Polytypic Programming

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

Abstract. Many functions have to be written over and over again for different datatypes, either because datatypes change during the development of programs, or because functions with similar functionality are needed on different datatypes. Examples of such functions are pretty printers, debuggers, equality functions, unifiers, pattern matchers, rewriting functions, etc. Such functions are called polytypic functions. A polytypic function is a function that is defined by induction on the structure of user-defined datatypes. This paper introduces polytypic functions, and shows how to construct and reason about polytypic functions. A larger example is studied in detail: polytypic functions for term rewriting and for determining whether a collection of rewrite rules is normalising.

1 Introduction

Complex software systems contain many datatypes, which during the development of the system will change regularly. Developing innovative and complex software is typically an evolutionary process. Furthermore, such systems contain functions that have the same functionality on different datatypes, such as equality functions, print functions, parse functions, etc. Software should be written such that the impact of changes to the software is as limited as possible. Polytypic programs are programs that adapt automatically to changing structure, and thus reduce the impact of changes. This effect is achieved by writing programs such that they work for large classes of datatypes.

Consider for example the function `length :: List a -> Int`, which counts the number of values of type `a` in a list. There exists a very similar function `length :: Tree a -> Int`, which counts the number of occurrences of `a`'s in a tree. We now want to generalise these two functions into a single function which is not only polymorphic in `a`, but also in the type constructor; we want to be able to write something like `length :: D a -> Int`, where `D` ranges over type constructors. We call such functions *polytypic functions* [20]. Once we have a polytypic `length` function, function `length` can be applied to values of any datatype. If a datatype is changed, `length` still behaves as expected. For example, the datatype `List a` has two constructors with which lists can be built: the empty list constructor, and the cons constructor, which prepends an element to a list. If we add a constructor with which we can append an element to a list, function `length` still behaves as expected, and counts the number of elements in a list.

Polytypic functions are useful in many situations, for example in implementing rewriting systems.

1.1 A problem

Suppose we want to write a term rewriting module. An example of a term rewriting system is the algebra of numbers constructed with `Zero`, `Succ`, `:+:`, and `:*:`, together with the following term rewrite rules [24].

```
x :+: Zero    ->  x
x :+: Succ y  ->  Succ (x :+: y)
x :*: Zero    ->  Zero
x :*: Succ y  ->  (x :*: y) :+: x
```

where `x` and `y` are variables. For confluent and normalising term rewriting systems, the relation $\stackrel{*}{\rightarrow}$, which rewrites a term to its normal form, is a function. For example `Succ (Succ Zero) :*: Succ (Succ Zero)` $\stackrel{*}{\rightarrow}$ `Succ (Succ (Succ (Succ Zero)))`.

We want to implement function $\stackrel{*}{\rightarrow}$ in a functional language such as Haskell [8], that is, we want to define a function `rewrite` that takes a list of rewrite rules and a term, and reduces redeces until no further reduction is possible. For the above example, we first define two datatypes: the datatype of numbers, and the datatype of numbers with variables, which is used for representing the rewrite rules. Variables are represented by integers.

```
data Number = Zero
            | Succ Number
            | Number :+: Number
            | Number :*: Number

data VNumber = Var Int
             | VZero
             | VSucc VNumber
             | VNumber :++: VNumber
             | VNumber :**: VNumber
```

A rewrite rule is represented by a pair of values of type `VNumber`.

We want to use function `rewrite` on different datatypes: rewriting is independent of the specific datatype. For example, we also want to be able to rewrite `SKI` terms, where an `SKI` term is a term built with the constant constructors `S`, `K`, `I`, and the application constructor `:@:`. We have the following rewrite rules for `SKI` terms:

```
((S :@: x) :@: y) :@: z  ->  (x :@: z) :@: (y :@: z)
(K :@: x) :@: y          ->  x
I :@: x                  ->  x
```

Since the type of function `rewrite` is independent of the specific datatype on which it is going to be used, we want to define function `rewrite` in a class.

```
class Rewrite a b where
  rewrite      :: [(b,b)] -> a -> a
  rewrite rs   =  fp (rewrite_step rs)
  rewrite_step  :: [(b,b)] -> a -> a

fp f x | f x == x    =  x
       | otherwise   =  fp f (f x)
```

Function `rewrite_step` finds a suitable redex (depending on the reduction strategy used), and rewrites it.

There are a number of problems with this solution. First, the solution is illegal Haskell, because of the two type variables in the class declaration. More important is that the relation between a datatype without and with variables is lost in the above declaration. But most important: although the informal description of `rewrite_step` is independent of a specific datatype, we have to give an instance of function `rewrite_step` on each datatype we want to use it. We would like to have a module that supplies a rewrite function for each conceivable datatype.

1.2 A solution

We extend Haskell with the possibility of defining polytypic functions. A polytypic function can be viewed as a family of functions: one function for each datatype. It is defined by induction on the structure of user-defined datatypes. If we define function `rewrite_step` as a polytypic function, then each time we use function `rewrite_step` on a datatype, code for function `rewrite_step` is automatically generated. Polytypic function definitions are type checked, and the generated functions are guaranteed to be type correct. Polytypic functions add the possibility to define functions over large classes of datatypes in a strongly typed language.

1.3 For whom?

Polytypic functions are general and abstract functions which occur often in everyday programming, examples are equality `==` and `map`. Polytypic functions are useful when building complex software systems, because they adapt automatically to changing structure, and they are useful for:

- Implementing term rewriting systems, program transformation systems, pretty printers, theorem provers, debuggers, and other general purpose systems that are used to reason about and manipulate different datatypes in a structured way.

– Generalising Haskell's [8] `deriving` construct. Haskell's `deriving` construct can be used to generate code for for example the equality function and the printing function on a lot of datatypes. There exist five classes in Haskell that can be used in the `deriving` construct, and users cannot add new classes to be used in it. The functions in these classes are easily written as polytypic functions.
– Implementing Squiggol's [28, 30, 31, 33] general purpose datatype independent functions such as `cata`, `map`, `zip`, `para` etc.
– Implementing general purpose, datatype independent programs for unification [14, 15], pattern matching [20], data compression [21], etc.

1.4 Writing polytypic programs

There exist various ways to implement polytypic programs. Three possibilities are:

– using a universal datatype;
– using higher-order polymorphism and constructor classes;
– using a special syntactic construct.

Polytypic functions can be written by defining a universal datatype, on which we define the functions we want to have available for large classes of datatypes. These polytypic functions can be used on a specific datatype by providing translation functions to and from the universal datatype. An advantage of using a universal datatype for implementing polytypic functions is that we do not need a language extension for writing polytypic programs. However, using universal datatypes has several disadvantages: type information is lost in the translation phase to the universal datatype, and type errors can occur when programs are run. Furthermore, different people will use different universal datatypes, which will make program reuse more difficult.

If we use higher-order polymorphism and constructor classes for defining polytypic functions [22, 15], type information is preserved, and we can use current functional languages such as Gofer and Haskell for implementing polytypic functions. However, writing such programs is rather cumbersome: programs become cluttered with instance declarations, and type declarations become cluttered with contexts. Furthermore, it is hard to deal with mutual recursive datatypes.

Since the first two solutions to writing polytypic functions are dissatisfying, we have extended Haskell with a syntactic construct for defining polytypic functions [16]. Thus polytypic functions can be implemented and type checked. The resulting language is called Polyp. Consult the page

`http://www.cs.chalmers.se/~johanj/polytypism/`

to obtain a compiler that compiles Polyp into Haskell (which subsequently can be compiled with a Haskell compiler), and for the latest developments on Polyp.

In order to be able to define polytypic functions we need access to the structure of the datatype `D a`. In this paper we will restrict `D a` to be a so-called *regular* datatype.

A datatype `D a` is regular if it contains no function spaces, and if the arguments of the datatype on the left- and right-hand side in its definition are the same. The collection of regular datatypes contains all conventional recursive datatypes, such as `Int`, `List a`, and different kinds of trees. Polytypic functions can be defined on a larger class of datatypes, including datatypes with function spaces [32, 11], but regular datatypes suffice for our purposes.

1.5 Background and related work

The basic idea behind polytypic programming is the idea of modelling datatypes as initial functor-algebras. This is a relatively old idea, on which a large amount of literature exists, see, amongst others, Lehmann and Smyth [26], Manes and Arbib [29], and Hagino [13].

Polytypic functions are widely used in the Squiggol community, see [10, 28, 30, 31, 33], where the 'Theory of Lists' [4, 5, 19] is extended to datatypes that can be defined by means of a regular functor. The polytypic functions used in Squiggol are general recursive combinators such as catamorphisms (generalised folds), paramorphisms, maps, etc. Sheard [42], and Böhm and Berarducci [2] give programs that automatically synthesise these functions. In the language `Charity` [6] polytypic functions like the catamorphism and map are automatically provided for each user-defined datatype. Polytypic functions for specific problems, such as the maximum segment sum problem and the pattern matching problem were first given by Bird et al. [3] and Jeuring [20]. Special purpose polytypic functions such as the generalised version of function `length` and the operator `(==)` can be found in [30, 34, 35, 40, 14]. Jay [18] has developed an alternative theory for polytypic functions, in which values are represented by their structure and their contents.

Type systems for languages with constructs for writing polytypic functions have been developed by Jay [17], Ruehr [38, 39], Sheard and Nelson [41], and Jansson and Jeuring [16]. Our extension of Haskell is based on the type system described in [16].

In object-oriented programming polytypic programming appears under the names 'design patterns' [12], and 'adaptive object-oriented programming' [27, 36]. In adaptive object-oriented programming methods are attached to groups of classes that usually satisfy certain constraints. The adaptive object-oriented programming style is very different from polytypic programming, but the resulting programs have very similar behaviour.

1.6 Overview

This paper is organised as follows. Section 2 explains the relation between datatypes and functors, and defines some basic (structured recursion) operators on some example datatypes. Section 3 introduces polytypic functions. Section 4 shows how to construct theorems for free for polytypic functions. Section 5 describes polytypic functions for unification. Section 6 describes polytypic functions for rewriting terms, and for determining whether a set of rewrite rules is normalising. Section 7 concludes the paper.

2 Datatypes and functors

A datatype can be modelled by an initial object in the category of F-algebras, where F is the functor describing the *structure* of the datatype. The essence of polytypic programming is that functions can be defined by induction on the structure of datatypes. This section introduces functors, and shows how they are used in describing the structure of datatypes. The first subsection discusses a simple non-recursive datatype. The other subsections discuss recursive datatypes, and give the definitions of basic structured recursion operators on these datatypes.

Just as in imperative languages where it is preferable to use structured iteration constructs such as **while**-loops and **for**-loops instead of unstructured **goto**s, it is advantageous to use structured recursion operators instead of unrestricted recursion when using a functional language. Structured programs are easier to reason about and more amenable to (possibly automatic) optimisations than their unstructured counterparts. Furthermore, since polytypic functions are defined for arbitrary datatypes, we cannot use traditional pattern matching in definitions of polytypic functions, and the only resources for polytypic function definitions are structured recursion operators. One of the most basic structured recursion operators is the *catamorphism*. This section defines catamorphisms on three datatypes, and shows how catamorphisms can be used in the definitions of a lot of other functions. Furthermore, it briefly discusses the fusion law for catamorphisms.

2.1 A datatype for computations that may fail

The datatype *Maybe a* is used to model computations that may fail to give a result.

$$\begin{array}{rcl} \mathit{data\ Maybe\ a} & = & \mathit{Nothing} \\ & | & \mathit{Just\ a} \end{array}$$

For example, we can define the expression *divide m n* to be equal to *Nothing* if n equals zero, and *Just* (m/n) otherwise.

To be able to use polytypic functions on the datatype *Maybe a* we have to extract the structure of this type. The datatype *Maybe a* can be modelled by the type *Mu FMaybe a*, where *Mu* is a special keyword that is used to denote datatypes which are represented by means of their structure, *FMaybe* is a so-called *functor* which describes the structure of the datatype *Maybe a*, and a is the argument of the datatype. Since we are only interested in the structure of *Maybe a*, the names of the constructors of *Maybe a* are not important. We define *FMaybe* using a conventional notation by removing *Maybe*'s constructors (writing () for the empty space we obtain by removing *Nothing*), and replacing | with +:

$$FMaybe\ a \quad = \quad () + a$$

where () is the empty product, the type containing one element, which is also denoted by (). The sum type $a+b$ consists of left-tagged elements of type a, and right-tagged elements of type b, and has constructors *inl*, which injects an element in the left component of a sum, and *inr*, which injects an element into the right component of a sum:

$$\begin{aligned} inl &:: a \to a + b \\ inr &:: b \to a + b \end{aligned}$$

We now abstract from the argument a in *FMaybe*. Function *Par* returns the parameter (the argument to the functor). Operator + and the empty product () are lifted to the function level:

$$FMaybe \quad = \quad () + Par$$

The function *inn* injects values of type $() + a$ into the type *Maybe* a. It is a variant of the function *unit* of the *Maybe*-monad. Function *out* is the inverse of function *inn*: it projects values out of the type *Maybe* a.

$$\begin{aligned} inn &:: FMaybe\ a \to Maybe\ a \\ out &:: Maybe\ a \to FMaybe\ a \end{aligned}$$

The definitions of these functions are omitted; in the polytypic programming system Polyp these functions are automatically supplied by the system for each user-defined datatype.

In category theory, a functor is a mapping between categories that preserves the algebraic structure of the category. Since a category consists of objects (types) and arrows (functions), a functor consists of two parts: a definition on types, and a definition on functions. *FMaybe* takes a type and returns a type. The part of the functor that takes a function and returns a function is called *fmap*.

$$\begin{aligned} fmap &:: (a \to b) \to FMaybe\ a \to FMaybe\ b \\ fmap &= \backslash f \to id + f \end{aligned}$$

The operator + is the 'fmap' on sums.

$$\begin{aligned} (+) &:: (a \to c) \to (b \to d) \to a + b \to c + d \\ (f + g)\ (inl\ x) &= inl\ (f\ x) \\ (f + g)\ (inr\ y) &= inr\ (g\ y) \end{aligned}$$

Exercise Use functions *inn*, *out*, and *fmap* to define the function

$$map \ :: \ (a \to b) \to Mu\ FMaybe\ a \to Mu\ FMaybe\ b$$

which takes a function f, and a value of type *Mu FMaybe* a, and returns *Nothing* in case the argument equals *Nothing*, and *Just* $(f\ x)$ in case the argument equals *Just* x. (*end of exercise*)

A function that handles values of type *Maybe* a consists of two components: a component that deals with *Nothing*, and a component that deals with values of the form *Just* x. Such functions are called catamorphisms (abbreviated to *cata*). In general, a catamorphism is a function that replaces constructors by functions. The definition of a catamorphism on the datatype *Maybe* a is very simple; definitions of catamorphisms on recursive types are more involved. To use function *cata*, we need the operator *junc*, which takes a function f of type $a \to c$ and a function g of type $b \to c$, and applies f to left-tagged values, and g to right-tagged values, throwing away the tag information:

$$\begin{array}{rcl} junc & :: & (a \to c) \to (b \to c) \to a + b \to c \\ (f\ \text{`}junc\text{`}\ g)\ (inl\ x) & = & f\ x \\ (f\ \text{`}junc\text{`}\ g)\ (inr\ y) & = & g\ y \end{array}$$

Function *cata* takes an argument e *'junc'* f of type *FMaybe* $a \to b$, and replaces the representation of *Nothing* in *Mu FMaybe* a by e (), and the representation of *Just* in *Mu FMaybe* a by f.

$$\begin{array}{rcl} cata & :: & (FMaybe\ a \to b) \to Mu\ FMaybe\ a \to\ b \\ cata & = & \backslash g \to g\ \cdot\ out \end{array}$$

For example, the function *size* that takes a *Maybe* a-value and returns 0 if it is of the form *Nothing*, and 1 otherwise, is defined by

$$size \quad = \quad cata\ ((\backslash x \to 0)\ \text{`}junc\text{`}\ (\backslash x \to 1))$$

This might seem a complicated way to define function *size*, but we will see later that this definition easily generalises to other datatypes. Another function that can be defined by means of *cata* is the function *map* defined in the above exercise.

Exercise The *Maybe*-monad contains two functions: the *unit* and *bind* functions. Function *unit* is defined as the constructor function *Just*, and function *bind* takes a value x :: *Maybe* a and a function $f :: a \to$ *Maybe* b, and returns *Nothing* in case x equals *Nothing*, and returns $f\ y$ in case x equals *Just* y. Define a function g for which the following equality holds.

$$x\ \text{`}bind\text{`}\ f \quad = \quad cata\ g\ x$$

where it is assumed that *bind* is defined on the type *Mu FMaybe* a. (*end of exercise*)

The prelude of Haskell 1.3 contains a function `maybe` defined by:

```
maybe                :: a -> (a -> b) -> Maybe a -> b
maybe n f Nothing    =  n
maybe n f (Just x)   =  f x
```

This function has the same functionality as function *cata* on the datatype *Maybe a*, and we will use it in the rest of the paper whenever we need a catamorphism on *Maybe a*.

2.2 A datatype for lists

Consider the datatype *List a* defined by

$$\mathit{data\ List\ a} \quad = \quad \mathit{Nil} \mid \mathit{Cons\ a\ (List\ a)}$$

Values of this datatype are built by prepending values of type a to a list. This datatype can be viewed as the fixed point with respect to the second argument of the datatype *FList a x* defined by

$$\mathit{data\ FList\ a\ x} \quad = \quad \mathit{FNil} \mid \mathit{FCons\ a\ x}$$

The datatype *FList a x* describes the structure of the datatype *List a*. Note that *FList* has one argument more than *FMaybe*: *FList* is a so-called *bifunctor*. The extra argument is needed to represent the occurrence of the datatype *List a* in the right-hand side of its definition. Again, since we are only interested in the structure of *List a*, the names of the constructors of *FList* are not important. Using the notation introduced when defining *FMaybe* we obtain the following definition for *FList*.

$$\mathit{FList\ a\ x} \quad = \quad () + a \times x$$

Note that juxtaposition is replaced with $\times$. The product type $a \times b$ consists of pairs of elements, and has two destructors *fst* and *snd*:

$$\begin{aligned} \mathit{fst} &:: a \times b \to a \\ \mathit{snd} &:: a \times b \to b \end{aligned}$$

We now abstract from the arguments a and x in *FList*. Function *Par* returns the parameter a (the first argument), and function *Rec* returns the recursive parameter x (the second argument). Operators $+$ and $\times$ and the empty product $()$ are lifted to the function level.

$$\mathit{FList} \quad = \quad () + \mathit{Par} \times \mathit{Rec}$$

The initial object in the category of *FList a*-algebras (the fixed point of *FList* with respect to its second component), denoted by *Mu FList a*, models the datatype *List a*. The initial object consists of two parts: the datatype *Mu FList a*, and a strict constructor function *inn*, that constructs elements of the datatype *Mu FList a*.

$$\mathit{inn} \quad :: \quad \mathit{FList\ a\ (Mu\ FList\ a)} \to \mathit{Mu\ FList\ a}$$

Function *inn* combines the constructors *Nil* and *Cons* in a single constructor function for the datatype *Mu FList a*. For example, the list containing only the integer 3,

$Cons\ 3\ Nil$, is represented by $inn\ (inr\ (3, inn\ (inl\ ())))$. Function out is the inverse of function inn.

$$\begin{aligned} out &:: Mu\ FList\ a \to FList\ a\ (Mu\ FList\ a) \\ out\ (inn\ x) &= x \end{aligned}$$

This definition by pattern matching is meaningful because inn is a constructor function.

Exercise Write functions

$$head :: Mu\ FList\ a \to a$$

which returns the first element of a nonempty list, and

$$tail :: Mu\ FList\ a \to Mu\ FList\ a$$

which returns all but the last elements of a nonempty list, using functions out and $junc$. (*end of exercise*)

Function $abstract$ takes a value of type $List$, and turns it into a value of type $Mu\ FList$.

$$\begin{aligned} abstract &:: List\ a \to Mu\ FList\ a \\ abstract\ Nil &= inn\ (inl\ ()) \\ abstract\ (Cons\ x\ xs) &= inn\ (inr\ (x, abstract\ xs)) \end{aligned}$$

So $abstract\ (Cons\ (2, Cons\ (1, Nil)))$ equals $inn\ (inr\ (2, inn\ (inr\ (1, inn\ (inl\ ())))))$. Function $concrete$ is the inverse of function $abstract$: it coerces a value of $Mu\ FList\ a$ back to a value of $List\ a$.

$$\begin{aligned} concrete &:: Mu\ FList\ a \to List\ a \\ concrete\ (inn\ (inl\ ())) &= Nil \\ concrete\ (inn\ (inr\ (x, xs))) &= Cons\ x\ (concrete\ xs) \end{aligned}$$

Functions $abstract$ and $concrete$ establish an isomorphism between $Mu\ FList$ and $List$.

$FList$ takes two types and returns a type. $FList$ is a bifunctor, which is witnessed by the existence of a corresponding action, called $fmap$, on functions. Function $fmap$ takes two functions and returns a function.

$$\begin{aligned} fmap &:: (a \to c) \to (b \to d) \to FList\ a\ b \to FList\ c\ d \\ fmap &= \backslash f \to \backslash g \to id + g \times f \end{aligned} \tag{1}$$

The operator $\times$ is the 'fmap' on products.

$$\begin{aligned} (\times) &:: (a \to c) \to (b \to d) \to a \times b \to c \times d \\ (f \times g)\ (x, y) &= (f\ x, g\ y) \end{aligned}$$

Exercise The type constructor *FList* and the function *fmap* together form a bifunctor. The proof of this fact requires a proof of

$$fmap\ f\ g \cdot fmap\ h\ j \;=\; fmap\ (f \cdot h)\ (g \cdot j)$$

(where function application binds stronger than function composition). Prove this equality. (*end of exercise*)

2.3 Catamorphisms on *Mu FList a*

Function *size* returns the number of elements in a *Mu FList a* (function `length` in Haskell). Given an argument list, the value of function *size* can be computed by replacing the constructor *Nil* by 0, and the constructor *Cons* by 1+, for example,

$$\begin{array}{llll} Cons\ (2, & Cons\ (5, & Cons\ (3, & Nil))) \\ 1+ & 1+ & 1+ & 0 \end{array}$$

So the size of this list is 3. We use a higher-order function to describe functions that replace constructors by functions: the catamorphism. The catamorphism on *Mu FList a* is the equivalent of function `foldr` on lists in Haskell. It is the basic structured recursion operator on *Mu FList a*. Function *cata* takes an argument e '*junc*' f of type $FList\ a\ b \to b$, and replaces *Cons* by f, and *Nil* by e:

$$\begin{array}{llll} Cons\ (2, & Cons\ (5, & Cons\ (3, & Nil))) \\ f\ (2, & f\ (5, & f\ (3, & e))) \end{array}$$

Function *cata* is defined using function *out* to avoid a definition by pattern matching. Function *fmap id* (*cata f*) applies *cata f* recursively to the rest of the list.

$$\begin{array}{lcl} cata & :: & (FList\ a\ b \to b) \to Mu\ FList\ a \to b \\ cata & = & \backslash f \to f \cdot fmap\ id\ (cata\ f) \cdot out \end{array}$$

We use function *cata* to define functions *size* and *map* on the datatype *Mu FList a*.

$$\begin{array}{lcl} size & :: & Mu\ FList\ a \to Int \\ size & = & cata\ ((\backslash x \to 0)\ \text{`}junc\text{`}\ (\backslash(x,n) \to n+1)) \end{array}$$

$$\begin{array}{rcl} map & :: & (a \to b) \to Mu\ FList\ a \to Mu\ FList\ b \\ map\ f & = & cata\ (inn \cdot fmap\ f\ id) \end{array}$$

The type constructor *Mu FList* and the function *map* form a functor, just as *FList* and *fmap* form a functor.

Exercise Define function `filter p`, which given a predicate `p` takes a list and removes all elements from the list that do not satisfy `p`, by means of function *cata*. (*end of exercise*)

Exercise Haskell's list selection operation `as !! n` selects the `n`-th element of the list `as`, for example, `[1,9,9,5] !! 3` = `5`. Using explicit recursion it reads:

```
(!!)            :: [a] -> Int -> a
(a:_ )!!0       =  a
(_:as)!!(n+1)   =  as!!n
```

Give an equivalent definition of `(!!)` on the datatype *Mu FList a* using *cata*. Note that the result of the *cata* has type $Int \to a$. (*end of exercise*)

2.4 Fusion

Function *cata* satisfies the so-called *Fusion law*. The fusion law gives conditions under which intermediate values produced by the catamorphism can be eliminated.

$$\begin{array}{ll} & h \cdot \mathit{cata}\ f = \mathit{cata}\ g \\ \Leftarrow & \quad \text{(Fusion)} \\ & h \cdot f = g \cdot \mathit{fmap}\ \mathit{id}\ h \end{array}$$

Fusion is a direct consequence of the free theorem [44] of the functional *cata*. It can also be proved using induction over lists. If we allow partial or infinite lists we get the extra requirement that h be strict.

We use Fusion to prove that the composition of *abstract* and *concrete* equals the identity catamorphism:

$$\mathit{abstract} \cdot \mathit{concrete} = \mathit{cata}\ \mathit{inn} \tag{2}$$

It is easy to prove that $\mathit{cata}\ \mathit{inn} = \mathit{id}$, so the proof of equality (2) is the first half of the proof that *concrete* and *abstract* establish an isomorphism.

To prove equality (2) we apply Fusion, using the fact that *concrete* equals the catamorphism $\mathit{cata}\ ((\mathit{const}\ \mathit{Nil})\ \text{`}\mathit{junc}\text{`}\ (\backslash\ (x, xs) \to \mathit{Cons}\ x\ xs))$.

$$\begin{array}{ll} & \mathit{abstract} \cdot \mathit{concrete} = \mathit{cata}\ \mathit{inn} \\ \Leftarrow & \quad \text{(Fusion)} \\ & \mathit{abstract} \cdot \mathit{const}\ \mathit{Nil}\ \text{`}\mathit{junc}\text{`}\ (\backslash(x, xs) \to \mathit{Cons}\ x\ xs) = \mathit{inn} \cdot \mathit{fmap}\ \mathit{id}\ \mathit{abstract} \end{array}$$

where function application binds stronger than infix operator application. Using the fact that function composition distributes over *junc*, and that a *junc* is uniquely determined by its two components, the proof is now split into two parts. We have to show that the following two equalities hold.

$$\begin{array}{rcl} \mathit{abstract}\ (\mathit{const}\ \mathit{Nil}\ ()) & = & \mathit{inn}\ (\mathit{fmap}\ \mathit{id}\ \mathit{abstract}\ (\mathit{inl}\ ())) \\ \mathit{abstract}\ ((\backslash(x, xs) \to \mathit{Cons}\ x\ xs)\ (x, xs)) & = & \mathit{inn}\ (\mathit{fmap}\ \mathit{id}\ \mathit{abstract}\ (\mathit{inr}\ (x, xs))) \end{array}$$

Both equalities are direct consequences of the definition of *abstract*.

Exercise The type constructor *Mu FList* and the function *map* form a functor. The proof of this fact requires a proof of

$$map\ f\ \cdot\ map\ g\ \ =\ \ map\ (f\ \cdot\ g)$$

Use fusion to prove this equality. (*end of exercise*)

2.5 A datatype for trees

The datatype *Tree a* is defined by

$$data\ Tree\ a\ \ =\ \ Leaf\ a \mid Bin\ (Tree\ a)\ (Tree\ a)$$

Applying the same procedure as for the datatype *List a*, we obtain the following functor that describes the structure of the datatype *Tree a*.

$$FTree\ \ =\ \ Par + Rec \times Rec$$

Functions *inn* and *out* are defined in the same way as functions *inn* and *out* on *Mu FList a*.

Exercise Write the function

$$is\_Leaf\ \ ::\ \ Mu\ FTree\ a \to Bool$$

which determines whether or not its argument is a leaf, using function *out*. (*end of exercise*)

Functions *abstract* and *concrete* are defined as follows on this datatype.

$$\begin{aligned}
abstract &\ ::\ Tree\ a \to Mu\ FTree\ a\\
abstract\ (Leaf\ x) &\ =\ inn\ (inl\ x)\\
abstract\ (Bin\ l\ r) &\ =\ inn\ (inr\ (abstract\ l, abstract\ r))\\
\\
concrete &\ ::\ Mu\ FTree\ a \to Tree\ a\\
concrete\ (inn\ (inl\ x)) &\ =\ Leaf\ x\\
concrete\ (inn\ (inr\ (l, r))) &\ =\ Bin\ (concrete\ l)\ (concrete\ r)
\end{aligned}$$

Function *cata* on *Mu FTree a* is defined in terms of functions *out* and *fmap*.

$$\begin{aligned}
fmap &\ ::\ (a \to c) \to (b \to d) \to FTree\ a\ b \to FTree\ c\ d\\
fmap &\ =\ \backslash f \to \backslash g \to f + g \times g \qquad (3)\\
\\
cata &\ ::\ (FTree\ a\ b \to b) \to Mu\ FTree\ a \to b\\
cata &\ =\ \backslash f \to f\ \cdot\ fmap\ id\ (cata\ f)\ \cdot\ out
\end{aligned}$$

Note that the definition of *cata* on the datatype *Mu FTree a* is exactly the same as the definition of *cata* on the datatype *Mu FList a*. Functions *size* and *map* are defined by

$$\begin{array}{rcl} size & :: & Mu\ FTree\ a \rightarrow Int \\ size & = & cata\ (\backslash x \rightarrow 1\ \text{`}junc\text{`}\ \backslash(x, y) \rightarrow x + y) \end{array}$$

$$\begin{array}{rcl} map & :: & (a \rightarrow b) \rightarrow Mu\ FTree\ a \rightarrow Mu\ FTree\ b \\ map\ f & = & cata\ (inn\ \cdot\ fmap\ f\ id) \end{array}$$

Exercise Define the function

$$min\ \ ::\ \ Ord\ a \Rightarrow Mu\ FTree\ a \rightarrow a$$

which returns the minimum element of a tree, by means of function *cata*. (*end of exercise*)

Exercise Define function

$$flatten\ \ ::\ \ Mu\ FTree\ a \rightarrow [a]$$

which returns a list containing the elements of the argument tree, using function *cata*. (*end of exercise*)

Exercise Formulate the Fusion law for trees, and prove that

$$length\ \cdot\ flatten\ \ =\ \ size$$

where function *length* returns the length of a list. (*end of exercise*)

2.6 Functors for datatypes

We have given functors that describe the structure of the datatypes *Maybe a*, *List a* and *Tree a*. For each regular datatype $D\ a$ there exists a bifunctor F such that the datatype is the fixed point in the category of $F\ a$-algebras [28]. The argument a of F encodes the parameters of the datatype $D\ a$. From the users point of view, a functor is a value generated by the following datatype.

$$data\ F\ \ =\ \ F + F \mid () \mid Con\ t \mid F \times F \mid Mu\ F\ @\ F \mid Par \mid Rec$$

Here t is one of the basic types *Bool*, *Int*, etc., $+$, $\times$, and @ are considered to be binary infix constructors, and () is a unary constructor with no arguments. Using this datatype, it is impossible to differentiate between the structure of datatypes such as:

$$\begin{array}{rcl} data\ Point\ a & = & Point\ (a, a) \\ data\ Point'\ a & = & Point'\ a\ a \\ FPoint & = & Par \times Par \end{array}$$

Functor *FPoint* describes the structure of both *Point a* and *Point' a*. This implies that it is impossible to use the fact that a constructor is curried or not in the definition of a polytypic function. Polyp's internal representation of a functor is (of course) more involved. We note the following about the datatype of functors:

- The operators $+$ and $\times$ are right-associative, so $f + g + h$ is represented as $f + (g + h)$. Operator $\times$ binds stronger than $+$. The empty product () is the unit of $\times$. Operator $+$ may only occur at top level, so $f \times (g + h)$ is an illegal functor. This restriction corresponds to the syntactic restriction in Haskell which says that | may only occur at the top level of datatype definitions.
- The alternative *Mu F* @ *F* in this datatype is used to describe the structure of types that are defined in terms of other user-defined datatypes, such as the datatype of *rose-trees*:

$$\begin{aligned} \mathit{data\ Rose\ a} &= \mathit{Fork\ a\ (List\ (Rose\ a))} \\ \mathit{FRose} &= \mathit{Par} \times (\mathit{Mu\ FList} \mathbin{@} \mathit{Rec}) \end{aligned}$$

- A datatype with more than one type argument can be represented by the type *Mu F* $(a_1 + \ldots + a_n)$, where each occurrence of a parameter in the datatype gives a *Par* in *F*. We have not yet decided how to represent datatypes with more than one type parameter in Polyp.
- In this paper we will not discuss mutually recursive datatypes, however, it will be possible to define polytypic functions over mutually recursive datatypes in Polyp.
- For a datatype that is defined using a constant type such as *Int* or *Char* we use the *Con* functor. Consider for example the structure of the following simple datatype of types:

$$\begin{aligned} \mathit{data\ Type\ a} &= \mathit{Const\ String} \mid \mathit{Var\ a} \mid \mathit{Fun\ (Type\ a)\ (Type\ a)} \\ \mathit{FType} &= \mathit{Con\ String} + \mathit{Par} + \mathit{Rec} \times \mathit{Rec} \end{aligned}$$

  and the datatype *Type a* is represented by *Mu FType a*.

The use of functors in the representation of datatypes is central to polytypic programming: families of functions (polytypic functions) are defined by induction on functors.

Exercise Give the functor *FExpr* that describes the structure of the datatype *Expr a* defined by

$$\begin{aligned} \mathit{data\ Expr\ a} &= \mathit{Con\ a} \\ &\mid \mathit{Var\ String} \\ &\mid \mathit{Add\ (Expr\ a)\ (Expr\ a)} \\ &\mid \mathit{Min\ (Expr\ a)\ (Expr\ a)} \\ &\mid \mathit{Mul\ (Expr\ a)\ (Expr\ a)} \\ &\mid \mathit{Div\ (Expr\ a)\ (Expr\ a)} \end{aligned}$$

Define the catamorphism on the datatype *Mu FExpr a*, and define subsequently the function *eval*, which takes an expression and an environment that binds variables to values, and returns the value of the expression in the environment.

$$eval :: Num\ a \Rightarrow Mu\ Expr\ a \rightarrow [(String, a)] \rightarrow a$$

(*end of exercise*)

Exercise Give the functor *FStat*, which describes the structure of the datatype *Stat a* of statements, defined by

$$\begin{array}{lcl} data\ Stat\ a & = & Assign\ String\ (Expr\ a) \\ & \mid & IfThenElse\ (Expr\ a)\ (List\ (Stat\ a))\ (List\ (Stat\ a)) \\ & \mid & While\ (Expr\ a)\ (List\ (Stat\ a)) \end{array}$$

(*end of exercise*)

3 Polytypic functions

This section introduces polytypic functions. We will define the polytypic versions of functions *fmap*, *cata*, *size*, and *map*. We will briefly discuss a type system that supports writing polytypic functions, and we will show how some of the functions that can be derived in Haskell can be defined as polytypic functions. In the following sections we will give some larger polytypic programs.

3.1 Basic polytypic functions

Functions `inn` and `out`
Functions `inn` and `out` are the basic functions with which elements of datatypes are constructed and decomposed in definitions of polytypic functions. These two functions are the *only* functions that can be used to manipulate values of datatypes in polytypic functions. One way to implement function `inn` is to define it as as the constructor function `In` of the datatype `Mu`:

```
data Mu f a  =  In (f a (Mu f a))

inn  :: f a (Mu f a) -> Mu f a
inn  =  In
```

`Mu` is a higher-order polymorphic type constructor: its argument `f` is a type constructor that takes two types and constructs a type. The datatype `Mu` is an abstraction for datatypes, and is only used in types of polytypic functions. It is impossible to produce elements of this type outside Polyp. Function `out` is the inverse of `inn`.

```
out           :: Mu f a -> f a (Mu f a)
out (inn x)   =  x
```

Function `out` is our main means for avoiding definitions by pattern matching. Instead of defining for example `f (Pattern x) = foo x` we now define `f = foo . out`, where we assume that values of the form `Pattern x` have been transformed into values of the form `Mu f a` for some `f`. This translation is taken care of by Polyp.

Functions `fmap` and `pmap`

A polytypic function is a function that is defined by induction on the structure of user-defined datatypes, i.e., by induction on functors, or a function defined in terms of such an inductive function.

A definition of a polytypic function by induction on functors starts with the keyword `polytypic`, followed by the name of the function and its type. The type declaration and the inductive definition of the function are separated by an equality sign. As a first example, consider the function `fmap`, the definition of a functor on functions. We will explain what we mean with this definition below.

```
polytypic fmap :: (a -> c) -> (b -> d) -> f a b -> f c d
  = \h j ->
      case f of
        f + g     ->  fmap h j -+- fmap h j
        ()        ->  id
        Con t     ->  id
        f * g     ->  fmap h j -*- fmap h j
        Mu f @ g  ->  pmap (fmap h j)
        Par       ->  h
        Rec       ->  j

pmap  :: (a -> b) -> Mu f a -> Mu f b
pmap  =  \h  ->  inn . fmap h (pmap h) . out

data Sum a b = Inl a | Inr b

(-+-)                :: (a -> c) -> (b -> d) -> Sum a b -> Sum c d
(f -+- g) (Inl x)  =  Inl (f x)
(f -+- g) (Inr x)  =  Inr (g x)

(-*-)              :: (a -> c) -> (b -> d) -> (a,b) -> (c,d)
(f -*- g) (a,b)  =  (f a , g b)
```

One can see this definition as a definition of a family of functions, one for each `f` on which `fmap` is used. For example, if `fmap` is used on an element of type *FList a b*, then definition (1) of *fmap* is generated, and if `fmap` is used on an element of type *FTree a b*, then definition (3) of *fmap* is generated. Note that the type variable `f` has kind `* -> * -> *`, that is, `f` takes two types and produces a type. We call variable `f` a *functor* variable. Function `fmap` and function `pmap`, which is used in the `Mu f @ g` case, are mutually recursive. Note that the different cases in the definition of a polytypic function correspond to the different components of the datatype for functors described in Section 2.

Exercise Give the instance of function `fmap` for the functor *FRose*. (*end of exercise*)

Function `cata`

Except for the type declaration, the definition of `cata` is the same as the definition of *cata* on *Mu FList a* and *Mu FTree a*. Function `cata` recursively replaces constructors by functions.

```
cata  :: (f a b -> b) -> Mu f a -> b
cata  =  \h -> h . fmap id (cata h) . out
```

The polytypic function `size` is an example of a catamorphism. It takes a value `x` of datatype `Mu f a` and counts the number of occurrences of values of type `a` in `x`.

```
size  :: Mu f a -> Int
size  =  cata fsize

polytypic fsize :: f a Int -> Int
  = case f of
      f + g     ->  fsize `junc` fsize
      ()        ->  \x -> 0
      Con t     ->  \x -> 0
      f * g     ->  \(x,y) -> fsize x + fsize y
      Mu f @ g  ->  \x -> sum (pmap fsize x)
      Par       ->  \x -> 1
      Rec       ->  \x -> x

junc              :: (a -> c) -> (b -> c) -> Sum a b -> c
junc f g (Inl x)  =  f x
junc f g (Inr x)  =  g x
```

where function `sum` sums the integers of a value of an arbitrary datatype. If function `size` is applied to a value of the datatype *List a* or *Tree a*, Polyp generates the right instantiation for function `size`.

Exercise The definition of function `fsize` requires the existence of polytypic functions `sum`, which sums the integers in a value of an arbitrary datatype.

```
sum  ::  Num a => Mu f a -> a
```

Define function `sum`. (*end of exercise*)

The first argument of function `cata` is a function of type `f a b -> b`. This kind of functions can only be constructed by means of functions `inn`, `out`, `fmap`, and functions defined by means of the `polytypic` construct. This implies that it is impossible to define the function `eval` on the datatype `Expr a` by means of a `cata`: the functor for `Expr a` contains four occurrences of the functor `Rec * Rec` (for addition, subtraction, multiplication, and division of expressions, respectively), and each polytypic function will behave in exactly the same way on these functors (so it will either add, subtract, multiply or divide all binary expressions). If we want to use function `cata` on a specific datatype we have to type the first argument of `cata` explicitly with a functor type. For this purpose, we introduce the special keyword `FunctorOf`. For example, for the following simple datatype of numbers:

```
data Number  =  Zero
             |  Succ Number
             |  Number :+: Number
             |  Number :*: Number
```

we can define the function that takes a `Number` and returns the equivalent integer by

```
value  :: Number -> Int
value  =  cata (fvalue :: FunctorOf Number -> Int)
  where fvalue = const 0     `junc`
                 id          `junc`
                 uncurry (+) `junc`
                 uncurry (*)
```

Here `FunctorOf` is a built-in 'function' that takes a regular datatype and returns a representation of its corresponding functor. So `FunctorOf Number` equals `() + Rec + Rec * Rec + Rec * Rec`.

3.2 Type checking definitions of polytypic functions

We want to be sure that functions generated by polytypic functions are type correct, so that no run-time type errors occur. For that purpose the polytypic programming system type checks definitions of polytypic functions. This subsection briefly discusses how to type check polytypic functions, the details of the type checking algorithm can be found in [16].

In order to type check inductive definitions of polytypic functions the system has to know the type of the polytypic function: higher-order unification is needed to infer the type from the types of the functions in the `case` branches, and general higher-order unification is undecidable. This is the reason why inductive definitions of polytypic functions need an explicit type declaration. Given an inductive definition of a polytypic function

```
polytypic foo :: ... f ...
```

```
  = case f of
      g + h -> bar
       ...
```

where `f` is a functor variable, the rule for type checking these definitions checks among others that the declared type of function `foo`, with `g + h` substituted for `f`, is an instance of the type of expression `bar`. For all of the expressions in the branches of the `case` it is required that the declared type is an instance of the type of the expression in the branch with the left-hand side of the branch substituted for `f` in the declared type. The expression `g + h` is an abstraction of a type, so by substituting `g + h` (or any of the other abstract type expressions) for `f` in the type of `foo` we mean the following: substitute `g + h` for `f`, and rewrite the expression obtained thus by means of the following rewrite rules:

```
(f + g) a b     ->  Sum (f a b) (g a b)
() a b          ->  ()
Con t a b       ->  t
(f * g) a b     ->  (f a b , g a b)
(Mu f @ g) a b  ->  Mu f (g a b)
Par a b         ->  a
Rec a b         ->  b
```

As an example we take the case `f * g` in the definition of `fsize`.

```
polytypic fsize :: f a Int -> Int
  = case F of
      ...
      f * g  ->  \(x,y) -> fsize x + fsize y
      ...
```

The type of the expression `\(x,y) -> fsize x + fsize y` is `(f a Int, g a Int) -> Int`. Substituting the functor to the left of the arrow in the case branch, `f * g`, for `f` in the declared type `f a Int -> Int` gives `(f * g) a Int -> Int`, and rewriting this type using the type rewrite rules, gives `(f a Int, g a Int) -> Int`. This type is equal to (and hence certainly an instance of) the type of the expression to the right of the arrow in the case branch, so this part of the polytypic function definition is type correct.

The conversion from user-defined datatypes to an internal representation of the datatype and vice versa is dealt with by the type checking algorithm. If a function expects an argument of type `Mu f a` for some `f`, and the actual argument has type `D a` for some datatype `D a`, the type checking algorithm converts the type of the argument to `Mu fD a`, where `fD` is the functor corresponding to the datatype `D a`, and vice versa.

3.3 More examples of polytypic functions

In this subsection we define a polytypic version of the function `match`, which takes a value of datatype `Mu f a` and a value of the same datatype to which a variable has been added (a variable is represented by a nullary constructor, so this datatype is of the form `Mu (() + f) a`), and returns a boolean denoting whether the second matches the first element. This is an example of a function that applies to values of different but related datatypes.

For the purpose of defining this function we define two auxiliary polytypic functions, `flatten` and `zip`, which are useful in many other situations too.

Function `flatten`
Function `flatten` takes a value `v` of a datatype `Mu f a`, and returns the list containing all elements of type `a` occurring in `v`. For example, `flatten (Bin (Bin (Leaf 1) (Leaf 3)) (Leaf 7))` equals `[1,3,7]`. This function is the central function in Jay's [18] representation of values of shapely types: a value of a shapely type is represented by its contents, obtained by flattening the value, and its structure, obtained by removing all contents of the value.

```
flatten  :: Mu f a -> [a]
flatten  =  cata fl

polytypic fl :: f a [a] -> [a]
  = case f of
      f + g     ->  fl `junc` fl
      ()        ->  \x -> []
      Con t     ->  \x -> []
      f * g     ->  \(x,y) -> fl x ++ fl y
      Mu f @ g  ->  concat . flatten . pmap fl
      Par       ->  \x -> [x]
      Rec       ->  \x -> x
```

By ordering the components of the constructors of datatypes we can make function `flatten` return a preorder, an inorder, or a postorder traversal of a tree.

```
data PreTree a  = PreLeaf a  | PreBin a (PreTree a) (PreTree a)
data InTree a   = InLeaf a   | InBin (InTree a) a (InTree a)
data PostTree a = PostLeaf a | PostBin (PostTree a) (PostTree a) a
```

Exercise Variants of function `fl` are the functions `fl_right`, which returns the list of elements that occur at the recursive (right argument) position, `fl_left`, which returns the list of elements that occur at the parameter (left argument) position, and `fl_all`, which returns the list of elements that occur at both the recursive and the parameter position.

```
polytypic fl_left   ::  f a b -> [a]
polytypic fl_right  ::  f a b -> [b]
polytypic fl_all    ::  f a a -> [a]
```

Define functions `fl_left`, `fl_right`, and `fl_all`. (*end of exercise*)

Exercise Define the polytypic function `structure` that takes a value `v` of datatype `Mu f a` and removes all contents of `v`, giving a value of type `Mu f ()`. (*end of exercise*)

Function `zip`
Haskell's `zip` function takes two lists, and pairs the elements at corresponding positions. If one list is longer than the other the extra elements are ignored. The polytypic version of function `zip`, called `pzip`, zips two values of datatype `Mu f a`; for example, `pzip (Bin (Leaf 1) (Leaf 2)) (Bin (Leaf 'a') (Leaf 'b'))` equals the tree `Bin (Leaf (1,'a')) (Leaf (2,'b'))`. Since it is impossible to zip two elements that have different structure, `pzip` returns a value of the form `Just x` if the values have the same shape, and `Nothing` otherwise. This implies that we need some functions manipulating values with occurrences of `Maybe` values. Functions `resultM` and `bindM` are the functions from the `Maybe`-monad.

```
resultM    :: a -> Maybe a
resultM x  =  Just x

bindM      :: Maybe a -> (a -> Maybe b) -> Maybe b
bindM x f  =  maybe Nothing f x

(>>=)  =  bindM

(<>)        :: (a -> Maybe b) -> (c -> Maybe a) -> c -> Maybe b
(g <> f) a  =  f a >>= g
```

where `maybe :: b -> (a -> b) -> Maybe a -> b` is an implementation of the catamorphism on the datatype `Maybe a`. Function `propagate` is a polytypic function that propagates occurrences of `Nothing` in a value of a datatype to top level. For example, if we apply `propagate` to `Bin (Leaf Nothing) (Leaf (Just 1))` we obtain `Nothing`.

```
propagate  :: Mu f (Maybe a) -> Maybe (Mu f a)
propagate  =  cata (mapM inn . fprop)

polytypic fprop :: f (Maybe a) (Maybe b) -> Maybe (f a b)
  = case f of
      f + g     ->  sumprop . fprop -+- fprop
      ()        ->  Just
      Con t     ->  Just
```

```
      f * g     ->  prodprop . fprop -*- fprop
      Mu f @ g  ->  propagate . pmap fprop
      Par       ->  id
      Rec       ->  id

sumprop  :: Sum (Maybe a) (Maybe b) -> Maybe (Sum a b)
sumprop  =  mapM Inl `junc` mapM Inr

prodprop                  :: (Maybe a,Maybe b) -> Maybe (a,b)
prodprop (Just x,Just y)  =  Just (x,y)
prodprop _                =  Nothing
```

where `mapM` is an implementation of the map function on the datatype `Maybe a`. Function `pzip` first determines whether or not the outermost constructors are equal by means of the auxiliary function `fzip`, and then applies `pzip` recursively to the children of the argument.

```
pzip  :: (Mu f a,Mu f b) -> Maybe (Mu f (a,b))
pzip  =  ((Just . In) <> (fprop . fmap Just pzip) <> fzip)
        .out -*- out

polytypic fzip :: (f a b,f c d) -> Maybe (f (a,c) (b,d))
  = case f of
      f + g     ->  (sumprop . fzip -+- fzip) <> sumzip
      ()        ->  const (resultM ())
      Con t     ->  resultM . fst
      f * g     ->  prodprop . fzip -*- fzip . prodzip
      Mu f @ g  ->  (propagate . pmap fzip) <> pzip
      Par       ->  resultM
      Rec       ->  resultM

sumzip        :: (Sum a b,Sum c d) -> Maybe (Sum (a,c) (b,d))
sumzip (x,y)  =  case (x,y) of
                   (Inl s,Inl t)  ->  Just (Inl (s,t))
                   (Inr s,Inr t)  ->  Just (Inr (s,t))
                   _              ->  Nothing

prodzip                :: ((a,b),(c,d)) -> ((a,c),(b,d))
prodzip ((x,y),(s,t))  =  ((x,s),(y,t))
```

Note that when `fzip` is applied to a pair of values that are represented by means of the `Con` functor, we have (arbitrarily) chosen to return the first of these. Function `pzip` is a strict function. To obtain a nonstrict version of function `pzip`, we define a polytypic version of function `zipWith`, called `pzipWith`.

Function zipWith

Function `pzipWith` is more general than its specialised version on the datatype of lists. It takes three functions three functions `f`, `g`, and `h`, and a pair of values `(x,y)`. If `x` and `y` have the same outermost constructor, function `pzipWith` is applied recursively to the fzipped children of its constructor, the pairs at the parameters are combined with function `g`, and the result of these applications is combined with function `f`. If `x` and `y` have different outermost constructors, `h` computes the result from `x` and `y`.

```
pzipWith :: (f c d -> d) ->
            ((a,b) -> c) ->
            ((Mu f a,Mu f b) -> d) ->
            (Mu f a,Mu f b) -> d
pzipWith f g h (x,y)  =  maybe (h (x,y))
                               (f . fmap g (pzipWith f g h))
                               (fzip (out x , out y))
```

The expression `pzipWith inn` has type `((a,b) -> c) -> ((Mu f a,Mu f b) -> Mu f c) -> (Mu f a,Mu f b) -> Mu f c`, and is the natural generalisation of Haskell's function `zipWith`. Function `pzipWith` can be used to implement a function with the same (lazy) behaviour as Haskell's `zip` for arbitrary datatypes.

Function match

Function `match` takes a value of a datatype `Mu f a` and a value of the same datatype extended with a variable `Mu (() + f)`, and returns a boolean denoting whether or not the second value matches the first value. The subfunctor `()` denotes the variable. A variable matches any value, and two values of type `Mu f` and `Mu (() + f)` (not the variable) match if they have the same outermost constructor, and if all of their children match. For example, consider the datatypes `Tree a` and `VarTree a` defined by

```
data Tree a     =         Leaf a |  Bin    (Tree a)    (Tree a)
data VarTree a  =  Var | VLeaf a | VBin (VarTree a) (VarTree a)
```

The functors for these datatypes are defined by

$$\begin{aligned} \mathit{FTree} &= \mathit{Par} + \mathit{Rec} \times \mathit{Rec} \\ \mathit{FVarTree} &= () + \mathit{FTree} \end{aligned}$$

For example, the tree with variables `VBin Var (VLeaf 3)` matches the tree `Bin (Leaf 2) (Leaf 3)`. Function `match` is defined in terms of a function `dist_left`, which distributes a sum on the left over a product, and a function `plus_to_Sum`, which takes a polytypic sum, and returns a value of the `Sum` type.

```
dist_left            :: (a , Sum b c) -> Sum (a,b) (a,c)
dist_left (a,Inl x)  =  Inl (a,x)
```

```
dist_left (a,Inr y)  =  Inr (a,y)

polytypic plus_to_Sum :: (f + g) a b -> Sum (f a b) (g a b)
  = case f of _ -> id
```

where _ matches any functor. Function `match` first determines whether or not its second argument is a variable, and returns `True` if that is the case. If its second argument is not a variable, function `match` compares the outermost constructors of its arguments by means of `fzip`, applies function `match` recursively, and checks that all results of the recursive applications return `True`.

```
match :: (Mu f a , Mu (() + f) a) -> Bool
match =  const True `junc` fmatch
        .dist_left
        .out -*- (plus_to_Sum . out)

fmatch  :: (f a (Mu f a) , f a (Mu (() + f) a)) -> Bool
fmatch  =  maybe False (and . fl_all . fmap (uncurry (==)) match)
          .fzip
```

where `uncurry` and `and` are defined in Haskell's prelude. Note that the second argument of function `match` has to be a value of a datatype of which the first constructor is a nullary constructor. So, in order to apply `match`, the order in which the constructors of a datatype are listed is significant. This is an undesirable situation, and we will show how to write a position independent definition of a function similar to `match` in Section 5.

3.4 Haskell's `deriving` construct

In Haskell there is a possibility to *derive* some classes for datatypes. For example, writing

```
data Tree a  =  Leaf a | Bin (Tree a) (Tree a) deriving Eq, Ord
```

generates instances of the class `Eq` (containing `(==)` and `(/=)`), and the class `Ord` (containing `(<)`, `(<=)`, `(>=)`, `(>)`, `max`, and `min`) for the datatype `Tree a`. There are five classes that can be derived in Haskell, besides the two classes above: `Ix`, functions for manipulating array indices, `Enum`, functions for enumerating values of a datatype, and `Text`, functions for printing values of a datatype. The functions in the derived classes are typical examples of polytypic functions. In fact, one reason for developing a language with which polytypic functions can be written was to generalise the rather ad-hoc `deriving` construct. All functions in the classes that can be derived can easily be written as polytypic functions, except for the functions in the class `Text`. To be able to write the functions in the class `Text` as polytypic functions we have to introduce a separate built-in function that gives access to constructor names. In this subsection we will define the polytypic versions of functions `(==)`, written `(!==!)`, and `(<=)`, written `(!<=!)`, by means of which all functions in the classes `Eq` and `Ord` are defined. Furthermore, we will give the function with which values of a datatype can be printed. The definitions of the polytypic versions of the functions in the other classes that can be derived can be found in Polyp's libraries.

Function `(!==!)`
Equality on datatypes is defined as follows. Function `(!==!)` zips its arguments, flattens the result to a list of pairs, and checks that each pair of values in this list consists of equal values. The arguments have equal shape if `pzip` returns a value of the form `Just z` for some `z`, and the arguments have equal contents if all pairs in the list of pairs we obtain by flattening `z` consist of equal elements.

```
(!==!)     :: Eq a => Mu f a -> Mu f a -> Bool
x !==! y   =  maybe False
                    (all (uncurry (==)) . flatten)
                    (pzip (x,y))
```

Exercise Define function `!==!` by means of function `pzipWith`. (*end of exercise*)

Function `(!<=!)`
The definition of function `(!<=!)` is more complicated than the definition of function `(!==!)`. The Haskell report defines `x <= y` for an arbitrary datatype as follows: the outermost constructor of `x` appears earlier in the datatype definition than the outermost constructor of `y`, or `x` and `y` have the same outermost constructor, and the children of `x` are lexicographically smaller than or equal to (under the ordering `(<=)`) the children of `y`. This implies that we need to be able to obtain the position of a constructor in its datatype definition. For this purpose we introduce the polytypic function `fcnumber`, which given a value of type `f a b` (usually obtained by means of function `out`, so the `b` is often `Mu f a`), returns the position of the constructor in the definition of the datatype corresponding to `Mu f a`.

```
polytypic fcnumber :: f a b -> Int
  = case f of
      f + g  ->  fcnumber `junc` ((1+) . fcnumber)
      _      ->  const 0
```

Here we use the fact that `+` is right-associative. Function `(!<=!)` is now defined as follows. It first fzips its arguments. If the arguments cannot be fzipped (i.e., `fzip` returns `Nothing`), function `order` determines which of the arguments comes first in the definition of the datatype, and returns `LT`, `EQ`, or `GT` accordingly. If the arguments to `(!<=!)` can be fzipped, functions `order` and `(!<=!)` are applied recursively, and the results are combined by flatting the result, and folding from left to right until we encounter a value unequal to `EQ`.

```
data Ordering  =  LT | EQ | GT

(!<=!)     :: Ord a => Mu f a -> Mu f a -> Ordering
x !<=! y  =  maybe (order (fcnumber (out x) , fcnumber (out y)))
                   (foldr op EQ
                   .fl_all
                   .fmap order (uncurry (!<=!))
                   )
                   (fzip (out x , out y))
  where order (x,y) | x < y   = LT
                    | x == y  = EQ
                    | x > y   = GT
        op EQ y = y
        op x  y = x
```

The class `Text`
The class `Text` contains functions for printing values of a datatype. To be able to define the functions in the class `Text` as polytypic functions, we have to introduce a separate built-in function that gives access to constructor names. This function is called `fconstructor_name`, and it is used in function `constructor_name`.

```
fconstructor_name  ::  f a b -> String

constructor_name  :: Mu f a -> String
constructor_name  =  fconstructor_name . out
```

For example, `constructor_name (Cons 1 Nil)` equals `"Cons"`. We will use this function in Section 5 for a function that behaves differently for different constructor names.

4 Parametricity for polytypic functions

In 'Theorems for free!' [44], Wadler shows how the parametricity theorem [37] can be used to construct 'free theorems' for polymorphic functions. This free theorem is obtained by just looking at the type of the function. For example, function `head` of type `[a] -> a` satisfies the theorem:

`head . map f  =  f . head`

for all strict functions `f`. And function `length` of type `[a] -> Int` satisfies the theorem:

`length  =  length . map f`

for all functions `f`. These theorems can be constructed automatically from the type of a function. Some free theorems have proven to be very useful in transforming programs, such as for example the fusion law [28], the free theorem of function `foldr`, and the acid-rain theorem [43].

In this paper we have generalised function `length` to the function `size` of type `Mu f a -> Int`. From the type of function `size` we can derive the following free theorem:

```
size  =  size . pmap f
```

where `pmap` is the map of type `(a -> b) -> Mu f a -> Mu f b`. This theorem holds for any `f` that describes the structure of a regular datatype, and some examples of datatypes on which this theorem holds are the datatype of lists and trees. So the law for function `length` given above is an instance of the law for `size`. Thus we obtain theorems for free for free: the above free theorem generates theorems for free in Wadler's sense.

In this subsection we will describe how to obtain a theorem for free for a polytypic function, for references to proofs of parametricity, see [7].

4.1 Parametricity explained

The key to deriving theorems from types is to read types as relations. This section outlines the essential ideas, and closely follows Wadler's [44] approach. We assume a basic knowledge of Wadler's paper, and we assume the same restrictions as in Wadler's paper. We will use an informal Haskell like notation for relations and sets.

Theorems for free are obtained using the following theorem.

Theorem (Parametricity). If t is a closed term of type F, then $(t,t) \in \Phi$, where Φ is the relation corresponding to the type F.

To use the parametricity theorem we have to explain how to obtain the relation corresponding to a type. For this purpose, we introduce some notation for relations, and we explain how to translate the types used in our programs into relations.

If `a` and `b` are sets, we write `r :: a <-> b` to indicate that `r` is a relation between `a` and `b`. We will often represent `r` by a function of type `(a,b) -> Bool` which returns `True` if and only if the pair of arguments is related by `r`. An example of a relation is the identity relation `id_a :: a <-> a` defined by `id_a (x,y) = x == y`. If a relation binds at most one value of type `b` to any value of type `a`, it can be represented by a function of type `a -> b`.

Our type language consists of constant types such as `Bool` and `Int`, and of three type constructors: functor types `f a b`, where `f` is a functor, function types `a -> b`, and polymorphic types `∀ a . t(a)`, where `t` is a function that given a type returns a type (the ∀ is usually not visible in our programs). We translate each of these categories of types into relations.

Constant types as relations
Constant types such as `Bool` and `Int`, may simply be read as identity relations: `id_Bool :: Bool <-> Bool`, and `id_Int :: Int <-> Int`.

Functor types as relations
The function `relate_functor` takes two relations `r :: a <-> a'` and `s :: b <-> b'`, and two values of a functor type `x :: f a b` and `y :: f a' b'` and determines whether or not `x` and `y` are related. `x` and `y` are related if they have the same structure (so `fzip (x,y)` does not return `Nothing`), and if the arguments at the parameter positions are related by `r`, and the arguments at the recursive positions are related by `s`.

```
relate_functor      :: ((a,a') -> Bool) -> ((b,b') -> Bool) ->
                       (f a b , f a' b') -> Bool
relate_functor r s  =  maybe False
                             (and . fl_all . fmap r s)
                      .fzip
```

Note that only values of functor types with equal functors can be related to each other; it is for example impossible to relate a sum type with a product type.

In the special case where `r` and `s` are functions of type `a -> a'` and `b -> b'`, respectively, `relate_functor` can be defined as a function of type `f a b -> f a' b'`.

```
relate_functor      :: (a -> a') -> (b -> b') -> f a b -> f a' b'
relate_functor r s  =  fmap r s
```

Function types as relations
Functions are related if they take related arguments into related results. The function `relate_function` takes two relations `r :: a <-> a'` and `s :: b <-> b'`, and two values of a function type `f :: a -> b` and `f' :: a' -> b'` and determines whether or not `f` and `f'` are related. `f` and `f'` are related if and only if for all pairs `(x,x')` related by `r`, the pairs `(f x,f' x')` are related by `s`.

```
relate_function :: ((a,a') -> Bool) -> ((b,b') -> Bool) ->
                   (a -> b , a' -> b') -> Bool
relate_function r s (f,f') =
  and [s (f x , f' x') | x <- a, x' <- a', r (x,x')]
```

where we informally assume that `a` and `a'` are (possibly infinite) sets.

In the special case where `r` and `s` are functions, the relation `relate_function r s` need not necessarily be a function of type `(a -> b) -> (a' -> b')`, but in this case we have

```
relate_function :: Eq b' => (a -> a') -> (b -> b') ->
                            (a -> b , a' -> b') -> Bool
relate_function r s (f,f') = and [s (f x) == f' (r x) | x <- a]
```

Forall types as relations
Polymorphic functions are related if they take related types into related results. Let `r(s)` be a relation depending on relation `s`. Then `r` corresponds to a function from relations to relations, such that for every relation `t :: a <-> a'` there is a corresponding relation `r(t) :: v(a) <-> v'(a')`. The relation ∀ `s . r(s) ::` ∀ `a . v(a) <->` ∀ `a' . v'(a')` is now defined by

```
relate_forall :: ((a <-> a') -> (v(a) <-> v'(a')) ->
                 (v(a) , v'(a')) -> Bool
relate_forall r (g,g') = and [ r(s) (g,g') | s <- (a <-> a')]
```

This definition should be read as: for all relations `s :: a <-> a'`, `r(s)` relates `g` and `g'`.

The parametricity theorem requires the construction of the relation corresponding to a type. This relation is obtained by recursively applying the `relate` functions defined in this section to a given type. We will give two examples in the following section.

4.2 Parametricity applied

In this section we give two examples of how to obtain free theorems for polytypic functions by hand. Free theorems can be derived automatically, see Fegaras and Sheard [9] for a function that given a type constructs its free theorem.

The free theorem for function `size`
Function `size` takes a value `v` of datatype `Mu f a`, and returns the number of occurrences of values of type `a` in `v`.

```
size :: ∀ a . Mu f a -> Int
```

Parametricity ensures that `(size,size)` is an element of the relation corresponding to ∀ `a . Mu f a -> Int`. The relation corresponding to this type is obtained by recursively applying the `relate` functions:

```
relate_forall (relate_function (relate_functor (Mu f a)) id_Int)
```

If we apply this function to the pair of functions `(size,size)`, the definition of `relate_forall` says that we have to show that for all relations `r :: a <-> a'`, we have

```
relate_function (relate_functor (Mu f r)) id_Int (size,size)
```

If we assume that `r :: a -> a'` is a function, then `relate_functor (Mu f r)` is a function, namely the `fmap` on `Mu f a`: `map r`. Since in this case both arguments to `relate_function` are functions, we obtain by definition of `relate_function` that the above expression is equal to:

```
and [ id_Int (size x) == size (map r x) | x <- a]
```

Since `id_Int` is the identity, this is equivalent to:

```
and [ size x == size (map r x) | x <- a]
```

or, removing the informal list-comprehension notation, for all functions `r :: a -> a'`, and for all `x` in `a`,

```
size x == size (map r x)
```

So first mapping a function `r` and then computing the size gives the same result as immediately computing the size: mapping a function over a value does not change its size.

The free theorem for function `cata`

Function `cata` has the following type:

```
∀a . ∀b . (f a b -> b) -> Mu f a -> b
```

Parametricity ensures that the pair `(cata,cata)` is an element of the relation corresponding to this type. To obtain this relation, we again apply the `relate` functions recursively.

```
relate_forall
  (relate_forall
     (relate_function
        (relate_function (relate_functor (f a b)) b)
        (relate_function (relate_functor (Mu f a)) b)
     )
  )
```

If we apply this function to the pair of functions `cata`, the definition of the relation `relate_forall` says that we have to show that for all relations `r :: a <-> a'` and `s :: b <-> b'` we have:

```
relate_function
  (relate_function (relate_functor (f r s)) s)
  (relate_function (relate_functor (Mu f r)) s)
  (cata,cata)
```

In the special case where `r` and `s` are functions, we have that `relate_functor (f r s)` is the function `fmap r s`, and `relate_functor (Mu f r)` is the function `map r`.

```
relate_function
  (relate_function (fmap r s) s)
  (relate_function (map r) s)
  (cata,cata)
```

By definition of `relate_function`, this is equivalent to: for all `(f,f')` related by `relate_function (fmap r s) s`:

```
relate_function (map r) s (cata f , cata f')
```

Since `map r` and `s` are functions, this is equivalent to: for all `x`,

```
s (cata f x) == cata f' (map r x)
```

This equality holds provided the pair of functions `(f,f')` is related by the relation `relate_function (fmap r s) s`, which, because `fmap r s` and `s` are functions, is equivalent to: for all `y`,

```
s (f y) == f' (fmap r s y)
```

Concluding, we have found that:

```
s (f y) == f' (fmap r s y)  ⇒  s (cata f x) == cata f' (map r x)
```

The fusion law given in Section 2.4 is an instance of this free theorem: instantiate the theorem with the functor for lists, and with `r` the identity function.

Exercise Give the free theorem for functions `flatten` and `pzip`. (*end of exercise*)

Very likely the results of this section can be extended in the sense that for example catamorphisms on different datatypes can be related, but the precise details are not clear to us.

5 Polytypic unification

Unifying two expressions that may contain variables amounts to finding expressions to substitute for the variables such that the two expressions are equal after performing the substitution. Use of unification is widespread, such as in type inference algorithms, rewriting systems, compilers, etc. [25]. The datatypes of the expressions to be unified in the different examples are all different, so a polytypic unification function is desirable. This section describes a polytypic unification algorithm.

As an example application of unification, consider the two expressions $f(x, f(a, b))$ and $f(g(y, a), y)$, where x and y are variables and f, g, a and b are constants. Since both expressions have an f on the outermost level, these expressions can be unified if x and $g(y, a)$ can be unified, and if $f(a, b)$ and y can be unified. The substitution $\{x \mapsto g(y, a), y \mapsto f(a, b)\}$ unifies these two pairs of expressions. The original pair of expressions is unified by applying the substitution twice (we have to apply the substitution twice because variable y occurs in the expression substituted for x), giving the unified expression $f(g(f(a, b), a), f(a, b))$.

Unification fails if its arguments have different outermost constructors or constants. Unifying x with $f(x)$ will give the substitution $\{x \mapsto f(x)\}$, which cannot be used to make two expressions equal by means of a finite number of applications. Our unification program does not fail in this case, but it is easy to extend it with a function that determines whether or not a substitution is cyclic.

5.1 Definitions and outline of the algorithm

Function `unify` takes a pair of values of type `Mu f a`, and returns either `Nothing` if the pair of values is not unifiable, or it returns `Just s` where `s` is a substitution that unifies the pair of values. In case the pair of values does not contain variables, function `unify` behaves exactly as the equality function, returning `Just s`, where `s` is the empty substitution, if and only if the the argument values are equal. Unification is defined on all datatypes `Mu f a`, and it assumes that variables are integers preceded by a constructor the name of which starts with the string `"Var"`. An example datatype on which we might want to use unification is

```
data Type a  =  VarType Int | ConType a (List (Type a))
```

Function `checkVar` determines whether or not a value is a variable. If its argument is a variable `Var i` (where the constructor `Var` may be followed by a string, for example `Type`), function `checkVar` returns `Just i`, otherwise it returns `Nothing`.

```
checkVar  :: Mu f a -> Maybe Int
checkVar  =  fcheckVar . out

fcheckVar    :: f a b -> Maybe Int
fcheckVar t  =  if "Var" == take 3 (fconstructor_name t)
                then Just (fgetVar t)
                else Nothing

polytypic fgetVar :: f a b -> Int
  = case f of
      f + g    ->  fgetVar `junc` fgetVar
      Con Int  ->  id
      _        ->  undefined
```

Function `vars` takes a value, and returns a list containing all variables that occur in the value.

```
vars  :: Mu f a -> [Int]
vars  =  cata fvars
  where fvars x = maybe (concat (fl_right x)) (:[]) (fcheckVar x)
```

A substitution is a function from variables to expressions. We represent a substitution by an association array:

```
type Subst f a      =  Array Int (Maybe (Mu f a))
start_subst bounds  =  array bounds
                              (map (:= Nothing) (range bounds))
addbind (i,t) arr   =  arr//[(i:=Just t)]
lookup i s          =  s!i
```

A unifier of a pair of expressions is a substitution that makes the two expressions equal. A substitution `s` is at least as general as `t` if and only if `t` can be factored by `s`, i.e. there exists a substitution `r` such that `t` = `r . s`, where we treat substitutions as functions. We want to define a function that given a pair of expressions finds the most general substitution that unifies the pair, or, if it is not unifiable, reports an error.

Exercise Define function

```
subst  ::  Subst f a -> Mu f a -> Mu f a
```

(*end of exercise*)

A pair of expressions has one of the following four forms. For each form we describe how unification proceeds.

- A pair of equal variables. A variable is trivially unifiable with itself.
- A pair of expressions. To unify two expressions we first check that their outermost constructors are equal, and subsequently that all children are pairwise unifiable.
- A pair of a variable and an expression (which may be a variable different from the first variable). To unify a variable with a expression we include the association of the variable with the expression in the substitution. If there already exists an association for the variable, the old and new association have to be unified.
- A pair of an expression and a variable. To unify an expression with a variable we apply the previous case with the arguments swapped.

Only the second case refers to the structure of expressions, the implementation of the other cases is immediate.

5.2 Function `unify`

Function `unify` takes a pair of expressions, and returns its most general unifier. It is defined in expressions of function `unify'`, which incrementally computes the substitution, and corresponding functions `unifyList` and `unifyList'` for a list of pairs of expressions. Function `unifyList` starts with the start substitution, and computes the contribution of each pair of expressions to the substitution. It uses function `mfoldl`, the monadic version of function `foldl`, to thread occurrences of `Nothing` through the computation, and function `varbounds`, to determine the bounds of the substitution array by computing the minimum and maximum variable number.

```
mfoldl      :: (b -> a -> Maybe b) -> b -> [a] -> Maybe b
mfoldl f e  =  foldl f' (resultM e)
  where f' mb a = mb >>= \b -> f b a

varbounds :: [(Mu f a , Mu f a)] -> (Int,Int)
varbounds ts = let v = concat (map (\(x,y) -> vars x++vars y) ts)
               in (minimum v,maximum v)

unify         :: Eq a => (Mu f a , Mu f a) -> Maybe (Subst f a)
unify p       =  unifyList [p]
unifyList xs  =  unifyList' (start_subst (varbounds xs)) xs
unifyList'    =  mfoldl unify'
```

Note that function `varbounds` assumes that each of its argument rewriting rules contains at least one variable; it is easy to adjust `varbounds` such that it also works for rewriting rules with no variables. The main unification engine is `unify'` which implements the description of the unification algorithm given above. It uses amongst others function `parEq`, which checks that all pairs of values occurring at the parameter position in a value obtained from function `fzip` consist of equal values, and function `update`, which checks that we do not try to unify a variable with an expression that contains the same variable, and which subsequently adds the binding of the variable with the expression to the substitution obtained by unifying the old expression bound to the variable with the new expression.

```
unify':: Eq a => Subst f a ->
                 (Mu f a , Mu f a) ->
                 Maybe (Subst f a)
unify' s (x,y) = uni (checkVar x , checkVar y)
  where
  uni (Just i  , Just j ) | i == j = Just s
  uni (Nothing , Nothing) = ((unifyList' s . fl_right)
                            <> checkEq
                            <> fzip
                            ) (out x , out y)
  uni (Just i  , _      ) = update s (i,y)
  uni ( _      , Just j ) = update s (j,x)

checkEq r  =  if parEq r then Just r else Nothing

parEq  :: Eq a => f (a,a) b -> Bool
parEq  =  all (uncurry (==)) . fl_left

update :: Eq a => Subst f a -> (Int , Mu f a) -> Maybe (Subst f a)
update s (i,t) = case lookup i s of
                   Nothing  ->  Just (addbind (i,t) s)
                   Just t'  ->  unify' (addbind (i,t) s) (t,t')
```

If we want unification to fail in case a variable is bound to an expression that contains the variable itself, we can add an occurs-check to function `update`, or we can check afterwards that the resulting substitution is acyclic.

6 Polytypic term rewriting

Rewriting systems is another area in which polytypic functions are useful. A rewriting system is an algebra together with a set of rewriting rules. In a functional language, the algebra is represented by a datatype, and the rewriting rules can be represented as a list of pairs of values of the datatype extended with variables. In this section we will define a function `rewrite` which takes a set of rewrite rules of some datatype extended with variables, and a value of the datatype without variables, and rewrites this value by means of the rewriting rules using the parallel-innermost strategy, until a normal form is reached. We use the parallel-innermost strategy because it is relatively easy to implement function `rewrite` as an efficient function when using this strategy. Function `rewrite` does not check if the rewriting rules in its first argument are normalising, so it will not terminate for certain inputs. The other main function defined in this section is a function that determines whether a set of rewriting rules is normalising. This function is based on a well-known method of recursive path orderings, as developed by Dershowitz on the basis of a theorem of Kruskal, see [24]. The results in this section are for a large part based on results from Berglund [1], in which more applications of polytypic functions in rewriting systems can be found.

6.1 A function for rewriting terms

Function `rewrite` takes two arguments with different but related types: a set of rewrite rules of a datatype extended with variables, and a value of the datatype without variables. To express this relation between the types of the arguments we have to make the presence of variables visible in the type. Let `Mu f a` be an arbitrary datatype. Then we can extend this datatype with variables (represented by integers) by adding an extra component to the sum represented by `f`: `Mu (Con Int + f) a`. Thus we obtain the following type for `rewrite`:

```
type MuVar f a  =  Mu (Con Int + f) a
type Rule f a   =  (MuVar f a , MuVar f a)

rewrite  ::  [Rule f a] -> Mu f a -> Mu f a
```

Later we will convert values of `Mu f a` to values of type `MuVar f a` and vice versa. Functions `toMuVar` and `fromMuVar` take care of these type conversions. Function `toMuvar` injects values of an arbitrary datatype into values of the datatype extended with variables. The resulting value does not contain variables. Function `fromMuVar` translates a variable-free value of the datatype extended with variables to the datatype without variables. This function fails when it is applied to a value that does contain variables.

```
toMuVar  :: Mu f a -> MuVar f a
toMuVar  =  cata ftoMuVar

polytypic ftoMuVar :: f a (MuVar f a) -> MuVar f a
  = case f of
      _  ->  \x -> inn (Inr x)

fromMuVar  :: MuVar f a -> Mu f a
fromMuVar  =  cata ffromMuVar

polytypic ffromMuVar :: (Con Int + f) a (Mu f a) -> Mu f a
  = case f of
      _  ->  \(Inr x) -> inn x
```

We will define function `rewrite` in a number of stages. The first definition is a simple, clearly correct but very inefficient implementation of `rewrite`. This definition will subsequently be refined to a function with better performance.

A first definition of function `rewrite`

Function `rewrite` rewrites its second argument with the rules from its first argument until it reaches a normal form. So function `rewrite` is the fixed-point of a function that performs a single parallel-innermost rewrite step, function `rewrite_step`. The fixed-point computation is surrounded by type conversions in order to be able to apply the functions for unification given in the previous section.

```
rewrite rs  =  fromMuVar . rewrite' rs . toMuVar

rewrite' rs  =  fp (rewrite_step rs)

fp f x  |  fx !==! x  =  x
        |  otherwise  =  fp f fx
  where fx = f x

rewrite_step  ::  [Rule f a] -> MuVar f a -> MuVar f a
```

Function `rewrite_step` is the main rewriting engine. Given a set of rules and a value `x`, it tries to rewrite all innermost redeces of `x`. This is achieved by applying `rewrite_step` recursively to `x`, and only rewriting the innermost redeces. At each recursive application function `rewrite_step` applies a function `innermost`. Function `innermost` determines whether or not one of the children has been rewritten. Only if this is not the case, it tries to reduce its argument. To determine whether or not one of the children has been rewritten, function `innermost` compares ist argument with the original argument of function `rewrite_step`. The recursive structure of function `rewrite_step` is that of a `cata`, but it needs access to the original argument too. Such functions are called *paramorphisms* [30].

```
para      :: (f a b -> Mu f a -> b) -> Mu f a -> b
para h x  =  h (fmap id (para h) (out x)) x

rewrite_step rs  =  para (innermost rs)

innermost :: Eq a => [Rule f a] -> (Con Int + f) a (MuVar f a) ->
                     MuVar f a -> MuVar f a
innermost rs x' x =
  if (inn x') !/=! x then inn x' else reduce rs x

reduce :: Eq a => [Rule f a] -> MuVar f a -> MuVar f a
reduce []            t  =  t
reduce ((lhs,rhs):rs) t  =  case unify (lhs,t) of
                              Just s   ->  subst s rhs
                              Nothing  ->  reduce rs t
```

Function `rewrite` is extremely inefficient. For example, if we represent natural numbers with `Succ` and `Zero`, and we use the rewriting rules for `Zero`, `Succ`, `:+:`, and `:*:` given in the introduction, it takes hundreds of millions of (Gofer) reductions to rewrite the representation of 2^8 to the representation of 256. One reason why `rewrite` is inefficient is that in each application of function `rewrite_step` the argument is traversed top-down to find the innermost redeces. Another reason is that function `rewrite_step` performs a lot of expensive comparisons.

Exercise Define a function `rewrite` that rewrites a term using the leftmost-innermost rewriting strategy. The only function that has to be rewritten is function `innermost`:

```
innermost rs x' x = if (inn x') !/=! x then ... else reduce rs x
```

where the `...` should be completed. The main idea here is to fzip `x'` and `out x`, and to use polytypic functions `changed` and `left` of type

```
polytypic changed  ::  f (a,a) (b,b) -> Bool
polytypic left     ::  f (a,a) (b,b) -> f a b
```

to obtain the leftmost-innermost reduced term. (*end of exercise*)

Avoiding unnecessary top-down traversals and comparisons

We want to obtain a function that rewrites a term in time proportional to the number of steps needed to rewrite the term. As a first step towards such a function, we replace the fixed-point computation by a double recursion. The double recursion avoids the unnecessary top-down traversals in search for the innermost redeces. The idea is to first recursively rewrite the children of the argument to normal form, and only then rewrite the argument itself.

For confluent and normalising term rewriting systems we have that first applying `rewrite'` to the subterms of the argument, and subsequently to the argument itself, gives the same result as applying function `rewrite'` to the argument itself.

```
rewrite' rs (inn x) = rewrite' rs (inn (fmap id (rewrite' rs) x))
```

It follows that function `rewrite'` can be written as a catamorphism, which uses function `rewrite'` in the recursive step. This version of function `rewrite` is called `rewritec`.

```
rewritec rs = cata frewrite
  where frewrite x = rewrite' rs (inn x)
```

Observe that in the recursive step, all subexpressions are in normal form. It follows that the only possible term that can be rewritten is the argument `inn x`. If `inn x` is a redex, then it is rewritten, and we proceed with rewriting the result. If `inn x` is not a redex, then `inn x` is in normal form. We adjust function `reduce` such that it returns `Nothing` if it does not succeed in rewriting its argument, and `Just x` if it does succeed with `x`.

```
rewritec rs  =  cata frewrite
  where
  frewrite x = maybe (inn x) (rewritec rs) (reduce rs (inn x))

reduce :: Eq a => [Rule f a] -> MuVar f a -> Maybe (MuVar f a)
reduce [] t               =  Nothing
reduce ((lhs,rhs):rs) t   =  case unify (lhs, t) of
                               Just s    ->  Just (subst s rhs)
                               Nothing   ->  reduce rs t
```

This function rewrites 2^8 much faster than the first definition of function `rewrite`, but it is still far from linear in the number of rewrite steps.

Efficient rewriting

A source of inefficiency in function `rewritec` is the occurrence of function `rewritec` in `frewrite`. If `reduce rs (inn x)` returns some expression `Just e`, `rewritec rs` is applied to `e`. When evaluating the expression `rewritec rs e` the whole expression `e` is traversed to find the innermost redeces, including all subterms which are known to be in normal form. For example, consider the expression `100 :*: 2`, where `2` and `100` abbreviate their equivalents written with `Succ` and `Zero`. Applying the second rule for `:*:`, this term is reduced to `(100 :*: 1) :+: 100`. Now, `rewritec rs` will traverse both subexpressions `100`, and find that they are in normal form, which we already knew. To avoid these unnecessary traversals, function `rewritec` is rewritten as follows. Instead of applying `rewritec rs` recursively to the reduced expression, we apply a similar function recursively to the right-hand side of the rule with which the expression is reduced. This avoids recursing over the expressions substituted for the variables in this rule, which are known to be in normal form. To define this function we use the polytypic version of function `zipWith`, called `pzipWith`. Function `pzipWith` is used in the definition of `frewrite` to zip the right-hand side of a rule with the expression obtained by substituting the appropriate expressions for the variables in this rule. This means that in case `pzipWith` encounters two arguments with a different outermost constructor, the left argument is a variable, and the right argument is an expression in normal form substituted for the variable. In that case we return the second argument. In case `pzipWith` encounters two arguments with the same outermost constructor, it tries to rewrite the zipped expression.

```
rewritec rs  =  cata frewrite
  where frewrite x = maybe (inn x) just (reduce rs (inn x))
        just = pzipWith frewrite fst snd

reduce :: Eq a =>
          [Rule f a] -> MuVar f a -> Maybe (MuVar f a,MuVar f a)
reduce [] t              =  Nothing
reduce ((lhs,rhs):rs) t  =  case unify (lhs,t) of
                              Just s    ->  Just (rhs,subst s rhs)
                              Nothing   ->  reduce rs t
```

Since the argument `t` of `reduce` does not contain variables, `t` does not contribute to bounds of the substitution array, and function `reduce` can be optimised as follows:

```
reduce [] t              =  Nothing
reduce ((lhs,rhs):rs) t  =  case unifyList' start [(lhs,t)] of
                              Just s    ->  Just (rhs,subst s rhs)
                              Nothing   ->  reduce rs t
  where start       = listArray (varbounds lhs) (repeat Nothing)
        varbounds l = (minimum v,maximum v) where v = vars l
```

The resulting rewrite function is linear in the number of reduction steps needed to rewrite a term to normal form. It rewrites the representation of 2^8 into the representation of 256 with the rules given for `Zero`, `Succ`, `:+:`, and `:*:` in the introduction about 500 times faster than the original specification of function `rewrite`. This function can be further optimised by partially evaluating with respect to the rules; we omit these optimisations.

6.2 Normalising sets of rewriting rules

Termination of function `rewrite` can only be guaranteed if its argument rules are normalising. A set of rules is normalising if all terms are rewritten to normal form (i.e. cannot be rewritten anymore) in a finite number of steps. It is undecidable whether or not a set of rewriting rules is normalising (unless all rules do not contain variables), but there exist several techniques that manage to prove normalising property for a large class of normalising rewriting rules. A technique that works in many cases is the method based on a well-known method of recursive path orderings, as developed by Dershowitz on the basis of a theorem of Kruskal, see [24]. In this section we will define a function `normalise` based on this technique.

```
normalise  ::  Eq a => [Rule f a] -> Bool
```

Note that if function `normalise` returns `False` for a given set of rules this does not necessarily mean that the rules are not normalising, it only means that function `normalise` did not succeed in constructing a witness for the normalising property of the rules.

The recursive path orderings
The recursive path orderings technique for proving the normalising property is rather complicated; it is based on a deep theorem from Kruskal. In this section we will see the technique in action; see [24] for the theory behind this technique.

A set of rules of type `[Rule f a]` is normalising according to the recursive path orderings technique if we can find an ordering on the constructors of the datatype `MuVar f a` such that each left-hand side of a rule can be rewritten into its right-hand side using a set of four special rules. These rules will be illustrated with the rewriting rules for `Zero`, `Succ`, `Add` and `Mul` given in the introduction:

```
Var 1 :++: VZero          ->  Var 1
Var 1 :++: VSucc (Var 2)  ->  VSucc (Var 1 :++: Var 2)
Var 1 :**: VZero          ->  VZero
Var 1 :**: VSucc (Var 2)  ->  (Var 1 :**: Var 2) :++: Var 1
```

We assume that the constructors of the datatype `VNumber` are ordered by `Var < VZero < VSucc < :++: < :**:`. The four rewriting rules with which left-hand sides have to be rewritten into right-hand sides are the following:

- Place a mark on top of a term. A mark is denoted by an exclamation mark `!`.
- A marked value `x` with outermost constructor `c` may be replaced by a value with outermost constructor `c' < c`, and with marked `x`'s occurring at the recursive child positions of `c'`. For example, suppose `y` equals `!(Var 1 :++: VSucc (Var 2))`, then `y -> VSucc y`, since `VSucc < :++:`.

- A mark on a value `x` may be passed on to zero or more children of `x`. For example, the mark on `y` in the above example may be passed on to the subexpression `VSucc (Var 2)`, so `!(Var 1 :++: VSucc (Var 2)) -> Var 1 :++: !(VSucc (Var 2))`.
- A marked value may be replaced by one of its children occurring at the recursive positions. For example, `!(VSucc (Var 2)) -> Var 2`.

Each of the right-hand sides of the rules for rewriting numbers can be rewritten to its left-hand side using these rules. For example,

```
  Var 1 :**: VSucc (Var 2)
-> { Rule 1 }
  !(Var 1 :**: VSucc (Var 2))
-> { Rule 2 }
  !(Var 1 :**: VSucc (Var 2)) :++: !(Var 1 :**: VSucc (Var 2))
-> { Rule 4 }
  !(Var 1 :**: VSucc (Var 2)) :++: Var 1
-> { Rule 3 }
  (Var 1 :**: !(VSucc (Var 2))) :++: Var 1
-> { Rule 4 }
  (Var 1 :**: Var 2) :++: Var 1
```

It follows that the set of rules for rewriting numbers is normalising.

Exercise Rewrite the left-hand sides into their corresponding right-hand sides for the other rules for rewriting numbers using the four special rewrite rules. (*end of exercise*)

Exercise Show that the following set of rewrite rules is normalising using the recursive path orderings technique.

$$\begin{array}{rcl} \neg\,(\neg\, x) & \rightarrow & x \\ \neg\,(x \vee y) & \rightarrow & \neg\, x \wedge \neg\, y \\ x \wedge (y \vee z) & \rightarrow & (x \wedge y) \vee (x \wedge z) \\ (x \vee y) \wedge z & \rightarrow & (x \wedge z) \vee (y \wedge z) \end{array}$$

(*end of exercise*)

Function `normalise`
A naive implementation of a function `normalise` that implements the recursive path orderings technique computes all possible orderings on the constructors, and tests for each ordering whether or not each left-hand side can be rewritten to its corresponding left-hand side using the four special rules. If it succeeds with one of the orderings, the set of rewriting rules is normalising. Since the four special rules themselves are not normalising this test may not terminate. To obtain a terminating function `normalise`, we implement a restricted version of the four special rules.

Thus, function `normalise` does not fully implement the recursive path orderings technique, but it still manages to prove the normalising property for a large class of sets of rewriting rules.

```
normalise rules  =  or [all (l_to_r ord) rules | ord <- allords]

allords  ::  [Mu f a -> Int]
l_to_r   ::  Eq a => (Mu f a -> Int) -> (Mu f a,Mu f a) -> Bool
```

Function `allords` generates all orderings, where an ordering is a function that given a value of the datatype returns an integer. Function `l_to_r` implements a restricted version of the four special rewrite rules.

Function `allords` is defined by means of two functions: function `perms`, which computes all permutations of a list, and function `fconstructors` which returns a represenation of the list of all constructors of a datatype. The definition of function `perms` is omitted.

```
polytypic fconstructors :: [f a b]
  = case f of
      f + g  ->  [Inl x | x <- fconstructors] ++
                 [Inr y | y <- fconstructors]
      _      ->  [undefined]


allords  =  map make_ord (perms fconstructors)
  where make_ord l x = index (fcnumber (out x)) (map fcnumber l)
        index n (m:ms)  |  n == m     =  0
                        |  otherwise  =  1 + index n ms
        index n []      = error "no index in list"
```

Exercise A straightforward optimisation of function `normalise` is obtained by only generating those orderings that do not immediately fail given the argument `rules`. For example, any ordering on `VNumber` with `:**: < :++:` will immediately fail on account of the fourth rewriting rule, which requires `:++: < :**:`. Define a function that takes a set of rules and generates all orderings that are not immediately ruled out on account of those rules. (*end of exercise*)

Finally, we have to implement function `l_to_r`. Given an ordering and a rewriting rule `(l,r)`, function `l_to_r` tries to rewrite `l` into `r`. Distinguish the following three cases:

- The outermost constructor of the right-hand side, `ocr`, is larger than the outermost constructor of the left-hand side, `ocl`, under the given ordering. In this case it is impossible to rewrite `l` into `r`, and function `l_to_r` returns `False`.
- `ocr` is smaller than `ocl` under the given ordering. In this case, function `l_to_r` computes the recursive components of the right-hand side. If there are no such,

it checks that the right-hand side itself is a subexpression of the left-hand side. If there are recursive components, function `l_to_r` checks that all of these are subexpressions of the left-hand side. For this purpose we define function `subexpr`, which takes two arguments, and determines whether or not the second argument is a subexpression of the first argument. A subexpression of `x` does not have to be a consecutive part of `x`, for example, the tree `Bin (Leaf 3) (Leaf 2)` is a subexpression of the tree `Bin (Bin (Leaf 3) (Leaf 4)) (Leaf 2)`. On lists, subexpressions are usually called subsequences.

```
subexpr      :: Eq a => Mu f a -> Mu f a -> Bool
subexpr l r  =
  pzipWith (and . fl_all)
           (uncurry (==))
           (\(x,y) -> (any (`subexpr` y) . fl_right . out) x)
           (l,r)
```

– The outermost constructors are equal under the given ordering. In this case, function `l_to_r` fzips the children of the left-hand side and the right-hand side. It checks that all pairs of values appearing at the parameter position consist of equal values, and it checks that there exists at least one recursive position pair. Furthermore, for each pair of values `(l,r)` appearing at a recursive position, `l_to_r ord (l,r)` has to hold.

We obtain the following definition of function `l_to_r`.

```
l_to_r ord (l,r)
  | ocl < ocr  = False
  | ocl > ocr  = let x = fl_right (out r) in
     if null x then subexpr l r else all (subexpr l) x
  | ocl == ocr =
     maybe undefined
           (\x -> parEq x && all' (l_to_r ord) (fl_right x))
           (fzip (out l , out r))
  where ocl = ord l
        ocr = ord r

all' p []  =  False
all' p xs  =  all p xs
```

where function `parEq` is defined in Section 5.

7 Conclusions and future work

This paper introduces polytypic programming: programming with polytypic functions. Polytypic functions are useful in applications where programs are datatype independent in nature. Typical example applications of this kind are the unification and rewriting system examples discussed in this paper, and there exist many more examples. Polytypic functions are also useful in the evolutionary process of developing complex software. Here, the important feature of polytypic functions is the fact that they adapt automatically to changing structure.

The code generated for programs containing polytypic functions is usually only slightly less efficient than datatype-specific code. In fact, polytypic programming encourages writing libraries of generally applicable applications, which is an incentive to write efficient code, see for example our library of rewriting functions.

The polytypic programming system Polyp is still under development. In the near future Polyp will be able to handle mutual recursive datatypes, datatypes with function spaces, and datatypes with multiple arguments.

Polytypic programming has many more applications than we have described in this paper. A whole range of applications can be found in adaptive object-oriented programming. Adaptive object-oriented programming is a kind of polytypic programming, in which constructor names play an important role. For example, Palsberg et al. [36] give a program that for an arbitrary datatype that contains the constructor names `Bind` and `Use`, checks that no variable is used before it is bound. This program is easily translated into a polytypic function, but we have yet to investigate the precise relation between polytypic programming and adaptive object-oriented programming.

Acknowledgements
During the two years we worked on polytypic programming we benefited from stimulating discussions and comments from Patrik Berglund, Graham Hutton, Mark Jones, Erik Meijer, Oege de Moor, Rinus Plasmeijer, Fritz Ruehr, Tim Sheard, Doaitse Swierstra, Phil Wadler, and Arjan van IJzendoorn. Oege de Moor, Graham Hutton and Arjan van IJzendoorn helped implementing predecessors of Polyp.

References

1. Patrik Berglund. A polytypic rewriting system. Master's thesis, Chalmers University of Technology, 1996. Forthcoming.
2. C. Böhm and A. Berarducci. Automatic synthesis of type λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.
3. Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic programming with relations and functors. To appear in Journal of Functional Programming, 1996.
4. 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.
5. 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.
6. R. Cockett and T. Fukushima. About `charity`. Unpublished article, see `http://www.cpsc.ucalgary.ca/projects/charity/home.html`, 1992.

7. M. Fiore et al. Domains and denotational semantics: History, accomplishments and open problems. Available via WWW: `http://www.cs.bham.ac.uk/~axj/papers.html`, 1996.
8. J.H. Fasel, P. Hudak, S. Peyton Jones, and P. Wadler. Sigplan Notices Special Issue on the Functional Programming Language Haskell. *ACM SIGPLAN notices*, 27(5), 1992.
9. Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions. In *Proceedings Principles of Programming Languages, POPL '96*, 1996.
10. M.M. Fokkinga. *Law and order in algorithmics.* PhD thesis, Twente University, 1992.
11. P. Freyd. Recursive types reduced to inductive types. In *Proceedings Logic in Computer Science, LICS '90*, pages 498–507, 1990.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1995.
13. T. Hagino. *Category Theoretic Approach to Data Types.* PhD thesis, University of Edinburgh, 1987.
14. P. Jansson. Polytypism and polytypic unification. Master's thesis, Chalmers University of Technology and University of Göteborg, 1995.
15. P. Jansson and J. Jeuring. Polytypic unification — implementing polytypic functions with constructor classes. Submitted for publication, see `http://www.cs.chalmers.se/~johanj/polytypism`, 1996.
16. P. Jansson and J. Jeuring. Type inference for polytypic functions. In preparation, see `http://www.cs.chalmers.se/~johanj/polytypism`, 1996.
17. C. Barry Jay. Polynomial polymorphism. In *Proceedings of the Eighteenth Australasian Computer Science Conference*, pages 237–243, 1995.
18. C. Barry Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.
19. J. Jeuring. Algorithms from theorems. In M. Broy and C.B. Jones, editors, *Programming Concepts and Methods*, pages 247–266. North-Holland, 1990.
20. J. Jeuring. Polytypic pattern matching. In S. Peyton Jones, editor, *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 238–248, 1995.
21. J. Jeuring. Polytypic data compression. In preparation, 1996.
22. Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925, pages 97–136. Springer-Verlag, 1995.
23. Mark P. Jones. Gofer. Available via ftp on `ftp.cs.nott.ac.uk`, 1995.
24. J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science*, pages 1–116. Oxford University Press, 1992.
25. K. Knight. Unification: A multidisciplinary survey. *Computing Surveys*, 21(1), 1989.
26. D.J. Lehmann and M.B. Smyth. Algebraic specification of data types: A synthetic approach. *Math. Systems Theory*, 14:97–139, 1981.
27. K.J. Lieberherr, I. Silva-Lepe, and C. Xiao. Adaptive object-oriented programming — using graph-based customization. *Communications of the ACM*, pages 94–101, 1994.
28. G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.
29. E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics.* Text and Monographs in Computer Science. Springer Verlag, 1986.
30. L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.
31. 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, FPCA '91, pages 124–144, 1991.

32. E. Meijer and G. Hutton. Bananas in space: Extending fold and unfold to exponential types. In S. Peyton Jones, editor, *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 324–333, 1995.
33. E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925, pages 228–266. Springer-Verlag, 1995.
34. O. de Moor. *Categories, relations and dynamic programming.* PhD thesis, Oxford University, 1992. Technical Monograph PRG-98.
35. O. de Moor. Categories, relations and dynamic programming. *Mathematical Structures in Computer Science*, 4:33–69, 1994.
36. J. Palsberg, C. Xiao, and K. Lieberherr. Efficient implementation of adaptive software. *TOPLAS*, 1995.
37. J.C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing 83*, pages 513–523, 1983.
38. Fritz Ruehr. Analytical and structural polymorphism. Unpublished manuscript, 1992.
39. Fritz Ruehr. *Analytical and Structural Polymorphism Expressed Using Patterns Over Types.* PhD thesis, University of Michigan, 1992.
40. T. Sheard. Type parametric programming. Oregon Graduate Institute of Science and Technology, Portland, OR, USA, 1993.
41. T. Sheard and N. Nelson. Type safe abstractions using program generators. Unpublished manuscript, 1995.
42. Tim Sheard. Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems*, 13(4):531–557, 1991.
43. A. Takano and E. Meijer. Shortcut deforestation in calculational form. In S. Peyton Jones, editor, *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, 1995.
44. P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, FPCA '89*, pages 347–359. ACM Press, 1989.

This article was processed using the LaTeX macro package with LLNCS style