Compiling Successor ML Pattern Guards

John Reppy
University of Chicago
USA
jhr@cs.uchicago.edu

Mona Zahir
University of Chicago
USA
zahirmona@gmail.com

Abstract

Successor ML is a collection of proposed language extensions to Standard ML. A number of these extensions address pattern matching; including adding richer record patterns, or-patterns, and pattern guards. Pattern guards in Successor ML are more general than those found in other languages, which raises some interesting implementation issues.

This paper describes the approach to pattern guards that we are developing as part of an effort to add Successor ML features to the Standard ML of New Jersey system. We present our approach in a way that is applicable to either backtracking or decision-tree implementations of pattern matching.

This paper was originally presented at the 2019 SIGPLAN ML Family Workshop in Berlin Germany, August 2019. It has been revised to fix some typographical errors and to better explain the algorithms.

1 Introduction

Successor ML [SML15] is a collection of proposed language extensions to the 1997 Definition of Standard ML [MTHM97]. A number of these extensions address pattern matching, by adding richer record patterns, or-patterns, and pattern guards; the last of these features is the focus of this paper.

Languages, such as Haskell, OCaml, and Scala, have a restricted form of conditional patterns, where the guards are part of the match or case syntax. Compiling such guards is fairly straightforward [Pet99b]. The pattern guards of Successor ML are more general, since they are a syntactic form of pattern and, thus, can be nested inside other pattern constructs. This more general version of pattern guards allows another extension, called *transformation patterns*,[1] to be defined as a derived form [Ros08], but it also raises some interesting implementation issues that have not been previously addressed in the literature.

As part of an ongoing effort to extend Standard ML of New Jersey with Successor ML features,[2] we have been prototyping a new pattern-match compiler. This paper describes the approach that we have devised to handle Successor ML's general form of pattern guards.

There are two common approaches to implementing pattern matching in lower-level code:

- Backtracking-based solutions, which have the advantage that the resulting low-level code is linear in the size of the input, but which may do redundant work [Aug85, LFM01].
- Decision-tree-based solutions, which avoid redundant tests, but can suffer exponential blowup in the size of the generated code [BM85, Pet99b, Mar08].

While these approaches have different properties, in both cases the compilation schemes typically use a *pattern matrix* as the central data structure and are organized around a set of rules for analyzing and transforming the pattern matrix. Our approach to compiling pattern guards extends the pattern matrix representation with *guard columns* and associated rules; thus, the approach presented in this paper can be applied to generate either backtracking or decision-tree implementations of the compiled match.

2 Successor ML Pattern Guards

As mentioned above, Successor ML extends the Standard ML pattern syntax with *pattern guards*, which have the form "p **with** p' = e." The semiformal semantics of this construct is given by the following inference rules:

$$\frac{s_0, E, v \vdash p \Rightarrow \mathit{FAIL}, s_1}{s_0, E, v \vdash p \;\mathbf{with}\; p' = e \Rightarrow \mathit{FAIL}, s_1}$$

$$\frac{\begin{array}{c} s_0, E, v \vdash p \Rightarrow \mathit{VE}_1, s_1 \\ s_1, E + \mathit{VE}_1 \vdash e \Rightarrow v', s_2 \qquad s_2, E + \mathit{VE}_1, v' \vdash p' \Rightarrow \mathit{FAIL}, s_3 \end{array}}{s_0, E, v \vdash p \;\mathbf{with}\; p' = e \Rightarrow \mathit{FAIL}, s_3}$$

$$\frac{\begin{array}{c} s_0, E, v \vdash p \Rightarrow \mathit{VE}_1, s_1 \\ s_1, E + \mathit{VE}_1 \vdash e \Rightarrow v', s_2 \qquad s_2, E + \mathit{VE}_1, v' \vdash p' \Rightarrow \mathit{VE}_2, s_3 \end{array}}{s_0, E, v \vdash p \;\mathbf{with}\; p' = e \Rightarrow E + \mathit{VE}_2, s_3}$$

where s is a state, VE is a value environment, and v is a value.

The third rule describes a successful match, where we first test if p matches v and then, if it does, evaluate e to a value v' and test if p' matches v'. The variables bound by p are in scope in e and the dynamic environment is further extended by the variable bound in p. Furthermore, the state is threaded through these three steps, which fixes the order of evaluation in the presence of side effects.

Successor ML also provides the form "p **if** e" as syntactic sugar for "p **with** `true` = e."

The complete semantics of Successor ML pattern matching can be found in the Definition [SML15], but the most important aspect of it is that patterns (and sub-patterns) are matched in left-to-right order. This order restricts the flexibility of a

[1]Transformation patterns are a simple view mechanism.
[2]Rossberg's *HaMLet S* [Ros08] is the only complete implementation of successor ML, but it does not compile patterns. Both MLton and Standard ML of New Jersey have implemented a small subset of Successor ML features, but not the pattern extensions.

pattern-match compiler when pattern guards are present, since the order of any side effects that might be present must be preserved. In fact, a guard might even modify the value being matched, as illustrated in the following code snippet:

```
let val x = ref 2 in
  case x of
  | (ref 1) => ...
  | (ref _ if (x := 1; false)) => ...
  | (ref 1) => ...
end
```

which will match the third clause of the **case** expression.

3 Clause Matrices and Guard Columns

Our approach to compiling pattern guards can be used in both back-tracking and decision-tree compilation schemes. In this section, we describe the common aspects of these two schemes. We assume that the translation produces an expression only simple patterns are used (*i.e.*, a tuple, variable, or single constructor application).

For purposes of the presentation, we consider a simplified pattern language, with wildcards, constructors, and the derived form of pattern guards:[3]

$$p ::= \_ \mid c(p_1, \ldots, p_n) \mid (p\ \mathbf{if}\ e)$$

Most presentations of pattern-matching algorithms work on a vector of n variables $\vec{x}$ and a *clause matrix* $P \to L$, where P is an $m \times n$ pattern matrix and L is an m-element column vector of actions. This input can be viewed as representing a case of the form

```
case (x_1, ..., x_n) of
| (p_1^1, ..., p_n^1) => l^1
| ...
| (p_1^m, ..., p_n^m) => l^m
```

(we follow the convention of using superscripts to denote row indices and subscripts to denote column indices).

We extend this representation of a clause matrix to include columns of guard expressions, where we use the syntax "$\{e\}$" to distinguish a guard from a pattern. We also extend the notation for variable vectors to include the "•" symbol in positions that correspond to guard columns. We define a *trivial entry* in a pattern matrix to be either a wild card or the trivial guard {true}.

Compilation of the clause matrix proceeds by case analysis on one of the columns of P. For simplicity, we usually pick the first column, but when compiling to a decision tree, it may be advantageous to pick a different column [SR00, Mar08]. The details of the case analysis and subsequent transformations depend on the particular compilation scheme, but we need the following transformation for both schemes.

The *guard-split transformation* $\mathcal{G}_j$ splits the jth column of a pattern matrix into a two columns: a pattern column and a guard column. $\mathcal{G}(P \to L)$ is defined as follows:

p_j^i	Row i in $\mathcal{G}_j(P \to L)$
$(p\ \mathbf{if}\ e)$	$p_1^i \quad \cdots \quad p \quad \{e\} \quad \cdots \quad p_n^i \quad \to \quad l^i$
p	$p_1^i \quad \cdots \quad p \quad \{\mathtt{true}\} \quad \cdots \quad p_n^i \quad \to \quad l^i$

4 Compiling Pattern Guards with Backtracking

We first consider how to extend the "Classical Backtracking Scheme" $\mathcal{C}$, as described by Le Fessant and Maranget [LFM01], to include pattern guards. The compilation scheme $\mathcal{C}$ takes two arguments: a vector of n variables $\vec{x}$ and a clause matrix $P \to L$ (P is $m \times n$), and is defined by analysis of the first column of P:

- If $n = 0$, then the result is e^1.
- If all the the p_1^i $(1 \le i \le m)$ are all wild cards, then the *Variable Rule* applies, which effectively drops x_1 and the first column of the P.
- If all the p_1^i $(1 \le i \le m)$ are all constructor patterns, then the *Constructor Rule* applies. In this case, we construct a simple **case** expression on x_1, where there is a clause

  $c(y_1, \ldots, y_k)$ **=>**
  $\qquad \mathcal{C}(\langle y_1, \ldots, y_k, x_2, \ldots, x_n\rangle, \mathcal{S}(c,\ P \to L)$

  The right-hand-side of the clause is the projection of those rows in $P \to L$ that have a pattern of the form $c(q_1, \ldots, q_k)$ in the first position, where we then replace the first pattern by $q_1 \cdots q_k$. If the constructors in the first column are not exhaustive, we add a default rule that backtracks.
- Otherwise, the *Mixture Rule* applies, which splits the matrix into maximal blocks of rows, where each block is covered by one of the other rules. Each block is recursively compiled to an expression and the expressions are combined in series using backtracking.[4]

Note that these rules test patterns in a left-to-right order that respects the semantics in the presence of effectful pattern guards. We extend this $\mathcal{C}$ with three rules related to pattern guards.

4.1 The Guard-split Rule

If any pattern p_1^i in the first column has the form $(p\ \mathbf{if}\ e)$, we apply the transformation $\mathcal{G}_1$, to the clause matrix $P \to L$ and then invoke $\mathcal{C}$ on the result. Thus, the Guard-split Rule is implemented as

$$\mathcal{C}(\langle x_1,\ \bullet,\ x_2,\ \ldots,\ x_n\rangle,\ \mathcal{G}(P \to L))$$

[3]Because of space limitations, we are ignoring variable bindings. These can either be handled using occurrences *à la* Pettersson [Pet99b] or by modifying the actions with **let** bindings.

[4]Le Fessant and Maranget use a special **catch/exit** construct to implement backtracking, but one can also continuation functions.

We apply the Guard-split Rule eagerly (*i.e.*, when any pattern in the first column is a pattern guard), since the transformation can expose larger blocks of rows for the other rules, which is particularly beneficial for backtracking implementations. This rule does not reorder patterns, so the left-to-right evaluation order is preserved.

4.2 The Guard-match Rule

If the first column of P is a guard column, p_1^1 is $\{e\}$ (where e is not the expression `true`), and p_1^i is $\{\texttt{true}\}$ for $1 < i \le m$, then we generate the code

$$\textbf{let val } x' = e \textbf{ in } C(\langle x', x_2, \ldots, x_n\rangle, \mathcal{M}(P \to L))$$

where $\mathcal{M}(P \to L)$ is defined as follows:

p_1^i	Row i in $\mathcal{M}(P \to L)$
$\{\texttt{true}\}$	$\_ \quad p_2^i \quad \ldots \quad p_n^i \quad \to \quad l^i$
$\{e\}$	$\texttt{true} \quad p_2^i \quad \ldots \quad p_n^i \quad \to \quad l^i$

Since the resulting code evaluates e before any patterns to the left or below are checked, the order of evaluation is correct. Combining the remaining patterns in the first row with the rest of P's rows allows for optimizations, such as reordering rows, to include the first row.

4.3 The Trivial-guard Elimination Rule

If the first column of P consists solely of the always-true guard, then we can remove the the first variable of $\vec{x}$, which will be $\bullet$, and the first column of P. We then call C on the reduced variable vector and clause matrix.

5 Compiling Pattern Guards to Decision Trees

Our approach to pattern guards can also be incorporated into a compilation scheme that generates decision trees (or DAGs). In this section, we decribe an approach based on Pettersson's algorithm [Pet99b].

Pettersson's compilation scheme $\mathcal{D}$ takes two arguments: a vector of n variables $\vec{x}$ and a clause matrix $P \to L$ (P is $m \times n$), and is defined by analysis of the first row of P:

- (Variable Rule) If all p_i^1 $(1 \le i \le n)$ are wild cards, then the result is l^1.
- (Mixture Rule) Otherwise, there is some i such that p_i^1 is a constructor pattern. In this case, we create a new sub-matrix for each unique outermost constructor in column i. For a constructor c, any row in $P \to L$ that either has a wild card or an outermost c in column i is included in the sub-matrix. We then create a simple **`case`** expression that matches x_i against the constructors from column i. If the set of constructors in column i is not exhaustive, then we add a default rule, which either raises the `Match` exception (if there were no variables in the column) or goes to an expression generated from the matrix formed by projecting out all rows with a variable in the ith column.

We can extend Pettersson's algorithm with pattern guards, by using variations of the three rules we introduced above.

Before discussing the transformations related to pattern guards, however, we need to consider the interaction between the Mixture Rule and pattern guards. Specifically, we cannot apply the Mixture Rule to a column that is to the **right** of a non-trivial guard. For example, consider the expression

```
case (x, y) of
| (_ if e, false) => ...
| (_, true) => ...
```

which produces the following pattern matrix (after applying $\mathcal{G}_1$).

$$\begin{pmatrix} \_ & \{e\} & \texttt{false} \\ \_ & \{\texttt{true}\} & \texttt{true} \end{pmatrix}$$

The Mixture Rule as stated allows one to specialize the third column, but this transformation results in incorrect code that fails to evaluate e when `y` is `true`.

Another issue that we must consider is when the values being matched are mutable and there is a pattern guard. In this case, any knowledge about the mutable values that we might have acquired prior to evaluating the guard expression may have been invalidated by the expression (*e.g.*, the example in Section 2).

Putting it all together, we get the following scheme for compiling patterns to decision trees:

- (Guard-split Rule) If there is a pattern p_j^i of the form (q **`if`** e), then we apply G_j to the matrix to split column j. We apply this transformation eagerly until there are no such patterns left.
- (Trivial-guard Elimination Rule) If the ith column in P consists solely of the always-true guard, then we remove the the ith variable from $\vec{x}$, which will be $\bullet$, and the ith column of P.
- (Variable Rule) If every p_i^1 $(1 \le i \le n)$ is trivial, then the result is l^1.
- (Guard-match Rule) If the leftmost non-trivial entry in the first row is of the form $\{e\}$ and is in column i, then we examine the guards in column i below the first row. If they are all trivial, then we generate the code

  $$\begin{array}{l}\textbf{let val } x' = e \textbf{ in} \\ \quad \mathcal{D}(\langle x_1, \ldots, x', \ldots, x_n\rangle, \mathcal{M}_i(P \to L)) \\ \textbf{end}\end{array}$$

  where $\mathcal{M}_i(P \to L)$ is $\mathcal{M}$ from Section 4.2 generalized to work on arbitrary columns.

  If there is a non-trivial guard in row $j > 1$, we split $P \to L$ into the first $j - 1$ rows ($P' \to L'$) and the rest of the clause matrix ($P'' \to L''$). We add a default rule to the end of ($P' \to L'$) that maps to the compilation of

$P'' \rightarrow L''$, and then compile the extended $P' \rightarrow L'$ as before.

- (Reference Rule) If the leftmost non-trivial entry in the first row is a constructor pattern involving the `ref` constructor and the pattern matrix contains at least one guard column or guard pattern, then we split the matrix into two sub-matrices, such that the last row of the first matrix is the first row that contains either a non-trivial guard or a pattern guard. We then connect these compiled sub-matrices by adding a default rule to the first matrix that maps to the code generated for the second matrix. If the partitioning process only produces a single matrix, then we apply the mixture rule.
- (Mixture Rule) Otherwise, the leftmost non-trivial entry in the first row must be a constructor pattern, so we apply the Mixture Rule as described above.

6 Interaction with Optimizations

Pattern guards pose some minor difficulties for the optimization techniques used in many compilation schemes. For example, the constructor rule usually computes the set of constructors that appear in the column being scrutinized; if this set covers the datatype, then no default case is required in the resulting switch. If a row contains a pattern guard, however, its constructor cannot be included in this analysis, since the guard may fail. This observation also applies when analyzing matches for useless patterns [Mar07].

Another example, is using heuristics to pick the column to scrutinize [SR00]. As demonstrated in the previous section, it is not sound to skip over columns that contain pattern guards, since that can cause guards to be skipped.

On the other hand, reordering incompatible rows to optimize the constructor rule [LFM01] is still a valid transformation. One detail is that one must extend the notion of compatibility to include pattern guards ((p **`if`** e) is compatible with q if p is compatible with q) and to treat guards as compatible with all other guards ($\{e\}$ is compatible with $\{e'\}$ for all e and e'). Moving a row above a guard is also not allowed when there is a column that involves references.

7 Conclusion

We have described a compilation scheme for the pattern guard mechanism found in Successor ML. Prototype implementations of our approach can be found at https://github.com/JohnReppy/compiling-pattern-guards. This scheme should be compatible with any pattern-matrix-based approach to compiling matches. We plan to incorporate our approach in a upcoming reimplementation of pattern matching in the SML/NJ system.

References

[Aug85] Augustsson, L. Compiling pattern matching. In *FPCA '85*, Nancy, France, September 1985. Springer-Verlag, pp. 368–381.

[BM85] Baudinet, M. and D. B. MacQueen. Tree pattern matching for ML. Available from http://www.smlnj.org/compiler-notes/85-note-baudinet.ps, 1985.

[LFM01] Le Fessant, F. and L. Maranget. Optimizing pattern matching. In *ICFP '01*, Florence, Italy, September 2001. ACM, pp. 26–37.

[Mar07] Maranget, L. Warnings for pattern matching. *JFP*, **17**(3), 2007, pp. 387–421.

[Mar08] Maranget, L. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, Victoria, BC, Canada, 2008. ACM, pp. 35–46.

[MTHM97] Milner, R., M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.

[Pet99a] Pettersson, M. *Compiling Natural Semantics*. Number 1549 in LNCS. Springer-Verlag, New York, NY, 1999.

[Pet99b] Pettersson, M. *Compiling Pattern Matching*, chapter 7, pp. 85–109. In *LNCS* [Pet99a], 1999.

[Ros08] Rossberg, A. *HaMLet S — To Become or Not Become Successor ML*, April 2008. Available at https://people.mpi-sws.org/~rossberg/hamlet/hamlet-succ-1.3.1S5.pdf.

[SML15] The Definition of Successor ML. Available at https://github.com/SMLFamily/Successor-ML, 2015.

[SR00] Scott, K. and N. Ramsey. When do match-compilation heuristics matter? *Technical Report CS-2000-13*, Dept. of Computer Science, University of Virginia, May 2000.