Optimizing Pattern Matching

Fabrice Le Fessant, Luc Maranget

INRIA Roquencourt, B.P. 105, 78153 Le Chesnay Cedex, France
(Email: {*Fabrice.Le_fessant*, *Luc.Maranget*}@inria.fr)

ABSTRACT

We present improvements to the backtracking technique of pattern-matching compilation. Several optimizations are introduced, such as commutation of patterns, use of exhaustiveness information, and control flow optimization through the use of labeled static exceptions and context information. These optimizations have been integrated in the Objective-Caml compiler. They have shown good results in increasing the speed of pattern-matching intensive programs, without increasing final code size.

1. INTRODUCTION

Pattern-matching is a key feature of functionnal languages. It allows to discriminate between the values of a deeply structured type, binding subparts of the value to variables at the same time. ML users now routinely rely on their compiler for such a task, they write complicated, nested, patterns. And indeed, tranforming high-level pattern-matching into elementary tests is a compiler job. Moreover, by considering the matching as a whole and because it knows some intimate details on runtime issues such as the representation of values, compiler code is often better than human code, both as regards compactness and efficiency.

There are two approaches to pattern-matching compilation, the underlying model being either decision trees [5] or backtracking automata [1]. Using decision trees, one produces *a priori* faster code (because each position in a term is tested at most once), while using backtracking automata, one produces *a priori* less code (because patterns never get copied, hence never get compiled more than once). The price paid in each case is losing the advantage given by the other technique.

This paper mostly focuses on producing faster code in the backtracking framework. Examining the code generated by the Objective-Caml compiler [11], which basically used the original algorithm of [1], on small frequently found programs, such as a list-merge function, or on large examples [13], we found that the backtracking scheme could still be improved.

Our optimizations improve the produced backtracking automaton by grouping elementary tests more often, removing useless tests and avoiding the blind backtracking behavior of previous schemes. To do so, the compiler uses new informations and outputs a new construct. New informations are incompatibility between patterns, exhaustiveness information and contextual information at the time of backtracking. As to the new construct, previous schemes used a lone "exit" construct whose effect is to jump to the nearest enclosing "trap-handler" ; we enrich both exits and traps-handlers with labels, resulting in finer control on execution flow.

Our optimizations also apply to or-patterns, a conveninent feature to group clauses with identical actions. Unsharing of actions is avoided by using our labelled exit construct. As or-patterns may contain variables, the exit construct is also extended in the direction of taking arguments.

All our optimizations are now implemented in the latest version of the Objective-Caml compiler, whose language of accepted patterns has been extended by allowing variables in or-patterns.

The structure of this article is the following: we first introduce some theoretical basics on pattern-matching in section 2 and describe the compilation scheme to backtracking automata in section 3. Then, we briefly introduce our optimizations and or-pattern compilation in an intuitive way in sections 4 and 5, while section 6 is a formalization of our complete compilation scheme. Finally, some experimental results are shown in section 7, and a comparison with other approaches is discussed in section 8.

2. BASICS

In this section, we introduce some notations and definitions. Most of the material here is folklore, but for, perhaps, or-patterns.

2.1 Patterns and Values

ML is a typed language, where new types of values can be introduced using *type definitions* such as:

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

This definition introduces a type `t`, with three *constructors* that build values of type `t`. These three constructors define the *complete signature* of type `t`. Every constructor has an arity, i.e. the number of arguments it takes. Here arity of `Nil` is zero, while the arities of `One` and `Cons` are one and two respectively. A constructor of arity zero is called a *constant* constructor, while other constructors are *non-constant* constructors.

Most native data types in ML – such as integers, records, arrays, tuples – can be seen as particular instances of such type definitions. For example, in the following we will consider lists (*nil* being the constant constructor `[]` and *cons* the infix constructor `::`), and tuples (the type of n-tuples defines one constructor of arity n, pairs being written with the infix constructor "`,`"). For our purpose, integers are constant constructors, and the signature of the integer type is infinite.

More formally, *patterns* and *values* are defined as follows:

$p ::=$		**Patterns**
	$\_$	wildcard
	x	variable
	$c(p_1, p_2, \ldots, p_a)$	constructor pattern
	$(p_1 \mid p_2)$	or-pattern
$v ::=$		**Values**
	$c(v_1, v_2, \ldots, v_a)$	constructor value

In the following, we freely replace variables-patterns by wildcards "$\_$" when their names are irrelevant. While describing compilation, convenient tools are vectors of values ($\vec{v} = (v_1\ v_2 \ldots v_n)$ and $\vec{v}_{n \leftrightarrow m} = (v_n ... v_m)$), vectors of patterns ($\vec{p} = (p_1\, p_2 \ldots p_n)$ and $\vec{p}_{n \leftrightarrow m} = (p_n ... p_m)$) and matrices of patterns ($P = (p^i_j)$).

In this paper, we present pattern-matching compilation as a transformation on an intermediate code in the compiler, called *lambda-code*. Here, another useful object is the *clause matrix* $(P \rightarrow L)$:

$$(P \rightarrow L) = \begin{pmatrix} p^1_1 & p^1_2 & \cdots & p^1_n & \rightarrow & l^1 \\ p^2_1 & p^2_2 & \cdots & p^2_n & \rightarrow & l^2 \\ & & \vdots & & & \\ p^m_1 & p^m_2 & \cdots & p^m_n & \rightarrow & l^m \end{pmatrix}$$

Which associates rows of patterns $(p^i_1\ p^i_2 \ldots p^i_n)$ to lambda-code actions l^i.

2.2 Pattern Matching in ML

A pattern can be seen as representing a set of values sharing a common prefix.

DEFINITION 1 (INSTANCE). *Let p be a pattern and v be a value belonging to a common type. The value v is an* instance *of the pattern p or p* matches *v, written $p \preceq v$ when one of the following rules apply:*

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

Seeing a pattern as the set of its instances, it is clear that or-patterns express set union.

In ML, patterns are a binding construct, more specifically, a successful match $p \preceq v$, binds the variables of p to some subterms of v. Such bindings can be computed while checking that p matches v, provided that the following set $\mathcal{V}(p)$ of variables defined by p is well-defined:

$$\begin{array}{rcl} \mathcal{V}(\_) & = & \emptyset \\ \mathcal{V}(x) & = & \{x\} \\ \mathcal{V}(c(p_1, \ldots, p_a)) & = & \mathcal{V}(p_1 \ldots p_a) \\ \mathcal{V}(p_1 \ldots p_a) & = & \mathcal{V}(p_1) \cup \ldots \cup \mathcal{V}(p_a) \\ & & \text{if for all } i \neq j, \mathcal{V}(p_i) \cap \mathcal{V}(p_j) = \emptyset \\ \mathcal{V}(p_1 \mid p_2) & = & \mathcal{V}(p_1), \ \text{ if } \mathcal{V}(p_1) = \mathcal{V}(p_2) \end{array}$$

The first "if" condition above is the linearity of patterns. The second condition is specific to or-patterns, it means that matching by either side of the or-pattern binds the same variables.

We then define the now dominant, textual priority scheme to disambiguate the case when several rows in a matrix match:

DEFINITION 2 (MATCHING PREDICATE). *Let P be a pattern matrix and $\vec{v} = (v_1 \ldots v_n)$ be a value vector. The value v* matches *line number i in P, if and only if the following two conditions are satisfied:*

- $(p^i_1 \ldots p^i_n) \preceq (v_1 \ldots v_n)$
- $\forall j < i,\ (p^j_1 \ldots p^j_n) \not\preceq (v_1 \ldots v_n)$

We will not give a full semantics for evaluating pattern-matching expressions, and more generally lambda-code. Intuitively, given a clause matrix $P \rightarrow L$ and a value vector $\vec{v}$ such that line number i in P matches $\vec{v}$, evaluating the matching of $\vec{v}$ by $P \rightarrow L$ in some environment ρ is evaluating l^i in ρ extended by the bindings introduced while matching $\vec{v}$ by $(p^i_1 \ldots p^i_n)$. If $\vec{v}$ is not matched by any line in P, we say that the pattern-matching P *fails*. If no such $\vec{v}$ exists, then the pattern-matching is said *exhaustive*.

Like pattern vectors, pattern matrices represent sets of value vectors. More specifically, when some line in P matches $\vec{v}$ we simply say that P matches $\vec{v}$. This looks obvious, but representing sets using matrices is at the core of our optimizations. One easily checks that the instances of P are the union of the instances of the lines of P. That is, when considering matrix globally, the second condition in definition 2 above is irrelevant. More important, row order is also irrelevant.

Finally, the instance relation induces relations on the patterns themselves.

DEFINITION 3 (RELATIONS ON PATTERNS). *We define the following three relations:*

1. *Pattern p is* less precise *then pattern q, written $p \preceq q$, when all instances of q are instances of p.*
2. *Pattern p and q are* equivalent, *written $p \equiv q$, when their instances are the same.*
3. *Patterns p and q are* compatible *when p and q share a common instance.*

Here some remarks are to be made. Because of typing, checking the precision relation is neither obvious nor cheap. Slight complications appear when q is a variable. For instance, `([]|_::_)` $\preceq$ `_` holds, while `(Nil|One _)` $\preceq$ `_` does not. Or-patterns are not responsible for this complication, since we also have `(_,_)` $\preceq$ `_`. In such cases one should "expand" p and consider, whether signatures are complete or not (see [12, Section 5.1]). By contrast, compatibility can be checked by a simple recursive algorithm. When compatible, patterns p and q admit a least upper bound, written $p \uparrow q$, which can be computed while checking compatibility:

$$\begin{cases} (p_1 \ldots p_a) \uparrow (q_1 \ldots q_a) & = & (p_1 \uparrow q_1 \ldots p_a \uparrow q_a) \\ \_ \uparrow q & = & q \\ p \uparrow \_ & = & p \\ c(p_1, \ldots, p_a) \uparrow c(q_1, \ldots, q_a) & = & c(r_1, \ldots, r_a) \\ \quad \text{where } (r_1 \ldots r_a) \text{ is } (p_1 \ldots p_a) \uparrow (q_1 \ldots q_a) \end{cases}$$

With the following additional rules for or-patterns:

$$(p_1 \mid p_2) \uparrow q = \begin{cases} p_1 \uparrow q, & \text{when } p_2 \text{ and } q \text{ not compatible} \\ p_2 \uparrow q, & \text{when } p_1 \text{ and } q \text{ not compatible} \\ (p_1 \uparrow q \mid p_2 \uparrow q), & \text{otherwise} \end{cases}$$
$$p \uparrow (q_1 \mid q_2) = (q_1 \mid q_2) \uparrow p$$

Proving that $p \uparrow q$ is indeed the least upper bound of p and q is easy, by considering patterns as sets of their instances. Note that $p \uparrow q$ is defined up to $\equiv$-equivalence, and that it encodes instance intersection.

3. COMPILATION

In this section, we present a backtracking compilation scheme close to the one described in [19, 1], and implemented in compilers such as the `hbc` compiler or the Objective Caml compiler. This classical scheme will be refined later into an optimized scheme, using same notations and concepts.

3.1 Output of the match compiler

The output of the compilation function, $\mathcal{C}$, consists of a combination of the actions in the pattern matching, using the following constructs:

- Let-bindings: `let` $(x\ l_x)\ l$, nested let-bindings are abbreviated as:

  `(let` $(x_1\ l_1)\ (x_2\ l_2)\ \cdots\ (x_n\ l_n)\ l$`)`

- Static exceptions, `exit` and traps, `catch` l_1 `with` l_2. If, when evaluating the body l_1, `exit` is encountered, then the result of evaluating `catch` l_1 `with` l_2 is evaluating the handler l_2, otherwise it is evaluating l_1. By contrast with dynamic exceptions, static exceptions are directly compiled as jumps to the associated handlers (plus some environment adjustment, such as stack pops), whereas traps do not generate any code.

- Switch constructs:

  `switch` l `with`
  `case` c_1`:` $l_1 \cdots$ `case` c_k`:` l_k
  `default:` d

  The result of a switch construct is the evaluation of the l_i, corresponding to the constructor c_i appearing as the head of the value v of l. If the head constructor of v doesn't appear in the case list, the result is the evaluation of the default d expression.

  The default clause `default:` d can be omitted, in such a case the switch behavior is unspecified on non-recognized values. Scheme $\mathcal{C}$ can thus omit the default clause when it is known that case lists will cover all possibilities at runtime. We use the keyword `switch*` to highlight switch constructs with no default clause.

  Those switch constructs are quite sophisticated, they compile later into more basic constructs: tests and branchs and jump tables. We in fact modified the Objective Caml compiler to improve the compilation of switch constructs, using techniques first introduced in the context of compiling the case statement of Pascal [3]. The key points are using range tests, which can typically be performed by one single (unsigned) test and branch plus possibly one addition, cutting sparse case lists into denser ones, and deciding which of jump tables or test sequence is more appropriate in every situation. A survey of these techniques can be found in [18].

- Accessors: `field` $n\ x$, where x is a variable and n is an integer offset. By convention, the first argument of non-constant constructors stands at offset zero.

- Sequences: l_1`;` l_2 and units: `()`

3.2 Initial state

Input to the pattern matching compiler $\mathcal{C}$ consists of two arguments: a vector of variables $\vec{x}$ of size n and a clause matrix $P \to L$ of width n and height m.

$$\vec{x} = (x_1\, x_2 \ldots x_n), \quad P \to L = \begin{pmatrix} p_1^1 & p_2^1 & \cdots & p_n^1 & \to & l^1 \\ p_1^2 & p_2^2 & \cdots & p_n^2 & \to & l^2 \\ & & \vdots & & & \\ p_1^m & p_2^m & \cdots & p_n^m & \to & l^m \end{pmatrix}$$

The initial matrix is generated from source input. Given a pattern-matching expression (in Caml syntax):

`match` x `with |` p^1`->` e^1`|` p^2`->` $e^2 \ldots$ `|` p^m`->` e^m

The initial call to $\mathcal{C}$ is:

`catch`
$$\mathcal{C}((x), \begin{pmatrix} p^1 & \to & l^1 \\ p^2 & \to & l^2 \\ & \to & \\ p^m & \to & l^m \end{pmatrix})$$
`with (failwith "Partial match")`

Where the l^i's are the translations to lambda-code of the e^i's, and `(failwith "Partial match")` is a runtime failure that will occur when the whole pattern matching fails.

3.3 Classical scheme

By contrast with previous presentations, we assume that matrix $P \to L$ has at least one row (i.e. $m > 0$). This condition simplifies our presentation, without restricting its generality. Hence, scheme $\mathcal{C}$ is defined by cases on non-empty clause matrices:

1. If n is zero (i.e. when there are no more columns), then the first row of P matches the empty vector ():

$$\mathcal{C}((), \begin{pmatrix} \to & l^1 \\ \to & l^2 \\ & \vdots \\ \to & l^m \end{pmatrix}) = l^1$$

2. If n is not zero, then a simple compilation is possible, using the following four rules.

(a) If all patterns in the first column of p are variables, $y^1, y^2, \ldots, y^m$, then:

$$\mathcal{C}(\vec{x}, P \to L) = \mathcal{C}((x_2\ x_3 \ldots x_n), P' \to L')$$

where

$$P' \to L' = \begin{pmatrix} p_2^1 & \cdots & p_n^1 & \to & \texttt{let (}y^1\ x_1\texttt{)}\ l^1 \\ p_2^2 & \cdots & p_n^2 & \to & \texttt{let (}y^2\ x_1\texttt{)}\ l^2 \\ & \vdots & & & \\ p_2^m & \cdots & p_n^m & \to & \texttt{let (}y^m\ x_1\texttt{)}\ l^m \end{pmatrix})$$

We call this rule, the *variable* rule. This case also handles wildcard patterns: they are treated like variables except for the let-binding on actions.

(b) If all patterns in the first column of P are constructor patterns $c(q_1, \ldots, q_a)$. Let $C = \{c_1, c_2, \ldots, c_k\}$ be the set of matched constructors. That is the set of head constructors of the p_1^i's.

Then, for a given constructor c in C, we define the *specialized* clause matrix $\mathcal{S}(c, P \to L)$ by mapping the following transformation on the rows of P.

p_1^i	$\mathcal{S}(c, P \to L)$
$c(q_1^i, \ldots, q_a^i)$	$q_1^i \cdots q_a^i \quad p_2^i \cdots p_n^i \to l_i$
$c'(q_1^i, \ldots, q_{a'}^i)\ (c' \neq c)$	No row

(Matrices $\mathcal{S}(c, P \to L)$ and $P \to L$ define the same matching predicate when x_1 is bound to some value $c(v_1, \ldots, v_a)$.) Furthermore, for a given constructor c of arity a, let $y_1, \ldots, y_a$ be fresh variables. Then, for any constructor c in C, we define the lambda-expression $r(c)$:

$$\begin{array}{l} \texttt{(let (}y_1\ \texttt{(field 0 }x_1\texttt{))} \\ \quad \ldots \\ \quad \texttt{(}y_a\ \texttt{(field (}a-1\texttt{) }x_1\texttt{))} \\ \quad \mathcal{C}((y_1, \ldots, y_a, x_2, \ldots, x_n), \mathcal{S}(c, P \to L))) \end{array}$$

Finally, compilation result is:

$$\begin{array}{l} \texttt{switch }x_1\texttt{ with} \\ \quad \texttt{case }c_1\texttt{: }r(c_1)\cdots\texttt{case }c_k\texttt{: }r(c_k) \\ \quad \texttt{default: exit} \end{array}$$

(Note that the defaut clause can be omitted when C makes up a full signature.) We call this rule, the *constructor* rule.

(c) If P has only one row and that this row starts with an or-pattern:

$$P = (\ (q_1 \mid \ldots \mid q_o)\ p_2\ \cdots\ p_n\ \to\ l\ ),$$

Then, compilation result is:

$$\mathcal{C}((x_1), \begin{pmatrix} q_1 & \to & () \\ & \vdots & \\ q_o & \to & () \end{pmatrix}); \mathcal{C}((x_2 \ldots x_n), (p_2 \ldots p_n \to l))$$

This rule is the *orpat* rule. Observe that it does not duplicate any pattern nor action. However, variables in or-patterns are not supported, since, in clause $q_i \to ()$, the scope of q_i variables is the action "()".

(d) Finally, if none of the previous rules applies, the clause matrix $P \to L$ is cut in two clause matrices $P_1 \to L_1$ and $P_2 \to L_2$, such that $P_1 \to L_1$ is the largest clause matrix on which one of the variable, contructor or orpat rule applies.

Then, compilation result is:

$$\texttt{catch }\mathcal{C}(\vec{x}, P_1 \to L_1)\texttt{ with }\mathcal{C}(\vec{x}, P_2 \to L_2)$$

This rule is the *mixture* rule.

This paper doesn't deal with optimizing let-bindings, which are carelessly introduced by scheme $\mathcal{C}$. This job is left to a later compilation phase.

4. OPTIMIZATIONS

We now describe some improvment on the previously described classical compilation scheme. For simplicity, we present examples and defer the full presentation of our scheme to section 6.

4.1 Optimizing the mixture rule

In this section and in the following, our running example is the classical list-merge:

```
let merge lx ly = match lx,ly with
| [], _ -> 1
| _, [] -> 2
| x::xs, y::ys -> 3
```

(We focus on pattern-matching compilation, replacing potentially arbitrary action code by integers).

Such a matching on pairs encodes matching on two arguments and the initial call to scheme $\mathcal{C}$ will be:

$$\mathcal{C}((\texttt{lx ly}), (P \to L))$$

Where $(P \to L)$ is:

$$(P \to L) = \begin{pmatrix} \texttt{[]} & \texttt{\_} & \to & \texttt{1} \\ \texttt{\_} & \texttt{[]} & \to & \texttt{2} \\ \texttt{x::xs} & \texttt{y::ys} & \to & \texttt{3} \end{pmatrix}$$

Applying the mixture rule twice yields three matrices:

$$\begin{array}{l} P_1 \to L_1 = (\ \texttt{[]}\quad \texttt{\_}\ \to\ \texttt{1}\ ) \\ P_2 \to L_2 = (\ \texttt{\_}\quad \texttt{[]}\ \to\ \texttt{2}\ ) \\ P_3 \to L_3 = (\ \texttt{x::xs}\quad \texttt{y::ys}\ \to\ \texttt{3}\ ) \end{array}$$

```
catch
  (catch
    (switch lx with case []: 1
       default: exit)
   with (catch
    (switch ly with case []: 2
       default: exit)
   with (catch
    (switch lx with
       case (::):
         (switch ly with
            case (::) : 3
            default: exit)
       default: exit))))
with (failwith "Partial match")
```

```
catch
  (catch
    (switch* lx with
      case []: 1
      case (::) :
        (switch ly with
           case (::): 3
           default: exit))
  with
    (switch ly with
       case []: 2
       default: exit)
with (failwith "Partial match")
```

Figure 1: Mixture optimization

Now, consider another clause matrix $(P' \to L')$:

$$(P' \to L') = \begin{pmatrix} \texttt{[]} & \_ & \to & 1 \\ \texttt{x::xs} & \texttt{y::ys} & \to & 3 \\ \_ & \texttt{[]} & \to & 2 \end{pmatrix}$$

Both clause matrices define the same matching function, namely they both map ([] v) to 1, (v_1::v_2 []) to 2 and (v_1::v_2 v_1'::v_2') to 3. Furthermore, $(P' \to L')$ can be obtained from $(P \to L)$ by swapping its second and third row. More generally, one easily checks that swapping two contiguous *incompatible* rows is legal. Then, applying the mixture rule to $(P' \to L')$, yields two matrices only:

$$P_1' \to L_1' = \begin{pmatrix} \texttt{[]} & \_ & \to & 1 \\ \texttt{x::xs} & \texttt{y::ys} & \to & 3 \end{pmatrix},$$
$$P_2' \to L_2' = \begin{pmatrix} \_ & \texttt{[]} & \to & 2 \end{pmatrix}$$

Final outputs for $P \to L$ and $P' \to L'$ are displayed on Figure 1. Hence, as a result of replacing $P \to L$ by $P' \to L'$, the two tests on `lx` that were performed separately on the left code are now merged in a single switch in the right code. Also notice that one trap disappears.

More generally, an optimized mixture rule should take advantage of pattern-matching semantics to swap rows when possible, so that as few cuts as possible are performed.

4.2 Using exhaustiveness information

The Objective Caml compiler checks the exhaustiveness of pattern matching expressions and issues a warning before compiling non-exhaustive pattern matchings. However, the exhaustiveness information can also be used for avoiding tests. Matrix P' of the previous section is exhaustive, this means that there will be no `"Partial match"` failure at runtime. As an immediate consequence, the switch: `(switch ly with case []: 2 default: exit)` always succeeds (this switch is the last one performed by the optimzed code in figure 1). Thus, we replace it by `2`. We can also supress the outermost trap. Hence, cumulating both optimizations described up to now, compilation of $P \to L$ finally yields:

```
catch
  (switch* lx with
    case []: 1
    case (::): (switch ly with
                  case []: 3
                  default: exit))
with 2
```

In the general case, benefiting from exhaustiveness information is achieved by slightly modifying scheme $\mathcal{C}$. It suffices to avoid emitting default clauses in switch constructs, when it is known that no exit should escape from produced code. This property holds initially for exhaustive pattern matchings, and transmits to all recursive calls, except for the call on $P_1 \to L_1$ in the mixture rule.

4.3 Optimizing exits

The two previous opimizations yield optimal code for the merge example. Hence we complicate the running example by considering a matching on objects of type `t` from section 2:

$$P \to L = \begin{pmatrix} \texttt{Nil} & \_ & \to & 1 \\ \_ & \texttt{Nil} & \to & 2 \\ \texttt{One x} & \_ & \to & 3 \\ \_ & \texttt{One y} & \to & 4 \\ \texttt{Cons (x,xs)} & \texttt{Cons (y,ys)} & \to & 5 \end{pmatrix}$$

The optimized mixture rule yields four matrices:

$$P_1 \to L_1 = \begin{pmatrix} \texttt{Nil} & \_ & \to & 1 \\ \texttt{Cons (x,xs)} & \texttt{Cons (y,ys)} & \to & 5 \end{pmatrix}$$
$$P_2 \to L_2 = \begin{pmatrix} \_ & \texttt{Nil} & \to & 2 \end{pmatrix}$$
$$P_3 \to L_3 = \begin{pmatrix} \texttt{One x} & \_ & \to & 3 \end{pmatrix}$$
$$P_4 \to L_4 = \begin{pmatrix} \_ & \texttt{One x} & \to & 4 \end{pmatrix}$$

For reasons that will appear immediately, we apply the mixture rule from bottom to top, thereby nesting trap handlers. The match being exhaustive, compilation yields the code displayed on the left part of Figure 2.

Now, consider what happens at run-time when (`lx ly`) is (`Cons` (v_1, v_2) `One` v). A first switch on `lx` leads to line 7, where a switch on `ly` is performed. This switch

```
1  catch
2    (catch
3       (catch
4         (switch lx with
5           case Nil: 1
6           case Cons:
7             (switch ly with
8                case Cons: 5
9                default: exit)
10
11          default: exit)
12      with
13        (switch ly with
14          case Nil: 2
15          default: exit))
16   with
17     (switch lx with
18       case One: 3
19       default: exit))
20 with 4
```

Unoptimized code

```
1  catch
2    (catch
3       (catch
4         (switch lx with
5           case Nil: 1
6           case Cons:
7             (switch* ly with
8                case Cons: 5
9                case Nil: (exit 2)
10               case One: (exit 4))
11          default: (exit 2))
12      with (2)
13        (switch ly with
14          case Nil: 2
15          default: (exit 3)))
16   with (3)
17     (switch lx with
18       case One: 3
19       default: (exit 4)))
20 with (4) 4
```

Optimized code

Figure 2: Exit optimization

fails, and the default action jumps to the nearest enclosing handler (line 13), where `ly` is tested against `Nil` resulting in another switch failure. Here, in our case, control goes to line 17, where another switch on `lx` (against `One x`) fails, resulting in final jump to line 20.

Hence, it would be appropriate to jump to line 20 right from the first test on `ys`. To do so, both exits and trap handlers are now *labelled* by integers. Note that this new feature does not really complicate the compilation of static exceptions. Then, it becomes possible to jump to different trap handlers from the same point and a better compilation of $P \to L$ is displayed in the right part of figure 2.

The code above maps vectors (`Cons` (v_1`,` v_2) `One` v) to 4 by executing two switches, while previous code needed four switches to perform the same task. Hence, exit optimization has a noticeable benefit as regards run-time efficiency. As regards code size, exit optimization may increase it, since some switches may have larger case lists. However, code size remains under control, since no extra switches are generated. Hence, final code size critically depends on how switches translate to machine-level constructs. For instance, machine-level code size obviously does not increase when switches translate to jump tables[1].

Surprisingly, performing exit optimization is quite simple and cheap: the needed information is available at compile-time by inspecting pattern matrices only. Reachable trap handlers are defined as pairs (P, e) of a pattern matrix and an integer. Reachable trap handlers originate from the division performed by the mixture rule. Here, $P_1 \to L_1$ is compiled with the reachable trap-handlers $(P_2, 2)$, $(P_3, 3)$ and $(P_4, 4)$. Then, the constructor rule specializes reachable trap handlers. Here, in the case where `lx` is `Cons` (v_1`,` v_2), specializing reachable trap handlers results in ((`Nil`), 2) and ((`One y`), 4) (note that specializing P_3 yields an empty matrix, which is discarded). Hence, while generating the first switch on `ly` (line 7), it is known that the code produced by compiling trap handler number 2 and 3 will surely exit when `ly` is `One` v, and a jump to trap handler number 4 can be generated by the compiler in that case.

4.4 Aggressive control flow optimization

The code produced by exit optimization still contains redundant tests, some of which can be removed without altering the handler structure introduced by the mixture rule. More specifically, we consider trap handler number 3 (line 16). It results from compiling P_3 and is a switch of `lx` against `One`.

The only `(exit 3)` lies in trap handler number 2 (line 15) and results from `ly` not being `Nil`, this gives us no direct information on `lx`. Now, looking upwards for `(exit 2)`, we can infer that trap handler number 2 is entered from two different points. In the first case (line 9), (`lx ys`) is fully kown as (`Cons` (v_1`,` v_2) `Nil`), in the second case (line 11), only `lx` is know to be `One` v. As `(exit 3)` on line 15 gets executed only when `ly` is not `Nil`, we can finally deduce that the first case never results in entering trap handler number 3. As a consequence, trap handler number 3 is executed in a context where `lx` necessarily is `One` v, the switch it performs is useless and line 16 can be simplified into "`3`". This elimination of useless tests[4] is usually performed at a lower level by combining dead code elimination[9] and conditional constant propagation[20, 6].

[1]Here, given the Objective Caml encoding of constructors, we are in the same desirable situation where the compilation of apparently larger switches does not result in producing more code.

Finally, after all optimizations, there remains one redundant switch in produced code, in trap-handler number 2 (line 12). As a result, vectors (`Cons` (v_1, v_2) `Nil`) are mapped to 2 by testing `ly` twice. One should notice that this is precisely the test that would get duplicated by compilation to decision trees.

Describing what is known on values while entering trap handlers is slightly involved. The key idea is representing set of value vectors as pattern matrices. We call such a set a *context*. Contexts for the three trap handlers of our example are:

Trap number	Context
2	$\begin{pmatrix} \texttt{One \_} & \texttt{\_} \\ \texttt{Cons (\_, \_)} & \texttt{Nil} \end{pmatrix}$
3	$\begin{pmatrix} \texttt{One \_} & \texttt{(One \_} \mid \texttt{Cons (\_, \_))} \end{pmatrix}$
4	$\begin{pmatrix} \texttt{Cons (\_, \_)} & \texttt{One \_} \end{pmatrix}$

If precise enough and exploited fully, we conjecture that contexts subsume exhaustiveness information. However as intuition suggests and experience confirms, contexts get larger while compilation progresses, potentially reaching huge sizes at the end of matrices. We cure this by safely over-approximating contexts when they get too large, replacing some patterns in them by wildcards. Hence the optimizations of section 4.2 is still worth considering, as being cheap and always applicable.

5. COMPILING OR-PATTERNS

Until now, code produced for or-patterns is uneefficient, because only one or-pattern can be compiled at a time, requiring multiple applications of the mixture rule before and after each or-pattern. Thanks to integer labelled exits, one easily avoids dividing matrices before or-patterns. Consider a "*car*" function for our three-constructors list:

```
let car list = match list with
| Nil -> -1
| (One x | Cons (x,_)) -> x
```

(In this example, we also show how to compile or-patterns with variables.)

Compilation proceeds by allocating a new trap-handler number 2 and expanding the clause "`One x | Cons (x,_)`" into two clauses with patterns "`One x`" and "`Cons (x,_)`". Actions for the new clauses are exits to 2:

$$\begin{array}{l} \texttt{catch} \\ \quad \mathcal{C}((\texttt{list}), \begin{pmatrix} \texttt{Nil} & \to & \texttt{-1} \\ \texttt{One x'} & \to & \texttt{(exit 2 x')} \\ \texttt{Cons (x', \_)} & \to & \texttt{(exit 2 x')} \end{pmatrix}) \\ \texttt{with (2 x)}\ \mathcal{C}((), (\ \to\ \texttt{x}\ )) \end{array}$$

Note that both exits and trap handlers now take yet another extra argument, the ocurrences of `x'` in exits are non-binding and refer to pattern variables, while the occurence of `x` in handler is binding. This new construct allows the compilation of or-patterns with variables. Implementation is not very tricky: the `catch` ... `with (2 x)` ... construct allocates one mutable variable, exits update this variable, which is read before entering the handler. In a native code compiler, such a variable is a temporary and ultimately a machine register. Finally, the generated lambda-code is as follow:

```
catch
  switch* list with
    case Nil: -1
    case One: (exit 2 (field 0 list))
    case Cons: (exit 2 (field 0 list))
with (2 x) x
```

Moreover, by the semantics of pattern-matching, cuts after or-patterns can also be avoided in many situations. In the case of one column matrices, where the expanded or-patterns express the full matching performed, all cuts can be avoided. Things get a bit more complicated when matrices have more than one column. Consider the following clause matrix,

$$P \to L = \begin{pmatrix} \texttt{(1|2)} & p_2 & \to & l^1 \\ \texttt{(3|4)} & q_2 & \to & l^2 \end{pmatrix}$$

We further assume a match on (`x y`) and that match failure should result in `(exit 1)` (the static exception label corresponding to match failure can be given as a third argument to the compilation scheme). Writting p_1 = `(1|2)` and q_1 = `(3|4)`, there are obviously no value vectors (v_1 v_2) such that v_1 is an instance of both p_1 and q_1. As a consequence, the following compilation is correct:

$$\begin{array}{l} \texttt{catch} \\ \quad \texttt{(catch} \\ \qquad \texttt{(switch x with} \\ \qquad\quad \texttt{case 1: (exit 2) case 2: (exit 2)} \\ \qquad\quad \texttt{case 3: (exit 3) case 4: (exit 3)} \\ \qquad\quad \texttt{default: (exit 1))} \\ \quad \texttt{with (2)}\ \mathcal{C}((\mathrm{y}), (\ p_2\ \to\ l^1\ ), 1)) \\ \texttt{with (3)}\ \mathcal{C}((\mathrm{y}), (\ q_2\ \to\ l^2\ ), 1) \end{array}$$

Intuitively, once `x` is checked, the choice beetween first and second row is made. Depending on the value of `y`, matching may still fail, but then, the whole matching fails. Conversely, matrix division cannot be avoided when matching by p_1 does not exclude matching by q_1, that is, when p_1 and q_1 are compatible. This is the case, for instance, when p_1 = `(1|2)` and q_1 = `(2|3)`. Then, a correct compilation is:

$$\begin{array}{l} \texttt{catch} \\ \quad \texttt{(catch} \\ \qquad \texttt{(switch x with} \\ \qquad\quad \texttt{case 1: (exit 2) case 2: (exit 2)} \\ \qquad\quad \texttt{default: (exit 3))} \\ \quad \texttt{with (2)}\ \mathcal{C}((\mathrm{y}), (\ p_2\ \to\ l^1\ ), 3)) \\ \texttt{with (3)} \\ \quad \texttt{(catch} \\ \qquad \texttt{(switch x with} \\ \qquad\quad \texttt{case 2: (exit 4) case 3: (exit 4)} \\ \qquad\quad \texttt{default: (exit 1))} \\ \quad \texttt{with (4)}\ \mathcal{C}((\mathrm{y}), (\ q_2\ \to\ l^2\ ), 1)) \end{array}$$

Note that the third argument to the first recursive call to the compilation scheme is "3" and not "1". As a consequence, vectors (2 v_2) such that p_2 does not match v_2 while q_2 matches v_2 get mapped correctly to l^2.

Figure 3: Operations on contexts

(a) Context specialization

$P \bullet Q$ row	$\mathcal{S}(c, P \bullet Q)$ row
$p_1^i \cdots p_k^i \bullet c(q_1^i, \ldots, q_a^i) \cdots q_n^i$	$p_1^i \cdots p_k^i \; c(\_, \ldots, \_) \bullet q_1^i \cdots q_a^i \; q_{a+1}^i \cdots q_n^i$
$p_1^i \cdots p_k^i \bullet \_ \cdots q_n^i$	$p_1^i \cdots p_k^i \; c(\_, \ldots, \_) \bullet \_ \cdots \_ \; q_{a+1}^i \cdots q_n^i$
$p_1^i \cdots p_k^i \bullet c'(q_1^i, \ldots, q_a^i) \cdots q_n^i$	no row

(b) Context collection

$P \bullet Q$ row	$\mathcal{COL}(P \bullet Q)$ row
$p_1^i \cdots p_{k-1}^i \; c(\_, \ldots, \_) \bullet q_1^i \cdots q_a^i \; q_{a+1}^i \cdots q_n^i$	$p_1^i \cdots p_k^i \bullet c(q_1^i, \ldots, q_a^i) \cdots q_n^i$

(c) Context pushing and popping

$P \bullet Q$ row	$\Downarrow(P \bullet Q)$ row	$\Uparrow(P \bullet Q)$ row
$p_1^i \cdots p_k^i \bullet q_1^i \cdots q_n^i$	$p_1^i \cdots q_1^i \bullet q_2^i \cdots q_n^i$	$p_1^i \cdots p_{k-1}^i \bullet p_k^i \cdots q_n^i$

6. OUR COMPILATION SCHEME

6.1 Initial state

The new scheme $\mathcal{C}^*$ takes five arguments and a typical call is $\mathcal{C}^*(\vec{x}, P \to L, ex, def, ctx)$, where $\vec{x} = (x_1 \ldots x_n)$ and $P \to L$ is a clause matrix of width n:

$$P \to L = \begin{pmatrix} p_1^1 & \cdots & p_n^1 & \to & l^1 \\ p_1^2 & \cdots & p_n^2 & \to & l^2 \\ & & \vdots & & \\ p_1^m & \cdots & p_n^m & \to & l^m \end{pmatrix}$$

Extra arguments are:

- The exhaustiveness argument ex is either *partial* or *total* depending on whether compilation can produce escaping `exit` constructs or not.
- Reachable trap handlers def are sequences $(P_1, e_1); \cdots; (P_t, e_t)$, where the e_i's are integers (trap handler numbers) and the P_i's are pattern matrices of width n.
- The context ctx is a pattern matrix of width $k + n$, equivalent to a pair of matrixes $P \bullet Q$, where each row is divided into a prefix (in P) of width k and a fringe (in Q) of width n.

$$P \bullet Q = \begin{pmatrix} p_1^1 & \cdots & p_k^1 & \bullet & q_1^1 & \cdots & q_n^1 \\ p_1^2 & \cdots & p_k^2 & \bullet & q_1^2 & \cdots & q_n^2 \\ & & & \vdots & & & \\ p_1^m & \cdots & p_k^m & \bullet & q_1^m & \cdots & q_n^m \end{pmatrix}$$

Informally, at any point in compilation, contexts are pre-order representations of matched value. The fringe defines the possible values for $\vec{x}$, while pending calls to the compilation scheme define the prefix. Transfers of patterns from fringe to prefix are performed on the arguments of recursive calls, while tranfers in the opposite direction are performed as results are collected.

Initial call to $\mathcal{C}^*$ for an exhaustive match is:

$$\mathcal{C}^*\left((x), \begin{pmatrix} p^1 & \to & l^1 \\ p^2 & \to & l^2 \\ & \vdots & \\ p^m & \to & l^m \end{pmatrix}, total, \emptyset, (\bullet \_)\right)$$

For a non-exhaustive match, ex is *partial*, def is the one-element sequence $((\_), 1)$ and a trap handler is added as in section 3.3.

The new scheme sends back lambda-code l and a *jumps summary*, $\rho = \{\ldots, i \mapsto ctx, \ldots\}$, which is a mapping from trap numbers to contexts. Jump summaries describe what is known on matched values at the places where `(exit` i `...)` occur in l.

6.2 Operations on contexts

We define the following four operations on contexts :

(a) Context *specialization*, $\mathcal{S}$, by a constructor c of arity a is defined by mapping the transformation of figure 3-(a) on contexts rows.

(b) Context *collection*, $\mathcal{COL}$, is the reverse of specialisation. It combines the prefix last element with the appropiate number of arguments standing at begining of fringe (see figure 3-(b)).

(c) Context *pushing* $\Downarrow$ and *popping* $\Uparrow$, which move the fringe limit one step forward and backward, without examining any pattern (see figure 3-(c)).

As contexts are used to represent set of values, we naturally define contexts union and intersection. Context *union* $P \bullet Q \cup P' \bullet Q'$ yields a new matrix whose rows are the rows of $P \bullet Q$ and $P' \bullet Q'$. Row order is not relevant. Context intersection $P \bullet Q \cap P' \bullet Q'$ is defined as a context whose rows are the least upper bounds of the compatible rows of $P \bullet Q$ and $P' \bullet Q'$. Context *extraction* $\mathcal{EX}$ is a particular case of context intersection.

$$\mathcal{EX}(p, P' \bullet Q') = (\_ \;\dots\; \_ \;\bullet\; p \;\dots\; \_) \cap P' \bullet Q'$$

Hence, when p is $c(\_, \dots, \_)$, context extraction retains those value vectors represented by $P' \bullet Q'$ whose $k+1^{\text{th}}$ components admit c as head constructor. Observe that such a computation involves extracting or-pattern arguments and making wildcards more precise.

Except for collection and popping, which consume prefix elements, all these operations can be extended to simple matrices, by using an empty prefix in input, and taking the fringe for output. Doing so, we obtain exactly the operations of section 3.3 used to compute pattern matrices (specialisation $\mathcal{S}$ in particular).

Operations on contexts are extended to jump summaries in the natural manner. For instance, the union of ρ and ρ' is defined as:

$$\rho \cup \rho' = \{\dots, i \mapsto \rho(i) \cup \rho'(i), \dots\}$$

Operations on matrices are extended to reachable trap handlers in a similar manner: for instance, pushing trap handlers is defined as pushing all matrices in them :

$$\Downarrow((P_1, e_1); \dots; (P_t, e_t)) = (\Downarrow(P_1), e_1); \dots; (\Downarrow(P_t), e_t)$$

6.3 Compilation scheme

We now describe scheme $\mathcal{C}^*$ by considering cases over the typical call.

1. If n is zero. then m must be at least 1 and we have:

$$\mathcal{C}^*((), \begin{pmatrix} \to & l^1 \\ \to & l^2 \\ & \vdots \\ \to & l^m \end{pmatrix}, ex, def, ctx) = l^1, \emptyset$$

   Observe that the jump summary is empty since no exit is outputed.

2. With respect to section 3.3, the variable rule only changes as regards the extra arguments ex, def and ctx. We only describe these changes. The performed recursive call sends back code l and jump summary ρ :

$$l, \rho = \mathcal{C}^*(\dots, \dots, ex, \Downarrow(def), \Downarrow(ctx))$$

   Exhaustiveness information ex does not change, while the other new arguments are pushed.

   The variable rule sends back l unchanged and ρ popped.

3. In the constructor rule, let $C = \{c_1, \dots, c_k\}$ be the matched constructors, let also Σ be the signature of their type. For a given constructor $c \in C$, the performed recursive call is:

$$\mathcal{C}^*(\dots, \dots, ex, \mathcal{S}(c, def), \mathcal{S}(c, ctx))$$

   Exhaustiveness information ex is passed unchanged, while the other two extra arguments are specialized (specialization of trap handlers being the natural extension of matrix specialization).

   Each recursive call sends back a lambda-code $l(c)$ and a jumps summary ρ_c. Lambda-code $l(c)$ gets wrapped into let-bindings like in section 3.3, yielding the final lambda-code $r(c)$. We then define a case list $\mathcal{L}$ and a jump sumary ρ_{rec} as follows:

$$\begin{aligned} \mathcal{L} &= \texttt{case } c_1\texttt{: } r(c_1) \cdots \texttt{case } c_k\texttt{: } r(c_k) \\ \rho_{\text{rec}} &= \{\dots, i \mapsto \bigcup_{c \in C} \mathcal{COL}(\rho_c(i)), \dots\} \end{aligned}$$

The case list is as before, while the jump summary is the union of the the jump summaries produced by recursive calls, once collected.

Optimizations are then performed. For clarity, optimizations are described as a two phase process: first, extend (or not extend) the case list $\mathcal{L}$ with constructors taken from $\Sigma \setminus C$, and add (or not add) a default case; then, compute the final jump summary. In our implementation, these two phases are merged.

A first easy case is when $\Sigma \setminus C$ is empty or when ex is *total*. Then, the case list $\mathcal{L}$ is not augmented.

Otherwise, we distinguish two cases :

(a) If $\Sigma \setminus C$ is finite, then for all constructors c in this set we consider the context

$$Q_c \bullet Q'_c = \mathcal{EX}(c(\_, \dots, \_), ctx)$$

   Then, trap handlers $def = (P_1, e_1); \dots; (P_t, e_t)$ are scanned left-to-right, stopping at the smallest i, such that the intersection $Q'_c \cap P_i$ is not empty. That is, we find the trap handler where to jump to when x_1 head constructor is c, in order to extend the case list as follows :

$$\mathcal{L} = \mathcal{L}\ \texttt{case } c\texttt{: (exit } e_i\texttt{)}$$

   It is possible that e_i does not exist (when Q'_c is empty). This means that x_1 head constructor will never be c at runtime.

(b) If $\Sigma \setminus C$ is infinite (as in the case of integers) or considered too large (as it might be in the case of characters), then, a default case is added to the case list :

$$\mathcal{L} = \mathcal{L}\ \texttt{default: (exit } e_1\texttt{)}$$

   That is, all non-recognized constructors lead to a jump the nearest enclosing reachable trap-handler. However it is still possible to extend the case list for particular constructors, applying the previous procedure (a) to the constructors that appear in the first column of reachable trap handler matrices and not in C.

The final jump summary is computed by considering the final case list $\mathcal{L}$. For a given trap handler number e_i let $\{c'_1, \dots, c'_{k'}\}$ be the set of constructors such that $\texttt{case } c'_j\texttt{: (exit } e_i\texttt{)}$ appears in $\mathcal{L}$. Then the jump summary ρ_{e_i} is defined as:

$$\rho_{e_i} = \{\, e_i \mapsto \mathcal{EX}(c'_1(\_, \dots, \_) \mid \cdots \mid c'_{k'}(\_, \dots, \_)), ctx) \,\}$$

Additionnaly, if there is a default clause, the jump summary ρ_d is defined as:

$$\rho_d = \{\, e_1 \mapsto ctx \}$$

Finally the constructor rule sends back a switch on case list $\mathcal{L}$ and the jump summary built by performing the union of ρ_{rec}, of all ρ_{e_i}'s and, when appropriate, of ρ_d.

The constructor rule performs many context unions, so that contexts may indeed get huge. Fortunately, contexts can be made smaller using a simple remark. Namely, let $\vec{p}$ and $\vec{q}$ be two rows in a context, such that $\vec{p}$ is less precise than $\vec{q}$ (i.e., all instances of $\vec{q}$ are instances of $\vec{p}$). Then, row $\vec{q}$ can be removed from the context, without modying its meaning as a set of value vectors. Hence, while performing context union, one can leave aside some pattern rows. If the produced context is still too large, then contexts are safely approximated by first replacing some patterns in them by wildcards (typically all the pattern in a given column) and then removing rows using the previous remark. Rough experiments lead us to set the maximal admissible context size to 32 rows, yielding satisfactory compilation time in pathological examples and exact contexts in pratical examples.

4. Or-pattern compilation operates on matrices whose first colmun contains at least one or-pattern. Additionally, when p_1^i is a or-pattern, then for all j, $i < j \le m$ one of the following, mutually exclusive, conditions must hold:

   (a) p_1^i and p_1^j are *not* compatible.

   (b) p_1^i and p_1^j are compatible but not equivalent, and $(p_2^i \dots p_n^i)$ is less precise than $(p_2^j \dots p_n^j)$

   (c) p_1^i and $p_1^{j'}$ are equivalent for all j', such that $i < j' \le j$.

   Conditions (a) and (b) guarantee that, whenever p_1^i matches the first value vector v_1 of a value $\vec{v}$, but row i does not match $\vec{v}$, then no further row in P matches $\vec{v}$ either. This is necessary since further rows of P won't be reachable in case of failure in the or-pattern trap handler. Condition (c) concerns equivalent, consecutive, or-patterns and can be seen as an optimization.

   Now, consider one maximal sequence $i \dots j$ of row numbers such that $p_1^i \dots p_1^j$ are equivalent to the or-pattern $p = q_1 \mid \cdots \mid q_o$. Further assume that all these or-patterns bind the same variables, $y_1, \dots, y_v$ (which can be achieved by α-conversion or simulated by inserting let-bindings). First, we allocate a fresh trap number e and divide $P \to L$ into the following or-body $P' \to L'$ and or-trap $P'' \to L''$ clauses:

$$P' \to L' = \begin{pmatrix} & & \vdots & & \\ p_1^{i-1} & \dots & p_n^{i-1} & \to & l^{i-1} \\ q_1 & \_ & \_ & \to & (\texttt{exit}\ e\ y_1 \dots\ y_v) \\ & & \vdots & & \\ q_o & \_ & \_ & \to & (\texttt{exit}\ e\ y_1 \dots\ y_v) \\ p_1^{j+1} & \dots & p_n^{j+1} & \to & l^{j+1} \\ & & \vdots & & \end{pmatrix}$$

$$P'' \to L'' = \begin{pmatrix} p_2^i & p_m^i & \to & l^i \\ & \vdots & & \\ p_2^j & p_m^j & \to & l^j \end{pmatrix}$$

   In the or-body matrix, observe that or-pattern are expanded while the other pattern in those or-pattern rows get replaced by wildcards and the actions get replaced by exits to e.

   Recursive calls are performed as follows:

$$\begin{array}{rcl} l', \rho' & = & \mathcal{C}^*(\vec{x}, P' \to L', ex, def, ctx) \\ l'', \rho'' & = & \dots \\ \multicolumn{3}{l}{\dots \mathcal{C}^*(\vec{x}_{2\leftrightarrow n}, P'' \to L'', ex, \Downarrow(\mathcal{EX}(p, def)), \Downarrow(\mathcal{EX}(p, ctx)))} \end{array}$$

   Outputed code finally is $\texttt{catch}\ l' \texttt{with}\ (e\ y_1 \dots\ y_v)\ l''$ while jump summary is $\rho = \rho' \cup \Uparrow(\rho'')$.

5. The mixture rule is responsible feeding the other rules with appropriate clause matrices. We first consider the case of a random division. Hence let us cut $P \to L$ into $Q \to M$ and $R \to N$ at some row. Then a fresh trap number e is allocated and a first recursive call is performed:

$$l_q, \rho_q = \mathcal{C}^*(\vec{x}, Q \to M, \mathit{partial}, (R, e); def, ctx)$$

   Notice that the exhaustiveness information is *partial*, since nothing about the exhaustiveness of Q derives from the exhaustiveness of P. Also notice that reachable trap handlers are extended.

   Then, a second recursive call is performed:

$$l_r, \rho_r = \mathcal{C}^*(\vec{x}, R \to N, ex, def, \rho_q(e))$$

   It is no surprise that the context argument to the new call is extracted from the jump summary of the previous call. Notice that argument ex does not change. Indeed, if exit must not escape from P, then it must not escape from R either.

   Then, the scheme can output the code

$$l = \texttt{catch}\ l_q\ \texttt{with}\ (e)\ l_r$$

   and send back the jump summary $(\rho_q \setminus \{e\}) \cup \rho_r$, where $\rho_q \setminus \{e\}$ stands for ρ_q with binding for e removed.

   Of course, our optimizing compiler does not perform a random division into two matrices. It instead divides $P \to L$ right away into several sub-matrices, so that the four previous rules apply to every matrix in the division. Then, recursive calls are performed, in an order such that handlers are nested (more formally, this means several calls to the random mixture rule, when the division includes three matrices or more).

   Matrix division is performed by scanning the rows of $P \to L$ (P for short) from top to bottom. But, where the classical mixture rule would stop accumulating rows (for instance at a constructor to variable transition), the new mixture rule instead attempts to build a larger matrix.

   We only describe the case when p_1^1 is a constructor pattern. Other cases can be infered easily from that one. Thus, having performed the classical mixture rule, we are in a situation where the i topmost rows of P have a constructor pattern in first position (i.e. are contructor rows for short) and where p_1^{i+1} is not a constructor pattern. At that oint, a matrix C has been built, which encompasses all the rows of P from 1 to i. Let us further write P' for what remains of P, and let O and R be two new, initially empty matrices. We then scan the rows of P' from top to bottom, appending them at the end of C, O or R. That is, given row number j in P':

(a) If ${p'}_1^j$ is a variable pattern, then append row j at the end of R.

(b) If ${p'}_1^j$ is a constructor pattern. Then, there are three cases:

   i. If row j is not compatible with all the rows of both R and O, then append row j at the end of C (i.e., move row j above all the rows that have been extracted from P' at previous stages).

   ii. If row j is not compatible with all the rows of R and that one of conditions (a) or (b) for applying the or-pattern rule are met by O with row j appended at the end, then do such an append.

   iii. Otherwise, append row j at the end of R.

(c) If ${p'}_1^j$ is a or-pattern, then there are three cases:

   i. Do the same attempt as step (ii) above.

   ii. Should attempt above fail only because some rows in O have a first pattern equivalent to p_1^j, then let j' be the indice of the last such row. Then, if for all valid row indices j'' in O, such that $j' < j''$, patterns $o_1^{j''}$ and p_1^j are not compatible, then insert row j after row j'' in O (i.e. insert row j into position $j'+1$ in O)

   iii. In all other cases, append row j at the end of R

When the scan of P' is over, three matrices, C, O and R have been built. In the case where O is empty, matrix C is valid input to the constructor rule; otherwise, appending the rows of O at the end of C yields valid input for applying (maybe more than once) the or-pattern rule, which will in turn yield valid input to the constructor rule (provided that `(_ | ... )` or patterns have been replaced by semantically equivalent wildcards in a previous phase). Thus, the matrix built by appending O at the end of C is recorded into the overall division and the division process is restarted with input R, unless R is empty.

Finally, the full process divides the input matrix P into several matrices, each of which is valid input to the other rules of the compilation scheme.

7. EXPERIMENTAL RESULTS

We compare the performance of the code generated by the Objective-Caml compilers version 3.00 and 3.01, where the former implements the scheme of section 3.3 and the latter implements our new optimizing scheme (there are other modifications, but they are of minor relevance to our purpose). For most programs there is little difference, this is natural since pattern-matching usually accounts for a small fraction of most programs running time. Hence, we here intend both to present some programs that demonstrate significant improvement, and to convince that such programs are worth considering.

Our first benchmark, `fib`, is a fibonacci function that we write using a or-pattern.

```
let rec fib n = match n with
| (0|1) -> 1
| _     -> fib (n-1) + fib (n-2)
```

Here, we simply measure the execution time of calling the function with argument 38. Our second benchmark, `pcf`, is a byte-code compiler and interpreter for PCF. We compute the geometric mean of execution time for a set of five different PCF programs. The time-consuming part of this program is the byte-code machine which we coded in the style of the byte-code machine included in [13], the winning entry of the 2000 ICFP programming contest. (we also give figures for this program under the name `raytrace`).

Experiments were performed on a lightly loaded 366Mhz Pentium Pro Linux PC, measuring wall-clock time (in seconds) and computing percentages :

	fib		raytrace		pcf	
V 3.00	5.36	100	1.69	100	8.12	100
V 3.01	3.74	71	1.62	96	5.08	63

Obviously, as demonstrated by the `fib` example, compilation of or-patterns has much improved. Testing other examples that include or-patterns confirm that fact. The `pcf` example is more interesting, it shows that our optimizations yield a 37% speed-up, in the case of a typical ML application (a quickly written, compact, prototype implementation of some programming language). It should also be noticed that the new compiler somehow equates the runtime performance of various coding styles (more specifically, runtime performance of the code produced by the new compiler is less dependent on the relative ordering of non-overlapping patterns and on the order of arguments in pairs), a feature that is important for a high-level construct such as pattern-matching. The `raytrace` example shows a small improvement, this suggests that our technique can be of some importance for another class of programs.

We also performed measurements on a 500Mhz Dec Alpha server. They suggest that the effects of our optimization do not depend on the targeted architecture.

	fib		pcf	
V 3.00	3.4	100	4.13	100
V 3.01	2.5	74	2.86	69

(The `raytrace` example is missing by lack of IEEE floating point arithmetic in the Objective Caml compiler for this architecture.)

More detailed information on these benchmarks are available at `http://caml.inria.fr/pattern/speed.html`.

8. RELATED WORK

8.1 Decision Trees *vs* Backtracking automata

Compiling to decision trees is the original approach to pattern matching compilation, it first appeared in the Hope compiler and is described in [5]. It is currently used in the SML-NJ compiler [7].

In this approach, there is no *mixture* rule: instead, the *constructor* rule applies as soon as there is at least one constructor in the first column, and a specialization matrix is created for each matched constructor, plus one additionnal matrix for the remaining contructors in the signature of the types of matched values, if any. Specialization is done by following the rules of section 6.2. This means that rows whose first pattern is a variable get copied several times.

On one hand, this approach guarantees that one constructor test is never performed twice. On the other hand, it suffers from the duplication of clauses in all specialized matrices. To cope with actions duplication, SML-NJ first generates a local function for each action, and then replaces actions by function calls. However, later compilation phases must take care of not generating closures for such local functions. The static exception solution looks more simple and can be adapted in the context of decision trees. More important perhaps, copied pattern rows are compiled independantly and this result in potentially large code size. Namely, examples exist that make the SML-NJ compiler produce exponential code, even if this compiler uses hash-consing to produce a DAG instead of a decision tree.

Compilation to backtracking automata is the classical scheme of section 3.3 (see also [1, 19]). It is currently in use in the Haskell-HBC and Objective-Caml compiler [11]. As we already argued, its main advantage is that both patterns and actions are never copied, yielding linear output size. Of course, the price paid is potentially testing the same subterm several times, resulting in potentially poor runtime performance. In that aspect, our new compilation scheme shows that this price can be reduced significantly.

Some authors [17] claim that detecting unused match cases and non-exhaustive matchings while compiling pattern-matching is an advantage of the decision tree approach, since there is no dead code in a decision tree. We do not agree, since the unused match case problem is NP-complete [16], it looks hasardous to rely on a data-structure to solve it. However signaling non-exhaustive matchings and unused match cases to the programmer is important. Rather, we compute such information independently with a simple algorithm [12] that basically traverse the decision tree without generating it. Advantages are not generating the tree, stopping search as soon as used match cases are found and applying various heuristics and matrix simplifications which are not relevant to direct compilation. Then, one of our optimizations uses exhaustiveness information (see section 4.2).

8.2 Compiling or-patterns

From available ML or Haskell compilers, we only found two compilers dealing with or-patterns: the SML-NJ compiler and the Objective-Caml compiler. The former one expands or-patterns in a pre-processing step, while the latter one uses the technique of section 3.3.

On one hand, the SML-NJ approach benefits from a very simple treatment of variables in or-patterns, which were previously forbidden in the Objective-Caml compiler. By replacing actions in or-patterns by static exceptions with arguments, we improve the Objective-Caml scheme with an efficient compilation of variables in or-patterns.

On the other hand, expanding or-patterns in the preprocessing step can result in an explosion of the pattern-matching compiler input (consider the n-tuple pattern `(1|2), (3|4), (5|6), ...`). Of course such cases "do not occur in practice", but we feel more confortable with our technique which avoids the risk.

8.3 Optimizations

Most optimizations dealing with pattern-matching in the litterature try to improve the order in which tests are performed. In the matrix-based description of compilers, one considers alternatives to systematically choosing the first column of matrices in the constructor rule. Hence, such an approach can be characterized as "column optimization", while our approach would rather be "row optimization". Since choosing the best column is thought to be NP-complete (to our knowledge, there is no published proof), most approaches describe heuristics. A typical and early work on such heuristics is [2], a more recent and thorough study is [15]. Another, related in practice, approach relies on sequentiality theory to identify directions which are columns that must be tested by all possible matchers [10, 14, 16, 12]. However, computing directions is expansive, and one can consider relying on cheaper heuristics.

These works rather apply to the decision tree technique, with a primary focus on reducing code size. It is unclear to us how to combine column and row optimization in practice and whether this would yield noticeable improvements or not.

There also exists a partial-evaluation based approach to pattern-matching optimization. [8] and later [17] specialize an ultra-naive pattern-matching interpreter to create an efficient pattern-matching compiler. Both authors use context information as we do, but the automatic process of partial evaluation does not find sophisticated optimizations. In both cases, the final compiler is the classical decision tree based compiler.

9. CONCLUSION

In our opinion this paper contribution is twofold. First, we propose an improvement on the classical technique of compiling pattern-matching expressions into backtracking automata, a technique that has remained virtually the same for about 15 years. Our improvements yield automata that will run faster, thereby alleviating the disadvantage of bactracking automata in practical cases. Moreover the very structure of the produced automata is not altered and hence the highly desirable property that output size is linear in the input size is preserved. As a second contribution, we propose a technique for efficiently compiling or-patterns with variables, still preserving the linearity of output size. Using or-patterns in place of "catch-all" wildcards results in more robust programs, while using one clause with a or-pattern in place of several clauses with identical actions results in more compact, sometime clearer, programs. ML programmers can now enjoy these benefits, without being afraid of degraded runtime efficiency or code size explosion.

10. REFERENCES

[1] Augustsson, L. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture*, J.-P. Jouannaud, Ed. Springer-Verlag, Berlin, DE, 1985, pp. 368–381. Lecture Notes in Computer Science 201Proceedings of. Conference at Nancy.

[2] Baudinet, M., and MacQueen, D. Tree pattern matching for ML,. unpublished paper, Dec. 1985.

[3] Bernstein, R. L. Producing good code for the case statement. *Software—Practice and Experience 15*, 10 (Oct. 1985), 1021–1024.

[4] Bodík, R., Gupta, R., and Soffa, M. L. Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)* (New York, June 15–18 1997), vol. 32, 5 of *ACM SIGPLAN Notices*, ACM Press, pp. 146–158.

[5] Cardelli, L. Compiling a functional language. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (Aug. 1984), ACM, ACM, pp. 208–217.

[6] Fraser, C. W. A compact, machine-independent peephole optimizer. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1979), ACM, ACM, pp. 1–6.

[7] Harper, R. W., MacQueen, D. B., and Milner, R. Standard ML. Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, Edinburgh, UK, 1986. Also CSR-209-86.

[8] Jørgensen, J. Generating a pattern matching compiler by partial evaluation. In *Glasgow Workshop on Functional Programming, Ullapool* (Glasgow University, July 1990), P. C. J. van Rijsbergen, Ed., Springer-Verlag, pp. 177–195.

[9] Knoop, J., Rüthing, O., and Steffen, B. Partial dead code elimination. In *Proceedings of the Conference on Programming Language Design and Implementation* (New York, NY, USA, June 1994), ACM Press, pp. 147–158.

[10] Laville, A. Implementation of lazy pattern matching algorithms. In *ESOP'88* (1988), H. Ganzinger, Ed., vol. 300, pp. 298–316.

[11] Leroy, X. The objective caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jŕôme Vouillon. Available from `http://caml.inria.fr`.

[12] Maranget, L. Two techniques for compiling lazy pattern matching. Research Report 2385, INRIA Rocquencourt, Oct. 1994.

[13] PLClub, and Caml'R Us. Objective-caml: Winner of the first and second prizes of the p rogramming contest. ACM SIGPLAN International Conference on Functional Programming (ICFP '2000).

[14] Puel, L., and Suárez, A. Compiling pattern matching by term decomposition. *Journal of Symbolic Computation 15*, 1 (Jan. 1993), 1–26.

[15] Scott, K., and Ramsey, N. When do match-compilation heuristics matter? Tech. Rep. CS-2000-13, Department of Computer Science, University of Virginia, May 2000.

[16] Sekar, R. C., Ramesh, R., and Ramakrishnan, I. V. Adaptive pattern matching. In *Automata, Languages and Programming, 19th International Colloquium* (Vienna, Austria, 13–17 July 1992), W. Kuich, Ed., vol. 623 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 247–260.

[17] Sestoft, P. ML pattern match compilation and partial evaluation, 1996.

[18] Spuler, D. A. Compiler code generation for multiway branch statements as a static search problem. Tech. Rep. 94/3, Department of Computer Science, James Cook University, 1994.

[19] Wadler, P. Compilation of pattern matching. In *The Implementation of Functional Programming Languages*, S. L. Peyton Jones, Ed. Prentice-Hall International, 1987, ch. 7.

[20] Wegman, M., and Zadeck, F. K. Constant propagation with conditional branches. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages* (New Orleans, LS, Jan. 1985), B. K. Reid, Ed., ACM Press, pp. 291–299.