ML pattern match compilation and partial evaluation*

Peter Sestoft
(sestoft@dina.kvl.dk)

Department of Mathematics and Physics
Royal Veterinary and Agricultural University
Thorvaldsensvej 40, DK-1871 Frederiksberg C, Denmark

Abstract: We derive a compiler for ML-style pattern matches. It is conceptually simple and produces reasonably good compiled matches. The derivation is inspired by the instrumentation and partial evaluation of naïve string matchers. Following that paradigm, we first present a general and naïve ML pattern matcher, instrument it to collect and exploit extra information, and show that partial evaluation of the instrumented general matcher with respect to a given match produces an efficient specialized matcher.

We then discard the partial evaluator and show that a match compiler can be obtained just by slightly modifying the instrumented general matcher. The resulting match compiler is interesting in its own right, and naturally detects inexhaustive matches and redundant match rules.

1 Introduction

Consel and Danvy [3] studied a general string matcher, taking two inputs: a *pattern* string to look for, and an *object* string to look in. The matcher finds the first occurrence (if any) of the pattern in the object string. They showed that partial evaluation of a naïve string matcher with respect to a static pattern did not improve the algorithm. Then they instrumented the matcher to collect and exploit information about previous partial matches, taking care to make this information static, and showed that partial evaluation of the instrumented matcher would produce efficient Knuth-Morris-Pratt style matchers [11].

Jørgensen applied the same idea to tree pattern matching [9, 10]. He studied a general tree pattern matcher, taking two inputs: a *match*, which is a sequence of *patterns*, and an *object* term. The matcher finds the first pattern (if any) which is matched by the object. Jørgensen showed that the instrumentation approach carries over to the new setting, producing efficient specialized matchers from an instrumented version of the naïve matcher.

Continuing Jørgensen's work, we handle arbitrary ML-style pattern matches, present a general naïve matcher, and instrument it to collect positive and negative information. Initially we use a partial evaluator to specialize the instrumented matcher, but then observe that a specialized matcher implements a decision *tree*:

* Version March 5, 1996. To appear in Danvy, Glück, and Thiemann (editors): Dagstuhl Seminar on Partial Evaluation. Lecture Notes in Computer Science, Springer-Verlag.

it has no loops and no sharing — no memoization is needed. We therefore discard the partial evaluator, modify the instrumented matcher, and obtain a conceptually simple and fairly efficient ML match compiler.

In Section 2 we describe ML-style tree pattern matching, and in Section 3 we present a naïve pattern matcher, taking two inputs: a sequence of match rules to be tested, and an object term to match against the rules. In Section 4 we show how to represent positive and negative information about terms, and in Section 5 we instrument the naïve matcher to collect such information. In Section 6 we partially evaluate the instrumented matcher with respect to static match rules, and obtain a specialized matcher which performs no repeated tests. In Section 7 we construct the final match compiler by replacing dynamic variables and expressions of the instrumented matcher with symbolic terms. Section 8 discusses related work, and Section 9 contains the conclusion.

2 ML pattern matching

We assume that the object language is statically typed, so that the set of constructors that may appear at some point in a pattern or term is known at compile-time. This description fits e.g. Standard ML and Haskell, but not Scheme.

2.1 Constructors, patterns, terms, and matching

For the theoretical development, we use slightly generalized notions of constructor and pattern. A *constructor* is characterized by

- its *name*; in implementations this is typically a numerical tag;
- its *arity*, which is the number of arguments it takes (and is assured to get, by the type system);
- its *span*, which is the total number of constructors in its type.

A *pattern* is either a variable, or a constructor applied to a possibly empty sequence of patterns. Patterns must be *linear*: no variable may occur more than once in a pattern. A *(proper) term* is a pattern not containing variables. Constructors and patterns can be modelled as follows:

```
type con = { name : string, arity : int, span : int }
datatype pat = PVar of string | PCon of con * pat list
```

Example 1. For instance, the three constructors `Null`, `Leaf`, and `Node` declared by the Standard ML declaration

```
datatype ´a tree =
    Null | Leaf of ´a | Node of ´a tree * ´a * ´a tree
```

can be represented as follows:

```
val Nullc = {name = "Null", arity = 0, span = 3}
val Leafc = {name = "Leaf", arity = 1, span = 3}
val Nodec = {name = "Node", arity = 3, span = 3}
```

Then the SML term `Node(Leaf 9, 12, Node(Leaf 4, 7, Null))` would be encoded by the following term:

```
PCon(Nodec, [PCon(Leafc, [9]), 12,
             PCon(Nodec, [PCon(Leafc,[4]),7,PCon(Nullc,[])])])
```

A *match rule* or *mrule* is a pair of a pattern and an (unspecified) right-hand side, and a *match* is a sequence of match rules:

```
type 'rhs match = (pat * 'rhs) list
```

A *match object* is the term to be matched against a pattern. A term *matches* a pattern if there is a substitution of terms for variables in the pattern which makes the term and the pattern equal. The problem of pattern matching, given an object and a match, is to find the first match rule `(pat, rhs)` for which the object matches the pattern `pat`, or to decide that the no such rule exists in the match.

2.2 Representing ML data

We assume the reader is familiar with Standard ML pattern matching [14].

In an ML datatype, such as `'a tree` above, a constructor `Null`, `Leaf`, or `Node` is encoded by taking its arity to be the declared number of arguments[2], and its span to be the number of constructors belonging to the datatype.

An ML tuple, such as `(1, Null, true, "abc")`, is represented by a single constructor with span 1 and arity equal to the number of components, e.g. the arity is 4 in this example, and 2 for the pair constructor.

The components of an ML record can be sorted by their labels, so records can be handled as tuples in an implementation.

The ML special constants of type `int`, `real`, and `string` are encoded by constructors with arity 0 and infinite span. Character constants of type `char` are encoded with arity 0 and span 256 (for the ASCII character set).

A vector[3] constructor `#[_, ..., _]` has arity equal to the number of listed arguments, and span 1. Its name must reflect the arity, so that vector constructors of different arities are distinct. The ML exception constructors, such as `Div` or `Io`, have arity 0 or 1 and have infinite span. The ML `ref` constructor has arity 1 and span 1.

Hence all matchable ML data can be represented by our notions of constructor, pattern, and term.

---

[2] In principle, an ML value constructor takes either zero or one argument, which may be a tuple. It is immaterial for the present development whether one follows this model strictly, or allows a constructor to take multiple arguments. For purposes of matching, there is no difference between the pattern $C(v)$ and the pattern $C(v_1, \ldots, v_a)$ where $a = arity(C)$ and $v, v_1, \ldots, v_a$ are variables.

[3] Character and vector constructors are extensions to Standard ML, implemented by Standard ML of New Jersey, Moscow ML, and other systems.

Example 2. ML booleans and tuple, pair, and list constructors, can be encoded as follows:

```
val tt = PCon({name="true",  arity=0, span=2}, [])
val ff = PCon({name="false", arity=0, span=2}, [])

fun tup args = PCon({name="", span=1, arity=length args}, args)
fun pair(x, y) = tup[x, y]

val nil       = PCon({name="nil",  arity=0, span=2}, [])
fun cons(a,b) = PCon({name="cons", arity=2, span=2}, [a,b])
```

3 A naïve ML pattern matcher

The general pattern matcher takes as input a match object (a term) and a match (a sequence of mrules), and produces as output the right-hand side of the first matching mrule, or fails.

It works by sequentially matching the object against the mrule patterns (the left-hand sides); if a pattern fails, then the next one is tried, etc., until the object matches some pattern, or there are no more patterns. As in a naïve string matcher, this is potentially inefficient, because tests of the same subterm may be performed repeatedly. The worst-case complexity is the product of the number of mrules and the size of the object term.

The naive matcher shown below takes two inputs: an object term `origobj`, and a match `allmrules`. It is implemented by three mutually recursive functions, with the following specifications:

- `fail(rules)` will attempt to match the original object term `origobj` against each pattern `pat` from a rule `(pat, rhs)` in `rules`; it succeeds with the `rhs` if the rule matches; it fails otherwise.
- `succeed(work, rhs, rules)` succeeds, returning the `rhs`, if for every pair `(pat, obj)` in `work`, the object `obj` matches the pattern `pat`; or succeeds if the original object `origobj` matches some rule from `rules`; fails otherwise. Actually, `work` is a stack of lists of `(pattern, object)` pairs; this saves some append operations.
- `match(pat1,obj1,work,rhs,rules)` succeeds, returning the `rhs` if `obj1` matches `pat1`, and for every pair `(pat, obj)` in `work`, `obj` matches `pat`; or succeeds if `origobj` matches some rule in `rules`; fails otherwise.

That is, `fail` expresses disjunction of matches, `succeed` expresses conjunction of matches, and `match` does the actual matching work. The `rules` parameter is a list of alternatives to backtrack to; and the `work` parameter is a stack of match hypotheses still to be checked. These parameters are first-order representations of the failure and success continuations, respectively. This is useful in connection with our representation of positive and negative information in the instrumented matcher; see Section 4.3. Jørgensen's matcher represented the continuations by functions [9]. The function `main` implements the matcher:

```
fun main (origobj, allmrules) =
let fun fail []                          = NONE
      | fail ((pat1, rhs1) :: rulerest) =
        match(pat1, origobj, [], rhs1, rulerest)
    and succeed([],             rhs, rules) = SOME rhs
      | succeed(work1 :: workr, rhs, rules) =
        case work1 of
            ([], []) => succeed(workr, rhs, rules)
          | (pat1::patr, obj1::objr) =>
                match(pat1, obj1, (patr,objr)::workr, rhs, rules)
    and match(PVar _, _, work, rhs, rules) =
        succeed(work, rhs, rules)
      | match(PCon(pcon,pargs),PCon(ocon,oargs),work,rhs,rules) =
        if ocon = pcon then
            succeed((pargs, oargs) :: work, rhs, rules)
        else
            fail rules
in fail allmrules end
```

Note that there are no arity tests; static typing ensures that if two constructor names are equal, then they are applied to the same number of arguments, whether occurring in a pattern or a term.

4 Positive and negative information about terms

We now instrument the naive matcher to record and exploit information about the match object. The only source of such information is the test `ocon = pcon` in the `match` function above. We shall record *positive* information from tests which succeed, and *negative* information from tests which fail.

4.1 Term descriptions

This information will be recorded in the form of a *term description*. A *positive* term description is a skeletal term, consisting of a constructor together with a tuple of argument term descriptions. A *negative* term description is a set of impossible constructors:

```
datatype termd = Pos of con * termd list | Neg of con list
```

The positive term description $\texttt{Pos}(c, [t_1, \ldots, t_a])$ describes any term whose topmost constructor is c, and whose components are described by t_1 through t_a. The negative term description $\texttt{Neg}\{c_1, \ldots, c_n\}$ describes any term whose topmost constructor is none of $c_1, \ldots, c_n$. In a negative term description $\texttt{Neg}(S)$, the cardinality of S must be strictly less than the span of any constructor in S. Otherwise the term description is contradictory, and describes the empty set of terms. Due to static typing, all constructors in S must belong to the same type and hence have the same span.

Example 3. For terms of type `'a tree`, the term description `Neg[Node]` describes terms which are either `Null` or `Leaf(`a`)` for some a. The positive term description `Pos(Node,[Neg [Node], Neg [], Pos(Null,[])])` describes terms of the form `Node(`t`, `a`, Null)` where t does not have form `Node(_,_,_)`, and a is some term.

The set of term descriptions is

$$\begin{array}{rl} \mathit{TermD} = & \{\ \mathtt{Neg}(S) \mid \forall c \in S.\mathit{card}(S) < \mathit{span}(c)\ \} \\ & \cup\ \{\ \mathtt{Pos}(c, [t_1, \ldots, t_a]) \mid a = \mathit{arity}(c), t_i \in \mathit{TermD}\ \} \end{array}$$

There is a natural information preordering $\sqsubseteq$ on the set TermD of term descriptions, defined as follows, where $\bot = \mathtt{Neg}\{\}$:

$$\begin{array}{ll} \mathtt{Neg}(S_1) \sqsubseteq \mathtt{Neg}(S_2) & \text{iff } S_1 \subseteq S_2 \\ \mathtt{Neg}(S) \sqsubseteq \mathtt{Pos}(c, [t_1, \ldots, t_a]) & \text{iff } c \notin S \\ \mathtt{Pos}(c, [\bot, \ldots, \bot]) \sqsubseteq \mathtt{Neg}(S) & \text{iff } \mathit{card}(S \cup \{c\}) = \mathit{span}(c) \\ \mathtt{Pos}(c, [t_1, \ldots, t_a]) \sqsubseteq \mathtt{Pos}(c, [u_1, \ldots, u_a]) & \text{iff } t_i \sqsubseteq u_i \text{ for all } i = 1, \ldots, a \end{array}$$

The relation $\sqsubseteq$ is a preordering because $\mathtt{Neg}(S)$ provides the same information as $\mathtt{Pos}(c, [\bot, \ldots, \bot])$ when $\mathit{card}(S \cup \{c\}) = \mathit{span}(c)$; taking the quotient over the congruence induced by this equivalence gives a partial order. The least element is $\bot = \mathtt{Neg}\{\}$, which describes any term, and represents the absence of information. A maximal element is an encoding of a proper term, describing just that term.

In the instrumented matcher, the information about the match object is $\bot$ initially, and grows monotonically during the execution of the instrumented matcher. The `work` stack of `(pat, obj)` pairs must be extended to a stack of `(pat, obj, dsc)` triples, which records a subterm description `dsc` along with every subterm `obj` of the object.

The auxiliary function `addneg: termd * con -> termd` adds negative information to a partial term:

```
fun addneg(Neg nonset, con) = Neg(con :: nonset)
```

It has two preconditions: $\mathtt{con} \notin \mathtt{nonset}$, and $\mathit{card}(\mathtt{nonset} \cup \{\mathtt{con}\}) < \mathit{span}(\mathtt{con})$.

4.2 Static matching

Static matching attempts to match a constructor `ocon` from the object against a constructor `pcon` from the pattern, using only the object term description which has been constructed by those matching steps already performed. Static matching produces one of three answers: `Yes`, `No`, or `Maybe`, as follows:

a. If the object constructor is (positively) known to be c, and $\mathit{pcon} = c$, then the answer is `Yes`.
b. If the object constructor is (positively) known to be c, and $\mathit{pcon} \neq c$, then the answer is `No`.

c. If the object constructor is (negatively) known not to be any of $c_1, \ldots, c_n$, and $pcon = c_i$ for some $1 \leq i \leq n$, then the answer is `No`.
d. If the object constructor is (negatively) known not to be any of $c_1, \ldots, c_n$ (all distinct), and $pcon \neq c_i$ for all $1 \leq i \leq n$, and $span(pcon) = 1 + n$, then the answer is `Yes`.
e. Otherwise the answer is `Maybe`.

For constructors with span 1, such as tuples and pairs, rule (d) ensures that matching succeeds unconditionally. Since inequality of SML exception constructors cannot be safely decided at compile-time, rule (b) never applies for exception constructors. This is easily ensured in an implementation, and we shall disregard exception constructors for the remainder of the paper. Constructors of type `int`, `real`, and `string`, and exception constructors, have infinite span, so rule (d) never applies for those.

Using the span to optimize matches is well-known in match compilers for typed languages, and was suggested also by Jørgensen [9, page 191].

Static matching of the object term description against a pattern constructor `pcon` is implemented by a function `staticmatch: con * termd -> matchresult`.

4.3 Context descriptions

During the matching of a composite object against a composite pattern, we need to manage partial term descriptions, describing the part of the term which is already matched at a given point. Since we match top-down and left-right, the part of the description already computed is above and to the left of the current subterm. Hence a left *context description* must describe the constructors on the path from the root to the current subterm, and those of their arguments which are to the left of the current subterm.

For example, assume we are matching the object

```
Node(Node(Null, 1, Leaf 2), 3, Null)
```

against the pattern

```
Node(Node(x, 1, Leaf y), z, Null)
```

and assume the current subterm is the rightmost occurrence of `Null`. Then the current context description should be

```
Node(Node(⊥, 1, Leaf ⊥), ⊥, [])
```

where $\perp$ denotes the absence of information, and $[\,]$ is the 'hole' containing the current subterm.

This context is represented by the path of constructors from the current subterm up to the root, together with descriptions of those of their arguments which are to the left of the hole. Hence a context description is a list of constructors and argument descriptions:

```
type context = (con * termd list) list
```

In particular, the example context shown above is described by

```
[(Node, [⊥, Pos(Node,[⊥, 1, Pos(Leaf,[⊥])])])]
```

This represents the partial application of constructor `Node` to two arguments, described by `Pos(Node, ...)` and $\perp$. The general form of a context description is

$$[(c_n, [t_{nk_n}, \ldots, t_{n1}]), \ldots, (c_1, [t_{1k_1}, \ldots, t_{11}])]$$

where the t_{ij} are term descriptions, $k_i < arity(c_i)$ for $i < n$, and $k_n \leq arity(c_n)$. If $k_n < arity(c_n)$, then it represents the context

$$c_1(t_{11}, \ldots, t_{1k_1}, c_2(\ldots c_n(t_{n1}, \ldots, t_{nk_n}, [\,], \perp, \ldots, \perp) \ldots \perp), \perp, \ldots, \perp)$$

where information-less descriptions $\perp$ are added on the right to saturate the constructor applications. If $k_n = arity(c_n)$, then it represents a context of form

$$c_1(t_{11}, \ldots, t_{1k_1}, c_2(\ldots c_n(t_{n1}, \ldots, t_{nk_n}), [\,], \ldots \perp), \perp, \ldots, \perp)$$

In particular, the empty list `[]` of partially applied constructors (with $n = 0$) represents the top-level context $[\,]$.

When the matching of a subterm succeeds, it produces a term description `dsc` which is an argument to the local-most constructor c_n. Hence we partially fill the hole with `dsc`, by adding `dsc` to the list of argument descriptions for the local-most constructor c_n. Function `augment: context * termd -> context` accomplishes this:

```
fun augment([],                  dsc) = []
  | augment((con, args)::rest, dsc) = (con, dsc :: args) :: rest
```

When argument descriptions $[t_{na}, \ldots, t_{n1}]$ for all arguments of the local-most constructor c_n have been found, a positive term description $\texttt{Pos}(c_n, [t_{n1}, \ldots, t_{na}])$ is constructed, and the remainder of the context is augmented with this description. This is done by function `norm`:

```
fun norm ((con, args) :: rest) =
    augment(rest, Pos(con, rev args))
```

When the matching of a subterm fails, the current match rule must be abandoned and the next one tried, but first the object term description must be reconstructed. There are three components:

(1) the context description `ctx`, which describes the part of the object term to the left of the current subterm;
(2) the current subterm description `dsc`; and
(3) the term descriptions on the `work` stack, which together describe the part of the object term to the right of the current subterm.

These three components are reassembled to a term description as follows. If the context and the `work` stack are empty, then the current subterm description is the desired result. Otherwise both are non-empty, and the local-most partial constructor application (in the context) is further applied to the current subterm description, plus the top-most list of argument descriptions from the `work` stack, to obtain a term description. Then the rest of the context is recursively applied to this description and to the rest of the `work` stack.

This procedure is implemented by the `builddsc` function:

```
fun builddsc([], dsc, []) = dsc
  | builddsc((con, args)::rest, dsc, (_, _, dargs) :: work) =
    builddsc(rest, Pos(con, rev args @ (dsc :: dargs)), work)
```

Jørgensen used an occurrence environment, mapping occurrences (also called positions, or paths) of a term to term descriptions, to represent the knowledge acquired through partial matching of a term [9]. Hence he did not need a notion of context.

5 The instrumented ML pattern matcher

The instrumented matcher improves on the naïve one by (1) recording positive and negative information about the object term, also in the `work` list, and (2) attempting static matching using this information, resorting to actual tests on the object only when the static matching produces the result `Maybe`. The function `main: pat * ´rhs match -> ´rhs option` implements the instrumented matcher:

```
fun main(origobj, allmrules) =
let fun fail(dsc, [])                             = NONE
      | fail(dsc, (pat1, rhs1) :: rulerest) =
        match(pat1, origobj, dsc, [], [], rhs1, rulerest)
    and succeed(ctx, [], rhs, rules)          = SOME rhs
      | succeed(ctx, work1::workr, rhs, rules) =
        case work1 of
            ([], [], []) => succeed(norm ctx, workr, rhs, rules)
          | (pat1::patr, obj1::objr, dsc1::dscr) =>
                match(pat1, obj1, dsc1, ctx,
                      (patr,objr,dscr)::workr, rhs, rules)
    and match(PVar _, obj, dsc, ctx, work, rhs, rules) =
        succeed(augment(ctx, dsc), work, rhs, rules)
      | match(PCon(pcon, pargs), PCon(ocon, oargs),
              dsc, ctx, work, rhs, rules) =
        let fun args f = List.tabulate(#arity pcon, f)
            fun getdargs (Neg _) = args (fn _ => Neg [])
              | getdargs (Pos(con, dargs)) = dargs
            fun succeed´ () =
                succeed((pcon, []) :: ctx,
                        (pargs,oargs,getdargs dsc)::work,
                        rhs, rules)
            fun fail´ newdsc =
                fail(builddsc(ctx, newdsc, work), rules)
        in case staticmatch(pcon, dsc) of
            Yes   => succeed´ ()
          | No    => fail´ dsc
          | Maybe => if ocon = pcon then succeed´ ()
                     else fail´ (addneg(dsc, pcon))
        end
in fail(Neg [], allmrules) end
```

If `allmrules` is static and `origobj` dynamic, only the underlined parameters and test will be dynamic. In addition, `work` is a partially static list of triples `(pats, objs, dscs)` where the patterns `pats` and the term descriptions `dscs` are static, and the `objs` are dynamic. Hence all tests can be eliminated by the partial evaluator, except the test `ocon = pcon`, performed when the static matching returns `Maybe`. This is also the only place new information is born, by recording the outcome of the dynamic test.

Two new auxiliary functions are used. Function `getdargs` returns the list of argument descriptions from a term description: if it is positive, then the argument descriptions `dargs` are explicitly provided; if it is negative, then all argument descriptions are implicitly $\perp$.

Function `success´` is called after a successful match of a constructor; it extends the context with positive information about the matched constructor. Function `fail´` is called after an unsuccessful match of a constructor; if the match is dynamic, then new negative information is added to the subterm description before `builddsc` reconstructs the object term description.

6 Experiments with a simple partial evaluator

We rewrote the above instrumented ML program in a subset of Scheme, separating binding times by splitting the `work` list into three parallel lists. It was partially evaluated using the Scheme0 partial evaluator, a prototypical polyvariant program specializer for a first-order Scheme subset [8, Chapter 5]. The results are as expected, although clumsy, since the partial evaluator performs no postunfolding etc. Indeed, the simplicity of the Scheme0 partial evaluator ensures that the result is due to the recording and exploitation of static information, rather than powerful optimizations and transformations performed by the partial evaluator. This is important, as the ultimate goal is to write the match compiler without a partial evaluator.

Consider the following example datatype:

```
datatype lam =
    Var of int
  | Lam of int * lam
  | App of lam * lam
  | Let of int * lam * lam
```

and the following example match on this type:

```
    Var x                              => 111
  | Lam(x, Var y)                      => 222
  | Lam(x, Lam(y, z))                  => 333
  | Lam(x, App(y, z))                  => 444
  | App(Lam(x, y), z)                  => 555
  | App(App(x, y), z)                  => 666
  | Let(x, Let(y, z, v), w)            => 777
  | Lam(x, Let(y, z, v))               => 888
  | Let(x, y, App(z, v))               => 999
  | App(App(Lam(x, Lam(y, z)), v), w) => 1010
```

The residual program contains 66 functions with a total of 10 conditionals, testing some subterm for equality with a constructor. The control structure is shown below, after unfolding of trivial function calls and removal of unused variables, and (ab)using the notation `#con` for extraction of the topmost term constructor. Observe that test is performed twice. The occurrences of `FAIL` show that the match was inexhaustive. The non-occurrence of the right-hand side `1010` shows that the last match rule is redundant (it is covered by the sixth rule):

```
fun f1 obj = if #con obj = Var then 111
             else if #con obj = Lam then f2 obj
             else if #con obj = App then f3 obj
             else f4 obj
and f2 obj = if #con(#2 obj) = Var then 222
             else if #con(#2 obj) = Lam then 333
             else if #con(#2 obj) = App then 444
             else 888
and f3 obj = if #con(#1 obj) = Lam then 555
             else if #con(#1 obj) = App then 666
             else FAIL
and f4 obj = if #con(#2 obj) = Let then 777
             else if #con(#3 obj) = App then 999
             else FAIL
```

7 ML match compilation without partial evaluation

7.1 Decision trees and access paths

Inspection shows that `match` will not be specialized twice with the same values of `pat`, `work` and `rules` — its first, fifth, and seventh parameters. Hence the call graph of a specialized matcher is a *tree*.

Thus memoization in the partial evaluator serves no purpose, and we can generate decision trees directly. We then obtain a match compiler which, given a match, generates a decision tree. The decision tree performs tests on subterms of the object term; such subterms are denoted by symbolic *access paths*. An access path is either `Obj`, meaning the entire object term, or `Sel(`i`, `acc`)`, meaning the i'th component of the subterm pointed out by acc:

```
datatype access = Obj | Sel of int * access
```

A decision is either `Failure`, which immediately fails; or `Success(`rhs`)` which immediately succeeds, returning rhs; or `IfEq(`acc`, `con`, `d_t`, `d_f`)` which tests whether con equals the constructor at subterm acc of the object term, and then evaluates decision d_t or decision d_f according as the outcome was true or false:

```
datatype ´rhs decision =
    Failure
  | Success of ´rhs
  | IfEq of access * con * ´rhs decision * ´rhs decision
```

7.2 The match compiler

Now the match compiler itself is obtained from the instrumented matcher just by replacing the dynamic data and computations by symbolic data and computations. Namely,

– the value NONE returned by fail is replaced by the decision tree Failure;
– the value SOME rhs returned by succeed is replaced by Success rhs;
– a new function getoargs produces access paths into the object term;
– the dynamic parameter PCon(ocon, oargs) is replaced by access path obj;
– the dynamic test if ocon = pcon then e1 else e2 is replaced by the generation of a node IfEq(obj, pcon, e1, e2) in the decision tree.

The resulting match compiler compile: ´rhs match -> ´rhs decision is strikingly similar to the instrumented matcher shown in Section 5. The differences are underlined:

```
fun compile allmrules =
let fun fail(dsc, [])                              = Failure
      | fail(dsc, (pat1, rhs1) :: rulerest) =
        match(pat1, Obj, dsc, [], [], rhs1, rulerest)
    and succeed(ctx, [], rhs, rules)            = Success rhs
      | succeed(ctx, work1::workr, rhs, rules) =
        case work1 of
            ([], [], []) => succeed(norm ctx, workr, rhs, rules)
          | (pat1::patr, obj1::objr, dsc1::dscr) =>
               match(pat1, obj1, dsc1, ctx,
                     (patr, objr, dscr) :: workr, rhs, rules)
    and match(PVar _, obj, dsc, ctx, work, rhs, rules) =
        succeed(augment(ctx, dsc), work, rhs, rules)
      | match(PCon(pcon, pargs),obj,dsc,ctx,work,rhs,rules) =
        let fun args f = List.tabulate(#arity pcon, f)
            fun getdargs (Neg _) = args (fn _ => Neg [])
              | getdargs (Pos(con, dargs)) = dargs
            fun getoargs () = args (fn i => Sel(i+1, obj))
            fun succeed´ () =
                succeed((pcon, []) :: ctx,
                        (pargs,getoargs (),getdargs dsc)::work,
                        rhs, rules)
            fun fail´ newdsc =
                fail(builddsc(ctx, newdsc, work), rules)
        in case staticmatch(pcon, dsc) of
              Yes   => succeed´ ()
            | No    => fail´ dsc
            | Maybe => IfEq(obj, pcon,
                            succeed´ (),
                            fail´ (addneg(dsc, pcon)))
        end
in fail(Neg [], allmrules) end
```

7.3 Properties of the match compiler

The above match compiler produces a decision tree, the size of which is at most linear in the sum of the pattern sizes. However, the tree is not optimal, because we perform the matching in strict top-down, left-right order. The decision tree may be larger than necessary, and may perform more tests at runtime than necessary.

Example 4. This example from Baudinet and MacQueen shows that their algorithm may create more compact trees than ours:

```
datatype color = red | blue | green;

case ... of
   (true,  green) => 111
 | (false, green) => 222
```

Our match compiler produces the following code which tests the first component before the second one, thus duplicating the test on the `green` constructor:

```
IfEq(Sel(1, Obj), true,
     IfEq(Sel(2, Obj), green,
              Success 111,
              Failure),
     IfEq(Sel(2, Obj), green,
              Success 222,
              Failure))
```

Baudinet and MacQueen's algorithm would produce a decision tree which tests the second component first, leading to a smaller decision tree.

Example 5. Our match compiler will compile the following

```
case ... of
    (large_pattern, true)  => 111
  | ( _,            false) => 222
```

into a decision tree which attempts to match the first component to the large pattern, before it checks whether the second component is `true`. It seems more sensible to perform the latter simple test first: if the simple test fails, there is no need to check the large pattern.

We have not yet tested the match compiler on real programs, so it is unclear whether the non-optimal behaviour above is problematic in practice.

Example 6. The time and space consumption of the match compiler itself is exponential in n on matches with $n + 1$ rules, each a $2n$-tuple, of the following form:

```
datatype t = A | B
fun f x = case x of
   (A,A,_,_,_,_,_,_,_,_) =>  0
 | (_,_,A,A,_,_,_,_,_,_) =>  1
 | (_,_,_,_,A,A,_,_,_,_) =>  2
 | (_,_,_,_,_,_,A,A,_,_) =>  3
 | (_,_,_,_,_,_,_,_,A,A) =>  4
 | (A,B,A,B,A,B,A,B,A,B) => ~1
```

Simple practical experiments show that the match compilers used in Standard ML of New Jersey, Edinburgh ML, and the ML Kit take exponential time, and sometimes space, too.

The exponential space consumption can be avoided by using memoization while constructing the decision tree, as discussed in Section 7.5 below.

7.4 Inexhaustive matches and redundant cases

For every path through the generated decision tree, there is an object term which takes that path: all paths are feasible. Conversely, the decision tree is complete: for every object term there is a path through the decision tree, leading to a leaf (`Failure` or `Success`). This permits detection of inexhaustive matches and redundant cases, as required by e.g. the Definition of Standard ML [14].

Namely, a match can fail if and only if a `Failure` decision appears in the decision tree. Hence an inexhaustive match can be detected just by setting a global variable to `true` if a `Failure` decision is generated by the match compiler.

For the same reasons, the right-hand side of a match rule can be exercised if and only if it appears as a `Success` leaf in the decision tree. Hence the redundant match rules can be found by keeping a global set of match rules, initially containing all match rules. Whenever a `Success(`rhs`)` decision is being generated, remove rhs from the set. At the end of the match compilation, the set contains just the redundant rules.

7.5 Other refinements

A simple memoization scheme can be applied during the construction of the decision tree, turning the tree into a dag (directed acyclic graph), which will contain no two isomorphic subdags. An approach similar to hash-consing suffices. Whenever a branching node `IfEq(acc, con, t1, t2)` is about to be constructed, it is checked whether there is already a node with the same components, and if so, this node is used instead. An experimental implementation of this memoization works well and produces compact decision dags. It avoids the exponential space consumption problem on pathological matches discussed in Example 6 above.

The decision dag as constructed contains only binary equality tests. In practice, one wants to replace sequences of binary equality tests by more efficient switches, or indexed jumps through tables, whenever possible. A *switch* has the form `Switch(acc, rules, default)` where `acc` is a dynamic access path; `rules` is a list $[(c_1, dec_1), \ldots, (c_n, dec_n)]$ of pairs of constructors and decisions, where all constructors are distinct; and `default` is the decision to be evaluated if the constructor at `acc` does not occur in `rules`. The decision dag can be 'switchified' by a simple linear-time postprocessing.

Common subexpression elimination can be applied to the decision dag to avoid repeated extraction of the same object subterm.

7.6 Examples

Example 7. The match

```
   (x, nil) => 111
 | (nil, x) => 222
```

is compiled into the optimal decision tree

```
IfEq(Sel(2, Obj), nil,
     Success 111,
     IfEq(Sel(1, Obj), nil,
          Success 222,
          Failure))
```

Example 8. The example from Section 6 is compiled into the decision tree shown below. The match compiler detects that the match is inexhaustive and that rule 10 (the last one) is redundant. In the decision tree, sequences of binary equality tests have been replaced by switches, as proposed above. Its structure is identical to that of the residual matcher generated by partial evaluation in Section 6.

```
Switch(Obj,
       [(Var, Success 111),
        (Lam, Switch(Sel(2, Obj),
                     [(varc, Success 222),
                      (lamc, Success 333),
                      (appc, Success 444)],
                     Success 888)),
        (App, Switch(Sel(1, Obj),
                     [(Lam, Success 555),
                      (App, Success 666)],
                     Failure))],
       IfEq(Sel(2, Obj), Let,
            Success 777,
            IfEq(Sel(3, Obj), App,
                 Success 999,
                 Failure)))
```

8 Related work

Partial evaluation: Futamura and Nogi [4] showed that in principle, efficient string matchers à la Knuth-Morris-Pratt can be generated from a general naïve matcher. This required a generalized partial evaluator which would record the outcome of previous tests and use a theorem prover to decide subsequent tests. Apparently it was not implemented.

Consel and Danvy [3] demonstrated that even a simple partial evaluator could generate KMP-style string matchers from an *instrumented* naïve matcher.

Jørgensen [9, 10] applied the instrumentation approach to tree pattern matching, and showed that efficient compiled matches were generated; his work is a close precursor of the present work. Jørgensen also discussed several of the techniques presented here, but did not obtain a stand-alone match compiler by discarding the partial evaluator.

Glück and Jørgensen [5] achieved the programme outlined by Futamura and Nogi, generating a KMP-style string matcher from a general naïve matcher, using a specializer with positive and negative information, but without a theorem prover.

Match compilation: Augustsson described a top-down, left-right compilation algorithm for ML-style pattern matching in lazy functional languages [1]; see also Wadler's exposition in [16, Chapter 5]. It does not naturally discover inexhaustive matches and redundant cases.

In a lazy language, the order of testing of subterms affects evaluation order and hence termination, so terms are usually tested from top-down and left-right for simplicity and transparency. Our naïve matcher, and hence our match compiler, implements pattern matching as required by the Haskell report [6]. Our compiled matchers, like Jørgensen's, are more efficient than Augustsson's and Wadler's on non-uniform matches.

Jørgensen showed that some inefficiencies in matchers generated by Augustsson's and Wadler's method can be removed by a simple postprocessing, recording and using information from previous matches. The resulting compiled matches are always compact and usually efficient; however, the approach does not naturally detect inexhaustive matches and redundant rules.

Petterson [15] presented an improvement of the techniques of Augustsson and Wadler, based on finite automata, but retaining the top-down, left-right matching order. His match compiler has many similarities to that derived here; in particular, it naturally detects inexhaustive matches and redundant rules. Petterson compares the use of memoization (Section 7.5) to minimization of automata.

The top-down, left-right matching order does not produce optimal compiled matchers. Huet and Lévy [7] studied optimality for *unambiguous* matches (non-overlapping patterns only). However, programming languages usually allow patterns to overlap, but impose an order on them, and make the match deterministic by selecting the first matching one.

To adapt Huet and Lévy's work to programming practice, Laville [12] studied optimal compilation of ordered ambiguous pattern matches for lazy languages, relaxing the left-right matching order. As far as we know, his algorithms have not been used in a practical compiler.

Puel and Suárez [17] used constrained terms, similar to our term descriptions, instead of Laville's ordered patterns. An ordered collection of ambiguous patterns can be compiled into an unordered collection of unambiguous constrained patterns, from which one can construct a decision tree.

Augustsson's top-down, left-right match compilation technique is not optimal for strict languages, in which subterms may be tested in any order without compromising termination. Even so, the technique is being used successfully in e.g. the Caml Light system [13], which shows that the generated matchers are efficient enough in practice.

Baudinet and MacQueen [2] studied match compilation for Standard ML. Their goal was to construct as compact a decision tree as possible; according to the paper, this also generally minimizes the number of tests performed at runtime. Example 4 above shows that their method indeed may produce a more compact and efficient decision tree than ours. Since constructing a minimal decision tree is an NP-complete problem, heuristics are used to decide the order of subterm tests. The match compiler currently used in the Standard ML of New Jersey system was written by Bill Aitken and seems to incorporate the heuristics proposed by Baudinet and MacQueen [19].

Ramesh, Ramakrishnan, and Warren [18] studied automata-based match compilation for Prolog. Their results show that large procedures with complicated arguments benefit from the automata-based match compilation, whereas the usual simple 'indexing' on the first argument is preferable for small procedures with shallow argument terms.

9 Conclusion

We have developed a simple technique for compiling ML-style pattern matches, recording, as positive and negative information about the term being matched, the outcome of all tests previously considered in the match. Inexhaustive matches and redundant cases are discovered as a natural by-product of this approach. The match compiler is likely to be correct, since it was derived in a step-wise manner from a naïve general matcher. The compiled matchers are compact and as efficient as possible, given the fixed top-down, left-right matching order.

For strict languages, the match compiler is not optimal, but neither is it worse than some match compilers already in use. We expect to use it in the Moscow ML compiler [20, 21].

It may seem ironic that we first discard the partial evaluator because no memoization is needed, and then reintroduce memoization explicitly to obtain sharing of isomorphic subtrees in the decision tree. However, the memoization performed by existing partial evaluators is based on identity of available data (function arguments), not on the use made of these data (the corresponding function results), and would not lead to sharing of subtrees.

The present work can be considered a case study in program derivation. Although the resulting match compiler makes no explicit use of partial evaluation, it owes its existence to inspiration from that field.

Acknowledgement: Thanks to Stephan Diehl, Jesper Jørgensen, Jacques Noyé, Alberto Pettorossi, Sergei Romanenko, and the anonymous referees for helpful comments and pointers.

References

1. L. Augustsson. Compiling pattern matching. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture, Nancy, France,*

September 1985. (Lecture Notes in Computer Science, vol. 201), pages 368–381. Springer-Verlag, 1985.
2. M. Baudinet and D. MacQueen. Tree pattern matching for ML (extended abstract). Draft paper, AT&T Bell Laboratories, 1985.
3. C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30:79–86, January 1989.
4. Y. Futamura and K. Nogi. Generalized partial computation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. Amsterdam: North-Holland, 1988.
5. R. Glück and J. Jørgensen. Generating optimizing specializers. In *IEEE Computer Society International Conference on Computer Languages, Toulouse, France, 1994*, pages 183–194. IEEE Computer Society Press, 1994.
6. P. Hudak, S. Peyton Jones, P. Wadler, et al. Report on the programming language Haskell – a non-strict, purely functional language, version 1.2. *SIGPLAN Notices*, 27(5):R1–R162, May 1992.
7. G. Huet and J.-J. Lévy. Call by need computations in non-ambiguous linear term rewriting systems. Rapport de Recherche 359, IRIA Rocquencourt, France, 1979.
8. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
9. J. Jørgensen. Generating a pattern matching compiler by partial evaluation. In S.L. Peyton Jones, G. Hutton, and C. Kehler Holst, editors, *Functional Programming, Glasgow 1990*, pages 177–195. Berlin: Springer-Verlag, 1991.
10. J. Jørgensen. Compiler generation by partial evaluation. Master's thesis, DIKU, University of Copenhagen, Denmark, 1992. Student Project 92-1-4.
11. D.E. Knuth, J.H. Morris, and V.R. Pratt. Past pattern matching in strings. *SIAM Journal of Computation*, 6(2):323–350, 1977.
12. A. Laville. Implementation of lazy pattern matching algorithms. In H. Ganzinger, editor, *ESOP'88, 2nd European Symposium on Programming, Nancy, France, March 1988*, pages 298–316. Berlin: Springer-Verlag, 1988.
13. X. Leroy. The Zinc experiment: An economical implementation of the ML language. Rapport Technique 117, INRIA Rocquencourt, France, 1990.
14. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.
15. M. Petterson. A term pattern-match compiler inspired by finite automata theory. In U. Kastens and P. Pfahler, editors, *Compiler Construction, Paderborn, Germany, October 1992 (Lecture Notes in Computer Science, vol. 641)*. Berlin: Springer-Verlag, 1992.
16. S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.
17. L. Puel and A. Suárez. Compiling pattern matching by term decomposition. *Journal of Symbolic Computation*, 15(1):1–26, January 1993.
18. R. Ramesh, I.V. Ramakrishnan, and D.S. Warren. Automata-driven indexing of prolog clauses. *Journal of Logic Programming*, 23(3):151–202, 1995.
19. J. Reppy. Personal communication, November 1995.
20. S. Romanenko and P. Sestoft. *Moscow ML Language Overview, version 1.31*, October 1995. Available as `ftp://ftp.dina.kvl.dk/pub/mosml/doc/mosmlref.ps.Z`.
21. S. Romanenko and P. Sestoft. *Moscow ML Owner's Manual, version 1.31*, October 1995. Available as `ftp://ftp.dina.kvl.dk/pub/mosml/doc/manual.ps.Z`.