# A Term Pattern-Match Compiler Inspired by Finite Automata Theory†

*Mikael Pettersson (mpe@ida.liu.se)*

Department of Computer Science, Linköping University, Sweden

*ABSTRACT*

This paper presents a new algorithm for compiling term pattern-matching for functional languages. Earlier algorithms may produce duplicated code, and redundant or sub-optimal discrimination tests for certain combinations of patterns, in particular when a pattern column contains a mixture of constructors and variables. This algorithm, which was inspired by finite automata theory, addresses these problems and solves them to some extent. It does so by viewing patterns as regular expressions and optimizing the finite automaton that is built to recognize them. It also makes checking patterns for *exhaustiveness* and *irredundancy* cheap operations.

## 1. Introduction

Term pattern-matching is a key feature of modern (mostly-) functional programming languages such as Standard ML [HMT90] and Haskell [Hudak *et al* 91]. Pattern-matching is used both to determine the shape of objects, and to bind variables to parts of them. Standard algorithms exist for compiling pattern-matching to simpler constructs [Augustsson85,Wadler87]. They produce good code for most cases, but have problems with certain combinations of patterns (when they must fall back to the so-called ‘‘mixture rule’’).

The algorithm presented in this paper was inspired by finite automata theory. A pattern is now viewed as a regular expression over atomic values, constructor names and wildcards. The pattern-match compiler takes a sequence of patterns and compiles them to an acyclic deterministic finite automaton. Equivalent states are merged, and the automaton is then transformed to an expression in the compiler's intermediate language. This approach has some interesting consequences:

- Since equivalent states are merged, the problem of duplication of the right-hand side expressions is avoided.
- Since finite automata encode all information in their states, they never need to backtrack or inspect a particular value more than once.

The rest of this paper is organized as follows: Section 2 discusses some problems with the standard algorithms. Section 3 defines the new algorithm, and Section 4 illustrates it on two examples. Section 5 discusses some implementation issues, and Section 6 compares this with related

† This paper was presented at the 1992 International Workshop on Compiler Construction (CC'92). The proceedings are published in Springer-Verlag LNCS-641.

algorithms. Section 7 concludes with some comparisons between this algorithm and an earlier implementation of the standard one.

## 2. Problems with earlier algorithms

The well-known pattern-match compilation algorithms in [Augustsson85] and [Wadler87] (henceforth referred to as the "standard" algorithms since they are very similar) work well in many cases. For certain inputs, however, they may produce poor code. This happens when the selected pattern column contains a mixture of variable and non-variable patterns. In this case the column is "chopped up" into alternating sub-sequences of constructors and variables. The problem with this is that when a discrimination test is done on a sub-sequence, the original context is lost, and this may lead to redundant or sub-optimal tests. This section presents some examples that illuminate these weak spots. All examples are written in Standard ML, as is the generated code.

### 2.1. Copied Expressions

Consider the following function definition (from [Wadler87]):

```
fun unwieldy [] [] = A
  | unwieldy xs ys = B xs ys
```

A naïve implementation of the standard algorithms would generate:

```
fun unwieldy xs ys =
  case xs
    of nil      => (case ys
                       of nil    => A
                        | _      => B xs ys)
     | _        => B xs ys
```

Since there are two ways for the second equation to match, the second expression `B xs ys` appears two times in the generated code.

To eliminate expression copying, Augustsson proposes a new control operator `DEFAULT`. If it appears in the right-hand side of some `CASE` expression, then control is transferred to the default entry of the nearest enclosing `CASE`[1].

```
fun unwieldy xs ys =
  case xs
    of nil      => (case ys
                       of nil    => A
                        | _      => DEFAULT)
     | _        => B xs ys
```

This has the drawback of being far too low-level for a compiler using a goto-less high-level language as its intermediate code. A much cleaner solution, and no less efficient if the compiler does basic analysis and integration of local procedures, is to wrap up shared expressions as local

[1] Wadler uses `[]` and `FAIL` instead, but the effect is the same.

procedures. The above example could then become:

```
fun unwieldy xs ys =
  let fun default() = B xs ys
  in
    case xs
      of nil    => (case ys
                      of nil    => A
                       | _      => default())
       | _      => default()
  end
```

Most modern compilers for Scheme, Standard ML and Common Lisp are able to compile the tail-calls to `default` as simple branches [Steele78, Krantz *et al* 86, AJ89].

### 2.2. Repeated and Sub-optimal Tests

Consider the following definition (again adapted from [Wadler87]):

```
fun demo [] ys = A ys
  | demo xs [] = B xs
  | demo (x'::xs') (y'::ys') = C x' xs' y' ys'

fun demo xs ys =
  case xs
    of nil        => A ys
     | _          =>                      (* PE(1) *)
         case ys
           of nil => B xs
            | _   =>                      (* PE(2) *)
                case xs                   (* <--(1) *)
                  of (x'::xs')    =>
                       (case ys           (* <--(2) *)
                          of (y'::ys')    => C x' xs' y' ys'
                           | _            => ERROR)
                   | _    => ERROR
```

The problem is that the repeated tests (marked with arrows above) must check for all constructors of the type, even though it is easily seen that only some actually can appear. The nested tests need to test for all constructors *only* if control could come to the default case via some use of the `DEFAULT` operator. The pattern-match compiler should optimize nested tests when this is not the case.

A referee commented on an earlier version of this paper that this sometimes is solved by *partial evaluation*. The pattern-match compiler would do a post-optimization pass on the generated expression. At every simple match *pat*⇒*exp*, it would optimize *exp* with the knowledge that the matched variable was *pat*. For instance, at the line marked `PE(1)`, it would first note that `xs` must be a pair, and then be able to simplify the repeated test marked `<--(1)`. The point of the algorithm presented in this paper is that it achieves this optimization directly, without the need for

an extra pass.

In the important special case where the repeated pattern only tests for a single constructor, as in the example above, the `CASE` expression should be eliminated and replaced by a `LET` to directly bind the variables to the components of the value.

## 3. Term Pattern-Matching and Finite Automata

The basic intuition behind this algorithm is that patterns can be viewed as alternation- and repetition-free regular expressions consisting of constructor names (possibly with sub-patterns), atomic values and wildcards. These regular expressions are easily mapped to tree-based deterministic finite automata. This in turn implies that backtracking never is needed: all the necessary information is encoded in the states of the automata. These automata are essentially *n*-ary trees, with arcs labeled by constructors, atomic values or wildcards, and non-leaf nodes corresponding to particular positions in the original patterns. Two internal nodes are equivalent if their sets of outgoing arcs are the same, and they correspond to the same position in the patterns. Therefore, the trees can easily be optimized by merging equivalent states in a single bottom-up traversal: this results in DAG-shaped automata that define the control flow of the generated pattern-matching code.

Using the analogy of finite automata makes it possible to directly describe an algorithm that neither needs backtracking nor generates duplicate code. Viewing patterns as regular expressions also makes it quite easy to define the mapping from patterns to states and arcs in the automata.

### The Algorithm

The algorithm is divided into four logical steps: *renaming of patterns*, *generating the DFA*, *merging of equivalent states*, and *mapping to intermediate code*. Each of these steps will now be described in detail.

The input to the algorithm is a sequence of *match rules* (*row of n patterns* ⇒ *expression*), and a sequence of *n root variables*. The order of the match rules is significant: the first (or top-most) rule that matches is the one whose expression will be evaluated.

The intermediate data structure of the algorithm is an acyclic deterministic finite automaton. Each state in the automaton is either *final* or a *test* state. There is one final state for each expression in the right-hand sides of the match rules, and one for matching failure. A test state corresponds to a simple case-expression that performs a discrimination test on a certain variable, and is characterized by that variable and a sequence of arcs: (simple pattern, new state)[2]. Each state also has some additional attributes:

- a unique *stamp* for identification purposes
- a set of *free variables* (defined and used below in step 4)
- a *reference count* indicating the number of direct references to the state

[2] A simple pattern is either an atomic value, a wildcard, or a constructor applied only to variables or wildcards.

**Step 1: Renaming**

The first step is to augment each pattern with its corresponding *path variable*. In order for the state merging process (step 3) to work, it must be guaranteed that two different test states that test the same part of the input object use the same variable name. For each match rule, the pattern row and the expression are subject to the following preprocessing:

Each (sub-) pattern is replaced by an internal pattern *path=pattern*, where the path is defined as follows:

- The path of each top-level pattern in the pattern row is the name of the corresponding root variable (the one with the same index).
- If a tuple pattern $(pat_1, \ldots, pat_n)$ has path `p`, then each $pat_i$ will have path `p$`$i$.
- If a constructor application *ctor pat* has path `p`, then *pat* has path `p$2` (`p$1` is the path of the tag, if there is one).

Since every pattern comes with a variable, there is no need for variable patterns in the internal form: wildcards are used instead. The pattern preprocessing also results in a renaming substitution mapping original variables in the patterns to their corresponding path variables. This substitution is first applied to the expression in the right-hand side, then the expression is replaced by a new final state containing the renamed expression. The reference count of this state is initially 0.

The result is the pair (*row of internal patterns, final state*).

*Example:* The input is the variable `x` and the match rules:

```
nil                     => E1
cons(y,ys)              => E2
```

The renaming step results in the following structures:

```
x=nil                   q1=E1
x=cons(x$2$1=_,x$2$2=_) q2=E2[x$2$1/y,x$2$2/ys]
```

where `q1` and `q2` are two new final states.

**Step 2: Generating the DFA**

The second step is to map the sequence of internal patterns and final states to a deterministic finite automaton, using an algorithm *match*. *Match* takes as input a matrix of patterns, and a column of final states (one for each row in the matrix). It returns the start state of the constructed automaton. Given the pre-processed match rules, the initial matrix is defined by the pattern rows, and the states by the newly constructed final states. *Match* proceeds by doing case analysis on the pattern matrix, and choosing some column and one of the two cases in the algorithm.

**The Variable Rule**

```
match   {v1=_,v2=_,..,vn=_}     {q1}
        {..               }     {..}
```

The top-most row has only variable (non-constructor) patterns. Therefore, `q1` is the top-most final state to match the input, and the result is simply `q1` with its reference count field incremented by one.

**The Mixture Rule**

```
                {v=pat1, pats1.. }      {q1}
match           {..               }      {..}
                {v=patn, patsn.. }      {qn}
```

The is the real work-horse of the algorithm. There is some column whose top-most pattern is a constructor. (For simplicity, it is depicted above as the left-most column, but any column will do.) The goal is to build a test state with the variable `v` and some outgoing arcs (one for each constructor and possibly a default arc). For each constructor `c` in the selected column, its arc is defined as follows:

> Let $\{i_1, \ldots, i_j\}$ be the row-indices of the patterns in the column that match `c`. Since the patterns are viewed as regular expressions, this will be the indices of the patterns that either have the same constructor `c`, or are wildcards.
>
> Let $\{pat_1, \ldots, pat_j\}$ be the patterns in the column corresponding to the indices computed above, and let $n$ be the arity of the constructor `c`, i.e. the number of sub-patterns it has. For each $pat_i$, its $n$ sub-patterns are extracted; if $pat_i$ is a wildcard, $n$ wildcards are produced instead, each tagged with the right path variable. This results in a pattern matrix with $n$ columns and $j$ rows. This matrix is then appended to the result of selecting, from each column in the rest of the original matrix, those rows whose indices are in $\{i_1, \ldots, i_j\}$. Finally the indices are used to select the corresponding final states that go with these rows. Note that the order of the indices is significant; selected rows do *not* change their relative orders.
>
> The arc for the constructor `c` is now defined as `(c',state)`, where `c'` is `c` with any immediate sub-patterns replaced by their path variables (thus `c'` is a simple pattern), and `state` is the result of recursively applying *match* to the new matrix and the new sequence of final states.

Finally, the possibility for matching failure is considered. If the set of constructors is exhaustive, then no more arcs are computed. Otherwise, a default arc `(_,state)` is the last arc. If there are any wildcard patterns in the selected column, then their rows are selected from the rest of the matrix and the final states, and the state is the result of applying *match* to the new matrix and states. Otherwise, the error state is used after its reference count has been incremented.

**Step 3: Optimizing the DFA**

The third step is to merge equivalent states. Since the DFAs are acyclic, this process can be efficiently implemented in a bottom-up fashion. Two final states are equivalent if they have the same stamp. Two test states are equivalent if they test the same variable (this is the reason for the renaming step), and their sets of outgoing arcs are equal. Two arcs are equal if their simple patterns are equal and the target states have the same stamps.

Here it has been described as a separate step, but this can easily be integrated into the *match* algorithm. Whenever *match* is about to create a new test state, it first checks whether an equivalent

one already exists. If so, the reference count field of the old state is incremented, and the old state is returned. Otherwise a new state with reference count 1 is created and returned.

**Step 4: Generating Intermediate Code**

The fourth and final step is to map the automaton to an expression in the intermediate language. Since shared states are to be implemented as local procedures, the notion of the *free variables* of a state becomes important:

- The free variables of a final state are the path variables from the substitution created in step 1 when renaming the expression of this state. The error state has no free variables.
- The free variables of an arc (*simple pattern, new state*) are the free variables of the state, minus the variables bound in the pattern.
- The free variables of a test state is the union of the free variables of the outgoing arcs, plus the test variable.

As for step 3, the free variable computation has been described as a separate step. Again, this is not necessary since the free-variable field can be computed once and for all when the state initially is created.

Starting with the start state, the intermediate language expression is constructed as follows:

- The translation of a final state is the expression it contains.
- The translation of a test state is a simple case-expression, whose variable is the variable of the state, and whose sequence of match rules is the result of translating all the outgoing arcs.
- The translation of an arc (*pattern, state*) is the match rule *pattern*⇒*expression*, where the expression is the translation of the state reference.
- A reference to a non-shared state is replaced by the translation of the state itself.
- A reference to a shared state, i.e. one whose reference count field is greater than one, is made into a call to a local procedure. The name of this procedure is the stamp of the state, and the arguments are the free variables of the state (in some canonical order). The body of the procedure is the translation of the state itself.

The resulting intermediate expression may refer to procedures corresponding to shared states. In this case, the expression is wrapped in a `LETREC` with bindings for each such procedure.

## 4. Two Examples

### 4.1. The DEMO Function

```
fun demo [] ys = E1                    where E1=A ys
  | demo xs [] = E2                    where E2=B xs
  | demo (x'::xs') (y'::ys') = E3      where E3=C x' xs' y' ys'
```

**Step 1: Renaming**

Here the names `xs` and `ys` will be used as the root variables. The result of the renaming is:

```
xs=nil, ys=_                                                  q1=E1
xs=_, ys=nil                                                  q2=E2
xs=cons(xs$2$1=_,xs$2$2=_), ys=cons(ys$2$1=_,ys$2$2=_)  q3=E3'
```

where `E3'=E3[xs$2$1/x',xs$2$2/xs',ys$2$1/y',ys$2$2/ys']`.

**Step 2: Generating the DFA**

```
        {xs=nil,ys=_                                                  } {q1}
match0: {xs=_,ys=nil                                                  } {q2}
        {xs=cons(xs$2$1=_,xs$2$2=_),ys=cons(ys$2$1=_,ys$2$2=_)} {q3}
```

The mixture rule is applicable in the first column. The constructor `nil` matches rows 1 and 2, and `cons` matches rows 2 and 3. The constructors are exhaustive, thus no default case.

```
q0:     case xs
          of nil                      => match1
           | cons(xs$2$1=_,xs$2$2=_) => match2
```

where:

```
match1: {ys=_  } {q1}
        {ys=nil} {q2}

match2: {xs$2$1=_,xs$2$2=_,ys=nil                     } {q2}
        {xs$2$1=_,xs$2$2=_,ys=cons(ys$2$1=_,ys$2$2=_)} {q3}
```

(notice how the `xs` in row 2 was split into two new wildcard patterns to make it compatible with the two new sub-patterns from the first `cons`)

Considering `match1`, this is a case for the variable rule. It immediately reduces to `q1`.

Considering `match2`, this is a case for the mixture rule in column 3 (the `ys` variable). The `nil` constructor only matches row 1, and `cons` row 2. Again, no default case.

```
q4:     case ys
          of nil                  => match3
           | cons(ys$2$1,ys$2$2) => match4
```

where:

```
match3: {xs$2$1=_,xs$2$2=_} {q2}

match4: {ys$2$1=_,ys$2$2=_,xs$2$1=_,xs$2$2=_} {q3}
```

For both of the new matches, the variable rule is applicable, and they reduce to their respective final states.

The final automaton is:

```
q0:     case xs
          of nil                  => q1
           | cons(xs$2$1,xs$2$2) => q4

q4:     case ys
          of nil                  => q2
           | cons(ys$2$1,ys$2$2) => q3
```

with the final states `q1`, `q2` and `q3` defined as in the beginning.

**Step 3: Optimization**

In this example, no two states are equivalent.

**Step 4: Intermediate Code**

```
fun demo xs ys =
  case xs
    of nil                        => A ys
     | cons(xs$2$1,xs$2$2)        =>
         case ys
           of nil                 => B xs
            | cons(ys$2$1,ys$2$2) => C xs$2$1 xs$2$2 ys$2$1 ys$2$2
```

### 4.2. The UNWIELDY Function

```
fun unwieldy [] [] = A
  | unwieldy xs ys = B xs ys
```

**Step 1: Renaming**

Here the names `xs` and `ys` will be used as the root variables. The result of the renaming is:

```
xs=nil, ys=nil                          q1=A
xs=_, ys=_                              q2=B xs ys
```

**Step 2: Generating the DFA**

```
match0: {xs=nil, ys=nil} {q1}
        {xs=_,   ys=_  } {q2}
```

The mixture rule is applicable in the first column. The constructor `nil` matches rows 1 and 2. `nil` is not exhaustive, so there will be a default arc for the wildcard rows, row 2 here.

```
q0:     case xs
          of nil        => match1
           | _          => match2
```

where:

```
match1: {ys=nil} {q1}
        {ys=_  } {q2}

match2: {ys=_  } {q2}
```

Considering `match2` first, this is a case for the variable rule, so it immediately reduces to `q2`.

Considering `match1`, this is a case for the mixture rule in column 1. The `nil` constructor matches rows 1 and 2, but is not exhaustive, so there will be a default arc for the wildcard row 2.

```
q3:     case ys
          of nil        => match3
           | _          => match4
```

where:

```
match3: {} {q1}
        {} {q2}

match4: {} {q2}
```

The variable rule is applicable in both cases, so `match3` reduces to `q1` and `match4` to `q2`.

The final automaton is:

```
q0:     case xs
          of nil        => q3
           | _          => q2

q3:     case ys
          of nil        => q1
           | _          => q2
```

with the final states `q1` and `q2` defined as in the beginning.

**Step 3: Optimization**

The two references to `q2` are merged by incrementing `q2`'s reference count.

**Step 4: Intermediate Code**

State `q2` has a reference count greater than one, so it is made into a local procedure. It has no free variables since no substitution was made in step 1.

```
fun unwieldy xs ys =
  let fun q2() = B xs ys
  in
    case xs
      of nil            =>
          (case ys
             of nil     => A
              | _       => q2())
       | _              => q2()
  end
```

## 5. Implementation Notes

### 5.1. Data Representation

The algorithm as described compiles complex pattern-matching to simple patterns, i.e. patterns with no nested constructors. A really good implementation should however go one step further and also consider low-level representation issues before emitting the final intermediate code. In particular, the application of a `DATATYPE` constructor can be represented in several different ways depending on the rest of its `DATATYPE` declaration. See [Appel90] and [Cardelli84] for further information about representation choices. It should be noted though that such representation optimizations are often limited to statically-typed and strict languages like Standard ML. Lazy languages implemented by graph reduction often choose uniform representations in order to speed up the reduction process (see e.g. [Peyton Jones 87]).

A special case deserves to be mentioned here. Simple `CASE`-expressions whose patterns are the constructors of a `DATATYPE`, can often execute in O(1) time by the use of jump tables. The implementation of simple `CASE`-expressions has been studied in the context of ordinary imperative languages, see e.g. the excellent summary in [Bernstein85].

### 5.2. Compile-time Warnings

The definition of Standard ML requires that the compiler give warnings if a pattern-match is *non-exhaustive* or *redundant*. These conditions are easily checked by inspecting the reference counts of the final states in the automaton. If the failure state has a non-zero reference count, then the match can fail, and the non-exhaustiveness message is triggered. If any final state corresponding to a right-hand side expression has a zero reference count, then that expression can never be evaluated, and the patterns must contain redundant equations.

## 6. Comparisons with Related Algorithms

There are both interesting similarities and differences between this and the standard algorithm. Both operate on a matrix of patterns, repeatedly choosing a column on which to perform a discrimination test, and continuing with several smaller matches. The primary difference, apart from this algorithms use of an intermediate data structure, appears to be the treatment of matching failure. The standard algorithm uses a third argument to *match*, which acts as a "failure continuation". In the sub-matches generated by the mixture rule, the fail expression is defined by the

continued matching on the bottom part of the pattern matrix (from the first variable and downwards) (or `DEFAULT` if backtracking is used). This is why the same part of the input can be inspected multiple times. In contrast, the algorithm described in this paper always considers *all* the rows that might match at the same time. It is possible that this may cause an explosion in the number of states of the DFA (analogously to the worst cases when nondeterministic finite automata are converted to deterministic ones), although this has not been observed in practise.

It should be emphasised that this algorithm only is applicable to strict languages, or lazy languages with strict top-down, left-right evaluation in pattern-matching constructs. There are some approaches that try to deal with termination problems in lazy pattern-matching, see e.g. [PS90].

## 7. Experiences and Conclusions

The algorithm presented here, with refinements for exploiting the actual representations of various datatypes, has been implemented twice. First as a prototype in Standard ML, used to debug and verify the ideas and algorithms. A second implementation was done in Scheme to replace an earlier one used in the author's macro package for SML-like datatypes and pattern-matching in Scheme. This package had been used extensively in the implementation of the author's SML-in-Scheme system (roughly 6.500 lines of code).

To this author's surprise, the use of the new algorithm did not lead to a reduction in code size. For a total of 231 pattern-matching constructs, 1671 states were generated, of which only 38 (2.3%) were shared. Only two out of the 231 automatons had two shared states; none had more. It was conjectured therefore that programmers tend to write rather simple pattern-matching constructs, and avoid complex ones. Whether this is because they do not trust their compilers or whether the troublesome constructs rarely are needed, is an open question.

The primary advantages of the new algorithm's use of the finite automata analogy appears therefore to be the ease by which someone can follow the algorithms, and in the low complexity of the implementation. Speaking from practical experience, having implemented three generations of pattern-match compilers, this author definitely feels that the algorithm described here was the easiest one to implement, and the one that required the least amount of debugging.

## 8. Acknowledgements

The primary inspiration for using finite automata to implement term pattern-matching came from Hoffman and O'Donnell's paper on *tree* pattern-matching [HO82]. The problem of tree pattern-matching is to find *all* the places in an input tree where some pattern matches. Its primary application appears to be in tree-rewriting systems, such as code generators automatically built from template-rewrite specifications [AGT89].

## References

[AGT89] A. V. Aho, M. Ganapathi, S. W. K. Tjiang, *Code Generation Using Tree Matching and Dynamic Programming* (ACM TOPLAS, Vol. 11(4), October 1989).

[AJ89] Andrew W. Appel and Trevor Jim, *Continuation-Passing, Closure-Passing Style* (Proceedings 1989 ACM Symposium on Principles of Programming

Languages).

[Appel90] Andrew W. Appel, *A Runtime System* (Lisp and Symbolic Computation, Vol. 3, 343-380, 1990. Also in the "New Jersey" SML distribution.)

[Augustsson85] Lennart Augustsson, *Compiling Pattern-Matching* (Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture, Springer LNCS-201).

[Bernstein85] Robert L. Bernstein, *Producing Good Code for the Case Statement* (Software — Practice and Experience, Vol. 15(10), October 1985).

[Cardelli84] Luca Cardelli, *Compiling a Functional Language* (Proceedings 1984 ACM Conference on Lisp and Functional Programming).

[HO82] Cristoph M. Hoffman and Michael J. O'Donnell, *Pattern Matching in Trees* (Journal of the ACM, Vol. 29(1), January 1982).

[HMT90] Robert Harper, Robin Milner and Mads Tofte, *The Definition of Standard ML* (The MIT Press).

[Hudak *et al* 91] Paul Hudak *et al*, *Report on the Programming Language Haskell, Version 1.1* (June, 1991).

[Kranz *et al* 86] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams, *ORBIT: An optimizing compiler for Scheme* (Proceedings SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices, Vol. 21(7), July 1986).

[Peyton Jones 87] Simon L. Peyton Jones, *The Implementation of Functional Programming Languages* (Prentice-Hall, 1987).

[PS90] L. Puel and A. Suárez, *Compiling Pattern Matching by Term Decomposition* (Proceedings of the 1990 ACM Conference on Lisp and Functional Programming).

[Steele78] Guy L. Steele Jr., *Rabbit: a compiler for Scheme* (AI-TR-474, MIT, 1978).

[Wadler87] Philip Wadler, *Efficient Compilation of Pattern-Matching* (Chapter 5 of [Peyton Jones 87]).