Tree Pattern Matching for ML
(extended abstract)

Marianne Baudinet

Stanford University
Computer Science Department
Stanford, CA 94305

David MacQueen

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

December 6, 1985

Tree Pattern Matching for ML
(extended abstract)

Marianne Baudinet

Stanford University
Computer Science Department
Stanford, CA 94305

David MacQueen

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

1. Introduction

In the programming language ML, a function can be defined by a sequence of pattern-expression pairs, called *rules*. When such a function is invoked, its argument is matched against the patterns and the first rule whose pattern matches is selected and its expression is evaluated. If none of the patterns match the argument, a run-time exception is generated. A pattern is either simple (a variable or constant), a tuple of patterns, or a constructor operation applied to a tuple of patterns. Thus patterns are tree-shaped terms. An argument value can also be viewed as a tree-shaped term formed with constructors, tuples, and atomic values (*i.e.* values of primitive, abstract, or functional types). Function application therefore involves determining which of a sequence of pattern trees matches the given argument tree. This paper addresses the problem of compiling such sequences of patterns into efficient pattern-matching code. The goal is to minimize the number of tests or discriminations that have to be applied to any given argument to determine the first pattern it matches.

Our approach is to transform a sequence of patterns into a decision tree, *i.e.* a tree which encodes the patterns and defines the order in which subterms of any given value term have to be tested at run-time to determine which pattern matches the value. Each internal node of a decision tree corresponds to a matching test and each branch is labeled with one of the possible results of the matching test and with a list of the patterns which remain potential candidates in that case. It is then straightforward to translate the decision tree into code for pattern matching. During the construction of a decision tree it is also easy to determine whether the pattern set is "exhaustive", meaning every possible argument value matches at least one pattern, and whether there are any "redundant" patterns (*i.e.* patterns that are matched only by values that are already matched by an earlier pattern). Nonexhaustive definitions and redundant patterns are anomalies that can be usefully reported by the compiler.

Our goal in constructing the decision tree is simply to minimize the total number of test-nodes. This minimizes the size of the generated code and also generally minimizes the number of tests performed on value terms. However, we have discovered that finding the decision tree with the minimum number of nodes is an NP-complete problem. This result is established by reduction from one of the trie index construction problems (pruned O-trie space minimization), which was proved to be NP-complete in [Co76, CS77]. Therefore, we developed a set of efficient heuristics that in practice produce an optimal decision tree in almost all cases.

The problem of string matching has been thoroughly studied and efficient algorithms have been developed for it ([AC75, BM77]). In their work on term-rewriting languages [HO79, HO82a, HO82b, OD85], Hoffmann and O'Donnell have generalized some of these algorithms to tree pattern matching. Their goal was to find the position of all the subterms of a given subject term that are matched by one of a given set of patterns, using techniques that can assimilate local changes in the value term without having to rescan the entire tree. In [HO82a, OD85] a bottom-up method was proposed. It is based on the idea that one searches for the position of all the subterms matched by one of the given patterns. Two other methods were also proposed: a top-down algorithm [HO82a, OD85] and a method called flattened pattern matching [OD85], which both consisted in reducing the tree matching problem to string matching and then using the Aho-Corasick [AC75] string matching algorithm. These techniques turn out to be inappropriate for the ML pattern matching problem, which involves finding the first pattern in the sequence which matches the *whole* given value term. Reducing pattern matching in ML to string matching is not a good idea in general, since the overhead of transforming the tree structured argument would be excessive. It is much better to work directly with the existing tree structure.

Augustsson [Au85] presents a technique for compiling ML pattern matching used in the Lazy ML system developed at Gothenburg. Their approach merges the patterns into a nested case construct, with a nested case for each level of structure in the patterns. Because their compiler uses lazy evaluation they are restricted to consider pattern components in a strictly left-to-right order to preserve termination properties of programs. This involves some backtracking and redundant effort which is avoided by our technique, because we assume call-by-value and have the freedom to examine components of the argument in any order. [Ca84] alludes to an earlier version of our algorithm for pattern matching in ML.

We start by describing the context in which tree pattern matching occurs in ML and defining possible properties of sequences of patterns. We then describe the abstract syntax of patterns and value terms and specify the matching problem. Next, we describe the sort of decision trees we use and define what we call the *dispatching* problem. Since it is NP-complete, we introduce several heuristics that provide a practical solution for the dispatching problem.

2. Pattern matching in ML

ML is a functional language in the weak sense that functions are ''first-class'' values and the basic unit of organization of programs is the function; it is not ''purely applicative'' because it includes a reference construct admitting assignment as well as exceptions with dynamic handlers. In ML, as in other functional languages, a typical function analyzes its argument by recognizing its structure and breaking it down into components. This can be done in the conventional way using conditional expressions in conjunction with recognizer predicates (*e.g. null*) to discriminate between cases and then applying selector functions (*e.g. hd* and *tl*) to extract components. In ML one can also define functions in *clausal form*, *i.e.* as a sequence of *rules*, each of which has its associated argument pattern and body expression, somewhat in the style of Prolog (but with an entirely different semantics). The sequence of rules itself is called a *match*. In this form, pattern matching is used to analyze the argument, simultaneously discriminating between cases and binding components to variables appearing in the pattern. These two forms of function definition are illustrated by the following examples.[1]

Example 1

```
[1]  fun diff (x, y) = if null x then y
                       else if null y then x
                                 else diff (tl x, tl y);

[2]  fun diff (nil, y) = y |     (1)
         diff (x, nil) = x |      (2)
         diff (cons(hdx,tlx), cons(hdy,tly)) = diff (tlx, tly);    (3)
```

[1] and [2] are equivalent ML definitions of a function *diff* with an argument which is a pair of lists. [1] uses the if-then-else construction while [2] is in clausal form. The match contains three patterns which cover all the possible arguments, *i.e.* they are exhaustive. (1) and (2) match all the cases where at least one component of the pair is *nil* while (3) matches the remaining values where both components are non-null and therefore are the result of consing an object onto a list. ML allows non-exhaustive matches at the risk of possible run-time exceptions. Non-exhaustive matches appear in the definition of functions which are designed to be applied to restricted kinds of data. For example, if one expects that a function is to be applied only to non-null lists, then one might omit the *nil* case in its definition.

Patterns (1) and (2) are said to be *non-disjoint* or to *overlap* because there is at least one possible value which they both match, *i.e. (nil, nil)*. Matches are ordered sequences of rules, and the ordering of the rules imposes a priority ordering among their respective patterns. Thus, in definition [2], pattern (2) can be considered the matching pattern only for arguments which are not

[1] *hd* and *tl* are the usual list selector functions, analogous to the Lisp *car* and *cdr* functions. *cons* is the list constructor, analogous to the Lisp *cons*. *null* is the usual predicate recognizing the empty list.

matched by pattern (1).

When pattern (3) matches an argument, *hdx* and *hdy* both get bound to a value during the matching process, but those values are never used. ML allows us to avoid introducing such useless variables by using *wildcard*, denoted ‘‘_ ’’. By definition, wildcard matches any value. So, rule (3) in definition [2] of *diff* could be written as:

diff (cons(_,tlx), cons(_,tly)) = diff (tlx, tly); (3′)

As mentioned above, even though patterns can overlap they should be *irredundant i.e.*, each pattern in a given sequence must match some value which is not matched by some pattern earlier in the sequence. For example, if a variable or wildcard is a pattern in a match, it has to be the last one of the match, otherwise all the following patterns would be redundant. Finally, patterns must be *linear*, meaning that no variable can occur more than once in a pattern. Nonlinear patterns are a special case of ‘‘conditional patterns’’ for which there is a side condition that must be satisfied before the match is successful. ML does not support such conditional patterns.

ML is a strongly typed language, and pattern matching is intimately connected with the type system. In particular, pattern matching is supported by certain types known as *concrete* or *data* types. These types generalize Pascal’s variant record and enumeration types as well as Lisp’s data-type of lists. A concrete type is a disjoint union of types, possibly recursive. A typical example of an ML concrete type declaration is

datatype tree = nulltree | *leaf of int* | *node of* (*tree * string * tree*)

This defines a type *tree* of binary trees with leaves having integer labels and interior nodes with string labels. It is a disjoint union of the three types *unit*[2], *int*, and *tree * string * tree*, and *nulltree, leaf,* and *node* are called (*data*)*constructors* because they construct tree values from elements of the component types. Their types are implicitly declared as part of the declaration of *tree* as follows:

$$nulltree : tree$$

$$leaf : int \rightarrow tree$$

$$node : tree * string * tree \rightarrow tree$$

The special property of data constructors is that the values they produce do not reduce to any simpler form. For example, a tree node remains forever a tree node and its three arguments or components are uniquely determined by the node value. Furthermore, any value of type tree must have been constructed using exactly one of the constructors *nulltree*, *leaf*, or *node*. This makes it feasible to analyze values of type tree by matching against pattern terms that are also built using these three constructors. For instance, in the scope of the above declaration we could write the following clausal function definition:

[2] *unit* is the trivial type containing one component. A nullary constructor such as *nulltree* might be given the type *unit* $\rightarrow$ *tree*, but the convention is to treat it as a constant of type *tree*.

fun depth nulltree = 0 |

depth(*leafn*) = 1 |

depth(*node*(t1, s, t2)) = 1+ *max*(*depth* t1, *depth* t2)

Concrete types also include n-ary type constructors. These are introduced by parameterized definitions such as the following definition of *list*:[3]

datatype α *list* = *nil* | *cons of* (α * α *list*)

The unary type constructor *list* thus defined can be used to construct list types like *int list* or (*bool* * (*int* $\rightarrow$ *stinglist*)) *list*. The structure of values of such compound types is determined by the top-level type constructor, so any value of type τ *list* is constructed by *nil* or *cons*, no matter what type τ may be. ML allows *polymorphic* types, which are, roughly speaking, type schemas with free type variables. The constructors associated with type constructors are polymorphic. For example, *nil* : α *list* and *cons* : α * α *list* $\rightarrow$ α *list*. A type which does not contain any type variable is said to be *monomorphic* (e.g. *int list*). Replacing a type variable in a polymorphic type by any ML type produces an *instance* of that type. The ordering "is an instance of", denoted $\leq$ was introduced in [Mi78]. For example, *int list* $\leq$ α *list*. A polymorphic value can be used with any instance of its polymorphic type: *e.g. cons*(3, *nil*): *int list*.

Other types in ML include primitive types like *int*, functional types like *int* $\rightarrow$ *int*, and abstract types. So far as pattern matching is concerned these types are atomic or unstructured; their values cannot be analyzed by pattern matching and will only match variables or wildcard (or integer constants in the special case of *int*). We will call such values *atomic*.

3. Syntax of patterns and terms, matching criterion and matching problem

Syntax of patterns and terms

We consider a somewhat simplified version of the ML syntax introduced in [Mi85].

p ::=	_	(*wildcard*)
	| v	(*variable*)
	| c	(*constant*)
	| $(p_1, \ldots, p_n)$	(*tuple of patterns*)
	| $k\ p'$	(*constructor* k *applied to pattern* p')

All the variables appearing in a pattern have to be distinct and they are considered to be bound by their appearance in the pattern. The scope of such bindings is the body expression associated with the pattern in its rule. Patterns also have to be well-typed, which puts further restrictions on their formation. For instance, *cons*(3) is not well-typed, while *cons*(x) is well typed and implies that the type of the bound variable x is τ * τ *list* for some appropriate type τ (note that the argument of

Early Greek letters like α are used as type parameters and type constructors like *list* are always written as postfix operators.

cons does not have to be an explicit pair).

The patterns match values which are tree-shaped data structures formed by the application of constructors and tupling to atomic values. Thus values can be thought of as having an abstract term structure, and we will call them *value terms* to emphasise this view. Their abstract syntax is analogous to that of patterns:

t ::=	*atom*	(*atomic value*)
	| c	(*constant*)
	| $(t_1, \ldots, t_n)$	(*tuple of terms*)
	| $k\ t'$	(*constructor k applied to term t'*)

Matching criterion for a single pattern

The definition of when a pattern matches a value term is straightforward and can be expressed formally in terms of structural induction on the pattern. It is useful to define a more refined notion of *agreement between a pattern and a value term on a subterm*. Informally, a pattern agrees with a value term on a subterm (of the value term) iff they have the same structure (constructor or tuple-arity) at each node along the path from the root to the subterm (inclusive). Example 2 illustrates this concept.

Example 2. Consider the value term $t = (true,\ cons\ (1,\ nil))$ and the following sequence of patterns of type $(bool\ *\ (\alpha\ list))$:

(1) $p_1 = (true,\ \_)$

(2) $p_2 = (false,\ nil)$

(3) $p_3 = (false,\ cons\ (x,\ nil))$

(4) $p_4 = (false,\ cons\ (y,\ z))$

p_1 agrees with t on its first component because they both have *true* as first component, whereas p_2, p_3 and p_4 do not. p_1 also agrees with t on its second component because wildcard matches and therefore agrees with any value term. p_3 and p_4 both have a second component of the form *cons* p' and therefore agree with t on this subterm. They also agree with t on both arguments of *cons* (variables match and therefore agree with any value term).

Formally, given a pattern p and a value term t, *p agrees with t on a subterm t^* of t* if and only if one of the following conditions is true:

- $p = \_$ (wildcard).
- p is a variable.
- $p = c$, where c is a constant, and $t = c$.
- $p = (p_1, \ldots, p_n)$, $t = (t_1, \ldots, t_n)$, and $t^* = t$.
- $p = (p_1, \ldots, p_n)$, $t = (t_1, \ldots, t_n)$, t^* is a subterm of t_j, for some j such that $1 \le j \le n$, and p_j agrees with t_j on the subterm t^*.

- $p = k\ p'$ and $t^* = t = k\ t'$.
- $p = k\ p'$, $t = k\ t'$, t^* is a subterm of t' and p' agrees with t' on t^*.

Clearly a pattern p matches a term t if and only if p agrees with t on all the subterms of t. A pattern does not need to be of exactly the same type as a value in order to match it. In fact, p of type τ may match t of type σ if and only if σ is an instance of τ i.e., $\sigma \leq \tau$.

Matching criterion for a sequence of patterns: the matching problem

A matching problem consists of a finite sequence of patterns $p_1, \ldots, p_n$ of type τ and a value term t of type τ' where $\tau' \leq \tau$. The solution is the first pattern in the sequence which matches t, if one exists. If none of the patterns matches t, the matching problem has no solution.

4. Decision trees and the dispatching problem

A function in clausal form and a particular argument value determine a matching problem consisting of the pattern sequence from the function's rules and the value term represented by the argument. The ML system must solve this matching problem in the course of evaluating the function application. The most straightforward approach would be to attempt to match the argument with each pattern in turn (using the obvious top-down matching routine), starting over after every failure. This algorithm is quite inefficient since the information gained about the structure of the argument in each partially successful match is discarded before doing the next match. We propose instead to analyze the whole sequence of patterns at compile time and generate efficient run-time matching code. Efficiency depends on avoiding backtracking, and carefully choosing the order in which to explore the argument so as to determine the correct matching pattern with as little effort as possible (*i.e.* the smallest number of case discrimination tests). Example 2 exhibits the effect of the order of subterm testing on the efficiency of the matching process.

Example 2 (continued). We consider again the term $t = (true,\ cons\ (1,\ nil))$ and the sequence of patterns of Example 2:

(1) $p_1 = (true,\ \_)$

(2) $p_2 = (false,\ nil)$

(3) $p_3 = (false,\ cons\ (x,\ nil))$

(4) $p_4 = (false,\ cons\ (y,\ z))$

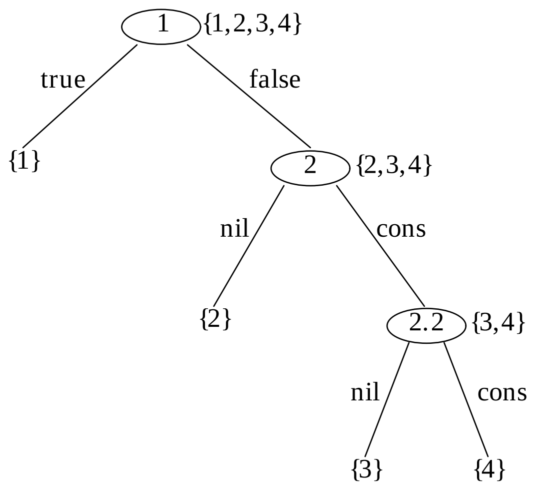
We want to find which pattern of the sequence matches t, *i.e.*, the first pattern that agrees with t on all its subterms. To do this, we must compare the subterms of t with subpatterns, and there are different orders in which this can be performed. We can start testing on the first component or on the second component of the tuple t. Suppose we start testing on the first one. We find that p_1 is the only pattern which agrees with t on its first component. And since the second component of p_1 is _ (wildcard) which matches everything, no further test has to be performed to establish that p_1 is the solution of this matching problem. Another possibility would be to start testing on the second component of t instead. This discards p_2 whose second component does not agree with *cons*. We are left with patterns p_1, p_3 and p_4. Now, there are three tests that can possibly be performed : test on the first component of t, on the first component of the argument of *cons* or on the second component of the argument of *cons*. The test on the first component of t still leads immediately to the desired solution. The tests on the arguments of *cons* are useless : they do not help discarding any pattern because p_1, p_3 and p_4 all agree with t on these subterms. We will say that these tests are irrelevant to p_1, p_3 and p_4 (see section 5).

We describe our goal as follows.

The dispatching problem (1)

> Given a sequence of patterns $p_1, \ldots, p_n$ of type τ, find out in which order the subterms of any possible term t of type τ' ($\tau' \leq \tau$) have to be examined to determine with the minimum number of tests on the subterms of t, which pattern p_i ($1 \leq i \leq n$) is the solution to the matching problem defined by $p_1, \ldots, p_n$ and t.

When compiling a function in clausal form, we build a *decision tree* which determines the order in which subterms of any term are to be examined to find the first pattern that matches that term. We attempt to make this decision tree optimal or minimal, in the sense that the order imposed on subterm-testing is such that the matching pattern can be found with a minimum number of tests. Each node of a decision tree represents a test that can be carried out on a sub-term discriminating between constructor cases. The branches coming out of a node correspond to the possible results of the test performed at that node (*i.e.* possible constructors for that subterm, which are determined by the type of the subterm). Each branch is labeled with a type constuctor and a set of pattern indices representing the patterns that were still possibly matching (“live”) before the test and that have this constructor as result of the test. At run-time, when a value term has to be matched against one of the patterns, the code executed corresponds to going down the decision tree from the root to one of the leaves, executing the tests corresponding to the test-nodes along the path. Figure 1 exhibits the optimal decision tree for the match given in Example 2.



- Figure 1 -

A decision tree is either an empty tree or a test-node. Each test-node carries two kinds of information: an identification of the subterm of a value term on which the test is to be carried out (the numbers in the nodes on Figure 1) and a set containing the indices of the patterns which have not been discarded yet at this point in the tree (the sets attached to the nodes on the figure). Each branch coming out of a test-node is labelled with constructors of the type of the subterm to which the test applies. No constructor can appear on more than one branch coming out of a test-node. A leaf is a node that carries either a singleton containing the index of the pattern which is identified as the matching pattern by the sequence of tests corresponding to the path from the root to it if such a pattern exists, or an empty set.

The decision tree of a sequence of patterns is an empty tree if the sequence is empty or if it has only one rule with a pattern equal to wildcard or a variable (i.e., no test is necessary to perform a matching). Otherwise, it is a test-node having attached to it the set of indices of all the patterns in the sequence.

A *minimal decision tree* is a decision tree which has the minimum possible number of test-nodes.

The dispatching problem can be reformulated as follows :

The dispatching problem (2)

Build an optimal decision tree for a given sequence of patterns p_1, .., p_n of type τ.

Theorem: The dispatching problem is NP-complete.

The proof uses reduction from the pruned O-trie space minimization problem described in [Co76, CS77]. This result motivates us to look for practical heuristics for building decision trees.

5. Heuristic approach

Suppose we are given a sequence of patterns p_1, ..., p_n and we wish to build the associated optimal decision-tree. This effectively involves choosing for any given value term t, the order in which subterms of t have to be tested. The subterm-testing order can be different for different terms.

In order to get an optimal decision-tree, we have to select, at every stage of the top-down construction, the best test-node among a set of possible tests. As indicated by the NP-completeness of the dispatching problem, such a search leads to an exponential explosion of the computation. Therefore the selection of a test uses heuristics and actually produces the heuristically optimal test. In this section we describe a sequence of heuristics that have been found to produce optimal trees in almost all cases. The relevance heuristic is applied to the original set of possible tests. If it succeeds in isolating one test, the search ends, otherwise, the branching factor heuristic is applied to the tests considered equivalently optimal by the relevance heuristic. If the branching factor heuristic does not isolate a test, then the arity factor heuristic is applied to the set of tests considered equivalent under the previous heuristic.

The relevance heuristic

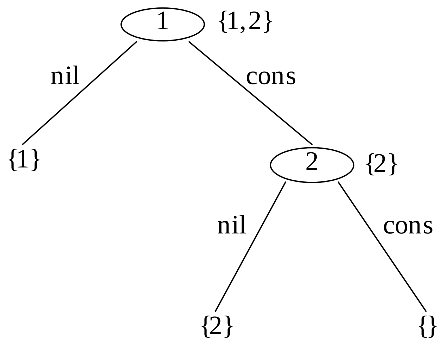
This heuristic tends to minimize the decision-tree and also the matching time by taking advantage of the fact that we deal with sequences rather than sets of patterns and that if two patterns overlap, the one with the lowest index is preferred. Example 3 illustrates the idea.

Example 3. We consider the following patterns of type (α *list* * α *list*).

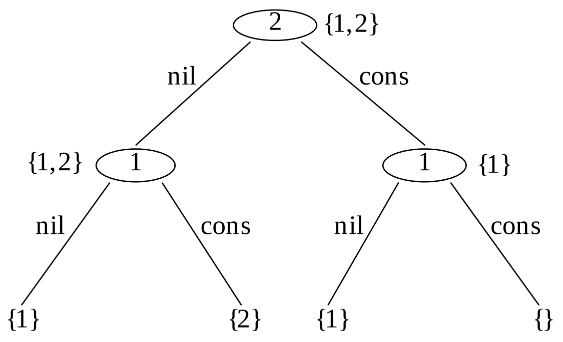
(1) (*nil*, *x*)

(2) (*y*, *nil*)

There are only two possible decision-trees for this sequence (see Figures 2 and 3) depending on whether one starts testing on the first or on the second component of the argument.



- Figure 2 -



- Figure 3 -

The tree on Figure 2 has two decision-nodes whereas the one on Figure 3 has three such nodes, so the former is optimal. Pattern (1) matches any value term which has *nil* as first component whatever the second component is. Pattern (2) matches all the value terms having *nil* as second component and which are not matched by (1). Therefore, it is better to start testing on the first component which will more quickly isolate pattern (1) as a leaf of the decision-tree. We will say that the test on the second component is **not relevant** to pattern (1) because pattern (1) appears on all the branches coming out of that test-node. The test on the first component of the tuple is relevant to pattern (1) but not to pattern (2). The test on the second component is relevant to pattern (2) but not to pattern (1). Since we deal with sequences of patterns, we try to select the tests that are relevant to patterns of lower index first.

A test on a subterm is **relevant** to a pattern p_i if and only if p_i does not agree with all the possible values on that subterm. In terms of decision-trees, a test on a subterm is relevant to a pattern p_i if and only if i does not appear in the set of live rule indices which label each successor of that test. Given a set of possible tests *tset* and a set of live rule indices *rset*, the relevance heuristic searches for the least index i in *rset* such that at least one test in *tset* is relevant to p_i. If there is no such index, no test in *tset* is relevant to any pattern in *rset* and one has reached a leaf of the tree. Otherwise, one computes the subset *trel* of *tset* containing the tests that are relevant to p_i. If *trel* is a singleton, its element is the next desired next test. Otherwise, the next heuristic selection is applied on *trel*.

The branching factor heuristic

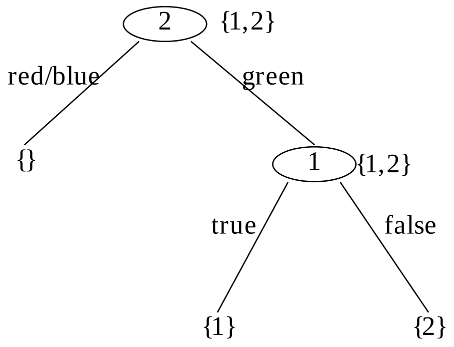
The branching factor heuristic tries to minimize the number of test-nodes by favoring the choice of tests with low branching factor first. The intuition behind this is that the decision-tree will end up with fewer internal nodes if the nodes selected first (i.e., closer to the root) have lower branching factor (i.e., lower number of successors). Example 4 illustrates the idea.

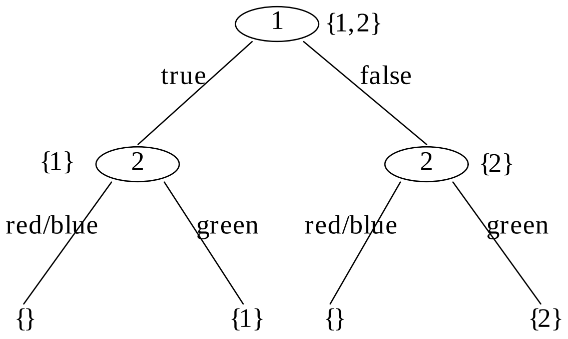
Example 4. We consider the following sequence of patterns of type (*bool* * *color*) where *color* is a concrete type defined by : *type color = red* | *blue* | *green*.

(1) (*true*, *green*)

(2) (*false*, *green*)

Figures 4 and 5 exhibit the two possible decision-trees for this sequence of patterns.





- Figure 4 - *- Figure 5 -*

Even though the *color* type has three possible constructors, only one of them, *green*, is present in the sequence of patterns and the remaining ones are not covered by any pattern (there is no pattern with a variable or wildcard as that subterm). This sequence is obviously not exhaustive. The effective branching factor of the test on the second component of the tuple is one. There are two constructors of the type *bool* (*true* and *false*) and they are both present in the sequence. So the branching factor of that test is two. Figure 4 shows the optimal decision-tree. Its root is a test on the second component of the tuple i.e., the test with the least branching factor.

The criterion we implemented for this heuristic selection encompasses not only the branching factor but also some information which tends to favor tests on subterms where no pattern agrees with all values (i.e., has a variable or wildcard as that subterm or as a more general subterm). This relies on the idea that when a pattern agrees with all values on a particular subterm, the test on this subterm is irrelevant to the pattern and performing it does not help discriminating the pattern. The full paper will give a more detailed description of how to evaluate the criterion used by this heuristic.

The arity factor heuristic

The **arity factor** of a test on a subterm of type τ is equal to the sum of the arities of the constructors which appear at least once in the corresponding subterm of a pattern in the sequence.

Our strategy consists in selecting tests with low arity factor first. This corresponds intuitively to minimizing the potential branching of the tree and once again in trying to put off the selection of nodes with high branching, therefore minimizing the number of test-nodes in the tree.

6. Conclusions

The problem of pattern matching is crucial to the efficiency of function calls in ML, and therefore it is critical to the efficiency of the language as a whole. Although the NP-completeness result indicates the intrinsic difficulty of a complete analysis of a set of patterns, we have found that in practice the heuristically based methods described here work extremely well. The algorithm embodying these heuristics has been extensively tested and will be incorporated in a new ML compiler being developed by one of the authors.

Future work on this problem will be directed to refining our analysis of the behavior of the heuristics and acquiring statistical evidence of their effectiveness in an actual compiler. There are several possible generalizations of the matching problem to which these methods could be applied, including nonlinear and conditional pattern matching.

References

[AC75] A. V. Aho and M. J. Corasick, ‘‘Efficient String Matching: An Aid to Bibliographic Search’’, *Communications of the ACM,* Vol. 18, No. 6, June 1975, pp. 333-340.

[AG] A. V. Aho, M. Ganapathi, ‘‘ Efficient Tree Pattern Matching: an Aid to Code Generation’’, *12th Annual ACM Symposium on Principles of Programming Languages*, New Orleans, January 1985, pp. 334-340.

[Au84] L. Augustsson, ‘‘A Compiler for Lazy ML’’, *1984 ACM Symposium on Lisp and Functional Programming,* Austin, Texas, August 1984, pp. 218-227.

[Au84] L. Augustsson, ‘‘Compiling pattern matching’’, *Functional Programming Languages and Computer Architecture*, J-P. Jouannaud, Ed., Lecture Notes in Computer Science, Vol 201, Springer-Verlag, Berlin, 1985, pp. 368-381.

[Ca84] L. Cardelli, ‘‘Compiling a Functional Language’’, *1984 ACM Symposium on Lisp and Functional Programming,* Austin, Texas, August 1984, pp. 208-217.

[Ca83a] L. Cardelli, ‘‘The Functional Abstract Machine’’, Technical Report TR-107, AT&T Bell Laboratories, Murray Hill, New Jersey, 1983.

[Ca83b] L. Cardelli, ‘‘ML under Unix’’, *Polymorphism newsletter,* Vol. I.3, December 1983.

[Co76] D. Comer, *Trie Structured Index Minimization,* Ph.D. Thesis, Pennsylvania State University, August 1976.

[CS77] D. Comer and R. Sethi, ‘‘The Complexity of Trie Index Construction’’, *Journal of the ACM,* Vol. 24, No. 3, July 1977, pp. 428-440.

[GJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability:A Guide to the Theory of NP-Completeness,* W. H. Freeman and Company, San Francisco, 1979.

[GMW82] M. J. Gordon, A. J. Milner and C. P. Wadsworth, *Edinburgh LCF,* Springer-Verlag Lecture Notes in Computer Science, Vol. 78, Berlin, 1982.

[HO79] C. M. Hoffmann and M. J. O’Donnell, ‘‘An Interpreter Generator Using Tree Pattern Matching’’, *6th Annual ACM Symposium on Principles of Programming Languages,* San Antonio, Texas, January 1979, pp. 169-179

[HO82a] C. M. Hoffmann and M. J. O’Donnell, ‘‘Pattern Matching in Trees’’, *Journal of the ACM,* Vol. 29, No. 1, January 1982, pp. 68-95.

[HO82b] C. M. Hoffmann and M. J. O'Donnell, ''Programming with Equations'', *ACM Transactions on Programming Languages and Systems,* Vol. 4, No. 1, January 1982, pp. 83-112.

[OD85] M. J. O'Donnell, *Equational Logic as a Programming Language,* MIT Press, Cambridge, Massachusetts, 1985.

[Mi78] R. Milner, ''A Theory of Type Polymorphism in Programming'', *Journal of Computer and System Sciences,* Vol. 17, No. 3, December 1978, pp. 348-375.

[Mi84a] R. Milner, ''A Proposal for Standard ML'', *1984 ACM Symposium on Lisp and Functional Programming,* Austin, Texas, August 1984, pp. 184-197.

[Mi84b] R. Milner, ''The Standard ML Core Language'', Internal Report CSR-168-84, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, October 1984.

[Mi85] R. Milner, ''The Standard ML Core Language (Revised)'', *Polymorphism*, II, 2, October 1985.