

Compiling Successor ML Pattern Guards

John Reppy

Mona Zahir

University of Chicago

August 2019

Successor ML

“Successor ML” is a collection of improvements to Standard ML ranging from minor syntactic improvements to core language extensions to module system changes.

The only implementation of Successor ML is Rossberg’s **HaMLeT-S** reference implementation.

A few years ago, however, MLton and SML/NJ began to add Successor ML features.

This talk is about the proposed pattern guard mechanism and how to implement it.

Successor ML Patterns

Successor ML adds several new pattern forms:

- ▶ “`... = pat`” — bind names to rows
- ▶ “`pat1 as pat2`” — conjunctive patterns
- ▶ “`pat1 | pat2`” — disjunctive patterns (in SML/NJ since 1992)
- ▶ “`pat1 with pat2 = exp`” — nested match
- ▶ “`pat if exp`” — conditional pattern, which is syntactic sugar for “`pat with true = exp`”

The last two forms are known as “**pattern guards**” and interleave evaluation of arbitrary expressions with pattern matching.

Note that Successor ML pattern guards are more general than similar mechanisms found in Erlang, Haskell, OCaml, Moby, and Scala, where guards are limited to boolean expressions that are part of the match rule syntax.

Pattern Guard Dynamics

With pattern guards in the language, the dynamic semantics of pattern matching must track the dynamic state in addition to the dynamic environment.

$$s_0, E, v \vdash p \Rightarrow VE/FAIL, s_1$$

Pattern Guard Dynamics

A guard pattern matches a value v when the first pattern p matches v and the second pattern p' matches the result of evaluating the expression e .

$$\frac{s_0, E, v \vdash p \Rightarrow VE_1, s_1 \quad s_1, E + VE_1 \vdash e \Rightarrow v', s_2 \quad s_2, E + VE_1, v' \vdash p' \Rightarrow VE_2, s_3}{s_0, E, v \vdash p \text{ **with** } p' = e \Rightarrow E + VE_2, s_3}$$

Pattern Guard Dynamics

The match fails when the first pattern p does not match the value v .

$$\frac{s_0, E, v \vdash p \Rightarrow \text{FAIL}, s_1}{s_0, E, v \vdash p \text{ \textbf{with} } p' = e \Rightarrow \text{FAIL}, s_1}$$

Pattern Guard Dynamics

The match also fails when the second pattern p' does not match the result of evaluating e .

$$\frac{s_0, E, v \vdash p \Rightarrow VE_1, s_1 \quad s_1, E + VE_1 \vdash e \Rightarrow v', s_2 \quad s_2, E + VE_1, v' \vdash p' \Rightarrow FAIL, s_3}{s_0, E, v \vdash p \textbf{ with } p' = e \Rightarrow FAIL, s_3}$$

Pattern Guard Examples

Here is an example of a constant-folder, where we only match the `Add` pattern if the result of adding the arguments is representable as a 32-bit number.

```
datatype value = Num of IntInt.int | ...  
datatype prim = Add of value * value | ...  
val i32Add : IntInf.int * IntInf.int -> IntInf.int option  
  
fun contract (Add(Num a, Num b) with SOME c = i32Add(a, b)) = Num c  
  | ...
```


Pattern Guard Examples

This example for escaping a character uses nested pattern guards to first bind the codepoint to `n` and then test if it requires escaping.

```
fun escape #"\" = "\\\""  
  | escape #"\\\" = "\\\"\\\""  
  | escape ((c with n = ord c) if (n < 32)) =  
    "\\\"^" ^ str(chr(n+64))  
  | escape c = str c
```

Pattern Guard Examples

Pattern guards may have side effects, which can change the values being matched against.

```
let val x = ref 2
in
  case x of
  | (ref 1) => "a"
  | (ref _ if (x := 1; false)) => "b"
  | (ref 2) => "c"
  | _ => "d"
end
```

Transformation Patterns

Transformation patterns are a “poor-man’s” view mechanism that can be defined as a derived form using guard patterns.

$$\begin{aligned} ? \text{ atexp} &\implies (x \text{ **if** atexp } x) \\ ? \text{ atexp atpat} &\implies (x \text{ **with** SOME(atpat) = atexp } x) \end{aligned}$$

where x is a fresh variable.

Here is the constant-folding example, where primops take variables as arguments:

```
datatype prim = Add of var * var | ...  
val lookup : var -> value option  
val i32Add : IntInf.int * IntInf.int -> IntInf.int option  
  
fun contract (Add(?lookup(Num a), ?lookup(Num b))  
    with SOME c = i32Add(a, b)) = Num c  
    | ...
```

Compilation Schemes

There are three kinds of compilation scheme used to implementing pattern matching:

1. row-by-row checking (*i.e.*, a direct implementation of the semantics)
2. compilation schemes that produce code that use backtracking
3. compilation schemes that produce decision trees

We describe our implementation of pattern guards using backtracking, but the paper also discusses applying our techniques to decision-tree schemes.

Preliminaries

We assume a simple pattern language

$$p ::= _ \mid c(p_1, \dots, p_n) \mid (p \text{ **if** } e)$$

An $m \times n$ **pattern matrix** P is a matrix of patterns with m rows and n columns.

$$\left[\begin{array}{ccc} p_1^1 & \cdots & p_n^1 \\ \vdots & & \vdots \\ p_1^m & \cdots & p_n^m \end{array} \rightarrow \begin{array}{c} l^1 \\ \vdots \\ l^m \end{array} \right]$$

We extend P to a **clause matrix** $P \rightarrow L$ by adding a m -element column of actions L .

Pattern Matching via Backtracking

We present our compilation scheme as a modification of the “Classic Backtracking Scheme” described by Le Fessant and Maranget at ICFP 2001.

The scheme is described by the function

$$\mathcal{C}(\vec{x}, P \rightarrow L, k)$$

where \vec{x} are the variables being matched, $P \rightarrow L$ is the clause matrix being compiled, and k is the backtracking continuation.

For a case expression

```
case exp of  
| pat1 => exp1  
...  
| patm => expm
```

we generate the code

```
let val x = exp  
    fun k () = raise Match  
in  
     $\mathcal{C} \left( \langle x \rangle, \begin{bmatrix} pat^1 & \rightarrow & exp^1 \\ \vdots & & \vdots \\ pat^m & \rightarrow & exp^m \end{bmatrix}, k \right)$   
end
```

Pattern Matching via Backtracking

There are three cases for defining \mathcal{C} , which we illustrate by example.

Pattern Matching via Backtracking

When the first column are all variables, the **Variable Rule** applies.

$$\mathcal{C} \left(\langle x_1, x_2, \dots \rangle, \begin{bmatrix} y^1 & \dots & \rightarrow & l^1 \\ y^2 & \dots & \rightarrow & l^2 \\ y^3 & \dots & \rightarrow & l^3 \end{bmatrix}, k \right) \xrightarrow{\text{Variable Rule}} \mathcal{C} \left(\langle x_2, \dots \rangle, \begin{bmatrix} \dots & \rightarrow & [y^1 \mapsto x_1]l^1 \\ \dots & \rightarrow & [y^2 \mapsto x_1]l^2 \\ \dots & \rightarrow & [y^3 \mapsto x_1]l^3 \end{bmatrix}, k \right)$$

Pattern Matching via Backtracking

When the first column are all constructor applications, the **Constructor Rule** applies.

$$\mathcal{C} \left(\langle x_1, x_2, \dots \rangle, \begin{bmatrix} A(q^1) & \dots & \rightarrow & l^1 \\ B(q^2) & \dots & \rightarrow & l^2 \\ B(q^3) & \dots & \rightarrow & l^3 \\ A(q^4) & \dots & \rightarrow & l^4 \\ B(q^5) & \dots & \rightarrow & l^5 \end{bmatrix}, k \right)$$



case x_1 **of**

$$\begin{aligned} &| A(y) \Rightarrow \mathcal{C} \left(\langle y, x_2, \dots \rangle, \begin{bmatrix} q^1 & \dots & \rightarrow & l^1 \\ q^4 & \dots & \rightarrow & l^4 \end{bmatrix}, k \right) \\ &| B(y) \Rightarrow \mathcal{C} \left(\langle y, x_2, \dots \rangle, \begin{bmatrix} q^2 & \dots & \rightarrow & l^2 \\ q^3 & \dots & \rightarrow & l^3 \\ q^5 & \dots & \rightarrow & l^5 \end{bmatrix}, k \right) \\ &| _ \Rightarrow k() \end{aligned}$$

Pattern Matching via Backtracking

Otherwise, the **Mixture Rule** applies, which splits the matrix into maximal blocks such that one of the other rules applies to each block.

$$\mathcal{C} \left(\vec{x}, \begin{bmatrix} P^1 \rightarrow L^1 \\ P^2 \rightarrow L^2 \\ P^3 \rightarrow L^3 \end{bmatrix}, k \right)$$



```
let
fun k3 () = C (x, [ P^3 -> L^3 ], k)
fun k2 () = C (x, [ P^2 -> L^2 ], k3)
in
  C (x, [ P^1 -> L^1 ], k2)
end
```

Guard Columns

To compile pattern guards we need to track pending guard evaluations.

Our approach is to modify the representation of pattern matrices to allow two flavors of column:


1. Pattern columns, which are as before.
2. **Guard columns**, which are columns of guard expressions. We write $\{e\}$ to denote an item in a guard column.

We introduce three new rules to manipulate guard columns (one introduction and two elimination).

Guard Split Rule

When the first column has a pattern guard as its first item, the **Guard Split Rule** applies.

$$\mathcal{C} \left(\langle x_1, \dots \rangle, \left[\begin{array}{ccc} (q^1 \text{ if } e^1) & \dots & \rightarrow l^1 \\ p_1^2 & \dots & \rightarrow l^2 \\ p_1^3 & \dots & \rightarrow l^3 \\ (q^4 \text{ if } e^4) & \dots & \rightarrow l^4 \\ p_1^5 & \dots & \rightarrow l^5 \end{array} \right], k \right)$$



$$\mathcal{C} \left(\langle x_1, \bullet, \dots \rangle, \left[\begin{array}{ccc} q^1 & \{e^1\} & \dots \rightarrow l^1 \\ p_1^2 & \{\text{true}\} & \dots \rightarrow l^2 \\ p_1^3 & \{\text{true}\} & \dots \rightarrow l^3 \\ q^4 & \{e^4\} & \dots \rightarrow l^4 \\ p_1^5 & \{\text{true}\} & \dots \rightarrow l^5 \end{array} \right], k \right)$$

Trivial Guard Rule

When the first column is a guard column and all of the items are trivial, then the **Trivial Guard Rule** applies.

$$\mathcal{C} \left(\langle \bullet, x_2, \dots \rangle, \left[\begin{array}{c|c|c|c|c} \{\text{true}\} & p_2^1 & \cdots & \rightarrow & l^1 \\ \{\text{true}\} & p_2^2 & \cdots & \rightarrow & l^2 \\ \{\text{true}\} & p_2^3 & \cdots & \rightarrow & l^3 \\ \{\text{true}\} & p_2^4 & \cdots & \rightarrow & l^4 \\ \{\text{true}\} & p_2^5 & \cdots & \rightarrow & l^5 \end{array} \right], k \right) \xrightarrow{\text{red arrow}} \mathcal{C} \left(\langle x_2, \dots \rangle, \left[\begin{array}{c|c|c|c|c} p_2^1 & \cdots & \rightarrow & l^1 \\ p_2^2 & \cdots & \rightarrow & l^2 \\ p_2^3 & \cdots & \rightarrow & l^3 \\ p_2^4 & \cdots & \rightarrow & l^4 \\ p_2^5 & \cdots & \rightarrow & l^5 \end{array} \right], k \right)$$

Guard Match Rule

When the first column has a pattern guard as its first item and the first item is non-trivial, the **Guard Match Rule** applies.

$$\begin{array}{c}
 \mathcal{C} \left(\langle \bullet, x_2, \dots \rangle, \left[\begin{array}{c} \{e^1\} \\ \{\text{true}\} \\ \{\text{true}\} \\ \{\text{true}\} \\ \{\text{true}\} \end{array} \begin{array}{c} p_2^1 \\ p_2^2 \\ p_2^3 \\ p_2^4 \\ p_2^5 \end{array} \begin{array}{c} \dots \\ \dots \\ \dots \\ \dots \\ \dots \end{array} \rightarrow \begin{array}{c} l^1 \\ l^2 \\ l^3 \\ l^4 \\ l^5 \end{array} \right], k \right) \\
 \\
 \text{let val } x = e^1 \\
 \text{in} \\
 \mathcal{C} \left(\langle x, x_2, \dots \rangle, \left[\begin{array}{c} \text{true} \\ - \\ - \\ - \\ - \end{array} \begin{array}{c} p_2^1 \\ p_2^2 \\ p_2^3 \\ p_2^4 \\ p_2^5 \end{array} \begin{array}{c} \dots \\ \dots \\ \dots \\ \dots \\ \dots \end{array} \rightarrow \begin{array}{c} l^1 \\ l^2 \\ l^3 \\ l^4 \\ l^5 \end{array} \right], k \right) \\
 \text{end}
 \end{array}$$

Conclusion and Future Work

There is a prototype implementation (and revised paper) at

`https://github.com/JohnReppy/compiling-pattern-guards`

The long-term goal is to incorporate these ideas in a rewrite of the SML/NJ pattern match compiler.