RESEARCH ON PROGRAM ANALYSIS SYSTEM
NATIONAL CREATIVE RESEARCH INITIATIVE CENTER
ROPAS KOREA ADVANCED INSTITUTE OF SCIENCE AND TECHNOLOGY

ROPAS MEMO
2001-12
June 28, 2001

# Compile Time Evaluation of Pattern Matching

(Rediscovering P. Thiemann's Higher-Order Redundancy Elimination)*

László Németh
laszlo@ropas.kaist.ac.kr

June 28, 2001

**Abstract**

There has been a great number of proposals for the optimal compilation of pattern matching both in the case of lazy and strict languages. Some of these proposals are optimal (in the sense that they perform a minimum number of matches to select the appropriate alternative) with respect to the body of the function being compiled, but do not take into account if the function is called with a known pattern.

We define an analysis and a transformation which helps to avoid the redundant tests which arise from known call patterns, and compare it to earlier work by Thiemann [8].

Gains from the transformation are quantified on a single benchmark, which although demonstrates the usefulness does not show percievable improvement.

## 1 Introduction

Pattern matching is a prominent and useful syntactic sugar in most functional languages. Accordingly, its compilation into an efficient sequence of nested `case` statements has received considerable attention both for lazy and strict languages [1, 10, 6]. Efficiency is usually measured in the number of simple tests to choose the appropriate alternative. Additionally further criteria — like not to reexamine values – are also included [4].

For lazy languages the problem of compiling a complex pattern into a sequence of simple tests is theoretically simpler than for strict languages, because lazy languages usually impose a top-down, left-to-right evaluation order on patterns to avoid complications with strictness. Strict languages do not suffer from the left-ro-right bias, which gives more freedom for the compiler. This extra freedom makes the optimal compilation of pattern matching NP complete.

The task of the match compiler is to compile complex pattern matches (in absyn[1]) into a nested sequence of simple tests (in lambda code). Its input is a list of pairs of abstract syntax patterns and lambda expressions and it produces a lambda expression. This lambda expression is constructed to implement a decision tree. This tree has `case` expressions as its nodes, and the edges are the possible outcome of the test.

*This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology.

[1]From now on we focus on pattern matching in the context of SML/NJ and freely refer to compiler details.

```
(* code from a Knuth-Bendix completion procedure *)
fun occurs n (Var m)        = (m=n)
  | occurs n (Term(_,sons)) = exists (occurs n) sons

fun rpo op_order ext =
  let fun rporec M N =
    if M=N then Equal
    else case M of
           Var m           => NotGE
         | Term(op1,sons1) => case N of
(*!*)                            Var n           => if occurs n M then Greater else NotGE
                               | Term(op2,sons2)
                               => case (op_order op1 op2) of
(*!!*)                              Greater => if for_all (fn N' => case rporec M N' of
                                                                   Greater => true
                                                                 | _       => false) sons2
                                               then Greater else NotGE
(* CFA would help *)              | Equal   => ext rporec M N
(*!!*)                            | NotGE   => if exists (fn M' => case rporec M' N of
                                                                  NotGE => false
                                                                | _     => true) sons1
                                               then Greater else NotGE
  in rporec
  end
```

Figure 1: Naive code

To simplify the task of the match compiler, the chunk of code for the decision tree is generated solely by examining the patterns being compiled. A better alternative is to compile the pattern match taking into account information about call patterns, gathered by a global static analysis, of the given function. In terms of decision trees, this allows generating smaller chunks of code (multiple entry points) which provides alternative entry points to the decision 'forest' when parts of the pattern are known.

## 2 An example

Consider the program fragment given in Figure 1. In the line marked `(*!*)`, when the call to `occurs` takes place its second argument `M` has already been successfully matched against `Term(op1, sons1)`. In the body of `occurs`, `M` is matched again which is a minor source of inefficiency. More worrying are the lines marked `(*!!*)`, because already matched terms are passed to `rporec` itself, so the wasted matching is repeated at each level of recursion. If the terms `M` or `N` are large this leads to a considerable amount of repeated work.

It is possible to rewrite `occurs` (and `rporec`) manually, but it is tedious and error prone. Instead, we propose a fully automatic transformation, preceded by an appropriate analysis, which gives the more efficient code shown in Figure 2.

## 3 The analysis

This section defines my version of the analysis. It differs from Thiemann's [9, 8] in that he uses abstract interpretation so he can attempt to predict the outcome of functions.

```
fun occurs n (Var m)        = (m=n)
  | occurs n (Term(_,sons)) = occurs_1 n sons
and occurs_1 n sons         = exists (occurs n) sons

fun rpo op_order ext =
  let
    fun rporec_3 M N op2 sons2 = if M = N
                                 then Equal
                                 else case M of
                                        Var m             => NotGE
                                      | Term (op1, sons1) => rporec_2 M op1 sons1 N op2 sons2

    and rporec_2 M op1 sons1 N op2 sons2
      = case (op_order op1 op2) of
          Greater => if for_all (fn N' => case rporec_1 M op1 sons1 N' of
                                             Greater => true
                                           | _       => false) sons2
                     then Greater else NotGE
        | Equal   => ext rporec M N                                      (* CFA would help *)
        | NotGE   => if exists (fn M' => case rporec_3 M' N op2 sons2 of
                                            NotGE => false
                                          | _     => true) sons1
                     then Greater else NotGE

    and rporec_1 M op1 sons1 N =
        case N of
          Var n             => if occurs_1 n sons1 then Greater else NotGE
        | Term (op2, sons2) => rporec_2 M op1 sons1 N op2 sons2

    and rporec M N = if M=N then Equal else case M of
                                              Var m           => NotGE
                                            | Term(op1,sons1) => rporec_1 M op1 sons1 N
  in rporec
  end
```

Figure 2: Optimised code

| | | | |
|---|---|---|---|
| $prg$ | $\rightarrow$ | $dec^*$ | (program) |
| $dec$ | $\rightarrow$ | $\textbf{fun}\ f\ pat_1 = exp_1 \| \ldots \| pat_n = exp_n$ | (function declaration) |
| $exp$ | $\rightarrow$ | $v$ | (variable) |
| | $\|$ | $k$ | (constant) |
| | $\|$ | $exp\ (exp_1, \ldots, exp_n)$ | (application) |
| | $\|$ | $\textbf{fn}\ pat_1 = exp_1 \| \ldots \| pat_n = exp_n$ | (lambda abstraction) |
| | $\|$ | $\textbf{let}\ v = exp_1\ \textbf{in}\ exp_2$ | (let expression) |
| | $\|$ | $\textbf{case}\ v\ \textbf{of}\ pat_1 \Rightarrow exp_1 \| \ldots \| pat_n \Rightarrow exp_n$ | (case expressions) |
| $pat$ | $\rightarrow$ | $\_$ | (wildcard pattern) |
| | $\|$ | $v$ | (variable pattern) |
| | $\|$ | $c\ (pat_1, \ldots, pat_n)$ | (constructor pattern) |
| | $\|$ | $v\ \textbf{as}\ pat$ | (layered pattern) |

Figure 3: Syntax of the language

My criteria is simpler, I only check if the scrutinee is discriminating (defined below). So, for example, in the case of:

```
case (f ...) of
  true  => e1
| false => e2
```

if Thiemann's abstract interpretation can determine that `f...` always returns `true`, then in the body of `e1` `f...` would be bound to `true`, and consequently, any function calls in `e1` where the argument is `f...` would result in a known call pattern[2]. In my presentation predicting the outcome of functions is not attempted, therefore my analysis would never find the known call pattern.

First, we define the expressions we can use in the transformation phase.

**Definition 1** *Constants (character, integer, string, real), (fully applied) value constructors, records with discriminating fields, and variables which in the given scope are bound to a discriminating expression are* discriminating. *All other forms of expressions are non-discriminating.*

In the terminology of the Definition [3], *atomic expressions* with the exception of local declarations and exception constructors are discriminating. Discriminating expressions play a central role in the analysis.

## 3.1 Simple analysis

The analysis is defined over the abstract syntax of SML/NJ. Instead of using the constructors from the compiler source, I take the liberty of using the usual nota-

[2]The example is somewhat artificial, because if the abstract interpretation can determine that `f...` always evaluates to `true` then, of course, the entire `case` expression can be simplified to `e1`.

$$
\begin{array}{rcl}
\mathcal{CE} & : & \mathit{exp} \to \mathit{env} \to (\mathit{exp}, \mathit{exp}\ \mathit{list})\ \mathit{list} \\
\mathcal{CE}\llbracket f(e_1, \ldots, e_n) \rrbracket \rho & = & \bigcup_{i=1}^{n} \mathcal{CE}\llbracket e_i \rrbracket \rho \cup \begin{cases} \{(f, [e_1, \ldots, e_n])\} & \text{if any } e_i \text{ is discriminating} \\ \emptyset & \text{otherwise} \end{cases} \\
\mathcal{CE}\llbracket \textbf{fn}\ (p_1, e_1) | \ldots | (p_n, e_n) \rrbracket \rho & = & \bigcup_{i=1}^{n} \mathcal{CE}\llbracket e_i \rrbracket (\mathit{extend}\ \rho\ (\mathit{asPatsOf}\ p_i)) \\
\mathcal{CE}\llbracket \textbf{let}\ v = e_1\ \textbf{in}\ e_2 \rrbracket \rho & = & \mathcal{CE}\llbracket e_1 \rrbracket \rho \cup \mathcal{CE}\llbracket e_2 \rrbracket \begin{cases} \mathit{extend}\ \rho\ \{(v, e_1)\} & \text{if } e_1 \text{ is discriminating} \\ \rho & \text{otherwise} \end{cases} \\
\mathcal{CE}\llbracket \textbf{case}\ e\ \textbf{of} \begin{array}{ll} & p_1 \Rightarrow e_1 \\ | & \ldots \\ | & p_n \Rightarrow e_n \end{array} \rrbracket \rho & = & \mathcal{CE}\llbracket e \rrbracket \rho \cup \bigcup_{i=1}^{n} \mathcal{CE}\llbracket e_i \rrbracket \begin{cases} \mathit{extend}\ \rho\ (\mathit{asPatsOf}\ p_i \cup (v, e)) & \text{if } e \text{ is a variable} \\ \mathit{extend}\ \rho\ (\mathit{asPatsOf}\ p_i) & \text{otherwise} \end{cases} \\
\mathcal{CE}\llbracket \texttt{other} \rrbracket \rho & = & \begin{cases} \emptyset & \text{if } \texttt{other} \text{ is not recursive in } \mathit{exp} \\ \text{gather calls recursively} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 4: Analysis to find transformable calls

tion: $f(e_1, \ldots, e_n)$ instead of `APPexp (e1, e2)`. Using absyn is a relatively reasonable choice, because

1. AST is an even more verbose language, and it is before elaboration and type checking, so we don't know if we are transforming a type correct program. Also, fixity parsing takes place later, which would be a headache to handle.

2. lambda is too late as by the time we get to it, pattern matching has already been translated.

The analysis is given in Figure 4. Explanation: at a function application the call patterns of the subexpressions are collected and, if the call has any discriminating arguments, a new call pattern is constructed from the operator and the operands. In a `let` expression the call patterns are collected from the local expression, and the body is processed in an extended environment, if the local expression is discriminating. The right hand side of function expressions, `FNexp`'s, are processed in an environment which contain `as` patterns of its pattern. Using `as` patterns is the only way one can bind a variable to a discriminating expression[3]. `case` expressions are equaly simple: first the scrutinee is processed, then the right hand sides of the alternatives in an environment appropriately extended. Nonrecursive *exp*'s, like `INTexp, VARexp`, etc result in no information, while recursive occurences of *exp* are traversed and call patterns are collected.

One (easily fixed) deficiency of this analysis as compared to Thiemann's is that it only collects *discriminating* calls unlike Thiemann's which collects *all* call patterns. Because of this, the result of his analysis is more precise: it can tell if a given function is never called with completely unknown patterns, so his transformation phase can take more evasive actions. In practice, however I don't think this matters much. If a function is visible outside its defining module one has to keep the original definition anyway.

[3] `as` patterns behave like a binding construct, and the function `asPatsOf` takes care of this.

### 3.2 Control Flow Analysis

The analysis defined in Figure 4 is rather naive, and it has a little deficiency: it does not spot all the opportunities for the transformation. Consider the lines in Figure 1 and 2 with the comment about CFA. The current function is passed to another one, presumably only to be called from that one. By CFA, it is possible to determine that the argument `ext` to `rpo` is always bound to the same function, which in turn only calls the argument `rporec`. So we have 'implicit' mutual recursion, which the simple analysis given above cannot handle.

CFA can have dramatic effects on the efficiency of the transformation. Assume that CFA determined that in `ext`, `rporec` is applied to $M$ and $N$ (line with the comment about CFA). Instead of calling `rporec` we could call the specialised version `rporec_2`, and this in turn would allow us to drop $M$ and $N$ from the argument list. This way we would get all the benefits of the transformation without the drawbacks of arity raising.

## 4 The transformation

The transformation builds on the result of the analysis, $R$, which is a set of function, argument list pairs. To transform the function $f$ which has at least one discriminating call pattern:

1. Select all the rules from the definition of $f$ which matches (at least) one of the known patterns for $f$ in $R$.

2. Simplify the patterns of the selected rules by eliminating the discriminating pattern i.e. if there is a $(x :: xs)$ known pattern, replace it by two arguments $x$ and $xs$ (this is sometimes called *arity raising* [7]). A further improvement on this is to add only those variables which are free in the right-hand side, because this reduces the number of arguments passed.

3. Define $f'$ with the simplified patterns from step 2 and their right-hand sides.

4. Replace $f$ by $f'$, with appropriate arguments, at all the call-sites for this known pattern i.e. a call to $f \ldots (x :: xs)$ is replaced by $f' \ldots x\ xs$. If $f$ names $(x :: xs)$ with an `as` pattern, it might be necessary to pass $(x :: xs)$ to $f'$ as well.

This last step may seem to require an exta traversal of the program being transformed, but it can be done in one pass, since the transformation can be interleaved with replacing the known call sites with the new calls.

### 4.1 Diversion

It is interesting to note that after transformation the exhaustiveness of the patterns of the (possibly) mutually recursive set of functions is not possible to determine. Assume that $c1, c2, c3$ are the only constructors of a datatype, and the analysis determined that $f$ has two known call patterns with $c1$, and $c2$ or $c3$ i.e. it is known that in the second known pattern $c1$ can not occur.

```
fun f (c1 ...) = e1
  | f (c2 ...) = e2
  | f (c3 ...) = e3
```

If the right-hand sides of $f$ and $f'$ are mutually recursive we get:

```
fun f  (c1's args) = e1
and f' (c2 ...)    = e2
  | f' (c3 ...)    = e3
```

and the standard algorithm will report a non-exhaustive pattern for $f'$. In this case, the standard algorithm is easily amended to check exhaustiveness for a set of mutually recursive bindings. However, if the resulting bindings are not mutually recursive we may transform $f$ as:

```
fun f  (c1's args) = e1
fun f' (c2 ...)    = e2
  | f' (c3 ...)    = e3
```

If neither $f$ calls $f'$ nor $f'$ calls $f$ then there is no way of telling that the patterns for $f'$ are exhaustive. It may be desirable to avoid the erroneous warning, but it is unclear how to do so.

## 5 Evaluation

There is no complete implementation of the transformer, only the analysis. Hand rewritten programs show little improvement in runtimes. It also must be pointed out that running the program on the given set of rules (`Geom_rules`) takes less than a second, which makes it hardly a representative benchmark. The change from naive code (Figure 1 to Figure 2 in the case of `Geom_rules` of `knuth-bendix.sml` written by X. Leroy) to optimised code results in the following reductions in the number of pattern matches:

| | before | after | saving % |
|---|---|---|---|
| `occurs` | 4091 | 3995 | 3 |
| `rporec` | 2470 | 2177 | 12.1 |

The reduction in the case of `occurs` is uninteresting, but the 12.1% for `rporec` is not too bad. Unfortunately, there are many factors that can offset the improvement gained by the transformation:

- *arity raising* increases the number of arguments to functions which depending on the implementation can dramatically increase runtimes (only a fixed number of arguments can be passed in registers: if this limit is exceeded as a result of arity raising the slowdown is non-linear).

- writing *local recursive functions* (there seem to be an obsession to do this in the benchmark code I've looked at) limits the usefulness of the transformation. Consider the code in Figure 1, where the local recursive function `rporec` is only called with discriminating patterns from its own body. Assume that there were other calls from other functions with discriminating patterns. The appropriate

```
fun rpo op_order ext =
  let fun rporec_1 n M = if occurs n M then Greater else NotGE
      and rporec_2 M op1 sons1 N op2 sons2
          = case (op_order op1 op2) of
              Greater => if for_all (fn N' => case rporec M N' of
                                                 Greater => true
                                               | _       => false) sons2
                         then Greater else NotGE
            | Equal   => ext rporec M N
            | NotGE   => if exists (fn M' => case rporec M' N of
                                                NotGE => false
                                              | _     => true) sons1
                         then Greater else NotGE
      and rporec M N
          = if M=N then Equal
            else case M of
                   Var m          => NotGE
                 | Term(op1,sons1) => case N of
                                        Var n           => rporec_1 n M
                                      | Term(op2,sons2) => rporec_2 M op1 sons1 N op2 sons2
  in rporec
  end
```

Figure 5: The worker-wrapper split

wrappers of `rporec` should be inlined to these call sites, but it is not possible because `op_order` and `ext` are free in the body of `rporec`. So `rporec` should be lambda lifted before the transformation. But lambda lifting also increases the number of arguments passed around in the recursion, which in turn leads to a slowdown.

# 6 Comparison with the worker-wrapper technique

The motivation to use the worker-wrapper technique [5] instead of an analysis followed by a transformation is that sometimes, with a clever worker-wrapper split, the analysis can be avoided entirely. Consider the code given in Figure 1, and make a worker-wrapper split according to the following rule:

**Rule 1** *Define the wrapper of the function $f$ to be the inner-most* `case` *expression which scrutinises only explicit arguments to $f$, and the workers to be the alternatives of this* `case` *expression*[4].

Applying this rule gives us the code given in Figure 5: `rporec` is the wrapper and `rporec_1` and `rporec_2` are the workers. The `Var n` branch is an example where we do not define a separate worker as the constructor `NotGE` is inexpensive. The split on its own is not beneficial, but later transformations (see Figure 6) make the difference. In particular, since `rporec` is small, it can be inlined into the body of `rporec_1` and `rporec_2`. The simplification steps (only the `Greater` branch of `rporec_2`) are shown in Figure 6. So we end up with very much the same code as in Figure 2, because in both

[4] An improvement on this scheme is to distinguish 'small' expressions (computationally inexpensive like primitives, constants, non-recursive calls etc) from possibly expensive ones and define workers only for the possibly expensive alternatives.

```
  if for_all (fn N' => case rporec M N' of
                          Greater => true
                        | _       => false) sons2
  then Greater else NotGE
= { inlining rporec }
  if for_all (fn N' => case (if M=N then Equal
                              else case M of
                                     Var m           => NotGE
                                   | Term(op1,sons1)
                                   => case N of
                                        Var n           => rporec_1 n M
                                      | Term(op2,sons2) => rporec_2 M op1 sons1 N op2 sons2) M N' of
                          Greater => true
                        | _       => false) sons2
  then Greater else NotGE
= { two beta reductions }
  if for_all (fn N' => case (if M=N' then Equal
                              else case M of
                                     Var m           => NotGE
                                   | Term(op1,sons1)
                                   => case N' of
                                        Var n           => rporec_1 n M
                                      | Term(op2,sons2) => rporec_2 M op1 sons1 N' op2 sons2) of
                          Greater => true
                        | _       => false) sons2
  then Greater else NotGE
= { M = Term (...) simplify inner case }
  if for_all (fn N' => case (if M=N' then Equal
                              else case N' of
                                        Var n           => rporec_1 n M
                                      | Term(op2,sons2) => rporec_2 M op1 sons1 N' op2 sons2) of
                          Greater => true
                        | _       => false) sons2
  then Greater else NotGE
```

Figure 6: Simplification steps

branches of `rporec_2` the repeated tests on $M$, and $N$ respectively are avoided. The `NotGE` branch of `rporec_2` is completely analogous. Since `occurs` would be also put through this worker-wrapper split, its wrapper would also get inlined into its worker and into `rporec_1` which would give us the same code as in Figure 2.

## 7 Conclusions and Future Work

Intuitively, the analysis followed by the proposed transformation is worth incorporating into an optimising compiler, because if there are any known call patterns this will eliminate redundant matches. Unfortunately, this reduction in the number of matches does not translate into a reduction in runtime for the programs I have tried.

It is unclear how much redundancy there is in pattern matching. It appears to be possible, along the lines of [2], to define a formally optimal algorithm for pattern matching, which never examines an already matched value, but according to the benchmarks pattern matching as a whole takes up relatively small proportion of the overall runtime. Also, this algorithm would require major restructuring of code (splitting functions, arity raising, dropping arguments etc) for possibly little gain.

## References

[1] L. Augustsson. A pattern matching compiler. In H. Ganzinger and N. D. Jones, editors, *Proceedings of a Workshop on Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*. Springer-Verlag, Oct. 1986.

[2] F. Henglein and J. Jorgensen. Formally Optimal Boxing. In *Proceedings of 21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Portland, Oregon, Jan. 1994. ACM Press.

[3] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1991.

[4] M. Pettersson. A term pattern-match compiler inspired by finite automata theory. *Lecture Notes in Computer Science*, 641:258–??, 1992.

[5] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. *Lecture Notes in Computer Science*, 523, 1991.

[6] L. Puel and A. Suárez. Compiling pattern matching by term decomposition. *Journal of Symbolic Computation*, 15(1):1–26, Jan. 1993.

[7] S. A. Romanenko. Arity Raiser and Its Use in Program Specialization. In *ESOP'90, Copenhagen, Denmark*, volume 432 of *LNCS*, pages 341–360. Springer-Verlag, 1990.

[8] P. Thiemann. Avoiding repeated tests in pattern matching. *Lecture Notes in Computer Science*, 724, 1993.

[9] P. Thiemann. Higher-order redundancy elimination. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 73–83, Orlando, Florida, 25 June 1994.

[10] P. Wadler. Efficient compilation of pattern-matching. In S. L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, chapter 5. Prentice-Hall, 1987.