Polytypism and polytypic unification

Patrik Jansson

Chalmers University of Technology, 1995

Master's Thesis in Computing Science

Supervisor: Johan Jeuring

Abstract

This report describes what polytypic programming is, a new system for writing polytypic functions, and a number of useful example functions including generalised versions of map, zip and a specific lazy array based unification algorithm.

Sammanfattning

Denna rapport beskriver hur man med ett programsystem automatiskt kan generera funktioner som fungerar för alla trädtyper och detta systems tillämpning på ett antal användbara funktioner som map, zip och en speciell unifieringsalgoritm baserad på lata fält.

Polytypism and polytypic unification

1 Introduction 3
- 1.1 Background 3
- 1.2 Preliminaries and notation 4
- 1.3 Overview 4

2 Polytypism 5
- 2.1 Map 5
- 2.2 Cata 5
- 2.3 Functors 7

3 Basic polytypic functions 10
- 3.1 Predefined functions and types 10
  - 3.1.1 Products 10
  - 3.1.2 Sums 11
- 3.2 Notation 11
- 3.3 Object and arrow 12
- 3.4 In and out 13
- 3.5 Map 14
- 3.6 Cata 15
- 3.7 Ana 16
- 3.8 Hylo 16
- 3.9 Flatten 17
- 3.10 Partial functions 18
- 3.11 Zip 19

4 Program construction combinators 20
- 4.1 A combinator example 20
- 4.2 Basic building blocks 20
- 4.3 Binary combinators 20
- 4.4 Other combinators 21

5 System 22
- 5.1 Overall structure 22
- 5.2 Constructing a functor for a datatype 22
  - 5.2.1 Representation of types 23
  - 5.2.2 Mutually recursive datatypes 23
  - 5.2.3 Functorize 24
- 5.3 Implementing the combinators 24
  - 5.3.1 Types for expressions 24
  - 5.3.2 Basic building blocks 25
  - 5.3.3 Binary combinators 25
  - 5.3.4 Other combinators 25
- 5.4 Function generators 25
- 5.5 Simplification 26

6 Unification 27
- 6.1 Introduction to unification 27
- 6.2 Definitions 27
- 6.3 Outline of a unification algorithm 28
- 6.4 Unification with lazyArray 28
- 6.5 Polytypic unification 30

7 Conclusions 32

8 References 33

Appendix A - example of generated code

Appendix B - prelude

1 Introduction

This report describes polytypism, unification and a specific array based polytypic unification algorithm. We will also describe a system for generating code for polytypic functions.

1.1 Background

As an example of a polytypic[1] function we can take unification. Unification is the process of making two expressions containing variables equal by substituting expressions for the variables. Unification is used for type inference by unifying type expressions containing type variables, in automatic proof systems by unifying proofs with proof methods and in compilers to unify the patterns of function definitions with function calls [10]. Each of these applications requires a unification algorithm for their specific type. As the unification algorithm is so widely used it would be nice to have a general algorithm for unification parametrised by a datatype. This means that given a datatype, it automatically generates a unification algorithm for this type. This is a typical polytypic algorithm and we will give an implementation of it in section 6.5.

Interest in polytypic functions arose when it became clear that most of the theory of lists [1] could be generalised to other datatypes [11]. This has theoretical interest as many of the powerful methods of calculating programs from specifications, and thereby proving their correctness, that had been developed for list based programs [5] also could be generalised [2].

The practical interest comes mainly from the prospects of not having to rewrite all those standard functions that are defined for lists in most functional languages for every datatype in a program. After having written a number of versions of the function map on different tree types one realises that with a suitable theory these should be possible to generate automatically. The categorial theory of datatypes [13], where datatypes are initial fixpoints of functors, provides a sound mathematical basis for defining and reasoning about polytypic functions. One actual implementation of a system for automatic generation of polytypic functions, Hollum, was written by Jeuring et. al. [7].

Hollum reads a Haskell program, parses it, extracts the datatype definitions, generates a number of polytypic functions for each of the types and outputs the result as a Haskell program. The system is self contained but does not handle all tree types as it can not generate code for mutual recursive functions, and also requires that all constructors in the types have exactly one argument (though that argument may be a tuple). The fact that this last requirement is caused by the parser used, and the difficulty in analysing and optimizing the output (which is text) made me interested in

1. Also called type parametric or generic function.

writing a system that only works from and to parsed programs as there are already enough parsers and compilers to take care of the rest.

1.2 Preliminaries and notation

We will use the purely functional lazy programming language Haskell [4] in the examples, but they would probably work with minor modifications in other functional languages as ML or Scheme. All code in the system as well as all the code generated by the system runs in Gofer [9], a language that includes most of Haskell and has a fast interpreter which has been used in the development of the system.

In Haskell (and Gofer), function composition is denoted by a period and has lower precedence than application. (Application is as usual written with juxtaposition) This means that `(fst . map f . sort) l` will be interpreted as `fst (map f (sort l))`. Anonymous functions (often called λ-expressions) are written `(\ args -> exp)` so that the definition `f x y = 1+x+y` is equivalent to `f = (\x y -> 1+x+y)`. Infix operators, such as + can be made prefix by wrapping them in parenthesis; `(+)`, and a normal function `func` taking at least two argument can be made infix by using single backquotes; `` `func` ``. Comments are written as; `{- comment section, possibly many lines long -}`
`-- line comment, extends to the end of the line`
Expressions can be explicitly typed by writing `exp :: type`.

We will write `const :: a -> b -> a`, where `const x y = x` for the constant function. (Often written with just K but in Haskell words starting with capital letters are type constructors.) We will use the operator `(++) :: [a] -> [a] -> [a]` to concatenate two lists and `concat :: [[a]] -> [a]` to concatenate a list of lists.

1.3 Overview

Section 2 introduces polytypism and gives some simple examples of polytypic functions. Section 3 describes a number of polytypic functions that are generated by the system in more detail. In section 4 we show program construction combinators with which the code for polytypic functions can be built. Section 5 describes a system for generating polytypic functions and its implementation. In section 6 we describe unification and we generalise a specific lazy unification algorithm to a polytypic unification algorithm. We end the report in section 7 with some concluding remarks.

In appendix A we show all the code generated by the system for two directly recursive types and one pair of mutually recursive types. The prelude in appendix B contains some common functions used by the functions defined in appendix A.

2 Polytypism

A polytypic program is a program that works for types of different structures. There are at least two other kinds of polymorphism in functional programming. Normal (parametric) polymorphic functions that can be written in most functional languages work for a class of types having the same structure but with different values of the type variables. An example hereof is `length` which calculates the length of a list, no matter what type the list elements have. Overloading, or ad-hoc-polymorphism, is yet another way of making (apparently) the same function work for a class of types, but here one definition of the function has to be supplied for every type in the class. A typical example of an often overloaded function is + which normally works both for integers and floating point numbers.

A polytypic function has one definition that works even for types of different structure. It can but need not be polymorphic in the traditional sense; the function `alleven` that returns true if and only if all integers in a structure are even is polytypic, but not polymorphic.

In the following two sections we will give some examples of polytypic functions.

2.1 Map

One simple example of a polytypic function is the generalisation of `map`. The commonly used `map_L` (normally called just `map`) on lists takes a function `f` and applies it to all elements (if any) in the list. The essence here, which can be generalised to other types than lists, is that `map` takes a function and applies it to all elements of a data structure without changing the form of the structure. In figure 1 and 2 we give the definitions of map for two simple types. Note the similarity between the type definitions and the function definitions.

```
data List a =            map_L :: (a -> b) -> List a -> List b
       Nil               map_L f Nil         = Nil
     | Cons a (List a)   map_L f (Cons x xs)= Cons (f x) (map_L f xs)
```

FIGURE 1. The definition of map for lists.

```
data Tree a =                 map_T :: (a -> b) -> Tree a -> Tree b
       Leaf a                 map_T f (Leaf x)  = Leaf (f x)
     | Bin (Tree a) (Tree a)  map_T f (Bin l r) = Bin (map_T f l) (map_T f r)
```

FIGURE 2. The definition of map for simple binary trees with information in the nodes only.

2.2 Cata

A more general and very powerful polytypic function is `cata`[1]. On lists `cata_L` is normally called `foldr` (fold right) and takes a start value, an operator and a list and

1. cata abbreviates catamorphism

inserts the operator between the elements of the list, with the start value on the far right end. (See figure 3)

```
data List a =               cata_L :: b -> (a -> b -> b) -> List a -> b
       Nil                  cata_L e op Nil         = e
     | Cons a (List a)      cata_L e op (Cons x xs) = x `op` (cata_L e op xs)
```

FIGURE 3. The definition of cata for lists

Some simple examples of functions defined by list-catamorphisms are:

```
sum_L :: List Int -> Int
sum_L   = cata_L 0 (+)
all_L :: (a -> Bool) -> List a -> Bool
all_L p = cata_L True (\x b -> (p x) && b)
```

where `sum_L` computes the sum of a list of numbers and `all_L p` determines if all elements of a list satisfy the predicate `p`.

The key to generalising the definition of `cata` to other tree types is to observe that `cata_L` simply replaces the datatype constructor `Nil` with the supplied constant `e` and the constructor `Cons` with the operator `op`. The recursive occurrence of `List a` in the datatype definition is replaced by a recursive call to `cata_L` in the function definition. In the general case every constructor in the definition of the datatype is replaced by a function of the same arity[1] and recursive datatypes are transformed to recursive functions with exactly the same structure.

Now the definition of `cata_T` is straightforward; we just replace `Leaf` by a function `f` and `Bin` by a binary operator `op`:

```
data Tree a =               cata_T :: (a -> b) -> (b -> b -> b) -> Tree a -> b
       Leaf a               cata_T f op (Leaf x)  = f x
     | Bin (Tree a) (Tree a) cata_T f op (Bin l r) = op (ca l) (ca r)
                                       where ca = cata_T f op
```

FIGURE 4. The definition of the catamorphism for binary trees.

Despite the simplicity of the definition, this is a very sophisticated higher order function with which we can define many other functions. Informally a catamorphism can calculate anything for a structure that can be calculated given the information in the top node and the result of the catamorphism on all substructures.

1. The arity of a function is the number of arguments it takes. We will consider constants to be functions with arity zero.

Some examples are:

```
map_T :: (a -> b) -> Tree a -> Tree b
map_T f   = cata_T (Leaf . f) Bin
  -- This definition is equivalent to the one above.

flatten_T :: Tree a -> [a]
flatten_T = cata_T wrap (++)
             where wrap x = [x]
  -- The structure is flattened to a list.

unzip_T :: Tree (a,b) -> (Tree a,Tree b)
unzip_T   = cata_T (prod (Leaf,Leaf)) (prod . prod (Bin,Bin))
            where prod (f,g) (a,b) =(f a,g b)
  -- Takes a tree of pairs and returns a pair of trees.

size_T :: Tree a -> Int
size_T    = cata_T (const 1) (\l r -> 1+l+r)
  -- Counts the number of constructors.

depth_T :: Tree a -> Int
depth_T   = cata_T (const 0) (\l r -> 1+(max l r))
  -- Calculates the maximal level of the constructors.

leftmost_T :: Tree a -> a
leftmost_T= cata_T id const
  -- returns the leftmost element of the tree.

mirror_T :: Tree a -> Tree a
mirror_T  = cata_T Leaf (flip Bin)
  -- Mirrors the tree in a line through its root.
```

Except for the last one (that depends on the fact that the two arguments to `Bin` are of the same type) all these functions can be generalised to all tree types.

Using a catamorphism for that datatype lots of functions from a datatype to something else can be written. A simple example of a function that is not a catamorphism is `tail` that gives the tail of a list. For `tail` to be a catamorphism we would need a function that could calculate `tail (x:xs)` from `x` and `tail xs` which is clearly impossible since the information about the first element of `xs` is lost.

2.3 Functors

The recursive definitions of the two types in the previous section can be thought of as being the fixpoints with respect to the parameter x of these two types:

```
data FList x a = FNil
               | FCons a x

data FTree x a = FLeaf a
               | FBin x x
```

The structure of these types and the names of the constructors is all that is needed to generate `map`, `cata` and other polytypic functions. When dealing with these functions theoretically it is useful to go one step further and ignore the actual names of the constructors retaining only the structure, as all datatypes that differ only in the names of the constructors are isomorphic. This stripped structure of the type will be called a functor[1].

By ignoring the specific constructor names we get for `List` the functor $L\ x\ a\ =\ 1 + a \times x$, and for `Tree` we get $T\ x\ a\ =\ a + x \times x$. Here the arguments a and x can be thought of as sets, the sum means disjoint union of sets and the product is the cross product.

The concept of functors is essential to the underlying theory of polytypism and almost all polytypic functions are defined either using induction on the structure functors or by combining other polytypic functions. This means that it is very important to have a sufficiently rich structure representing functors, as the polytypic functions we define will work only for the types that we can represent by fixpoints of functors.

In the system functors are represented as elements of the datatype `Func`:

```
data Func a =
       Prod   [Func a]  -- direct product of functors
     | Sum    [Func a]  -- sum of functors
     | Comp a [Func a]  -- composition of a constructor
                        --   with a list of functors
     | Par Int Int      -- type parameter, first the position
                        --   number in the mut. rec. group
                        --   then the local argument number
     | Rec Int          -- recursive parameter
```

For a functor representing a datatype in Haskell the top level is always a `Sum` with a list of functors representing the alternatives, one for each constructor in the type. Each of these alternatives is a `Prod` with a list of the representations of the arguments of the corresponding constructor. For `List` and `Tree` we get:

```
funcList = Sum [Prod [],
                Prod [Par 1 1, Rec 1] ]
funcTree = Sum [Prod [Par 1 1],
                Prod [Rec 1, Rec 1] ]
```

The second argument to `Par` is the position of the corresponding parameter in the type's argument list, and as both these types only have one argument it is 1. The first argument to Par and the only argument to Rec is 1 for all directly recursive (as opposed to the mutually recursive) types.

1. A functor is mathematically a function between categories that preserves the algebraic structure of the category. As such they can be applied to both functions (arrows) and datatypes (objects) [13].

As an example of a group of mutually recursive datatype definitions we can take `Zig` and `Zag` defined by:

```
data Zig a b = Blib
             | Ping a (Zag a b)
data Zag a b = Blob
             | Pong b (Zig a b)
```

By replacing the right hand side recursive references to `Zig a b` and `Zag a b` by `Rec 1` and `Rec 2` respectively we get:

```
funcZig = Sum [Prod [],
               Prod [Par 1 1, Rec 2] ]
funcZag = Sum [Prod [],
               Prod [Par 2 2, Rec 1] ]
```

Finally `Comp` is used when a type definition refers to another type which does not itself refer back to the first type. (That is, these two types are not in the same mutual recursive group.) A typical example of this is rose trees:

```
data Rose a = Fork a (List (Rose a))
funcRose = Sum [Prod [Par 1 1,
                      Comp "List" [Rec 1] ] ]
```

It is important to note here that not all types can be represented by a functor in the system. The datatypes we can handle do not include function spaces (though it could be included in this formalism, see [12]) and requires that recursive occurrences of the datatype be exactly equal to the left hand sides of the definitions. But as we have seen it does handle ordinary tree types including mutual recursive ones. Two examples of types that we can not handle are:

```
data Zigzag a b = Bliob | Piong a (Zigzag b a)
data Strange a = This a | That (Strange (a,a))
```

The first example has a recursive reference to itself which is not identical to the left hand side, but this can be worked around by instead using the pair of mutually recursive types `Zig` and `Zag` above. The second example is worse, even trying to define a map by hand on this type fails in Haskell.

```
map_Strange f (This x) = This (f x)
map_Strange f (That s) = That (map_Strange (prod f f) s)
  where prod f g (a,b) = (f a,g b)
```

Intuitively this is what a map on this type should do, but the problem is that `map_Strange` is used on different types on the left and right hand side which gives us an error message from the type checker.

3 Basic polytypic functions

To be able to reason about functional programs in a more mathematical style we will try to write functions with a limited number of combinators, standard functions and types. The following sections will describe how most of the functions generated by the system work and how they can be used.

3.1 Predefined functions and types

We use a number of families of functions indexed by integers where each family would preferably be implemented as one function taking this index as its first argument. As the Haskell type system doesn't support the types such functions would have we have implemented these function families by just extending the families names with integers. In the following we will still use the index notation but only as a more readable form of just extending the name with the number. (so $\texttt{prod}_3$ will mean `prod3`.)

3.1.1 Products

The first function families comes together with a type that is predefined in most functional languages; the tuple or product type. We define type synonyms the get type constructors for all tuples.

$$\texttt{Prod}_n :: a_1 \to \ldots \to a_n \to (a_1, \ldots, a_n)$$
$$\texttt{type Prod}_n\ a_1 \ldots a_n = (a_1, \ldots, a_n)$$

To easily write functions from a tuple to a tuple we also define

$$\texttt{prod}_n :: (a_1\to b_1) \to \ldots \to (a_n\to b_n) \to$$
$$\qquad \texttt{Prod}_n\ a_1 \ldots a_n \to \texttt{Prod}_n\ b_1 \ldots b_n$$
$$\texttt{prod}_n\ f_1 \ldots f_n\ (x_1, \ldots, x_n) = (f_1\ x_1, \ldots, f_n\ x_n)$$

which can be seen as the map for the type $\texttt{Prod}_n$ as it takes one function for every type parameter and applies that function to the corresponding argument.

As a generalisation of `uncurry :: (a -> b -> c) -> (a,b) -> c` we define

$$\texttt{uncurry}_n :: (a_1 \to \ldots \to a_n \to b) \to \texttt{Prod}_n\ a_1 \ldots a_n \to b$$
$$\texttt{uncurry}_n\ f\ (x_1, \ldots, x_n) = f\ x_1 \ldots x_n$$

which can be seen as the catamorphism on tuples where the tuple constructor $\texttt{Prod}_n$ is replaced by the function `f`.

The last tuple-related function `split` which builds a tuple from any type and thus can be seen as the anamorphism on tuples:

$$\texttt{split}_n :: (a\to b_1) \to \ldots \to (a\to b_n) \to a \to \texttt{Prod}_n\ b_1 \ldots b_n$$
$$\texttt{split}_n\ f_1 \ldots f_n\ x = (f_1\ x, \ldots, f_n\ x)$$

3.1.2 Sums

To represent a choice between different alternatives we will use the type $\texttt{Sum}_n$ defined by

```
data Sum_n a_1 ... a_n = In_n1 a_1 | ... | In_nn a_n
```

We will identify $\texttt{Sum}_1\ \texttt{a}$ with `a` and $\texttt{In}_{11}\ \texttt{x}$ with `x`.

The map on this type is

```
sum_n :: (a_1->b_1) -> ... -> (a_n->b_n) ->
         Sum_n a_1 ... a_n -> Sum_n b_1 ... b_n
sum_n f_1 ... f_n = s
  where s (In_n1 x) = In_n1 (f_1 x)
             ...
        s (In_nn x) = In_nn (f_n x)
```

The function $\texttt{join}_n$ that joins all the `n` cases together to one resulting type is defined by

```
join_n :: (a_1->b) -> ... -> (a_n->b) ->
          Sum_n a_1 ... a_n -> b
join_n f_1 ... f_n = j
  where j (In_n1 x) = f_1 x
             ...
        j (In_nn x) = f_n x
```

This can be seen as the catamorphism on $\texttt{Sum}_n$ as the constructors $\texttt{In}_{ni}$ are replaced by $\texttt{f}_i$ for all `i`.

The composition of a sum and a join can be simplified to just a join: (For brevity we here introduce a vector notation defined in the next section.)

$$\texttt{join}_n\ \underline{\texttt{f}}\ .\ \texttt{sum}_n\ \underline{\texttt{g}}\ \texttt{==}\ \texttt{join}_n\ \underline{\texttt{(f.g)}}$$

3.2 Notation

Polytypic functions will be written with an index specifying what type they work on as in $\texttt{map}_{\texttt{Tree}}$. In the system this is written `mapTree` but we will use the index notation as a remainder that we think about polytypic functions as functions taking a type as first argument and that the actual implementation is just a way of simulating this behaviour.

Normally we will work with an unspecified example type $\texttt{D}\ \texttt{a}_1\ \ldots\ \texttt{a}_n$:

```
data D a_1 ... a_n = C_1 e_1,1 ... e_k1,1
                   |     ...
                   | C_n e_1,n ... e_kn,n
```

Here the C_i are type constructors, $e_{i,j}$ are type expressions using the type parameters $a_1 \ldots a_n$ and k1 to kn are the arities of the constructors C_1 to C_n. This type is not mutually recursive to make the examples easier to read, but all functions described in this chapter works also for groups of mutually recursive types[1]. For readability we will in some examples use a vector notation and write $\underline{a}$ instead of $a_1 \ldots a_n$ where n always will be the number of type parameters of the type D. This vector notation is also used for composition, pair forming, etc. The following examples, where n==3 and D==Test, shows the intended interpretation of this:

short form	stands for
f_1 $\underline{a}$	f1 a1 a2 a3
map_D $\underline{f.g}$	mapTest (f1.g1) (f2.g2) (f3.g3)
f_D c $\underline{f}$. g	fTest c f1 f2 f3 . g
D $\underline{(a,b)}$	Test (a1,b1) (a2,b2) (a3,b3)

3.3 Object and arrow

Mathematically a functor is a function that can be applied to objects (types) and arrows (functions). To simulate this behaviour we provide for each functor one type F_D (which can be seen as a function from its type parameters to a type) and one function f_D.

The datatype F_D has exactly the same structure as the functor. This means that it has the same structure as the type D with a recursive parameter added which is used in place of the recursive references. (See the examples in section 2.3) The right hand side of the type definition is generated by a catamorphism on the type Func where the constructor Sum in Func is replaced by the code calling Sum_n, Prod by a call to $Prod_{ki}$, Comp by a call to the referred type and Rec and Par by their corresponding parameters from the left hand side. We get something of the form:

$$
\begin{aligned}
\texttt{type } F_D\ r\ \underline{a} = Sum_n\ & (Prod_{k1}\ t_{1,1}\ \ldots\ t_{1,k1}) \\
& \ldots \\
& (Prod_{kn}\ t_{n,1}\ \ldots\ t_{n,kn})
\end{aligned}
$$

where $t_{i,j}$ is the result of this catamorphism on $e_{i,j}$.

The function f_D is the map on the type F_D and will therefore also denoted by map_{FD}. Its structure is also very similar to that of the functor and it is defined by a catamorphism on the type Func with almost the same arguments as the cata for F_D. The difference is that, as we generate a map here, we change the calls to the type

1. In appendix A the code for all functions generated by the system is shown for a mutually recursive pair of types.

constructors to calls to the maps over those type constructors. The function definition has the form:

```
fD :: (ra -> rb) -> (a1->b1) -> ... -> (an->bn) ->
         FD ra a -> FD rb b
fD r a = sumn (prodk1 g1,1 ... g1,k1)
                    ...
               (prodkn gn,1 ... gn,kn)
```

We exemplify this for two datatypes; for `List` we get:

```
--funcList = Sum [Prod [], Prod [Par 1 1, Rec 1]]
type FList ra a = Sum2  ()      (a,ra)
fList rg g      = sum2 prod0 (prod2 g rg)
```

and for `Rose` we get: (remember that we have identified `Sum1 a` with `a`)

```
--data Rose a = Fork a (List (Rose a))
--funcRose = Sum [Prod [Par 1 1, Comp "List" [Rec 1] ] ]
type FRose ra a =       (a,(List ra))
fRose rg g      = (prod2 g (mapList rg))
```

3.4 In and out

We define a function $\texttt{out}_{\texttt{D}}$ so that the pattern matching for all the cases of the type `D a` is done by this function once and for all. $\texttt{out}_{\texttt{D}}$ strips off the constructors from the top level of a value of type `D a` replacing them by `Sum`s and `Prod`s.

```
outD :: D a -> FD (D a) a
outD (C1 x1 ... xk1) = Inn1 (x1, ... ,xk1)
    ...
outD (Cn x1 ... xkn) = Innn (x1, ... ,xkn)
```

It is important to note that the $\texttt{out}_{\texttt{D}}$ is not recursive and that the substructures of the result can be of the types `a` and `D a`. As an example we can take `out` for the type `List`:

```
outList :: List a -> FList (List a) a
outList (Nil)       = In21 ()
outList (Cons x xs) = In22 (x,xs)
```

The dual function $\texttt{in}_{\texttt{D}}$ does the opposite of $\texttt{out}_{\texttt{D}}$; it puts back all the correct constructors to create a value of the type `D`. It is so to say *the* constructor of the type `D` as it is a `join` of the different constructors.

```
inD :: FD (D a) a -> D a
inD = joinn (uncurryk1 C1) ... (uncurrykn Cn)
```

Here we can see the real use of the predefined functions letting us write $\mathtt{in}_D$ in a concise way without any pattern matching. `join` matches the cases of the top level `sum` and `uncurry` makes the constructors applicable to tuples. As examples we show `in` for `List` and `Rose`:

```
in_List = join_2 (const Nil) (uncurry_2 Cons)
in_Rose = uncurry_2 Fork
```

Both $\mathtt{in}_D$ and $\mathtt{out}_D$ are isomorphisms, and they are each others inverses:

```
in_D . out_D == id == out_D . in_D
```

3.5 Map

The map on a datatype `D` is a higher order function taking as many functions as arguments as the type has type parameters. When the resulting function `m` is applied to an object of type `D`, each of these argument functions is applied to all occurrences of elements of their specific type leaving the structure of the object unchanged.

```
map_D :: (a_1->b_1) -> ... -> (a_n->b_n) ->
           D a_1 ... a_n -> D b_1 ... b_n
map_D g_1 ... g_n = m
  where m = in_D . ( f_D m g_1 ... g_n ) . out_D
        m       :: D a ->                            D b
        out_D   :: D a -> F_D (D a) a
        f_D m g ::        F_D (D a) a -> F_D (D b) b
        in_D    ::                       F_D (D b) b -> D b
```

The function `m` first removes the constructors with $\mathtt{out}_D$. Then, using $\mathtt{f}_D$ `==` $\mathtt{map}_{FD}$, `m` is called recursively for all occurrences of the recursive argument in $\mathtt{F}_D$. Finally $\mathtt{in}_D$ injects the resulting object into the type `D` again.

It is instructive to compare what this definition will give us for `Tree` with the intuitive definition of `map_T` in section 2.1. (The following is a calculation, not a definition.)

```
map_Tree f == m == in_Tree . f_Tree m f . out_Tree ==
{ Insert the definitions of in and f }
join_2 Leaf (uncurry_2 Bin) . sum_2 f (prod_2 m m) . out_Tree ==
{ Use a law for join and sum: join_n f . sum_n g == join_n f.g }
join_2 (Leaf.f) (uncurry_2 Bin . prod_2 m m) . out_Tree ==
{ Use a law for uncurry and prod:
  uncurry_n f . prod_n g == (\Prod_n x -> f g x) }
join_2 (Leaf.f) (\(x_1,x_2)-> Bin (m x_1) (m x_2)) . out_Tree
```

The `join` together with the `out` on the far right is equivalent to pattern matching on the left so we get:

```
map_Tree f = m
  where m (Leaf x) = Leaf (f x)
        m (Bin x1 x2)= Bin (m x1) (m x2)
```

which can be compared with `map_T` from section 2.1:

```
map_T f (Leaf x)  = Leaf (f x)
map_T f (Bin l r) = Bin (map_T f l) (map_T f r)
```

3.6 Cata

The catamorphism is, like map, a higher order function, but it takes only one argument[1] instead of one for each parameter.

The definition of the catamorphism is similar to the definition of map; first $\texttt{out}_D$ is used to remove the constructors, then $\texttt{f}_D$ applies the catamorphism to the recursive substructures. The difference is that here we leave the values on the top level unchanged by using `id` in the call to $\texttt{f}_D$ and, most importantly, we insert the values into another type than `D` by using the argument function `i` instead of $\texttt{in}_D$.

```
cata_D :: (F_D b a -> b) -> D a -> b
cata_D i = c
  where c = i . (f_D c id) . out_D
        c        :: D a ->                         b
        out_D    :: D a -> F_D (D a) a
        f_D c id ::        F_D (D a) a -> F_D b a
        i        ::                       F_D b a -> b
```

Here `id` means `n` copies of the identity function.

Note that we could have defined `map` using `cata`:

```
map_D :: (a_1->b_1) -> ... -> (a_n->b_n) -> D a -> D b
map_D g = cata_D i
  where i = in_D . f_D id g
        i        :: F_D (D b) a ->               D b
        f_D id g :: F_D (D b) a -> F_D (D b) b
        in_D     ::                F_D (D b) b -> D b
```

This is equivalent to the definition in section 3.5 but slightly less efficient.

1. Actually, for a group of mutually recursive types, the catamorphism (for all types) takes one argument function for each type in the group.

3.7 Ana

The anamorphism is the dual of the catamorphism in the sense that everything in the definition of `ana` has the same types as in the definition of `cata` except that all arrows are reversed.

```
ana_D :: (b -> F_D b a) -> b -> D a
ana_D o = a
  where a = in_D . (f_D a id) . o
        a       :: b ->                         D a
        o       :: b -> F_D b a
        f_D a id ::      F_D b a -> F_D (D a) a
        in_D    ::                 F_D (D a) a -> D a
```

The anamorphism can be used to build elements of a type from something else. As an example we can build perfect trees of depth `d` filled with an element `x` by using the `ana` on `Tree`:

```
ana_Tree :: (b -> F_Tree b a) -> b -> Tree a
```

If we take the argument to the resulting function to be a pair of the desired depth and the element to fill with we get `b == (Int,a)` so that

```
F_Tree b a == Sum_2 a (b,b) == Sum_2 a ((Int,a),(Int,a))
o :: (Int,a) -> Sum_2 a ((Int,a),(Int,a))
o (0  ,x) = In_21 x
o (n+1,x) = In_22 ( (n,x) , (n,x) )
perf == ana_Tree o :: (Int,a) -> Tree a
perf (2,'a') == Bin ( Bin (Leaf 'a') (Leaf 'a') )
                    ( Bin (Leaf 'a') (Leaf 'a') )
```

3.8 Hylo

Often a programming problem can be solved using an intermediate type `D` such that we first construct an element of `D` using $\mathtt{ana}_D$ and then build the result by destructing this element with $\mathtt{cata}_D$. The resulting program is a composition of a catamorphism with an anamorphism. This composition can be simplified by unfolding the definitions of `cata` and `ana`:

```
cata_D i . ana_D o == c . a ==
{ the definitions of cata and ana }
i . (f_D c id) . out_D . in_D . (f_D a id) . o ==
{ out_D . in_D == id }
i . (f_D c id) . (f_D a id) . o
{ f_D == map_FD, property of all maps: map f . map g == map f.g }
i . (f_D (c . a) id) . o
```

This composition is a hylomorphism:

```
hyloD :: (FD c a -> c) -> (b -> FD b a) -> b -> c
hyloD i o = h
  where h = i . (fD h id) . o
        h       :: b ->                    c
        o       :: b -> FD b a
        fD h id ::      FD b a -> FD c a
        i       ::                FD c a -> c
```

As we can see from the type, the hylomorphism never really refers to the datatype `D` directly; using a hylomorphism is using a virtual datastructure.

The cata- and anamorphisms can be defined as special cases of the hylomorphism where

```
cataD i = hyloD i outD
anaD  o = hyloD inD o
```

3.9 Flatten

We now have the necessary basis to go on to define more specific polytypic functions. By flattening an element of a datatype `D` with respect to a type parameter a_i we mean making a list of all occurrences of values of the type a_i in the element. As this is a function from `D` to a list it is natural to try to use a catamorphism to define it. For this to be possible we must construct a function (the argument `i` to $cata_D$) that, given the flattened substructures and the data in the top level nodes, calculates the resulting list. Intuitively this is done by simply concatenating all sublists and prepending all elements in the top level. This is not much worse in practice though one has to keep track of some indices.

We will call the generated functions fl_{D1} ... fl_{Dn} where

```
flDi :: D a -> [ai]
flDi = cataD fl
  where fl = cup flFDi (concat.flFDr)
        fl :: FD [ai] a -> [ai]
        flFDi :: FD [ai] a -> [ai]
        flFDr :: FD [ai] a -> [[ai]]
        cup f g x = (f x) ++ (g x)
```

Here we have used flatten on the nonrecursive type F_D with respect to a_i (this is fl_{FDi}) and with respect to the parameter representing the recursive argument (fl_{FDr} where r is the letter r and not an index variable).

```
fl_FDi :: F_D b a -> [a_i]
fl_FDr :: F_D b a -> [b]
```

The functions $fl_{F?}$ are defined by induction over the structure of the functor which means that the definition can be generated by a catamorphism on `Func` with suitable arguments.

3.10 Partial functions

For the definition of zip we will need to deal with partial functions. For this purpose we introduce the type `Maybe` (defined by `data Maybe a = No | Yes a`) and implement partial functions as functions returning `Yes x` for arguments where they are well defined and `No` otherwise. Composition of partial functions is strict in `No` and is written with the function `cmp`. We also provide `partfun`, a simple way of defining a partial function from a predicate and a normal function.

```
cmp :: (b -> 1+ c) -> (a -> 1+ b) -> a -> 1+ c
cmp f g x = case g x of
              No -> No
              (Yes x)  -> f x
partfun :: (a -> Bool) -> (a -> b) -> a -> 1+ b
partfun p f x | p x       = Yes (f x)
              | otherwise = No
```

We will usually write the type `Maybe a` as `1+ a`.[1]

The polytypic function $prop_D$ is used to propagate `No` out of the type `D`. This means that if there is a `No` somewhere in an element `e` of `D 1+ a` the value of $prop_D$ `e` is `No`, and otherwise it is `Yes e'` where `e'` is `e` with all occurences of `Yes x` replaced by just `x`.

```
prop_D :: D 1+ a -> 1+ D a
prop_D = cata_D p
  where p = (Yes . in_D) `cmp` prop_FD
        p        :: F_D (1+ D a) 1+ a ->             1+ D a
        prop_FD  :: F_D (1+ D a) 1+ a -> 1+ F_D (D a) a
        Yes . in_D ::                    F_D (D a) a -> 1+ D a
prop_FD :: F_D (1+ a) 1+ b -> 1+ F_D a b
```

1. The functor representing the type `Maybe` is $M\ a = 1 + a$ or just $M = 1 +$.

We define a partial function hylomorphism `parthylo` that works as a normal hylomorphism for `Yes` values and propagates `No` values up to the top level.

```
parthylo_D :: (F_D c a -> 1+ c) -> (b -> 1+ F_D b a) -> b -> 1+ c
parthylo_D pi po = ph
  where ph = pi `cmp` (prop_FD . f_D ph pid) `cmp` po
        pid= Yes
       ph        :: b ->                                   1+ c
        po        :: b -> 1+ F_D b a
        f_D ph pid ::         F_D b a ->
                                F_D (1+ c) 1+ a
        prop_FD    ::           F_D (1+ c) 1+ a -> 1+ F_D c a
       pi        ::                            F_D c a -> 1+ c
```

As we mentioned in section 3.8 (where `hylo` is defined), `cata` and `ana` can be very easily defined by using `hylo` and analogously we can define the partial function version of `ana` using `parthylo`:

```
partana_D :: (b -> 1+ F_D b a) -> b -> 1+ D a
partana_D ro = parthylo_D (Yes.in_D) ro
```

3.11 Zip

The normal zip on lists takes two lists and return a list of pairs where the i:th pair contains the i:th element from both lists. The polytypic zip takes a pair of structures to a structure of pairs. The problem with this is that for a general datatype this operation is well defined only if the elements have exactly the same form. For lists this is often handled by just truncating the longer list but for a general type it is not clear how this truncating should be done. It is, for example, impossible to zip the tree `Leaf x` with `Bin l r`. We have therefore chosen to define zip as a partial function:

```
zip_D :: (D a, D b) -> 1+ D (a,b)
```

If it had not been for `1+` on the right hand side this would be a typical case for a normal anamorphism as zip builds values of type `D`. As we know that zip is strict in `No` we can instead define zip by a partial function anamorphism.

```
zip_D = partana_D z
  where z = zip_FD . prod_2 out_D out_D
        z :: (D a,D b) -> 1+ F_D (D a,D b) (a,b)
zip_FD :: (F_D a b, F_D c d) -> 1+ F_D (a,c) (b,d)
```

The zip on the functor (`zip_FD`) can be defined by induction over the structure of the functor and thus it can be generated by a catamorphism on the type `Expr`. As an example of one of these cases we show the zip for a tuple:

```
zipProd_n ((a_1,...,a_n),(b_1,...,b_n)) = Yes ((a_1,b_1),...,(a_1,b_1))
```

4 Program construction combinators

In the system we need function builders that given a functor generate the code for that specific instance of a polytypic function. To simplify writing a function builder starting with a description of a polytypic function in the notation of section 3 we will in this section describe a number of code combinators and basic code-building blocks.

4.1 A combinator example

As an example of the use of the program construction combinators, consider the definition of `cata` for some type `D a`:

$\mathtt{cata_D\ f\ =\ f\ .\ f_D\ (cata_D\ f)\ id\ .\ out_D}$

With the combinators the program code generating this function definition can be written as:

```
ca -=  f -. (arc d -@ [ca,idc]) -. (outc d)
  where ca       = catac d -@ [f]
        arc   d = q ("f"   ++d)
        outc  d = q ("out" ++d)
        catac d = q ("cata"++d)
        idc     = q "id"
        f       = q "f"
```

provided `d` is the name of the datatype.

In the following sections we will describe the combinators `-=`, `-.`, `-@` and the basic building blocks `arc`, `outc`, ... in more detail.

4.2 Basic building blocks

Here we have shorthand notation for the code representing many common functions; `id` is generated by `idc`, `const` by `constc` and so on. To give the possibility to use functions not included here there is also a function `quote` or just `q` that embeds any string in the code. The type `Expr` that is used to represent the expressions is defined in section 5.3.

```
idc :: Expr String
q   :: a -> Expr a
```

4.3 Binary combinators

The basic building blocks are put together to expressions with operators for composition (`-.`) and application (`-@`).

```
(-.) :: Expr a -> Expr a   -> Expr a
(-@) :: Expr a -> [Expr a] -> Expr a
```

The composition operator takes two expressions and forms a new expression representing the composition of the two just as the normal composition (`.`). The application operator takes an expression (hopefully representing a function) and an expression list and forms the application of the first expression to the list of arguments. Neither of these operators do any typechecking of their arguments, leaving the responsibility of generating typecorrect programs to the programmer using the combinators.

The code generated for mutual recursive datatypes is often a list a similar functions generated in parallel and to simplify this we also provide (`-.-`) and (`-@-`) which are vectorized versions of (`-.`) and (`-@`) in the same sense as normal addition is generalised to vectors. This means that for two lists of the same length the one-element version of the operator is used componentwise so that:

```
[f1, ... , fn] -.- [g1, ... gn] == [f1 -. g1, ... , fn -. gn]
```

To build function (or variable) definitions from these expressions we provide a definition operator (`-=`) (together with its vectorized version (`-=-`)) which are used in place of the ordinary equality sign with `left -= right` forming a definition given the left and right hand sides.

```
(-=) :: Expr a -> Expr a -> Def a
```

4.4 Other combinators

To generate type definitions we have chosen to use identically the same operators to build datatype definitions as function definitions and then mark datatype definitions as such by applying the function `datadef` as in:

```
datadef [(q "Test" -@ [q "a"])  -=
              (sumF 2 [q "One" -@ [q "a"],
                       q "Two" -@ [q "a",q "a"]])]
```

generating the code representing:

```
data Test a =
          One a
        | Two a a
```

After being marked by `datadef` a list of function declarations will be interpreted as a list of datatype definitions. In the same way the function `typedef` markes definitions as type synonym declarations.

We also supply a number of functions which are combinations of simple building blocks and the application operator to use as a shorthand notation for functions that are almost always applied to all their arguments immediately. A simple example is:

```
quotef :: a -> [Expr a] -> Expr a
quotef str list = q str -@ list
```

5 System

As there is currently no (typed) language available that can handle polytypic functions we have, loosely based on Hollum, made a system that generates (a representation of) the Haskell code defining a number of polytypic functions given a type declaration.

5.1 Overall structure

We have taken advantage of the fact that there already are compilers which can handle both the parsing of program text to a datatype representing the definitions and the transformation of programs (as objects of this type) to executable code, so we will only deal with getting from a parsed program to a new parsed program with the declarations of the necessary generalised functions added. More specifically the system is based on nhc [14] but the part that depends on the specific compiler has been kept small by, as the first and last step, transforming input to, and output from, our internal representations.

The generalised functions are generated in three phases:

- The parsed program is scanned for type declarations and these are transformed to functors.
- The generalised function definitions (internally represented as equalities between expressions) are built from the structure of the functors. (Some intermediate functions are also defined to make the use and definition of the generalised functions easier.)
- Finally the functions built are transformed from the internal representation to the datatype of parsed programs in the compiler, and appended to the original program. This phase also cleans up the code a bit by doing some purely algebraic optimizations of the expressions (such as removing unnecessary `id`’s)

To make testing easier the system can also show the program text in the last step.

5.2 Constructing a functor for a datatype

The first phase of the function generation converts definitions to functors in three steps; it takes a list of declarations from after the parser, extracts the datatype definitions, groups them into lists of mutually recursive type definitions[1] and converts these (with some restrictions) to functors.

1. A group of type definitions is mutually recursive if all definitions, directly or indirectly, refer to each other.

5.2.1 Representation of types

A type definition in Haskell has the structure

$$T\ v_1 \ldots v_n \quad = \quad \begin{array}{ll} C_1 & a_{1,1} \ldots a_{1,k_1} \\ \ldots & \ldots \\ C_m & a_{m,1} \ldots a_{m,k_m} \end{array} \quad \sim \sum_{i=1}^{m} \langle C_i \times \prod_{j=1}^{k_i} a_{i,j} \rangle$$

where on the left side T is the name of the type, the v_i are type variables and on the right sides the C_i are type constructors and the $a_{i,j}$ are type expressions.

We store this information in the following structure:

```
 -- ( (typename, [type variables]),
      [(constructor name, [arguments])] )
type Data a = ( (a,[a]), [(a,[Typ a])] )
```

Here a type expression can be either a type name with type expressions as arguments, a type variable or a type tuple:

```
data Typ a =
       TypCons  a [Typ a]
     | TypVar   a
     | TypTuple [Typ a]
```

The result type from the parser used must include at least the information needed to build this structure.

The first step in the conversion from datatype definitions to functors takes the parsed data into a list of `Data`, throwing away everything else. After this step nothing remains of the dependence on the datatype of the parsed result.

5.2.2 Mutually recursive datatypes

In the second step we sort the type definitions into mutually recursive groups, each of which will later become a mutually recursive functor list. We do this by seeing the list of type definitions as a directed graph, where each defined type name is a node that is connected to all the type names referenced on the right hand side of its definition. With this view finding the mutual recursive groups is the same as finding the strongly connected components[1] of this digraph. These can be found by standard graph searching[2]. Every directly recursive type will be treated as a group of mutually recursive types with just one element.

1. A strongly connected component (s.c.c.) of a digraph is a subgraph in which all nodes are connected to all other nodes, directly or indirectly.
2. For each node we find the intersection of the set of all nodes reachable from it, and the set of all nodes that can reach it. This intersection is the set of all elements in the s.c.c. to which this node belongs.

5.2.3 Functorize

In the third step every type definition gets its top level list of constructors replaced by a `Sum`. The elements in each constructors argument list are transformed and collected into a `Prod`. The transformation of these elements takes a description of a type expression to a functor; `trans :: Typ a -> Func a`, where

```
data Func a =
       Prod   [Func a]  -- direct product of functors
     | Sum    [Func a]  -- sum of functors
     | Comp a [Func a]  -- composition of a constructor
                        --   with a list of functors
     | Par Int Int      -- type parameter, first the position
                        --   number in the Func list
                        --   then the local argument number
     | Rec Int          -- recursive parameter
```

As the structure of the type expression is preserved in this process it is natural to define this transformation by means of a catamorphism on the type `Typ a`. This catamorphism replaces type variables by their positions on the left hand side of the type definition (`var2par`), tuples by `Prod`s and occurrences of any of the mutually recursive type names to `Rec` provided that all the arguments are type variables, and in the right order (`comp2rec`).

```
comp2rec :: a -> [Func a] -> Func a
var2par  :: a             -> Func a
Prod     :: [Func a]      -> Func a
cata_Typ :: (a -> [b] -> b) -> (a -> b) -> ([b] -> b) ->
               Typ a -> b
trans    :: Typ a         -> Func a
trans = cata_Typ comp2rec var2par Prod
```

If some recursive reference is not identical to any left hand side, or if a function type is encountered `trans` will report an error and functorize will fail.

5.3 Implementing the combinators

We have chosen to represent a program as a list of definitions where each definition is marked as being either a datatype definition or a function definition but otherwise identical in structure. We represent definitions simply by a pair of a left and a right hand side expression.

5.3.1 Types for expressions

An expression has a very simple structure, it is either a primitive function or constant or an application of one expression to a list of expressions[1].

```
data Expr a = EApp (Expr a) [Expr a]
            | EPrim (PrimExpr a)
```

1. Compared with λ-calculus we lack the abstraction case, but with the primitive functions supplied we can write all the functions we need.

Here `PrimExpr` contains the quote case, and the representation of a number of useful functions. It would have been sufficient to use the quote case for everything, but having special cases for different functions makes it easy to do simplifications of the generated expressions in a separate phase.

```
data PrimExpr a = PEQuote a
                | PEprod Int       | PEProd Int
                | PEsum  Int       | PESum  Int
                | PEid | PEconst ...
```

The `Int` argument to some of these constructors is the index on these functions. As an example the expression `EPrim (PEprod 3)` represents the function $\mathtt{prod}_3$.

5.3.2 Basic building blocks

To be able to make changes to these types without having to change the whole system we have defined a number of very simple functions which correspond to the different cases of the type definitions:

```
q :: a -> Expr a
q = EPrim . PEQuote
prodc :: Int -> Expr a
prodc = EPrim . PEprod
...
idc :: Expr a
idc = EPrim PEid
...
```

5.3.3 Binary combinators

With the chosen type for expressions the definition of the application and composition operators is very simple:

```
(-@) :: Expr a -> [Expr a] -> Expr a
f -@ l = EApp f l
(-.) :: Expr a -> Expr a -> Expr a
f -. g = dot -@ [f,g]
  where dot = EPrim PEdot
```

The definition operator `(-=)` just pairs the left and right arguments and marks this as a function definition as default.

5.3.4 Other combinators

The functions `datadef` and `typedef` change the marks on a list of definitions to make them datadefinitions and typedefinitions respectively.

5.4 Function generators

We provide functions generating the functions defined in section 3 (object and arrow, in and out, cata, ana, hylo, map, flatten, zip, parthylo and prop) for all datatypes for which we can construct a functor. We also provide functions generating the functions needed for unification (unify, match and datatypes with variables added, see section 6.5) but only for directly recursive types. All the generating functions take a description of a functor as first argument and can be mapped over the list of functor descriptions that is generated by the functorize step.

5.5 Simplification

The simplification function `simp` is applied to both the left and the right hand side of all function definitions. It is written as a catamorphism on the type `Expr`:

```
simp :: Expr a -> Expr a
simp = cata_Expr simpapp (EPrim . simpprim)
  where cata_Expr :: (b -> [b] -> b) -> (PrimExpr a -> b) ->
                        Expr a -> b
        simpapp :: Expr a -> [Expr a] -> Expr a
        simpprim:: PrimExpr a -> PrimExpr a
```

The function `simpprim` simplifies the indexed functions for small indices by replacing $\texttt{prod}_1$ and $\texttt{sum}_1$ with `id`, $\texttt{uncurry}_0$ with `const` and a number of other cases like these. Everything else is left unchanged.

```
simpprim (PEProd 1) = PEid
simpprim (PEsum 1)  = PEid
...
simpprim (PEuncurry 0) = EPrim PEconst
...
simpprim p = p
```

The function `simpapp` simplifies applications with four rules:

```
id . f    == f          f . id == f
(f . g) x == f (g x)      id x == x
```

This can easily be extended to handle more complicated cases but already now it makes the generated code much more readable in some cases and also a bit more efficient.

After these simplifications the code is either transformed to the datatype representing programs in the compiler, or printed as text. Both of these transformations are written as catamorphisms on the type `Expr`.

6 Unification

In this section we will describe unification (closely following Fokkinga [3]), a specific implementation of a unification algorithm using lazy arrays, and the generalisation of this algorithm to a polytypic unification algorithm.

6.1 Introduction to unification

Unification is, informally, the process of making two given expressions containing variables equal by substituting expressions for the variables. For example consider unifying $f(x, f(a, b))$ with $f(g(y, a), y)$ where x, y are variables and a, b are constants. As both expressions are of the form $f(\ , \ )$ we only need to unify x with $g(y, a)$ and, with the same substitution, $f(a, b)$ with y. These two pairs can be trivially unified with the substitution $\sigma = [x \to g(y, a), y \to f(a, b)]$. Thus the original pair is unified by $\sigma \cdot \sigma = \sigma^2$ (we need to apply the substitution twice as y occurs in the substitution from x) to form the unified expression $f(g(f(a, b), a), f(a, b)))$.

The unification fails if we ever have to unify two different constructors or constants (which we will treat as nullary constructors) or if we have to assign two different (that is, not unifiable) expressions to the same variable.

If we try to unify x with $f(x)$ we will get $\sigma = [x \to f(x)]$ which does not by any finite number of iterations make the two expressions equal, but whose fixpoint can be seen as an infinite substitution that actually makes the expressions equal. In most applications one does not consider infinite substitutions but by allowing them in the result from the unification algorithm we can still choose if it should be allowed or not thus making the program more general.

6.2 Definitions

To make the notation more precise we will call the expressions that we unify terms. A term is recursively defined as either a numbered variable or an expression consisting of a constructor followed by a list of terms.

A substitution is a function from variables to terms, but we will also use it on terms by mapping the function to all variables in the term. Note that after one application of a substitution the resulting expression may still contain variables from the substitution.

A unifier of a pair of terms is a substitution that makes the two terms equal. A unifier of a list of such pairs is a substitution that unifies all of the pairs.

A substitution s is at least as general as S iff $S = r \cdot s$ where r is a substitution. With this order the identity substitution is at least as general as all other substitutions.

The problem of unification can now be specified as the task of finding the most general unifier of a list of pairs of terms.

6.3 Outline of a unification algorithm

We want a function that given a lists of pairs of expressions finds the most general substitution or, if some pair is not unifiable, reports an error. To do this we can look at one pair at a time collecting assignments for the variables as we go through the list always checking that every new assignment conforms with the old ones.

To unify a pair of terms we can have a number of different cases depending on the form of the pair:

1. `(Exp a, Exp b)`: To unify two expressions we have first check that their top level constructors are identical and then that all their arguments are pairwise unifiable by a recursive call to unify.
2. `(Var i, Var i)`: A variable is trivially unifiable with itself without any new associations.
3. `(Var i, term)`: To unify a variable with any term we have to include the association of `i` with the term, in the substitution. If there is already an association for this variable the old and the new terms must be unified which can be done with a recursive call to unify.
4. `(term, Var i)`: This case is handled by the previous case by just swapping the elements of the pair as unification is symmetric.

This description of the algorithm does not depend on the datatype of the expressions which makes it a perfect candidate for a polytypic algorithm. Case 2, 3 and 4 do not directly refer to the expressions so the only part that necessarily differs between unification programs for different types is case 1, the matching of two expressions.

6.4 Unification with lazyArray

One important choice that has to be made when implementing the unification algorithm is how to represent the substitutions. We know that when an association has been added to the substitution it will never be changed but probably looked up a number of times. In our algorithm we will also need to do lookups before the whole substitution is determined. This can all be implemented in an efficient way by using a lazy array, lazier than the standard Haskell `Array` construct that evaluates all indices to check that they are nonequal and within the bounds. We have therefore chosen `lazyArray` (defined in [8]) which has the desired properties.

This version of the unification algorithm is based on the one occurring in [8].

```
 1 :unifyT :: (Eq b, Enum a, Ix a) =>
 2 :      (a,a) -> [(Term a b,Term a b)] -> Array a [Term a b]
 3 :unifyT r q = amap (map snd) a
 4 :  where
 5 :    a = lazyArray r (unify 1 q)
 6 :    unify :: Int -> [(Term a b, Term a b)] ->
 7 :                    [Assoc a (Int,Term a b)]
 8 :    unify u [] = []   -- the empty list is trivially unified
 9 :    unify u ((Exp a,Exp b):q) = unify u ((match (a,b))++q)
10 :    unify u ((Var n,Var m):q) | m == n = unify u q
11 :    unify u ((Var m,t):q) =
12 :            (m := (u,t)):
13 :            case head (a!m) of
14 :              (u',t') | u == u'   -> unify (u+1) q
15 :                      | otherwise -> unify u ((t',t):q)
16 :    unify u ((t,v):q) = unify u ((v,t):q)
17 :    match p = case matchD p of
18 :                No -> error "can't match expressions"
19 :                Yes q -> q
```

The algorithm takes two arguments; `r` is a range of the form `(min,max)` that contains all variables occurring in the second argument `q`, which is the list of pairs of terms to be unified. The range is needed to make an array and could have been calculated by searching through the whole list of terms `q` but this is left to the user.

The output is an array with range `r` and with lists of terms as elements. If a certain variable is free in `q`, the corresponding list in the resulting array will be empty, but otherwise the first element of this list will be the term the variable is set equal to.

The internal function `unify` produces a list of variable associations that are lazily inserted into an array by `lazyArray` on line 5. This means that when the assignment for a variable in `a` is asked for, for example by the array indexing `a!m` on line 13, `lazyArray` will search through its list of assignments until it finds one for the required variable *and*, at the same time, when passing the other assignments on the way, it will insert them too into the array. `unify` is the main unification algorithm and contains the four cases of the previous section:

In the first case (line 9) the function `match` takes care of checking the constructors and, if they are equal, zips the arguments to a list but otherwise reports an error. As all these pairs of subexpressions also have to unified, `unify` is called recursively with the new pairs concatenated to (the beginning of) the list `q`.

The second case where two equal variables are matched just calls `unify` with the rest of the list.

In the third case, where a variable is matched against a term (line 11), we must check if there already is an association for that variable in `a`. This is dangerous since if no association has yet been made the program would get stuck waiting for itself. To prevent this from happening we first emit this association paired with a unique number `u` to the resulting list (line 12) and then check to see what the first association for this variable is (line 13). If the unique numbers are equal the looked up association is the one we just emitted and thus it is the first association for this variable so all is well and we can go on to unify the rest of the list (line 14). If the numbers are not equal we have an earlier association for the same variable so we have to check that the old and the new associated terms can be unified by calling `unify` with this pair prepended to the list.

The resulting array `a` is finally stripped of the internally used numbers with the `amap` on line 3.

As an example of the use of this algorithm we can take the first pair of terms from the introduction in section 6.1:

```
p = ( app 'f' [x, app 'f' [a,b]],
      app 'f' [app 'g' [y,a],x]  )
  where app x l = Exp (x,l)
        a = app 'a' []
        b = app 'b' []
        x = Var 1
        y = Var 2
l = unifyT (1,2) [p]
```

The result is an array that satisfies the following two equalities: (where we assume the same `where` definitions as in the definition of `p`.)

```
l!1 == [app 'g' [y,a]]
l!2 == [app 'f' [a,b]]
```

6.5 Polytypic unification

The polytypic unification algorithm for a datatype `D` uses as terms elements of type `VD v` which is the type `D` with variables of type `v` added:

```
data VD v a = VDVar v | VDExp (F_D (VD v a) a)
```

This means that a value of the type `VD` is either a variable, or an expression which is almost of the type `D`. Everything in `D` is same except that instead of recursive references to `D` itself we insert references to `VD`.

In the way we have written the lazy unification algorithm almost all type dependence is already abstracted out. When we change from the type `Term` to `VD` we get `VDExp` instead of `Exp` and `VDVar` instead of `Var` in the pattern matching of the four unification cases. To get rid of this dependence too we can use $\mathtt{out}_{VD}$ on the input to remove the constructors and use $\mathtt{in}_{VD}$ on the output to get back to the correct type. What is left after this is just the three functions $\mathtt{in}_{VD}$, $\mathtt{out}_{VD}$ and $\mathtt{match}_{D}$ and these can be supplied as arguments to an intermediate function `unifyI`. This function is completely independent of `D` and can therefore be defined once and for all in a prelude and need not be generated by the system. The differences between `unifyI` and the lazy array unification algorithm from section 6.4 are marked by *italics* below:

```
unifyI matchD inVD outVD r q = amap (map (inVD.snd)) a
  where
    a = lazyArray r (unify 1 (outlist q))
    {- these lines are the same except that
       Var becomes In_21 and Exp becomes In_22 -}
    match p = case matchD p of
                No -> error "can't match expressions"
                Yes q -> outlist q
    outlist = map (prod2 outVD outVD)
```

With `unifyI` defined the definition of `unify` is trivial.

```
unify_D = unifyI match_D in_VD out_VD
```

The function `match` takes care of most of case 1 in the unification algorithm. First, by using $\mathtt{zip}_{FD}$ it checks that the constructors in the top level of the two expressions are equal. After the zip the expression can contain pairs of constants which all must be equal if the expressions are to be unifiable. This is checked by first flattening the expression using $\mathtt{fl}_{FDi}$ (for all `i`) and then mapping == over the resulting lists. If any of these checks fail, `match` returns `No`, but if all succeeds it returns `Yes q` where `q` is the list of pairs of subexpressions which need to be unified.

```
match_D :: (Eq a_1, ..., Eq a_n) =>
           (F_D b a, F_D b a) -> 1+ [(b,b)]
match_D = partfun pred fl_FDr `cmp` zip_FD
  where pred = and_n (alleq . fl_FD1) ... (alleq . fl_FDn)
        pred :: FD (b,b) (a,a) -> Bool
        alleq = all (uncurry (==))
        zip_FD :: (F_D b a, F_D b a) -> 1+ FD (b,b) (a,a)
        fl_FDr ::                    FD (b,b) (a,a) -> [(b,b)]
```

7 Conclusions

In this report we have explained what polytypism is, we have presented a number of basic polytypic functions, and we have shown how polytypic functions can be used. We have described a system in which a number of polytypic functions can be automatically generated using a formalism rather close to normal functional programming. The generating functions take a description of a datatype and generate functions that in some way depend on this type. We have shown that by small changes of a traditional unification algorithm, we obtain a polytypic version working for all directly recursive tree like types.

The system could be a good basis for the implementation of a polytypic programming language where types are (maybe somewhat restricted) values and polytypic functions are functions taking a type as an argument. There are a number of problems with doing this as we would need new notation to express types as values, a new type system and many other things.

A quicker way of making this system more available for experimentation would be to extend a Haskell compiler with one new keyword `generating` to be used with the same syntax as the `deriving` clause after a datatype declaration. The compiler could then call our system after the parser requesting that the code, defining the polytypic functions named in the `generating` construct, be generated. It would then compile the generated functions together with the rest of the program. In this way the programmer could use these polytypic functions very easily.

8 References

[1] BIRD, R. S. An Introduction to the Theory of Lists. Broy, M, editor, *Logic of Programming and Calculi of Discrete Design*, vol. F36 of NATO ASI Series, pages 5-42, Springer-Verlag, 1987.

[2] BIRD R.S., de MOOR O., HOOGENDIJK P. *Generic programming with relations and functors*. Submitted for publication, 1993.

[3] FOKKINGA, M. *Algorithmic synthesis of the Unification Algorithm.* Unpublished manuscript. 1989.

[4] HUDAK P., PEYTON JONES S.L., WADLER P. (editors) Report on the Programming Language Haskell. Version 1.2. *ACM SIGPLAN notices*, 27 (5), May 1992

[5] JEURING, J. Algorithms from Theorems. In M. Broy and C.B. Jones, editors, *Programming Concepts and Methods*, pages 247-316. North-Holland, 1990.

[6] JEURING, J. *Constructive Algorithmics: calculating programs from their specification.* Lecture notes for course on the subject at Chalmers University of Technology, 1994.

[7] JEURING J., HUTTON G.,de MOOR O. *Hollum - a generic-programming preprocessor for Gofer.* Unpublished. Available by anonymous ftp from `ftp.cs.chalmers.se` in `pub/users/johanj/hollum.tar.Z`.

[8] JOHNSSON, T., FUNCTIONAL PEARLS Efficient Graph Algorithms Using Lazy Monolithic Arrays. Submitted for publication (in J. Functional Programming), 1993.

[9] JONES, M. P. *An Introduction to Gofer.* version 2.20, draft, included as part of the standard Gofer distribution. 1991.

[10] KNIGHT,K. Unification: A Multidisciplinary Survey. *Computing Surveys*. Vol 21, no 1, p. 93, acm press, 1989.

[11] MALCOLM, G. Data structures and program transformation. *Science of Computer Programming*, Vol 14, pp 255-279, 1990.

[12] MEIJER, E. and HUTTON, G. *Bananas in Space - extending fold and unfold to exponential types*. To appear in FPCA 95

[13] PIERCE, B. C. *Basic Category Theory for Computer Scientists* Foundations of Computing Series, The MIT Press, 1991.

[14] RÖJEMO, N. *Highlights from nhc - a space efficient Haskell compiler.* To appear in FPCA 95

Appendix A - example of generated code

Following is all the code generated by the system for the types `List`, `Rose`, `Zig` and `Zag`. The first two types are not mutually recursive but `Rose` uses `List`. `Zig` and `Zag` are mutually recursive and therefore defined in parallel.

```
-------------------------------------------------------------
data List a = Nil | Cons a (List a)
fList r1 p11 = (sum2 id (prod2 p11 r1))
type FList r1 p11 = (Sum2 () (p11,r1))
inList = (join2 (const Nil) (uncurry2 Cons))
outList Nil = (In21 ())
outList (Cons x1 x2) = (In22 (x1,x2))
cataList r1 = (r1 . ((fList (cataList r1) id) . outList))
anaList r1 = (inList . ((fList (anaList r1) id) . r1))
hyloList i1 o1 = (i1 . ((fList (hyloList i1 o1) id) . o1))
mapList f1 = (inList . ((fList (mapList f1) f1) . outList))
propList = (cataList ((mapMaybe inList) . propFList))
propFList = (propsum2 . (sum2 propprod0 propprod2))
zipList = (parthyloList (Yes . inList) (zipFList . (prod2
     outList outList)))
zipFList = (cmp (propsum2 . (sum2 (cmp propprod0 zipprod0)
      (cmp (propprod2 . (prod2 Yes Yes)) zipprod2))) zipsum2)
parthyloList i1 o1 = (cmp i1 (cmp (propFList . (fList
      (parthyloList i1 o1) Yes)) o1))
flList ca flF1 = (ca (cup2 flF1 (concat . flFListr1)))
flList1 = (flList cataList flFList1)
flFListr1 = (flsum2 flprod0 (flprod2 nil wrap))
flFList1 = (flsum2 flprod0 (flprod2 wrap nil))
data VList a1 a2 = (VListVar a1) | (VListExp (FList (VList a1
     a2) a2))
inVList = (join2 VListVar VListExp)
outVList (VListVar x1) = (In21 x1)
outVList (VListExp x1) = (In22 x1)
matchList p = (cmp (partfun (and1 ((all (uncurry2 (==))) .
     flFList1)) flFListr1) zipFList p)
unifyList p = (unify matchList inVList outVList p)
-------------------------------------------------------------
```

```
-------------------------------------------------------------
data Rose a = Fork a (List (Rose a))
fRose r1 p11 = (prod2 p11 (mapList r1))
type FRose r1 p11 = (p11,((List) r1))
inRose = (uncurry2 Fork)
outRose (Fork x1 x2) = (x1,x2)
cataRose r1 = (r1 . ((fRose (cataRose r1) id) . outRose))
anaRose r1 = (inRose . ((fRose (anaRose r1) id) . r1))
hyloRose i1 o1 = (i1 . ((fRose (hyloRose i1 o1) id) . o1))
```

```
mapRose f1 = (inRose . ((fRose (mapRose f1) f1) . outRose))
propRose = (cataRose ((mapMaybe inRose) . propFRose))
propFRose = (propsum1 . (propprod2 . (prod2 id (propList .
      (mapList id)))))
zipRose = (parthyloRose (Yes . inRose) (zipFRose . (prod2
      outRose outRose)))
zipFRose = (cmp (propsum1 . (cmp (propprod2 . (prod2 Yes (cmp
      (propList . (mapList Yes)) zipList))) zipprod2)) zipsum1)
parthyloRose i1 o1 = (cmp i1 (cmp (propFRose . (fRose
      (parthyloRose i1 o1) Yes)) o1))
flRose ca flF1 = (ca (cup2 flF1 (concat . flFRoser1)))
flRose1 = (flRose cataRose flFRose1)
flFRoser1 = (flsum1 (flprod2 nil (cup1 (concat . (flList1 .
      (mapList wrap))))))
flFRose1 = (flsum1 (flprod2 wrap (cup1 (concat . (flList1 .
      (mapList nil))))))
data VRose a1 a2 = (VRoseVar a1) | (VRoseExp (FRose (VRose a1
      a2) a2))
inVRose = (join2 VRoseVar VRoseExp)
outVRose (VRoseVar x1) = (In21 x1)
outVRose (VRoseExp x1) = (In22 x1)
matchRose p = (cmp (partfun (and1 ((all (uncurry2 (==))) .
      flFRose1)) flFRoser1) zipFRose p)
unifyRose p = (unify matchRose inVRose outVRose p)
------------------------------------------------------------


------------------------------------------------------------
data Zig a b = Blib | Ping a (Zag a b)
data Zag a b = Blob | Pong b (Zig a b)
fZig r1 r2 p11 p12 = (sum2 id (prod2 p11 r2))
fZag r1 r2 p21 p22 = (sum2 id (prod2 p22 r1))
type FZig r1 r2 p11 p12 = (Sum2 () (p11,r2))
type FZag r1 r2 p21 p22 = (Sum2 () (p22,r1))
inZig = (join2 (const Blib) (uncurry2 Ping))
inZag = (join2 (const Blob) (uncurry2 Pong))
outZig Blib = (In21 ())
outZig (Ping x1 x2) = (In22 (x1,x2))
outZag Blob = (In21 ())
outZag (Pong x1 x2) = (In22 (x1,x2))
cataZig r1 r2 = (r1 . ((fZig (cataZig r1 r2) (cataZag r1 r2) id
      id) . outZig))
cataZag r1 r2 = (r2 . ((fZag (cataZig r1 r2) (cataZag r1 r2) id
      id) . outZag))
anaZig r1 r2 = (inZig . ((fZig (anaZig r1 r2) (anaZag r1 r2) id
      id) . r1))
anaZag r1 r2 = (inZag . ((fZag (anaZig r1 r2) (anaZag r1 r2) id
      id) . r2))
hyloZig i1 i2 o1 o2 = (i1 . ((fZig (hyloZig i1 i2 o1 o2)
      (hyloZag i1 i2 o1 o2) id id) . o1))
```

```
hyloZag i1 i2 o1 o2 = (i2 . ((fZag (hyloZig i1 i2 o1 o2)
      (hyloZag i1 i2 o1 o2) id id) . o2))
mapZig f1 f2 = (inZig . ((fZig (mapZig f1 f2) (mapZag f1 f2) f1
      f2) . outZig))
mapZag f1 f2 = (inZag . ((fZag (mapZig f1 f2) (mapZag f1 f2) f1
      f2) . outZag))
propZig = (cataZig ((mapMaybe inZig) . propFZig) ((mapMaybe
      inZag) . propFZag))
propZag = (cataZag ((mapMaybe inZig) . propFZig) ((mapMaybe
      inZag) . propFZag))
propFZig = (propsum2 . (sum2 propprod0 propprod2))
propFZag = (propsum2 . (sum2 propprod0 propprod2))
zipZig = (parthyloZig (Yes . inZig) (Yes . inZag) (zipFZig .
      (prod2 outZig outZig)) (zipFZag . (prod2 outZag outZag)))
zipZag = (parthyloZag (Yes . inZig) (Yes . inZag) (zipFZig .
      (prod2 outZig outZig)) (zipFZag . (prod2 outZag outZag)))
zipFZig = (cmp (propsum2 . (sum2 (cmp propprod0 zipprod0) (cmp
      (propprod2 . (prod2 Yes Yes)) zipprod2))) zipsum2)
zipFZag = (cmp (propsum2 . (sum2 (cmp propprod0 zipprod0) (cmp
      (propprod2 . (prod2 Yes Yes)) zipprod2))) zipsum2)
parthyloZig i1 i2 o1 o2 = (cmp i1 (cmp (propFZig . (fZig
      (parthyloZig i1 i2 o1 o2) (parthyloZag i1 i2 o1 o2) Yes
      Yes)) o1))
parthyloZag i1 i2 o1 o2 = (cmp i2 (cmp (propFZag . (fZag
      (parthyloZig i1 i2 o1 o2) (parthyloZag i1 i2 o1 o2) Yes
      Yes)) o2))
flZig ca flF1 flF2 = (ca (cup3 flF1 (concat . flFZigr1) (concat
      . flFZigr2)) (cup3 flF2 (concat . flFZagr1) (concat .
      flFZagr2)))
flZig1 = (flZig cataZig flFZig1 flFZag1)
flZag1 = (flZig cataZag flFZig1 flFZag1)
flZig2 = (flZig cataZig flFZig2 flFZag2)
flZag2 = (flZig cataZag flFZig2 flFZag2)
flFZigr1 = (flsum2 flprod0 (flprod2 nil nil))
flFZigr2 = (flsum2 flprod0 (flprod2 nil wrap))
flFZig1 = (flsum2 flprod0 (flprod2 wrap nil))
flFZig2 = (flsum2 flprod0 (flprod2 nil nil))
flFZagr1 = (flsum2 flprod0 (flprod2 nil wrap))
flFZagr2 = (flsum2 flprod0 (flprod2 nil nil))
flFZag1 = (flsum2 flprod0 (flprod2 nil nil))
flFZag2 = (flsum2 flprod0 (flprod2 wrap nil))
flFZag2 = (flsum2 flprod0 (flprod2 wrap nil))
------------------------------------------------------------
```

Appendix B - prelude

Following are all the functions needed to make the code in appendix A run.

```
-- Product
type Prod2 a b = (a,b)
prod2 f g (a,b) = (f a, g b)
uncurry2 f (x,y) = f x y
split2 f g a = (f a, g a)

-- Sum
data Sum2 a b = In21 a | In22 b
sum2 f g (In21 a) = In21 (f a)
sum2 f g (In22 b) = In22 (g b)
join2 f g (In21 a) = f a
join2 f g (In22 b) = g b

-- Flatten
cup0       x = []
cup1 f     x = f x
cup2 f g   x = f x ++ g x
cup3 f g h x = f x ++ g x ++ h x

flsum1 = id
flsum2 = join2

flprod0     ()    = []
flprod1 f   (a)   = f a
flprod2 f g (a,b) = f a ++ g b

nil x     = []
wrap a    = [a]

-- Partial functions
data Maybe a = No | Yes a
mapMaybe f No = No
mapMaybe f (Yes x) = Yes (f x)
cmp f g x = case g x of
              No -> No
              (Yes x)  -> f x
partfun p f x | p x       = Yes (f x)
              | otherwise = No

-- Prop
propprod0 ()              = Yes ()
propprod1 (Yes x)         = Yes (x)
propprod1 _ = No
propprod2 (Yes x1,Yes x2) = Yes (x1,x2)
propprod2 _ = No

propsum1 = id
propsum2 (In21 (Yes x)) = Yes (In21 x)
propsum2 (In22 (Yes x)) = Yes (In22 x)
propsum2 _ = No
```

```
-- Zip
zipprod2 ((a,b),(c,d)) = Yes ((a,c),(b,d))
zipprod1 ((a),  (b)  ) = Yes ((a,b))
zipprod0 ((),   ()   ) = Yes ()
zipsum2 (In21 x,In21 y)= Yes (In21 (x,y))
zipsum2 (In22 x,In22 y)= Yes (In22 (x,y))
zipsum2 _ = No
zipsum1 (x,y) = Yes (x,y)

-- Match
and0       x = True
and1 p1    x =  p1 x
and2 p1 p2 x = (p1 x) && (p2 x)

-- UNIFICATION
-- unify is independent of the datatype it works on!
-- The unification algorithm
--   q is the list of pairs of terms to unify
--   r is a range covering all variables in the expressionpairs
     in q
unify matchD inVD outVD r q = amap (map (inVD.snd)) a
  where
    a = lazyArray r (unify (1::Int) (outlist q))
    unify u [] = []
    unify u ((In22 a,In22 b):q) = unify u (outlist (match
      (a,b)) ++ q)
    unify u ((In21 n,In21 m):q) | m == n = unify u q
    unify u ((In21 m,t):q) =
            (m := (u,t)):
            case head (a!m) of
              (u',t') | u == u' -> unify (u+1) q
                      | otherwise -> unify u ((t',t):q)
    unify u ((t,v):q) = unify u ((v,t):q)
    outlist = map (prod2 outVD outVD)
    match p = case matchD p of
               No -> error "unify: can't match expressionpair"
                Yes l -> l


{-
-- A simulation of lazyArray
lazyArray (min,max) xs = array (min,max) [ i:= [ v | (j:=v) <-
     xs , i==j ]
                                         | i <- [min..max]
                                         ]
-}
```