| Nil | One _ | Cons (_,_) -> false
```

Intuitively, something is wrong: the last pattern looks too complicated. Indeed the following code is more concise and equivalent:

```
let f = function
| One x | Cons (x,_) -> x=1
| Nil  -> false
```

Unfortunately, algorithm $\mathcal{U}$ silently accepts the first, "*bad*", code. A good compiler should suggest that we might replace this bad code by the second, "*good*", code.

In fact, algorithm $\mathcal{U}$ already does such a suggestion in the case of the following, "*bad*", code, where the or-pattern is expanded:

```
let f = function
| One x | Cons (x,_) -> x=1
| Nil -> false
| One _ -> false | Cons (_,_) -> false
```

Here, the compiler can tell us that the last two clauses are useless and we normally react by deleting them.

The discussion shows what is wrong in example

```
let f = function
| One x | Cons (x,_) -> x=1
| Nil | One _ | Cons (_,_) -> false
```

Clause `Nil | One _ | Cons (_,_) -> false` is useful because of pattern `Nil`. However, patterns `One _` and `Cons (_,_)` are useless, since they can be deleted without altering `f` behavior. Moreover, a more positive definition of useless pattern is easily built upon the standard notion of useless clause by expanding or-patterns. On the practical side the Objective Caml compiler will here flag two *useless patterns* (and no useless clause):

```
let f = function
| One x | Cons (x,_) -> x=1
| Nil | One _ | Cons (_,_) -> false
Warning: this pattern is unused.
Warning: this pattern is unused.
```

6.2 Expansion of or-patterns

Because there can be many or-patterns, it is our interest to consider the expansion of exactly one or-pattern amongst many, so that to avoid producing code of exponential size. Consider:

```
let f = function
| (1|2), (3|4), (5|6), ..., (2k−1|2k) -> true
| _ -> false
```

To check the arguments of or-pattern $(2k-1|2k)$, one expansion suffices:

```
let f = function
| (1|2), (3|4), (5|6), ..., 2k−1 -> true
| (1|2), (3|4), (5|6), ..., 2k   -> true
| _ -> false
```

And we can safely assert that patterns $2k-1$ and $2k$ are useful by using known algorithm $\mathcal{U}$.

Expansion considers that, in or-pattern $(p_1 \,|\, p_2)$, the left alternative p_1 has a higher priority than the right alternative p_2. This left-to-right bias allows a clear decision in the following boundary examples.

```
let f1 = function (1|_) -> true
and f2 = function (_|1) -> true
and f3 = function (1|1) -> true
and f4 = function (_|_) -> true
```

Expansion shows in what sense the right alternative of or-patterns is useful for `f1` and useless in the remaining cases.

```
let f1 = function 1 -> true | _ -> true
and f2 = function _ -> true | 1 -> true
and f3 = function 1 -> true | 1 -> true
and f4 = function _ -> true | _ -> true
```

It may seem that giving good diagnostics forces us into reconsidering the definition of matching, which does not specify any order for trying to match or-pattern arguments (except for Haskell matching). In fact for strict and Laville's matching, we still can avoid specifying such an order: those definitions of pattern matching rely on what row is matched and not on how it is matched. However, in practice, for the sake of consistency between diagnostics and produced code (because of variables in or-patterns, execution can indeed reveal the matched alternative), the pattern matching compiler must take left-to-right order into account. This is easily done by the (strict) compiler of (Le Fessant & Maranget, 2001), which performs the expansion during pattern matching compilation, as by any compiler that features or-patterns by performing expansion before pattern matching compilation such as the SML/NJ compiler (Appel & MacQueen, 1991).

In the next section we describe a refinement of our algorithm $\mathcal{U}$. This refinement aims at finding useless patterns and relies on the expansion of or-patterns. As a preliminary, we first note that expansion does not alter the output of algorithm $\mathcal{U}$.

Lemma 4 (*Utility of expansion*)
For all three definitions of pattern matching, we have:

$$\mathcal{U}(P, ((r_1 \,|\, r_2)\ q_2 \cdots q_n)) = \mathcal{U}(P, (r_1\ q_2 \cdots q_n)) \vee \mathcal{U}(P @ (r_1\ q_2 \cdots q_n), (r_2\ q_2 \cdots q_n))$$

where $P @ \vec{p}$ means matrix P with row $\vec{p}$ added at the bottom.

Proof

For Haskell matching, the equality follows from definitions ($\mathcal{H}((r_1 \,|\, r_2), v) = T$, if and only if $\mathcal{H}(r_1, v) = T$ or $\mathcal{H}(r_1, v) = F \wedge \mathcal{H}(r_2, v) = T$). Hence, the equality also holds for the other two semantics of pattern matching. □

6.3 Rules for finding useless patterns

It is certainly easier to first consider finding useless sub-patterns in the case of one or-pattern. Let P be a pattern matrix and let $\vec{q}$ be a pattern vector, such that q_1 is the or-pattern $(r_1 \,|\, r_2)$. We wish to make a distinction between four possibilities, r_1 and r_2 are both useful, r_1 alone is useless, r_2 alone is useless, and both r_1 and r_2 are useless (and thus row $\vec{q}$ is useless). More concretely we design a new function $\mathcal{U}'(P, \vec{q})$ that returns a set of useless patterns, that is, $\emptyset$ in the first case, $\{r_1\}$ in the second case, $\{r_2\}$ in the third case, and the distinguished set $\top_{q_1}$ in the fourth case.

From P and $\vec{q}$ we define two *expanded* matchings:

$$P' = P, \quad \vec{q}\,' = (r_1\ q_2 \cdots q_n) \qquad \text{and} \qquad P'' = P @ \vec{q}\,', \quad \vec{q}\,'' = (r_2\ q_2 \cdots q_n)$$

Where $P@\vec{q}\,'$ means adding row $\vec{q}\,'$ to the bottom of matrix P. Then, we use $\mathcal{U}$ to compute the utility of both expansions and we write E_{r_1} and E_{r_2} for the results of these computations, logically encoding *True* by $\emptyset$ and *False* by $\top$ (E sets are sets of *useless* patterns). Then we combine E_{r_1} and E_{r_2} into E_{q_1} by the rules given in the left table of Figure 1.

Fig. 1. Rules for combining the utility of or-pattern arguments

E_{r_1}	E_{r_2}	$E_{(r_1 \,\|\, r_2)}$
$\emptyset$	$\emptyset$	$\emptyset$
$\top$	$\top$	$\top$
$\emptyset$	$\top$	$\{r_2\}$
$\top$	$\emptyset$	$\{r_1\}$

E_{r_1}	E_{r_2}	$E_{(r_1 \,\|\, r_2)}$
$\emptyset$	$\{r'', \ldots\}$	$\{r'', \ldots\}$
$\{r', \ldots\}$	$\emptyset$	$\{r', \ldots\}$
$\top$	$\{r'', \ldots\}$	$\{r_1, r'', \ldots\}$
$\{r', \ldots\}$	$\top$	$\{r', \ldots, r_2\}$
$\{r', \ldots\}$	$\{r'', \ldots\}$	$\{r', \ldots, r'' \ldots\}$

If r_1 and r_2 are themselves or-patterns, we would like to compute the utility of their arguments. To do so, $\mathcal{U}'$ is called recursively. As a consequence, results other than $\emptyset$ and $\top$ are possible, when r_1 or r_2 are partially useless. We combine those new results as described in the second table of figure 1. Those extra rules complete the definition of pattern utility by expansion. As an example of such nested expansions, assume $q_1 = ((r_1 \,|\, r_2) \,|\, (r_3 \,|\, r_4))$, the utility of r_4 is computed on the expansion consisting in matrix$P@((r_1 \,|\, r_2)\ q_2 \cdots q_n)@(r_3\ q_2 \cdots q_n)$ and vector $(r_4\ q_2 \cdots q_n)$.

If several components of $\vec{q}$ are or-patterns, we perform several independent expansions. For instance, let us assume that both q_1 and q_2 are or-patterns, we first proceed as described above, yielding one result E_{q_1}. Then, we expand P and $\vec{q}$ along their second column. Such an expansion can be defined easily as the composition of swapping the first two columns of P and $\vec{q}$ and of the expansion introduced at the beginning of this section. This process yields another result E_{q_2}.

We now need to combine the two results E_{q_1} and E_{q_2}. Let us first consider the case when neither E_{q_1} nor E_{q_2} is $\top$. Then, those two results are (possibly empty) sets of useless patterns which we combine by set union, yielding the new result $E_{q_1} \cup E_{q_2}$. It should perhaps be remarked that the patterns in E_{q_1} and E_{q_2} are always considered pairwise distinct, even when they happen to be structurally equal. Namely, given our objective of finding useless patterns, different sub-patterns of a given pattern always are distinct. In practice, patterns in E sets may be encoded as positions from the input source file. Let us now consider the case when E_{q_1} is $\top$. We assume, as we show later, that $\mathcal{U}'$ is a conservative extension of $\mathcal{U}$. That is $\mathcal{U}'(P, \vec{q}) = \top$, if and only if $\mathcal{U}(P, \vec{q}) = \mathit{False}$. Hence, E_{q_1} is $\top$ implies $\mathcal{U}(P, \vec{q})$ is *False* — *i.e.* $\vec{q}$ is useless w.r.t. P. However, swapping two columns in P (and $\vec{q}$) does not change $\mathcal{U}$ result (lemma 3). Thus we also have $E_{q_2} = \top$. Conversely, if E_{q_2} is $\top$, then E_{q_1} necessarily is $\top$. Overall, whether expansion is performed along first or second column does not matter and the value of $\mathcal{U}'(P, \vec{q})$ should be $\top$. As a conclusion, we can state the combination of the utility of two disjoint or-patterns to be $E_{q_1} \cup E_{q_2}$, provided we adopt the extra definition $\top \cup \top = \top$.

6.4 Computation of useless patterns

We now give a precise description of algorithm $\mathcal{U}'$, as implemented in the Objective Caml compiler. The key idea is to use specialization ($\mathcal{S}(c, P)$ of Section 3.1) as a tool to discover or-patterns, and to apply expansions as in the previous section to check utility of arguments of or-patterns. For doing so, it is convenient to partition the *columns* of matrices and vectors into three subparts. We note those separations with "$\bullet$". That is, $\mathcal{U}'$ takes such "dotted" matrices and vectors as argument, written $P \bullet Q \bullet R$ and $\vec{p} \bullet \vec{q} \bullet \vec{r}$. Dotted matrices and vectors define matchings in the ordinary sense, provided we erase the dots. More precisely we concatenate the subparts column-wise, written "&", and consider $\mathcal{U}(P \,\&\, Q \,\&\, R, \vec{p} \,\&\, \vec{q} \,\&\, \vec{r})$. This new notation emphasizes the distinction between column-wise (or vertical) concatenation and row-wise (or horizontal) concatenation, which we write "@".

Figure 2 defines some useful operations on dotted matrices. It is assumed that sub-matrix P has n columns ($n > 0$). Informally, the first phase of algorithm $\mathcal{U}'$ destructures the patterns of $\vec{p}$ (using $\mathcal{S}$ from figure 2), in order to find or-patterns. When found the corresponding columns are transferred to the R subpart (using $\Rightarrow_2$), ready for the expansion phase. Other columns are transferred to the Q subpart (using $\Rightarrow_1$).

To compute the utility of clause number i in some matching

`match` ... `with` p_1 `->` e_1 `|` p_2 `->` e_2 `|` ... `|` p_m `->` e_m

Fig. 2. Operations on dotted matrices

(a) Specialization by constructor c

row in $P \bullet Q \bullet R$	row(s) in $\mathcal{S}(c, P \bullet Q \bullet R)$
$c(t_1, \ldots, t_a) \cdots p^i_n \bullet q^i_1 \cdots q^i_k \bullet r^i_1 \cdots r^i_z$	$t_1 \cdots t_a \; p^i_2 \cdots p^i_n \bullet q^i_1 \cdots q^i_k \bullet r^i_1 \cdots r^i_z$
$\_ \quad \cdots p^i_n \bullet q^i_1 \cdots q^i_k \bullet r^i_1 \cdots r^i_z$	$\_ \cdots \_ \; p^i_2 \cdots p^i_n \bullet q^i_1 \cdots q^i_k \bullet r^i_1 \cdots r^i_z$
$c'(t_1, \ldots, t_{a'}) \cdots p^i_n \bullet q^i_1 \cdots q^i_k \bullet r^i_1 \cdots r^i_z$	no row
$(t_1 \mid t_2) \quad \cdots p^i_n \bullet q^i_1 \cdots q^i_k \bullet r^i_1 \cdots r^i_z$	$\mathcal{S}\left(c, \begin{pmatrix} t_1 \cdots p^i_n \bullet q^i_1 \cdots q^i_k \bullet r^i_1 \cdots r^i_z \\ t_2 \cdots p^i_n \bullet q^i_1 \cdots q^i_k \bullet r^i_1 \cdots r^i_z \end{pmatrix}\right)$

(b) Right shifts

row in $P \bullet Q \bullet R$	row in $\Rightarrow_1(P \bullet Q \bullet R)$	row in $\Rightarrow_2(P \bullet Q \bullet R)$
$p^i_1 \cdots p^i_n \bullet q^i_1 \cdots q^i_k \bullet r^i_1 \cdots r^i_z$	$p^i_2 \cdots p^i_n \bullet p^i_1 \; q^i_1 \cdots q^i_k \bullet r^i_1 \cdots r^i_z$	$p^i_2 \cdots p^i_n \bullet q^i_1 \cdots q^i_k \bullet p^i_1 \; r^i_1 \cdots r^i_z$

we consider the initial call

$$\mathcal{U}'(\begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_{i-1} \end{pmatrix} \bullet \; \bullet \;, (p_i) \bullet \; \bullet \;)$$

The typical call $\mathcal{U}'(P \bullet Q \bullet R, \vec{p} \bullet \vec{q} \bullet \vec{r})$ yields four situations. First three situations are the first "search for or-patterns" phase and apply when P has columns.

1. If p_1 is a constructed pattern $c(t_1, \ldots, t_a)$, we state:

$$\begin{aligned} &\mathcal{U}'(P \bullet Q \bullet R, (c(t_1, \ldots, t_a) \; p_2 \cdots p_n) \bullet \vec{q} \bullet \vec{r}) = \\ &\qquad \mathcal{U}'(\mathcal{S}(c, P \bullet Q \bullet R), (t_1 \cdots t_a \; p_2 \cdots p_n) \bullet \vec{q} \bullet \vec{r}) \end{aligned}$$

2. If p_1 is a wildcard, we transfer P first column into Q:

$$\mathcal{U}'(P \bullet Q \bullet R, (\_ \; p_2 \cdots p_n) \bullet \vec{q} \bullet \vec{r}) = \mathcal{U}'(\Rightarrow_1(P \bullet Q \bullet R), \Rightarrow_1(\vec{p} \bullet \vec{q} \bullet \vec{q}))$$

3. If p_1 is a or-pattern, we transfer P first column into R:

$$\mathcal{U}'(P \bullet Q \bullet R, ((t_1 \mid t_2) \; p_2 \cdots p_n) \bullet \vec{q} \bullet \vec{r}) = \mathcal{U}'(\Rightarrow_2(P \bullet Q \bullet R), \Rightarrow_2(\vec{p} \bullet \vec{q} \bullet \vec{q}))$$

4. If P has no columns, there are two subcases depending on whether there are columns in R or not.
   (a) If no or-patterns were found in initial $\vec{p}$ ($\vec{r} = ()$), we just call $\mathcal{U}$.

$$\begin{aligned} \mathcal{U}'(\; \bullet Q \bullet \;, \; \bullet \vec{q} \bullet \;) &= \emptyset \quad \text{if } \mathcal{U}(Q, \vec{q}) = \mathit{True} \\ \mathcal{U}'(\; \bullet Q \bullet \;, \; \bullet \vec{q} \bullet \;) &= \top \quad \text{if } \mathcal{U}(Q, \vec{q}) = \mathit{False} \end{aligned}$$

   (b) Otherwise R and $\vec{r}$ possess z columns ($z > 0$) and all the components of $\vec{r}$ are or-patterns. For a given column index j in $\vec{r}$, we write $\vec{r} \setminus j$ for the vector build from $\vec{r}$ by suppressing component r_j. Similarly we write $R \setminus j$

for matrix R with column j erased. Finally $[R]_j$ is the matrix R reduced to its column j. By hypothesis r_j is a or-pattern $(t_1 \mid t_2)$ and we perform the two recursive calls:

$$E_{t_1} = \mathcal{U}'([R]_j \bullet (R \setminus j) \,\&\, Q \bullet\ , (t_1) \bullet (\vec{r} \setminus j) \,\&\, \vec{q} \bullet\ )$$
$$E_{t_2} = \mathcal{U}'([R]_j @ (t_1) \bullet ((R \setminus j) \,\&\, Q) @ ((\vec{r} \setminus j) \,\&\, \vec{q}) \bullet\ , (t_2) \bullet (\vec{r} \setminus j) \,\&\, \vec{q} \bullet\ )$$

This formulas may be a bit complicated, they simply express the expansion of pattern r_j in a general setting (see the beginning of Section 6.3 for the particular case of exactly one or-pattern). One should notice that other columns j' of R and $\vec{r}$ (with $j' \neq j$) get transfered to subpart Q and will not be expanded by further calls to $\mathcal{U}'$. That is, expansion of or-patterns is performed only once.
Then, we combine E_{t_1} and E_{t_2} by the rules of Figure 1, yielding E_{r_j}. Finally, we state:

$$\mathcal{U}'(\ \bullet Q \bullet R,\ \bullet \vec{q} \bullet \vec{r}) = \bigcup_{j=1}^{z} E_{r_j}$$

We now prove that the new algorithm $\mathcal{U}'$ is a conservative extension of the original algorithm $\mathcal{U}$.

Proposition 2
Let P be a pattern matrix and $\vec{q}$ be a pattern vector. Then, $\mathcal{U}'(P \bullet \bullet, \vec{p} \bullet \bullet) = \top$ is equivalent to $\mathcal{U}(P, \vec{p}) = \mathit{False}$.

Proof
We prove the stronger property:

$$\mathcal{U}'(P \bullet Q \bullet R, \vec{p} \bullet \vec{q} \bullet \vec{r}) = \top \iff \mathcal{U}(P \,\&\, Q \,\&\, R, \vec{p} \,\&\, \vec{q} \,\&\, \vec{r}) = \mathit{False}$$

Proof is by induction on the definition of $\mathcal{U}'$. Most cases are obvious, case 4-(a) is the base case, inductive cases 2. and 3. follow from lemma 3 on the irrelevance of column order, while inductive case 1. follows from inductive case 1. of the definition of algorithm $\mathcal{U}$.

Case 4-(b) (P is empty, R is not empty) is the most interesting. We first consider one expansion. In order to simplify notations a bit, we state $T = Q \,\&\, R$ and $\vec{t} = \vec{q} \,\&\, \vec{r}$ and we express the expansion of $r_j = (o_1 \mid o_2)$ as follows:

$$E_{o_1} = \mathcal{U}'(P' \bullet Q' \bullet\ , (o_1) \bullet \vec{q}\,' \bullet\ ), \qquad E_{o_2} = \mathcal{U}'(P'' \bullet Q'' \bullet\ , (o_2) \bullet \vec{q}\,' \bullet\ )$$

Where it should be clear that P'' is $P' @ (o_1)$ and Q'' is $Q' @ \vec{q}\,'$. That is, $T'' = P'' \,\&\, Q''$ is $T' = P' \,\&\, Q'$ with the row $(o_1) \,\&\, \vec{q}\,'$ added. Furthermore, T' and $\vec{t}\,' = (o_1 \mid o_2) \,\&\, \vec{q}\,'$ are images of T and $\vec{t}$ by the same permutation of columns. Hence, by lemmas 3 and 4, we have:

$$\mathcal{U}(T, \vec{t}) = \mathcal{U}(T', \vec{t}\,') = \mathcal{U}(T', (o_1) \,\&\, \vec{q}\,') \vee \mathcal{U}(T' @ ((o_1) \,\&\, \vec{q}\,'), (o_2) \,\&\, \vec{q}\,')$$

By induction and since $E_{(o_1 \mid o_2)} = \top$, if and only if $E_{o_1} = \top$ and $E_{o_2} = \top$, we have:

$$E_{r_j} = \top \iff \mathcal{U}(T, \vec{t}) = \mathit{False}$$

And we can conclude, since $\mathcal{U}'(\ \bullet Q \bullet R,\ \bullet \vec{q} \bullet \vec{r}) = E_{r_1} \cup \cdots \cup E_{r_z}$. $\square$

One can make the computation of $\mathcal{U}'$ slightly more efficient by the following two techniques:

- At inductive step 2, if all patterns in the first column of P are wildcards, we delete this column in place of transferring it into Q. This will avoid repeated deletion by $\mathcal{U}$.
- At inductive step 4-(b), if E_{r_1} is $\top$, we can immediately return $\top$.

7 Performance

7.1 Elements of complexity analysis

Given some pattern matching expression of m patterns, algorithm $\mathcal{I}$ (Section 5) is called once and $\mathcal{U}'$ (previous section) is called m times with matrix arguments of 0, $\ldots$, $m-1$ rows. We can roughly assimilate $\mathcal{I}$ with $\mathcal{U}$. As regards $\mathcal{U}'$, after a first, linear phase, $\mathcal{U}$ is called once, if they are no or-patterns, or as many times as there are or-pattern alternatives. Hence, in the absence of or-patterns and making the very optimistic assumption of $\mathcal{U}$ being linear in its input size, we expect a quadratic behavior in the size of the input patterns, which we assimilate with source file size. Namely, S being input size, we perform at most S calls of $\mathcal{U}$ on arguments of size at least S.

Here, we do not claim to perform a thorough complexity analysis. Rather, we intend to define a reachable target for the running time of a practical implementation. As a matter of fact, even when or-patterns are not considered, the useful clause problem is NP-complete (Sekar *et al.*, 1992). It is thus no surprise that algorithm $\mathcal{U}$ can exhibit exponential time behavior in the size of its arguments P and $\vec{q}$. This exponential behaviour originates from inductive step 2-(a) (Section 3.1): the rows of matrix P whose first pattern are wildcards get copied into all specialized matrices $\mathcal{S}(c_k, P)$, and recursive calls on all these matrices may be performed. The most unfavorable situation occurs when all patterns in $\vec{q}$ are wildcards, patterns in P contains mostly wildcards and a few different constructor patterns (such that constructors collected column-wise are complete signatures), and when $\mathcal{U}(P, \vec{q})$ is false. And indeed, one can construct a series of matching that confirms the exponential time behaviour of a straightforward implementation of algorithm $\mathcal{U}$. Matching of increasing size generated randomly also confirm that the exponential time behaviour can occur.

7.2 Safeguards

Input to the Objective Caml compiler *a priori* is neither nasty nor random, it consists of programs written by human beings attempting to solve specific problems, not to break the compiler. If such reasonable input triggers exponential behavior, then the compilation of useful programs may perhaps become unfeasible.

We now search for means that may prevent $\mathcal{U}$ from reacting exponentially to reasonable input. We call such a mean a *safeguard*. An exhaustive matching is a potentially unfavorable situation, since we here compute $\mathcal{U}(P, (\_ \cdots \_)) = \mathit{False}$.

Hence, we examine how human beings write exhaustive matchings. An easy and frequent way to ensure exhaustiveness is to complete matchings with some wildcard pattern. Hence, we could first scan matrices searching for rows of wildcards, and announce exhaustive matching as soon as we discover one. However, this process would not be very general. Let us rather consider strict pattern matching and remark that for any strict value we have $\_ \preceq v$. That is, for any pattern p and strict value v, we have $p \preceq v \Longrightarrow \_ \preceq v$ (or $\_ \npreceq v \Longrightarrow p \npreceq v$). Hence, if some row of some matrix P is made of wildcards, then that row contains all the non-matching potential of the whole matrix P.

Wildcards can also appear inside patterns, as in the following example:

```
match v with | (1,2) -> 1 | (1,_) -> 2
```

Then, for all strict values v we have `(1,2)` $\preceq v \Longrightarrow$ `(1,_)` $\preceq v$ (or `(1,_)` $\npreceq v \Longrightarrow$ `(1,2)` $\npreceq v$). Thus, before we compute exhaustiveness, we can delete pattern `(1,2)` from the matching, because its presence does not add non-matching values w.r.t. the remaining pattern `(1,_)` .

We now consider the general case.

Definition 12
Let $\vec{p}$ and $\vec{q}$ be two pattern vectors. By definition, $\vec{p}$ *subsumes* $\vec{q}$ when, for all *strict* value vectors v, we have:

$$\vec{q} \preceq \vec{v} \Longrightarrow \vec{p} \preceq \vec{v}$$

Lemma 5
Let P be a pattern matrix. We further assume that there exist two rows of P, $\vec{p}^{\,i}$ and $\vec{p}^{\,i}$, such that $\vec{p}^{\,i}$ subsumes $\vec{p}^{\,j}$. And we write P' for P without row $\vec{p}^{\,j}$. Then, for any pattern vector $\vec{q}$ we have :

$$\mathcal{U}(P, \vec{q}) = \mathcal{U}(P', \vec{q})$$

Proof
By definition of "$\npreceq$" we have

$$P \npreceq \vec{v} \iff P' \npreceq \vec{v} \wedge \vec{p}^{\,j} \npreceq \vec{v}$$

Furthermore by definition of $P' \npreceq \vec{v}$ and by hypothesis "some row in P' subsumes $\vec{p}^{\,j}$", we have:

$$P' \npreceq v \Longrightarrow \vec{p}^{\,i} \npreceq \vec{v} \Longrightarrow \vec{p}^{\,j} \npreceq \vec{v}$$

Finally, for any strict value $\vec{v}$ we have:

$$P' \npreceq \vec{v} \iff P \npreceq \vec{v}$$

□

Example 4
Let us consider the following matrix P.

$$P = \begin{pmatrix} \texttt{\_} & \texttt{(\_,\_)} \\ \texttt{(\_,\_)} & \texttt{\_} \end{pmatrix}$$

Because the constructor of pairs "`,`" is alone in its signature, pattern `(_,_)` subsumes pattern `_`. Namely, any (strict) value v that possesses the type $t_1 \times t_2$ of a pair can be written $v = (v_1, v_2)$ and is thus an instance of `(_,_)`[3]. As a consequence, erasing any of the two rows of P does not matter and we have:

$$\mathcal{U}(P, \vec{q}) = \mathcal{U}((\ \_ \quad (\_,\_)\ ), \vec{q}) = \mathcal{U}((\ (\_,\_) \quad \_\ ), \vec{q})$$

It turns out that we can decide the relation "$\vec{p}$ subsumes $\vec{q}$" by using predicate $\mathcal{U}$.

Lemma 6
Let $\vec{p}$ and $\vec{q}$ be two pattern vectors. Then $\vec{p}$ subsumes $\vec{q}$, if and only if $\mathcal{U}(\vec{p}, \vec{q})$ does not hold.

Proof
The property is obvious by considering the strict semantics of pattern matching.

$$\mathcal{U}(\vec{p}, \vec{q}) = \mathit{False} \iff (\forall \vec{v}, \vec{p} \preceq \vec{v} \vee \vec{q} \not\preceq \vec{v}) \iff (\forall \vec{v}, \vec{q} \preceq \vec{v} \Longrightarrow \vec{p} \preceq \vec{v})$$

Notice that $\forall \vec{v}$ here means "for all strict values". □

By lemma 5 and previous lemma 6 we now have a mean to delete rows from matrix P before we compute $\mathcal{U}$. And this mean is valid for all our semantics of pattern matching. More precisely, we have:

$$\mathcal{U}(\vec{p}^{\,i}, \vec{p}^{\,j}) = \mathit{False} \Longrightarrow \mathcal{U}(P, \vec{q}) = \mathcal{U}(P', \vec{q})$$

(Where P' is P without row number i.)

In practice, before computing $\mathcal{U}(P, \vec{q})$, our first safeguard consists in deleting any row of P that is subsumed by another row of P. In the worst case, (when no row is subsumed) this requires computing $\mathcal{U}(\vec{p}^{\,i}, \vec{p}^{\,j})$ for all pairs of (distinct) rows in P. However, in the absence of or-patterns, computing $\mathcal{U}(\vec{p}^{\,i}, \vec{p}^{\,j})$ cannot be exponential (since bad case 2-(a) may only occur for constructor signatures of size one). We can thus approximate its cost as a quadratic number of "ordinary" pattern operations whose cost is roughly linear in the size of input patterns (in the absence of or-patterns). As a conclusion, if this first safeguard saves us from an exponential computation, it is worth its price.

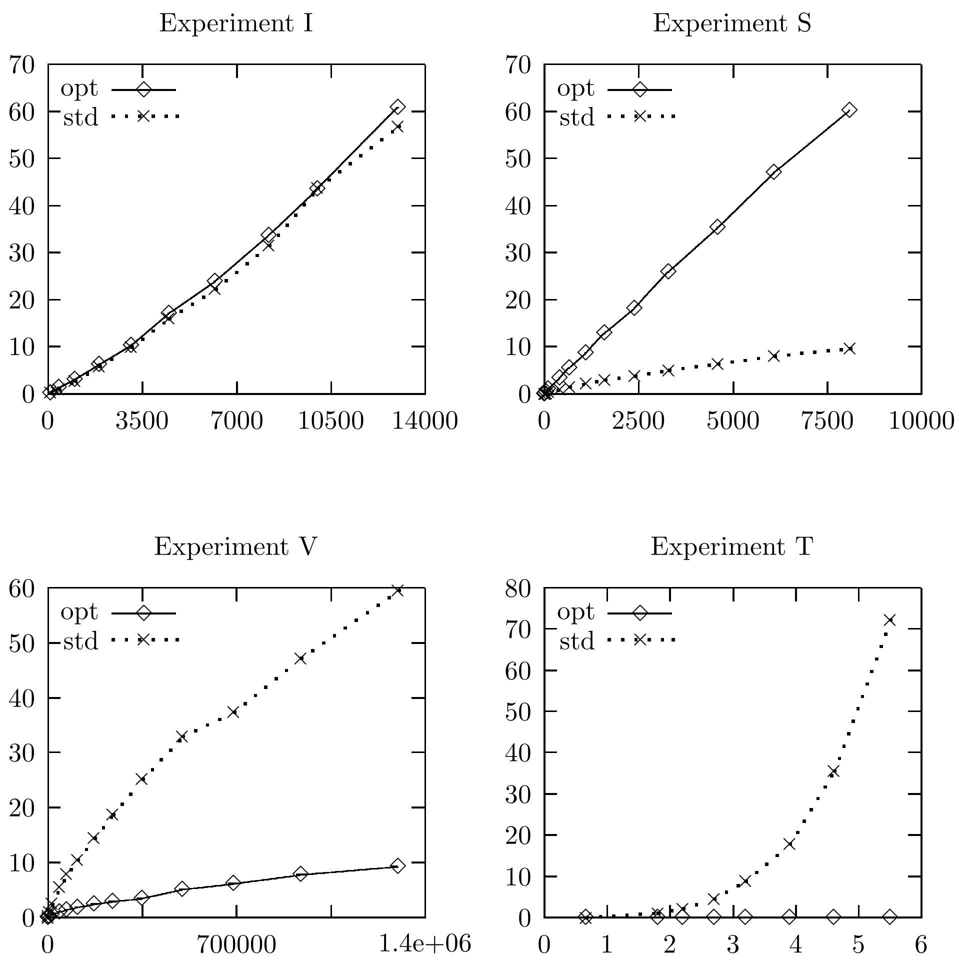
We now design a second, less expensive, safeguard. Two patterns are said to be incompatible when they have no instance in common. Thus, before computing $\mathcal{U}(P, \vec{q})$ we can delete some row $\vec{p}^{\,i}$ from P, provided $\vec{p}^{\,i}$ and $\vec{q}$ are incompatible (consider strict matching). It turns out that incompatibility of patterns can be computed as an "ordinary" pattern operation by using the rules for computing the incompatibility of pattern and value (*c.f.* Section 4.2). Thus, we can attempt to simplify matrix P with m rows for the price of m ordinary patterns operations. Unfortunately, this process does not help while checking exhaustiveness, since $(\_ \cdots \_)$ is compatible with any pattern vector. Furthermore, the first phase of computing $\mathcal{U}'(P, \vec{q})$ for checking utility in fact gets rid of rows that are incompatible with

[3] It is worth noticing that all instances of `_` no longer are instances of `(_,_)` when partial values are considered. Namely, the undefined value Ω is not an instance of `(_,_)`.

checked row $\vec{q}$ (at step 1.). However, the price of this second safeguard is low and it may be a good idea to perform it before performing the first safeguard. As a result, we may reduce the size of input to the first safeguard. Of course, this makes sense because the first safeguard performs a quadratic number of ordinary pattern operations while the second safeguard performs a linear number of ordinary pattern operations.

7.3 Measures

Fig. 3. Effect of safeguards in four experiments. User CPU time (in seconds) as a function of file size squared (in kilobytes^2)



We performed some measures of the (user) machine time taken by our implementation, which is integrated in the Objective Caml compiler. We measure machine time both with and without pattern-matching diagnostics enabled. Then, a simple subtraction yields the time taken by the detection of anomalies.

Measures apply to series of pattern-matching expressions of identical structure and increasing size. Three of our series T, S and V, may drive compilation of pattern matching to decision trees into producing code of exponential size. We selected those because algorithm $\mathcal{U}$ is similar to compilation of pattern matching to decision trees. The fourth I series is the matching of n constant constructors. We selected series I because of a real example that triggered important compilation times while we were implementing algorithm $\mathcal{U}$. All series are defined precisely in appendix A. In Figure 3, the X-axis expresses the square of input file size (in kilobytes^2) while the Y-axis express (user) machine time on a Linux 1Ghz PC. We perform measures both with and without the safeguards described in the previous section (those are named "opt" and "std").

Experiments S and V demonstrate that algorithm $\mathcal{U}$ is more resistant to exponential behavior than simple compilation to decision trees. Namely, exponential behavior of compilation to decision trees have been reported for those examples (Sestoft, 1996; Maranget, 1992). Observe that algorithm $\mathcal{U}$ does not exhibit exponential running time even without safeguards (written "std" in Figure 3).

However, algorithm $\mathcal{U}$ is not immune from exponential behavior, as demonstrated by experiment T. In that particular case, safeguards prove efficient and we decided to retain them in our implementation for that reason. Adopting safeguards incurs some penalty, since safeguards significantly degrade performance in experiment S. However, it should be noticed that it requires a 160×160 matrix to reach one minute of analysis time in experiment S (with safeguards), whereas it takes only a 21×21 matrix to reach a similar time in experiment T (without safeguards).

Finally the plots of figure 3 suggest that the running time of pattern matching analysis with safeguards is approximatively quadratic in input size, at least for our examples and on the studied domain. However, notice that experiment I suggest a slightly more than quadratic running time.

We also estimate the time taken by our algorithm relatively to total compilation time of ordinary programs. To do so, we measure the running time of the compilation of some programs, with pattern matching analysis disabled (NO), with pattern matching analysis enabled but without safeguards (STD), and with safeguards (OPT). We made some effort not to measure irrelevant operations such as linking and the application of external tools. We made ten measures in each experiment and we computed geometric means. The compiled programs are the Objective Caml compiler itself (`ocamlc`), the `hevea` LaTeX to HTML translator and another compiler (`zyva`). All those programs use pattern matching significantly and are of respectable size.

	NO	STD	OPT
`ocamlc`	13.63	13.92	14.02
`hevea`	7.84	8.01	8.05
`zyva`	4.61	4.70	4.74

Those results show that the price of pattern-matching analysis remains quite low in practice. One might even conclude that safeguards are not very useful when it comes to compiling "actual" programs. Still, we adopt safeguards because they are designed to avoid exponential behavior in some specific situations, which do not show up in the above examples.

8 Conclusion

To the best of our knowledge there is no literature on the analysis of pattern matching. the works closest to ours are (Sekar *et al.*, 1992; Maranget, 1992), which give some algorithms for computing directions (also call indices). Those algorithms are in fact very similar to our algorithm $\mathcal{U}$. And our present work can be seen as a thorough exploration of how to apply the initial algorithm to pattern matching diagnostics, while directions are confined to the compilation of Laville's semantics.

Works on the compilation of pattern matching to decision trees (see for instance (Baudinet & MacQueen, 1985; Aitken, 1992; Sestoft, 1996)) sometimes recall that this compilation technique yields "non-exhaustive" and "useless-clause" warnings as by-products. Our approach is rather different: we consider diagnostics independantly of compilation. As a consequence, our algorithm for producing diagnostics is not only independent from any compilation strategy, but is also proved correct with respect to three pattern matching semantics.

We now examine how a few compilers perform pattern matching diagnostics. The SML/NJ compiler (Appel & MacQueen, 1991) compiles pattern matching by following the decision tree approach, thus naturally producing diagnostics. However, it flags "useless patterns" as "useless clauses", which can confuse programmers. It should be noticed that for the sake of precise warnings for "useless patterns" (*i.e.* useless arguments in or-patterns), we defined some partial expansion of or-patterns, whereas compilation calls for a complete expansion. From our understanding of its code and architecture, the SML/NJ compiler performs such a complete expansion before the compilation of pattern matching, and then ignores the existence of or-patterns. Additionally, the SML/NJ takes exponential time (and produce code of exponential size) in experiments V and S.

The Glasgow Haskell compiler `ghc` (Peyton Jones *et al.*, 1993) also carries out pattern matching checks. Surprisingly, by default, the "useless clause" diagnostic is enabled while the "non-exhaustive match" diagnostic is disabled. As `ghc` compiles pattern matching to backtracking automata (Wadler, 1987; Augustsson, 1985), the source of the compiler contains specific code for producing pattern matching diagnostics. This code apparently proceeds exactly along the lines of compilation to decision trees. Of course, no tree is produced, instead the analyzer computes the leaves of the tree. This approach results in producing several examples of non-matching values in the case of non-exhaustive matchings. Unfortunately, it also results in exponential running times and excessive memory consumption in examples T, V and S.

Those comparisons demonstrate that our algorithm, though inspired by compilation to decision trees, is more efficient. First, our algorithm does not build any tree data-structure of potentially exponential size. More significantly, the compilation scheme can be seen as the search of all matching values, while, our algorithm only searches for one matching value. The benefits of this approach are numerous: the search is stopped as soon as one matching value is found; in some important and frequent case (inductive step 2-(b) in Section 3.1) our algorithm performs one recursive call, where compilation performs two or more; and finally, some simplification on the input patterns that we perform are inappropriate to compilation.

In our opinion, pattern matching analysis has been somehow neglected. For instance, although the Definition of Standard ML (Harper *et al.*, 1991) requires compliant implementation to flag "redundant" and "non-exhaustive" matches, the Haskell Report (Hudak, P., Peyton Jones, S. L. *et al.*, 1998) does not mention any similar requirement. It can certainly be considered that such a question is of minor importance in the context of a language definition. But, we believe that providing warnings against statically checkable, common, programming errors is an important feature of any mature compiler. And of course, we also believe useless clauses and non-exhaustive matchings to be such errors.

Finally, our approach of studying pattern matching anomalies on the semantical level finally results in more adequate and precise warnings, tailored to various programming situations. Additionally, our technique is applicable to both ML and Haskell; and the cost of our implementation seems to be under control.

Acknowledgements

I thank Jean-Jacques Lévy and James Leifer for their comments. I also thank Jacques Garrigue, whose work on the typing of polymorphic variant (Garrigue, 2004) makes use of exhaustiveness information. Jacques' comments on my code and ideas encouraged me to write this paper.

References

Aitken, W. (1992). *SML/NJ match compiler notes*. `http://www.smlnj.org/compiler-notes/matchcomp.ps`.

Appel, Andrew W., & MacQueen, David B. (1991). Standard ML of New Jersey. *Pages 1–13 of: International symposium on programming language implementation and logic programming*. Springer-Verlag. Lecture Notes in Computer Science 583. `http://www.smlnj.org/`.

Augustsson, L. (1985). Compiling pattern matching. *Pages 368–381 of: Functional programming languages and computer architecture*. Springer-Verlag. Lecture Notes in Computer Science 201.

Baudinet, M., & MacQueen, D. (1985). *Tree pattern matching for ML,*. unpublished paper. `http://www.smlnj.org/compiler-notes/85-note-baudinet.ps`.

Garrigue, J. 2004 (March). Typing deep pattern-matching in presence of polymorphic variants. *Proc. of jssst workshop on programming and programming languages*.

Harper, Robert, Milner, Robin, & Tofte, Mads. (1991). *The definition of standard ML*. The MIT Press.

Hudak, P., Peyton Jones, S. L. *et al.* (1998). *Haskell 98, a non-strict, purely functionnal language*. `http://www.haskell.org/onlinereport/`.

Huet, G., & Lévy, J.-J. (1991). Call by need computations in non-ambiguous linear term rewriting systems. Lassez, J.-L., & Plotkin, G. D (eds), *Computational logic, essays in honor of alan robinson*. The MIT Press.

Kahn, G., & Plotkin, G. D. (1978). *Domaines concrets*. Tech. rept. 336. IRIA Laboria. (In French).

Laville, A. (1991). Comparison of priority rules in pattern matching and term rewriting. *Journal of symbolic computations*, **11**(4), 321–348.

Le Fessant, F., & Maranget, L. (2001). Optimizing pattern-matching. *International conference on functional programming*. ACM press.

Leroy, X., Doligez, D. *et al.* (2003). *The Objective Caml Language (version 3.07)*. `http://caml.inria.fr`.

Maranget, L. (1992). Compiling lazy pattern matching. *Pages 21–31 of: LISP and functional programming*. ACM Press.

Pettersson, M. (1992). A term pattern-match compiler inspired by finite automata theory. *4th workshop on compiler construction*. Springer-Verlag. Lecture Notes in Computer Science 641.

Peyton Jones, S. L., Hall, C. V., Hammond, K., Partain, W., & Wadler, P. (1993). The glasgow haskell compiler: a technical overview. *Proc. UK joint framework for information technology (JFIT) technical conference*. `http://www.haskell.org/ghc/`.

Sekar, R. C., Ramesh, R., & Ramakrishnan, I. V. (1992). Adaptive pattern matching. *International colloquium on automata languages and programming*. Springer-Verlag. Lecture Notes in Computer Science 623.

Sestoft, P. (1996). ML pattern match compilation and partial evaluation. *Pages 446–464 of: Partial evaluation*. Springer-Verlag. Lecture Notes in Computer Science 1110.

Wadler, P. (1987). Efficient compilation of pattern matching. *Chap. 6 of: Peyton Jones, S. L., the implementation of functional programming languages*. Prentice-Hall.

A Series of examples

Series I This series is simple matching by n constant constructors: For a given integer n:

$\texttt{type t = }\mathtt{A}_0\texttt{ | }\mathtt{A}_1\texttt{ | }\cdots\texttt{ | }\mathtt{A}_{n-1}$

```
let f = function
| A₀ -> 0
| A₁ -> 1
    ...
| Aₙ₋₁ -> n − 1
```

Series S This series, taken from (Sestoft, 1996), is a variation of matching by a diagonal matrix. For a given integer n, S_n is a $n+1 \times 2n$ matrix P with, for all i in $[1 \ldots n]$:

$$s^i_{2i} = s^i_{2i-1} = \mathtt{A}, \quad s^i_j = \_ \text{ otherwise}$$

And:

$$s^{n+1}_{2k+1} = \mathtt{A}, \quad s^{n+1}_{2k} = \mathtt{B}$$

For instance, here is S_4:

```
type t = A | B

let f = function
| A,A,_,_,_,_,_,_ -> 1
| _,_,A,A,_,_,_,_ -> 2
| _,_,_,_,A,A,_,_ -> 3
| _,_,_,_,_,_,A,A -> 4
| A,B,A,B,A,B,A,B -> 5
```

Series V This series is taken from (Sekar *et al.*, 1992; Maranget, 1992). It is best defined inductively. We first define B_n as a matrix whose only non-wildcard patterns are the diagonal:

$$b_i^i = \mathtt{B}, \quad b_j^i = \_ \text{ otherwise}$$

Then, V_n is the $n+1 \times \frac{n(n+1)}{2}$ matrix defined as follows:

$$\mathrm{V}_1 = \begin{pmatrix} \mathtt{A} \\ \mathtt{B} \end{pmatrix}, \quad \mathrm{V}_n = \left(\begin{array}{c|c} \mathtt{A}\ \mathtt{A} \ldots \mathtt{A} & \_\ \_\ \ldots \_ \\ \hline \mathrm{B}_n & \mathrm{V}_{n-1} \end{array}\right)$$

For instance, here is V_3:

```
type t = A | B

let f = function
 | A,A,A,_,_,_ -> 1
 | B,_,_,A,A,_ -> 2
 | _,B,_,B,_,A -> 3
 | _,_,B,_,B,B -> 4
```

Series T This series is made of triangular matrices. T_n is the $2n \times n$ matrix defined as follows:

$$t_j^{2k+1} = \_ \text{ (when } j < 2k+1), \quad t_j^{2k+1} = \mathtt{A} \text{ (otherwise)}$$
$$t_j^{2k} = \_ \text{ (when } j < 2k), \quad t_j^{2k} = \mathtt{B} \text{ (otherwise)}$$

For instance, here is T_4:

```
type t = A | B

let f = function
| A,A,A,A -> 1
| B,B,B,B -> 1
| _,A,A,A -> 2
| _,B,B,B -> 2
| _,_,A,A -> 3
| _,_,B,B -> 3
| _,_,_,A -> 4
| _,_,_,B -> 4
```