Compiling Pattern Matching to Good Decision Trees

Luc Maranget
INRIA — France
Luc.maranget@inria.fr

Abstract

We address the issue of compiling ML pattern matching to compact and efficient decisions trees. Traditionally, compilation to decision trees is optimized by (1) implementing decision trees as dags with maximal sharing; (2) guiding a simple compiler with heuristics. We first design new heuristics that are inspired by *necessity*, a concept from lazy pattern matching that we rephrase in terms of decision tree semantics. Thereby, we simplify previous semantic frameworks and demonstrate a straightforward connection between necessity and decision tree runtime efficiency. We complete our study by experiments, showing that optimizing compilation to decision trees is competitive with the optimizing match compiler of Le Fessant and Maranget (2001).

Categories and Subject Descriptors D 3. 3 [*Programming Languages*]: Language Constructs and Features—Patterns

General Terms Design, Performance.

Keywords Match Compilers, Decision Trees, Heuristics.

Note This version includes an appendix, which is absent from published version.

1. Introduction

Pattern matching is certainly one of the key features of functional languages. Pattern matching is a powerful high-level construct that allows programming by directly following case analysis. Cases to match are expressed as "algebraic" patterns, *i.e.* ordinary terms. Definitions by pattern matching are roughly similar to term rewriting rules: a series of rules is defined; and execution is performed on *subject* values by finding a rule whose left-hand side is matched by the value. With respect to plain term rewriting (Terese 2003), the semantics of ML simplifies two issues. First, matches are always attempted at the root of the subject value. Secondly, the matched rule is unique, thanks to the textual priority scheme.

Any ML compiler translates the high-level pattern matching definitions into low-level tests, organized in *matching automata*. Matching automata fall in two categories: *decision trees* and *backtracking automata*. Compilation to backtracking automata has been introduced by Augustsson (1985). The primary advantage of the technique is a linear guarantee for code size. However, backtracking automata may backtrack. Therefore, they may scan subterms more than once. As a result, backtracking automata are potentially inefficient at runtime. The optimizing compiler of Le Fessant and Maranget (2001) alleviates this problem.

In this paper we study compilation to decision trees. The primary advantage of decision trees is that they never test a given subterm of the subject value more than once (and their primary drawback is potential code size explosion). We aim to refine naive compilation to decision trees, and to compare the output of the resulting optimizing compiler with optimized backtracking automata.

Compilation to decision trees is sensitive to the testing order of subject value subterms. The situation can be explained by the example of a human programmer attempting to translate a ML program into a lower-level language without pattern matching. Let `f` be the following function[1] that takes three boolean arguments:

```
let f x y z = match x,y,z with
| _,F,T -> 1
| F,T,_ -> 2
| _,_,F -> 3
| _,_,T -> 4
```

Where `T` and `F` stand for true and false, respectively.

Apart from preserving ML semantics (*e.g.* `f F T F` should evaluate to `2`), the game has one rule: never test `x`, `y` or `z` more than once. A natural idea is to test `x` first, *i.e.* to write:

```
let f x y z = if x then fT__ y z else fF__ y z
```

Where functions `fT__` and `fF__` are defined by pattern matching:

```
let fT__ y z =              let fF__ y z =
  match y,z with              match y,z with
  | F,T -> 1                  | F,T -> 1
  | _,F -> 3                  | T,_ -> 2
  | _,T -> 4                  | _,F -> 3
                              | _,T -> 4
```

The matchings above are built from the initial matching, by selecting the rows that still can be matched once the value of `x` is known.

Compilation goes on by considering `y` and `z`, resulting in the following low-level `f1`:

```
let f1 x y z =
  if x then
    if y then
      if z then 4 else 3
    else
      if z then 1 else 3
  else
    if y then 2
    else
      if z then 1 else 3
```

We can do a little better, by introducing a local function definition to share the common subexpression **if** `z` **then** `1` **else** `3`.

[1] We use OCaml syntax.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ML'08, September 21, 2008, Victoria, BC, Canada.
Copyright © 2008 ACM 978-1-60558-062-3/08/09...$5.00

But we can do even better, by testing y first, and then x first when y is true, resulting in the second low-level f2.

```
let f2 x y z =
  if y then
    if x then
      if z then 4 else 3
    else 2
  else
    if z then 1 else 3
```

Function f2 is obviously more concise than f1. It is also more efficient. More specifically, when y is false, function f2 performs 2 tests, whereas function f2 performs 3 tests. Otherwise, the two functions perform the same tests. Choosing a good subterm testing order is the task of match-compiler *heuristics*.

In this paper we tackle the issue of producing f2 and not f1 automatically. We do so first from the point of view of theory, by defining *necessity*. Necessity is borrowed to the theory of lazy pattern matching: a subterm (of subject values) is *needed* when its reduction is mandatory for the lazy semantics of matching to yield a result. Instead, we define necessity by universal quantification over decision trees: a subterm is needed when it is examined by all possible decision trees. Necessity provides inspiration and justification for new heuristics, which we study experimentally.

2. Simplified source language

Most ML values can be defined as sorted ground terms over some signatures. Signatures are introduced by (algebraic) data type definitions. In our context, a *signature* is thus the complete set of the constructors for a datatype. We omit type definitions and consider the following values:

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

An implicit typing discipline is assumed. In particular, a constructor c has a fixed *arity*, written a above. In the following, we shall adopt the convention of writing a for the arity of constructor c — and a' for the arity of c' etc. Moreover, one knows which signature constructor c belongs to. In examples, we omit $()$ after constants constructors. *i.e.* we write Nil, true, 0, etc.

We also consider the usual *occurrences*. Occurrences are sequences of integers that describe the positions of subterms. More precisely an occurrence is either empty, written Λ, or is an integer k followed by an occurrence o, written $k{\cdot}o$. Occurrences are paths to subterms, in the following sense:

$$\begin{array}{rcll} v/\Lambda & = & v & \\ c(v_1, \ldots, v_a)/k{\cdot}o & = & v_k/o & (1 \leq k \leq a) \end{array}$$

Following common practice we omit the terminal Λ of non-empty occurrences. We assume familiarity with the standard notions of prefix (*i.e.* o_1 is a prefix of o_2 when v/o_2 is a subterm of v/o_1), and of incompatibility (*i.e.* o_1 and o_2 are incompatible when o_1 is not a prefix of o_2 nor o_2 is a prefix of o_1). We consider the *leftmost-outermost* ordering over occurrences that corresponds to the standard lexicographic ordering over sequences of integers, and also to the standard prefix depth-first ordering over subterms.

We use the following simplified definition of patterns:

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

The main simplification is replacing all variables with wildcards. Formally, a wildcard is a variable with a unique, otherwise irrelevant, name. In the following, we shall consider pattern vectors, $\vec{p}$, which are sequences of patterns $(p_1 \cdots p_n)$, pattern matrices P, and matrices of clauses $P \to A$:

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

By convention, vectors are of size n, matrices are of size $m \times n$ — m is the height and n the width. Pattern matrices are natural and convenient in the context of pattern matching compilation. Indeed, they express the simultaneous matching of several values. In clauses, actions a^j are integers. Row j of matrix P is sometimes depicted as $\vec{p}^{\,j}$.

Clause matrices are an abstraction of pattern matching expressions as can be found in ML programs. Abstraction consists in replacing the expressions of ML with integers, which are sufficient to our purpose. Thereby, we avoid the complexity of describing the semantics of ML (Milner et al. 1990; Owens 2008), still preserving a decent level of precision.

2.1 Semantics of ML matching

Generally speaking, value v is an *instance* of pattern p, written $p \preceq v$, when there exists a substitution σ, such that $\sigma(p) = v$. In the case of linear patterns, the aforementioned instance relation is equivalent to the following inductive definition:

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

Please note that the last line above defines the instance relation for vectors.

We also give an explicit definition of the relation "value v is not an instance of pattern p", written $p \# v$:

$$\begin{array}{rcll} (p_1 \mid p_2) & \# & v & \text{iff } p_1 \# v \text{ and } p_2 \# v \\ c(p_1, \ldots, p_a) & \# & c(v_1, \ldots, v_a) & \text{iff } (p_1 \cdots p_a) \# (v_1 \cdots v_a) \\ (p_1 \cdots p_a) & \# & (v_1 \cdots v_a) & \text{iff there exists } i, p_i \# v_i \\ c(p_1, \ldots, p_a) & \# & c'(v_1, \ldots, v_{a'}) & \text{with } c \neq c' \end{array}$$

For the values and patterns that we considered so far, relation # is the negation of $\preceq$. However this will not remain true, so we rather adopt a non-ambiguous notation.

DEFINITION 1 (ML matching). *Let P be a pattern matrix of width n and height m. Let $\vec{v}$ be a value vector of size n. Let j be a row index ($1 \leq j \leq m$).*

Row j of P filters $\vec{v}$ (or equivalently, vector $\vec{v}$ matches *row j), when the following two propositions hold:*

1. *Vector $\vec{v}$ is an instance of $\vec{p}^{\,j}$. (written $\vec{p}^{\,j} \preceq \vec{v}$).*
2. *For all j', $1 \leq j' < j$, vector $\vec{v}$ is not an instance of $\vec{p}^{\,j'}$ (written $\vec{p}^{\,j'} \# \vec{v}$).*

Furthermore, let $P \to A$ be a clause matrix. If row j of P filters $\vec{v}$, we write:

$$\mathit{Match}[\vec{v}, P \to A] \stackrel{\text{def}}{=} a^j$$

The definition above captures the intuition behind textual priority rule of ML: attempt matches from top to bottom, stopping as soon as a match is found.

2.2 Matrix decomposition

The compilation process transforms clause matrices by the means of two basic decomposition operations, defined in Figure 1. The first operation is *specialization* by a constructor c, written $\mathcal{S}(c, P \to$

Pattern p_1^j	Row(s) of $\mathcal{S}(c, P \to A)$
$c(q_1, \ldots, q_a)$	$q_1 \cdots q_a \quad p_2^j \cdots p_n^j \;\to\; a^j$
$c'(q_1, \ldots, q_{a'})$ $(c' \neq c)$	No row
$_$	$\overbrace{_ \cdots _}^{\times a} \quad p_2^j \cdots p_n^j \;\to\; a^j$
$(q_1 \mid q_2)$	$\begin{pmatrix} \mathcal{S}(c, (q_1\, p_2^j \cdots p_n^j \to a^j)) \\ \mathcal{S}(c, (q_2\, p_2^j \cdots p_n^j \to a^j)) \end{pmatrix}$

Row p_1^j	Row(s) of $\mathcal{D}(P)$
$c(q_1, \ldots, q_a)$	No row
$_$	$p_2^j \cdots p_n^j \;\to\; a^j$
$(q_1 \mid q_2)$	$\begin{pmatrix} \mathcal{D}(q_1\, p_2^j \cdots p_n^j \to a^j) \\ \mathcal{D}(q_2\, p_2^j \cdots p_n^j \to a^j) \end{pmatrix}$

Figure 1. Matrix decomposition

A), (left of Figure 1) and the second operation computes a default matrix, written $\mathcal{D}(P \to A)$ (right of Figure 1). Both transformations apply to the rows of $P \to A$, taking order into account, and yield the rows of the new matrices. Generally speaking, the transformations simplify the initial matrix by erasing some rows, although or-pattern expansion can formally increase the number of rows.

Specialization by constructor c simplifies matrix P under the assumption that v_1 admits c as a head constructor. For instance, given the following clause matrix:

$$P \to A \stackrel{\mathsf{def}}{=} \begin{pmatrix} \texttt{[]} & _ & \to 1 \\ _ & \texttt{[]} & \to 2 \\ _\texttt{::}_ & _\texttt{::}_ & \to 3 \end{pmatrix}$$

we have:

$$\mathcal{S}((\texttt{::}), P \to A) = \begin{pmatrix} _ & _ & \texttt{[]} & \to 2 \\ _ & _ & _\texttt{::}_ & \to 3 \end{pmatrix}$$

$$\mathcal{S}(\texttt{[]}, P \to A) = \begin{pmatrix} _ & \to 1 \\ \texttt{[]} & \to 2 \end{pmatrix}$$

It is to be noticed that row number 2 of $P \to A$ finds its way into both specialized matrices. This is so because its first pattern is a wildcard.

The default matrix retains the rows of P whose first pattern p_1^j admits all values $c'(v_1, \ldots, v_a)$ as instances, where constructor c' is not present in the first column of P. Let us define:

$$Q \to B \stackrel{\mathsf{def}}{=} \begin{pmatrix} \texttt{[]} & _ & \to 1 \\ _ & \texttt{[]} & \to 2 \\ _ & _ & \to 3 \end{pmatrix}$$

Then we have:

$$\mathcal{D}(Q \to B) = \begin{pmatrix} \texttt{[]} & \to 2 \\ _ & \to 3 \end{pmatrix}$$

The following lemma reveals the semantic purpose of decomposition. More precisely, specialization $\mathcal{S}(c, P \to A)$ expresses exactly what remains to be matched, once it is known that v_1 admits c as a head constructor; while the default matrix expresses what remains to be matched, once it is known that the head constructor of v_1 does not appear in the first column of P.

LEMMA 1 (Key properties of matrix decompositions). *Let $P \to A$ be a clause matrix.*

1. *For any constructor c, the following equivalence holds:*

$$\mathit{Match}[(c(w_1, \ldots, w_a)\; v_2 \cdots v_n), P \to A] = k$$
$$\Updownarrow$$
$$\mathit{Match}[(w_1 \cdots w_a\; v_2 \cdots v_n), \mathcal{S}(c, P \to A)] = k$$

Where $w_1, \ldots w_a$ and $v_2, \ldots, v_n$ are any values of appropriate types.

2. *Let c be a constructor that does not appear as a head constructor of the patterns of the first column of P. For all values $w_1, \ldots, w_a, v_2, \ldots, v_n$ of appropriate types, we have the equivalence:*

$$\mathit{Match}[(c(w_1, \ldots, w_a)\; v_2 \cdots v_n), P \to A] = k$$
$$\Updownarrow$$
$$\mathit{Match}[(v_2 \cdots v_n), \mathcal{D}(P \to A)] = k$$

Proof: Mechanical application of definitions. Q.E.D.

3. Target language

Decision trees are the following terms:

$\mathcal{A}$ ::=	**Decision trees**
$\mathsf{Leaf}(k)$	success (k is an action, an integer)
Fail	failure
$\mathsf{Switch}_o(\mathcal{L})$	multi-way test (o is an occurrence)
$\mathsf{Swap}_i(\mathcal{A})$	stack swap (i is an integer)

Decision tree structure is clearly visible, with multi-way tests being $\mathsf{Switch}_o(\mathcal{L})$, and leaves being $\mathsf{Leaf}(k)$ and Fail. The additional nodes $\mathsf{Swap}_i(\mathcal{A})$ are not part of tree structure strictly speaking. Instead, they are control instructions for evaluating decision trees.

Switch case lists ($\mathcal{L}$ above) are non-empty lists of pairs constituted by a constructor and a decision tree, written $c{:}\mathcal{A}$. The list may end with an optional *default case*, written $\texttt{*}{:}\mathcal{A}$:

$$\mathcal{L} ::= c_1{:}\mathcal{A}_1; \cdots; c_z{:}A_z; [\texttt{*}{:}\mathcal{A}]?$$

We shall assume well-formed switches in the following sense:

1. Constructors c_k are in the same signature, and are distinct.
2. The default case is present, if and only if the set $\{ c_1, \ldots, c_z \}$ is not a signature, *i.e.* when there exists some constructor c that does not appear in $\{ c_1, \ldots, c_z \}$

For the sake of precise proofs, we give a semantics for evaluating decision trees (Figure 2). Decision trees are evaluated with respect to a stack of values. The stack initially holds the subject value. An evaluation judgment $\vec{v} \vdash \mathcal{A} \hookrightarrow k$ is to be understood as "evaluating tree $\mathcal{A}$ w.r.t. stack $\vec{v}$ results in the action k".

Evaluation is over at tree leaves (rule MATCH). The heart of the evaluation is at switch nodes, as described by the two rules SWITCHCONSTR and SWITCHDEFAULT. Case selection is performed by auxiliary rules that express nothing more than search in an association list. In rule CONT, we use the meta-notation $[c|*]$ to represent either constructor c or the special constant $*$ that signals default case selection. Since switches are well-formed, the

Rules for decision trees

(MATCH)
$$\vec{v} \vdash \mathsf{Leaf}(k) \ \hookrightarrow\ k$$

(SWAP)
$$\frac{(v_i\ \cdots\ v_1\ \cdots\ v_n) \vdash \mathcal{A} \ \hookrightarrow\ k}{(v_1\ \cdots\ v_i\ \cdots\ v_n) \vdash \mathsf{Swap}_i(\mathcal{A}) \ \hookrightarrow\ k}$$

(SWITCHCONSTR)
$$\frac{c \vdash \mathcal{L} \ \hookrightarrow\ c{:}\mathcal{A} \qquad (w_1\ \cdots\ w_a\ v_2\ \cdots\ v_n) \vdash \mathcal{A} \ \hookrightarrow\ k}{(c(w_1, \ldots, w_a)\ v_2\ \cdots\ v_n) \vdash \mathsf{Switch}_o(\mathcal{L}) \ \hookrightarrow\ k}$$

(SWITCHDEFAULT)
$$\frac{c \vdash \mathcal{L} \ \hookrightarrow\ *{:}\mathcal{A} \qquad (v_2\ \cdots\ v_n) \vdash \mathcal{A} \ \hookrightarrow\ k}{(c(w_1, \ldots, w_a)\ v_2\ \cdots\ v_n) \vdash \mathsf{Switch}_o(\mathcal{L}) \ \hookrightarrow\ k}$$

Auxiliary rules for case selection

(FOUND)
$$c \vdash c{:}\mathcal{A}; \mathcal{L} \ \hookrightarrow\ c{:}\mathcal{A}$$

(DEFAULT)
$$c \vdash *{:}\mathcal{A} \ \hookrightarrow\ *{:}\mathcal{A}$$

(CONT)
$$\frac{c \neq c' \qquad c \vdash \mathcal{L} \ \hookrightarrow\ [c|*]{:}\mathcal{A}}{c \vdash c'{:}\mathcal{A}; \mathcal{L} \ \hookrightarrow\ [c|*]{:}\mathcal{A}}$$

Figure 2. Evaluation of decision trees

search always succeeds. That is, the auxiliary rules for case selection are complete. It is to be noticed that switches always examine the value on top of the stack, *i.e.* value v_1. It is also to be noticed that the occurrence o in $\mathsf{Switch}_o(\mathcal{L})$ serves no purpose during evaluation. At the moment, occurrences are informative tags on switch nodes. The two rules SWITCHCONSTR and SWITCHDEFAULT differ significantly concerning what is made of the arguments of v_1, which are either pushed or ignored. In all cases, the value examined is popped. Decision trees feature a node that performs an operation on the stack: $\mathsf{Swap}_i(\mathcal{A})$ (rule SWAP). This trick allows the examination of any value v_i from the stack, by the combination $\mathsf{Swap}_i(\mathsf{Switch}_o(\mathcal{L}))$.

Finally, since there is no rule to evaluate Fail nodes, match failures and all other errors (such as induced by ill-formed stacks) are not distinguished by the semantics. Such a confusion of errors is not harmful in our simple setting.

4. Compilation scheme

Before describing compilation to decision trees proper, we settle the issue of matching order for or-pattern arguments. As a matter of fact, the definition of $(p_1 \,|\, p_2) \preceq v$ as $p_1 \preceq v$ or $p_2 \preceq v$ is slightly ambiguous in the presence of variables. Consider:

```
let f xs = match ys with (_::ys|ys) -> ys
```

Without additional specification, the value of `f [1 ; 2]` can be either `[2]` (the first alternative is matched) or `[1 ; 2]` (the second alternative is matched). We claim that the first result is more natural, because it can be expressed as a left-to-right expansion rule. That is, function `f` above is equivalent to:

```
let f xs = match ys with
| _::ys -> ys
| ys -> ys
```

The left-to-right expansion rule serves as a basis for compiling or-patterns. As a consequence of expansion, $(_ \,|\, p_1 \,|\, \cdots \,|\, p_n)$ can be replaced by _ before compilation takes place. Moreover, we define the following *generalized constructor patterns*:

$q ::=$	**Generalized constructor patterns**
$c(p_1, \ldots, p_a)$	(p_k's are any patterns)
$(q \,|\, p)$	(p is any pattern)

Then, by preprocessing of or-patterns, any pattern is either a generalized constructor pattern or a wildcard.

Compilation scheme $\mathcal{CC}$ is described as a non-deterministic function that takes two arguments: a vector of occurrences $\vec{o}$ and a clause matrix. The occurrences of $\vec{o}$ define the *fringe*, that is, the subterms of the subject value that need to be checked against the patterns of P to decide matching. The fringe $\vec{o}$ is the compile-time counterpart of the stack $\vec{v}$ used during evaluation. More precisely we have $v_i = v/o_i$, where v is the subject value.

Compilation is defined by cases as follows.

1. If matrix P has no row (*i.e.* $m = 0$) then matching always fails, since there is no row to match.

$$\mathcal{CC}(\vec{o}, \emptyset \rightarrow A) \stackrel{\mathsf{def}}{=} \mathsf{Fail}$$

2. If the first row of P exists and is constituted by wildcards, then matching always succeeds and yields the first action.

$$\mathcal{CC}(\vec{o}, \begin{pmatrix} _ & \cdots & _ & \rightarrow & a^1 \\ p_1^2 & \cdots & p_n^2 & \rightarrow & a^2 \\ & & \vdots & & \\ p_1^m & \cdots & p_n^m & \rightarrow & a^m \end{pmatrix}) \stackrel{\mathsf{def}}{=} \mathsf{Leaf}(a^1)$$

In particular, this case applies when there is at least one row ($m > 0$) and no column ($n = 0$).

3. In any other case, matrix P has at least one row and at least one column ($m > 0$, $n > 0$). Furthermore, there exists at least one column of which at least one pattern is not a wildcard. Select one such column i.

   (a) Let us first consider the case when i is 1. Define Σ_1 the set of the head constructors of the patterns in column 1.

$$\Sigma_1 \stackrel{\mathsf{def}}{=} \cup_{1 \leq j \leq m} \mathcal{H}(p_1^j)$$

$$\mathcal{H}(_) \stackrel{\mathsf{def}}{=} \emptyset \qquad \mathcal{H}(c(\ldots)) \stackrel{\mathsf{def}}{=} \{\, c \,\}$$

$$\mathcal{H}((q_1 \,|\, q_2)) \stackrel{\mathsf{def}}{=} \mathcal{H}(q_1) \cup \mathcal{H}(q_2)$$

Let $c_1, \ldots, c_z$ be the elements of Σ_1. By hypothesis, Σ_1 is not empty ($z \geq 1$). For each constructor c_k in Σ_1, perform the following inductive call that yields decision tree $\mathcal{A}_k$:

$$\mathcal{A}_k \stackrel{\mathsf{def}}{=} \mathcal{CC}((o_1{\cdot}1\ \cdots\ o_1{\cdot}a \quad o_2\ \cdots\ o_n), \mathcal{S}(c_k, P \rightarrow A))$$

The notation a above stands for the arity of c_k. Notice that o_1 disappears from the occurrence vector, being replaced with the occurrences $o_1{\cdot}1, \ldots o_1{\cdot}a$. The decision trees $\mathcal{A}_k$ are grouped into a case list $\mathcal{L}$:

$$\mathcal{L} \stackrel{\mathsf{def}}{=} c_1{:}A_1; \cdots; c_z{:}\mathcal{A}_z$$

If Σ_1 is not a signature, an additional recursive call is performed on the default matrix, so as handle the constructors that do not appear in Σ_1. Accordingly, the switch case list is

completed with a default case:

$$\mathcal{A}_\mathcal{D} \stackrel{\text{def}}{=} \mathcal{CC}((o_2 \cdots o_n), \mathcal{D}(P \to A))$$

$$\mathcal{L} \stackrel{\text{def}}{=} c_1{:}A_1; \cdots; c_z{:}\mathcal{A}_z; *{:}\mathcal{A}_\mathcal{D}$$

Finally compilation yields a switch that tests occurrence o_1:

$$\mathcal{CC}(\vec{o}, P \to A) \stackrel{\text{def}}{=} \mathsf{Switch}_{o_1}(\mathcal{L})$$

(b) If $i > 1$ then swap columns 1 and i in both $\vec{o}$ and P, yielding $\vec{o}'$ and P'. Then compute $\mathcal{A}' = \mathcal{CC}(\vec{o}', P' \to A)$ as above, yielding decision tree $\mathcal{A}'$, and define:

$$\mathcal{CC}(\vec{o}, P \to A) \stackrel{\text{def}}{=} \mathsf{Swap}_i(\mathcal{A}')$$

Notice that the function $\mathcal{CC}$ is non-deterministic because of the unspecified choice of i at step 3.

Compilation to decision tree is a simple algorithm: inductive step 3 above selects a column i (*i.e.* a subterm v/o_i in the subject value), the head constructor of v/o_i is examined, and compilation goes on, considering all possible constructors.

One crucial property of decision trees is that no subterm v/o_i is examined more than once. The property is made trivial by decision tree semantics — evaluation of a switch pops the examined value. It should also be observed that if the components o_i of $\vec{o}$ are pairwise incompatible occurrences, then the property still holds for recursive calls.

We have already noticed that the occurrence o that decorates a switch node $\mathsf{Switch}_o(\mathcal{L})$ plays no part during evaluation. We can now further notice that occurrences o_i are not necessary to define the compilation scheme $\mathcal{CC}$. Hence, we often omit occurrences, writing $\mathcal{CC}(P \to A)$ and $\mathsf{Switch}(\mathcal{L})$. At the moment, occurrences provide extra intuition on decision trees: they tell which subterm of the subject value is examined by a given $\mathsf{Switch}_o(\mathcal{L})$ node.

Naive compilation is defined by the trivial choice function that selects the minimal i, such that at least one pattern p_i^j is not a wildcard. It is not difficult to see that o_i is minimal for the leftmost-outermost ordering on occurrences, among the occurrences o_k such that at least one pattern p_k^j is not a wildcard.

A real match compiler can be written by following compilation scheme $\mathcal{CC}$. There are differences though. First, the real compiler targets a more complex language. More precisely, the occurrence vector $\vec{o}$ is replaced with a variable vector $\vec{x}$ and the target language has explicit local bindings in place of a simple stack. Furthermore, real multi-way tests are more flexible: they operate on any variable (not only on top of stack). Second, decision trees produced by the real compiler are implemented as dags with maximal sharing — see (Pettersson 1992) for a detailed account.

In spite of these differences, our simplified decision trees offer a good approximation of actual matching automata, especially if we represent them as pictures, while omitting $\mathsf{Swap}_i(\mathcal{A})$ nodes.

5. Correctness

The main interest for having defined decision tree semantics will appear later, while considering semantic properties subtler than simple correctness. Nevertheless, we state a correctness result for scheme $\mathcal{CC}$.

PROPOSITION 1. *Let $P \to A$ be a clause matrix. Then we have:*

1. *If for some value vector $\vec{v}$, we have $\mathit{Match}[\vec{v}, P \to A] = k$, then for all decision trees $\mathcal{A} = \mathcal{CC}(P \to A)$, we have $\vec{v} \vdash \mathcal{A} \hookrightarrow k$.*
2. *If for some value vector $\vec{v}$ and decision tree $\mathcal{A} = \mathcal{CC}(P \to A)$ we have $\vec{v} \vdash \mathcal{A} \hookrightarrow k$, then we have $\mathit{Match}[\vec{v}, P \to A] = k$.*

Proof: Consequence of Lemma 1 and by induction over $\mathcal{A}$ construction. Q.E.D.

It is to be noticed that, as a corollary, the non-determinism of $\mathcal{CC}$ has no semantic impact: whatever column choices are, the produced decision tree implements ML matching faithfully.

6. Examples

Let us consider a frequently found pattern matching expression, which any match compiler should probably compile optimally.

EXAMPLE 1. *The classical merge of two lists:*

```
let rec merge = match xs,ys with
| [],_ -> ys
| _,[] -> xs
| x::rx,y::ry -> ...
```

Focusing on pattern matching compilation, we only consider the following "occurrence[2]" vector and clause matrix.

$$\vec{o} = (\mathtt{xs}\ \mathtt{ys}) \qquad P \to A = \begin{pmatrix} \mathtt{[]} & _ & \to 1 \\ _ & \mathtt{[]} & \to 2 \\ _\mathtt{::}_ & _\mathtt{::}_ & \to 3 \end{pmatrix}$$

The compiler now has to choose a column to perform matrix decomposition. That is, the resulting decision tree will either examine xs first, or ys first.

Let us first consider examining xs. We have $\Sigma_1 = \{\, \mathtt{::}, \mathtt{[]} \,\}$, a signature. We do not need to consider the default matrix, and we get:

$$\mathcal{CC}((\mathtt{xs}\ \mathtt{ys}), P \to A) = \mathsf{Switch}_{\mathtt{xs}}((\mathtt{::}){:}\mathcal{A}_1;\ \mathtt{[]}{:}\mathcal{A}_2)$$

Where:

$$\mathcal{A}_1 = \mathcal{CC}((\mathtt{xs}{\cdot}1\ \mathtt{xs}{\cdot}2\ \mathtt{ys}), \mathcal{S}((\mathtt{::}), P \to A))$$

$$\mathcal{A}_2 = \mathcal{CC}(\mathtt{ys}, \mathcal{S}(\mathtt{[]}, P \to A))$$

The rest of compilation is deterministic. Let us consider for instance $\mathcal{A}_1$. We have (section 2.2):

$$\mathcal{S}((\mathtt{::}), P \to A) = \begin{pmatrix} _ & _ & \mathtt{[]} & \to 2 \\ _ & _ & _\mathtt{::}_ & \to 3 \end{pmatrix}$$

Only the third column has non-wildcards, hence we get (by compilation steps 3, then 2)

$$\mathcal{A}_1 = \mathsf{Swap}_3(\mathsf{Switch}_{\mathtt{ys}}((\mathtt{::}){:}\mathsf{Leaf}(3);\ \mathtt{[]}{:}\mathsf{Leaf}(2)))$$

Computing $\mathcal{A}_2$ is performed by a direct application of step 2:

$$\mathcal{S}(\mathtt{[]}, P \to A) = \begin{pmatrix} _ & \to 1 \\ \mathtt{[]} & \to 2 \end{pmatrix} \qquad \mathcal{A}_2 = \mathsf{Leaf}(1)$$

The resulting decision tree is best described as the picture of Figure 3. We now consider examining ys first. That is, we swap the

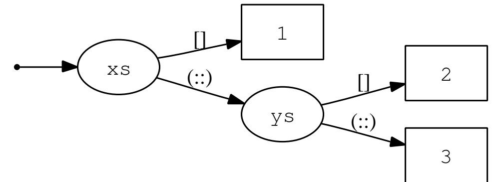

Figure 3. Compilation of list-merge, left-to-right

[2] In examples, initial occurrences are written as names. Formally we can define xs to be occurrence 1 and ys to be occurrence 2.

two columns of $\vec{o}$ and P, yielding the new arguments:

$$\vec{o}' = (\texttt{ys}\ \texttt{xs}) \qquad P' \rightarrow A = \begin{pmatrix} _ & \texttt{[]} & \rightarrow 1 \\ \texttt{[]} & _ & \rightarrow 2 \\ _\texttt{::}_ & _\texttt{::}_ & \rightarrow 3 \end{pmatrix}$$

Specialized matrices are as follows:

$$\mathcal{S}(\texttt{::}, P' \rightarrow A) = \begin{pmatrix} _ & _ & \texttt{[]} & \rightarrow 1 \\ _ & _ & _\texttt{::}_ & \rightarrow 3 \end{pmatrix}$$

$$\mathcal{S}(\texttt{[]}, P' \rightarrow A) = \begin{pmatrix} \texttt{[]} & \rightarrow 1 \\ _ & \rightarrow 2 \end{pmatrix}$$

Finally, compilation yields the decision tree of Figure 4. Notice

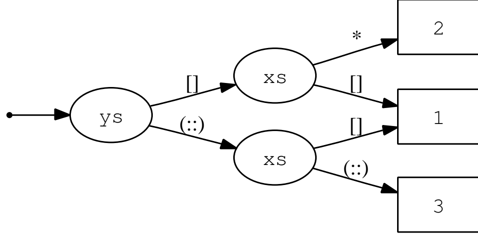

Figure 4. Compilation of list-merge, right-to-left

that the leaf "1" is pictured as shared, thereby reflecting actual implementation. The pictures clearly suggest that the left-to-right decision tree is better than the right-to-left one, in two important aspects.

1. The first decision tree is smaller. A simple measure of decision tree size is the number of internal nodes, that is, the number of switches.
2. The first decision tree is more efficient: if `xs` is the empty list `[]`, then the tree of Figure 3 reaches action 1 by performing one test only, while the tree of Figure 4 always performs two tests.

In this simple case, all decision trees are available and can be compared. A compiler cannot rely on such a *post-mortem* analysis, which can be extremely expensive. Instead, a compilation *heuristic* will select a column in P at every critical step 3, based upon properties of matrix P. Such properties should be simple, relatively cheap to compute and have a positive impact on the quality of the resulting decision tree.

Before we investigate heuristics any further, let us consider an example that illustrates the implementation of decision trees by dags with maximal sharing.

EXAMPLE 2. *Consider the following matching expression where* `[_]` *is OCaml pattern for "a list of one element" (*i.e. `_::[]`*):*

```
match xs,ys with  [_],_ -> 1 | _,[_] -> 2
```

Naive compilation of the example yields the decision tree that is depicted as a dag with maximal sharing in Figure 5. The dag of Figure 5 has $2{+}2 = 4$ switch nodes, where a plain tree implementation has $2 + 2 \times 2 = 6$ switch nodes. Now consider a simple generalization: a diagonal pattern matrix, of size $n \times n$ with $p_i^i = $ `[_]` and $p_i^j = _$ for $i \neq j$. It is not difficult to see that the dag representation of the naive decision tree has $2n$ switch nodes, where the plain tree representation has $2^{n+1} - 2$ switch nodes. Or-pattern compilation also benefits from maximal sharing. Let us consider for instance the n-tuple pattern `(1|2),...,(1|2)`. Compilation produces a tree with $2^n - 1$ switches and a dag with n switches. One may also remark that, for a clause matrix of one row, such that some of its patterns are or-patterns, maximal sharing makes column choice indifferent. While, without maximal sharing, or-patterns should better be expanded last. As a conclusion, maximal sharing is a simple idea that may yield important savings in code size.

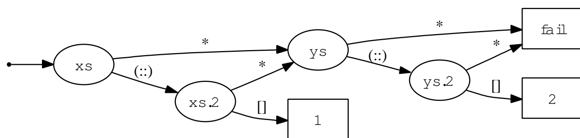

Figure 5. Decision tree as a dag with maximal sharing

Finally, here is a more a realistic example.

EXAMPLE 3. *This example is extracted from a bytecode machine for PCF (Plotkin 1977) (or mini-ML). Transitions of the machine depend upon the values of three items: accumulator* `a`*, stack* `s` *and code* `c`*. The heart of a ML function that implements the machine consists in the following pattern matching expression:*

```
let rec run a s e c = match a,s,c with
  | _,_,Ldi i::c -> 1
  | _,_,Push::c  -> 2
  | Int n2,Val (Int n1)::s,IOp op::c -> 3
  | Int 0,_,Test (c2,_)::c -> 4
  | Int _,_,Test (_,c3)::c -> 5
  | _,_,Extend::c -> 6
  | _,_,Search k::c -> 7
  | _,_,Pushenv::c -> 8
  | _,Env e::s,Popenv::c -> 9
  | _,_,Mkclos cc::c -> 10
  | _,_,Mkclosrec cc::c -> 11
  | Clo (cc,ce),Val v::s,Apply::c -> 12
  | a,(Code c::Env e::s),[] -> 13
  | a,[],[] -> 14
```

Compiling the example naively yields the decision tree of Figure 6. There is no sharing, except at leaves. For the sake of clarity, all leaves are omitted, except $\mathsf{Leaf}(4)$. There are 56 switches in the decision tree of Figure 6. As to runtime efficiency, all paths to $\mathsf{Leaf}(4)$, are emphasized, showing that it takes 5 to 8 tests to reach action 4. Figure 7 gives another decision tree, constructed by systematic exploration of column choices. The size of the resulting tree (17 switches) is thus guaranteed to be minimal (as a tree not as a dag with maximal sharing). The tree of Figure 7 is also more efficient at runtime: it takes only 4 tests to reach action 4.

It is in fact not difficult to produce the minimal tree by local column choices only: the first initial choice should be to examine `c`, then a second choice should be to examine `c.1` when `c` is a non-empty list. In both cases, the right choice can be made by selecting a column i such that no pattern p_i^j is a wildcard for $1 \leq j \leq m$.

In the rest of the paper we generalize the idea, and then experimentally check its validity.

7. Necessity

Figures 3 and 4 give all the possible decision trees that result from compiling example 1. Now, let us examine the paths from root to leaves. In both trees, all paths to $\mathsf{Leaf}(2)$ and $\mathsf{Leaf}(3)$ traverse two switch nodes. By contrast, the paths to $\mathsf{Leaf}(1)$ traverse one switch node in the tree of Figure 3 and two switch nodes in the tree of Figure 4. We can argue that the first tree is better because it has shorter paths.

A path in a decision tree is an ordinary path from root to leaf. When a path traverses a node $\mathsf{Switch}_o(\mathcal{L})$ we say that the path tests

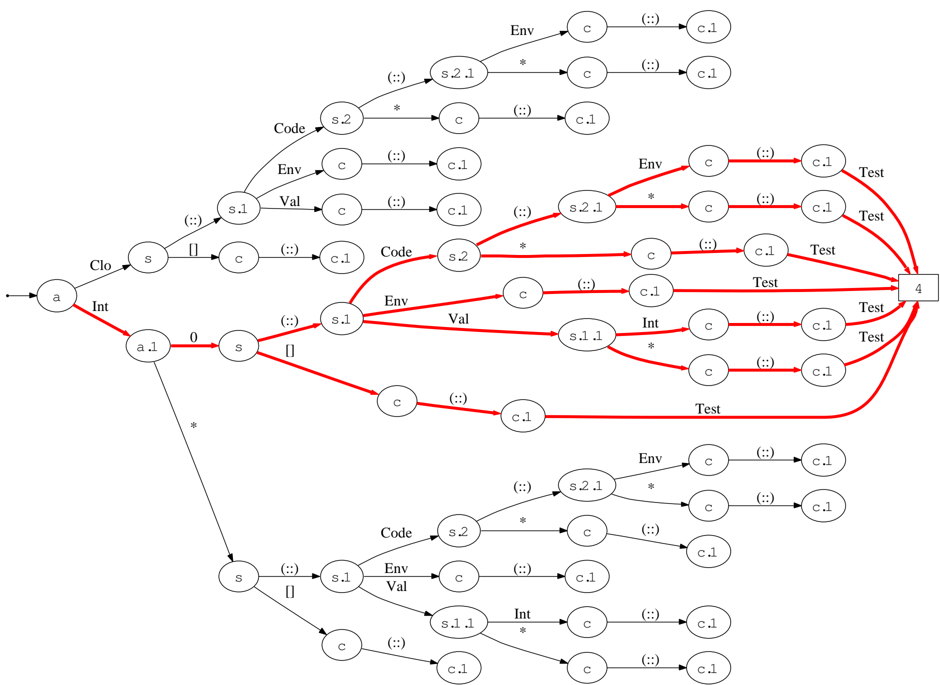

Figure 6. Naive decision tree for example 3

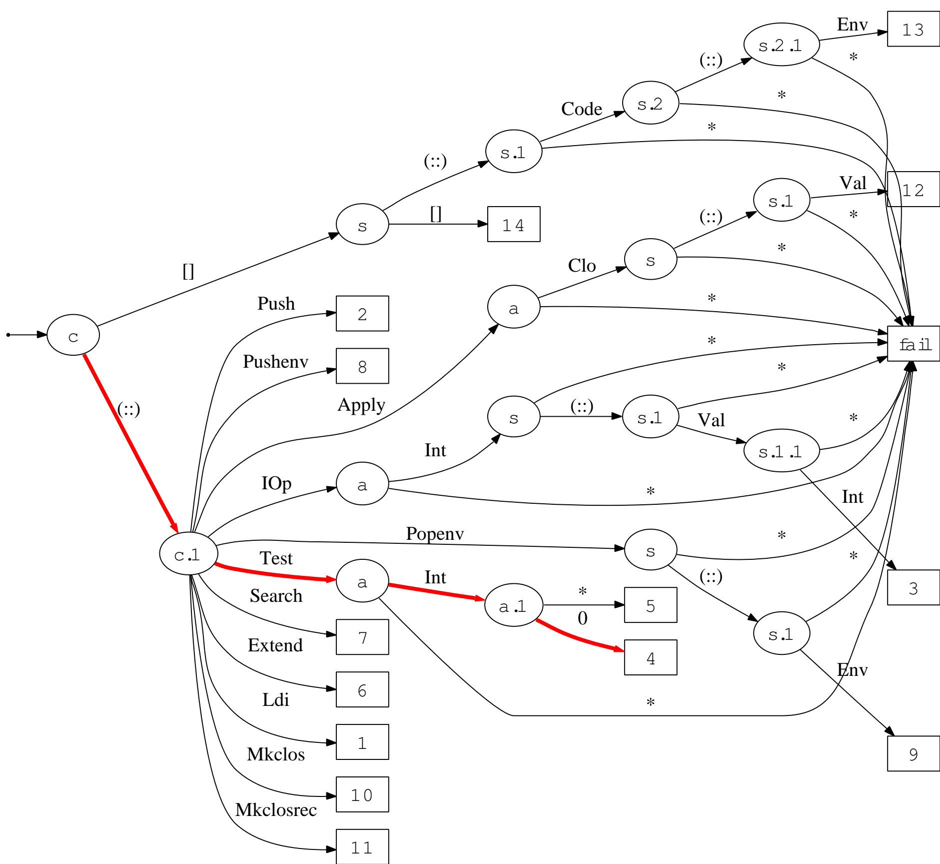

Figure 7. Minimal decision tree for example 3

the occurrence o. In the example, we note that xs is tested by all paths to $\mathsf{Leaf}(1)$, while ys is not.

DEFINITION 2 (Necessity). *Let P be a pattern matrix. Let $\vec{o}$ be the conventional occurrence vector, s.t. $o_i = i$, and let A be the conventional action vector, s.t. $a^j = j$. Column i is* needed for row j *when all paths to leaf* $\mathsf{Leaf}(j)$ *in all decision trees* $\mathcal{CC}(\vec{o}, P \rightarrow A)$ *test occurrence i. If column i is needed for all rows j, $1 \leq j \leq m$, column i is* needed *(for matching by P).*

Necessity-based heuristics will favor needed columns over non-needed ones, based upon the simple idea to perform immediately work that needs to be performed anyway. From this idea, we expect decision trees with shorter paths. Additionally, we may expect that decision trees with shorter paths also are smaller. In the list-merge example the following matrix summarizes necessity information:

$$N = \begin{pmatrix} \bullet & \\ \bullet & \bullet \\ \bullet & \bullet \end{pmatrix}$$

Where $n_i^j = \bullet$, if and only if column i is needed for row j. It is intuitively clear (and we shall prove it), that if p_i^j is a constructor pattern, then $n_i^j = \bullet$. However, this is not a necessary condition since we also have $p_1^2 = _$ and $n_1^2 = \bullet$. It is worth observing that, here, column 1 is needed, and that by selecting it we produce a decision tree with optimal path lengths (and optimal size).

More generally we can make the following two remarks:

1. Let P be a matrix with a needed column i and a non-needed column i'. Let $\mathcal{A}$ and $\mathcal{A}'$ be some trees resulting from selecting i and i' respectively. Obviously, all paths in $\mathcal{A}$ test o_i, while all paths in $\mathcal{A}'$ test $o_{i'}$. Moreover, by Definition 2, all paths in $\mathcal{A}'$ test o_i, whereas there may exist some paths in $\mathcal{A}$ that do not test $o_{i'}$. In fact, as a consequence of the forthcoming proposition 2, at least one such a path exists. Thus, selecting i seems to be a good idea.
2. Let a column made of constructor patterns only be a *strictly needed* column. If at every critical compilation step 3, we discover a strictly needed column and select it, then the resulting decision tree will possess no more switch nodes than the number of constructors in the original patterns. The remark is obvious since no pattern is copied during compilation.

Of course, these remarks are not sufficient when several or no needed columns exist. In any case, necessity looks like an interesting basis for designing heuristics. However, we should first be able to compute necessity from matrix P.

7.1 Computing necessity

The first step of our program is to relate the absence of a switch node on a given occurrence to decision tree evaluation. To that aim, we define a supplementary value $\lightning$ (reading "crash"), and consider extended value vectors $\vec{v}$, such that exactly one component is $\lightning$. That is, there exists an index ω, with $v_\omega = \lightning$ and $v_i \neq \lightning$ for $i \neq \omega$.

The semantics of decision tree evaluation (Fig. 2) is left unchanged. As a result, if the top of the stack v_1 is $\lightning$, then the evaluation of $\mathsf{Switch}_{o_1}(\mathcal{L})$ is blocked. We rather express the converse situation:

LEMMA 2. *Let $P \rightarrow A$ be a clause matrix with at least one row and at least one column ($n > 0, m > 0$). Let $\vec{o}$ be a vector of pairwise incompatible occurrences. Let finally $\mathcal{A} = \mathcal{CC}(\vec{o}, P \rightarrow A)$ be a decision tree. Then we have the equivalence: there exists an extended vector $\vec{v}$ with $v_\omega = \lightning$ and an action k such that $\vec{v} \vdash \mathcal{A} \hookrightarrow k$, if and only if $\mathcal{A}$ has a path to* $\mathsf{Leaf}(k)$ *that does not test o_ω*

Proof: See appendix Q.E.D.

The second step of our technique to compute necessity is to relate decision tree evaluation and matching for extended values. By contrast with decision tree semantics, which we left unchanged, we slightly alter matching semantics. The instance relation $\preceq$ is extended by adding the following two rules to the rules of section 2.1:

$$\begin{array}{rcll} _ & \preceq & \lightning & \\ (p_1 \mid p_2) & \preceq & \lightning & \text{iff } p_1 \preceq \lightning \end{array}$$

Clearly, we have $p \preceq \lightning$, if and only if p is not a generalized constructor pattern, that is, given our preprocessing phase that simplifies $(_ \mid p)$ as $_$, if and only if p is a wildcard. The two rules above are the only extensions performed, in particular $p \# \lightning$ never holds, whatever pattern p is. It is then routine to extend definition 1 of ML matching and lemma 1 (key properties of decompositions).

We now alter the correctness statement of compilation (Proposition 1) so as to handle extended values.

LEMMA 3. *Let $P \rightarrow A$ be a clause matrix. The following two implications hold:*

1. *Let $\vec{v}$ be an extended value vector such that $\mathsf{Match}[\vec{v}, P \rightarrow A] = k$. Then, there exists a decision tree $\mathcal{A} = \mathcal{CC}(P \rightarrow A)$ such that $\vec{v} \vdash \mathcal{A} \hookrightarrow k$.*
2. *If for some extended value vector $\vec{v}$ and decision tree $\mathcal{A} = \mathcal{CC}(P \rightarrow A)$ we have $\vec{v} \vdash \mathcal{A} \hookrightarrow k$, then we also have $\mathsf{Match}[\vec{v}, P \rightarrow A] = k$.*

Proof: See appendix Q.E.D.

Item 1 above differs significantly from the corresponding item in Proposition 1, since existential quantification replaces universal quantification. In other words, if an extended value matches some row in P, then all decision trees may not be correct, but there is at least one that is.

Finally, one easily relates matching of extended values and matching of ordinary values.

LEMMA 4. *Let P be a pattern matrix with rows and columns ($m > 0, n > 0$). Let $\vec{v}$ be an extended vector ($v_\omega = \lightning$), then row j of P filters $\vec{v}$, if and only if:*

1. *Pattern p_ω^j is not a generalized constructor pattern.*
2. *And row j of matrix P/ω filters $(v_1 \cdots v_{\omega-1}\ v_{\omega+1} \cdots v_n)$, where P/ω is P with column ω deleted:*

$$P/\omega = \begin{pmatrix} p_1^1 \cdots p_{\omega-1}^1 & p_{\omega+1}^1 \cdots p_n^1 \\ p_1^2 \cdots p_{\omega-1}^2 & p_{\omega+1}^2 \cdots p_n^2 \\ & \vdots \\ p_1^m \cdots p_{\omega-1}^m & p_{\omega+1}^m \cdots p_n^m \end{pmatrix}$$

Proof: Corollary of Definition 1 (ML matching). Q.E.D.

Finally, we reduce necessity to usefulness. By usefulness we here mean the usual usefulness notion diagnosed by ML compilers.

PROPOSITION 2. *Let P be a pattern matrix. Let i be a column index and j be a row index. Then, column i is needed for row j, if and only if one of the following two (mutually exclusive) propositions hold:*

1. *Pattern p_i^j is a generalized constructor pattern.*
2. *Or, pattern p_i^j is a wildcard, and row j of matrix P/i is useless*[3].

Proof: Corollary of the previous three lemmas. Q.E.D.

[3] redundant, in the terminology of Milner et al. (1990).

The proposition above not only makes necessity computable in practice, but also ensures equivalence with the necessity of lazy pattern matching (Maranget 1992, Lemma 5.1).

Consider again the example of list merge:

$$P \to A = \begin{pmatrix} \texttt{[]} & _ & \to 1 \\ _ & \texttt{[]} & \to 2 \\ _\texttt{::}_ & _\texttt{::}_ & \to 3 \end{pmatrix}$$

From the decision trees of Figures 3 and 4 we found that column 1 is needed for row 2. The same result follows by examination of the matrix $P/1$:

$$P/1 = \begin{pmatrix} _ \\ \texttt{[]} \\ _\texttt{::}_ \end{pmatrix}$$

Obviously, the second row of $P/1$ is useless, because of the initial wildcard. One could also have considered matrix P directly, remarking that no extended value vector $\vec{v} = (\lightning\ v_2)$ matches the second row of P, because $\vec{v}$ not being an instance of the first row of P (*i.e.* $(\texttt{[]}\ _)\ \#\ (\lightning\ v_2)$) implies $\texttt{[]}\ \#\ \lightning$, which is impossible.

Here is a slightly subtler example.

EXAMPLE 4. *Let $\vec{o}$ and P be the following occurrence vector and pattern matrix.*

$$\vec{o} = (\texttt{x}\ \texttt{y}) \qquad P = \begin{pmatrix} \texttt{true} & 1 \\ \texttt{false} & 2 \\ _ & _ \end{pmatrix} \qquad N = \begin{pmatrix} \bullet & \bullet \\ \bullet & \bullet \\ & \bullet \end{pmatrix}$$

N is the necessity matrix.

The first two rows of N are easy, because all the corresponding patterns in P are constructor patterns. To compute the third row of N, it is enough to to consider the following matrices:

$$P/1 = \begin{pmatrix} 1 \\ 2 \\ _ \end{pmatrix} \qquad P/2 = \begin{pmatrix} \texttt{true} \\ \texttt{false} \\ _ \end{pmatrix}$$

The third row of $P/1$ is useful, since it filters value 3 for instance; while the third row of $P/2$ is useless, since $\{\texttt{true}, \texttt{false}\}$ is a signature. Hence the necessity results for the third row: column x is not needed, while column y is. These results are confirmed by examining the paths to $\mathsf{Leaf}(3)$ in the two decision trees of Figure 8. One may argue that the second tree is better, since action 3 is reached without testing x when y differs from 1 and 2.

In the general case, it should probably be mentioned that the usefulness problem is NP-complete (Sekar et al. 1995). Nevertheless, our algorithm (Maranget 2007) for computing usefulness has been present in the OCaml compiler for years, and has proved efficient enough for input met in practice. The algorithm is inspired by compilation to decision trees, but is much more efficient, in particular with respect to space consumption and situations where a default matrix is present.

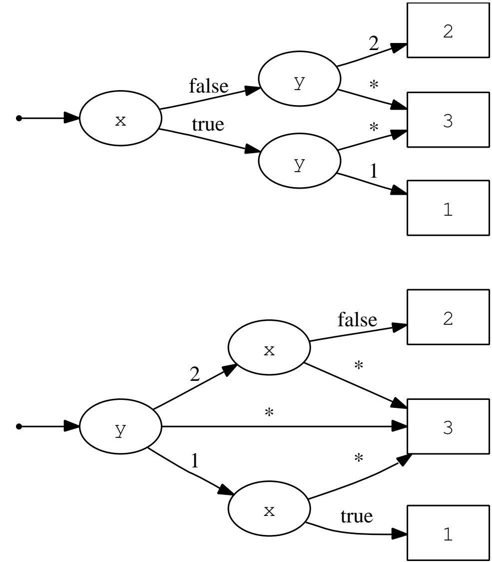

Figure 8. The two possible decision trees for example 4

8. Heuristics

8.1 Heuristic definitions

We first recall the heuristics selected by Scott and Ramsey (2000), which we adapt to our setting by considering generalized constructor patterns in place of constructor patterns. We identify each heuristic by a single letter (f, d, etc.) Each heuristic can be defined by the means of a score function, (also written f, d, etc.) from column indices to integers, with the heuristics selecting columns that *maximize* the score function. As regards the inventors and justifications of heuristics we refer to Scott and Ramsey (2000), except for the first row heuristic below, which we understand differently.

First row f Heuristic f favors columns i such that pattern p_i^1 is a generalized constructor pattern. In other words, the score function is $\mathsf{f}(i) = 0$ when $p_i^1 = _$, and $\mathsf{f}(i) = 1$ otherwise.

Heuristic f is based upon the idea that the first pattern row has much impact on the final decision tree. More specifically, if p_i^1 is a wildcard and column i is selected, then all decompositions will start by a clause with action a^1. As a result, every child of the emitted switch includes at least one leaf $\mathsf{Leaf}(a^1)$ and a path to it. Selecting i such that p_i^1 is a constructor pattern results in the more favorable situation where only one child includes $\mathsf{Leaf}(a^1)$ leaves. Baudinet and MacQueen (1985) describe the first row heuristic in a more complex way and call it the "*relevance*" heuristic. From close reading of their descriptions, we believe our simpler presentation to yield the same choices of columns, except, perhaps, for the marginal case of signatures of size 1 (*e.g.* pairs).

Small default d Given a column index i let $v(i)$ be the number of wildcard patterns in column i. The score function d is defined as $-v(i)$.

Small branching factor b Let Σ_i be the set of the head constructors of the patterns in column i. The score $\mathsf{b}(i)$ is the negation of the cardinal of Σ_i, minus one if Σ_i is not a signature. In other words, $\mathsf{b}(i)$ is the negation of the number of children of the $\mathsf{Switch}_{o_i}(\mathcal{L})$ node that is emitted by the compiler when it selects column i.

Arity a The score $\mathsf{a}(i)$ is the negation of the sum of the arities of the constructors of the Σ_i set.

Leaf edge ℓ The score $\ell(i)$ is the number of the children of the emitted $\mathsf{Switch}_{o_i}(\mathcal{L})$ node that are $\mathsf{Leaf}(a^k)$ leaves. This information can be computed naively, by first swapping columns 1 and i, then decomposing P (*i.e.* computing all specialized matrices, and the default matrix if applicable), and finally counting the decomposed matrices whose first rows are constituted by wildcards.

Rows r **(Fewer child rule)** The score function $\mathsf{r}(i)$ is the negation of the total number of rows in decomposed matrices. This information can be computed naively, by first swapping columns 1

and i, then decomposing P, and finally counting the numbers of rows of the resulting matrices.

We introduce three new heuristics based upon necessity.

Needed columns n The score $\mathsf{n}(i)$ is the number of rows j such that column i is needed for row j. The intention is quite clear: locally maximize the number of tests that are really useful.

Needed prefix p The score $\mathsf{p}(i)$ is the larger row index j such that column i is needed for all the rows j', $1 \leq j' \leq j$. As the previous one, this heuristics tends to favor needed columns. However, it further considers that earlier clauses (*i.e.* the ones with higher priorities) have more impact on decision tree size and path lengths than later ones.

Constructor prefix q This heuristic derives from the previous one, approximating "column i is needed for row j" by "p_i^j is a generalized constructor pattern". There are two ideas here: (1) avoid usefulness computations; and (2), avoid pattern copies. Namely, if column i is selected, then any row j such that p_i^j is a wildcard is copied. As a consequence, the other patterns in row j may be compiled more than once, regardless of whether column i is needed or not. Heuristic q can also be seen as a generalization of heuristic f.

Observe that heuristic d is a similar approximation of heuristic n.

It should be noticed that if matrix P has needed columns, then heuristics n and p will select these and only these. Similarly, if matrix P has strictly needed columns, then heuristics d and q will select these and only these. Heuristics n and p will also favor strictly needed columns but they will not distinguish them from other needed columns.

8.2 Combining heuristics

By design, heuristics select at least one column. However, a given heuristic may select several columns. Ties are broken first by composing heuristics. For instance, Baudinet and MacQueen (1985) seem to recommend the successive application of f, b and a, which we write fba. For instance, consider a variation on example 4.

$$P = \begin{pmatrix} \texttt{true} & \texttt{1} & _ \\ \texttt{false} & \texttt{2} & \texttt{[]} \\ _ & _ & _\texttt{::}_ \end{pmatrix}$$

Heuristic f selects columns 1 and 2. Amongst those, heuristic b selects column 1. Column selection being over, there is no need to apply heuristic a. The combination of heuristics is a simple technique to construct sophisticated heuristics out of simple ones. It should be noticed that combination order matters, since an early heuristic may eliminate columns that a following heuristics would champion.

Even when combined, heuristics may not succeed in selecting a unique column — consider a matrix with identical columns. We thus define the following three, last-resort, pseudo-heuristics:

Pseudo-heuristics N, L **and** R These select one column i amongst many, by selecting the minimal o_i in vector $\vec{o}$ according to various total orderings on occurrences. Heuristic N uses the leftmost ordering (this is naive compilation). Heuristics L and R first select shorter occurrences, and then break ties left-to-right or right-to-left, respectively. In other words, L and R are two variations on breadth-first ordering of the subterms of the subject value.

We call N, L and R pseudo-heuristics, because they do not examine matrix P and thus more rely on accidental presentation of matchings than on semantics. Varying the last-resort heuristic permits a more accurate evaluation of heuristics.

9. Performance

9.1 Methodology

We have written a prototype compiler that accepts pattern matrices and compiles them with various match compilers The implemented match compiler includes compilation to decision trees, both with and without maximal sharing and the optimizing compiler of Le Fessant and Maranget (2001), which is the standard OCaml match compiler. The prototype compiler targets matching automata, expressed as a simplified version of the first intermediate language of the OCaml compiler (Leroy et al. 2007). This target language features local bindings, indexed memory reads, switches, and local functions. Local functions implement maximal sharing or backtracking, depending upon the compilation algorithm enabled. We used the prototype to produce all the pictures in this paper, switch nodes being pictured as internal nodes, variables binding memory read expressions being used to decorate switch nodes, and local function definitions being rendered as nodes with several ingoing edges.

The performance of matching automata is estimated as follows:

1. Code size is estimated as the total number of switches.
2. Runtime performance is estimated as average path length. Ideally, average path length should be computed with respect to some distribution of subject values that is independent of the automaton considered. In practice, we compute average path length by assuming that: (1) all actions are equally likely[4] and (2), all constructors that can be found by a switch are equally likely[5].

To feed the prototype, we have extracted pattern matching expressions from a variety of OCaml programs, including the OCaml compiler itself, the Coq (Coq) and Why (Filliâtre 2008) proof assistants, and the Cil infrastructure for C program analysis (Necula et al. 2007). The selected matchings were identified by a modified OCaml compiler that performs match compilation by several algorithms and signals differences in number of switch generated — more specifically we used the standard OCaml match compiler and naive $\mathcal{CC}$ without sharing.

We finally have selected 54 pattern matching expressions, attempting to vary size and programming style (in particular, 35 expressions do not include or-patterns). The *test* of a heuristic consists in compiling the 54 expressions twice, ties left by the heuristic being broken by the pseudo-heuristics L and R. Each of these 108 compilations produces two pieces of data: automata size and average path length. We then compute the geometric means of data, considering ratios with respect to OCaml match compiler (base 100.0). Table 1 gives the results of testing single heuristics, figures being rounded to the nearest integer.

Results first demonstrate that sharing is mandatory for decision trees to compete with (optimized) backtracking automata as regards code size. Even more, with heuristics q, p and f, decision trees win over (optimized) backtracking automata. As regards path length, decision trees always win, with necessity heuristics p, q and n, yielding the best performance, heuristic f being quite close. Overall, heuristic q and p are the winners, but no clear winners.

For all cost measures, the pseudo-heuristics that base their choice on fringe occurrences only (*i.e.* N, L and R) behave poorly. Thus, to improve performance, real heuristics that analyze matrix P are called for. As a side note, the results also show a small bias of our test set against left-to-right ordering, since right-to-left R performs better then left-to-right L and N.

[4] Paths to Fail leaves are not considered.

[5] Except for integers, where the default case of a switch is considered as likely as the other cases.

	q	p	f	r	n	b	a	ℓ	R	d	N	L
Size (maximal sharing)	93	95	98	100	101	102	105	105	108	109	115	115
Size (no sharing)	122	124	129	122	126	151	161	134	135	138	150	168
Average path length	86	86	87	91	86	97	94	88	89	89	91	94

Table 1. Testing single heuristics

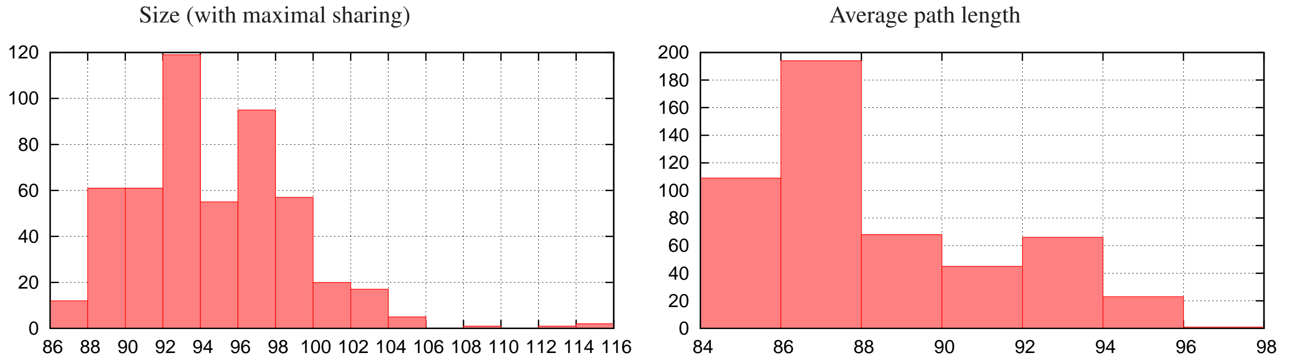

Table 2. Performance of combined heuristics, distribution by steps of 2.0.

9.2 Estimation of combined heuristics

We test all combinations of heuristics up to three heuristics, discarding a few combinations. Discarded combinations yield the same results as simpler combinations. For instance, qf and fq are equivalent to q. Overall, we test 507 combined heuristics. We summarize the results by the histograms of Table 2, showing the distribution of results. For instance, there are 12 combined heuristics that produce trees with sizes (as dags) in the range 86–88. These are qb[aℓ], fb[adℓr], fr, and fr[abdℓn].

The results show that combining heuristics yields significant improvements in size and little in path length. Good heuristics are the ones with performances in the best ranges for both size and path length. Unfortunately, no heuristic belongs to both ranges 86–88 for size and 84–88 for path length. We thus extend the acceptable size range to 86–90 and compute the intersection with the path length range 84–86. The process yields the unique champion heuristic pba. Intersecting size ranges 86–90 and path length range 84–88, yields 48 heuristics: fd[br], fnr, fr, fr[abdℓn], pb, pb[adℓnqr], pd[br],pnr, pq[br], pr, pr[abdℓnq], qb, qb[adℓnpr], qdr, qn[br], qp[br], qr, and qr[abdℓnp]. From those results, we draw a few conclusions:

1. Good primary heuristics are f, p and q. This demonstrates the importance of considering clause order in heuristics.
2. If we limit choice to combinations of at most two heuristics, r is a good complement to all primary heuristics. Heuristic b looks sufficient to break the ties left by p and q.
3. If we limit choice to heuristics that are simple to compute, that is if we eliminate n, p, r and ℓ, then good choices are fdb, qb and qb[ad]. Amongst those, qba is the only one with size in the best range.

As a personal conclusion, our favorite heuristic is the champion pba. If one wishes to avoid usefulness computations, we consider qba to be a good choice. To be fair, heuristics that are or were used in the SML/NJ compiler are not far away. From our understanding of its source code, the current version of the SML/NJ compiler uses heuristic fdb, earlier versions used fba. Heuristic fdb

	First PCF term: N	First PCF term: qba	Second PCF term: N	Second PCF term: qba
`ocamlc/ia32`	149	91	163	95
`ia32`	114	93	114	93
`ocamlc/amd64`	156	95	151	96
`amd64`	110	89	119	96

Table 3. Running times for the PCF interpreter (ratios of user time)

appears above, while fba misses it by little, with size in the range 86–88 and path length in the range 88–90.

9.3 Relation with actual performance

We have integrated compilation to decision trees (with maximal sharing) into the OCaml compiler.

Measuring the impact of varying pattern match compilation on actual code size and running time is a frustrating task. For instance, compiling the Coq system twice, with the standard OCaml match compiler and with $\mathcal{CC}$ guided by heuristic qba yields binaries which sizes differ by no more than 2%. Although some differences in the size of individual object files exist, those are not striking. This means that we lack material to analyze the impact of low level issues, such as how switches are compiled to machine code. Differences in compilation time and in running times of target programs are even more difficult to observe.

However, we cannot exclude the possibility of programs to which pattern matching matters. In fact we have one such program: an interpreter that runs the bytecode machine of example 3. As in previous experiments, differences in code size are dwarfed by non-pattern matching code. Differences in speed are observable by interpreting small PCF terms that are easy to parse and compile, but that run for long. We test two such terms, resulting in the ratios (still w.r.t. standard OCaml) of Table 3. Experiments are performed on two architectures: Pentium 1.4 Ghz and Xeon 3.0 Ghz, written `ia32` and `amd64`. For each architecture, the PCF interpreter is compiled to bytecode (by `ocamlc`) and to native code. For refer-

ence, average path length are 4.02 for the OCaml match compiler, 6.33 for N, and 3.14 for qba (ratios: 157 for N and 78 for qba). We see that differences in speed agree with differences in average path length. Moreover, running times are indeed bad if heuristics are neglected, especially for compilation to bytecode.

10. Related work

Maximal sharing can be achieved by the easy to implement and well established technique of *hash-consing* — see *e.g.* Filliâtre and Conchon (2006). With hash-consing, the asymptotic cost of producing the dag is about the same as the one of the tree. Some (Sekar et al. 1995; Nedjah and de Macedo Mourelle 2002) advocate another technique that do not suffer from this drawback. More precisely, they compute some key from the match compiler arguments, such that key identity implies tree identity. Such keys are also useful for establishing upper bounds on the size of the dag. In practice, hash-consing seems to be sufficient, both for the prototype and for the modified OCaml compiler.

Needed columns exactly are the *directions* of Laville (1988); Puel and Suárez (1989); Maranget (1992). All these authors build over lazy pattern semantics, adapting the seminal work of Huet and Lévy (1991). They mostly focus on the correct implementation of lazy pattern matching. By building over decision tree semantics, our present work leads more directly to heuristic design. Sekar et al. (1995) claim to have proved that selecting one of their *indices* (our needed columns) yields trees with shorter path lengths and smaller breadth (number of leaves in plain tree representation). Their results need a careful formulation in the general case, but are intuitively clear on example 4. The result on tree breadth is wrong, as demonstrated by the trees of Figure 8. The second decision tree is built by selecting needed columns and has breadth 5 (count edges to leaves), whereas the breadth of the first tree is 4. We conjecture the result on path lengths to be significant.

Scott and Ramsey (2000) study heuristics experimentally. We improve on them by designing and testing the necessity-based heuristics, and also by considering or-patterns and maximal sharing. We also differ in methodology: Scott and Ramsey (2000) count switch nodes and measure path lengths, as we do, but they do so for complete ML programs by instrumenting the SML/NJ compiler. As a result, their experiments are more expensive than ours, and they could not conduct systematic experiments on combination of three heuristics. Furthermore, by our prototype approach, we restrict the test set to matchings for which heuristics make a difference. Therefore, differences in measures are more striking. Of course, as regards actual compilation, we are in the same situation as Scott and Ramsey (2000): most often, heuristics do not make such a difference. However, heuristics matter to some of the tests of Scott and Ramsey (2000) (machine instruction recognizers). It would be particularly interesting to test the effect of necessity heuristics and of maximal sharing on those matchings, which, unfortunately, are not available.

11. Conclusion

Compilation to decision trees with maximal sharing, when guided by a good column heuristic, matches the performance of an optimizing compiler to backtracking automata, and can do better on some examples. Moreover, an optimizing compiler to decision trees is easier to implement than our own optimizing compiler to backtracking automata (Le Fessant and Maranget 2001). Namely, maximal sharing and simple heuristics (such as qba) are orthogonal extensions of the basic compilation scheme $\mathcal{CC}$. Thus, the resulting optimizing match compiler remains simple.

Designing optimizing match compilers that preserve more constrained semantics is a worthwhile direction for future research. In particular, a match compiler for Haskell must preserve the termination behavior of Augustsson (1985). Another example is the compilation of the active patterns of Syme et al. (2007). To that aim, the match compiler of Sestoft (1996) may be a valid starting point, because its definition follows ML matching semantics very closely.

References

Lennart Augustsson. Compiling pattern matching. In *Conference on Functional Programming Languages and Computer Architecture*, 1985.

Marianne Baudinet and David B. MacQueen. Tree pattern matching for ML. `http://www.smlnj.org/compiler-notes/85-note-baudinet.ps`, 1985.

Coq. The Coq proof assistant (v. 8.1). By the Coq team, `http://coq.inria.fr/`, 2007.

Jean-Christophe Filliâtre. The Why verification tool (v. 2.13). `http://why.lri.fr/`, 2008.

Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *Workshop on ML*. ACM Press, 2006.

Gérard Huet and Jean-Jacques Lévy. Call by need computations in non-ambiguous linear term rewriting systems. In Jean-Louis Lassez and Gordon D. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*. The MIT Press, 1991.

Alain Laville. Implementation of lazy pattern matching algorithms. In *European Symposium on Programming*. Springer-Verlag, 1988. LNCS 300.

Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *International Conference on Functional Programming*. ACM Press, 2001.

Xavier Leroy, Damien Doligez, Jacques Garrigue, Jérôme Vouillon, and Didier Rémy. The Objective Caml language (v. 3.10). `http://caml.inria.fr`, 2007.

Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17:387–422, May 2007.

Luc Maranget. Compiling lazy pattern matching. In *Conference on Lisp and Functional Programming*, 1992.

Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

George Necula, Scott McPeak, Westley Weimer, Ben Liblit, Matt Harren, Ramond To, and Aman Bhargava. Cil — infrastructure for C program analysis and transformation (v. 1.3.6). `http://manju.cs.berkeley.edu/cil/`, 2007.

Nadia Nedjah and Luiza de Macedo Mourelle. Optimal adaptive pattern matching. In *Developments in Applied Artificial Intelligence*, 2002. LNCS 2358.

Scott Owens. A sound semantics for OCaml-Light. In *European Symphosium On Programming*, 2008. LNCS 4960.

Mikael Pettersson. A term pattern-match compiler inspired by finite automata theory. In *Workshop on Compiler Construction*. Springer-Verlag, 1992. LNCS 641.

Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:225–255, December 1977.

Laurence Puel and Ascander Suárez. Compiling pattern matching by term decomposition. In *Conference on LISP and Functional Programming*. ACM Press, 1989.

Kevin Scott and Norman Ramsey. When do match-compilation heuristics matter? Technical Report CS-2000-13, University of Virginia, 2000.

R.C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. *SIAM Journal on Computing*, 24:1207–1243, December 1995.

Peter Sestoft. ML pattern matching compilation and partial evaluation. In *Dagstuhl Seminar on Partial Evaluation*. Springer-Verlag, 1996. LNCS 1110.

Son Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lighweight language extension. In *International Conferences on Functional Programming*. ACM Press, 2007.

Terese. *Term Rewriting Systems*. Cambridge University Press, 2003. Terese is Marc Bezem, Jan Willem Klop and Roel de Vrier.

A. Some proofs

A.1 Lemma 2

By induction on the construction of $\mathcal{A}$.

1. The case $m = 0$ is excluded by hypothesis.
2. If the first row of P consists of wildcards, then we have $\mathcal{A} = \mathsf{Leaf}(a^1)$. Then observe, on the one hand, that for any (extended) value vector $\vec{v}$, we have $\vec{v} \vdash \mathcal{A} \hookrightarrow a^1$ (rule MATCH, Fig 2); while, on the other hand, the only path to $\mathsf{Leaf}(a^1)$ is empty and thus does not traverse any $\mathsf{Switch}_{o_\omega}(\cdots)$ node.
3. $\mathcal{A}$ is produced by induction. There are two subcases.
   (a) If $\mathcal{A}$ is $\mathsf{Switch}_{o_1}(\mathcal{L})$. We first prove the direct way. That is, let us assume the existence of an extended vector $\vec{v}$, with $v_\omega = \lightning$ and $\vec{v} \vdash \mathcal{A} \hookrightarrow k$. Then, by the semantics of decision trees (rule SWITCHCONSTR or SWITCHDEFAULT from Fig. 2), we have $\omega \neq 1$ and there exists a decision tree $\mathcal{A}'$ from the case list $\mathcal{L}$ and a value vector $\vec{v}\,'$, such that $\vec{v}\,' \vdash \mathcal{A}' \hookrightarrow k$. By construction of $\mathcal{A}$, the decision tree $\mathcal{A}'$ is $\mathcal{CC}(\vec{o}\,', Q \to B)$, where $Q \to B$ is a decomposition (defined in Section 2.2) of $P \to A$. From $\omega \neq 1$, vector $\vec{v}\,'$ is an extended vector, that is there exists a unique index ω', with $v'_{\omega'} = \lightning$ — more precisely, either $\omega' = a+\omega-1$ when $Q \to B$ is the specialization $\mathcal{S}(c, P \to A)$, or $\omega' = \omega - 1$ when $Q \to B$ is the default matrix. In both cases, again by construction of $\mathcal{A}$, we further have $o'_{\omega'} = o_\omega$. Besides, the components of $\vec{o}\,'$ are pairwise incompatible occurrences, as the components of $\vec{o}$ are. By applying induction to $\mathcal{A}'$, there exists a path in $\mathcal{A}'$ that reaches $\mathsf{Leaf}(k)$ and that does not test $o'_{\omega'} = o_\omega$. We can conclude, since o_1 and o_ω are incompatible and thus *a fortiori* different.

      Conversely, let us assume the existence of a path in $\mathcal{A}$ that reaches $\mathsf{Leaf}(k)$ and that does not test o_ω. Then, we must have $\omega \neq 1$, since $\mathcal{A}$ starts by testing o_1. The path goes on in some of $\mathcal{A}$ child, written $\mathcal{A}' = \mathcal{CC}(\vec{o}\,', Q \to B)$, as we already have defined above — in particular there exists ω', with $o'_{\omega'} = o_\omega$. By induction there exists $\vec{v}\,'$ (whose size n' is the width of Q), with $v'_{\omega'} = \lightning$ and $\vec{v}\,' \vdash \mathcal{A}' \hookrightarrow k$. We then construct $\vec{v}$ with $v_\omega = \lightning$ and $\vec{v} \vdash \mathcal{A} \hookrightarrow k$ and thus conclude. Exact $\vec{v}$ depends on the nature of $Q \to B$. If $Q \to B$ is the specialization $\mathcal{S}(c, P \to A)$, we define $\vec{v}$ as $(c(v'_1, \ldots, v'_a)\ v'_{a+1} \cdots v'_{n'})$. Here we have $\omega = \omega' - a + 1$, noticing that we have $\omega' > a$ (from $o'_{\omega'} = o_\omega$). Otherwise, $Q \to B$ is the default matrix and there exists a constructor c that does not appear in $\mathcal{L}$. Then, we construct $\vec{v} = (c(w_1, \ldots, w_a)\ v'_1 \cdots v'_{n'})$, where $w_1, \ldots, w_a$ are any values of the appropriate types. Here we have $\omega = \omega' + 1$.

   (b) If $\mathcal{A}$ is $\mathsf{Swap}_i(\mathcal{A}')$ where $\mathcal{A}'$ is $\mathsf{Switch}_{o_i}(\mathcal{L}) = \mathcal{CC}(\vec{o}\,', Q \to A)$, the arguments $\vec{o}\,'$ and Q being $\vec{o}$ and P with columns 1 and i swapped. We can conclude by induction, having first observed that assuming either the existence of $\vec{v}$ with $v_\omega = \lightning$ and $\vec{v} \vdash \mathcal{A} \hookrightarrow k$, or the existence of a path that does not test o_ω both imply $i \neq \omega$.

A.2 Lemma 3

We prove item 1 by induction over the structure of P.

1. The case $m = 0$ is impossible, since then P has no row at all, and hence no row that can filter values.
2. If the first row of P consists in wildcards. Then $\mathcal{A}$ must be $\mathsf{Leaf}(a^1)$ and is appropriate.
3. Otherwise, by hypothesis there exists an extended vector $\vec{v}$ ($v_\omega = \lightning$) that matches some row of P in the ML sense. We first show the existence of a column index i, such that $i \neq \omega$ and that one of the patterns in column i is a generalized constructor pattern. There are two cases to consider.
   (a) If $\vec{v}$ matches the first row of P, then, $\vec{p}^{\,1} \preceq \vec{v}$. Since we are in case 3 of compilation, there exists a column index i such that p^1_i is not a wildcard. By the extended definition of $\preceq$ we have $i \neq \omega$.
   (b) If $\vec{v}$ matches some row of P other than the first row, then, by definition of ML matching, we have $\vec{p}^{\,1}\ \#\ \vec{v}$. Thus, by definition of # on vectors, there exists a column index i, such that $p^1_i\ \#\ v_i$. Since $p\ \#\ \lightning$ never holds, we have $v_i \neq \lightning$ (and thus $i \neq \omega$). Furthermore, p^1_i is a generalized constructor pattern, since , for any value w, $(q_1 \mid q_2)\ \#\ w$ implies $q_1\ \#\ v$; and that $_\ \#\ w$ never holds.

   Now that we have found i, compilation can go on by decomposition along column i. Additionally, we know that there exists constructor c with $v_1 = c(w_1, \ldots, w_a)$.

   (a) If i equals 1. Then, there are two subcases, depending on whether c is a head constructor in the first column of P or not (*i.e.* $c \in \Sigma_1$ or not). Let us first assume $c \in \Sigma_1$ Then, by lemma 1-1, we get $\vec{v}\,' = (w_1 \cdots w_a\ v_2 \cdots v_n)$, such that $\mathsf{Match}[\vec{v}\,', \mathcal{S}(c, P \to A)] = k$. Notice that $\vec{v}\,'$ is an extended value vector. Hence, by induction, there exists a decision tree $\mathcal{A}'$ with $\vec{v}\,' \vdash \mathcal{A}' \hookrightarrow k$. The other decompositions of P can be be compiled in any manner. Finally, we build a case list $\mathcal{L}$ and define $\mathcal{A} = \mathsf{Switch}(\mathcal{L})$.

      The case where $c \notin \Sigma_1$ is similar, considering $\vec{v}\,' = (v_2 \cdots v_n)$ and the default matrix.

   (b) If i differs from 1. Then, we swap columns 1 and i in both $\vec{v}$ and P and reason as above.

We omit the proof of item 2, which is by induction over the structure of P, using lemma 1 in the other direction.