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)

November 23, 2000

Abstract

Pattern-matching is one of the main features of functionnal programming languages. In this paper, we present an improvement of the backtracking technique for the compilation of pattern-matching, which has been integrated in the new Objective-Caml compiler. It uses several optimizations, such as commutation of rows, use of exhaustivness information, and control flow optimization through the use of labeled static exceptions and context information during backtracking. These techniques have shown good results, both in decreasing the size of code, and increasing the speed in pattern-matching intensive programs.

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, a good compiler can generate better code than an human, both as regards compactness and efficiency, and it does so more quickly. There are roughly two approaches to this process, pattern matching being translated either to decision trees [Car84] or to backtracking automata [Aug85]. Both produce good code. Using decision trees, one produces *a priori* faster code (because a position in a term are 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 loosing the advantage given by the other technique.

TO ADD SOME-WHERE As a consequence, we will focus on the compilation of patterns restricted to constructors, even if our technique easily applies to other data types.

This compilation has the desirable feature of never duplicating any pattern or action. As a consequence output size is linear in input size.

This paper mostly focuses on producing faster code in the backtracking framework. Examining the code generated by the Objective-Caml compiler [Ler00], which basically used the original algorithm of [Aug85], on small frequently found programs, such as a list-merge function, or on large examples [PU00], 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. They also apply to or-patterns, a conveninent feature to group clauses with identical actions. To do so, the compiler uses new informations, such as incompatibility between patterns, exhaustiveness information and contextual information at the time of backtracking. Parallely, the target language is extended with labeled static exceptions. Static exceptions are also extended in the other direction of taking arguments, for compiling or-patterns.

All our optimizations are now implemenented in the lattest version of the Objective-Caml compiler, whose language of accepted patterns has also 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 argument it takes. Here arity of `Nil` is zero (this is *constant* constructor), while the arities of the *non-constant* constructors `One` and `Cons` are one and two respectively.

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

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

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* $(P \to L)$:

$$(P \to L) = \begin{pmatrix} p_1^1 & p_2^1 & \cdots & p_n^1 & \to & l^1 \\ p_1^2 & p_2^2 & \cdots & p_n^2 & \to & l^2 \\ & & \vdots & & & \\ p_1^m & p_2^m & \cdots & p_n^m & \to & l^m \end{pmatrix}$$

Which associates rows of patterns $(p_1^i\ p_2^i \ldots p_n^i)$ 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:*

$$\begin{cases} \_ & \preceq & v & \\ x & \preceq & v & \\ (p_1 \mid p_2) & \preceq & v & \textit{if and only if } p_1 \preceq v \textit{ or } p_2 \preceq v \\ c(p_1, \ldots, p_a) & \preceq & c(v_1, \ldots, v_a) & \textit{if and only if } (p_1 \ldots p_a) \preceq (v_1 \ldots v_a) \\ (p_1 \ldots p_a) & \preceq & (v_1 \ldots v_a) & \textit{if and only if } p_i \preceq v_i \textit{ for all } i, 1 \leq i \leq a \end{cases}$$

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:

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

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

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 $\vec{v} = (v_1 \dots v_n)$ be a value vector. The value v* matches *line number i in P, if and only if the following two conditions are satisfied:*

$$(p_1^i \dots p_n^i) \preceq (v_1 \dots v_n) \quad \textit{and} \quad \forall j < i,\ (p_1^j \dots p_n^j) \npreceq (v_1 \dots v_n)$$

We will not give a full semantics for evaluating pattern matching expressions, and more generally lambda-code. Intuitively, given a clause matrix $P \to L$ and a value vector $\vec{v}$, such that, line number i in P matches $\vec{v}$, then evaluating the matching of $\vec{v}$ by $P \to L$ in some environment ρ is evaluating l^i in ρ extended by the bindings introduced while matching $\vec{v}$ by $(p_1^i \dots p_n^i)$.

Like pattern vectors, pattern matrices represent sets of value vectors. More sepcifically, 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 pattern 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 [Mar94, Section X]). 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:

$$\left\{\begin{array}{rcll}
_\uparrow q & = & q & \\
p \uparrow _ & = & p & \\
(p_1 \,|\, p_2) \uparrow q & = & (p_1 \uparrow q \,|\, p_2 \uparrow q) & \\
p \uparrow (q_1 \,|\, q_2) & = & (p \uparrow q_1 \,|\, p \uparrow q_2) & \\
c(p_1, \dots, p_a) \uparrow c(q_1, \dots, q_a) & = & c(r_1, \dots, r_a) & \text{where } (r_1 \dots r_a) = (p_1 \dots p_a) \uparrow (q_1 \dots q_a) \\
(p_1 \dots p_a) \uparrow (q_1 \dots q_a) & = & (p_1 \uparrow q_1 \dots p_a \uparrow q_a) &
\end{array}\right.$$

This is easily proved, 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 here a compilation scheme close to the one described in [Wad87b, Aug85], and implemented in compilers such as the `hbc` compiler (without or-patterns) 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 abreviated 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
    case c1: l1··· case ck: lk
    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. In examples, we use the keyword `switch*` to introduce 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 [Ber85]. The key points are, using range tests, which can typically be performed by one single (unsigned) test and branch plus possibly one addotion, 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 [Spu94]. We plan a work on this topic.
- 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.

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

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

$$\texttt{match}\ x\ \texttt{with}\ \texttt{|}\ p^1\texttt{->}\ e^1\ \texttt{|}\ p^2\texttt{->}\ e^2 \ldots\ \texttt{|}\ p^m\texttt{->}\ e^m$$

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

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

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 by hypothesis the first row of P exists and matches the empty vector (). Hence we get:

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

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

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

$$\mathcal{C}(\vec{x}, P) = \mathcal{C}((x_2\ x_3 \dots x_n), \begin{pmatrix} p_2^1 & p_3^1 & \cdots & p_n^1 & \rightarrow & \texttt{let (}y^1\ x_1\texttt{)}\ l^1 \\ p_2^2 & p_3^2 & \cdots & p_n^2 & \rightarrow & \texttt{let (}y^2\ x_1\texttt{)}\ l^2 \\ & & & \vdots & & \\ p_2^m & p_3^m & \cdots & p_n^m & \rightarrow & \texttt{let (}y^m\ x_1\texttt{)}\ l^m \end{pmatrix})$$

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

   (b) If all patterns in the first column of P are constructor patterns $c(q_1, \dots, q_a)$. Let $C = \{c_1, c_2, \dots, c_k\}$ be the set of matched constructors. That is the set of head constructors of the p_i^1's.
   Then, for a given constructor c in C, we define the *specialized* clause matrix $\mathcal{S}(c, P \rightarrow L)$ by mapping the following transformation on the rows of P.

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

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

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

   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 ommitted when C makes up a full signature.) We call this rule, the *constructor* rule.

   (c) If there is only one row in P with an or-pattern in the first column:

$$P = \big((q_1 \,|\, ... \,|\, q_o)\ p_2\ \cdots\ p_n\ \rightarrow\ l\ \big),$$

   Then, compilation result is:

$$\mathcal{C}((x_1), \begin{pmatrix} q_1 & \cdots & \rightarrow () \\ ... & & \\ q_o & \cdots & \rightarrow () \end{pmatrix})\texttt{;x}\ \mathcal{C}((x_2\ x_3 \dots x_n), (p_2 ... p_n \rightarrow l))$$

   This rule is the *orpat* rule. This rule only exists in the Objective-Caml compiler, and does not duplicate any pattern nor action.

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

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

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 leaved 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 defeer the full presentation of our scheme to section 6.

Il faut parler du fait qu'on supprime aussi les clauses inutiles

4.1 Optimizing the mixture rule

In this section and in the following, our running examples 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 and the initial call to scheme $\mathcal{C}$ will be: $\mathcal{C}((\texttt{lx ly}), (P \to L))$ Where $(P \to L)$ is:

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

Applying the mixture rule twice yields three matrices:

$$P_1 \to L_1 = \begin{pmatrix} \texttt{Nil} & \_ & \to 1 \end{pmatrix}, \quad P_2 \to L_2 = \begin{pmatrix} \_ & \texttt{Nil} & \to 1 \end{pmatrix}, \quad P_3 \to L_3 = \begin{pmatrix} \texttt{x::xs} & \texttt{y::ys} & \to 3 \end{pmatrix}$$

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 clauses matrices define the same matching function, namely they both map $(\texttt{[]}\ v)$ to 1, $(v_1::v_2\ \texttt{[]})$ 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:

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

Final outputs for $P \to L$ and $P' \to L'$ are:

```
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")
```

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, (see [Mar94] for a description of the algorithm used). 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 int 0: 2 default: exit)` always succeeds, and we can replace it, first by `(switch* ys with int 0: 2)`, then by 2. We can also supress the outermost trap. Hence, cumulating both optimizations, compilation of $P \to L$ finally yields the optimal lambda-code:

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

In the general case, benefiting from exhaustivness information is achieved by a small modification of 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

Introduire les exceptions statiques avec label et arguments

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

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

The optimized mixture rule yields four matrices:

$$P_1 \to L_1 = \begin{pmatrix} \texttt{Nil} & \_ & \to \texttt{1} \\ \texttt{Cons (x,xs)} & \texttt{Cons (y,ys)} & \to \texttt{5} \end{pmatrix}, \quad P_2 \to L_2 = \begin{pmatrix} \_ & \texttt{Nil} & \to \texttt{2} \end{pmatrix},$$

$$P_3 \to L_3 = \begin{pmatrix} \texttt{One x} & \_ & \to \texttt{3} \end{pmatrix}, \quad P_4 \to L_4 = \begin{pmatrix} \_ & \texttt{One x} & \to \texttt{4} \end{pmatrix}$$

For reasons that will appear immediatly, we apply the mixture rule from bottom to top, thereby nesting trap handlers. The match being exhaustive, compilation yields:

```
catch
  (catch
     (catch
       (switch lx with
         case Nil: 1
         case Cons: (switch ly with case Cons: 5 default: exit)
         default: exit)
     with
       (switch ly with case Nil: 2 default: exit))
  with
    (switch lx with case One: 3 default: exit))
with
  4
```

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 line 6 above, where a switch on `ly` is performed. This switch may fail for two reasons, either `ly` is `Nil`, or `One` v. Yet, in both cases the action taken is a jump to the nearest enclosing handler (line 9), where `ly` is tested again against `Nil`. Here, in our case of (`lx ly`) being (`Cons (`v_1`, `v_2`) One `v), control then goes to line 11, where another switch on `lx` (against `One x`) fails, resulting in final jump to line 13.

Hence, it would be appropriate to jump to line 13 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 $P \rightarrow L$ is:

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

The code above maps value 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. However, code size remains under control, since no extra switches are generated. In fact, there are many situations where machine-level code size does not increase (This is the case here, since both the original switch on `ly` and the modified one compile into one test and branch that discriminates beetween the constant and non-constant cases, plus one test and branch in the non-constant case)

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 integer. Reachable trap handlers originate from the division performed by the mixture rule. Here, $P_1 \rightarrow L_1$ 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 then generating the switch on `ly`, 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 15). It results from compiling P_3 and is a switch of `lx` against `One x`.

The only `(exit 3)` lies in trap handler number 2 (line 13) and results from `ly` not being `Nil`, this gives us no direct information on `lx`. Now, looking for `(exit 2)`, we can infeer that trap handler number 2 is entered from two different points. In the first case (line 8), (`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 13 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 15 can be simplified into "`3`". This elimination of useless tests is similar to the dead code elimination which is usualy performed at a lower level.

Finally, after all optimizations, there remains one redundant switch in produced code, in trap-handler number 2 (line 13). 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 matrices. We call such a set a *context*, then contexts for the three trap handlers of our example are:

Trap number	Context
2	$\begin{pmatrix} \texttt{One}\ \_ & \_ \\ \texttt{Cons}\ (\_,\ \_) & \texttt{Nil} \end{pmatrix}$
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 unefficient, 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. In the next example, we also show how to compile or-patterns with variable. Consider a "*car*" function for our three-constructors list:

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

A new trap-handler number 2 is allocated and the or-pattern clause is expanded, action for the new clauses are exits to 2:

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

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 `trap` ... `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 ultimatly 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 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 vector $(v_1\ v_2)$ such that v_1 is an instance of both p_1 and p_2. As a consequence, the following compilation is correct:

```
catch
  (catch
    (switch x with
      case 1: (exit 2) case 2: (exit 2)
      case 3: (exit 3) case 4: (exit 3)
      default: (exit 1))
  with (2)
   C((y), ( p2 → l1 ), 1))
with (3)
  C((y), ( q2 → l2 ), 1)
```

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. Conversly, matrix division cannot be avoided when matching by p_1 does not exclude matching by p_2, that is when p_1 and p_2 are compatible. This is the case, for instance, when p_1 = `(1|2)` and q_2 = `(2|3)`. Then a correct compilation is:

```
catch
  (catch
    (switch x with
      case 1: (exit 2) case 2: (exit 2) default: (exit 3))
  with (2)
   C((y), ( p2 → l1 ), 3))
with (3)
  (catch
    (switch x with
      case 2: (exit 4) case 3: (exit 4) default: (exit 1))
  with (4)
   C((y), ( q2 → l2 ), 1))
```

(Note that the third argument to the first recursive call to the compilation scheme is "3" and not "1".)

6 Our Compilation Scheme

6.1 Contexts

Contexts are a compile-time representation of matched values. Given a pattern matrix P of width n and in integer k, $1 \leq k \leq n$. The $P\bullet_k$ is matrix P, where each row is divided into a a prefix of length k and a fringe:

$$P\bullet_k = \begin{pmatrix} p_1^1 & \cdots & p_k^1 & \bullet & p_{k+1}^1 & \cdots & p_n^1 \\ p_1^2 & \cdots & p_k^2 & \bullet & p_{k+1}^2 & \cdots & p_n^2 \\ & & & \vdots & & & \\ p_1^m & \cdots & p_k^m & \bullet & p_{k+1}^m & \cdots & p_n^m \end{pmatrix}$$

Informally, at any point in compilation, contexts are pre-order representations of matched value. At some point during compilation, the vector of variable $\vec{x}$ defines the fringe (i.e., we have $\vec{x} = (x_{k+1} \ldots x_n)$) 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.

Context union $P\bullet_k \cup Q\bullet_k$ applies to context of same width n, it produces a matrix of width n whose rows are the rows of P and Q. When viewing contexts as the set of their instances, context union is set union.

Context specialization by a constructor c of arity a is defined by mapping the following transformation on contexts rows:

$P\bullet_k$ row	$\mathcal{S}(c, P\bullet_k)$ row
$p^i_1 \cdots p^i_k \bullet c(q^i_1, \ldots, q^i_a) \cdots p^i_n$	$p^i_1 \cdots p^i_k \quad c(\_, \ldots, \_) \bullet q^i_1 \cdots q^i_a \quad p^i_{k+1} \cdots p^i_n$
$p^i_1 \cdots p^i_k \bullet \quad \_ \quad \cdots p^i_n$	$p^i_1 \cdots p^i_k \quad c(\_, \ldots, \_) \bullet \_ \cdots \_ \quad p^i_{k+1} \cdots p^i_n$
$p^i_1 \cdots p^i_k \bullet c'(q^i_1, \ldots, q^i_a) \cdots p^i_n$	no row

Collecting, $\mathcal{COL}(P\bullet_{k+1})$, is the reverse of specialisation. It combines the prefix last element with the appropiate number of arguments standing at begining of fringe:

$P\bullet_{k+1}$ row	$\mathcal{COL}(P\bullet_{k+1})$ row
$p^i_1 \cdots p^i_k \quad c(\_, \ldots, \_) \bullet q^i_1 \cdots q^i_a \quad p^i_{k+1} \cdots p^i_n$	$p^i_1 \cdots p^i_k \bullet c(q^i_1, \ldots, q^i_a) \cdots p^i_n$

We also define context *pushing* and *popping*, which moves the fringe limite one step forward and backward, whithout examining any pattern:

$P\bullet_k$ row	$\Downarrow(P\bullet_k)$ row	$\Uparrow(P\bullet_k)$ row
$p^i_1 \cdots p^i_k \bullet p^i_{k+1} \cdots p^i_n$	$p^i_1 \cdots p^i_{k+1} \bullet p^i_{k+2} \cdots p^i_n$	$p^i_1 \cdots p^i_{k-1} \bullet p^i_k \cdots p^i_n$

Finally, context extraction $\mathcal{EX}(p, P\bullet_k)$, is a limited form of intersection:

$P\bullet_k$ row	$\mathcal{EX}(p, P\bullet_k)$ row	Condition
$p^i_1 \cdots p^i_k \bullet p^i_{k+1} \cdots p^i_n$	$p^i_1 \cdots p^i_k \bullet p \uparrow p^i_{k+1} \cdots p^i_n$	When p and p^i_{k+1} are compatible
$p^i_1 \cdots p^i_k \bullet p^i_{k+1} \cdots p^i_n$	no row	Otherwise

Note that $\mathcal{EX}(p, P\bullet_k)$ instances are the instances of both $P\bullet_k$ and $(\_ \ldots \_ \bullet p \ldots \_)$.

Context operations $\mathcal{S}$, $\Downarrow$ and $\mathcal{EX}$ take their arguments from the fringe only. Hence the same operations are easily defined on pattern matrices. For instance, given a matrix P, $\Downarrow(P)$ is P without its leftmost column.

The new scheme $\mathcal{C}^*$ takes five arguments and a typical call is $\mathcal{C}(\vec{x}, P \to L, ex, def, ctx)$, where $\vec{x} = (x_{k+1} \ldots x_n)$ and $P \to L$ is a clause matrix of width $n - k$. Extra argument are:

- The exhaustiveness argument *ex* is either *partial* or *total* depending on whether compilation is allowed to output exit constructs or not.
- Reachable trap handlers *def* replace the e argument. One single trap handler number is now replaced by a sequence $(P_1, e_1); \ldots; (P_t, e_t)$, where the P_i's are pattern matrices of width $n - k$.
- The context, *ctx*, is a context $P\bullet_k$ of total width n and prefix width k.

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

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

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 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 jumps occured.

We then describe schem $\mathcal{C}^*$, by considering cases over the typical call.

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

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

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

2. The variable rule changes only with respect to argument management. First, the following recursive call is performed

$$Variable\ \frac{\mathcal{C}^*((x_{k+1}\dots x_n), \Downarrow(P) \rightarrow L', ex, \Downarrow(def), \Downarrow(ctx)) = l, \rho}{\mathcal{C}^*((x_k\dots x_n), P \rightarrow L, ex, def, ctx) = l, \Uparrow(\rho)}$$

   This call sends back lambda-code l and a jump summary ρ. Then the variable rule sends back l unchanged and ρ popped:

$$\Uparrow(\rho) = \{\, i \mapsto \Uparrow(ctx), i \mapsto ctx \in \rho \,\}$$

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 $c \in C$ of arity a, The recursive call performed is:

$$\mathcal{C}^*((y_1, \dots, y_a, x_2, \dots, x_n), \mathcal{S}(c, P) \rightarrow L), ex, \mathcal{S}(c, def), \mathcal{S}(c, ctx))$$

   Observe that ex is passed unchanged, while both reachable trap-handlers def and context ctx are specialized, so that they now apply to value vectors whose $k+1$-th component is $c(v_1, \dots, v_a)$.

   The recursive call sends back a lambda-code l_c and a jumps summary ρ_c. Lambda-code l_c gets wrapped into let-binding 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 as follows:

$$\mathcal{L} = \texttt{case}\ c_1\texttt{:}\ r(c_1) \cdots \texttt{case}\ c_k\texttt{:}\ r(c_k), \quad \rho = \{\, \dots, i \mapsto \bigcup_{c \in C} \mathcal{COL}(\rho_c(i)), \dots \}$$

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

   Optimizations are then performed, they all regard extending (or not extending) C (matched constructors), $\mathcal{L}$ (the case list) and ρ (the jumps summary).

   Exhaustiveness If ex is *total*, then optimizations are over.

   Exit optimization If ex is *partial*, reachable trap handlers def are scanned left-to-right. For a given trap-handler (P_i, e_i), we consider two cases:

   (a) If the first column of P_i contains a variable, then search is over. Let C' be what remains of Σ (i.e. $C' = \Sigma \setminus C$). There are three subcases, depending on whether C' is empty, non-empty and finite or infinite (i.e., Σ is the signature of integers).

      i. If $C' = \emptyset$ then do nothing.

      ii. Otherwise, let C' be $\{c'_1, \dots, c'_{k'}\}$, with $k' \geq 1$. and state:

$$\begin{array}{l} \mathcal{L} = \mathcal{L}\ \texttt{case}\ c'_1\texttt{: (exit}\ e_i\texttt{)} \ \cdots\ \texttt{case}\ c'_{k'}\texttt{: (exit}\ e_i\texttt{)} \\ \rho = \rho \cup \{e_i \mapsto \mathcal{EX}((c'_1(\_, \dots, \_) \mid \cdots \mid c'_{k'}(\_, \dots, \_)), ctx)\} \end{array}$$

      iii. If C' is infinite (or considered too large, cf. the characters), then the current context ctx cannnot be precised (unless the patterns are extended with a "*not*" pattern) and we state:

$$\mathcal{L} = \mathcal{L}\ \texttt{default: (exit}\ e_i\texttt{)}, \quad \rho = \rho \cup \{e_i \mapsto ctx\},$$

      Hence, we here over-estimate the set of values such that `(exit` e_i`)` is performed.

   (b) If the first column of P_i is made of constructor-patterns and or-patterns only, then let $C' = \{c'_1, \dots, c'_{k'}\}$ be the root constructors of these patterns not matched so far and state:

$$\begin{array}{l} C = C \cup C', \quad \mathcal{L} = \mathcal{L}\ \texttt{case}\ c'_1\texttt{: (exit}\ e_i\texttt{)} \ \cdots\ \texttt{case}\ c'_{k'}\texttt{: (exit}\ e_i\texttt{)} \\ \rho = \rho \cup \{e_i \mapsto \mathcal{EX}((c'_1(\_, \dots, \_) \mid \cdots \mid c'_{k'}(\_, \dots, \_)), ctx)\} \end{array}$$

      Search continues by examining (P_{i+1}, e_{i+1}). Observe that, by the initial value $((\_), 1)$ of reachable trap handlers, case (a) must occur.

Context optimization It is in fact performed, while building the case list.

Before adding any case `case` c: l to the case list $\mathcal{L}$, we first check whether context $\mathcal{EX}(c(\_,\ldots,\_), ctx)$ is empty or not. In the former case, the case list is not extended, in the latter case, we proceed normally.

After optimizations, the scheme sends back a switch on case list $\mathcal{L}$ and the jump summary ρ. Observe that switches may carry a case list whose length is the cardinal of Σ. Final code size now critically depends on how well switches are translated into machine-level code.

4. Or-pattern compilation operates on matrices whose first colmun contains at least one or-pattern. Additinally, when p^i_{k+1} is a or-pattern, then for all j, $i < j \leq m$ one of the following, mutually exclusive, conditions must hold:

(a) p^i_{k+1} and p^j_{k+1} are *not* compatible.

(b) p^i_{k+1} and p^j_{k+1} are compatible but not equivalent, and $(p^i_{k+2}\cdots p^i_n)$ is less defined than $(p^j_{k+2}\cdots p^j_n)$

(c) p^i_{k+1} and $p^{j'}_{k+1}$ are equivalent for all j', such that $i < j' \leq j$.

Now, consider one maximal sequence $i\ldots j$ of row numbers such that $p^i_{k+1}\ldots p^j_{k+1}$ 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, $y_1,\ldots,y_v$ (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:

$$P' \to L' = \begin{pmatrix} & \vdots & & & \\ p^{i-1}_{k+1} & p^{i-1}_{k+2} & p^{i-1}_n & \to & l^{i-1} \\ q_1 & \_ & \_ & \to & (\texttt{exit}\ e\ y_1\ldots\ y_v) \\ & \vdots & & & \\ q_o & \_ & \_ & \to & (\texttt{exit}\ e\ y_1\ldots\ y_v) \\ p^{j+1}_{k+1} & p^{j+1}_{k+2} & p^{j+1}_n & \to & l^{j+1} \\ & \vdots & & & \end{pmatrix} \quad \text{and} \quad P'' \to L'' = \begin{pmatrix} p^i_{k+2} & p^i_m & \to & l^i \\ & \vdots & & \\ p^j_{k+2} & p^j_m & \to & l^j \end{pmatrix}$$

Recursive calls are performed as follows:

$$\begin{array}{rcl} l', \rho' & = & \mathcal{C}^*(\vec{x}, P' \to L', ex, def, ctx) \\ l'', \rho'' & = & \mathcal{C}^*((x_{k+2}\ldots x_n), P'' \to L'', ex, \Downarrow(\mathcal{EX}(p, def)), \Downarrow(\mathcal{EX}(p, ctx))) \end{array}$$

Outputed code finally is `catch` l'`with (`$e\ y_1\ldots\ y_v$`)` 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 divison. 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:

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

Notice that the exhaustiveness information is *partial*, since nothing about the exhaustiveness of Q derives from the exhaistivness of P. Also notice that that reacheable 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 $l =$ `catch` l_q `with (`e`)` l_r and send back the jump summary $(\rho_q \setminus \{e \mapsto \ldots\}) \cup \rho_r$, where $(\rho_q \setminus \{e \mapsto \ldots\})$ stands for ρ_q minus the binding for e.

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 one of the three 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 divisiion includes three matrices or more).

Matrix division is performed by scanning the rows of $P \to L$ (P for short) from top to bottom, accumulating rows as long as the variable, constructor or or-pattern rule apply. 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 consider the case of optimizing application of the constructor rule, since the case of optimizing the application of the other two rules can be deduced easily from the constructor case.

Thus, 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_{n-k}^{i+1} is not a constructor pattern. Hence at that proint, 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 will scan the rows of P' from top to bottom, moving then into one of C, O or R. That is, given row number j in P':

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

(b) If ${p'}_{n-k}^{j}$ 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 upward all rows previously extracted from P')

   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 ${p'}_{n-k}^{j}$ is a or-pattern, then there are three cases:

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

   ii. Should attempt above fail only because some rows in O have a first pattern equivalent to p_{n-k}^{i+1}, 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 $o_{k+1}^{j''}$ and p_{k+1}^{j} are not compatible, then insert row j after row j'' in O (i.e. move row j upward in O,

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

Hence, when the scan of P' is over, appending the rows of O at the end of C yields valid input for the or-pattern rule or for the constructor rule (when O is empty). Thus, this extended matrix is recorded into the overall division and the division process is restarted with input R, unless R is empty.

7 Experimental results

8 Related Work

Approaches for pattern-matching compilation certainly fall into two classes: decision trees and backtracking automatas. Both approaches have been used and are still being used in current ML compilers.

Parler des extensions au pattern-matching par les active patterns [Erw97], les vues [Wad87a, Oka98, ME00], et les types abstraits [BC93] ???

8.1 Decision trees

This is the approach used for the first time in the Hope compiler, and described in [Car84]. It is currently used in the SML-NJ compiler [HMM86].

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 aditionnal 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 6.1. This means that some rows (the ones 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. This first concerns actions, SML-NJ solves this problem by first generating local functions for each action, and then replacing actions by function calls. In such a case, another optimization must take care of not generating closures for such local functions. The static exception solution looks more simple and can be adopted in the context of decision trees. More important perhaps, copied pattern rows are compiled independantly and this result in potentially large code size. Previous experiments [Mar94] showed us that the final code size for (naive) decision trees is 5 % to 10 % larger that the one for backtracking automata. There also exists examples 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.

Some authors as [Ses96] claims that detecting unsused match cases (their actions are absent from the decision tree) and non-exhaustive matchings is another advantage of the decision tree approach. We do not agree, since the unused match case problem is NP-complete [SRR92], it looks dangerous to rely on a data-structure to solve it. Rather, we use a simple algorithm [Mar94] that basically traverse the decision tree without generating it. Advantages are not generating the tree, stopping search immediately when used match cases are found and applying various heuristics and matrix simplifications which are not relevant to direct compilation. This algorithm is used in a previous phase of the Objective Caml compiler, warnings are issued when appropriate. In this paper we make use of exhaustiveness information for optimizations. By contrast, there is no need, in our opinion, to attempt discarding code corresponding to unused match cases, since they are programming errors anyway.

8.2 Backtracking automatas

This other approach was described as the classical scheme in section 3.3 of this paper (see also [Aug85, Wad87b]. It is currently in use for instance in the Haskell-HBC and Objective-Caml compiler [Ler00].

As we aleady argued, the main advantage of this approach is that both patterns and actions are never copied, yielding linear output size (cost being counted "by switches"). 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 priced can be reduced significantly.

8.3 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 described in 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 improves 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.

8.4 Optimization

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 the best column is thought to be NP-complete (to our knowledge, there is there is no published proof), most approaches describe heuristics. A typical and early work on such heuristics is [BM85], a more recent and thorough study is [SR00]. Another, related in practice, approach relies on sequentiality theory to identify directions which are columns that must be tested by all possible matchers [Lav88, PS93, SRR92, Mar94]. Directions can be used as a primary column finding heuristics, other heuristics are still useful, as secondary heuristics. Moreover, computing directions is expansive, and one can consider relying on cheaper heuristics.

All these works except the last one focus on the decision tree technique, with a primary focus on reducing code size and side benefits in improving runtime performance (some useless tests are not performed). 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. [Jør90] and later [Ses96] 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 descision tree based compiler.

9 Conclusion

References

[Aug85] L. Augustsson. Compiling pattern matching. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 368–381. Springer-Verlag, Berlin, DE, 1985. Lecture Notes in Computer Science 201Proceedings of. Conference at Nancy.

[BC93] F. Burton and R. Cameron. Pattern matching with abstract data types, 1993.

[Ber85] Robert L. Bernstein. Producing good code for the case statement. *Software—Practice and Experience*, 15(10):1021–1024, October 1985.

[BM85] Marianne Baudinet and David MacQueen. Tree pattern matching for ML,. unpublished paper, December 1985.

[Car84] Luca Cardelli. Compiling a functional language. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 208–217. ACM, ACM, August 1984.

[Erw97] M. Erwig. Active patterns. *Lecture Notes in Computer Science*, 1268:21–??, 1997.

[HMM86] R. W. Harper, D. B. MacQueen, and R. Milner. Standard ML. Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, Edinburgh, UK, 1986. Also CSR-209-86.

[Jør90] Jesper Jørgensen. Generating a pattern matching compiler by partial evaluation. In Professor C. J. van Rijsbergen, editor, *Glasgow Workshop on Functional Programming, Ullapool*, pages 177–195, Glasgow University, July 1990. Springer-Verlag.

[Lav88] A. Laville. Implementation of lazy pattern matching algorithms. In H. Ganzinger, editor, *ESOP'88*, volume 300, pages 298–316, 1988.

[Ler00] Xavier Leroy. 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`.

[Mar94] Luc Maranget. Two techniques for compiling lazy pattern matching. Research Report 2385, INRIA Rocquencourt, October 1994.

[ME00] Simon Peyton Jones Martin Erwig. Pattern guards and transformational patterns, September 2000.

[Oka98] C. Okasaki. Views for standard ml, 1998.

[PS93] Laurence Puel and Ascánder Suárez. Compiling pattern matching by term decomposition. *Journal of Symbolic Computation*, 15(1):1–26, January 1993.

[PU00] PLClub and Caml'R Us. Objective-caml: Winner of the first and second prizes of the programming contest. September 2000. ACM SIGPLAN International Conference on Functional Programming (ICFP '2000).

[Ses96] P. Sestoft. Ml pattern match compilation and partial evaluation, 1996.

[Spu94] D. A. Spuler. Compiler code generation for multiway branch statements as a static search problem. Technical Report 94/3, Department of Computer Science, James Cook University, 1994.

[SR00] Kevin Scott and Norman Ramsey. When do match-compilation heuristics matter? Technical Report CS-2000-13, Department of Computer Science, University of Virginia, May 2000.

[SRR92] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. In Werner Kuich, editor, *Automata, Languages and Programming, 19th International Colloquium*, volume 623 of *Lecture Notes in Computer Science*, pages 247–260, Vienna, Austria, 13–17 July 1992. Springer-Verlag.

[Wad87a] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction, 1987.

[Wad87b] Philip Wadler. Compilation of pattern matching. In S. L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, chapter 7. Prentice-Hall International, 1987.