Polytypic Pattern Matching

Johan Jeuring
Chalmers University of Technology and University of Göteborg
S-412 96 Göteborg, Sweden
email: johanj@cs.chalmers.se

Abstract

The (exact) pattern matching problem can be informally specified as follows: given a pattern and a text, find all occurrences of the pattern in the text. The pattern and the text may both be lists, or they may both be trees, or they may both be multi-dimensional arrays, etc. This paper describes a general pattern-matching algorithm for all datatypes definable as an initial object in a category of F-algebras, where F is a regular functor. This class of datatypes includes mutual recursive datatypes and lots of different kinds of trees. The algorithm is a generalisation of the Knuth, Morris, Pratt like pattern-matching algorithm on trees first described by Hoffmann and O'Donnell.

1 Introduction

Most editors provide a search function that takes a string of symbols and returns the first position in the text being edited at which this string of symbols occurs. The string of symbols is called a pattern, and the algorithm that detects the position at which a pattern occurs is called a (exact) pattern-matching algorithm.

Pattern matching is not only important on strings, but also on multidimensional arrays, see [2, 4, 14], and various kinds of trees that occur in term rewriting systems, transformational programming systems, code generators, theorem provers, etc., see [7]. The literature on pattern matching on different datatypes is extensive and still growing.

Some of the pattern-matching algorithms on different datatypes exhibit a very similar structure: the pattern-matching algorithm on two-dimensional arrays given by Baker [2] and Bird [4] can be considered as a two-dimensional version of the Knuth, Morris and Pratt (KMP) pattern-matching algorithm on strings [16]. Similarly, the bottom-up tree pattern-matching algorithm from Hoffmann and O'Donnell [9] can be considered as a version of KMP on trees. This suggests that there exists a general algorithm, which, when instantiated on the datatype list, would give the KMP pattern-matching algorithm, and when instantiated on trees, would give a KMP-like tree pattern-matching algorithm. This general KMP-like pattern-matching algorithm is the subject of this paper.

We call a general algorithm like the general KMP-like pattern-matching algorithm a *polytypic* (Webster: 'having or involving several different types') algorithm. Sheard [26] calls these algorithms 'type parametric', and Bird et al. [3] call them 'generic'. An example of a polytypic algorithm is the familiar `map` function from functional programming. The `map` function takes a function f and a list x, and applies f to all elements of x. A similar function can be defined on binary trees: the `map` function on binary trees takes a function f and a binary tree t, and applies f to all elements of t. For a lot of datatypes `D`, the definition of the function `map_D` on that datatype is completely standard. A generic definition of the map of a datatype can be given by exploiting the categorical approach to modelling datatypes as initial objects in categories of functor-algebras [18]. Initial objects exist for regular functors, and the class of datatypes definable by means of a regular functor includes mutual recursive datatypes and lots of different kinds of trees. There exist regular functors for all datatypes defined as a `data` in Haskell in which there are no occurrences of function spaces and occurrences of the datatype being defined in the right-hand side expression have the same arguments as the datatype at the left-hand side. Other examples of polytypic algorithms can be found in the functional programming language Haskell [11], where it is possible to derive a limited number of functions for datatypes. For example, writing `deriving Eq` after a datatype (which satisfies a number of conditions) gives an equality function `==` on that datatype.

Polytypic algorithms are now standard in the Squiggol community, see [22, 21], where the 'Theory of Lists' [5, 6, 13] is extended to datatypes that can be defined by means of a regular functor. The category theory needed for this extension has been developed by Manes and Arbib [19] and Hagino [8]. The polytypic algorithms used in Squiggol are general recursive functions such as catamorphisms, paramorphisms, maps, etc. The polytypic pattern matching (PPM) algorithm described in this paper is an example of a special purpose polytypic algorithm. Other special purpose polytypic algorithms can be found in [21, 3, 26, 15]. Polytypic algorithms should not be confused with parametric polymorphic algorithms, such as `length :: [a] -> Int`, which have equal behaviour on different types, or ad-hoc polymorphic algorithms, such as `*` defined on both `Int` and `Float`, definitions of which for different datatypes may be completely unrelated. An instantiation of a polytypic algorithm on a specific datatype can be parametric polymorphic, an example is the `map` function, but it need not be. An example of a polytypic algorithm that is not parametric polymorphic is the algorithm that sums all the integers in a structure.

This paper describes a polytypic KMP-like pattern-matching algorithm. Ssince the algorithm matches sets of patterns, it is actually a polytypic version of Aho and Corasicks [1] extension of KMP to sets of patterns. The PPM algorithm is a generalisation of the KMP-like bottom-up tree pattern-matching algorithm of Hoffmann and O'Donnell. The PPM algorithm is linear in the size of the subject, at the cost of preprocessing the patterns which possibly requires time exponential in the size of the patterns. There exist several techniques for improving the performance of the preprocessing phase [7], but we have not implemented these.

The PPM algorithm offers several advantages. Firstly, when confronted with a pattern-matching problem in a specific kind of trees, the PPM algorithm gives an immediate solution. This avoids (sometimes complicated) encoding and decoding of datatypes in trees. Secondly, the PPM algorithm forces one to abstract from a particular kind of trees, and hence to concentrate on the essentials of pattern matching and datatypes.

The PPM algorithm is constructed using generalised definitions and results from Squiggol [5, 21, 18]. We first give a specification of pattern matching, and subsequently transform this specification into the desired pattern-matching algorithm. Since the transformations we use are correctness preserving, the resulting algorithm is a correct algorithm for pattern matching. Building upon the theory developed in Squiggol, Bird et al. [3] and Jeuring [15] describe a lot of polytypic algorithms. The basic idea in this theory is to associate a (category theory) functor with each datatype, and to define polytypic algorithms either by induction on the structure of functors, or by combining polytypic algorithms. Often we do not only define polytypic algorithms, but also new datatypes used by polytypic algorithms. For example, given a datatype `D`, the PPM algorithm assumes the existence of a datatype `VD` that is obtained by adding a nullary constructor to the definition of `D`. This nullary constructor plays the role of a variable. Since category theory provides a convenient modelling of datatypes as initial objects in categories of functor-algebras, it is a convenient vehicle for describing polytypic algorithms. No other results of category theory are used.

The PPM algorithm has been implemented using a polytypic programming preprocessor to Haskell. Using this preprocessor it is possible to generate functions like `map`, `size`, and `foldr` for arbitrary datatypes. The preprocessor has been extended such that given a datatype `D`, it generates a function `pm_D` which is the implementation of the KMP-like algorithm on the datatype `D`. The PPM algorithm can also be implemented using compile-time reflection [10], or dynamic typing [25].

This paper is organised as follows. Section 2 briefly describes the instantiations of the polytypic KMP pattern-matching algorithm on two example datatypes. Section 3 introduces several polytypic algorithms that are needed in the specification of the pattern-matching algorithm, specifies polytypic pattern matching, and derives the PPM algorithm. Section 4 concludes the paper and describes future work.

2 Examples of pattern-matching programs

This section briefly describes pattern-matching programs on the datatypes `L a` of lists, and `E a` of expression trees. Both programs are certainly not optimal, and can be improved in many ways. The goal of presenting this code is not to show the most efficient program, but instead to convey a number of concepts of polytypic pattern matching. The code presented in this section is semantically equivalent to the code that is automatically generated by the polytypic programming preprocessor. The automatically generated code contains more brackets, and some of the functions from the automatically generated code can be nicely combined into one function.

2.1 Example: pattern matching on lists

This subsection shows the instantiation on the datatype `L a` of lists of the PPM algorithm. The simple structure of this datatype enables us to simplify many of the general functions. The resulting program slightly differs from the usual KMP program.

General definitions

Values of the datatype `L a` are lists; they are either `Nil` or the `Cons` of a value of the type `a` and a list. There exists an equality `==` on the datatype `L a`. The equivalents of `foldr` and `scanr` (see the preludes of Gofer or Haskell) on the datatype `L a` are called `cata_L`, and `scan_L`, respectively.

```
data L a  =  Nil | Cons (a, L a)

cata_L :: ((a,b) -> b) -> b -> (L a -> b)
cata_L f e =
  \x -> case x of
          Nil        -> e
          Cons (x,xs) -> f (x,cata_L f e xs)

scan_L :: ((a,b) -> b) -> b -> (L a -> L b)
scan_L f e =
  let g (x,y) = Cons (f (x,hd y),y)
  in cata_L g (Cons (e,Nil))
```

where `hd :: L a -> a` returns the head of a nonempty list. We write `;` for `Cons`, and `"abc"` for the list `Cons ('a', Cons ('b', Cons ('c', Nil)))`. Function `pre_L`, which returns the prefixes of a list, is an example of a `cata_L`, and function `suf_L`, which returns the suffixes of a list, is an example of a `scan_L`.

```
?pre_L "acb"
["","a","ac","acb"]

pre_L :: L a -> [L a]
pre_L =
  let f (x,y) = Nil:map (\xs -> Cons (x,xs)) y
  in cata_L f [Nil]

?suf_L "acb"
"acb";"cb";"b";"";Nil

suf_L :: L a -> L (L a)
suf_L = scan_L Cons Nil
```

Sets are used a lot in our programs. We suppose that we have a class `Set` with operators `union` and `intersect`, an empty set `empty`, an equality `===`, an operator `remove` which removes a set from a set, and a function `bigunion` which returns the union of a list of sets. We let the type `[a]` be an instance of the class `Set`.

Finally, we define a function `decode` which given a default value, a table, and a key, selects a value from the table if the key appears in the table, and returns the default value otherwise. We also need a converse of `decode` which takes a table with sets as second elements.

```
decode :: Eq a => b -> [(a,b)] -> a -> b
```

```
decode def [] key = def
decode def (t:tab) key
  = snd t,              fst t == key
  = decode def tab key, otherwise

code :: Set b => a -> [(a,b)] -> b -> a
code def [] key = def
code def (t:tab) key
  = fst t,            snd t === key
  = code def tab key, otherwise
```

Pattern matching

By definition, a pattern matches a list if it is a prefix of that list, where a list `x` is a prefix of a list `y` if `x 'elem' pre_L y`.

```
match_L :: Eq (L a) => (L a,L a) -> Bool
match_L (p,q) = p 'elem' pre_L q
```

For example, `match_L ("ab","abc")` holds. Given a collection of patterns (values of the datatype `L`), the pattern-matching problem on `L` requires finding occurrences of these patterns in a text (another value of the datatype `L`). For example, matching the collection of patterns `["baa", "daa"]` in the text `"yabaadaabaa"` results in the following collection of matching patterns for each suffix of `"yabaadaabaa"`. Abbreviate the different 'matchsets' (the precise definition of a matchset and an explanation for the sequence of numbers will appear later) by: `0` = {}, `2` = {`"baa"`}, `5` = {`"daa"`}. If number `n` appears below a position in the string, this means that the collection represented by `n` matches at the suffix of the string starting at that position.

```
yabaadaabaa
00200500200
```

The central idea of KMP pattern matching is to determine for each suffix of the text, the collection of suffixes of the patterns that match, and then to use the fact that if `p` matches the string `x;xs`, then the tail of `p` matches `xs`. This fact can be expressed as the following property of function `match_L`. For all lists `p` and `x;xs`, we have:

$$\texttt{match\_L (p,x;xs)} \;\Rightarrow\; \texttt{p == ""} \vee \texttt{match\_L (tl p,xs)}$$

where `tl :: L a -> L` a returns the tail of a nonempty list. For the string `"yabaadaabaa"` we now obtain, adding the empty string `""` to the sets represented by `2`,`5`, and using in addition the abbreviations `1` = {`""`}, `3` = {`""`,`"a"`}, `4` = {`""`,`"a"`,`"aa"`}:

```
yabaadaabaa
13243543243
```

To determine the collection of suffixes of patterns matching a string `x;xs`, it suffices to determine the collection of suffixes of patterns matching the string `xs`, and to use a table obtained by preprocessing the patterns to determine the collection of suffixes of patterns matching at the string `x;xs`. The table is an association list, which associates pairs of elements and integers with integers. The integers denote sets of patterns. For example, the table we would obtain for the above example would associate the pair `('a',3)` with the number `4`, and `('d',4)` with `5`. There exist several ways to implement association lists. We have chosen the following type of tables.

```
type Table_L a = [((a,Int),Int)]
```

The program for pattern matching has the following structure. We assume that the value `ma_L` (abbreviating 'matches all') is the set with just the empty list (`1` in the above example).

```
pm_L :: (Eq (a,Int),Set [a],Set [L a]) =>
        [L a] -> L a -> L Int
pm_L ps = scan_L (pmf (pp_L ps)) ma_L

pmf :: Eq (a,Int) =>
       Table_L a -> (a,Int) -> Int
pmf tab (a,s) = decode ma_L tab (a,s)
```

So function `pm_L` returns a list, which at each position contains a number that denotes the set of suffixes matching at that position. Each of these numbers is obtained by inspecting the previous number (denoting a matchset) and the current value, and computing the number denoting the matchset matching at the current position. It remains to describe function `pp_L`, i.e., the preprocessing phase that constructs the table used in pattern matching.

Preprocessing the patterns

The preprocessing phase is essential to KMP pattern matching. The preprocessing functions given here deviate from the usual ones appearing in KMP pattern matching on lists. This is because the functions we construct are obtained from the general construction of preprocessing functions. We can distinguish four functions in the preprocessing phase: a function `matchsets_L` which determines the so-called matchsets, the function `elts_L` which determines the set of all elements of type `a` in the patterns of type `L a`, the function `table_L` which constructs the desired table, and, finally, the function `pats_L` which returns the set of suffixes of the patterns. For example,

```
? pats_L ["baa","daa"])
["baa","aa","a","","daa"]

? elts_L ["baa","daa"])
bad

? flatten_L (b;a;a;Nil)
baa
```

These functions are defined as follows.

```
pp_L :: (Set [a], Set [L a]) =>
        [L a] -> Table_L a
pp_L ps =
  let s = pats_L ps
  in table_L (elts_L ps) (matchsets_L s) s

pats_L :: Set [L a] => [L a] -> [L a]
pats_L = bigunion . map (flatten_L . suf_L)

elts_L :: (Eq a, Set [a]) => [L a] -> [a]
elts_L = bigunion . map flatten_L

flatten_L :: L a -> [a]
flatten_L = cata_L (uncurry (:)) []
```

Functions `matchsets_L` and `table_L` are defined below. For the collection of patterns `["baa", "daa"]` used in the above example, function `table_L` returns the following association list (the numbers denote the matchsets introduced in the previous subsection).

```
[ (('d',0),1), (('d',1),1), (('d',2),1)
, (('d',3),1), (('d',4),5), (('d',5),1)
, (('a',0),1), (('a',1),3), (('a',2),3)
, (('a',3),4), (('a',4),4), (('a',5),3)
, (('b',0),1), (('b',1),1), (('b',2),1)
, (('b',3),1), (('b',4),2), (('b',5),1)
]
```

Matchsets

The sets of suffixes of patterns represented by numbers above are so called *matchsets*. A matchset `ms` of a set of (suffixes of) patterns `ps` is a set of patterns for which there exists a list `x` such that all elements of `ms` match at `x`, and no element of `ps` not in `ms` matches at `x`. For example, for the patterns `["baa", "daa"]` we have the matchsets numbered `0,...,5` defined earlier. Note that the set {`""`,`"aa"`} is not a matchset, since at any position where `"aa"` matches, `"a"` matches too. The matchsets of a set of patterns are determined as follows.

```
matchsets_L :: (Eq (L a), Set [L a]) =>
               [L a] -> [(Int,[L a])]
matchsets_L s =
  zip [0..] (filter (ismatchset_L s) (subs s))

subs :: [a] -> [[a]]
subs = foldr (\x xs -> xs ++ map (x:) xs) [[]]

ismatchset_L :: (Eq (L a), Set [L a]) =>
                [L a] -> [L a] -> Bool
ismatchset_L ps ms =
  and [not (match_L (p,q))
      | p <- remove ps ms, q <- ms]
  &&
  and [samems_L (p,q) | p <- ms, q <- ms]

samems_L :: Eq (L a) => (L a,L a) -> Bool
samems_L (p,q) = match_L (p,q) || match_L (q,p)
```

The last two functions `ismatchset_L` and `samems_L` (abbreviating 'same matchset') determine whether or not the set of patterns `ms` is a matchset with respect to the set of patterns `ps`. Function `matchsets_L` assigns a number to each matchset.

Constructing the table

Given the matchsets we construct the table used by function `pmf`. The keys of the table are generated by `gen_L`. Function `gen_L` generates all possible situations that can occur while matching the list of patterns against an argument, except for those cases where the argument contains a value that does not occur in the patterns. For these cases function `pmf` returns the default value `ma_L`.

```
gen_L        :: [a] -> [b] -> [(a,b)]
gen_L as ns  =  [(a,n) | a <- as, n <- ns]

table_L :: Set [L a] =>
           [a] -> [(Int,[L a])] -> [L a] ->
           Table_L a
table_L as ms s =
  let cd (l,r) = code ma_L ms ([Nil] `union` h')
        where h' = h (l,decode [] ms r)
      h (l,rs) =
        filter (`elem` s) [Cons(l,q) | q <- rs]
  in [(g,cd g) | g <- gen_L as (map fst ms)]
```

Function `table_L` returns a list of pairs `(g, cd g)`, where function `cd` takes a pair consisting of an element `x` of type `a` and a number `n` denoting a matchset (matching, say, a list `xs`), and returns an integer denoting the matchset that matches at `x;xs`.

2.2 Example: pattern matching on expression trees

This subsection shows the instantiation on the datatype `E a` of expression trees of the PPM algorithm. An important new concept compared with the pattern-matching program on `L a` is the introduction of a pattern variable for expression trees.

General definitions

Values of the datatype `E a` are either a constant of type `a` or a sum of two expressions, or a product of two expressions. There exists an equality on the datatype `E a`, and `cata_E` is defined as follows.

```
data E a = Con a
         | Sum (E a, E a)
         | Prod (E a, E a)

cata_E :: (a -> b) ->
          ((b,b) -> b) ->
          ((b,b) -> b) ->
          E a -> b
cata_E f g h =
  let j = cata_E f g h
  in \x -> case x of
             Con a      -> f a
             Sum (x,y)  -> g (j x,j y)
             Prod (x,y) -> h (j x,j y)
```

An example of a `cata_E` is the function `size_E`, which takes a value `x` of type `E`, and returns the number of constructors in `x`.

```
size_E :: E a -> Int
size_E = let f (n,m) = n + m + 1
         in cata_E (const 1) f f
```

Again, we will need the function `scan_E`, the equivalent on the datatype `E a` of function `scanr`. The definition of function `scan_E` is slightly more complicated than the definition of `scan_L`, since we now need a new datatype, called `LE a` (for 'labelled expression trees') to represent the result of an application of `scan_E`. Values of the datatype `LE a` represent values of the same structure as values of `E a`, but they may be labelled with values. The result of matching a collection of patterns in a tree is a value of datatype `LE Int`, where a value of the datatype `LE Int` is a value of the same structure as the argument tree, which is labelled at each point with an integer denoting the collection of (suffixes of) patterns matching at that point.

```
data LE a = LCon a
          | LSum (LE a, LE a, a)
          | LProd (LE a, LE a, a)

scan_E :: (a -> b) ->
          ((b,b) -> b) ->
          ((b,b) -> b) ->
          E a -> LE b
scan_E f g h =
  let j k = \(x,y) -> (x,y,k (top x,top y))
  in cata_E (LCon.f) (LSum.j g) (LProd.j h)
```

where top :: LE a -> a returns the top element of a labelled expression. An example of a scan_L is the function suf_E, which given a value of type E, returns all its suffixes.

```
suf_E :: E a -> LE (E a)
suf_E = scan_E Con Sum Prod
```

For example

```
?suf_E (Sum (Con 1,Prod (Con 2, Con 3)))
LSum (LCon (Con 1)
     ,LProd (LCon (Con 2)
            ,LCon (Con 3)
            ,Prod (Con 2,Con 3)
            )
     ,Sum (Con 1,Prod (Con 2,Con 3))
     )
```

Adding a pattern variable

Since we want to allow for occurrences of a variable in patterns now, the pattern-matching problem is formulated in a slightly different way on this datatype. We assume there exists a datatype VE a values of which represent expression trees with possibly occurrences of a variable, i.e., expression trees in which zero or more subtrees are replaced by a variable.

```
data VE a = VCon a
          | VSum (VE a, VE a)
          | VProd (VE a, VE a)
          | VarE
```

A pattern is a value of the datatype VE a. Functions cata_VE and scan_VE are defined similar to cata_E and scan_E (where we assume that a datatype LVE a of labelled expression trees with possibly occurrences of a variable has been defined). A pattern p matches a value v if match_E (v,p) holds, where function match_E is defined below. For example, the pattern VSum (VCon 1,VarE) matches the expression tree Sum (Con 1, Prod (Con 2, Con 3))

```
match_E :: Eq a => (E a, VE a) -> Bool
match_E (_,VarE)                = True
match_E (Con x,VCon y)          = x == y
match_E (Sum (x,y),VSum (s,t))  = m (x,s) (y,t)
match_E (Prod (x,y),VProd (s,t)) = m (x,s) (y,t)
match_E (_,_)                   = False

m (x,s) (y,t) = match_E (x,s) && match_E (y,t)
```

Pattern matching

Function pm_E requires the existence of a datatype that is a sum of three components. Functions in_VE and out_VE are inverses of each other. They construct and destruct values of type VE, respectively.

```
data Sum3 a b c = In31 a | In32 b | In33 c

in_VE :: Sum3 a (VE a,VE a) (VE a,VE a) -> VE a
in_VE (In31 a)     = VCon a
in_VE (In32 (x,y)) = VSum (x,y)
in_VE (In33 (x,y)) = VProd (x,y)

out_VE :: VE a -> Sum3 a (VE a,VE a) (VE a,VE a)
out_VE (VCon a)      = In31 a
out_VE (VSum (x,y))  = In32 (x,y)
out_VE (VProd (x,y)) = In33 (x,y)
```

The type Sum3 is used in the type of the association list, Table_VE, the result type of the function that takes care of preprocessing the patterns.

```
type FE a b = Sum3 a (b,b) (b,b)

type Table_VE a = [(FE a Int,Int)]
```

Function pm_E takes a set of patterns and an expression, and returns a labelled expression of the same structure. The labels are integers denoting the matchsets matching at the position where the label occurs. Function pm_E is expressed in terms of function scan_E as follows.

```
pm_E :: (Eq (FE a Int), Set [a], Set [VE a]) =>
        [VE a] -> E a -> LE Int
pm_E ps = scan_E
            (pmfC tab)
            (pmfS tab)
            (pmfP tab)
  where tab = pp_VE ps

pmfC :: Eq (FE a Int) =>
        Table_VE a -> a -> Int
pmfS :: Eq (FE a Int) =>
        Table_VE a -> (Int,Int) -> Int
pmfP :: Eq (FE a Int) =>
        Table_VE a -> (Int,Int) -> Int

pmfC tab a     = decode ma_E tab (In31 a)
pmfS tab (x,y) = decode ma_E tab (In32 (x,y))
pmfP tab (x,y) = decode ma_E tab (In33 (x,y))
```

Preprocessing the patterns

In the definition of function pm_E we assumed the existence of a function pp_VE which preprocesses the patterns. It takes a set of patterns, and returns an association list in which each element x of type FE a Int is associated to an integer. This integer denotes the matchset that matches at the point where the matchsets represented by the integers in x (in case the first component is of the form In32 (x,y) or In33 (x,y)) match at the subtrees.

Function pp_VE is obtained by replacing L by VE in function pp_L, and by adding the pattern VarE to the set of suffixes of patterns.

```
pp_VE :: (Eq a, Set [a], Set [VE a]) =>
         [VE a] -> Table_VE a
pp_VE ps =
  let s = pats_VE ps
  in table_VE (elts_VE ps) (matchsets_VE s) s

pats_VE :: Set [VE a] => [VE a] -> [VE a]
pats_VE ps = [VarE] `union` bigunion pss
  where pss = map (flatten_LVE . suf_VE) ps
```

Just as functions suf_L and suf_E, function suf_VE is defined as a scan of the constructors.

```
suf_VE  :: VE a -> LVE (VE a)
suf_VE  =  scan_VE VCon VSum VProd VarE
```

Function elts_VE is obtained by replacing L by VE in elts_L, and flatten_LVE is an obvious generalisation of flatten_L.

Matchsets

Functions matchsets_VE and ismatchset_VE are obtained by replacing L by VE in their equivalents on the datatype

L. Function match_VE is the trivial adjustment of function match_E to the following type.

```
match_VE  :: Eq a => (VE a, VE a) -> Bool
```

Function samems_VE determines whether or not two trees should be in the same matchset. Two trees are in the same matchset if one of them matches the other, or if they are *independent*. Two expression trees p and q are independent if there exists trees x, y, and z such that p matches at x but q not, q matches at y but p not, and, finally, both p and q match at z. For example, the trees VSum (VarE,VCon 1) and VSum (VCon 2,VarE) are independent, and the trees VProd (VarE,VCon 2) and VProd (VCon 1,VCon 2) are not independent. Function samems_VE, which determines whether or not two patterns belong to the same matchset, is expressed in terms of function match_VE and ind_VE. Function ind_VE defined below is a bit more liberal than the definition of independent trees given above (it also returns True if one of the arguments matches the other), but since it is only called on arguments p and q that do not match each other, it has the same effect.

```
samems_VE :: Eq a => (VE a,VE a) -> Bool
samems_VE (p,q) =  match_VE (p,q)
                || match_VE (q,p)
                || ind_VE (p,q)

ind_VE :: Eq a => (VE a,VE a) -> Bool
ind_VE (VCon a,VCon b)             = False
ind_VE (VSum (x,y),VSum (s,t))     = i (x,y) (s,t)
ind_VE (VProd (x,y),VProd (s,t)) = i (x,y) (s,t)
ind_VE (_,_)                       = False

i (x,y) (s,t) =
     match_VE (x,s) && match_VE (t,y)
  || match_VE (s,x) && match_VE (y,t)
  || samems_VE (x,s) && samems_VE (y,t)
```

Note that we can give an equivalent definition of ind_L on the datatype L, but then no two lists are independent.

Constructing the table

Function table_VE constructs the table used in function pm_E. To construct the table we consider all possibilities of combining matchsets matching at subtrees, and determine the matchset matching at the Sum or Prod of those subtrees. First we define the function combining matchsets matching at subtrees, or, if there are no subtrees, possible elements occurring at a VCon. The structure of function gen_E differs slightly from that of gen_L, because the version of gen_L we obtain using the general construction has been simplified.

```
gen_E :: [a] -> [b] -> [FE a b]
gen_E as ns =
  concat (map cross_E (inject_E as ns))

inject_E :: a -> b -> [FE a b]
inject_E as ms =
  [In31 as,In32 (ms,ms),In33 (ms,ms)]

cross_E :: Sum3 [b] ([c],[d]) ([e],[f]) ->
           [Sum3 b (c,d) (e,f)]
cross_E (In31 as) = [In31 a | a <- as]
cross_E (In32 (ms,ns)) =
  [In32 (m,n) | m <- ms, n <- ns]
cross_E (In33 (ms,ns)) =
  [In33 (m,n) | m <- ms, n <- ns]
```

For each of the possibilities generated by gen_E, function table_VE determines the new matchset, and returns the pair consisting of a possible argument and a number denoting the new matchset.

```
table_VE :: (Eq a, Set [VE a]) =>
            [a] -> [(Int,[VE a])] -> [VE a] ->
            Table_VE a
table_VE as ms s =
  let cd g = code ma_E ms ([VarE] `union` h')
         where h' = h (d g)
      h = filter (`elem` s).map in_VE.cross_E

      d (In31 a)     = In31 [a]
      d (In32 (x,y)) =
        In32 (decode [] ms x,decode [] ms y)
      d (In33 (x,y)) =
        In33 (decode [] ms x,decode [] ms y)
  in [(g, cd g) | g <- gen_E as (map fst ms)]
```

Note that two kinds of functions can be distinguished in the pattern-matching programs on L a and E a: the ones that refer explicitly to the structure of the datatype, such as cata, match, and ind, and the ones that just use the name of the datatype, such as pm, elts, and matchsets.

3 Polytypic pattern matching

This section specifies and derives the PPM algorithm. The datatypes L a and E a in the definitions of the pattern-matching programs in the previous section are replaced by an arbitrary datatype. We reuse the names used in the pattern-matching programs on lists and expression trees in the description of the PPM algorithm. So for example, functions *table* and *matchsets* will be the polytypic versions of functions table_E and matchsets_E, respectively.

The previous section shows that we use several other polytypic functions, such as a polytypic scan and polytypic equality in the description of a PPM algorithm. The second subsection briefly describes the polytypic functions we use for specifying and constructing the PPM algorithm. To define the polytypic functions we use many constructs from Constructive Algorithmics or the Bird-Meertens calculus [21, 18, 22]. We lack the space to define all polytypic functions, so we will restrict ourselves to giving the types of the polytypic functions, and to define a number of polytypic functions at the beginning of the section on polytypic functions. Most of the definitions can be found in the literature on Constructive Algorithmics. We suppose the reader is familiar with this approach in which datatypes are defined as initial objects of categories of functor-algebras. We briefly introduce the notation in the first subsection. The third subsection specifies the PPM problem, and the fourth subsection derives the PPM algorithm.

We use category theory to describe the polytypic functions. We could have used the code used to define polytypic functions in the polytypic programming preprocessor, but since writing polytypic functions using this preprocessor is still quite cumbersome, we refrained from doing so.

3.1 Preliminaries

In the sequel we will use functions *map*, *or*, *bigunion*, etc., on sets instead of lists. The definitions of these functions on set are very similar to their definitions on list, and are therefore omitted, see [20].

Sums and products

Though Haskell is based in CPO, we assume for simplicity that we are working in a language like Set. We suppose the language we are working in has sums and products, that is, given types a and b, the types $a+b$ and $a\times b$ exist. Furthermore, we have the usual projection functions fst and snd and injection functions inl and inr. Multiplication distributes over addition; for example, we have a function $\mathit{distleft} : a\times(b+c) \to a\times b+a\times c$. Given functions $f : a \to c$ and $g : b \to c$ the case construct $[f,g] : a+b \to c$ applies f to left-tagged values, and g to right-tagged values, and discards the tag information.

Functors and datatypes

In what follows we assume that F is a bifunctor such that the category of $F\ (a,\_)$-algebras has an initial object. Here, functor $F\ (a,\_)$ is obtained by fixing the first argument of F.

$$\begin{array}{rcl} F\ (a,\_)\ x & = & F\ (a,x) \\ F\ (a,\_)\ f & = & F\ (id_a,f) \end{array}$$

The initial object in the category of $F\ (a,\_)$-algebras consists of a datatype $D\ a$ and a bijective morphism $\mathit{in} : F\ (a, D\ a) \to D\ a$ with inverse out. For example, the datatype `L a` corresponds to the initial object of the category of $G\ (a,\_)$-algebras, and the datatype `E a` corresponds to the initial object of the category of $H\ (a,\_)$-algebras, where functors G and H are defined by

$$\begin{array}{rcl} G\,(a,x) & = & 1+a\times x \\ G\,(a,f) & = & id_1+id_a\times f \end{array}$$

$$\begin{array}{rcl} H\ (a,x) & = & a+x\times x+x\times x \\ H\ (a,f) & = & id_a+f\times f+f\times f \end{array}$$

The functions in from the initial algebras for these functors are encodings of the constructors: $in_{G\ (a,\_)} = [\mathtt{Nil},\mathtt{Cons}]$, and $in_{H\ (a,\_)} = [\mathtt{Con},\mathtt{Sum},\mathtt{Prod}]$.

From initiality of $(D\ a, \mathit{in})$ we obtain the following characterisation of catamorphisms. For $f : F\ (a,b) \to b$ we have $\mathit{cata}\ f : D\ a \to b$ with

$$h = \mathit{cata}\ f \quad\equiv\quad h\cdot \mathit{in} = f\cdot F\ (a,h) \tag{1}$$

It follows from the characterisation of catamorphisms that:

$$\mathit{cata}\ f \quad=\quad f\cdot F\ (a, \mathit{cata} f)\cdot \mathit{out}$$

In fact, this is how instances of cata are defined in the polytypic programming preprocessor. We try to avoid using pattern matching in the definitions of polytypic functions. So instead of using pattern matching at the left-hand side of the definition, we use the 'destructor' function out at the right-hand side of the definition.

3.2 Polytypic functions

Cross and cp

In the construction of the association list in the preprocessing phase we use a function gen (functions `gen_L` and `gen_E` in the previous section). In the definition of this function we use function cross of the following type.

$$\mathit{cross} \quad : \quad F\ (\{a\},\{b\}) \to \{F\ (a,b)\}$$

where F is a bifunctor. Function `cross_E` defined in Section 2 corresponds with cross_H, where functor H is defined above. To give an example of a polytypic definition, we define function cross. A definition in category theory can be found in [23, 24]. A polytypic function like cross is defined by induction on the structure of regular (bi)functors. A regular functor is either a constant functor, the projection on the first or second component, the sum of two functors, the product of two functors, or the composition of a type functor (the action of a type functor D induced by a regular functor F on a type a is the type $D\ a$; its action on arrows is the map on the type $D\ a$, see [17]) with a functor. The following grammar generates regular functors.

$$F \quad ::= \quad a \mid \mathit{fst} \mid \mathit{snd} \mid F+F \mid F\times F \mid DF$$

Function cross is defined by induction on regular bifunctors.

$$\begin{array}{rcl} \mathit{cross}_a\ x & = & \{x\} \\ \mathit{cross}_{\mathit{fst}}\ x & = & x \\ \mathit{cross}_{\mathit{snd}}\ x & = & x \\ \mathit{cross}_{F+G}\ x & = & [\mathit{map}\ \mathit{inl}\cdot \mathit{cross}_F, \mathit{map}\ \mathit{inr}\cdot \mathit{cross}_G]\ x \\ \mathit{cross}_{F\times G}\ (x,y) & = & \{(s,t) \mid s\leftarrow \mathit{cross}_F\ x, t\leftarrow \mathit{cross}_G\ y\} \\ \mathit{cross}_{DF}\ x & = & \mathit{cp}_D\ (D \mathit{cross}_F\ x) \end{array}$$

Function cp is defined for a type functor induced by a regular bifunctor.

$$\begin{array}{rcl} \mathit{cp} & : & D\ \{a\} \to \{D\ a\} \\ \mathit{cp}_D & = & \mathit{cata}\ (\mathit{map}\ \mathit{in}_{F\ (a,\_)}\cdot \mathit{cross}_F) \end{array}$$

Fl and flatten

Datatypes are used to structure information. Sometimes we are not interested in the structure, but just in the elements stored in the structure. For that purpose we define functions fl and $\mathit{flatten}$ of type

$$\begin{array}{rcl} \mathit{fl} & : & F\ (a,\{a\}) \to \{a\} \\ \mathit{flatten} & : & D\ a \to \{a\} \end{array}$$

for bifunctor F and monofunctor D. In Section 2 we have defined function $\mathit{flatten}$ for the typefunctor `L` of lists. These functions are defined similarly to functions cross and cp, respectively. Function fl is defined by induction on regular bifunctors.

$$\begin{array}{rcl} \mathit{fl}_a\ x & = & \{\ \} \\ \mathit{fl}_{\mathit{fst}}\ x & = & \{x\} \\ \mathit{fl}_{\mathit{snd}}\ x & = & x \\ \mathit{fl}_{F+G}\ x & = & [\mathit{fl}_F, \mathit{fl}_G]\ x \\ \mathit{fl}_{F\times G}\ (x,y) & = & \mathit{fl}_F\ x\ \cup\ \mathit{fl}_G\ y \\ \mathit{fl}_{DF}\ x & = & \mathit{bigunion}\ (\mathit{flatten}_D\ (D\mathit{fl}_F\ x)) \end{array}$$

Function $\mathit{flatten}$ is defined for a type functor D induced by a regular bifunctor F, just as function cp.

$$\mathit{flatten}_D \quad = \quad \mathit{cata}\ \mathit{fl}_F$$

Using function $\mathit{flatten}$ we can separate the data and shape of an element of a datatype. Datatypes of which the elements can be separated in a data part and a shape part are called shapely types by Jay and Cockett [12].

Scan

We give just the type of polytypic scan. Two examples of scan, `scan_L` and `scan_E`, have been given in Section 2. A precise definition is given in [3]. First we define the labelled type $LD\ a$ corresponding to the type $D\ a$. This labelled datatype is the initial $J\ (a, \_)$-algebra, where functor J is defined by

$$J\ (a, x) \quad = \quad F\ (1, x) \times a$$

Since we assume that $\times$ distributes over $+$, the type we obtain if F is the functor H (which induces the type of expression trees `E`) is isomorphic to the type `LE` given in the previous section. We use the datatype $LD\ a$ for the result type of function *scan*.

$$scan \quad : \quad (F\ (a, b) \to b) \to D\ a \to LD\ b$$

All suffixes of a value of the datatype $D\ a$ are obtained by applying function *suf* to it. Function *suf* is defined by

$$\begin{array}{rcl} suf & : & D\ a \to LD\ (D\ a) \\ suf & = & scan\ in \end{array}$$

The following equality for *suf* is easily proved [3].

$$LD\ (cata\ f) \cdot suf \quad = \quad scan\ f \qquad (2)$$

Equality

Given an equality $(=)_a$ on the type a, and a datatype $D\ a$ we have an equality function $(=)_D : D\ a \times D\ a \to Bool$. This equality is defined in terms of a polytypic *zip* function, a generalisation of the `zip` function defined on lists. The definition of function *zip* is omitted. Function *zip* returns a set that contains the zip of its two arguments if the arguments can be zipped, and that is empty otherwise:

$$\begin{array}{rcl} zip & : & F\ (a, b) \times F\ (c, d) \to \{F\ (a \times c, b \times d)\} \\ (=)_D & = & or \cdot map\ (and_F \cdot F\ ((=)_a, (=)_D)) \cdot zip \cdot out \times out \end{array}$$

The pair of functions $out \times out$ corresponds with the fact that we try to avoid using pattern matching in definitions of polytypic functions. Functions *or* and *map*... encode the fact that if *zip* returns the empty set, $(=)_D$ should return *False*. Function and_F is defined in terms of the functions $fl1 : F\ (a, b) \to \{a\}$ and $fl2 : F\ (a, b) \to \{b\}$, which flatten an F structure with respect to the first, and the second component, respectively. The definitions of these functions are very similar to the definition of function *fl*, and therefore omitted.

$$and_F\ x \quad = \quad and\ (fl1_F\ x\ \cup\ fl2_F\ x)$$

Adding a variable to a datatype

Given a datatype $D\ a$ we construct the datatype $VD\ a$ values of which are values of $D\ a$ which may contain zero or more occurrences of a variable. In the datatype of expressions `E` in Section 2 the datatype `VE` of expressions with variables is an extension of `E` with an extra nullary constructor representing the variable. We translate this approach to the categorical approach to modelling datatypes. Given a datatype $D\ a$ the datatype with variables is the initial $K\ (a, \_)$-algebra, where functor $K\ (a, \_)$ is defined by

$$K\ (a, x) \quad = \quad F\ (a, x) + 1$$

Let $(VD\ a, in')$ be the initial $K\ (a, \_)$-algebra, and let $()$ be the object of the type 1. Then the variable *var* is defined by

$$var \quad = \quad in'\ (inr\ ())$$

Matching

Given a datatype $D\ a$, we define two matching functions:

$$\begin{array}{rcl} match_D & : & D\ a \times VD\ a \to Bool \\ matchV_{VD} & : & VD\ a \times VD\ a \to Bool \end{array}$$

The first of these is needed in the specification of polytypic pattern matching, the second is needed in the final algorithm. We define just the first matching function.

$$\begin{array}{rcl} match_D & = & [l, const\ True] \cdot distleft \cdot out_D \times out_{VD} \\ & & \textbf{where} \\ & & l = or \cdot map\ (and_F \cdot F\ ((=)_a, match_D)) \cdot zip \end{array}$$

Note the similarity with the polytypic equality function. The only difference is the occurrence of functions *distleft* and *const True*. These functions encode the fact that a variable matches any value.

3.3 Specifying polytypic pattern matching

We specify the PPM problem in three phases.

Given a set of patterns, the pattern-matching problem requires finding for each suffix the subset of patterns matching at that structure.

$$\begin{array}{rcl} pm & : & \{VD\ a\} \to D\ a \to LD\ \{VD\ a\} \\ pm\ ps & = & LD\ (fm\ ps) \cdot suf \end{array}$$

$$\begin{array}{rcl} fm & : & \{VD\ a\} \to D\ a \to \{VD\ a\} \\ fm\ ps\ t & = & filter\ (\backslash p \to match_D\ (t, p))\ ps \end{array}$$

To allow for an incremental computation of the sets of patterns matching at a suffix, we change the first argument of *fm* in the definition of *pm*. The set of patterns *ps* is replaced by the set of patterns obtained by taking the union of all suffixes of the patterns in *ps*.

$$pm\ ps \quad = \quad LD\ (fm\ (pats\ ps)) \cdot suf$$

$$\begin{array}{rcl} pats & : & \{VD\ a\} \to \{VD\ a\} \\ pats\ ps & = & \{var\}\ \cup\ bigunion\ (map\ (flatten \cdot suf)\ ps) \end{array}$$

Each set of patterns matching at a suffix is a matchset of *ps*. We can precompute the matchsets of the set of patterns *pats ps*, and encode these as numbers. Function *pm* returns now a structure containing numbers, which denote matchsets. Suppose function *matchsets* returns the matchsets. The definitions of the functions *ismatchset*, *samems*, and *ind*, used in the definition of function *matchsets*, are omitted.

$$\begin{array}{rcl} pm & : & \{VD\ a\} \to D\ a \to LD\ Int \\ pm\ ps & = & LD\ (code\ ma\ ms \cdot fm\ s) \cdot suf \qquad (3) \\ & & \textbf{where}\ \ s\ =\ pats\ ps \\ & & \qquad\quad ms\ =\ matchsets\ s \end{array}$$

$$matchsets \quad : \quad \{VD\ a\} \to [(Int, \{VD\ a\})]$$

where function $code : a \to [(a, b)] \to b \to a$ and function $decode : b \to [(a, b)] \to a \to b$ are defined as the functions with the same name in Section 2. The value *ma* is an integer denoting the matchset $\{var\}$. From now on we will omit the default argument of functions *code* and *decode*. This concludes the specification of polytypic pattern matching.

3.4 Deriving a PPM algorithm

Given a datatype, specification (3) of function pm can be implemented on that datatype, but the result is a very inefficient program for pattern matching. In this subsection we derive a program for pattern matching that is often more efficient. The calculation starts with definition (3) of pm, and uses the equalities given for $cata$ (1) and $scan$ (2) to derive a scan for function pm. Since the laws applied in the derivation are correctness preserving, the resulting program is a correct implementation of function pm.

Deriving a $scan$ for pm

We apply equality (2) to obtain a scan for function pm. Suppose function $code\ ms \cdot fm\ s$ can be written as $cata\ (cd\ ms\ s)$ with $cd\ ms\ s : F\ (a, Int) \to Int$. Then we have

$$pm\ ps \quad = \quad scan\ (cd\ ms\ s)$$

If function $cd\ ms\ s$ can be computed in constant time, this program requires time $O(n)$ on an input of size n (where the size of a value is the number of constructors with which it is built).

Deriving a catamorphism for $code\ ms \cdot fm\ s$

To obtain a $scan$ for function pm, we want to construct a function cd such that

$$code\ ms \cdot fm\ s \quad = \quad cata\ (cd\ ms\ s)$$

The definition of cd is obtained by applying the characterisation of catamorphisms, equality (1). Function cd has to satisfy

$$code\ ms \cdot fm\ s \cdot in \quad = \quad cd\ ms\ s \cdot F\ (a, code\ ms \cdot fm\ s)$$

Function fm satisfies the following equality, which shows that it is a catamorphism. This equality states that the set of patterns matching a value $in\ t$ can be obtained by first matching the set of patterns against the subcomponents t of the value, and then combining these sets of patterns.

$$\begin{aligned} fm\ s\ (in\ t) \quad &= \quad (h \cdot F\ (a, fm\ s))\ t\ \cup\ \{var\} \\ &\quad\ \ \textbf{where} \\ &\quad\ \ h = filter\ (\in s) \cdot map\ (in' \cdot inl) \cdot cross_{F\ (a,\_)} \end{aligned} \tag{4}$$

Note that we use a $cross$ on a monofunctor here. Function $cross_{F\ (a,\_)}$ is defined in terms of function $cross_F$ by:

$$\begin{aligned} cross_{F\ (a,\_)} \quad &: \quad F\ (a, \{b\}) \to \{F\ (a, b)\} \\ cross_{F\ (a,\_)} \quad &= \quad cross_F \cdot F\ (\{\cdot\}, id) \end{aligned}$$

where function $\{\cdot\}$ is the singleton set function: $\{\cdot\}\ a = \{a\}$. The proof of equality (4), by mutual inclusion, is omitted. Using the fact that $decode$ is the inverse of $code$, and $F\ id = id$ for a functor F, we have

$$id \quad = \quad F\ (a, decode\ ms) \cdot F\ (a, code\ ms) \tag{5}$$

We use the above equalities to construct a catamorphism for function $code\ ms \cdot fm\ s$.

$$\begin{aligned} & code\ ms\ (fm\ s\ (in\ t)) \\ = \quad & \quad \text{equality (4) for } fm \\ & code\ ms\ ((h \cdot F\ (a, fm\ s))\ t\ \cup\ \{var\}) \\ = \quad & \quad \text{equality (5)} \\ & code\ ms\ (h\ (F\ (a, decode\ ms)\ (j\ t))\ \cup\ \{var\}) \end{aligned}$$

where $j = F\ (a, code\ ms \cdot fm\ s)$. Thus we have expressed value $code\ ms\ (fm\ s\ (in\ t))$ in terms of $F\ (a, code\ ms \cdot fm\ s)\ t$, and it follows that $code\ ms \cdot fm\ s = cata\ (cd\ ms\ s)$ where function cd is defined by

$$\begin{aligned} cd\ ms\ s\ t \quad &= \quad code\ ms\ ((h \cdot F\ (a, decode\ ms))\ t\ \cup\ \{var\}) \\ &\quad\ \ \textbf{where} \\ &\quad\ \ h = filter\ (\in s) \cdot map\ (in' \cdot inl) \cdot cross_{F\ (a,\_)} \end{aligned}$$

Function cd is extremely inefficient. Each time cd is evaluated, sets of patterns are decoded, all possible combinations of the patterns are constructed and tested on membership of s, and subsequently coded.

Preprocessing the patterns

We obtain a faster algorithm by preprocessing the patterns. When function cd is used to determine the value of an application of function pm, the recursive components of its third argument t are coded matchsets. Since the result of function $code$ is a number between 0 and the number of matchsets, and all values of type a that do not occur in a pattern cannot be used to build new matching patterns, there are a limited number of different values t can have. We can precompute these values by means of a function gen, determine the result of function cd on these values, and store the results in a table. We subsequently replace $cd\ ms\ s$ by function $decode\ tab$ in the definition of pm. Function pp takes care of preprocessing the patterns.

$$\begin{aligned} pm\ ps \quad &= \quad scan\ (decode\ tab) \\ &\quad\ \ \textbf{where}\ \ tab\ =\ pp\ ps \end{aligned}$$

$$\begin{aligned} pp \quad &: \quad \{VD\ a\} \to [(F\ (a, Int), Int)] \\ pp\ ps \quad &= \quad table\ (elts\ ps)\ (matchsets\ s)\ s \\ &\quad\ \ \textbf{where}\ \ s\ =\ pats\ ps \end{aligned}$$

$$\begin{aligned} table \quad &: \quad \{a\} \to [(Int, \{VD\ a\})] \to \{VD\ a\} \to \\ &\quad\ \ [(F\ (a, Int), Int)] \\ table\ as\ ms\ s \quad &= \quad [(g, cd\ ms\ s\ g) \\ &\quad\ \ \mid g \leftarrow gen\ as\ (map\ fst\ ms) \\ &\quad\ \ ] \end{aligned}$$

It remains to define the polytypic functions gen and $elts$. Function $elts$ is a simple application of function $flatten$. Function gen injects two sets of values into a set containing F structures over these sets, and subsequently applies $cross$ to obtain a set of F structures over the values in these sets.

$$\begin{aligned} elts \quad &: \quad \{VD\ a\} \to \{a\} \\ elts\ ps \quad &= \quad bigunion\ (map\ flatten\ ps) \end{aligned}$$

$$\begin{aligned} gen \quad &: \quad \{a\} \to \{Int\} \to \{F\ (a, Int)\} \\ gen\ as\ ns \quad &= \quad bigunion\ (map\ cross_F\ (inject\ as\ ns)) \end{aligned}$$

The definition of function $inject$ of type

$$inject \quad : \quad \{a\} \to \{b\} \to \{F\ (\{a\}, \{b\})\}$$

is omitted.

4 Conclusions

We have constructed a PPM algorithm. The algorithm is a generalisation of the bottom-up tree pattern-matching algorithm from Hoffmann and O'Donnell [9]. Each instantiation of the PPM algorithm results in a program that matches a set of patterns *ps* in a structure *t* in time linear in the size of *t*. However, the preprocessing phase, which preprocesses the set of patterns *ps*, might require exponential time. For applications in which the set of patterns does not change frequently, for example in transformational programming systems, this need not be a problem, since we can apply partial evaluation to obtain a linear-time program. For other applications the exponential time preprocessing phase is unacceptable, and for these applications the existing techniques for improving the performance of the preprocessing phase of bottom-up tree pattern matching [7] should be generalised to polytypic pattern matching. Future research will consist of implementing these preprocessing techniques, and of further improvements and extensions to the current code.

The work on polytypic pattern matching is part of ongoing research on polytypic functions. Future research will consist of both constructing more polytypic functions, for example for drawing arbitrary trees tidily and unification, and designing a practical programming language in which polytypic functions can be written. We are currently working on extending the `deriving` construct in a Haskell compiler with the possibility to derive polytypic functions.

Acknowledgements

The polytypic programming preprocessor has been built together with Graham Hutton and Oege de Moor. Graham Hutton and Patrik Jansson commented on a previous version of this paper. Oege de Moor had a great influence on the research reported on in this paper, both with his previous work and his visit to Chalmers.

References

[1] A.V. Aho and M.J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] T.P. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM Journal on Computing*, 7(4):533–541, 1978.

[3] Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic programming with relations and functors. Submitted for publication, 1993.

[4] R.S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, 1977.

[5] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 5–42. Springer–Verlag, 1987.

[6] R.S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, volume F55 of *NATO ASI Series*, pages 151–216. Springer–Verlag, 1989.

[7] J. Cai, R. Paige, and R. Tarjan. More efficient bottom-up tree pattern matching. In *Proceedings* 15^{th} *Colloquium on Trees in Algebra and Programming*, pages 72–86, 1990. LNCS 431.

[8] T. Hagino. *Category Theoretic Approach to Data Types*. PhD thesis, University of Edinburgh, 1987.

[9] C.M. Hoffmann and M.J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1982.

[10] James Hook and Tim Sheard. A semantics of compile-time reflection. Oregon Graduate Institute of Science and Technology, Beaverton, OR, USA, 1993.

[11] P. Hudak, S.L. Peyton Jones, and P. Wadler (eds.). Report on the programming language Haskell, version 1.2. *ACM SIGPLAN notices*, 27(5), May 1992.

[12] C. Barry Jay and J.R.B. Cockett. Shapely types and shape polymorphism. In Donald Sannella, editor, *Proceedings Programming Languages and Systems-ESOP '94*, pages 302–316. Springer-Verlag, 1994. LNCS 788.

[13] J. Jeuring. Algorithms from theorems. In M. Broy and C.B. Jones, editors, *Programming Concepts and Methods*, pages 247–266. North-Holland, 1990.

[14] J. Jeuring. The derivation of a hierarchy of algorithms for pattern matching on arrays. In G. Hains and L.M.R. Mullin, editors, *Proceedings ATABLE-92, Second international workshop on array structures*, pages 199–213, 1992. DIRO publication number 841, Université de Montréal.

[15] J. Jeuring. Polytypic combinatorial functions. Unpublished manuscript, 1994.

[16] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1978.

[17] G. Malcolm. *Algebraic data types and program transformation*. PhD thesis, University of Groningen, 1990.

[18] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[19] E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics*. Text and Monographs in Computer Science. Springer Verlag, 1986.

[20] L. Meertens. Algorithmics—towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334. North–Holland, 1986.

[21] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.

[22] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts*, pages 124–144, 1991.

[23] O. de Moor. *Categories, relations and dynamic programming*. PhD thesis, Oxford University, 1992. Technical Monograph PRG-98.

[24] O. de Moor. Categories, relations and dynamic programming. *Mathematical Structures in Computer Science*, 4:33–69, 1994.

[25] J. Peterson. Dynamic typing in Haskell. Technical Report YALEU/DCS/RR-1022, Yale University, Department of Computer Science, 1993.

[26] T. Sheard. Type parametric programming. Oregon Graduate Institute of Science and Technology, Portland, OR, USA, 1993.