When Do Match-Compilation Heuristics Matter?

Kevin Scott and Norman Ramsey
Department of Computer Science
University of Virginia
jks6b@cs.virginia.edu   nr@cs.virginia.edu

May 2000

Abstract

Modern, statically typed, functional languages define functions by pattern matching. Although pattern matching is defined in terms of sequential checking of a value against one pattern after another, real implementations translate patterns into automata that can test a value against many patterns at once. Decision trees are popular automata.

The cost of using a decision tree is related to its size and shape. The only method guaranteed to produce decision trees of minimum cost requires exponential match-compilation time, so a number of heuristics have been proposed in the literature or used in actual compilers. This paper presents an experimental evaluation of such heuristics, using the Standard ML of New Jersey compiler. The principal finding is that for most benchmark programs, all heuristics produce trees with identical sizes. For a few programs, choosing one heuristic over another may change the size of a decision tree, but seldom by more than a few percent. There are, however, machine-generated programs for which the right or wrong heuristic can make enormous differences: factors of 2–20.

1 Introduction

Starting with HOPE (Burstall, MacQueen, and Sannella 1980), functional languages have enabled programmers to define functions by *pattern matching*, i.e., by a sequence of (pattern, expression) pairs. When a function defined by pattern matching is applied to an argument, the implementation finds the first pattern that matches the argument, and the value of the corresponding expression becomes the result of applying the function. Pattern matching is especially well suited to defining functions over algebraic datatypes, and it is widely used in statically typed functional languages, including Standard ML (Milner *et al.* 1997), Objective CAML (Leroy *et al.* 1998), Haskell (Hudak *et al.* 1992; Peyton Jones *et al.* 1999), and Miranda (Turner 1985).

Although the semantics of pattern matching is given by a sequence of comparisons of argument values against patterns, no implementation takes such a

simple, inefficient approach. A sequence of patterns can be compiled into an automaton that identifies the matching pattern efficiently. The *decision tree* is popular because it is easy to ensure that it examines each word of the argument at most once. An internal node designates a word to be tested, and each of its multiple outgoing edges is labeled with a possible value of the word. Internal nodes may also have "default" edges, which are used when the value found does not label any outgoing edge.[1] A leaf node represents a successful match, and it identifies the pattern matched. The *backtracking automaton* is also popular; it is like a decision tree, except that leaf nodes may point back to internal nodes.

The run-time cost of pattern matching is proportional to the number of tests made by the matching automaton, which in a decision tree is the number of nodes on the path from the root to the leaf matched. The static cost of pattern matching, i.e., the code size, is roughly proportional to the total number of states in the automaton. These costs are determined by the order in which parts of the argument(s) are examined.

In implementations of strict languages, pattern-matching automata may examine parts of a value in any order. In implementations of lazy languages, the situation is more complicated; an automaton must not evaluate parts of the argument unnecessarily, lest a function that should terminate fail to do so. Augustsson (1985) argues that automata should examine parts of values from left to right, on the grounds that using a fixed order makes the automaton's behavior predictable. Laville (1991) argues that using a fixed order may result in unnecessary work or unnecessary nontermination, and that an automaton should use any order that evaluates only those parts of a value that are needed to know which pattern matches—but there are lists of patterns for which no such order exists. This paper dodges the question; we consider match compilation only for strict languages, which, because they do not constraint the order of examination, offer the most opportunities for optimization anyway.

Optimizing the number of nodes in a decision tree may be NP-hard, and so several sources have suggested optimization heuristics (Cardelli 1984; Baudinet and MacQueen 1985; Maranget 1992; Ramsey 1999). The primary contribution of this paper is an experimental evaluation of these heuristics.

We consider three measures of the cost of a tree: the number of nodes, the length of the longest path from the root to a leaf, and the average length of all paths from the root to a leaf. We also consider running times of programs compiled using Standard ML of New Jersey. For most benchmarks, heuristics don't matter: these costs don't change. For a few benchmarks, choice of heuristics may change the costs by up to 10–12%. Finally, for a handful of machine-generated instruction recognizers, the choice of heuristics may have an enormous influence, changing costs by factors of 2–20.

[1] One match-compilation algorithm produces trees in which the internal nodes contain both a word designation and a value, with two outgoing edges labeled "equal" and "unequal" (Sestoft 1996). These trees are then post-processed into the decision trees described here.

Constructors	C	
Patterns	$p ::= x \mid C(p_1, \ldots, p_n)$,	$n \geq 0$
Subject trees	$t ::= C(t_1, \ldots, t_n)$,	$n \geq 0$
Paths	$\pi ::= \Lambda \mid k.\pi$	
Substitutions	σ	
Match judgements	$t = \sigma p$	

Table 1: Elements used in match compilation

Patterns	Subject trees
$p_1 = C_1(C_2, C_3)$	$t_1 = C_1(C_2, C_3)$
$p_2 = C_1(x, C_4)$	$t_2 = C_1(C_3, C_4)$
$p_3 = C_1(x, C_5)$	$t_3 = C_1(C_3, C_3)$
$p_4 = x$	$t_4 = C_6(C_2, C_3)$

Figure 1: Example patterns and subject trees

2 The problem of pattern matching

In functional languages, a pattern matcher takes a value and identifies the *first* of a list of patterns that matches the value. We call the value a *subject tree* or *term*. Patterns and subject trees are defined recursively. A subject tree is a *constructor* applied to a (possibly empty) list of subject trees. A *pattern* is either a *variable* or a constructor applied to a (possibly empty) list of patterns. A *linear pattern* is one in which no variable appears more than once. ML requires that patterns be linear, and our match compiler works only with linear patterns. Table 1 shows schemata for patterns and subject trees, and Figure 1 shows examples.

Constructors are drawn from sets determined by the types of values being matched. For example, constructors of an algebraic data type are declared as part of the data type's definition, and constructors of integers are integer literals. Constructors in Figure 1 are labeled C_1, C_2, etc. We assume that the arity of each constructor is fixed, and we write applications of nullary constructors without parentheses.

For ease of presentation, our definitions take a simplified view of subject trees and patterns. Unlike a real functional language, our language does not distinguish integer and real literals or tuple and record constructors from datatype constructors. Subject trees in a real language may contain "atomic" values, like functions, which are not the result of applying any constructor. Patterns in real languages may include "wildcards," which can be represented in our language as fresh variables occuring nowhere else in an expression. We omit these features only for clarity in presentation; our experiments use full Standard ML patterns.

Matching is determined by substitution of values for variables. By definition, a pattern p *matches* a subject tree t if and only if there exists a substitution σ such that $t = \sigma p$. This definition differs from classic tree pattern matching (Hoffmann and O'Donnell 1982) in that the pattern must match at the root of the subject tree. Figure 2 gives an axiom and an inference rule for pattern

$$\text{VAR} \quad t = \{x \mapsto t\}x$$

$$\text{CON} \quad \frac{t_1 = \sigma_1 p_1, \ldots, t_n = \sigma_n p_n \qquad \mathit{dom}(\sigma_i) \text{ disjoint} \qquad \sigma = \sigma_1 \circ \cdots \circ \sigma_n \circ I}{C(t_1, \ldots, t_n) = \sigma C(p_1, \ldots, p_n)}$$

Figure 2: Rules for matching linear patterns

case e
  of $\{C_1(C_2, C_3)$ `=>` $1,$
    $C_1(x, C_4)$ `=>` $2,$
    $C_1(x, C_5)$ `=>` $3,$
    x `=>` 4
  $\}$

Figure 3: Example pattern match

matching; together they specify a recursive algorithm that accepts p and t and either returns σ or fails. The linearity requirement is expressed by the condition "$\mathit{dom}(\sigma_i)$ disjoint."

We write a pattern match as follows:

$$\textbf{case } e \textbf{ of } \{p_i \texttt{ => } e_i\}, \text{ where } 1 \leq i \leq n.$$

The braces should be taken as EBNF repetition, not as set notation; the order of cases is significant. Figure 3 shows an example match.

A match is evaluated as follows. First e is evaluated, and its value t becomes the subject tree. Then the implementation finds the least j such that p_j matches t with substitution σ_j. Finally it evaluates e_j in the current environment, extended by σ_j, and the result is the value of the **case** expression. If no p_i matches t, the value of the **case** expression is undefined. Real functional languages typically work around this problem by appending a p_{n+1} `=>` e_{n+1} such that p_{n+1} always matches and evaluating e_{n+1} raises an exception or causes a run-time error. Figure 4 shows the match-evaluation rule formally; it can be taken as the specification of a naïve pattern matcher that starts with $j = 1$ and attempts to match for increasing j until the matching pattern is identified.

The problem of *match compilation* is this: given a list of patterns $\{p_i\}$, construct an automaton that maps each term t to the least i and σ_i such that $t = \sigma_i p_i$. We call an automaton *efficient* if it examines each constructor of t at most once. Automata constructed by algorithms presented in Baudinet and MacQueen (1985), Laville (1991), Puel and Suárez (1993), Maranget (1992), and

$$\frac{\Gamma \vdash e = t \qquad t = \sigma p_j \qquad \neg\exists i < j. \exists \sigma'. t = \sigma' p_i \qquad \Gamma, \sigma \vdash e_j = t'}{\Gamma \vdash \textbf{case } e \textbf{ of } \{p_i \texttt{ => } e_i\} = t'}$$

Figure 4: Evaluation rule for a pattern match

$$t|_\Lambda = t \qquad\qquad \frac{t_k|_\pi = t}{C(t_1,\ldots,t_n)|_{k.\pi} = t}, 1 \le k \le n$$

Figure 5: Rules for extracting a subtree at a path

Sestoft (1996) are efficient in this sense; automata constructed by algorithms presented in Augustsson (1985), Wadler (1987), and Maranget (1994) are not.

Ideally, we would like to generate matching automata that are not only efficient, but also inexpensive by some other measure. For example, we might prefer an automaton that minimizes the number of states, or the length of the longest path from the initial state to an accepting state, or the average length of all paths from the intial state to accepting states. We have been unable to identify match compilers that produce automata that minimize any of these costs. Indeed, Baudinet and MacQueen (1985) suggests that minimizing the number of nodes in a decision tree is NP-complete, by reduction from one of the trie-index construction problems in Comer and Sethi (1977). We have been unable to duplicate the reduction or to locate a proof of this hypothesis.

3 Pattern-matching automata and decision trees

In this paper, we consider algorithms for building efficient decision trees, and we compare heuristics that determine the sizes and shapes of these trees. A *decision tree* is a pattern-matching automaton in which every state except the initial state has a unique predecessor. The automaton uses paths π to refer to subtrees of the subject tree. Paths are sequences of integers, or more precisely, a path π is either empty (written Λ) or is an integer followed by a path, as shown in Table 1. Figure 5 gives the rules by which a path can be used to designate a subtree of a tree t.

Each non-accepting state in a decision tree is a *TEST* node, which is labeled with a path π and has one or more outgoing edges labeled with constructors. *TEST* nodes may also have a "default" outgoing edge. Each accepting state is a *MATCH* node, which is labeled with a rule and a substitution. Substitutions are represented as lists of (π, v) pairs, where π is a path and v is a variable, and (π, v) stands for the substitution mapping v to $t|_\pi$, where t is the subject tree. The following ML datatype can be used to represent decision trees.

datatype α *tree* = *TEST* **of** *path* $\times$ α *edge list* $\times$ α *tree option*
 | *MATCH* **of** $\alpha \times$ (*path* $\times$ *string*) *list*
withtype α *edge* = *constructor* $\times$ α *tree*

Constructors labeling outgoing edges of the same *TEST* node must be distinct. Figures 6 and 7 show example decision trees for the match of Figure 3. To illustrate the use of the default tree, we have assumed that each type has additional constructors beyond $C_1, \ldots, C_5$.

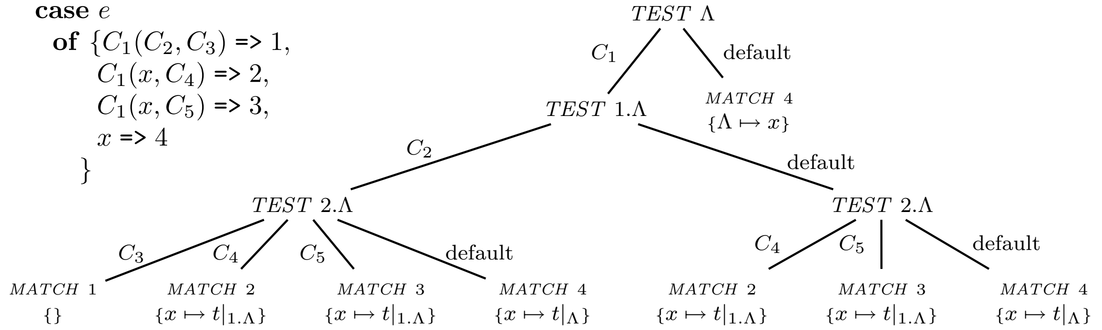

Figure 6: Example match and its decision tree (Left to Right heuristic)

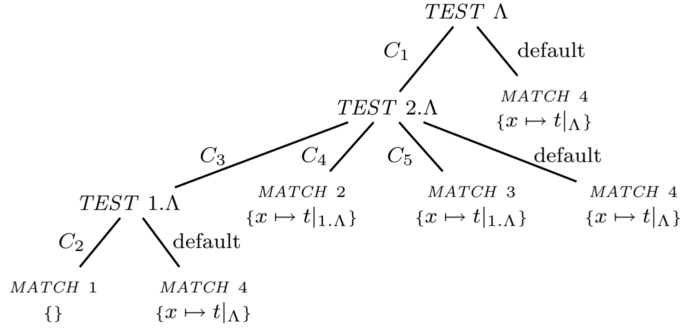

Figure 7: Alternate decision tree (Large Branching Factor heuristic)

Using a decision tree to match is straightforward; the matching automaton walks the tree until it reaches a *MATCH* node, which tells it which rule matches and with what substitution.

$$
\begin{array}{l}
\mathit{match}(\mathit{MATCH}(i, \{(\pi_j, x_j)\}))\ t = (i, \{x_j \mapsto t|_{\pi_j}\}) \\
\mathit{match}(\mathit{TEST}(\pi, \{(C_j, \mathit{node}_j)\}, \mathit{node}_{\mathit{default}}))\ t = \\
\quad \textbf{case}\ t|_\pi\ \textbf{of}\ C(t_1, \ldots, t_k)\ \texttt{=>} \\
\quad\quad \textbf{if}\ \exists j : C = C_j\ \textbf{then} \\
\quad\quad\quad \mathit{match}\ \mathit{node}_j\ t \\
\quad\quad \textbf{else} \\
\quad\quad\quad \mathit{match}\ \mathit{node}_{\mathit{default}}\ t
\end{array}
$$

Because constructor labels are distinct, if there is a j with $C = C_j$, that j is unique.

3.1 Match compilation

Published algorithms for strict match compilation have two flavors. Sestoft (1996) derives an *indirect-style* match compiler by partially evaluating a naïve matcher; the automaton has a sequence of equality tests for each path. Such an automaton can be post-processed into a decision tree. *Direct-style* match compilers

build a decision tree directly. Since many heuristics for match compilation are defined in terms of properties of tree nodes (e.g., outdegree), we discuss only direct-style compilers.

Figure 8 presents our recursive algorithm for match compilation. The algorithm is nondeterministic in the choice of π made in "**let** π be a path..."; different heuristics result in different choices of π. With each decision-tree node, our algorithm keeps a list of *unmatched frontiers*, which identify which rules might match when control reaches that node. This frontier is a pair (i, f), where i identifies the rule, and f is a set of (π, p) pairs in which each π occurs at most once. The *TEST* nodes along the path from the root to the current node record the parts of the original pattern that are known to match; the set f records the remaining, unmatched parts. We write $dom(f)$ for the set $\{\pi \mid \exists p : (\pi, p) \in f\}$, and when $\pi \in dom(f)$, we write $f@\pi$ for the unique p such that $(\pi, p) \in f$.

Match compilation maintains the invariant that rule i can match tree t at a node if and only if rule i is listed at that node in some frontier (i, f) and there is a σ such that $(p, \pi) \in f \Rightarrow t|_\pi = \sigma p$. We establish this invariant by letting the initial frontier for rule i be the pair $(i, \{(\Lambda, p_i)\})$. The invariant holds because by definition, rule i can match if and only if there is a σ with $t = \sigma p_i$.

The match compiler terminates, by inserting a *MATCH* node, when the unmatched frontier of the first rule listed is all variables, in which case the invariant guarantees that the rule matches, and the matching substitution can be obtained directly from the variables and paths in the unmatched frontier for the rule.[2] Termination is guaranteed because each recursive call to *compile* passes a list of frontiers in which the number of constructors is strictly smaller, so eventually the compiler reaches a state in which the first unmatched frontier is all variables.

When the match compiler cannot create a *MATCH* node, it selects a path π to a node to be tested in the subject tree. This path must be associated with a pattern of the form $C(p_1, \ldots, p_n)$ in at least one unmatched frontier. Such a path is guaranteed to exist whenever the compiler failed to create a *MATCH* node. The match compiler may choose *any* such path; in practice, the path is chosen by one or more of the heuristics described in the next section.

Once a path π is chosen, the match compiler creates a *TEST* node with an outgoing edge for each constructor C used in a pattern at π in any unmatched frontier. The outgoing edge leads to a new decision tree created by a recursive call to *compile*. The recursive call uses a list of frontiers with fewer constructors than *frontiers*, as required for termination. This list is computed by the *project* function. For an outgoing edge labeled with C, this function assumes that $t|_\pi = C(t_1, \ldots, t_n)$, and it updates the frontiers accordingly. The function $mapPartial(project(C, \pi))$ eliminates frontiers for rules that cannot match because they call for a different constructor C' in the position π. The functions *map*, *filter*, and *mapPartial*, and the constructors *SOME* and *NONE* are part

[2]The linearity of the patterns guarantees that each variable maps to exactly one subtree of the subject tree, and so the substitution is well defined.

$compile\ frontiers =$
  if $\neg\exists(\pi, p) \in snd(hd(frontiers)) : p$ has the form $C(p_1, \ldots, p_n)$ **then**
    $MATCH(hd(frontiers))$
  else
    let π be a path such that $\exists(i, f) \in frontiers : \pi \in dom(f) \wedge f@\pi$ has the form $C(p_1, \ldots, p_n)$
        $CS = \{C \mid \exists(i, f) \in frontiers : \pi \in dom(f) \wedge f@\pi = C(\ldots)\}$
        $edges = map(\lambda C.(C, compile(mapPartial(project(C, \pi))frontiers)))CS$
        $defaults = filter(\lambda(i, f).\pi \notin dom(f) \vee f@\pi$ is a variable$)frontiers$
    in $TEST(\pi, edges, SOME\ (compile\ defaults))$
    end
where
  $snd(i, f) = f$
  $project(C, \pi)(i, f) =$
    if $\pi \in dom(f)$ **then**
      case $f@\pi$
        of x => $SOME\ (i, f)$
          | $C'(p_1, \ldots, p_n)$ =>
              if $C = C'$ **then** $SOME\ (i, f - \{(\pi, f@\pi)\} \cup \bigcup_{j=1}^{n}\{(\pi.j, p_j)\})$
              else $NONE$
    else
      $SOME\ (i, f)$

The compiler is called initially with *frontiers* equal to a list of $(i, \{(\Lambda, p_i)\})$ pairs.

Figure 8: Match compiler

The algorithm shown is slightly simpler than what we use in practice. If *defaults* is empty, or if a type analysis shows that the set *CS* includes all the constructors that could be used, we omit the default tree, by using *NONE* instead of *SOME* (*compile defaults*). Furthermore, if there is no default tree, and if there is only a single outgoing edge in *edges* (i.e., if the type has but a single constructor), then the *TEST* node is not necessary; *compile* can return the node pointed to by the single outgoing edge.

of the standard basis for SML'97 (Gansner and Reppy 1999); their signatures are given in Appendix A.

3.2 Match-compilation heuristics

In Figure 8, the choice of π—which node to examine next—is nondeterministic. The algorithm used to choose π determines the size and shape of the resulting decision tree, but the only method known for finding the "best" decision tree is exhaustive search, which is impractical. In a real compiler, heuristics are used to choose π. These heuristics are usually expressible in terms of properties of the *TEST* node that would be created for each particular choice of π. Here, we describe various heuristics that have been proposed in the literature or are used in actual match compilers. We state each heuristic as a function from a path π to an integer *score*; paths with higher scores are preferred.

Relevance A path π is deemed *relevant* to rule i if there is an outedge from π's *TEST* node on which rule i does not appear. (Our algorithm permits only choices of π that are relevant to some rule, except when the type system permits only one possible constructor C, in which case no *TEST* node is created.) Formally, π is relevant to rule i iff $(i, f) \in \mathit{frontiers} \wedge \pi \in \mathit{dom}(f) \wedge f@\pi = C(\ldots)$. The relevance heuristic prefers paths that are relevant to early rules. For each path π, the heuristic finds the least i such that π is relevant to rule i, and it assigns a score of $-i$ to π. If π is not relevant to any rule, it is assigned a score of $-N-1$, where N is the number of rules.

Baudinet and MacQueen (1985) recommends this heuristic.

Small Defaults This heuristic assigns to path π a score $-N$, where N is the number of rules such that $(i, f) \in \mathit{frontiers} \wedge (\pi \notin \mathit{dom}(f) \vee f@\pi$ is a variable). These are the rules that are used to build the default tree.

Baudinet and MacQueen (1985) recommends an unspecified combination of this heuristic with the branching-factor heuristic presented below. According to Aitken (1992) and our inspection of the source code, SML/NJ uses this heuristic as its primary heuristic.

Fewer Child Rules This heuristic assigns to path π a score $-N$, where N is computed by adding the number of rules that appear in the *frontiers* sets of each child.

The GAML compiler (Maranget 1992) uses this heuristic as its primary heuristic, and the New Jersey Machine-Code Toolkit (Ramsey and Fernández 1995) uses this heuristic.

Small Branching Factor This heuristic assigns to path π a score $-N$, where N is the number of outgoing edges from the *TEST* node that would be created, plus 1 for the default tree if applicable.

SML/NJ uses this heuristic to break ties when two paths have the same score on the Small Defaults heuristic. The GAML compiler uses this heuristic to break ties when to paths have the same score on the Fewer Child Rules heuristic, except it does not count 1 for the default tree.

Large Branching Factor This heuristic, the opposite of Small Branching Factor, assigns to path π a score N, where N is the number of outgoing edges from the $TEST$ node that would be created, plus 1 for the default tree if applicable.

Cardelli (1984) recommends this heuristic on the grounds that it is likely to produce a shallower tree.

Arity Factor This heuristic assigns to path π a score $-N$, where N is computed by adding the arities of the constructors in set CS, as computed in Figure 8.

Baudinet and MacQueen (1985) recommends this heuristic.

Leaf Edges This heuristic assigns to path π a score N, where N is the number of children of the $TEST$ node that are $MATCH$ nodes.

The New Jersey Machine-Code Toolkit uses this heuristic.

Artificial rule This heuristic assigns to path π a score of -1 if there exists a child of the $TEST$ node whose *frontiers* set contains only the artificial rule added for the case in which no patterns match, and a score of 0 otherwise.

The New Jersey Machine-Code Toolkit uses this heuristic.

Left to right This heuristic assigns scores that are higher for short paths than for longer and higher for paths further to the left. In particular, the score for path π is guaranteed to be higher than the score for path $\pi.j$, and the score for path $\pi.j$ is higher than the score for path $\pi.k$ if and only if $j < k$. The details of the score computation depend on the set of paths that appear in *frontiers*, but the heuristic guarantees to examine paths in top-down, left-to-right order.

This heuristic, as implemented by Sestoft (1996), is used in Moscow ML, Objective CAML, and the ML Kit, and it is used to break ties in Standard ML of New Jersey.

Right to left Top-down, right-to-left order.

4 Related work

Much work on match compilation has been done in the context of lazy languages, in which the semantics taken for pattern matching affects the termination properties of functions. In lazy languages, different matching semantics require different match compilers, only some of which can make use of heuristics. Less attention has been paid to match compilation for strict languages,

in which there is a single semantics for matching. This semantics places fewer constraints on match compilers, leaving room for more experimentation with heuristics.

4.1 Match compilation for lazy languages

Augustsson (1985), which describes a match compiler for the lazy language LML, appears to be the first published explanation of a match compiler for a programming language. Augustsson notes that the order of selection of paths affects the termination properties of the program, and in order to have predictable termination properties, he defines the semantics of pattern matching as checking each pattern in Left to Right order. Augustsson's match compiler differs from ours in its treatment of variable patterns. If $f@\pi$ is a variable, Augustsson's compiler does not replicate f out all outgoing edges of the $TEST$ node for π. Instead, it creates links to the "default" tree from "don't match" leaves in the subtrees reached by C_i edges. The resulting backtracking automaton is a dag, not a tree, and it may test the same paths more than once, so it is not efficient in the sense used in this paper, but it is more compact than a decision tree. Augustsson represents this automaton using "simple **case** expressions," augmented with a `default` construct, which provides a link to the default case of the nearest ancestor **case** expression. Augusstson discusses the translation of this representation into virtual-machine code. Wadler (1987) recapitulates Augustsson (1985) with some refinements and extensions, as part of a textbook treatment of the implementation of lazy functional languages (Peyton Jones 1987).

Laville (1991) proposes an alternate semantics for lazy pattern matching. This semantics is based on *partial terms*, which model incompletely evaluated values in lazy languages. The rules for deciding $t = \sigma p$ permit unevaluated terms in the range of σ. Rather than require partial terms to be evaluated left to right, this semantics permits parts of terms to remain unevaluated as along as they are not needed to prove $t = \sigma p_i \wedge \neg \exists j < i : t = \sigma p_j$. Laville calls a matching automaton *lazy* if it evaluates no part of a term unnecessarily. A lazy automaton is guaranteed to terminate whenever a decision is possible, whereas a non-lazy automaton might fail to terminate if it unnecessarily evaluates a divergent computation. Given a list of patterns, Laville's match compiler either builds a lazy matching automaton or shows no such automaton exists. The essence of the technique is that if two patterns overlap, it may be possible to replace each by a list of patterns such that no two patterns chosen from the lists overlap. This technique can accomodate different priorities for deciding between overlapping patterns, not just textual ordering. Unfortunately, it does not work for types with infinite constructor sets, like integers.

Puel and Suárez (1993) takes a similar approach, translating overlapping patterns into equivalent "constrained patterns," which partition the universe of partial terms into sets that do or do not match a given pattern, or that do or do not terminate. The match compiler adapts the techniques of Huet and Lévy (1979) to work on these constrained patterns.

Augustsson (1985)	Maranget (1992)	This paper
Matrix of p_{ij}	Clause matrix	Unmatched frontiers
Matrix column i	Column i	Path π_i in $dom(f)$
Matrix row j	Row j	f where $(j, f) \in \mathit{frontiers}$
e_i	V_i	$t\|_{\pi_i}$
p_{ji}	p_i^j	$f@\pi_i$, where $(j, f) \in \mathit{frontiers}$
	Column i all variables	$\neg\exists(j, f) \in \mathit{frontiers} : \pi_i \in dom(f) \wedge f@\pi$ has the form $C(p_1, \ldots, p_n)$
	Σ	CS
	Row deletion	*project* returns *NONE* and *mapPartial* removes the frontier

Table 2: Two formulations of match compilation, compared

Maranget (1992) presents an algorithm for building efficient decision trees for pattern matches using the Laville semantics. The algorithm produces lazy decision trees when possible. The primary contribution is that the algorithm constructs a decision tree directly, without requiring intermediate data structures that may be exponentially large. This paper also recognizes that there may be more than one efficient, lazy decision tree, and it recommends the Fewer Child Rules and Small Branching Factor heuristics.

Maranget (1994) adapts the methods of Maranget (1992) to create a "decision dag" in the style of Augustsson (1985). This dag may be lazy without being efficient, but its size is bounded by the size of the list of patterns. Methods that produce pure decision trees, including the algorithm used in this paper, are subject to potential exponential blowups in code size. Maranget (1994) reports experience with such blowups when implementing the technique of Puel and Suárez (1993), and it presents results from one benchmark, in which decision dags are 5-10% smaller than related decision trees.

Maranget's match compiler is nearly an instance of the compiler specified in Figure 8. The primary difference is that in Maranget's compiler, every unmatched frontier has the same domain. This invariant could be maintained in our compiler by changing the definition of *project*, replacing *SOME* (i, f) by *SOME* $(i, f - \{(\pi, f@\pi)\} \cup \bigcup_{j=1}^{n} \{(\pi.j, x_j)\})$ where n is the arity of C and each x_j is a fresh variable (wildcard).

For interested readers, Table 2 shows the parallels between Augustsson (1985), Maranget (1992), and our compiler.

4.2 Match compilation for strict languages

Cardelli (1984) discusses match compilation briefly, using a "rows and columns" approach like that of Maranget (1992), but less formal. This match compiler

prefers π's that yield *TEST* nodes with many outgoing edges, i.e., it uses the Large Branching Factor heuristic.

Baudinet and MacQueen (1985) discusses match compilation and notes that a strict evaluation model means that paths can be tested in any order. The paper states without proof that compiling to a decision tree with a minimum number of nodes is NP-complete, and it presents the Relevance, Low Branching Factor, and Arity Factor as heuristics that produce good trees in practice.

Sestoft (1996) derives a match compiler by partially evaluating a naïve pattern matcher. The compiler produces an efficient "if-then-else tree," which is post-processed into an efficient decision tree. This simple, elegant method is used in Moscow ML, Objective CAML, and the ML Kit. It is unclear whether it could be adapted to use heuristics other than Left to Right or Right to Left.

5 Experiments

We have implemented the heuristics of Section 3.2 in the Standard ML of New Jersey compiler, version 110.0.3. We have done so by replacing the functions `metric` and `metricBetter` in the file `matchcomp.sml` in directory `translate`. The modifications and new code total less thatn 100 lines. The modified compiler uses a list of heuristics stored in a ref cell, so we can test different heuristics without rebuilding the compiler. We compare the resulting lists of scores using lexicographic ordering; as in standard SML/NJ, ties are broken using Left to Right.

We have implemented 9 of the 10 heuristics in Section 3.2; the implementation of Leaf Edges presented some late-breaking difficulties. With 9 heuristics, there are over 9! possible lists; even if we could evaluate one every 10 seconds, it would take more than six weeks to check them all. We have therefore measured the effects of single heuristics and of lists of two heuristics, as well as selected longer lists of especial interest.

We hope to be able to report results with Objective CAML in the full paper.

We have measured both static and dynamic properties of the automata produced by different heuristics. The static properties include

- the number of nodes in the decision tree
- the maximum path length (height) of the decision tree
- the average path length of the decision tree

Because the values of these properties depend on the patterns that are compiled, we report not the values themselves but the ratios of the values to a baseline. We have chosen the Left to Right heuristic as the baseline; this choice is appropriate both because it supports the elegant match-compilation algorithm of Sestoft (1996) and because it is used in several compilers for ML and CAML.

The only dynamic property we have measured is running time. *To assist in understanding the effects of different heuristics, we hope in the full paper to be able to report measurements of the number of TEST nodes traversed.*

Benchmark	Lines	Source	Description
barnes-hut	1226	NJB	Barnes-Hut N-body code
boyer	910	NJB	tautology checker
fft	194	NJB	fast Fourier transform
knuth-bendix	584	NJB	Knuth-Bendix completion
lexgen	1305	NJB	lexical analyzer generator
life	141	NJB	game of life
logic	345	NJB	simple Prolog-like interpreter
mandelbrot	60	NJB	computes Mandelbrot set
mlyacc	7272	NJB	parser generator
ray	447	NJB	ray tracer
simple	913	NJB	spherical fluid-dynamics simulation
tsp	536	NJB	travelling salesperson
vliw	3680	NJB	VLIW translator
smlnj	125121	SMLNJ	SML/NJ compiler, version 110.0.3
alpha	385	MD	instruction decoder for Alpha
sparc	603	MD	instruction decoder for SPARC
mips	178	MD	instruction decoder for MIPS
pentium	6520	MD	instruction decoder for Pentium

NJB = SML/NJ benchmark suite
SMLNJ = SML/NJ compiler sources
MD = derived from descriptions for NJ Machine-Code Toolkit

Table 3: Benchmarks

5.1 Benchmarks

We have measured behavior of programs from the SML/NJ benchmark suite (located at `ftp://ftp.research.bell-labs.com/dist/smlnj/benchmarks`), and of SML/NJ compiling itself. We have also devised some especially challenging benchmarks based on the problem of recognizing binary representations of machine instructions. Table 3 identifies all the benchmarks.

The binary recognizers are derived from formal specifications of binary representations of instructions (Ramsey and Fernández 1997). From each machine specification, we have derived ML functions that recognize each combination of instruction and addressing mode. The argument to the recognizer is a large tuple holding values of fields of the instruction being recognized. Table 4 shows the number of fields in the tuple, the number of patterns in the recognizer's definition, and the average number of constructors appearing in each pattern. (In this context, constructors are integer field values.) The ML recognizers are

Machine	Fields	Patterns	Constructors/Pattern
MIPS	16	172	4.1
Alpha	13	371	4.4
SPARC	12	581	4.6
Pentium	25	6475	9.0

Table 4: Properties of recognizers for machine instructions

not completely equivalent to the original recognizers; side conditions have been discarded during the translation.

The numbers in Table 4 show why these functions are especially challenging for match compilation. The number of π's at early choice points is quite large, so there is potential for exponential blowup. For any given π, for the number of f's with $\pi \in dom(f)$ is small, so the number of rules eliminated by $project(C, \pi)$ is small. The Pentium recognizer is especially challenging; there are only two heuristics (Small Defaults and Few Child Rules) for which the compiler terminates in reasonable time with reasonable memory consumption. Without results for Left to Right, we have no basis for comparison, so we do not report results for the Pentium benchmark.

5.2 Results

For all normal benchmarks except lexgen, mlyacc, ray, vliw, and smlnj, the heuristics made no difference at all. On the static measurements for ray, it was possible to do up to 1% worse then Left to Right, but not better. For lexgen, it was possible to do up to 2–5% worse then Left to Right, but not better. For mlyacc, it was possible to do up to 40% worse, but not better.

For vliw, improvements of up to 19% in tree size were possible, along with improvements of up to 5% in path length; it was also possible to do worse.

For smlnj, Fewer Child Rules coupled with Small Defaults was 5–10% better than Left to Right in number of nodes and up to 1% better in average path length. Many other heuristic combinations were 2–5% better in number of nodes, including the small defaults, small branching factor combination used in production SML/NJ. Compilers using Small Branching Factor failed to compile the file `parse/ml.grm.sml`, even when ties were broken using other heuristics. For vliw, things are more interesting. Fewer Child Rules is 10% or more better than Left to Right on nodes, but 2–5% worse on paths. The best overall in terms of paths and nodes is Small Defaults, with ties broken by Right to Left; this combination does 10% better on nodes and 2–5% better on paths. Small Defaults does nearly as well when ties are broken by Small Branching Factor or Fewer Child Rules.

Figure 9 shows the tree sizes and average path lengths of the interesting benchmarks under selected heuristics. The tree-height (maximum path length) costs were uninteresting and have been omitted. Costs are normalized with respect to the cost for Left to Right; longer bars are better. We have used

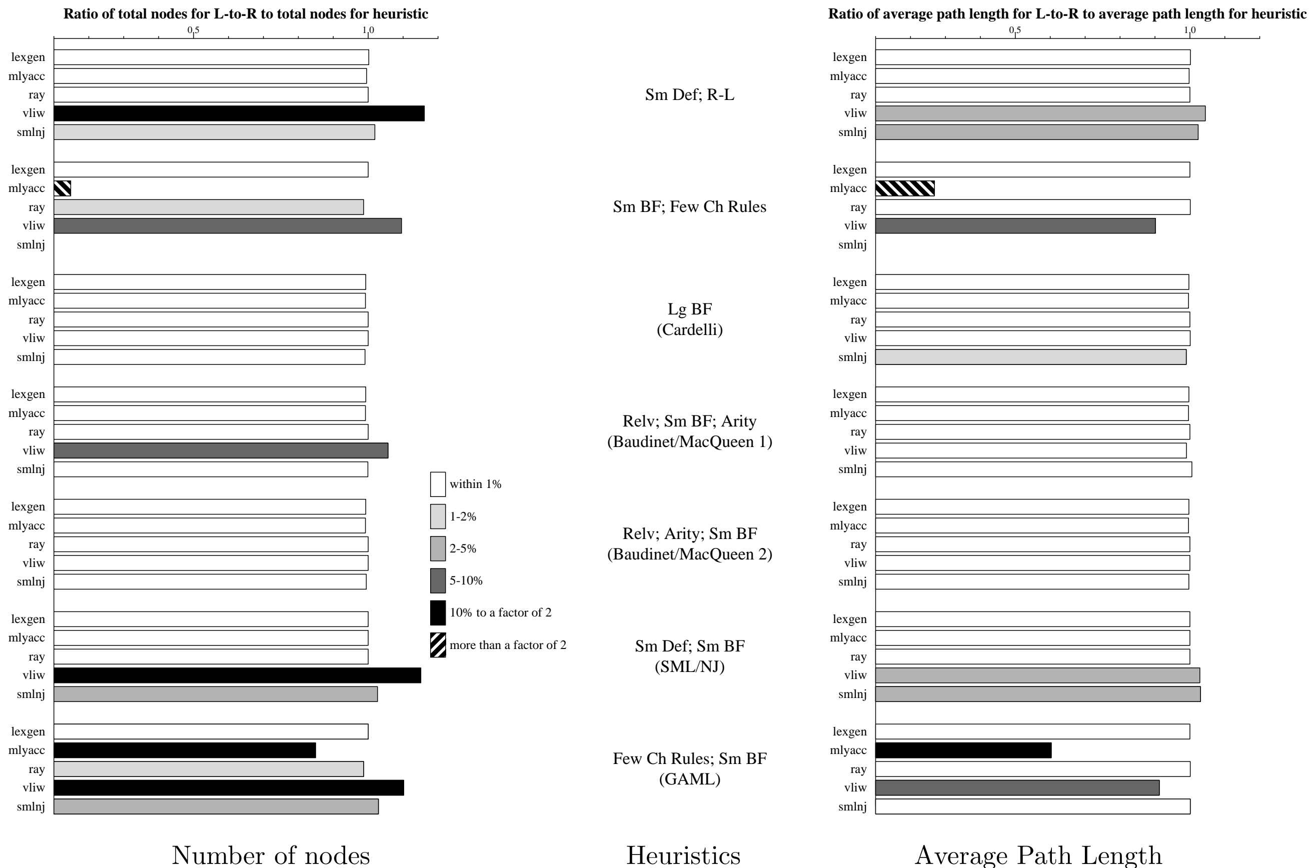

Figure 9: Static costs of selected benchmarks and heuristics

shading to highlight how the costs differ from Left to Right; darker shading indicates larger differences. The preponderance of white bars show results that differ from Left to Right by at most 1%.

The best results for vliw are obtained by Small Defaults, with ties broken by Right to Left. We have included Small Branching Factor, with ties broken by Fewer Child Rules, as an example of a heuristic that shows small improvements on some benchmarks, but can do disastrously badly, e.g., on mlyacc and smlnj. (We could not even compile smlnj using Small Branching Factor, because the compiler consumed too much time and memory.) The remaining combinations of heuristics are those recommended in the literature or used in the SML/NJ or GAML compilers. The standard SML/NJ heuristics do very well on vliw and are the best for other benchmarks.

As expected, Cardelli's Large Branching Factor heuristic produces larger trees than those recommended by Baudinet and MacQueen, which include Small

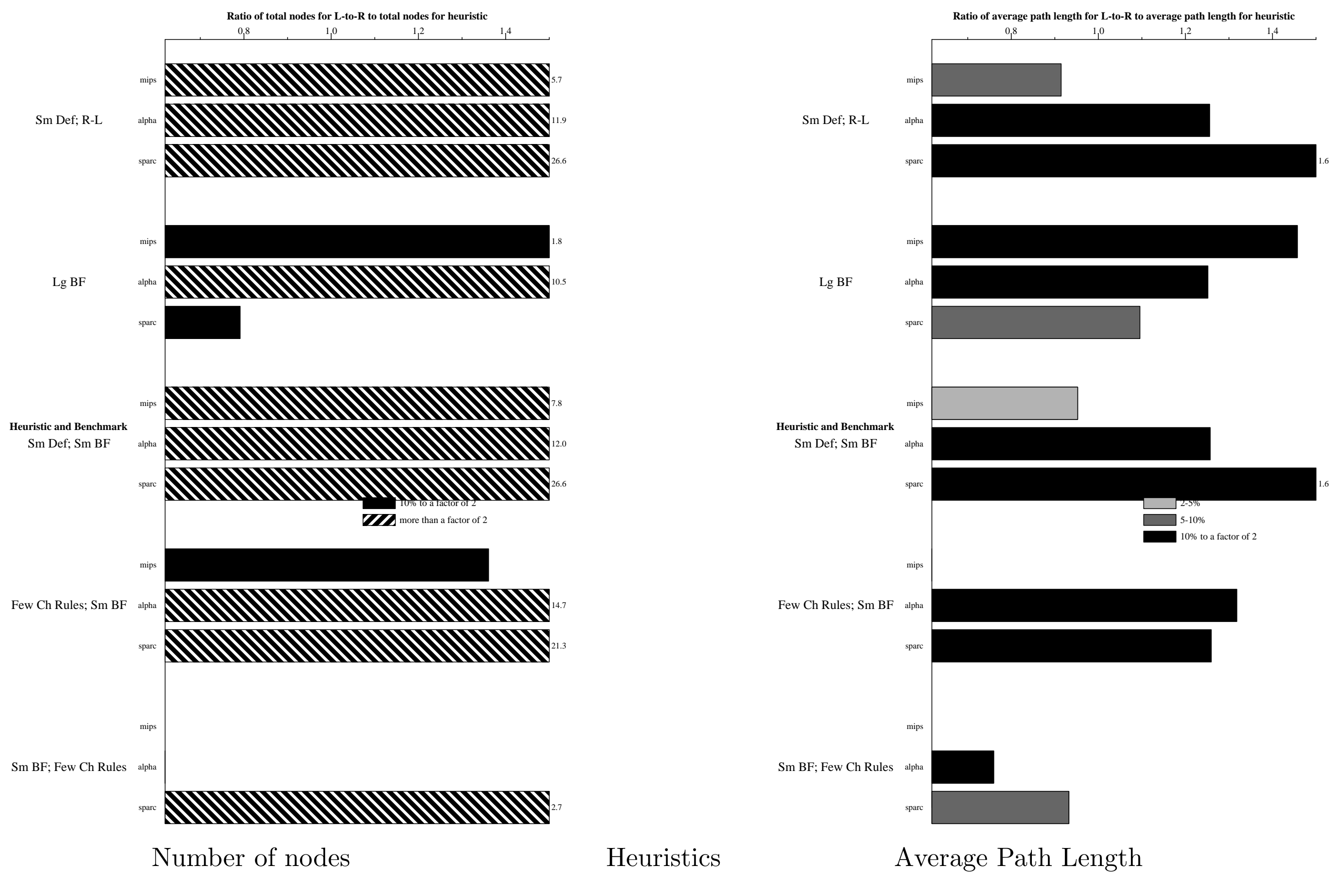

Figure 10: Static costs of machine recognizers (selected heuristics)

Benchmark	Number of nodes	Average path length
ray	$r = 0.02$	$r = 0.02$
vliw	$r = 0.23$	$r = 0.06$
lexgen	$r = 0.30$	$r = 0.29$
mlyacc	$r = 0.04$	$r = 0.05$

Table 5: Correlations of running times with static cost measures

Branching Factor. Perhaps surprisingly, however, Large Branching Factor does not produce trees with shorter average path lengths.

We are aware that putting the entire SML/NJ sources in a single benchmark is likely to hide interesting behavior, and we plan to correct this defect in the full paper.

Figure 10 shows the results for the instruction-recognizer benchmarks. Ratios greater than 1.5 are pinned at 1.5; otherwise most bars would be too short to see. Values for these ratios are given to the right of the bars. Space limitations prevent further discussion of these benchmarks.

We measured running times on a 200 MHz Pentium with 128MB of RAM running Linux kernel 2.0.30. The effect of heuristics on running time is not obvious on casual inspection, but there is a statistical correlation between running times and some static measures of cost. We have computed the linear correlation (Bevington and Robinson 1992, §11.2) between running times, number of nodes, and average path length. Table 5 shows these correlations. The results show that the static measures may help predict the running times of the vliw and lexgen benchmarks. For the final paper, we plan a more thorough analysis that will show *how much* of the variation in running time is predicted by the tree size or the average path length.

Our experiments show that most programs compiled with a simple left-to-right heuristic are not changed by the use of other heuristics. Still, there are a few programs for which heuristics can make a big difference in static measures that correlate with code size. We don't observe big differences in running times, but a statistical analysis (to be completed for the final paper) should reveal how much of the variation in running times is predicted by static measures like number of nodes or average path length.

Acknowledgements

We thank Dave MacQueen for providing internal documentation of the match compiler used in Standard ML of New Jersey. This work has been supported in part by NSF grant ASC-9612756, NSF CAREER award CCR-9733974, and DARPA contract MDA904-97-C-0247.

References

Aitken, William. 1992 (Summer).
The SML/NJ match compiler.
Internal documentation obtained from Dave MacQueen
`<dbm@research.bell-labs.com>`.

Augustsson, Lennart. 1985 (September).
Compiling pattern matching.
In Jouannaud, Jean-Pierre, editor, *Functional Programming Languages and Computer Architecture*, LNCS.

Baudinet, Marianne and David MacQueen. 1985 (December).
Tree pattern matching for ML (extended abstract).
Unpublished manuscript, AT&T Bell Laboratories.

Bevington, Philip R. and D. Keith Robinson. 1992.
Data Reduction and Error Analysis for the Physical Sciences. second edition.
New York: McGraw-Hill.

Burstall, Rod M., David B. MacQueen, and Donald T. Sannella. 1980 (August).
Hope: An experimental applicative language.
In *Conference Record of the 1980 LISP Conference*, pages 136–143. ACM, ACM.

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

Comer, Douglas and Ravi Sethi. 1977 (July).
The complexity of trie index construction.
Journal of the ACM, 23(3):428–440.

Gansner, Emden and John Reppy, editors. 1999.
The Standard ML basis library.
Book in preparation. Preliminary version available at
`http://cm.bell-labs.com/cm/cs/what/smlnj/doc/basis/index.html`.

Hoffmann, Christoph M. and Michael J. O'Donnell. 1982.
Pattern matching in trees.
Journal of the ACM, 29(1).

Hudak, Paul, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. 1992 (May).
Report on the programming language Haskell, a non-strict, purely functional language, version 1.2.
SIGPLAN Notices, 27(5):R1–R164.

Huet, Gérard and Jean-Jacques Lévy. 1979 (August).
Call by Need Computations in Non-Ambiguous Linear Term Rewriting Systems.
Technical Report 359, IRIA.

Laville, Alain. 1991 (April).
Comparison of priority rules in pattern matching and term rewriting.
Journal of Symbolic Computation, 11(4):321–348 (or 321–347??).

Leroy, Xavier, Didier Rémy, Jérôme Vouillon, and Damien Doligez. 1998.
The Objective Caml system, documentation and user's guide.
INRIA.
Available at `http://pauillac.inria.fr/ocaml/htmlman/`.

Maranget, Luc. 1992 (June).
Compiling lazy pattern matching.
In White, Jon L., editor, *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 21–31, San Francisco, CA.

———. 1994 (October).
Two techniques for compiling lazy pattern matching.
Technical Report RR-2385, INRIA.

Milner, Robin, Mads Tofte, Robert Harper, and David MacQueen. 1997.
The Definition of Standard ML (Revised).
Cambridge, Massachusetts: MIT Press.

Peyton Jones, Simon L., John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. 1999 (February).
Haskell 98: A non-strict, purely functional language.
Available from `www.haskell.org`.

Peyton Jones, Simon L. 1987.
The Implementation of Functional Programming Languages. International Series in Computer Science.
Englewood Cliffs, NJ: Prentice Hall.

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

Ramsey, Norman and Mary F. Fernández. 1995 (January).
The New Jersey Machine-Code Toolkit.
In *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302, New Orleans, LA.

———. 1997 (May).
Specifying representations of machine instructions.
ACM Transactions on Programming Languages and Systems, 19(3):492–524.

Ramsey, Norman. 1999.
Match compiler for New Jersey Machine-Code Toolkit (ML version).
This source code can be downloaded from
`http://www.cs.virginia.edu/nr/toolkit`, or it can be browsed at
`http://www.cs.virginia.edu/nr/toolkit/working/sml/WWW/match.html`.

Sestoft, Peter. 1996 (February).
ML pattern match compilation and partial evaluation.
In Danvy, O., R. Glück, and P. Thiemann, editors, *Dagstuhl Seminar on Partial Evaluation*, Vol. 1110 of *Lecture Notes in Computer Science*, pages 446–464, Berlin.

Turner, David A. 1985 (September).
Miranda: A non-strict functional language with polymorphic types.
In Jouannaud, Jean-Pierre, editor, *Proceedings IFIP Conference on Functional Programming Languages and Computer Architecture*, Vol. 201 of *Lecture Notes in Computer Science*, pages 1–16, New York, N.Y.

Wadler, Philip. 1987.
Efficient compilation of pattern-matching.
In *The Implementation of Functional Programming Languages* (Peyton Jones 1987), Chapter 5, pages 78–103.

A ML basis

These are the functions used in the paper.

```
datatype 'a option = SOME of 'a | NONE
val map        : ('a -> 'b) -> 'a list -> 'b list
val mapPartial : ('a -> 'b option) -> 'a list -> 'b list
val filter     : ('a -> bool) -> 'a list -> 'a list
```

$map\,f\,l$ applies f to every element of l and returns a list of the results. $mapPartial\,f\,l$ does the same, except it includes only results of the form $SOME\ x$, and it strips off the $SOME$. $filter\,p\,l$ applies predicate p to the elements of l and returns a list of those elements satisfying p.