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 (Le Fessant & Maranget, 2001), some compiler designers choose the backtracking technique. This is, for instance, the case of Objective Caml (Leroy *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-exhaustiveness 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 some 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:

$v ::=$	**(Defined) values**
$c(v_1, v_2, \ldots, v_a)$	$a \geq 0$

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.
- Although many values, including integers, strings, can be seen as terms built over known signatures (maybe of infinite size), not all values can be seen as such. 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 of ML. Instead, we shall rely on the following

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

informal axiom: given any type t, we assume the existence of at least one value that possesses t as a type.

Strictly speaking, the axiom does not hold, at least in Caml where one can define a type with no values in it:

```
type t_empty
```

Then we can define the following data type and matching:

```
type t = C of t_empty
let f x = match x with C y -> y
```

Semantically the above clause `C y -> y` is useless: no value exists that matches the pattern `C y`. But our checker will *not* flag the clause as such, since it assumes the existence of a value of type `t_empty`. Similarily, our checker will flag the following match as non-exhaustive:

```
type tt = A | B of t_empty
let g x = match x with A -> true
```

The non-exhaustiveness diagnostic is wrong, strictly speaking, since there does not exist a value $B(v)$, where v has type `t_empty`. As a consequence, the match above matches all possible values of type `tt`. We view this issue as a minor one, considering that the non-empty type axiom holds for the vast majority of types.

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 annotations 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 follows.

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

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 axiom "types are not empty", any pattern p admits at least one instance.

In Definition 1 above we used the very convenient notations $\vec{p} = (p_1 \cdots p_a)$ and $\vec{v} = (v_1 \cdots v_a)$. 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 ($m = 0$ and $n \geq 0$) are written $\emptyset$; while non-empty matrices of empty rows ($m > 0$ and $n = 0$) are written $\begin{pmatrix}\end{pmatrix}$. 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. $\forall j < i,\ (p^j_1 \cdots p^j_n) \not\preceq (v_1 \cdots v_n)$

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, typing is 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 $\vec{v} = (\texttt{Nil Nil})$ matches the first row of P, since $(\texttt{Nil } _) \preceq (\texttt{Nil Nil})$ (because $p_1^1 = \texttt{Nil} \preceq \texttt{Nil} = v_1$) and $p_2^1 = _ \preceq \texttt{Nil} = v_2$).
2. Vector $\vec{w} = (\texttt{One(0) Nil})$ matches the second row of P, since we have
   (a) $(_ \texttt{ Nil}) \preceq (\texttt{One(0) Nil})$, that is $\vec{w}$ is an instance of the second row of matrix P.
   (b) $(\texttt{Nil } _) \not\preceq (\texttt{One(0) Nil})$ (because $p_1^1 = \texttt{Nil} \not\preceq \texttt{One(0)} = w_1$), that is, $\vec{w}$ is not an instance of the first row of matrix P.
3. Vector $\vec{z} = (\texttt{One(1) One(1)})$ 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, $\texttt{0} \not\preceq \texttt{1}$ implies $\texttt{One(0)} \not\preceq \texttt{One(1)}$, 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, a pattern can be interpreted as the set of its instances. Similarly, a matrix can be interpreted as the union of the instances of its 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 $P^{[1\ldots i)} \not\preceq \vec{v}$ 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.

Useless clauses are sometimes called redundant. In our opinion, "useless" is more precise, since it conveys the semantical nature of the concept better.

Example 2

Let P and Q be the following two pattern matrices.

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

Matrix P is not exhaustive, since, for instance, vector $\vec{v} = (\texttt{One (0) One (0)})$ 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 match" 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
Corollary of definitions. □

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. More precisely, it is important to notice that $\vec{v}$ matches some row in P (Definition 2) is equivalent to $P \preceq \vec{v}$ (Definition 3). In other words, the order of rows in P is irrelevant while computing $\mathcal{U}(P, \vec{q})$.

3.1 Solving the useful clause problem

In this section we compute $\mathcal{U}$ recursively. We proceed by first defining a recursive fonction $\mathcal{U}_{\text{rec}}$ and then showing $\mathcal{U} = \mathcal{U}_{\text{rec}}$. The definition of $\mathcal{U}_{\text{rec}}$ 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 of size n. Induction proceeds by decomposing P and $\vec{q}$ along first column.

Base case If there is no column (*i.e.* $n = 0$), then the value of $\mathcal{U}_{\text{rec}}(P, ())$ depends upon the number of rows m of matrix P.

1. If P has some rows (*i.e.* $m > 0$), we define $\mathcal{U}_{\text{rec}}(\begin{pmatrix}\end{pmatrix}, ())$ to be false.
2. If m is zero, then we define $\mathcal{U}_{\text{rec}}(\emptyset, ())$ to be true. More generally, although not really necessary, we can define $\mathcal{U}_{\text{rec}}(\emptyset, \vec{q})$ to be true for any vector $\vec{q}$ of any size n.

Base cases are summarized as follows:

$$\mathcal{U}_{\text{rec}}(\begin{pmatrix}\end{pmatrix}, ()) = \mathit{False} \qquad \mathcal{U}_{\text{rec}}(\emptyset, \vec{q}) = \mathit{True}.$$

Induction If there are columns ($n > 0$), then there are three sub-cases depending upon the nature of pattern q_1.

1. Pattern q_1 is a constructed pattern, that is $q_1 = c(r_1, \ldots, r_a)$. From matrix P, we extract the new *specialized* matrix $\mathcal{S}(c, P)$. 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	$\mathcal{S}(c, P)$
$c(r_1, \ldots, r_a)$	$r_1 \cdots r_a \quad p_2^i \cdots p_n^i$
$c'(r_1, \ldots, r_{a'})$ $(c' \neq c)$	No row
$_$	$_ \cdots _ \quad p_2^i \cdots p_n^i$
$(r_1 \mid r_2)$	$\mathcal{S}(c, \begin{pmatrix} r_1 & p_2^i \cdots p_n^i \\ r_2 & p_2^i \cdots p_n^i \end{pmatrix})$

Notice that a given row $\vec{p}^{\,i}$, may induce one, none or several rows in $\mathcal{S}(c, P)$. In the following, we note $\mathcal{S}(c, \vec{q})$ the application of $\mathcal{S}$ to a vector, when it yields a vector.

$$\begin{aligned} \mathcal{S}(c, (c(r_1, \ldots, r_a)\ q_2 \cdots q_n)) &= (r_1 \cdots r_a\ q_2 \cdots q_n) \\ \mathcal{S}(c, (_\ q_2 \cdots q_n)) &= (\underbrace{_ \cdots _}_{a \text{ times}}\ q_2 \cdots q_n) \end{aligned}$$

We also consider specialisation of value vectors when relevant, that is when $v_1 = c(w_1, \ldots, w_a)$.
Finally, in the case where q_1 is $c(r_1, \ldots, r_a)$, we define:

$$\mathcal{U}_{\text{rec}}(P, \vec{q}) = \mathcal{U}_{\text{rec}}(\mathcal{S}(c, P), \mathcal{S}(c, \vec{q})).$$

2. Pattern q_1 is a wildcard. Let $\Sigma = \{c_1, c_2, \ldots, c_z\}$ be the set of 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). The computation of $\mathcal{U}_{\text{rec}}$ depends on whether set Σ is a complete signature or not. In the former case, any instance $\vec{v}$ of $\vec{q}$ necessarily possesses a first component whose root constructor belongs to Σ. In the latter case, it turns out that it suffices to examine those constructors that do *not* belong to Σ. Here, computing $\mathcal{U}_{\text{rec}}$ significantly departs from compilation to decision trees, which of course has to take all constructors into account.
   (a) Set Σ constitutes a complete signature. Then we define:

$$\mathcal{U}_{\text{rec}}(P, \vec{q}) = \bigvee_{k=1}^{z} \mathcal{U}_{\text{rec}}(\mathcal{S}(c_k, P), \mathcal{S}(c_k, \vec{q})).$$

   (b) Set Σ is not a complete signature. From P, we extract the new *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
$_$	$p_2^i \cdots p_n^i$
$(r_1 \mid r_2)$	$\mathcal{D}(\begin{pmatrix} r_1 & p_2^i \cdots p_n^i \\ r_2 & p_2^i \cdots p_n^i \end{pmatrix})$

   Matrix $\mathcal{D}(P)$ is defined in all situations, whether Σ is a complete signature or not. However, $\mathcal{D}(P)$ is useful for computing $\mathcal{U}_{\text{rec}}$ only in the latter case. We define:

$$\mathcal{U}_{\text{rec}}(P, (_\ q_2 \cdots q_n)) = \mathcal{U}_{\text{rec}}(\mathcal{D}(P), (q_2 \cdots q_n)).$$

   Observe that when Σ is empty, *i.e.* when the first column of P is made of wildcards and of or-patterns thereof, then Σ is not a complete signature. Thus the definition above also apply.

3. When pattern q_1 is an or-pattern $(r_1 \mid r_2)$, we define:

$$\mathcal{U}_{\text{rec}}(P, ((r_1 \mid r_2)\ q_2 \cdots q_n)) = \mathcal{U}_{\text{rec}}(P, (r_1\ q_2 \cdots q_n)) \vee \mathcal{U}_{\text{rec}}(P, (r_2\ q_2 \cdots q_n)).$$

We now establish a few "key" properties of matrix specialization (1 below) and of the default matrix (2 to 4 below). Basically, the key property of specialization expresses that matching by P and $\mathcal{S}(c, P)$ are equivalent for value vectors whose first component admits c as a root constructor; while the key properties of the default matrix express the equivalence of matching by P and $\mathcal{D}(P)$ in more detailled situations.

Lemma 1 (*Key properties*)
For any matrix P, constructor c, and value vector $\vec{v}$ such that $v_1 = c(w_1, \ldots, w_a)$ (all being of the appropriate types), we have:

$$P \not\preceq \vec{v} \iff \mathcal{S}(c, P) \not\preceq \mathcal{S}(c, \vec{v}\,). \tag{1}$$

Additionally, for any value vector $\vec{v}$, we have:

$$P \not\preceq (v_1\ v_2 \cdots v_n) \Longrightarrow \mathcal{D}(P) \not\preceq (v_2 \cdots v_n). \tag{2}$$

Furthermore, given any matrix P, let Σ be set of the root constructors of P's first column. If Σ is not empty, then for any constructor c not in Σ and any value vector $(w_1 \cdots w_a\ v_2 \cdots v_n)$, we have:

$$\mathcal{D}(P) \not\preceq (v_2 \cdots v_n) \Longrightarrow P \not\preceq (c(w_1, \ldots, w_a)\ v_2 \cdots v_n). \tag{3}$$

If Σ is empty, then, for any value vector $\vec{v}$, we have instead:

$$\mathcal{D}(P) \not\preceq (v_2 \cdots v_n) \Longrightarrow P \not\preceq (v_1\ v_2 \cdots v_n). \tag{4}$$

Proof
Mechanical application of definitions. □

We could of course have formulated the key properties by reversing implications and by using $\preceq$ in place of $\not\preceq$. However, we adopt the negated formulation, to match Definition 2. Nevertheless, we shall also consider (1) when P has exactly one row. In that case, for any value vector $\vec{v}$ such that $v_1 = c(w_1, \ldots, w_a)$, we write more directly:

$$\vec{q} \preceq \vec{v} \iff \mathcal{S}(c, \vec{q}\,) \preceq \mathcal{S}(c, \vec{v}\,).$$

Proposition 2
For any matrix P and pattern vector $\vec{q}$ of appropriate sizes and types, we have:

$$\mathcal{U}(P, \vec{q}\,) = \mathcal{U}_{\text{rec}}(P, \vec{q}\,).$$

Proof
Base cases are easy. Let first $\vec{q}$ be the empty pattern vector, written (). The set of $\vec{q}$ instances consists of the unique empty value vector, also written (). If P's rows exist and are empty, then P's first row filters the value vector ().

$$\mathcal{M}(\begin{pmatrix} \end{pmatrix}, ()) = \emptyset.$$

Moreover, if P has no rows, then it cannot filter any value, We have:

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

And we conclude, since $\vec{q}$ has at least one instance for any $\vec{q}$.

To prove inductive cases, it suffices to show that $\mathcal{U}$ meets the equations that define $\mathcal{U}_{\text{rec}}$.

1. If $q_1 = c(r_1, \ldots, r_a)$ for some constructor c, then we need prove:

$$\mathcal{U}(P, \vec{q}\,) = \mathcal{U}(\mathcal{S}(c, P), \mathcal{S}(c, \vec{q}\,)).$$

However, by (1) applied to both P and $\vec{q}$, we have the stronger result:

$$\mathcal{M}(P,\vec{q}) = \{\vec{v} \mid \mathcal{S}(c,\vec{v}) \in \mathcal{M}(\mathcal{S}(c,P),\mathcal{S}(c,\vec{q}))\}\,.$$

Namely, remember that $\mathcal{U}(P,\vec{q})$ means that the set $\mathcal{M}(P,\vec{q})$ of matching values is not empty (Definition 6).

2. If q_1 is a wildcard, then let $\Sigma = \{c_1,\ldots,c_z\}$ be as in the definition of $\mathcal{U}_{\text{rec}}$.
   (a) If Σ is a complete signature. For any c_k in Σ, we define the set M_k:

   $$M_k = \mathcal{M}(\mathcal{S}(c_k,P),\mathcal{S}(c_k,\vec{q})).$$

   By typing, for any value v_1 of the appropriate type, we have $q_1 \preceq v_1$, if and only if there exists a constructor c_k in Σ and values $w_1,\ldots,w_{a_k}$ such that $v_1 = c_k(w_1,\ldots,w_{a_k})$. Thus, by property (1), one easily shows:

   $$\mathcal{M}(P,\vec{q}) = \bigcup_{k=1}^{z} \{\,\vec{v} \mid \mathcal{S}(c_k,\vec{v}) \in M_k\,\}\,.$$

   And we can conclude:

   $$\mathcal{U}(P,\vec{q}) = \bigvee_{k=1}^{z} \mathcal{U}(\mathcal{S}(c_k,P),\mathcal{S}(c,\vec{q})).$$

   (b) In all situations, we have (by (2)):

   $$\mathcal{M}(P,\vec{q}) \subseteq \{\vec{v} \mid (v_2\cdots v_n) \in \mathcal{M}(\mathcal{D}(P),(q_2\cdots q_n))\}\,.$$

   In the case where Σ is empty, the reverse inclusion holds (by 4). And we can conclude, by the "type are not empty" axiom.
   It is worth noticing that the reverse inclusion does not hold when Σ is non-empty. Namely, when considering sets of matching values $\mathcal{M}$, we have to take all possible values into account. Anyway, by the inclusion above, we have: $\mathcal{U}(P,\vec{q}) \Longrightarrow \mathcal{U}(\mathcal{D}(P),(q_2\cdots q_n))$.
   Conversely, assume $\mathcal{U}(\mathcal{D}(P),(q_2\cdots q_n)) = \mathit{True}$, and let t be the type of the first component of tested value vectors. Then, there exists $(v_2\cdots v_n)$ such that $\mathcal{D}(P) \not\preceq (v_2\cdots v_n)$ and $(q_2\cdots q_n) \preceq (v_2\cdots v_n)$. Furthermore, 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 $c \notin \Sigma$. Thus, by our 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 the validity of $\mathcal{U}(P,\vec{q})$, by (3) and $q_1 = _ \preceq v_1$.

3. If q_i is an or-pattern $(r_1 \mid r_2)$, then, by definition of $\preceq$ for or-patterns, we have:

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

□

3.2 Detecting the anomalies

Since we know how to compute $\mathcal{U}$, we can detect pattern matching anomalies. Given some expression `match ... with` p_1 `->` e_1 `|` p_2 `->` e_2 `| ... |` p_m `->` e_m, exhaus-

tiveness is checked by computing:

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

Furthermore, the usefulness of clause number i is checked by computing:

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

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 words, 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 $_ \preceq \Omega$, value Ω possesses all types). 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`. 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`. 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 c, then expression e

[2] In this section, we use Haskell syntax

is a head-normal form and we 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, when $\mathcal{P}(v)$ holds, $\mathcal{P}(w)$ also holds for all w such that $v \preceq_\Omega w$. With monotonic predicates, matching decisions do not change during computations. One should notice that, given any pattern p, the 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 restrictions* 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 proving $\mathcal{U}_\mathcal{P} = \mathcal{U}_{\text{rec}}$. Thus, we re-examine the proof of Proposition 2 in the context of lazy pattern matching.

Base cases follow from basic restrictions.

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

To prove the inductive cases, it suffices to reformulate the key properties of Lemma 1, replacing $P \not\preceq \vec{v}$ by $\mathcal{P}(P, \vec{v})$. However, key properties now are rather assumed than established.

Definition 9 (*Key properties*)
We say that predicate $\mathcal{P}$ meets key properties when the following four properties hold. For any matrix P, constructor c, and value vector $\vec{v}$ such that $v_1 = c(w_1, \ldots, w_a)$, we have:

$$\mathcal{P}(P, \vec{v}) \iff \mathcal{P}(\mathcal{S}(c, P), \mathcal{S}(c, \vec{v})). \tag{1}$$

Additionally, for any value vector $\vec{v}$, we have:

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

Furthermore, given any matrix P, let Σ be set of the root constructors of P's first column. If Σ is not empty, then for any constructor c not in Σ and any value vector $(w_1 \cdots w_a\ v_2 \cdots v_n)$, we have:

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

If Σ is empty, then, for any value vector $\vec{v}$, we have 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 that assuming key properties suffices to prove that $\mathcal{U}_{\mathcal{P}}$ can be computed as $\mathcal{U}$ is, since Ω does not show in the proof of Proposition 2. Indeed, monotonicity plays some part here.

Proposition 3
We have $\mathcal{U}_{\mathcal{P}} = \mathcal{U}_{\mathrm{rec}}$.

Proof
Base cases follow from basic restrictions; while the proofs of all inductive cases in Proposition 2, except 2-(a), apply unchanged.

Hence, we assume q_1 to be a wildcard and the set Σ to be a complete signature. We need prove:

$$\mathcal{U}_{\mathcal{P}}(P, ((_ : t)\ q_2 \cdots q_n)) = \bigvee_{k=1}^{z} \mathcal{U}_{\mathcal{P}}(\mathcal{S}(c_k, P), \mathcal{S}(c_k, \vec{q})).$$

By (1), for any constructor c_k in Σ and any vector $\vec{v}$ such that $v_1 = c_k(w_1, \ldots, w_{a_k})$, we have: $\vec{v} \in \mathcal{M}_{\mathcal{P}}(P, \vec{q}) \iff \mathcal{S}(c_k, \vec{v}) \in \mathcal{M}_{\mathcal{P}}(\mathcal{S}(c_k, P), \mathcal{S}(c_k, \vec{q}))$. Hence, a potential difficulty arises for vectors $\vec{v}$ in $\mathcal{M}_{\mathcal{P}}(P, \vec{q})$, when v_1 is Ω. 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 $(c(w_1, \ldots, w_a)\ v_2 \cdots v_n) \in \mathcal{M}_{\mathcal{P}}(P, \vec{q})$. That is (by (1) in the forwards direction), $\mathcal{U}_{\mathcal{P}}(\mathcal{S}(c_k, P), \mathcal{S}(c_k, \vec{q}))$ holds for all c_k in Σ. □

As an immediate consequence of the proposition above, 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 $\preceq$, `True` $\preceq \Omega$ does not hold, hence Ω cannot match first clause. However, $\mathcal{P}(\begin{pmatrix} \texttt{True} \end{pmatrix}, \Omega)$ 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 any value v in $\{\Omega, \texttt{False}, \texttt{True}\}$, we have $_ \prec v$. Thus we have:

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

Hence, by our first basic restriction, 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 $P \# \vec{v}$, when making $\vec{v}$ more precise cannot produce an instance of P. That is, $P \# \vec{v}$ means

$$\forall \vec{w}, \vec{v} \preceq_\Omega \vec{w} \Longrightarrow P \not\preceq \vec{w}.$$

Definition 10 (*Lazy pattern matching (Laville)*)
Define $\mathcal{P}(P, \vec{v}) = P \# \vec{v}$ in the generic definition 8.

The first basic restriction follows by letting $\vec{w}$ be $\vec{v}$ in the definition of $P \# \vec{v}$, the second restriction follows from $\emptyset \not\preceq \vec{w}$ for all $\vec{w}$, and monotonicity is a consequence of the transitivity of $\preceq_\Omega$.

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

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

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 restriction, $\mathcal{P}(P, \vec{w})$ does not hold, and, by the monotonicity of $\mathcal{P}$, $\mathcal{P}(P, \vec{v})$ does not hold either.

Incompatibility is easily computed by the following rules.

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

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, the Haskell committee adopted another semantics for pattern matching. Their definition is aware of the presence of Ω and solves the difficulty by specifying left-to-right testing order.

4.3 Pattern matching in Haskell

By interpreting the Haskell report (Hudak *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.

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

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

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

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:

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

Informally, $\mathcal{H}(P, \vec{v}) = T$ 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 Haskell way of pattern matching in our setting.

Definition 11 (*Haskell pattern matching*)
Define $\mathcal{P}(P, \vec{v})$ 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 restrictions and

key properties (decomposing along first columns is instrumental). Hence, save for or-patterns, algorithm $\mathcal{U}_{\text{rec}}$ also computes the utility of pattern matching in Haskell.

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:

$$\mathcal{H}((p_1 \mid p_2), v) = \mathcal{H}(p_1, v) \vee_{\perp} \mathcal{H}(p_2, v).$$

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 $\mathcal{H}((\texttt{True} \mid _), \Omega) = \perp$.

However, the left-to-right implication still holds, and the following definition of Haskell pattern matching makes sense.

Definition 12 (*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:

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

From this definition of matching, we define the utility predicate $\mathcal{U}_{\mathcal{H}}$ and the set of matching values $\mathcal{M}_{\mathcal{H}}$ as we did in Definition 6.

The definition above is *not* the application of the generic definition 8 to $\mathcal{P}(P, \vec{v}) = (\mathcal{H}(P, \vec{v}) = F)$, because we have written $\mathcal{H}(\vec{p}^{\,i}, \vec{v}) = T$ in place of the instance relation $\vec{p}^{\,i} \preceq \vec{v}$. However, those two relations are closely connected.

Lemma 2
Let p be a pattern and v be a value such that $\mathcal{H}(p, v) = \perp$. There exists value w such that $v \preceq_{\Omega} w$ and $\mathcal{H}(p, w) \neq \perp$. Furthermore, if $p \preceq v$, then $\mathcal{H}(p, w) = T$.

Proof
We first prove the existence of w by induction on p.

- If $p = c(p_1, \ldots, p_a)$, then, by hypothesis $\mathcal{H}(p, v) = \perp$, we have two sub-cases.
  - Value v is $c(v_1, \ldots, v_a)$, with $\mathcal{H}(p_i, v_i) = \perp$ for i in some (non-empty) index set I. Applying induction hypothesis to all such i yields values v'_i such that $v_i \preceq_{\Omega} v'_i$ and $\mathcal{H}(p_i, v'_i) \neq \perp$. Then, we define $w = c(w_1, \ldots, w_a)$ where $w_i = v'_i$ for $i \in I$, and $w_i = v_i$ otherwise.
  - Otherwise, value v is Ω. Let v' be $c(\Omega, \ldots, \Omega)$. If $\mathcal{H}(p, v')$ is not $\perp$, then we define $w = v'$. Otherwise, we reason as in the previous case.
- If $p = (q_1 \mid q_2)$, we have two sub-cases.
  - If $\mathcal{H}(q_1, v) = F$ and $\mathcal{H}(q_2, v) = \perp$, then (by induction) there exists a value w such that $\mathcal{H}(q_2, w) \neq \perp$ and, by definition of $\mathcal{H}$, we have $\mathcal{H}(p, w) = \mathcal{H}(q_2, w)$.

— If $\mathcal{H}(q_1, v) = \bot$, then (by induction) there exists a value w', such that $v \preceq_\Omega w'$ and $\mathcal{H}(q_1, w') \neq \bot$ If $\mathcal{H}(q_1, w') = T$, we define w to be w' and we conclude. Otherwise, $\mathcal{H}(q_1, w') = F$ and thus $\mathcal{H}((q_1 \mid q_2), w') = \mathcal{H}(q_2, w')$. Then we conclude, either directly, or by induction in the case where $\mathcal{H}(q_2, w') = \bot$.

Additionally, $\mathcal{H}(p, w) = T$ holds under the extra hypothesis $p \preceq v$, by $\mathcal{H}(p, w) = F \Longrightarrow p \not\preceq w$ and by the monotonicity of $\preceq$. □

We could have been tempted by the simpler lemma: $H(p, v) = \bot$ implies the existence of w such that $v \preceq_\Omega w$ and $\mathcal{H}(p, v) = T$. However, this proposition is false, as shown by the example $p =$ `(true, false)` and $v =$ `(Ω, true)`.

In any case, the above lemma is proved and it suffices to relate Haskell matching to generic lazy matching, and thus to compute the utility of Haskell matching.

Proposition 4
We have $\mathcal{U}_\mathcal{H} = \mathcal{U}_{\text{rec}}$.

Proof
We note $\mathcal{U}_{\mathcal{H}\preceq}$ the utility predicate that results from the generic definition, taking $\mathcal{P}(P, \vec{v})$ to be $\mathcal{H}(p, \vec{v}) = F$. From generic proposition 3, we have $\mathcal{U}_{\mathcal{H}\preceq} = \mathcal{U}_{\text{rec}}$. (Formally we check that predicate $\mathcal{H}(P, \vec{v}) = F$ meets key properties even when some of the patterns in P are or-patterns).

Then we show $\mathcal{U}_\mathcal{H} = \mathcal{U}_{\mathcal{H}\preceq}$. From the implication $\mathcal{H}(\vec{q}, \vec{v}) = T \Longrightarrow \vec{q} \preceq \vec{v}$, we have $\mathcal{U}_\mathcal{H}(P, \vec{q}) \Longrightarrow \mathcal{U}_{\mathcal{H}\preceq}(P, \vec{q})$; the converse implication follows from Lemma 2. □

It is time to clearly stress on some important consequence of propositions $\mathcal{U} = \mathcal{U}_{\text{rec}}$, $\mathcal{U}_\mathcal{P} = \mathcal{U}_{\text{rec}}$ and $\mathcal{U}_\mathcal{H} = \mathcal{U}_{\text{rec}}$: all our utility predicates are in fact equal. This suggests a quite powerful and elegant"semantical" proof technique, which we immediately demonstrate.

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

Proof
Consider strict matching. Since predicates $P \not\preceq \vec{v}$ and $\vec{q} \preceq \vec{v}$ do not depend on column order, we have $\mathcal{U}(P, \vec{q}) = \mathcal{U}(P', \vec{q}')$. From $\mathcal{U}_\mathcal{H} = \mathcal{U}$, we conclude. □

First observe that proving the irrelevance of column order for Haskell matching by induction on matrix and pattern structure would be quite cumbersome.

Also notice that the lemma above is not obvious, since Haskell matching depends upon column order in a strong sense. For instance, let P, $\vec{q}$, P' and $\vec{q}'$ be as follows.

$P = \begin{pmatrix} \texttt{True} & \texttt{False} \end{pmatrix}$ $\quad \vec{q} = (\texttt{False}\ _)$ $\quad P' = \begin{pmatrix} \texttt{False} & \texttt{True} \end{pmatrix}$ $\quad \vec{q}' = (_\ \texttt{False})$.

Matrix P' (resp. vector $\vec{q}'$) is P (resp. $\vec{q}$) with columns swapped. The sets of matching values are as follows.

$$\begin{aligned} \mathcal{M}_\mathcal{H}(P, \vec{q}) &= \{\, (\texttt{False}\ \Omega), (\texttt{False}\ \texttt{True}), (\texttt{False}\ \texttt{False}) \,\} \\ \mathcal{M}_\mathcal{H}(P', \vec{q}') &= \{\, (\texttt{True}\ \texttt{False}), (\texttt{False}\ \texttt{False}) \,\} \end{aligned}$$

Swapping the components of the elements of $\mathcal{M}_{\mathcal{H}}(P, \vec{q})$ does not yield $\mathcal{M}_{\mathcal{H}}(P', \vec{q}')$, since $(\Omega\ \texttt{False})$ does not belong to $\mathcal{M}_{\mathcal{H}}(P', \vec{q}')$. However, some of the values of the $\mathcal{M}_{\mathcal{H}}$ sets above are related by the permutation. Moreover, the equality $\mathcal{U}_{\mathcal{H}} = \mathcal{U}$ can be seen as telling us that there is at least one such value.

From now on, we simply write $\mathcal{U}$ for any utility predicate, regardless of semantics. We also write "algorithm $\mathcal{U}$" for $\mathcal{U}_{\text{rec}}$.

PART TWO
Implementation

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

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

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 matching, hoping this will make it exhaustive. 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, one achieves exhaustive match by adding a clause with pattern `_::_`.

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 $\mathcal{I}$ normally 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 $\bot$.

Base case If $n = 0$, we define:

$$\mathcal{I}(\left(\ \right), 0) = \bot \qquad\qquad \mathcal{I}(\emptyset, 0) = ().$$

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

Induction If $n > 0$, then let Σ be the set of constructors that appear at the root of the patterns (and of or-pattern alternatives) in the first column of P.

1. We first assume that Σ is a complete signature. Then, we should perform the recursive calls $\mathcal{I}(\mathcal{S}(c_k, P), a_k + n - 1)$, for all c_k taken from Σ.

If *all* those computations return $\bot$, then $\mathcal{I}(P,n)$ also is $\bot$. 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 define $\mathcal{I}(P,n)=(c_k(r_1,\ldots,r_{a_k})\ p_2\cdots p_n)$. 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 $\mathcal{I}(\mathcal{S}(c_{k'},P),a_{k'}+n-1)$ returns a pattern vector, algorithm $\mathcal{I}$ is non-deterministic.

2. If Σ is not a complete signature, we only perform the recursive call $\mathcal{I}(\mathcal{D}(P),n-1)$ and we define $\mathcal{I}(P,n)=\bot$ when the recursive call returns $\bot$. 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 define $\mathcal{I}(P,n)=(_\ p_2\cdots p_n)$. If Σ is not empty, then we define $\mathcal{I}(P,n)=(c(_,\ldots,_)\ p_2\cdots p_n)$, 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 an or-pattern that includes all the extra constructors.

It should be clear that $\mathcal{I}(P,n)=\bot$, if and only if P is exhaustive. 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 $\mathcal{I}((\texttt{One 1}),1)$.

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

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 recompile the corrected program, then the compiler will flag other non-matching patterns such as `One 0`. Hence, the whole information is 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 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 with our 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 patterns 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 as to avoid producing code of exponential size. Consider function `f` below.

```
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 $\mathcal{U}$, which does not exhibit exponential behavior here, provided that we compute disjunctions sequentially.

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:

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

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

Proof
For Haskell matching, the equality follows from definitions — $\mathcal{H}((r_1 \mid r_2), v_1) = T$, if and only if $\mathcal{H}(r_1, v_1) = T$ or $\mathcal{H}(r_1, v_1) = F \wedge \mathcal{H}(r_2, v_1) = T$. □

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, with q_1 being the or-pattern $(r_1 \mid r_2)$. 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 (more exactly a set of useless pattern positions), that is, $\emptyset$ in the first case, $\{r_1\}$ in the second case, $\{r_2\}$ in the third case, and the distinguished set $\top$ in the fourth case.

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

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

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}	$E_{(r_1 \mid r_2)}$
$\emptyset$	$\emptyset$	$\emptyset$
$\top$	$\top$	$\top$
$\emptyset$	$\top$	$\{r_2\}$
$\top$	$\emptyset$	$\{r_1\}$

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

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 of 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 $E_{q_1} \cup E_{q_2}$. 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 define 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, before performing expansions as we did in the previous section. In practice, it is convenient to partition the *columns* of matrices and vectors into three subparts. We note those separations with "•". That is, $\mathcal{U}'$ takes such "dotted" matrices and vectors as arguments, written $P \bullet Q \bullet R$ and $\vec{p} \bullet \vec{q} \bullet \vec{r}$. Dotted matrices and vectors stand for triples of matrices and vectors. Later in this section, component $\vec{q}$ will holds patterns that cannot contain or-patterns (*i.e.* wildcards), while all the components of $\vec{r}$ will be or-patterns.

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), looking for or-patterns. When or-patterns are found, the corresponding columns are transferred to the R subpart

Fig. 2. Operations on dotted matrices

(a) Specialization by constructor c

row in $P \bullet Q \bullet R$	row(s) in $\mathcal{S}(c, P \bullet Q \bullet R)$
$c(t_1, \ldots, t_a) \cdots p_n^i \bullet q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i$	$t_1 \cdots t_a \; p_2^i \cdots p_n^i \bullet q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i$
$_ \qquad \cdots p_n^i \bullet q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i$	$_ \cdots _ \; p_2^i \cdots p_n^i \bullet q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i$
$c'(t_1, \ldots, t_{a'}) \cdots p_n^i \bullet q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i$	no row
$(t_1 \mid t_2) \quad \cdots p_n^i \bullet q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i$	$\mathcal{S}\left(c, \begin{pmatrix} t_1 \cdots p_n^i \bullet q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i \\ t_2 \cdots p_n^i \bullet q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i \end{pmatrix}\right)$

(b) Right shifts

row in $P \bullet Q \bullet R$	row in $\Rightarrow_1(P \bullet Q \bullet R)$	row in $\Rightarrow_2(P \bullet Q \bullet R)$
$p_1^i \cdots p_n^i \bullet q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i$	$p_2^i \cdots p_n^i \bullet p_1^i \; q_1^i \cdots q_k^i \bullet r_1^i \cdots r_z^i$	$p_2^i \cdots p_n^i \bullet q_1^i \cdots q_k^i \bullet p_1^i \; r_1^i \cdots r_z^i$

(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 `match ... with` p_1 `->` e_1 `|` p_2 `->` e_2 `|` ... `|` p_m `->` e_m, we perform the initial call

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

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 "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 define:
$$\begin{aligned}\mathcal{U}'(P \bullet Q \bullet R, (c(t_1, \ldots, t_a) \; p_2 \cdots p_n) \bullet \vec{q} \bullet \vec{r}) = \\ \mathcal{U}'(\mathcal{S}(c, P \bullet Q \bullet R), (t_1 \cdots t_a \; p_2 \cdots p_n) \bullet \vec{q} \bullet \vec{r}).\end{aligned}$$
2. If p_1 is a wildcard, we transfer P's 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 an or-pattern, we transfer P's 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 sub-cases depending on whether there are columns in R or not.
   (a) If there were no or-patterns inside initial $\vec{p}$ ($\vec{r} = ()$), we simply 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 an or-pattern $(t_1 \mid t_2)$ and we perform the following 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 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 one or-pattern is performed exactly once.
Then, we combine E_{t_1} and E_{t_2} by the rules of Figure 1, yielding E_{r_j}. Finally, we define:

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

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

Proposition 5
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 following 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. is like inductive case 1. in Proposition 2.

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 define $S = Q \,\&\, R$ and $\vec{s} = \vec{q} \,\&\, \vec{r}$. Furthermore, we express the expansion of $r_j = (t_1 \mid t_2)$ as follows.

$$E_{t_1} = \mathcal{U}'(P' \bullet Q' \bullet \,, (t_1) \bullet \vec{q}' \bullet \,), \qquad E_{t_2} = \mathcal{U}'(P'' \bullet Q'' \bullet \,, (t_2) \bullet \vec{q}' \bullet \,)$$

Where it should be clear that P'' is $P' @ (t_1)$ and Q'' is $Q' @ \vec{q}'$. We further define S' to be $P' \,\&\, Q'$ and S'' to be $P'' \,\&\, Q''$. Notice that S'' is S' with the row $(t_1) \,\&\, \vec{q}'$ added. Let also $\vec{s}'$ be the vector $(t_1 \mid t_2) \,\&\, \vec{q}'$. Matrix S' and vector $\vec{s}'$ are the images of S and $\vec{s}$ by the same permutation of columns. By lemmas 3 and 4, we have:

$$\mathcal{U}(S, \vec{s}) = \mathcal{U}(S', \vec{s}') = \mathcal{U}(S', (t_1) \,\&\, \vec{q}') \vee \mathcal{U}(S'', (t_2) \,\&\, \vec{q}').$$

By induction, we have the following two equivalences.

$$E_{t_1} = \top \iff \mathcal{U}(S', (t_1)\,\&\,\vec{q}\,') = \mathit{False} \qquad E_{t_2} = \top \iff \mathcal{U}(S'', (t_2)\,\&\,\vec{q}\,') = \mathit{False}$$

Then, since $E_{(t_1\,|\,t_2)} = \top$, if and only if $E_{t_1} = \top$ and $E_{t_2} = \top$, we have:

$$E_{r_j} = \top \iff \mathcal{U}(S, \vec{s}) = \mathit{False}.$$

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

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, ..., $m-1$ 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. Disregarding or-patterns and (unreasonably) assuming that $\mathcal{U}$ is linear in its input size, we expect a quadratic behavior in the size of the pattern matrices, which we define as the sum of the sizes of its patterns, and will assimilate to source file size in experiments. Namely, for a matrix of size S with m rows, we perform $m+1$ calls of $\mathcal{U}$ on arguments of size at most S, where m is, at worst, of the same order magnitude as 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 behavior 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 is as follows: $\mathcal{U}(P, \vec{q})$ is false; all patterns in $\vec{q}$ are wildcards; and the patterns in P contain mostly wildcards and a few different constructor patterns such that constructors collected column-wise are complete signatures. And indeed, one can construct a series of matchings that confirms the exponential time behavior of a straightforward implementation of algorithm $\mathcal{U}$. Besides, randomly generated matrices also confirm that the exponential time behavior can occur. Such problematic random matrices test vectors of booleans and many of their patterns are wildcards.

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 match is a potentially unfavorable situation, since we here compute $\mathcal{U}(P, (_ \cdots _)) = \mathit{False}$. Hence, we examine how human beings write exhaustive matches. An easy and frequent way to ensure exhaustiveness is to complete a matching with a wildcard pattern. Hence, we could first scan matrices searching for rows of wildcards, and announce exhaustive match 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 defined value we have $_ \preceq v$. That is, for any pattern p and defined value v, we have $p \preceq v \Longrightarrow _ \preceq v$ (or $_ \not\preceq v \Longrightarrow p \not\preceq v$). 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 any defined value v, we have `(1,2)` $\preceq v \Longrightarrow$ `(1,_)` $\preceq v$ (or `(1,_)` $\not\preceq v \Longrightarrow$ `(1,2)` $\not\preceq v$). Thus, before we compute exhaustiveness, we can delete pattern `(1,2)` from the matching, because its presence does not add non-matching values to those values that do not match the remaining pattern `(1,_)`.

We now consider the general case.

Definition 13
Let $\vec{p}$ and $\vec{q}$ be two pattern vectors. By definition, $\vec{p}$ *subsumes* $\vec{q}$ when, for all *defined* value vectors $\vec{v}$, 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 $\vec{p}^{\,j}$, 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 $\not\preceq$ we have:

$$P \not\preceq \vec{v} \iff P' \not\preceq \vec{v} \wedge \vec{p}^{\,j} \not\preceq \vec{v}.$$

Furthermore, by definition of $P' \not\preceq \vec{v}$ and by hypothesis "row $\vec{p}^{\,i}$ subsumes $\vec{p}^{\,j}$", we have:

$$P' \not\preceq v \Longrightarrow \vec{p}^{\,i} \not\preceq \vec{v} \Longrightarrow \vec{p}^{\,j} \not\preceq \vec{v}$$

Finally, for any defined value $\vec{v}$ we have: $P' \not\preceq \vec{v} \iff P \not\preceq \vec{v}$. □

Example 4

Let us consider the following matrix P.

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

Because the constructor of pairs "," is alone in its signature, pattern $(_,_)$ subsumes pattern $_$. Namely, any (defined) 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:

$$\mathcal{U}(P, \vec{q}) = \mathcal{U}(\begin{pmatrix} _ & (_,_) \end{pmatrix}, \vec{q}) = \mathcal{U}(\begin{pmatrix} (_,_) & _ \end{pmatrix}, \vec{q})$$

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 defined 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 j.

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 here for constructor signatures of size one, which, in that particular case, yields exactly one recursive call. We can thus approximate the cost of our first safeguard as a quadratic number of "ordinary" pattern operations whose cost is roughly linear in the size of input patterns (still disregarding or-patterns). As a conclusion, if this first safeguard saves us from exponential computations, 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 (see Section 4.2). Thus, we can attempt to

[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 $(_,_)$.

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 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. In fact, example I of the following section exhibits S^3 behavior without the second safeguard, whereas, with second safeguard enabled, it exhibits quasi-quadratic behavior.

7.3 Measures

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

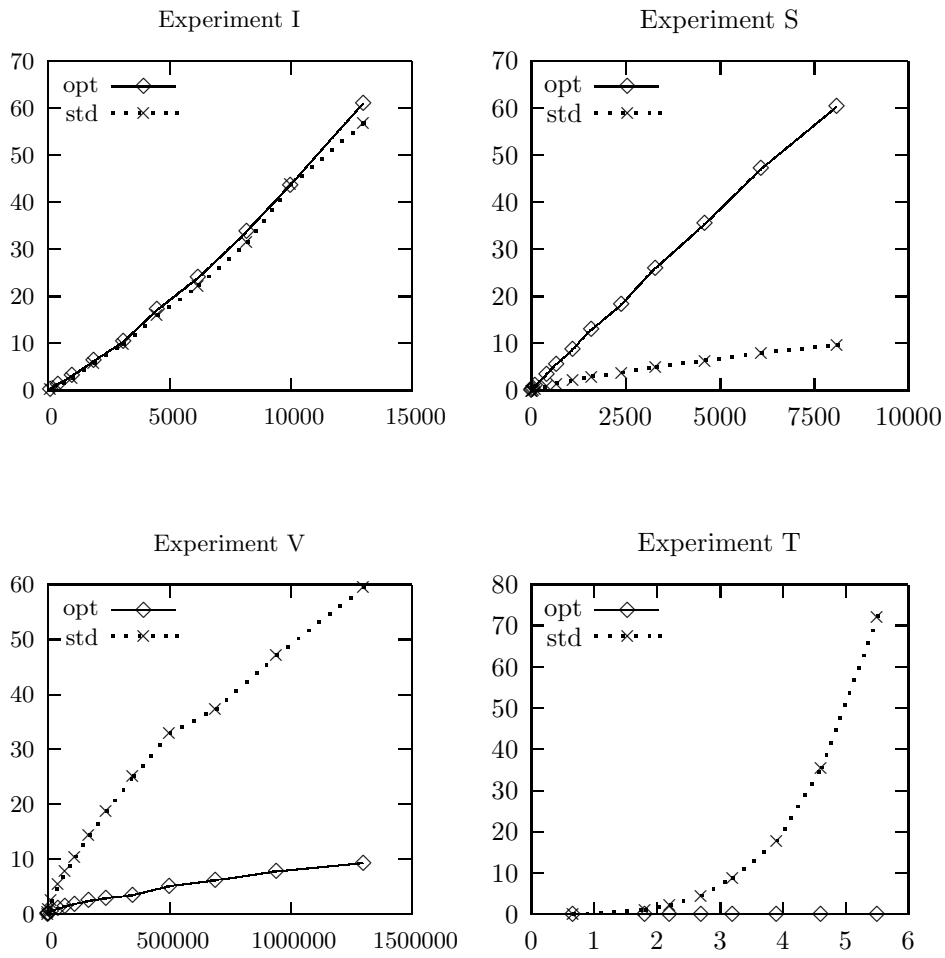
Measures apply to series of pattern-matching expressions of identical structure and increasing size. Three of our series T, S and V, may drive compilation of pattern matching to decision trees into producing code of exponential size. We selected those because algorithm $\mathcal{U}$ is similar to compilation of pattern matching to decision trees. The fourth series, I, is the matching of n constant constructors. We selected series I because of a real example that triggered excessive compilation times while we were implementing algorithm $\mathcal{U}$. All series are defined precisely in appendix A. In Figure 3, the X-axis shows the size of matchings squared, (expressed as source file size squared), while the Y-axis shows 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 161×320 matrix to reach one minute of analysis time in experiment S (with safeguards), whereas it takes only a 42×21 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

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



examples and on the studied domain. However, notice that experiment I suggests 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. Our choice follows the general philosophy behind preferring backtracking automata to decision trees: avoid exponential behaviors as much as possible.

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 called indices). Those algorithms are in fact very similar to our algorithm $\mathcal{U}$. Indeed, 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 sometimes recall that this compilation technique yields "non-exhaustive match" and "useless clause" warnings as by-products — see for instance (Baudinet & MacQueen, 1985; Aitken, 1992; Sestoft, 1996). However there are good reasons to adopt backtracking automata: a potentially exponential behavior is avoided, and, in practice, output code size is somehow reduced. In our opinion, designers that wish to adopt backtracking automata should not be prevented to do so by the question of pattern matching diagnostics. Hence, we consider diagnostics directly. As a consequence, our algorithm for producing diagnostics is not only independent from any compilation strategy, but is also proved correct with respect to several 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, thereby 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 matches. 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. Such efficiency stems from several causes. 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 *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 matches to be such errors.

Finally, our approach of studying pattern matching anomalies on the semantical level 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, A. W., & MacQueen, D. B. (1991). Standard ML of New Jersey. *International Symposium on Programming Language Implementation and Logic Programming*. Springer-Verlag. Lecture Notes in Computer Science 583.

Augustsson, L. (1985). Compiling Pattern Matching. *Functional Programming Languages and Computer Architecture*. Springer-Verlag. Lecture Notes in Computer Science 201.

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

Garrigue, J. (2004). Typing Deep Pattern-Matching in Presence of Polymorphic Variants. *JSSST Workshop on Programming and Programming Languages*.

Harper, R. W., Milner, R., & Tofte, M. (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. *Lisp and Functional Programming*. ACM press.

Pettersson, M. (1992). A Term Pattern-Match Compiler Inspired by Finite Automata Theory. *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. *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. *Dagstuhl Seminar on Partial Evaluation*. Springer-Verlag. Lecture Notes in Computer Science 1110.

Wadler, Philip. (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:

```
type t = A_0 | A_1 | ··· | A_{n-1}

let f = function
| A_0 -> 0
```

```
| A_1 -> 1
    ...
| A_{n-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 $[1 \dots n]$:

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

And:

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

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.

$$b^i_i = \texttt{B}, \quad b^i_j = _ \text{ otherwise}$$

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

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

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
```

This series is a real challenge to pattern matching compilers, especially to those that target decision trees: whatever column is selected, compilation will produce code of exponential size.

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

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

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
```

This series yields exponential behavior of naive compilation (along first column) to decisions trees.