How to compile pattern matching

Jules Jacobs

May 2, 2021

Abstract

This is a note about compiling ML-style pattern matching to decision trees. I try to keep it simple while still presenting the key idea of efficient pattern match compilation [Mar08].

1 Introduction

Our goal is to compile ML-style pattern matching:

$$
\begin{array}{ll}
\textbf{match } a \textbf{ with} & \\
\mid \mathsf{Add}(\mathsf{Zero}, \mathsf{Zero}) & \Longrightarrow e_1 \\
\mid \mathsf{Mul}(\mathsf{Zero}, x) & \Longrightarrow e_2 \\
\mid \mathsf{Add}(\mathsf{Succ}(x), y) & \Longrightarrow e_3 \\
\mid \mathsf{Mul}(x, \mathsf{Zero}) & \Longrightarrow e_4 \\
\mid \mathsf{Mul}(\mathsf{Add}(x, y), z) & \Longrightarrow e_5 \\
\mid \mathsf{Add}(x, \mathsf{Zero}) & \Longrightarrow e_6 \\
\mid x & \Longrightarrow e_7
\end{array}
$$

In order to keep things straight, I'll use green variables a for bound local variables that have a value and red variables x for variables that will be given a value by pattern matching.

The naive way to compile this is to test the patterns from top to bottom, and try to match a against each pattern. This is inefficient: if the outer constructor of a is an Add but the first pattern fails, then the second pattern will test if a is a Mul, even though it is already known to be an Add at this point. Then the third pattern will test once more whether a is an Add, which is redundant. We'd like to compile pattern matching to code that does **no unnecessary tests**.

This does mean that we *must* start by trying to match the first pattern: if the first pattern succeeds then any work trying to match other patterns is unnecessary. So our strategy will be to emulate the naive strategy that tries to match the first pattern completely, but when we fail in the middle of the matching process, we remember what we learnt to compile a specialised version of the remaining pattern matches that takes what we learnt into account.

2 Compiling the example

It will be useful to represent a **match** expression not as a list of patterns implicitly testing against an outer variable a, but as a list of clauses that each specify explicitly what it tests against:

$$
\begin{array}{ll}
\textbf{match} & \\
\mid a \textbf{ is } \mathsf{Add}(\mathsf{Zero}, \mathsf{Zero}) & \Longrightarrow e_1 \\
\mid a \textbf{ is } \mathsf{Mul}(\mathsf{Zero}, x) & \Longrightarrow e_2 \\
\mid a \textbf{ is } \mathsf{Add}(\mathsf{Succ}(x), y) & \Longrightarrow e_3 \\
\mid a \textbf{ is } \mathsf{Mul}(x, \mathsf{Zero}) & \Longrightarrow e_4 \\
\mid a \textbf{ is } \mathsf{Mul}(\mathsf{Add}(x, y), z) & \Longrightarrow e_5 \\
\mid a \textbf{ is } \mathsf{Add}(x, \mathsf{Zero}) & \Longrightarrow e_6 \\
\mid a \textbf{ is } x & \Longrightarrow e_7
\end{array}
$$

In general, each clause can have multiple tests and will be of the form

$$\mid a_1 \textbf{ is } pattern_1,\ \ldots,\ a_k \textbf{ is } pattern_k \quad \Longrightarrow e$$

Where $a_1, \ldots a_k$ are bound variables, and $pattern_1, \ldots, pattern_k$ are patterns. This additional flexiblity will turn out to be useful during pattern match compilation.

Our goal will be an algorithm that takes as input such a list of clauses, and outputs a tree of simple primitive pattern matches (**match**#) that test against a single constructor:

$$
\begin{array}{ll}
\textbf{match\#}\ a \textbf{ with} & \\
\mid \mathsf{C}(a_1, \ldots, a_n) & \Longrightarrow [A] \\
\mid \_ & \Longrightarrow [B]
\end{array}
$$

Let's see how to do this for the example. We start working on the first test by testing a against the Add constructor with the following pattern match:

$$
\begin{array}{ll}
\textbf{match\#}\ a \textbf{ with} & \\
\mid \mathsf{Add}(a_1, a_2) & \Longrightarrow [A] \\
\mid \_ & \Longrightarrow [B]
\end{array}
$$

Then we have to solve the following sub problem for A:

$$
\begin{array}{ll}
[A] = & \\
\quad \textbf{match} & \\
\quad \mid a_1 \textbf{ is } \mathsf{Zero},\ a_2 \textbf{ is } \mathsf{Zero} & \Longrightarrow e_1 \\
\quad \mid a_1 \textbf{ is } \mathsf{Succ}(x),\ a_2 \textbf{ is } y & \Longrightarrow e_3 \\
\quad \mid a_1 \textbf{ is } x,\ a_2 \textbf{ is } \mathsf{Zero} & \Longrightarrow e_6 \\
\quad \mid a \textbf{ is } x & \Longrightarrow e_7
\end{array}
$$

Notice how we now have multiple tests per clause, corresponding to the a_1 and a_2 in the generated **match**, which become bound variables a_1 and a_2 in A.

We can simplify these problems by pushing test against bare variables such as a_2 **is** y into the right hand sides. Then the sub problem for A becomes:

$$
\begin{array}{ll}
[A] = & \\
\quad \textbf{match} & \\
\quad \mid a_1 \textbf{ is } \mathsf{Zero}, a_2 \textbf{ is } \mathsf{Zero} & \Longrightarrow e_1 \\
\quad \mid a_1 \textbf{ is } \mathsf{Succ}(x) & \Longrightarrow \textbf{let } y = a_2 \textbf{ in } e_3 \\
\quad \mid a_2 \textbf{ is } \mathsf{Zero} & \Longrightarrow \textbf{let } x = a_1 \textbf{ in } e_6 \\
\quad \mid & \Longrightarrow \textbf{let } x = a \textbf{ in } e_7
\end{array}
$$

(actually, we could have done this for the last test already in the previous step)

We now continue matching by trying to match $a_1 = \mathsf{Zero}$, by generating the pattern match:

$$\begin{array}{ll} \textbf{match\#}\ a_1\ \textbf{with} & \\ \mid \mathsf{Zero} & \Longrightarrow [C] \\ \mid \_ & \Longrightarrow [D] \end{array}$$

And continuing recursively for $[C]$ and $[D]$.

The sub problem for $[B]$ is obtained by removing all the clauses with a **is** $\mathsf{Add}(\ldots,\ldots)$ from the original problem:

$$\begin{array}{l} [B] = \\ \quad \textbf{match} \\ \quad \mid a\ \textbf{is}\ \mathsf{Mul}(\mathsf{Zero}, x) \quad \Longrightarrow e_2 \\ \quad \mid a\ \textbf{is}\ \mathsf{Mul}(x, \mathsf{Zero}) \quad \Longrightarrow e_4 \\ \quad \mid a\ \textbf{is}\ \mathsf{Mul}(\mathsf{Add}(x, y), z) \quad \Longrightarrow e_5 \\ \quad \mid a\ \textbf{is}\ x \quad \Longrightarrow e_7 \end{array}$$

3 The general algorithm

Let's generalise and see what's going on. Given a list of clauses to generate a pattern matching tree for, we use this algorithm:

1. Push tests against bare variables a **is** y into the right hand sides using **let** $y = a$, so that all the remaining tests are against constructors.
2. Select one of the tests a **is** $\mathsf{C}(\mathscr{P}_1, \ldots, \mathscr{P}_n)$ in the first clause using some heuristic.
3. Generate this pattern match:

$$\begin{array}{ll} \textbf{match\#}\ a\ \textbf{with} & \\ \mid \mathsf{C}(a_1, \ldots, a_n) & \Longrightarrow [A] \\ \mid \_ & \Longrightarrow [B] \end{array}$$

4. Create the two sub problems $[A]$ and $[B]$ as follows by iterating over all the clauses. One of three cases can happen:
   (a) The clause contains a test a **is** $\mathsf{C}(\mathscr{P}_1, \ldots, \mathscr{P}_n), \ldots \mathrm{REST} \ldots$ for a.
   Add the expanded clause a_1 **is** $\mathscr{P}_1, \ldots, a_n$ **is** $\mathscr{P}_n, \ldots \mathrm{REST} \ldots$ to A.
   Make sure that the fresh variable names $a_1, \ldots, a_n$ are used consistently in **match** a **with** $\mathsf{C}(a_1, \ldots, a_n)$ and in the tests a_1 **is** $\mathscr{P}_1, \ldots, a_n$ **is** $\mathscr{P}_n$.
   (b) The clause contains a test a **is** $\mathsf{D}(\mathscr{P}_1, \ldots, \mathscr{P}_n), \ldots \mathrm{REST} \ldots$ where $\mathsf{D} \neq \mathsf{C}$.
   Add this clause to B unchanged.
   (c) The clause contains no test for a. Add this clause to both A and B.
   (note that each clause can only have one test for a)
5. Recursively generate code for $[A]$ and $[B]$.

The recursion has two base cases:

- The list of clauses is empty: all patterns failed, so generate an error: "Error: Non-exhaustive pattern match."
- The first clause is empty (has zero tests): the first clause succeeded to match, so simply return the right hand side e_i.

4 Heuristic

How do we decide which test to pick from the first clause to branch on? Any test works, but we'd like to generate a compact pattern matching tree. Whenever we are in case (c) for one of the other clauses, we have to add that clause to both A and B. That can cause code explosion. To avoid it we want to select the test that causes the least code explosion. We therefore select a test that is present in the maximum number of other clauses.

Consider this example:

$$
\begin{array}{ll}
\textbf{match } a \textbf{ with} & \\
\mid \mathsf{Add}(\mathsf{Add}(x,y),\mathsf{Zero}) & \Longrightarrow e_1 \\
\mid \mathsf{Add}(\mathsf{Mul}(x,y),\mathsf{Zero}) & \Longrightarrow e_2 \\
\mid \mathsf{Add}(x,\mathsf{Mul}(y,z)) & \Longrightarrow e_3 \\
\mid \mathsf{Add}(x,\mathsf{Add}(y,z)) & \Longrightarrow e_4 \\
\mid \mathsf{Add}(x,\mathsf{Zero}) & \Longrightarrow e_5
\end{array}
$$

For the outer constructor we have no choice, but for the inner constructors we can either test $\mathsf{Add}(x,y)$ first or Zero first. If we test $\mathsf{Add}(x,y)$ first, then we end up duplicating the e_3 and e_4 clauses. Our heuristic chooses to match on Zero instead, which leads to optimal code without any duplication:

$$
\begin{array}{l}
\textbf{match\#} \ a \ \textbf{with} \\
\mid \mathsf{Add}(a_1,a_2) \Longrightarrow \\
\quad \textbf{match\#} \ a_2 \ \textbf{with} \\
\quad \mid \mathsf{Zero} \Longrightarrow \\
\quad\quad \textbf{match\#} \ a_1 \ \textbf{with} \\
\quad\quad \mid \mathsf{Add}(x,y) \Longrightarrow e_1 \\
\quad\quad \mid \mathsf{Mul}(x,y) \Longrightarrow e_2 \\
\quad\quad \mid \_ \Longrightarrow e_5 \\
\quad \mid \mathsf{Mul}(y,z) \Longrightarrow e_3 \\
\quad \mid \mathsf{Add}(y,z) \Longrightarrow e_4 \\
\quad \mid \_ \Longrightarrow \mathsf{ERR} \\
\mid \_ \Longrightarrow \mathsf{ERR}
\end{array}
$$

(I've merged subsequent tests of the same variable into a single match – see below)

5 Discussion

This algorithm is a bit different than the algorithms in the literature. It is based on the following observations and considerations.

Some of the literature spends effort on avoiding exponential code explosion, and opts to generate redundant tests instead [Aug85]. Exponential code explosion doesn't occur in practice [SR00]. Therefore the right approach seems to me to never generate redundant tests, and to try to avoid code duplication using a heuristic as e.g. [Mar08] does.

In fact, the literature shows that for real world code, different pattern matching algorithms generate almost identical code [SR00, Mar08]. Our approach here is to (1) always work on matching the first clause first, to avoid unnecessary tests and (2) greedily try to minimize duplication using the heuristic. This isn't exactly what is in the literature, but it is similar to the heuristics in [Mar08].

Generating binary tests instead of n-way tests that simultaneously branch on several possible constructors may be necessary to avoid duplication:

$$
\begin{array}{ll}
\textbf{match } (a,b) \textbf{ with} & \\
\mid (A,\_) & \Longrightarrow e_1 \\
\mid (\_,A) & \Longrightarrow e_2 \\
\mid (\_,B) & \Longrightarrow e_3 \\
\mid (\_,C) & \Longrightarrow e_4 \\
\mid (\_,D) & \Longrightarrow e_5 \\
\mid (B,E) & \Longrightarrow e_6 \\
\mid (C,F) & \Longrightarrow e_7
\end{array}
$$

If we do a 3-way branch on $a = A, B, C$, then we have to duplicate the middle clauses for both B and C. On the other hand, if we simply test if $a = A$, and then test $b = A, B, C, D, E, F$, then we have no duplication.

Executing an n-way branch may be compiled more efficiently using a jump table than a series of 2-way branches. It is not difficult to detect a series of 2-way branches on the same variable, and convert those back into an n-way branch.

Some of the literature uses decision DAGs instead of decision trees. The easiest way to generate a decision DAG is simply to generate a decision tree and then compress it to a DAG using hash consing [Mar08]. A simpler though less optimal solution is to only do this for the right hand sides, and not for the internal nodes of the decision tree: we could generate a separate basic block for each unique right hand side, and replace the right hand sides by a jump to the corresponding basic block.

To take advantage of types, you can keep track of the remaining possible constructors of each variable. This way you can avoid generating the error cases when there are no remaining possible constructors. You can then do exhaustiveness checking by looking if an error case was generated or not. You can check for redundant patterns by looking at whether one of the orignal right hand sides e_i doesn't appear in the decision tree.

6 Code

Scala code that implements the algorithm can be found at
https://julesjacobs.com/notes/patternmatching/pmatch.sc

Yorick Peterse implemented a much more advanced version in Rust. His version supports constructors, integer literals and ranges, or-patterns, guards, and exhaustiveness and redundancy checking. The repository also contains a good README explaining the algorithm and how to implement the advanced features. The syntax "a **is** *pattern*" was invented by Yorick. I initially used "$a = pattern$", but I think Yorick's syntax is clearer.

References

[Aug85] Lennart Augustsson. Compiling pattern matching. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 368–381, Berlin, Heidelberg, January 1985. Springer-Verlag.

[Mar92] Luc Maranget. Compiling lazy pattern matching. In *Proceedings of the 1992 ACM conference on LISP and functional programming*, LFP '92, pages 21–31, New York, NY, USA, January 1992. Association for Computing Machinery.

[Mar08] Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, ML '08, pages 35–46, New York, NY, USA, September 2008. Association for Computing Machinery.

[Pet92] Mikael Pettersson. A Term Pattern-Match Compiler Inspired by Finite Automata Theory. In *Proceedings of the 4th International Conference on Compiler Construction*, CC '92, pages 258–270, Berlin, Heidelberg, October 1992. Springer-Verlag.

[Ses96] Peter Sestoft. ML pattern match compilation and partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, pages 446–464, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[SR00] Kevin Scott and Norman Ramsey. When Do Match-compilation Heuristics Matter? Technical Report, University of Virginia, USA, 2000.