Merging Monads and Folds for Functional Programming

Erik Meijer
Department of Computer Science, Utrecht University
PO Box 80.089, NL-3508 TB Utrecht, The Netherlands
email: `erik@cs.ruu.nl`

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

Abstract. These notes discuss the simultaneous use of generalised fold operators and monads to structure functional programs. Generalised fold operators structure programs after the decomposition of the value they consume. Monads structure programs after the computation of the value they produce. Our programs abstract both from the recursive processing of their input as well as from the side-effects in computing their output. We show how generalised monadic folds aid in calculating an efficient graph reduction engine from an inefficient specification.

1 Introduction

Should I structure my program after the decomposition of the value it consumes or after the computation of the value it produces?

Some [Bir89, Mee86, Mal90, Jeu90, MFP91] argue in favour of structuring programs after the decomposition of the value they consume. Such *syntax directed* programs are written using a limited set of recursion functionals. These functionals, called catamorphisms or *generalised fold operators* are naturally derived from the recursive structure of recursive datatypes. Fold operators satisfy a number of laws that support *calculating* with programs. Using these laws efficient implementation programs can be calculated from inefficient specification programs.

Others [Wad90, Wad95, Mog91, Pat95] argue in favour of structuring programs after the computation of value they produce. Such *semantics directed* programs are composed by gluing together computations using a limited set of operators. These *monad* operators are naturally derived from the plumbing structures that arise when computing values of monadic types. Programming with monads provides a uniform means of integrating effects such as I/O, updatable state, exceptions, nondeterminism, etc., into a purely functional language.

As we show in this paper, syntax directed programming and semantics directed programming neatly combine. This means that it is possible to abstract both from the recursive processing of the input as well as from the side-effects in computing the output. The resulting programs often are of an astonishing clarity and conciseness.

1.1 Overview

This paper is organised as follows. Section 2 shows that many functions defined on lists can be defined by folding using the standard function `foldr`, and it shows how one can calculate programs using properties of function `foldr`. Section 3 generalises function `foldr` to other algebraic datatypes. Section 4 introduces monads. Section 5 introduces generalised monadic folds. Section 6 shows how properties of monadic folds can be used to calculate an efficient lazy monadic interpreter from an inefficient one. Section 7 concludes the paper.

2 Folding is replacing constructors by functions

Just as in imperative languages where it is preferable to use structured iteration constructs such as **while**-loops and **for**-loops instead of unstructured gotos, it is advantageous to use structured recursion operators instead of unrestricted recursion when using a functional language. Programs that are similar in content and behaviour should be expressed in similar form [SJW79]. Structured programs are easier to reason about and more amenable to (possibly automatic) optimisations than their unstructured counterparts.

Consider the following functions from the Gofer prelude. Each of them is defined by explicit recursion over lists. We will show that they all follow the same recursive pattern. Thus, they can be written in such a way that this similarity can readily be recognised and exploited.

Function `head` returns the first element of a nonempty list, so `head [3, 2, 4]` = `3`.

```
head        :: [a] -> a
head (a:_)  =  a
```

Function `filter` takes a predicate `p` and a list `as`, and returns those elements of list `as` that satisfy `p`. For example, `filter even [2,3,4]` = `[2,4]`.

```
filter            :: (a -> Bool) -> ([a] -> [a])
filter p []       =  []
filter p (a:as)   =  let as' = filter p as
                     in if p a then a:as' else as'
```

Function `take` takes an integer `n` and a list `as`, and returns the list consisting of the first (at most) `n` elements of list `as`, for example `take 2 [4,2,6,7]` = `[4,2]`.

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

Given a binary function `f`, an accumulator `b`, and a list `as`, for example `as` = `[4,1,5]`, the expression `foldl f b as` denotes the value `((b ‘f‘ 4) ‘f‘ 1) ‘f‘ 5`. (Note: in Gofer one can write a function `f :: a -> b -> c` in between its arguments by using quotes.)

```
foldl              :: (b -> a -> b) -> b -> ([a] -> b)
foldl f b []        =  b
foldl f b (a:as)    =  foldl f (b ‘f‘ a) as
```

Function `foldr` takes a binary function `f`, a value `b`, and a list `as` and returns the value obtained by replacing in list `as` the nil constructor `[]` by the value `b`, and the constructor `(:)` by the operator `f`. For example, `foldr f b [3,2,6]` = `3 ‘f‘ (2 ‘f‘ (6 ‘f‘ b))`.

```
foldr              :: (a -> b -> b) -> b -> ([a] -> b)
foldr f b []        =  b
foldr f b (a:as)    =  a ‘f‘ (foldr f b as)
```

Each of these examples is defined by induction over lists and follows the recursive pattern of `foldr`. The only difference is in the value `b` that is returned for the empty list, and the function `f` that is used to combine the head of the list with the result of the recursive call of the function on the tail of the list. For example for `head` we have that `head []` = `error "head []"` and `head (a:as)` = `a ‘f‘ (head as)` where `f = \ a _ -> a`. As a result we conclude that we can write function `head` using function `foldr`.

```
head  =  foldr (\ a _ -> a) (error "head []")
```

For function `filter p` we have that `filter p []` = `[]` for the empty list and `filter p (a:as)` = `a ‘f‘ (filter p as)` for a nonempty list `a:as`, where function `f` is defined by `f = \ a as’ -> if p a then a:as’ else as’`, hence

```
filter p  =  foldr (\ a as’ -> if p a then a:as’ else as’) []
```

Expressing the other two example functions, `take` and `foldl`, in terms of `foldr` is slightly more involved.

Consider the function `take n :: [a] -> a`. In contrast to function `filter`, the first parameter of `take` is not fixed, i.e., `take (n+1) (a:as)` is defined in terms of `take n as`. This makes it impossible to define `take n` in terms of `foldr` directly. The trick then is to swap the two arguments of function `take`:

```
take'          :: [a] -> Int -> [a]
take' []       =  \ _ -> []
take' (a:as)   =  \ n -> case n of 0 -> []; n+1 -> a : take' as n
```

Now we see that we can express `take` in terms of `foldr` as follows

```
take n as
 =  foldr (\ a h -> \ n -> case n of 0 -> []; n+1 -> a : h n)
          (\ _ -> []) as n
```

To express `foldl` in terms of `foldr` we use the same method. Swap the last two arguments of `foldl`:

```
foldl'           :: (b -> a -> b) -> [a] -> b -> b
foldl' f []      =  \ b -> b
foldl' f (a:as)  =  \ b -> foldl' f as (b `f` a)
```

and conclude that we can express `foldl` in terms of `foldr` as follows:

```
foldl f b as  =  foldr (\ a h -> \ b -> h (b `f` a)) (\ b -> b) as b
```

The advice for defining a function `h :: [a] -> b` using `foldr (\ a b -> a `f` b) b` is to think inductively in the following way: "suppose I have obtained the value `b :: b` by recursively applying function `h` to the tail of a list `a:as`. How do I combine this via a function `f :: a -> b -> b` with the head `a` of the list to obtain the result of computing `h (a:as)`?". Finding an appropriate operation `f` is often suggested by typing considerations; given values `a :: a` and `b :: b` there often is only one obvious choice to construct a result ``a `f` b :: b``. Remember also that the result of recursively applying `h` might be a function. In that case the sought operation `f` has type `a -> (c -> d) -> (c -> d)` and constructs a function given a value and a function. After all, functional programming is programming with functions.

Exercise The 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 `(!!)` using `foldr`. (*end of exercise*)

2.1 Using `foldr` allows programs to be optimised

The functional `foldr` satisfies three fundamental laws. The first law seems rather obvious. It states that folding a list with the constructors functions `[]` and `(:)` is the identity on lists.

```
   foldr (:) []
=      (Id)
   id
```

The identity law for lists can be proved by induction over lists. The second law for function `foldr` is known as the *Fusion law*. The fusion law gives conditions under which intermediate values *produced* by folding can be eliminated.

```
   h . foldr f b = foldr g (h b)
⇐      (Fusion)
   h (a `f` b) = a `g` (h b)
```

Fusion is the free theorem [Wad89] of the functional `foldr`. 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. In this paper we will mostly consider finite lists.

As a simple application of fusion we prove that multiplication distributes over summing the elements of a list:

```
(n*) . foldr (+) 0  =  foldr ((+) . (n*)) 0                (1)
```

This example is somewhat perverse as the expression on the left needs fewer multiplications than the expression on the right; sometimes it is useful to *introduce* intermediate values instead of removing them.

To prove equality (1) we calculate as follows:

```
   (n*) . foldr (+) 0  =  foldr ((+) . (n*)) 0
⇐      (Fusion)
   n*(a + m)  =  n*a + n*m  ∧  n*0  =  0
≡      arithmetic
   True
```

The condition `n*0 = 0` is obtained by matching the folds in the example with the folds in (Fusion).

The third law is known as the *Acid Rain Theorem* [TM95], or `foldr/build`-rule [GLPJ93]. The acid rain theorem gives conditions under we can eliminate an intermediate value *consumed* by folding. Thus we get 'deforestation for free', which explains the name of the theorem.

```
   foldr f b . g (:) []  =  g f b
⇐      (Acid Rain)
   g :: (A -> b -> b) -> b -> (C -> b)
   for some fixed types A and C
```

Acid rain follows from the free theorem of the list producing function `g`. It formalises the intuition that first building an intermediate list using `g (:) []` and then replacing each constructor `(:)` by a function `f` and each constructor `[]` by a value `b`, yields the same effect as building the resulting value using function `f` and value `b` directly. The polymorphic type of function `g` enforces that the constructors `(:)` and `[]` are actually used in building a list using `g (:) []`. Acid rain is related to *deforestation* [Wad88] and *virtual data structures* [SdM93].

As a simple application of acid rain we prove that mapping a function over a list does not change the length of that list.

```
length . map h  =  length                (2)
```

The functions `map h` and `length` can both be defined in terms of `foldr` as follows:

```
map    :: (a -> b) -> ([a] -> [b])
map h  =  foldr ((:) . h) []

length  :: [a] -> Int
length  =  foldr (\ _ n -> n+1) 0
```

To prove equality (2) we observe that `map h` = `g h (:) []` where `g h f b = foldr (f . h) b` and `g h :: (a -> b -> b) -> b -> ([c] -> b)` whenever `h :: c -> a`. The actual proof is now a routine calculation:

```
   length . map h
=      definition of length, observation
   foldr (\ _ n -> n+1) 0 . g h (:) []
=      observation, (Acid Rain)
   g h (\ _ n -> n+1) 0
=      definition of g and length
   length
```

Equality (2) can also be proved with the fusion law.

2.2 Constructive algorithmics is calculating with programs

Fusion and Acid Rain are useful in calculating programs from their specification. Engineers often solve problems by applying differential and integral calculus. Similarly, in ‘Constructive Algorithmics’ computer programs are constructed by means of a program calculus. The central idea is to calculate *with* programs. Specifications and programs are considered to be formal objects in the same theoretical framework and subject to the same calculation rules. The issue of program correctness is reduced to the correct application of simple laws in a calculation:

```
  specification
=     law
  ...
=     law
  implementation
```

Most of the usual approaches to program correctness consist of a method to reason *about* programs. Thus one needs two languages: a programming language, and a language in which one can reason about programs. Program calculation requires just one language.

In order to apply calculation rules they should be easy to remember and easy to manipulate with. This is the reason why surface aspects (emphasis on suitable *notation*) play an important rôle in the field of constructive algorithmics. A rich set of laws is indispensable for effectively calculating with progams. This is where foundational aspects (emphasis on suitable *theory*) come into play.

3 Folding can be generalised to other types

Folding is not unique to lists. Other datatypes can be folded too by replacing their constructor functions by functions of the appropriate type.

3.1 Snoc-lists

The name `foldr` means 'folding from the right' and reflects the fact that `foldr` processes the elements of its argument list from right-to-left. The name `foldl` means 'folding from the left' and reflects the fact that `foldl` processes the elements of its argument list from left-to-right. This is often convenient because the result of processing the first $i-1$ elements of the list is available when processing the i-th element. As we have seen, the function `foldl` is not the natural fold operator over lists; rather it corresponds to a function that folds so called *snoc-lists*. Snoc-lists are lists that are built from left-to-right.

```
data Snoc a  =  II | (Snoc a) :%: a
```

To fold a snoc-list we recursively replace the constant constructor `II :: Snoc a` by a constant `nil :: b` and the constructor function `(:%:) :: Snoc a -> a -> Snoc a` by a function `snoc :: b -> a -> b`. We pack the functions `snoc` and `nil` that replace the constructor functions into a tuple to stress that they belong together.

```
foldS :: (b, b -> a -> b) -> (Snoc a -> b)
foldS (nil,snoc)
 = fS
```

```
   where
      fS II            =  nil
      fS (as :%: a)    =  (fS as) `snoc` a
```

Just like the fold operator on cons-lists it is the case that folding using the constructor functions is the identity on snoc-lists.

```
foldS (II,(:%:))  =  id
```

Function `foldS` also satisfies a fusion law and an acid rain theorem. Again these follow from "theorems for free!". The fusion law for snoc-lists is given by

```
   h . foldS (b,f)  =  foldS (h b,g)
⇐      (Fusion)
   h (b `f` a)  =  (h b) `g` a
```

while the acid rain theorem is given by

```
   foldS (b,f) . g (II,(:%:))  =  g (b,f)
⇐      (Acid Rain)
   g :: (b, b -> A -> b) -> (C -> b) for fixed types A and C
```

Exercise Define (using `foldr`) a coercion function `c2s :: [a] -> Snoc a` that transforms ordinary lists into snoc-lists. Then show (using fusion or acid rain for snoc-lists) that `foldS (b,f) . c2s  =  foldl f b`. (*end of exercise*)

The moral of the above exercise is that we may consider the standard notation for lists `[a,...,z]` to be an abbreviation for `(...(II:%:a):%:...):%:z` when processing a list from left-to-right using `foldl`, while at the same time we may consider it to be an abbreviation for `a:(...:(z:[])...)` when processing a list from right-to-left using `foldr`.

There remains a nasty problem with this approach of treating cons-lists as if they were snoc-lists, namely when we want to return a cons-list as the result of folding a list from the left. Simulating snoccing a value `a` at the right end of a list `as` by concatenation `as ++ [a]` leads to a quadratic time complexity. This can be circumvented by returning a real snoc-list and later coercing the intermediate list into a cons-list via a function `s2c :: Snoc a -> [a]`.

Exercise Define a function `s2c :: Snoc a -> [a]` and show how in general the intermediate list that arises in a composition `s2c . f` can be eliminated. (*end of exercise*)

3.2 Peano numerals

There are numerous ways of viewing natural numbers as inductively defined types, even though they are not explicitly defined as such in Gofer. One possible view of the naturals is the *Peano* view; a number is either zero, `0`, or the successor `n+1` of a number `n`. Folding a number `n` with a constant `zero` and a function `succ` boils down to the `n`-fold application of function `succ` to the constant `zero`, thus `foldN (zero,succ) n = succ (...(succ zero))`.

```
foldN :: (a,a -> a) -> (Int -> a)
foldN (zero,succ)
 = fN
   where
      fN 0      =  zero
      fN (n+1)  =  succ (fN n)
```

The fusion law for folding integers `foldN` is defined as

```
     f . foldN (a,g)  =  foldN (f a,h)
  ⇐      (Fusion)
     f (g a)  =  h (f a)
```

The acid rain theorem for function `foldN` is defined as

```
     foldN (a,f) . g (0,(+1))  =  g (a,f)
  ⇐      (Acid Rain)
     g :: (a,a -> a) -> (B -> a) for some fixed type B
```

Together with the identity axiom for numbers `foldN (0,(+1)) = id`, fusion implies induction over numbers. Let `p :: Int -> Bool` be a predicate over the natural numbers, then

```
     \ n -> (n, p n)
  =      (Id)
     (\ n -> (n, p n)) . foldN (0, (+1))
  =      (Fusion), assume p (n+1)  =  p n || p (n+1)
     foldN ((0,p 0),\ (n,b) -> (n+1, b || p (n+1)))
  =      (Fusion), assume p 0  =  True
     (\ n -> (n,True)) . foldN (0, (+1))
  =      (Id)
     \ n -> (n,True)
```

Hence we can conclude that a predicate `p` is true for all natural numbers if `p` is true for number `0` and `p (n+1)` follows from `p n`.

```
     p  =  \ n -> True
  ⇐      (Induction)
     (p (n+1)  =  p n || p (n+1))  ∧  (p 0  =  True)
```

Many familiar arithmetic functions on natural numbers can be defined by folding. For example, here are definitions for addition and multiplication:

```
(<+>), (<*>) :: Int -> Int -> Int

n <+> m  =  foldN (m,(+1)) n
n <*> m  =  foldN (0,(<+> m)) n
```

Adding a number `n` to a number `m` starts with `m` and then increments this number `n` times. Multiplying a number `m` by a number `n` starts with `0` and then increments this number `n` times with steps of size `m`.

The proof that multiplication distributes over addition nicely breaks down into two steps, one that follows from fusion and one that follows from acid rain.

```
  (n <+> m) <*> k
=      (Acid Rain)
  foldN (m <*> k, (<+> k)) n
=      (Fusion)
  (n <*> k) <+> (m <*> k)
```

The second step immediately follows from fusion, using the assumption that addition `(<+>)` is commutative. The first step is more interesting since it shows that the premise of the acid rain theorem can sometimes be somewhat counter intuitive.

```
  (n <+> m) <*> k
=      rearrange, definition of (<+>)
  (<*> k) (foldN (m, (+1)) n)
=      define: g = \ (a,f) -> foldN (foldN (a,f) m, f) n
  (<*> k) (g (0,(+1)))
=      definition of (<*> k)
  foldN (0,(<+> k)) (g (0,(+1)))
=      (Acid Rain)
  g (0,(<+> k))
=      definition of g
  foldN (foldN (0,(<+> k)) m, (<+> k)) n
=      definition of (<*>)
  foldN (m <*> k, (<+> k)) n
```

Exercise Show that addition `(<+>)` is associative, and then prove using fusion that `foldN (m, (<+> k)) n  =  n <+> (m <*> k)`. (*end of exercise*)

Exercise Define the predicate `even :: Int -> Bool` that tests whether or not a natural number is even by means of function `foldN`. (*end of exercise*)

Exercise Express exponentiation as a `foldN`. (*end of exercise*)

Exercise Write function `take` by folding over the naturals instead of over lists, i.e., use function `foldN` rather than function `foldr`. (*end of exercise*)

3.3 Binary trees

All example datatypes we have seen so far have a linear structure. Our first example of a type which exhibits a branching structure is *binary trees.* A binary tree consists of either a leaf that contains a value of type `a` or a node comprising a left and a right subtree.

```
data Tree a  =  Leaf a | Node (Tree a) (Tree a)
```

Folding a tree follows the pattern set by folding lists and numbers, namely replace the constructors by functions.

```
foldB :: (a -> b, b -> b -> b) -> (Tree a -> b)
foldB (leaf,node)
 = fB
   where
      fB (Leaf a)       =  leaf a
      fB (Node as as')  =  node (fB as) (fB as')
```

Just as with `foldr` and `foldN` we can define many functions in terms of folding trees with `foldB` instead of using explicit recursion.

Exercise Using function `foldB`, write a function `tips :: Tree a -> [a]` that returns the list of values that occur in the leaves of a tree. (*end of exercise*)

Exercise Function `foldB` satisfies a fusion law and an acid rain theorem too. Formulate the fusion law and the acid rain theorem for binary trees. (*end of exercise*)

3.4 Rose trees (Rhododendrons)

Things get a bit more interesting when a datatype is defined using another datatype, such as for example the datatype `Rose` of multiway branching trees. A node in such a tree has a list of subtrees.

```
data Rose a  =  Fork a [Rose a]
```

The datatype `Rose` uses the datatype of lists, `[]`, in its definition. The main idea in the definition of folding rose trees is to use `map` to apply `foldR` recursively to the elements in the list of subtrees.

```
foldR :: (a -> [b] -> b) -> (Rose a -> b)
foldR (fork)
 = fR
   where
      fR (Fork a ts)  =  fork a (map fR ts)
```

An example fold over rose trees is function `postOrder :: Rose a -> [a]` that enumerates the values of a tree in postorder:

```
postOrder  :: Rose a -> [a]
postOrder  =  foldR (\ a ass -> (concat ass) ++ [a])
```

Exercise Define a function `depth :: Rose a -> Int` that returns the length of the longest path from the root in a rose tree. (*end of exercise*)

3.5 Typed Lambda Expressions

This section defines a fold for a slightly more complicated datatype: typed lambda expressions and types [Set89, page 441].

A simple *type* is either a variable or a function type.

```
data Type  =  TVar String | Type :->: Type
```

A *typed lambda expression* is either a variable, an application, or a typed abstraction which associates a type with the bound variable.

```
data Expr  =  Var String | Expr :@: Expr | (String,Type) ::: Expr
```

Folds on the datatypes `Type` and `Expr` are now defined as follows.

```
foldT :: (String -> a,a -> a -> a) -> (Type -> a)
foldT (tvar,arrow)
 = fT
   where
      fT (Tvar s)    =  tvar s
      fT (s :->: t)  =  arrow (fT s) (fT t)

foldE :: (String -> b, b -> b -> b, (String,Type) -> b -> b) ->
         (Expr -> b)
foldE (var,apply,lambda)
 = fE
   where
      fE (Var x)        =  var x
      fE (f :@: a)      =  apply (fE f) (fE a)
      fE ((x,t) ::: b)  =  lambda (x,t) (fE b)
```

3.6 Mutual recursive datatypes

Folds can be defined for mutually recursive datatypes too. An example pair of mutually recursive datatypes is the pair of types `Zig` and `Zag`. These types can be used to implement lists in which the type of the elements alternate. We will see an alternative way to implement this kind of lists in the next subsection.

```
data Zig a b  =  Nil | Cins a (Zag b a)
data Zag a b  =  Nal | Cans a (Zig b a)
```

The folds on the datatypes `Zig` and `Zag` are mutually recursive too. The folds for `Zig` and `Zag` take as arguments two pairs: one corresponding to the `Zig` type, and one corresponding to the `Zag` type.

```
foldZig :: (c,a -> d -> c) -> (d,b -> c -> d) -> (Zig a b -> c)
foldZig (nil,cins) (nal,cans)
 = fZig
   where
      fZig Nil          =  nil
      fZig (Cins a z)   =  cins a (fZag z)

      fZag  =  foldZag (nil,cins) (nal,cans)

foldZag :: (c,a -> d -> c) -> (d,b -> c -> d) -> (Zag b a -> d)
foldZag (nil,cins) (nal,cans)
 = fZag
   where
      fZag Nal          =  nal
      fZag (Cans a z)   =  cans a (fZig z)

      fZig  =  foldZig (nil,cins) (nal,cans)
```

3.7 Other types

Even for datatypes containing function spaces, such as the datatype `D` defined below, it is possible to define folds, [Fre90, Pat94, MH95].

```
data D  =  Func (D -> D)
```

It is however unknown how to construct folds for *non regular* datatypes, datatypes in which the recursive uses of the datatype at the right-hand side of the definition have arguments differing from the left-hand side. An example of such a datatype is the type of ropes:

```
data Rope a b  =  Nil | Twist a (Rope b a)
```

A working, but not very elegant, approach would be to define a fold for each form of recursive use. This corresponds to replacing the type `Rope a b` by the types `Zig a b` and `Zag a b`. But this approach does not always apply. Consider for example the datatype `ListTree`.

```
data ListTree a  =  Leaf a | Branch (ListTree [a]) (ListTree [a])
```

4 Monads separate values and computations

So much for programming using folds. We now turn our attention to structuring programs in a complementary way, namely after the *computation* of their result value.

4.1 Exceptions

Suppose we are required to write a function that computes the resistance of a list of resistors put in parallel [PBM82, page 52].

```
resistance  :: [Float] -> Float
resistance  =  (1 /) . foldr (\ r r' -> 1/r + r') 0
```

If function `resistance` is applied to the empty list, or if any of the resistors in the list is zero, execution of the whole program in which `resistance` is called is aborted and a program error is reported by the Gofer interpreter: `Program error: {primDivInt 13 0}`. This behaviour will certainly not be appreciated by the users of the program. Since it is our mission to spread the use of functional programming something must be done about this.

A more graceful approach is to let `resistance` signal an exception when a division by zero occurs. This is similar to what the Gofer interpreter does itself; we don't get a core dump when evaluating the Gofer expression `13/0`.

To record exceptions in computing a value of type `a` we use the type `Maybe`.

```
data Maybe a  =  No | Yes a
```

The type of the function `resistance` is enhanced to `resistance :: [Float] -> Maybe Float`. This makes explicit that the computation may either fail and return an exception `No` or succeed and return a proper value `Yes r`. Since the type of function `resistance` has changed, its definition should be changed as well. At each recursive application of `resistance`, the form of the result is now checked by a case distinction on `Maybe Float`. Moreover an exception is raised if a division by zero occurs.

```
resistance :: [Float] -> Maybe Float
resistance
 = (\ mr -> case mr of
              No    -> No
              Yes r -> if r == 0 then No else Yes (1/r)
   )
   .foldr (\ r mr' -> case mr' of
                        No     -> No
                        Yes r' -> if r == 0
                                  then No
                                  else Yes (1/r + r')
          )
          (Yes 0)
```

There are four patterns that occur a number of times: raising an exception by returning `No`, returning a proper value by injecting it into type `Maybe` using the constructor function `Yes` (three times), dispatching on the two alternatives of type `Maybe` (twice), and a test on the second operand of division (twice). We name these recurring patterns `zeroM`, `resultM`, `bindM` and `divM` respectively.

```
zeroM  :: Maybe a
zeroM  =  No

resultM    :: a -> Maybe a
resultM a  =  Yes a

bindM         :: Maybe a -> (a -> Maybe b) -> Maybe b
ma `bindM` f  =  case ma of No -> No; Yes a -> f a

divM         :: Float -> Float -> Maybe Float
r `divM` r'  =  if r' == 0 then zeroM else resultM (r/r')
```

Using these functions, exception handling becomes well structured: clumsy case-distinctions are expressed once and for all and need not be repeated throughout the program text at each application of an exceptional function.

```
resistance :: [Float] -> Maybe Float
resistance rs
 = foldr (\ r mr' -> mr'               `bindM` \ r'  ->
                     1 `divM` r        `bindM` \ r'' ->
                     resultM (r'' + r')
         )
         (resultM 0)
         rs `bindM` \ r -> 1 `divM` r
```

4.2 Substituting leaves of a tree

An *association list* is a list of key/value-pairs. The value bound to a key is found using function `lookup`. In case a key has no associated value function `lookup` raises an exception.

```
data Assoc a b  =  a := b

lookup             :: Eq a => a -> [Assoc a b] -> Maybe b
a `lookup` table  =  case [ b | a' := b <- table, a' == a] of
                        []     -> zeroM
                        (a:_) -> resultM a
```

Consider now the following problem. Given a binary tree and an association list, replace the values in the tree by their associated values. Since we have to traverse the tree recursively to substitute the leaves it is natural to define the substitution function by folding with `foldB`. Since the substitution might fail due to the fact that some of the leaves have no associated value in the table it is natural to make use of the operations `resultM` and `bindM` on exceptions (you may try yourself to write function `subst` using explicit recursion and without the exception handling operators).

```
subst :: Tree a -> ([Assoc a b] -> Maybe (Tree b))
subst
 = foldB (\ a
          -> \ table -> a `lookup` table      `bindM` \ b ->
                        resultM (Leaf b)
         ,\ mbs mbs'
          -> \ table -> mbs table             `bindM` \ bs  ->
                        mbs' table            `bindM` \ bs' ->
                        resultM (Node bs bs')
         )
```

But here is yet another recurring pattern that can be abstracted away: the table is passed down the recursive calls of function `subst` explicitly. Our next goal is therefore to hide the plumbing of the table `table` through the function `subst`. For this purpose we define two new operations `resultR` and `bindR` that extend the previous operations `resultM` and `bindM` by passing along an extra argument. For completeness we also lift function `zeroM` to accept an argument. To enhance readability we abbreviate `r -> Maybe a` as `ReaderM r a`.

```
type ReaderM r a  =  r -> Maybe a

resultR    :: a -> ReaderM r a
resultR a  =  \ _ -> resultM a
```

```
bindR          :: ReaderM r a -> (a -> ReaderM r b) -> ReaderM r b
ma `bindR` f  =  \ r -> ma r `bindM` \ a -> f a r

zeroR  :: ReaderM r a
zeroR  =  \ _ -> zeroM
```

Rewriting function `subst` using the above operations gives us nearly the conciseness we are after.

```
subst :: Tree a -> ReaderM [Assoc a b] (Tree b)
subst
 = foldB (\ a        -> lookup a             `bindR` \ b ->
                        resultR (Leaf b)
         ,\ mbs mbs' -> mbs                  `bindR` \ bs  ->
                        mbs'                 `bindR` \ bs' ->
                        resultR (Node bs bs')
         )
```

In section 5.1 we return to this example in order to transform it into an even more compact form. Right now we continue looking for some more monads.

Exercise Define a version of function `subst` such that the table is not passed down the tree as an explicit argument [WM91]. Hint: de-lambda-lift function `subst`. Explain why this trick does not work for the type checking example in the next section. (*end of exercise*)

4.3 Type checking

A typed lambda expression is *type correct* if the types specified for the bound variables of the expression agree with the types demanded by the ways in which the bound variables are used. Function `flip (|-) :: Expr -> ReaderM [Assoc Var Type] Type` attempts to synthesise the type of an expression given a *basis* that associates variables with their declared types.

When checking an abstraction, the basis is extended to record the type associated with the bound variable of the abstraction. For this purpose we introduce the functions `fetchR` and `restoreR` that respectively return the basis as the result and resume a computation using a new basis.

```
fetchR  :: ReaderM r r
fetchR  =  \ r -> resultM r

restoreR       :: r -> ReaderM r a -> ReaderM r a
restoreR r ma  =  \ _  -> ma r
```

To check the type of a variable we look up its type in the basis. To typecheck an abstraction `(x,s) ::: b`, we typecheck the body `b` assuming that variable `x` has type `s`. If the resulting type is `t`, then the whole abstraction has type `s :->: t`. To typecheck an application `f :@: a` we ensure that the type of `f` is a function type `s :->: t` and that the type `s'` of `a` is the same as the argument type `s` expected by `f`. The result of applying `f` to `a` then has type `t`.

```
env |- expr
= foldE (\ x       -> lookup x
        ,\ f a     -> f                         `bindR` \ s ->
                      case s of
                        (s :->: t) -> a         `bindR` \ s' ->
                                      if s == s'
                                      then resultR t
                                      else zeroR
                        _          -> zeroR
        ,\ (x,s) b -> fetchR                    `bindR` \ env ->
                      restoreR (x := s : env) b `bindR` \ t   ->
                      resultR (s :->: t)
        ) expr env
```

4.4 Chopping trees, marking vertices

A directed graph with vertices of type `a` can be represented by a value of type `Rose a`. For example, a small graph with nine vertices labelled with strings is given below: (note that this is a situation where lists are cyclic).

```
root = Fork "root" [a,b,c,d,e,f,g,h,i,j]
       where
          a = Fork "a" [g,j];   b = Fork "b" [a,i]
          c = Fork "c" [e,h];   d = Fork "d" []
          e = Fork "e" [d,h,j]; f = Fork "f" [i]
          g = Fork "g" [f,b];   h = Fork "h" []
          i = Fork "i" [];      j = Fork "j" []
```

The task of function `dff` ('depth first forest') is to eliminate all edges to vertices that have been visited earlier during a depth-first search of the graph, leaving a spanning forest of the original graph. For the above graph function `dff` would return the graph:

```
root = Fork "root" [a,c]
       where
          a = Fork "a" [g,j]; b = Fork "b" []
          c = Fork "c" [e];   d = Fork "d" []
          e = Fork "e" [d,h]; f = Fork "f" [i]
          g = Fork "g" [f,b]; h = Fork "h" []
          i = Fork "i" [];    j = Fork "j" []
```

Function `dff` is defined using an auxiliary function `chop`. While traversing the graph function `chop` maintains a list of vertices that have already been visited. To ensure that the updated list of visited nodes is threaded through the computation, function `chop` returns the augmented list of visited nodes as part of the answer. Since subtrees might not appear in the depth first forest, function `chop` returns an exceptional tree as the other half of its answer. So function `chop` has type `Rose a -> ReaderM [a] (Maybe (Rose a),[a])`. Function `dff :: Rose a -> Rose a` calls `chop` with an empty list of visited vertices and throws away the final list of all reachable vertices returned by function `chop`.

```
dff g  =  g' where Yes (Yes g',_) = chop g []
```

We don't even attempt to first write function `chop` using `ReaderM`-operations. Instead, we abbreviate type `ReaderM s (a,s)` as `StateM s a` and lift the operations `resultR`, `bindR`, and `zeroR` in advance to work on type `StateM s a`. To manipulate the list of visited vertices we provide the operations `peekS` and `pokeS`.

```
resultS    :: a -> StateM s a
resultS a  =  fetchR `bind` \ s -> resultR (a,s)

bindS          :: StateM s a -> (a -> StateM s b) -> StateM s b
ma `bindS` f  =  ma `bindR` \ (a,s) ->  restoreR s (f a)

zeroS  :: StateM s a
zeroS  =  zeroR

peekS    :: (s -> a) -> StateM s a
peekS f  =  fetchR `bindR` \ s -> resultR (f s,s)

pokeS    :: (s -> s) -> StateM s ()
pokeS f  =  fetchR `bindR` \s -> resultR ((),f s)
```

To chop a node, we first check whether this node has been visited before. If so, the node is pruned. If not, we mark the node as visited, recursively chop the subtrees of the node and return a node that keeps only non-pruned subtrees as children.

```
chop :: Rose a -> StateM [a] (Rose a)
chop
 = foldR (\ a mts
          -> peekS (a `elem`)                   `bindS` \ v ->
             if v then
               resultS No
             else
               pokeS (a:)                       `bindS` \ _  ->
               accumulateS mts                  `bindS` \ ts ->
               resultS (Yes (Fork a [t | Yes t <- ts]))
         )
```

Function `accumulateS :: [StateM s a] -> StateM s [a]` collects the results of chopping a list of subtrees from left-to-right into a result list.

```
accumulateS  :: [StateM s a] -> StateM s [a]
accumulateS  =  foldr (\ mt mts -> mt           `bindS` \ t ->
                                   mts          `bindS` \ ts ->
                                   resultS (t:ts)
                      )
                      (resultS [])
```

An example use of function `dff` is *topological sorting*, which is treated in more depth in Launchbury's notes [Lau95].

```
topSort  :: Rose a -> [a]
topSort  =  reverse . postOrder . dff
```

For example `topSort root` = `["root", "c", "e", "h", "d", "a", "j", "g", "b", "f", "i"]`.

Exercise Derive using acid rain (or fusion) a version of function `topSort` that does not create intermediate values. It is easiest to start with `reverse . postOrd`. (*end of exercise*)

4.5 Monads generalise `bind` and `result`

Let `m` be a type constructor (such as `Maybe`). A function of type `f :: a -> b` can be generalised to a *monadic* function `f :: a -> m b` that accepts an argument of type `a` and returns a result of type `b` with a possible side-effect (such as raising an exception) captured by type constructor `m`.

To effectively use monadic functions we need an operation `bind` to apply a function `f` of type `f :: a -> m b` to an argument `ma` of type `m a` and a function `result :: a -> m a` to construct a result of type `m a`. The expression ``ma `bind` \ a -> f a`` reads as "evaluate `ma`, name the result `a`, and then evaluate `f a`". A type constructor `m` that comes equipped with these two operations is called a *monad*. The requirement on type constructor `m` can be captured by a constructor class as follows:

```
class  Monad m  where
   result  ::  a -> m a
   bind    ::  m a -> (a -> m b) -> m b
```

The components of a monad should satisfy a number of laws. The left-unit law and right-unit law say how to remove occurrences of function `result` from an expression.

```
result a `bind` k  =  k a
  k `bind` result  =  k
```

Furthermore, `bind` has to be associative. For each instance of the class `Monad` we have to verify these laws, but as usual we omit the proofs.

4.6 Exceptions

Datatype `Maybe` is an instance of class `Monad` as witnessed by the following instance declaration.

```
instance  Monad Maybe  where
   result a     =  Yes a
   ma `bind` f  =  case ma of No -> No; Yes a -> f a
```

As an aside we note that `bindM` can be defined by folding over the type `Maybe`. In fact, for many monads `m` the `bind` operation can be defined as a generalised fold over `m a` in a canonical way.

Exercise Define folding over type `Maybe` and then write `bindM` in terms of `foldM`. (*end of exercise*)

Exceptions are an example of a monad with a zero; a value `zero :: Monad m => m a` such that `` zero `bind` f  =  zero ``.

```
class  Monad m => Monad0 m  where
   zero  ::  m a

instance  Monad0 Maybe  where
   zero  =  No
```

Exceptions are also an example of a monad with a plus; a binary operator `(++) :: Monad0 m => m a -> m a -> m a` such that the laws `zero ++ ma  =  ma` and `ma ++ zero  =  ma` hold.

```
class  Monad0 m => MonadPlus m  where
   (++)  ::  m a -> m a -> ma

instance  MonadPlus Maybe  where
   ma ++ ma'  =  case ma of No -> ma'; _ -> ma
```

The value `zero` is used to throw an exception that can be caught using `(++)`.

4.7 Readers

The monad of *state readers* passes an extra argument on top of a base monad. We repeat the type of `Reader`.

```
type Reader m r a  =  r -> m a

instance  Monad m => Monad (Reader m r)  where
```

```
     result a     =  \ _ -> result a
     ma `bind` f  =  \ r -> ma r `bind` \a -> f a r

  instance  Monad0 m => Monad0 (Reader r m)  where
     zero  =  \ _ -> zero

  instance  MonadPlus m => MonadPlus (Reader m r)  where
     ma ++ ma'  =  \ r -> ma r ++ ma' r
```

The reader monad provides two extra operations to fetch and replace the argument passed under water, respectively.

```
  fetch  :: Monad m => Reader m r r
  fetch  =  \ r -> result r

  restore    :: Monad m => r -> Reader m r a -> Reader m r a
  restore r  =  \ ma -> \ _ -> ma r
```

4.8 State transformers

The monad of *state transformers* can be considered as an extra layer on top of the reader monad to thread around global state.

```
  type State m s a  =  Reader m s (a,s)

  instance  Monad (Reader m s) => Monad (State m s)  where
     result a     =  fetch `bind` \ s -> result (a,s)
     ma `bind` f  =  ma `bind` \ (a,s) -> restore s (f a)

  instance  Monad0 (Reader m s) => Monad0 (State m s)  where
     zero  =  zero

  instance  MonadPlus (Reader m s) => MonadPlus (State m s)  where
     (++)  =  (++)
```

Two additional operations on the state monad are `peek` and `poke`. These operations are used to inspect and modify the state.

```
  peek    :: Monad (Reader m s) => (s -> a) -> State m s a
  peek f  =  fetch `bind` \ s -> result (f s,s)

  poke    :: Monad (Reader m s) => (s -> s) -> State m s ()
  poke f  =  fetch `bind` \ s -> result ((),f s)
```

Exercise Simplify the operations `bind` and `result` of the state transformer monad as much as possible. In most texts on monads, state transformers are defined using these simplified operations. (*end of exercise*)

Exercise The identity monad is based on the type constructor `type Id a = a`. Give the instance declaration `Monad Id` for the identity monad. (*end of exercise*)

4.9 Do-notation

One advantage of being able to overload the monad operations `result` and `bind` (and `zero`) is that we can use special syntactic sugar for monadic programs. The `do`-notation as originally proposed by Launchbury [Lau93] and implemented in Gofer [Jon93] provides the following additional form of expressions:

$$\begin{array}{rcl} \textit{expr} & ::= & \texttt{do}\{\textit{qualifier};\ \ldots;\ \textit{qualifier};\ \textit{expr}\} \\ \textit{qualifier} & ::= & \textit{pat}\ \texttt{<-}\ \textit{expr} \\ & | & \textit{expr} \\ & | & \texttt{if}\ \textit{expr} \end{array}$$

The `do`-notation can be removed by repeatedly applying the following rules where e ranges over expressions, p over patterns, and qse over sequences of qualifiers ending in an expression.

$$\begin{array}{rcl} \texttt{do}\{e\} & = & e \\ \texttt{do}\{p\ \texttt{<-}\ e;\ qse\} & = & e\ \texttt{`bind`}\ \texttt{f where f}\ p\ \texttt{=}\ \texttt{do}\{qse\};\ \texttt{f \_ = zero} \\ \texttt{do}\{e;\ qse\} & = & e\ \texttt{`bind`}\ \texttt{\textbackslash\ \_ ->}\ \texttt{do}\{qse\} \\ \texttt{do}\{\texttt{if}\ e;\ qse\} & = & \texttt{if}\ e\ \texttt{then do}\{qse\}\ \texttt{else zero} \end{array}$$

Rewritten using the `do` notation, the typechecker starts to look comprehensive at last.

```
env |- expr
= foldE (\ x       -> lookup x
        ,\ f a     -> do{ (s :->: t) <- f
                        ; s' <- a
                        ; if s == s'
                        ; result t
                        }
        ,\ (x,s) b -> do{ env <- fetch
                        ; t <- restore (x := s : env) b
                        ; result (s :->: t)
                        }
        ) expr env
```

Exercise Formulate the three monad laws in terms of the `do`-notation. (*end of exercise*)

5 Monadic folds

In the previous sections we have discussed folds and monads and we have shown how to write programs of type `Monad m => [a] -> m b` (or any other recursive type instead of lists `[a]`) that use both concepts at once. If such functions process the results of their recursive calls in a fixed way we can make yet another abstraction step. Let's look at two example functions to see how this works.

The function `mapr` maps a monadic function over a list starting from the right. In contrast to the normal `map`, the order in which the applications are carried out does matter when computing in a monad.

```
mapr :: Monad m => (a -> m b) -> ([a] -> m [b])
mapr f
 = foldr (\a mbs -> do{bs <- mbs; b <- f a; result (b:bs)})
         (result [])
```

Recall the function `resistance` defined in Section 4.1. Written using the `do` notation this function reads as follows.

```
resistance :: [Float] -> Maybe Float
resistance rs
 = do{ s <- foldr (\ r mr' -> do{ r' <- mr'
                                ; r'' <- 1 `divM` r
                                ; result (r'' + r')
                                }
                  )
                  (result 0)
                  rs
     ; 1 `divM` s
     }
```

The correspondence between functions `mapr` and `resistance` is that both the operations used for folding the construction function `(:)` first bind the value of their recursive call. They are of the form:

```
\ a mb -> do{b <- mb; mcons a b}
```

To capture this specific pattern of folding lists within a monad we define a *monadic fold* on lists:

```
mfoldr :: Monad m => (a -> b -> m b) -> m b -> ([a] -> m b)
mfoldr mcons mnil
 = foldr (\ a mb -> do{b <- mb; mcons a b}) mnil
```

Programs become more structured and modular by identifying monadic folds as a distinguished pattern of recursion. Moreover we can specialise the basic fold laws for monadic folds so that the number of mechanical steps in proofs is reduced.

The identity law for monadic folds states that monadically folding a list with monadic constructors is equivalent to injecting the list into the monad directly.

```
     mfoldr (\ a as -> result (a:as)) (result []) as
  =     (Id)
     result as
```

The monadic fusion law states that the monadic composition of a function that distributes monadically over the function that replaces constructor function `cons` with a monadic fold, is again a monadic fold.

```
     do{b <- mfoldr f mb as; h b}  =  mfoldr g (do{b <- mb; h b}) as
  ⇐     (Fusion)
     do{c <- mc; b <- f a c; h b}  =  do{c <- mc; b <- h c; g a b}
```

The monadic acid rain theorem states that it is not necessary to first build a list within a monad and then fold that list, provided the function that builds the list is polymorphic enough.

```
     do{as <- g (\ a as -> result (a:as)) (result []) c
       ;  mfoldr f n as
       }  =  g f n c
  =     (Acid Rain)
     g :: Monad m => (A -> b -> m b) -> m b -> C -> m b
     for some fixed types A and C.
```

From a categorical point of view, monadic folds naturally arise from an adjunction between the category of algebras and homomorphisms and another category of algebras and homomorphisms built upon the Kleisli category [Fok94]. This is the approach taken in the language ADL [KL95] too. A disadvantage of monadic folds is that they are in some sense too specific and hence not all functions of type `[a] -> m b` can be written as a monadic fold.

An example of a function that cannot (easily) be written using a monadic fold is function `mapl`. The function `mapl` maps a monadic function over a list starting from the left.

```
mapl :: Monad m => (a -> m b) -> ([a] -> m [b])
mapl f
 = foldr (\ a mbs -> do{b <- f a; bs <- mbs; result (b:bs)})
         (result [])
```

The problem with function `mapl f` is that it does not bind the result of its recursive call on the tail of its argument list before using the head of the list. When we view the argument list of `mapl` as a snoc-list and rewrite function `mapl` as a fold left, binding the result of the recursive call happens first.

```
mapl :: Monad m => (a -> m b) -> ([a] -> m [b])
mapl f
 = foldl (\ mbs a -> do{bs <- mbs; b <- f a; result (bs ++ [b])})
         (result [])
```

This specific pattern of folding lists within a monad is captured by the *monadic fold left* on lists:

```
mfoldl :: Monad m => (b -> a -> m b) -> m b -> ([a] -> m b)
mfoldl msnoc mnil
 = foldl (\ mb a -> do{b <- mb; msnoc b a}) mnil
```

Now we can define function `mapl` as a monadic fold: `mapl f = mfoldl (\ bs a -> do{b <- f a; result (bs ++ [b])}) (result [])`.

Exercise Formulate the fusion law and acid rain theorem for function `mfoldl`. (*end of exercise*)

In the rest of this section we give some more examples of monadic folds on lists. The function `accumulate` collects the results of executing a list of computations from left-to-right.

```
accumulate  :: Monad m => [m a] -> m [a]
accumulate  =  foldr (\ ma mas -> do{ a <- ma
                                    ; as <- mas
                                    ; result (a:as)
                                    }
                     )
                     (result [])
```

Using function `mfoldl`, function `accumulate` is defined as follows.

```
accumulate  :: Monad m => [m a] -> m [a]
accumulate  =  mfoldl (\ as ma -> do{a <- ma; result (as ++ [a])})
                      (result [])
```

The function `ignore` evaluates its argument only for its side-effect and throws away the result

```
ignore     :: Monad m => m a -> m ()
ignore ma  =  do{ma ; result ()}
```

The function `sequence` is used to execute a list of computations from left-to-right only for their side effects.

```
sequence  :: Monad m => [m a] -> m ()
sequence  =  ignore . accumulate
```

Exercise Using fusion and the monad laws calculate and efficient version of function `sequence` that does not built an intermediate list. (*end of exercise*)

5.1 Substituting leaves revisited

Just as function `mfoldr` on lists, function `mfoldB` captures passing on the monad through the recursive parts of the type.

```
mfoldB :: Monad m => (a -> m b, b -> b -> m b) -> (Tree a -> m b)
mfoldB (mleaf, mnode)
 = foldTree (\ a      -> mleaf a
            ,\ mb mb' -> do{b <- mb; b' <- mb'; mnode b b'}
            )
```

Note that evaluating the recursive calls to function `mfoldB` in the opposite order results in a different function.

Using the monadic fold over trees `mfoldB`, function `subst` can be written as:

```
subst  =  mfoldB (\ a      -> do{b <- lookup a; result (Leaf b)}
                 ,\ bs bs' -> result (Node bs bs')
                 )
```

Compare this definition with the definition of the same function in Section 4.2.

5.2 Renaming Lambda Expressions

Here are the monadic folds over expressions and types.

```
mfoldT :: Monad m => (String -> m a,a -> a -> m a) -> Type -> m a
mfoldT (mtvar,marrow)  =  foldT (mtvar
                                ,\ ms mt -> do{ s <- ms
                                              ; t <- mt
                                              ; s 'marrow' t
                                              }
                                )

mfoldE :: Monad m =>
          (String -> m a
          ,a -> a -> m a
          ,(String,Type) -> a -> m a
          ) -> (Expr -> m a)
mfoldE (mvar, mapply, mlambda)
= foldE (\ x        -> mvar x
        ,\ mf ma    -> do{f <- mf; a <- ma; mapply f a}
        ,\ (x,t) ma -> do{a <- ma; mlambda (x,t) a}
        )
```

The typechecker `(|-)` is a good example of a function that cannot be expressed (simply) as a monadic fold. The problem is that the body of an abstraction can only be checked *after* the basis is extended with an additional type assumption. An example that does conform to the recursion pattern of function `mfoldE` is Wadler's rename function for lambda expressions [Wad90]. This functions uses the state monad to supply a source of new variable names. The function call `substv x y b` replaces all free occurrences of variable `x` in `b` by variable `y`.

```
rename :: Expr -> State Id String Expr
rename
 = mfoldE (\ x       -> result (Var x)
          ,\ f a     -> result (f :@: a)
          ,\ (x,t) b -> do{ y <- new_name
                          ; result ((x,t) ::: (substv x y b))
                          }
          )
```

Exercise Write a renamer `rename :: Expr -> Reader (State Id String) [Assoc String String] Expr` that does the substitution `substv x y b` on the fly. Can you write this function as a monadic fold over expressions? (*end of exercise*)

6 Calculating abstract machines

Until now we have mainly concentrated on using monadic folds for defining toy functions. This section exemplifies calculating a more involved program in this style. The example discussed in this section is taken from [Joh95], in which Johnsson calculates an efficient but complex functional interpreter (or abstract machine) from a simple, evidently correct, and inefficient interpreter. The calculation in [Joh95] uses unfold-fold transformations. Here we redo the same calculation, using monadic folds and fusion instead, resulting in a shorter calculation, in which the focus is on the important steps.

6.1 A state monad

In the definition of the G-machine in Section 6.3 we use a special instance of the state monad of Section 4.8. Sharing and graph manipulation that goes on in a real graph reduction implementation can be modelled using the state monad. Suppose we have a type `Graph`, then the state monad we need is defined as follows.

```
type St a  =  Graph -> (a,Graph)
```

We obtain the same type if we define `St a` as `State Id Graph a`. For concreteness sake we simplify the `result` and `bind` we obtain from `State Id Graph a`.

```
instance  Monad St  where
  result a     =  \ g -> (a,g)
  ma `bind` f  =  \ g -> let (a,g1) = ma g in f a g1
```

A graph `g :: Graph` contains nodes (of type `Node`) linked by pointers (of type `Pointer`). Folding a node means replacing constructor functions by arbitrary functions.

```
data Node =  Nint Int
          |  Nadd Pointer Pointer
          |  Napp Name [Pointer]

foldNode :: (Int -> a
            ,Pointer -> Pointer -> a
            ,Name -> [Pointer] -> a
            ) -> (Node -> b)
foldNode (nint,nadd,napp)
 = fNode
   where
      fNode (Nint i)     =  nint i
      fNode (Nadd p q)   =  nadd p q
      fNode (Napp n ps)  =  napp n ps
```

The types `Graph` and `Pointer` are not relevant in this context. They are provided as abstract types with three operations. Function `store` stores a node in a graph, and returns a pointer to where it is stored. Function `get` fetches a node from the graph given a pointer. Function `update` stores a node at a given location.

```
store   ::  Node -> St Pointer
get     ::  Pointer -> St Node
update  ::  Node -> Pointer -> St ()
```

These functions satisfy a number of useful, and more or less straightforward, laws that we assume as axioms. The first law is the *store-get law* and says that storing a value and then immediately fetching it to use it in some other function, can be replaced by storing the value *and* passing it on for use in the other function, so without fetching it from the state.

```
  do{p <- store v; w <- get p; f w}
=      (Store-Get)
  do{p <- store v; f v}
```

The second law is a kind of dual of the first law. The *get-store* law says that fetching a node and then immediately storing it is the identity.

```
  do{p <- f; n <- get p; store n}
=     (Get-Store)
  f
```

The third law is the *store-update law* and says that a store followed by an update with value `w` of the same pointer can be replaced by just storing value `w`.

```
  do{p <- store v; update w p; f}
=     (Store-Update)
  do{p <- store w; f}
```

The fourth law, the *move-store law*, says that if `p` does not occur free in `f`, and `a` does not appear free in `v`, then storing node `v` and computing value `a` can be swapped.

```
  do{p <- store v; a <- f; k}
≅     (Move-Store)
  do{a <- f; p <- store v; k}
```

The left-hand side and right-hand side are not equal, since they do different things to the store, but apart from that they are equivalent in the sense that in all contexts `do{p <- store v; x <- f; k}`, simply using `k` produces the same result.

Exercise The function `new :: St Pointer` allocates a new unused pointer in the graph. Define function `store` in terms of function `new` and `update`. (*end of exercise*)

6.2 A tiny first order language

We consider a tiny first order language in which all values are integers, and the only operator is addition. Formal parameters are encoded by de Bruijn indices. The expressions are elements of the datatype `Expr`, which is slightly different from the datatype `Expr` defined in Section 3.5.

```
data Expr  =  Var Int
           |  Con Int
           |  Add Expr Expr
           |  App Name [Expr]

foldEx :: (Int -> a
          ,Int -> a
          ,a -> a -> a
          ,Name -> [a] -> a
```

```
           ) -> (Expr -> a)
foldEx (var,con,add,app)
 = fEx
   where
      fEx (Var i)     =  var i
      fEx (Con n)     =  con n
      fEx (Add e e')  =  add (fEx e) (fEx e')
      fEx (App f es)  =  app f (map fEx es)
```

A program is represented as an association list in which each pair consists of a function name and a right-hand side expression for that function name. For example, the concrete program

```
f x y  =  y + 1
g x    =  g x
main   =  f (g 2) 3
```

is represented by the following table.

```
["f"    := Add (Var 2) (Int 1)
,"g"    := App "g" [Var 1]
,"main" := App "f" [App "g" [Int 2],Int 3]
]
```

Given a global program environment `definitions` in which names are bound to expressions (the right-hand sides in a program), function `rhs` takes a name, and returns the right-hand side of the function.

```
rhs f  =  e where (Yes e) = lookup f definitions
```

The naive G-machine builds a graph from an expression. The graph is built bottom-up, so the program can be structured after the structure of its input. This is a perfect opportunity to exploit a monadic fold over expressions.

```
mfoldEx :: Monad m =>
           (Int -> m a
           ,Int -> m a
           ,a -> a -> m a
           ,Name -> [a] -> m a
           ) -> (Expr -> m a)
mfoldEx (var,con,add,app)
 = foldEx (var
          ,con
          ,\ ms mt -> do{s <- ms; t <- mt; add s t}
          ,\ f mas -> do{as <- accumulate mas; app f as}
          )
```

Note that function `accumulate` threads the state from left to right through the list of results. Threading the state from right to left gives another definition of a monadic expression fold.

6.3 The G-machine

In the naive G-machine, evaluating a function application is done by instantiating the right-hand side of the function definition given the actual arguments of the function application. A graph is built for the right-hand side, given a stack of pointers into the argument graphs, and the resulting graph is evaluated. Function `build` builds a graph given a stack of pointers and an expression.

```
build :: [Pointer] -> Expr -> St Pointer
build actuals
 = mfoldEx (\ i    -> result (actuals!!i)
           ,\ n    -> store (Nint n)
           ,\ p p' -> store (Nadd p p')
           ,\ f ps -> store (Napp f ps)
           )
```

Function `eval` evaluates a graph with root `p` and updates `p` with the resulting value. Note the intricate mutual recursion between functions `eval` and `evalNode`.

```
eval    :: Pointer -> St Pointer
eval p  =  do{ n <- get p
             ; r <- evalNode n
             ; update r p
             ; result p
             }

evalNode :: Node -> St Node
evalNode
 = foldNode
     (\ n     -> result (Nint n)
     ,\ p1 p2 -> do{ q1 <- eval p1
                   ; q2 <- eval p2
                   ; Nint n1 <- get q1
                   ; Nint n2 <- get q2
                   ; result (Nint (n1+n2))
                   }
     ,\ f ps  -> do{ ts <- accumulate (map eval ps)
                   ; p <- build ts (rhs f)
                   ; q <- eval p
                   ; get q
                   }
     )
```

The G-machine evaluates an expression `e` by first building a graph for expression `e` and then evaluating the resulting graph.

```
machine        :: [Pointer] -> Expr -> St Pointer
machine ps e   =  do{p <- build ps e; eval p}
```

Evaluating an expression thus is rather inefficient: function `build ps` builds a complete graph, which is subsequently consumed by function `eval`. Since `build ps` is a monadic fold expression, we can try to apply fusion for monadic expression folds to avoid building the graph.

The above definitions slightly differ from Johnsson's. The difference lies in the fact that `eval` returns a `St Pointer` instead of a `St Node`. To obtain Johnsson's machine it suffices to bind function `get` to the above machine. Fusion applies to the above definitions, but not to Johnsson's definitions.

6.4 Fusion for monadic expression folds

Since a monadic expression fold is defined in terms of a normal expression fold, we can apply fusion for expression folds to obtain a fusion law for monadic expressions folds. Fusion for monadic expression folds reads as follows.

```
   do{r <- mfoldEx f g h j x; k r}
=      (Fusion)
   mfoldEx f' g' h' j' x
```

provided four conditions hold. The first two conditions can be seen as definitions of the functions `f'` and `g'`.

```
do{x <- f y; k x}  =  f' y
do{x <- g y; k x}  =  g' y
```

The third condition requires that function `k` distributes over function `h` in a monadic way.

```
   do{s <- ms; t <- mt; r <- h s t; k r}
=
   do{s <- ms; p <- k s; t <- mt; q <- k t; h' p q}
```

The fourth and last condition requires that function `k` distributes over function `j`, again taking account of the monad.

```
   do{t <- accumulate ts; p <- j n t; k p}
=
   do{t <- accumulate (map (`bind` k) ts); j' n t}
```

6.5 Fusion applies to the G-machine

We apply the monadic expression fold fusion law to the composition of functions `do{p <- build ps x; eval p}`. We have

```
        do{p <- build ps x; eval p}  =  mfoldEx f' g' h' j' x
```

provided the components of the folds satisfy the conditions of the fusion law. We verify these conditions in turn.

The first condition
The first condition of the monadic expression fold fusion law can be seen as a definition of function `f'`. This function can be simplified slightly.

```
     do{p <- result (ps!!n); eval p}
   =     left-unit law
     eval (ps!!n)
```

So we define function `f'` by

```
  f' n  =  eval (ps!!n)
```

The second condition
The second condition of the monadic expression fold fusion law can be used as a definition too, now of function `g'`. Again, function `g'` can be simplified.

```
     do{p <- store (Nint i); eval p}
   =     definition of eval
     do{p <- store (Nint i); n <- get p; r <- evalNode n;
        update r p; result p}
   =     store-get law
     do{p <- store (Nint i); r <- evalNode (Nint i); update r p;
        result p}
   =     definition of evalNode
     do{p <- store (Nint i); r <- result (Nint i); update r p;
        result p}
   =     left-unit law
     do{p <- store (Nint i); update (Nint i) p; result p}
   =     store-update law
     do{p <- store (Nint i); result p}
   =     right-unit law
     store (Nint i)
```

So we define function `g'` by

```
g' i  =  store (Nint i)
```

The first two steps of the above derivation can be used in the subsequent derivations too. We name the equality obtained the *eval-store law*.

```
  do{p <- store v; eval p}
=      (Eval-Store)
  do{p <- store v; r <- evalNode v; update r p; result p}
```

The third condition
The third condition of the monadic expression fold fusion law is a real condition, and requires a more involved calculation.

```
  do{s <- ms; t <- mt; p <- store (Nadd s t); eval p}
=      eval-store law
  do{s <- ms; t <- mt; p <- store (Nadd s t);
     r <- evalNode (Nadd s t); update r p; result p}
=      definition of evalNode
  do{s <- ms; t <- mt; p <- store (Nadd s t); p1 <- eval s;
     p2 <- eval t; Nint n1 <- p1; Nint n2 <- p2;
     r <- result (add n1 n2); update r p; result p}
≅      move-store law
  do{s <- ms; t <- mt; p1 <- eval s; p2 <- eval t;
     n1 <- get p1; n2 <- get p2; r <- result (add n1 n2);
     p <- store (Nadd s t); update r p; result p}
≅      rearrange terms
  do{s <- ms; p1 <- eval s; t <- mt; p2 <- eval t;
     n1 <- get p1; n2 <- get p2; r <- result (add n1 n2);
     p <- store (Nadd s t); update r p; result p}
```

The last step is justified by the same reasoning as the move-store law: the two expressions do different things to the store, but the result of the expressions is the same. We have calculated an expression of the desired form, i.e., a function that first evaluates the arguments and then processes them. So we can define function `h'` as the last two lines of the last expression in the above calculation. This expression can be simplified as follows.

```
  do{n1 <- get p1; n2 <- get p2; r <- result (add n1 n2);
     p <- store (Nadd s t); update r p; result p}
```

```
=     left-unit law
  do{n1 <- get p1; n2 <- get p2; p <- store (Nadd s t);
     update (add n1 n2) p; result p}
=     store-update law
  do{n1 <- get p1; n2 <- get p2; p <- store (add n1 n2);
     result p}
=     right-unit law
  do{n1 <- get p1; n2 <- get p2; store (add n1 n2)}
```

So we define function `h'` by

```
h' p q  =  do{m <- get p; n <- get q; store (add m n)}
```

The fourth condition
Finally, for the fourth condition of the monadic expression fold fusion law we calculate as follows.

```
  do{t <- accumulate ts; p <- store (Napp n t); eval p}
=     eval-store law
  do{t <- accumulate ts; p <- store (Napp n t);
     r <- evalNode (Napp n t); update r p; result p}
=     definition of evalNode
  do{t <- accumulate ts; p <- store (Napp n t);
     t' <- accumulate (map eval t); s' <- build t' (rhs n);
     s <- eval s'; r <- get s; update r p; result p}
≅     move-store law
  do{t <- accumulate ts; t' <- accumulate (map eval t);
     s' <- build t' (rhs n); s <- eval s'; r <- get s;
     p <- store (Napp n t); update r p; result p}
≅     law for accumulate and eval below
  do{t <- accumulate (map (`bind` eval) ts);
     s' <- build t (rhs n); s <- eval s'; r <- get s;
     p <- store (Napp n t); update r p; result p}
```

The law for `accumulate` and `eval` applied above reads:

```
  do{ts' <- accumulate ts; t <- accumulate (map eval ts'); k}
≅
  do{t <- accumulate (map (`bind` eval) ts); k}
```

This law can be proved by induction over the actual arguments `ts`. The last expression of the above calculation is an expression of the desired form. For the definition of `j'` we simplify the composition of all but the first functions.

```
     do{s' <- build t (rhs n); s <- eval s'; r <- get s;
        p <- store (Napp n t); update r p; result p}
  =      definition of machine
     do{s <- machine t (rhs n); r <- get s; p <- store (Napp n t);
        update r p; result p}
  =      store-update law
     do{s <- machine t (rhs n); r <- get s; p <- store r; result p}
  =      right-unit law
     do{s <- machine t (rhs n); r <- get s; store r}
  =      get-store law
     machine t (rhs n)
```

So we define function `j'` by

```
   j' n t  =  machine t (rhs n)
```

The result
Using fusion for monadic expression folds, we have derived the following optimised version for `machine ps`.

```
   machine ps
    = mfoldEx (\ i    -> eval (ps!!i)
              ,\ n    -> store (Nint i)
              ,\ p p' -> do{ Nint m <- get p
                           ; Nint n <- get p'
                           ; store (Nint (m+n))
                           }
              ,\ f ps -> machine ps (rhs f)
              )
```

In a lazy functional language, binding `get` to this abstract machine results in the same machine as Johnsson derives. To derive this machine, Johnsson used about fifty percent more calculation steps.

7 Conclusion

Imperative languages such as C provide a fixed set of control structures such as **while**- and **for**-loops. These kind of loops are convenient for processing iterative structures such as arrays and files but are less useful for processing other (branching) structures such as trees. On the other hand, *generalised fold operators* are custom made control structures that exactly match the datatypes one wants to traverse. Folding structures functions by the way they consume their arguments.

Imperative languages such as C (or logic languages such as Prolog) provide a fixed set of computations such as state-based computations (or non-deterministic computations). On the other hand *monads* are custom made notions of computations that exactly match the particular computation one wants to perform. Monads structure functions by the way they compute their results.

We have shown that is is possible to use both ideas at once, leading to concise and clear programs. Sometimes processing the recursive calls of folding a datatype within a monad follows a fixed pattern. In that case we can even take a larger notational shortcut by using a *generalised monadic fold* operator. Generalised monadic fold operators are the natural lifting of normal generalised fold operators to the Kleisli category. Despite their mathematical elegance, the recursion patterns captured by generalised monadic fold operators are often too specific to be useful.

Acknowledgements
We thank Doaitse Swierstra for stimulating functional programming in Utrecht, Maarten Fokkinga for his rigorous explanation of monadic folds and maps, and Graham Hutton, Luc Duponcheel, Jeroen Fokker, Thomas Johnsson, and Doaitse Swierstra for their helpful comments and remarks.

References

[Bir89] 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.

[Fok94] M.M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28, University of Twente, June 1994.

[Fre90] P. Freyd. Recursive types reduced to inductive types. In *Proceedings Logic in Computer Science LICS90*, pages 498–507, 1990.

[GLPJ93] A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Proceedings Functional Programming Languages and Computer Architecture FPCA93*, pages 223–232, 1993.

[Jeu90] J. Jeuring. Algorithms from theorems. In M. Broy and C.B. Jones, editors, *Programming Concepts and Methods*, pages 247–266. North-Holland, 1990.

[Joh95] Thomas Johnsson. Fold-Unfold Transformations on State Monadic Interpreters. In *Proceedings of the Glasgow Functional Programming Workshop, Ayr 1994*, Workshops in Computing. Springer Verlag, 1995. (To appear).

[Jon93] Mark P. Jones. Release notes for Gofer 2.28. Included as part of the standard Gofer distribution, February 1993.

[KL95] R.B. Kieburtz and J. Lewis. Programming with algebras. In J. Jeuring and E. Meijer, editors, *Lecture Notes on Advanced Functional Programming Techniques*, LNCS. Springer-Verlag, 1995.

[Lau93] J. Launchbury. Lazy imperative programming. In *Proceedings ACM Sigplan Workshop on State in Programming Languages*, 1993. YALEU/DCS/RR-968, Yale University.

[Lau95] J. Launchbury. Graph algorithms with a functional flavour. In J. Jeuring and E. Meijer, editors, *Lecture Notes on Advanced Functional Programming Techniques*, LNCS. Springer-Verlag, 1995.

[Mal90] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[Mee86] L. Meertens. Algorithmics—towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334. North–Holland, 1986.

[MFP91] 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 FPCA91, Cambridge, Massachusetts*, pages 124–144, 1991.

[MH95] E. Meijer and G. Hutton. Bananas in space: Extending fold and unfold to exponential types. In S. Peyton Jones, editor, *Proceedings Functional Programming Languages and Computer Architecture FPCA95*, 1995.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[Pat94] R. Paterson. Control structures from types. `rap@doc.ic.ac.uk`, 1994.

[Pat95] R. Paterson. Notes on computational monads. `rap@doc.ic.ac.uk`, 1995.

[PBM82] G.N. Franz P.B. Brown and H. Moraff. *Electronics for the Modern Scientist.* Elsevier Science Publishing Co., Inc., 1982.

[SdM93] S.D. Swierstra and O. de Moor. Virtual datastructures. In Helmut Partsch Berhard Möller and Steve Schuman, editors, *Formal Program Development*, LNCS 755, pages 355–371. Springer-Verlag, 1993.

[Set89] R. Sethi. *Programming Languages: Concepts and Structures.* Addison-Wesley, 1989.

[SJW79] W. Strunk Jr. and E.B. White. *The Elements of Style.* MacMillan Publishing Co., Inc., 1979. Third Edition.

[TM95] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In S. Peyton Jones, editor, *Proceedings Functional Programming Languages and Computer Architecture FPCA95*, 1995.

[Wad88] P. Wadler. Deforestation: transforming programs to eliminate trees. In *Proceedings European Sytmposium on Programming, ESOP88*, pages 344–358. Springer-Verlag, 1988. LNCS 300.

[Wad89] P. Wadler. Theorems for free! In *Proceedings Functional Programming Languages and Computer Architecture FPCA89, Imperial College, London*, pages 347–359. ACM Press, 1989.

[Wad90] P. Wadler. Comprehending monads. In *Proceedings 1990 ACM Conference on Lisp and Functional Programming*, pages 61–78, 1990.

[Wad95] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Lecture Notes on Advanced Functional Programming Techniques*, LNCS. Springer-Verlag, 1995.

[WM91] J. van der Woude and L. Meertens. A tribute to attributes. *The Squiggolist*, 2(1):20–26, 1991.

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