Bananas in Space: Extending Fold and Unfold to Exponential Types

Erik Meijer and Graham Hutton
University of Utrecht
The Netherlands

http://www.cs.ruu.nl/people/{erik,graham}/

Abstract

Fold and unfold are general purpose functionals for processing and constructing lists. By using the categorical approach of modelling recursive datatypes as fixed points of functors, these functionals and their algebraic properties were generalised from lists to polynomial (sum-of-product) datatypes. However, the restriction to polynomial datatypes is a serious limitation: it precludes the use of exponentials (function-spaces), whereas it is central to functional programming that functions are first-class values, and so exponentials should be able to be used freely in datatype definitions. In this paper we explain how Freyd's work on modelling recursive datatypes as fixed points of difunctors shows how to generalise fold and unfold from polynomial datatypes to those involving exponentials. Knowledge of category theory is not required; we use Gofer throughout as our meta-language, making extensive use of constructor classes.

1 Introduction

During the 1980s, Bird and Meertens [6, 22] developed a calculus (nicknamed *Squiggol*) of recursion functionals on lists, using which efficient functional programs can be derived from specifications by using equational reasoning. Squiggol was subsequently generalised from lists to polynomial (sum-of-product) datatypes [20] by using the categorical approach of modelling recursive datatypes as fixed points of functors [21, 14]. This approach allows `foldr`, `unfold` and other recursion functionals to be uniformly generalised from lists to polynomial datatypes. The generalised functionals are given special names (such as catamorphism and anamorphism), and are written symbolically using special brackets (such as "banana" brackets $(\!|\ |\!)$ and "lens" brackets $[\!(\ )\!]$.) The categorical approach also provides a number of algebraic laws that can be used to derive, transform and reason about programs expressed using these functionals. The theory and practice of such generic functionals has been explored by many authors, e.g. [3, 7, 10, 13, 14, 24, 33].

The aim of the *bananas* paper of Meijer, Fokkinga and Paterson [27] was to bring the ideas of Squiggol closer to lazy functional languages. This was achieved by moving from the category *set* of sets and total functions (the world of standard category theory and Squiggol) to the category *cpo* of cpos and continuous functions (the world of *cpo*-categories [12] and lazy functional programming). However, a serious deficiency of the bananas paper — and more generally, the work of the Squiggol community — is its limitation to polynomial datatypes [20]. This precludes the use of exponentials (function-spaces), whereas it is central to functional programming that functions are first-class values, and so exponentials should be able to be used freely in datatype definitions. So to truly bring Squiggol closer to functional programming, the theory must be extended to deal with datatypes that involve exponentials.

Technically, exponentials are problematic because the exponential functor is contravariant in its first argument. A standard solution to the problem is to move from the category *cpo* to the category cpo^{ep} of cpos and embedding-projection pairs, on which category the exponential functor can be made covariant [34]. But while the setting of cpo^{ep} is technically sufficient, from a practical point of view it is not a convenient category upon which to base a programming calculus for reasoning about datatypes and recursion functionals, because the arrows in cpo^{ep} do not naturally correspond to programs.

An alternative solution that allows us to stay within *cpo* has been proposed by Freyd [12]. His key idea is to model recursive datatypes as fixed points of *difunctors*, functors on two variables, contravariant on the first, covariant on the second. In the present article (but see also [29, 28]) we explain to functional programmers how Freyd's work shows how to generalise fold and unfold from polynomial datatypes to those involving exponentials.

We use Gofer throughout as our meta-language, making extensive use of the constructor classes extension to the standard Gofer (or Haskell) class system [17, 19]. Using Gofer rather than category theory as our meta-language makes the concepts more accessible as well as executable, and eliminates the gap between theory and practice.

2 Polynomial datatypes

We begin in this section by reviewing the theory introduced in the bananas paper, by implementing it in Gofer. In particular, we implement the generic versions `cata` and `ana` of the recursion functionals `foldr` and `unfold`.

2.1 Functors and recursive datatypes

A (*covariant*) *functor* is a type constructor `f` that assigns a type `f a` to each type `a`, together with a polymorphic functional `map` that lifts a function `g :: a -> b` to a function `map g :: f a -> f b`. In Gofer, the concept of a functor can be encapsulated as a constructor class, as follows:

```
class Functor f where
   map :: (a -> b) -> (f a -> f b)
```

Such a declaration is not possible using the standard class system, because the parameter `f` of the class `Functor` is a type constructor rather than a type.

A familiar example of a functor is the type constructor `[]` (not to be confused with the empty list `[]`) for lists:

```
instance Functor [] where
   map f xs = [f x | x <- xs]
```

Technically, a functor must also preserve the identity function `id` and distribute over function composition `(.)`, i.e. the following two equations must hold:

$$\begin{array}{lcl} \texttt{map}\ \ \texttt{id} & = & \texttt{id} \\ \texttt{map}\ \ (\texttt{g}.\texttt{h}) & = & (\texttt{map g}).(\texttt{map h}) \end{array}$$

However it is not possible to express these extra requirements directly in the Gofer class definition of a functor. It is the responsibility of the programmer to check that they indeed hold for each instance of the class.

Given a functor `f`, its induced recursive datatype `Rec f` is defined as the fixed point of `f`. In Gofer this can be implemented as follows:

```
data Rec f = In (f (Rec f)) {- #STRICT# -}
```

Since `Rec f` is recursive, we have been forced to define it using `data` rather than `type`, and as a consequence have been required to introduce the fictitious strict constructor `In`. Strictness of `In` is necessary to obtain an isomorphism between `Rec f` and `f (Rec f)`. If `In` was not strict, there would be no value in `f (Rec f)` that corresponds to the "undefined" value `bot` in `Rec f`, defined by `bot = bot`.

The strictness pragma in the definition of `Rec f` is not currently permitted in Gofer. However, a number of Haskell implementations permit such constraints in datatype definitions (e.g. [2]), as will future releases of Gofer [18].

Consider a simple datatype of arithmetic expressions, built out of numbers and binary addition:

```
data Expr = Num Int | Add Expr Expr
```

To express this datatype as the fixed point of a functor, we first define a functor `E` which captures the recursive structure of arithmetic expressions:

```
data E e = Num Int | Add e e

instance Functor E where
   map g = \x -> case x of
                   Num n    -> Num n
                   Add e e' -> Add (g e) (g e')
```

It is a simple exercise to verify that `map` satisfies the two equations required of a functor. The type `Expr` of expressions can now be defined as the fixed point of functor `E`:

```
type Expr = Rec E
```

Some illustrative values of type `Expr` are

```
In bot
In (Num 3)
In (Add bot bot)
In (Add (In (Num 1)) bot)
In (Add bot (In (Num 5)))
In (Add (In (Num 7)) (In (Num 2)))
...
let e = In (Add e e) in e
```

It is clear from these examples that `In` plays no essential rôle, except as an explicit type coercion between `E Expr` and `Expr`, and in general, between `f (Rec f)` and `Rec f`. It is also clear that the type `Expr` defined using `Rec` is isomorphic to the original Gofer definition using recursion. If `Rec f` could be defined as a recursive `type` synonym, the two types would in fact be identical.

Parameterised datatypes can also be defined as fixed points of functors. The general method is to partially parameterise a binary type constructor with a type variable to give a functor. For example, the datatype

```
data List a = Nil | Cons a (List a)
```

of lists with elements of type `a` can be defined as follows:

```
data L a l = Nil | Cons a l

instance Functor (L a) where
   map g = \x -> case x of
                  Nil      -> Nil
                  Cons a l -> Cons a (g l)

type List a = Rec (L a)
```

(In the remainder of this paper, only the `Rec` definition of most recursive datatypes used will be given. Such definitions can be translated to normal Gofer recursive definitions simply by unfolding the definition of `Rec`.)

Mutually recursive datatypes can be reduced to direct recursive datatypes in a similar way to that in which mutually recursive functions can be reduced to direct recursive functions [5, 10]. So no generality is lost by restricting our attention to direct recursive datatypes.

Note that only the type constructor part of a functor is necessary to express datatypes as fixed points of functors. As we shall see in the next section, the `map` part comes into play when recursion functionals on datatypes are defined.

2.2 Invariants, algebras and catamorphisms

In Freyd's terminology [12], an isomorphism between types `f a` and `a` is an `f`-*invariant*. An example of an `f`-invariant is `In :: f (Rec f) -> Rec f`. Among all possible `f`-invariants, `In` is special in the sense that it is the *minimal* `f`-invariant. Minimality expresses that the function

```
copy :: Functor f => (Rec f -> Rec f)
copy (In x) = In (map copy x)
```

which recursively replaces the constructor `In` by itself is the identity function on the datatype `Rec f`. That `copy` = `id` holds is easily proved by structural induction.

Suppose now that we generalise `copy` to replace `In` not by itself but by an arbitrary function `phi :: f a -> a`. In this way we obtain the notion of a *catamorphism* [27]:

```
cata :: Functor f => (f a -> a) -> (Rec f -> a)
cata phi (In x) = phi (map (cata phi) x)
```

The functional `cata`—written as "banana" brackets ⦇ ⦈ in the Squiggol literature—is the generic version of the familiar recursion functional `foldr` on lists, generic in the sense that it can be used with any polynomial datatype. The term catamorphism comes from the Greek preposition $\kappa\alpha\tau\alpha$, meaning downwards, and reflects the fact that `cata phi` recursively walks down its argument replacing each occurrence of `In` by a function `phi` along the way.

Given a functor `f` and a specific type `a`, a function `phi :: f a -> a` is known as an `f`-*algebra*. Consider again the functor `E` for arithmetic expressions. The following function (which replaces the constructors `Num` and `Add` by the functions `id` and `(+)`) is an `E`-algebra of type `E Int -> Int`:

```
\x -> case x of
         Num n    -> id n
         Add e e' -> e + e'
```

Applying `cata` to the above algebra gives the standard evaluator for arithmetic expressions:

```
eval :: Expr -> Int
eval = cata (\x -> case x of
                      Num n    -> id n
                      Add e e' -> e + e')
```

This definition says that expressions can be evaluated by simultaneously replacing all `Num` constructors by the `id` function on integers, and all `Add` constructors by `(+)` on integers. Unfolding the definition to eliminate the use of `cata` and `map` makes clear that it has the expected behaviour:

```
eval (In x) =
   case x of
      Num n    -> n
      Add e e' -> (eval e) + (eval e')
```

2.3 Free theorems and fusion

A useful heuristic in functional programming is to inspect the "free theorem" [36] that comes from the type of a polymorphic function. The free theorem for `cata :: Functor f => (f a -> a) -> (Rec f -> a)` is the well-known *fusion* law [27]: for strict functions `h`,

$$\texttt{h.phi} = \texttt{phi'.(map h)} \quad \Rightarrow \quad \texttt{h.(cata phi)} = \texttt{cata phi'}$$

If we only consider finite elements of `Rec f` the strictness condition on `h` can be removed. Fusion can also be proved directly using a simple fixed point induction [27], for which it is also necessary that `h` be strict.

The hidden type information in the fusion law is exposed when using commuting diagrams instead of equations:

$$\begin{tikzcd} \texttt{f a} \arrow[r, "\texttt{map h}"] \arrow[d, "\texttt{phi}"'] & \texttt{f b} \arrow[d, "\texttt{phi'}"] \\ \texttt{a} \arrow[r, "\texttt{h}"'] & \texttt{b} \end{tikzcd} \quad \Rightarrow \quad \begin{tikzcd} \texttt{Rec a} \arrow[d, "\texttt{cata phi}"'] \arrow[dr, "\texttt{cata phi'}"] & \\ \texttt{a} \arrow[r, "\texttt{h}"'] & \texttt{b} \end{tikzcd}$$

Fusion captures a common pattern of inductive proof on programs expressed as catamorphisms, in a similar way to that in which `cata` itself captures a common pattern of recursion over polynomial datatypes. Minimality and fusion can together be used to show that `cata phi` satisfies a universal property, namely that `cata phi` is the *unique* function satisfying its defining equation.

Returning to our running example, an alternative way to evaluate arithmetic expressions is to use a stack of type `[Int]` to store intermediate values. Such a stack-based evaluator can be defined as follows:

```
eval' :: Expr -> ([Int] -> [Int])
eval' = cata (\x -> case x of
                       Num n    -> push n
                       Add e e' -> add.e'.e)
```

where `push a as = a:as` pushes a number onto the stack and `add (a:b:cs) = (b+a):cs` adds the top two values.

The fact that the stack-based evaluator leaves the expected value on top of the stack, i.e. for all finite expressions `e :: Expr` we have `push (eval e)` = `eval' e`, can easily be proved using fusion and the distribution of `push` over addition: `push (a+b)` = `add.(push a).(push b)` [25].

2.4 Coalgebras and anamorphisms

Using `cata` we can define functions with recursive datatypes as their source. Dually, it is also useful to have a functional for defining functions with recursive datatypes as their target. Let us begin by re-writing the function `copy` from which catamorphisms arose in the equivalent form

```
copy :: Functor f => (Rec f -> Rec f)
copy x = In (map copy (out x))
```

where `out (In x) = x` is the inverse of the isomorphism `In`. If we now generalise this version of `copy` by replacing the occurrence of `out :: Rec f -> f (Rec f)` in its definition by an arbitrary function `psi : a -> f a` (an `f`-*coalgebra*), we obtain the notion of an *anamorphism* [27]:

```
ana :: Functor f => (a -> f a) -> (a -> Rec f)
ana psi x = In (map (ana psi) (psi x))
```

The functional `ana`—written as "lens" brackets $[\![\ ]\!]$ in the Squiggol literature—is the generic version of the recursion functional `unfold` [8, p173] on lists. The Greek preposition $\alpha\nu\alpha$ means upwards, and its use here reflects the fact that `ana psi` recursively builds up its result by decomposing its argument using the function `psi`.

We illustrate the notion of an anamorphism by defining a function `n2b` that converts natural numbers to their binary representation. The first step is to define a type `Bin` of binary numbers as the fixed point of a functor `B`:

```
data B b = Empty | Zero b | One b

instance Functor B where
   map g = \x -> case x of
                    Empty  -> Empty
                    Zero b -> Zero (g b)
                    One b  -> One (g b)

type Bin = Rec B
```

The binary representation of a natural number is built by recursively splitting off its least significant bit:

```
n2b :: Int -> Bin
n2b = ana (\x -> case x of
                    0     -> Empty
                    2*n   -> Zero n
                    2*n+1 -> One n)
```

For example, `n2b 2` = `In (Zero (In (One (In Empty))))`. The dual function `b2n` that converts a binary number back to a natural number can be defined as a catamorphism:

```
b2n :: Bin -> Int
b2n = cata (\x -> case x of
                     Empty  -> 0
                     Zero b -> 2*b
                     One b  -> 1+2*b)
```

2.5 Free theorems and fusion

The free theorem for the functional `ana :: Functor f => (a -> f a) -> (a -> Rec f)` is also a fusion theorem:

$$\texttt{psi.h} = \texttt{(map h).psi'} \quad \Rightarrow \quad \texttt{(ana psi).h} = \texttt{ana psi'}$$

or in diagrammatic form,

$$\begin{tikzcd} \texttt{b} \arrow[d, "\texttt{psi}"'] & \texttt{a} \arrow[l, "\texttt{h}"'] \arrow[d, "\texttt{psi'}"] \\ \texttt{f b} & \texttt{f a} \arrow[l, "\texttt{map h}"] \end{tikzcd} \quad \Rightarrow \quad \begin{tikzcd} \texttt{b} \arrow[d, "\texttt{ana psi}"'] & \texttt{a} \arrow[l, "\texttt{h}"'] \arrow[dl, "\texttt{ana psi'}"] \\ \texttt{Rec f} & \end{tikzcd}$$

In this case there is no strictness requirement on `h`, since dualising `h.bot` = `bot` gives `bot.h` = `bot`, which is true for all functions `h`. Using fusion for anamorphisms, together with the fact that `ana out` is the identity function on `Rec f`, we can show that `ana psi` is in fact the unique function satisfying its defining equation.

2.6 Primitive and general recursion

Meertens has shown that every primitive recursive function, i.e. *paramorphism* [23], can be expressed as an `ana` followed by a `cata`. Let us briefly show how paramorphisms can be implemented in Gofer. The first step is to define a family of functors `P f`, one for each functor `f`:

```
type P f a = f (Rec f,a) in mapP, para, pp

mapP :: Functor f =>
   (a -> b) -> (P f a -> P f b)
mapP g = map (\(x,a) -> (x, g a))

instance Functor f => Functor (P f) where
   map = mapP
```

Note that `P` above is defined as a restricted type synonym [16] so that it can be partially applied. As a consequence, the functional `mapP` cannot be defined directly within the instance declaration for `P f`. A functional `para` that builds paramorphisms is defined now by:

```
para :: Functor (P f) =>
   (f (Rec f, a) -> a) -> (Rec f -> a)
para phi = cata phi . preds

preds :: Functor (P f) => Rec f -> Rec (P f)
preds = ana pp

pp :: Functor f => Rec f -> P f (Rec f)
pp (In x) = map (\a -> (a,a)) x
```

Again for technical reasons concerning types, the definition for `para` above has to be split up into parts.

It came as somewhat of a surprise to the authors to discover that a general fixed point operator can also be defined as the composition of an `ana` followed by a `cata`, thus providing the full power of recursion. (We have since discovered that this observation has already been made by Freyd [11].) The effect is that algebraic languages that provide `cata` and `ana` as the *only* means to define recursive functions are not limited in expressive power.

Using `cata` and `ana`, the least fixed point `fix f` of a function `f` can be computed as the infinite application `f (f (f ...))` in the following way: first use an anamorphism to build an infinite list `In (Cons f (In (Cons f (In (Cons f ...)))))`, and then use a catamorphism to replace each constructor `Cons` by function application.

```
fix :: Functor (L (a -> a)) => (a -> a) -> a
fix = cata (\(Cons f x) -> f x)
         . ana (\f -> Cons f f)
```

In general, many functions can be naturally expressed as the composition of an `ana` and a `cata`, so it seems useful to name this idiom. Functions expressed in this way are known as *hylomorphisms* [27]:

```
hylo :: Functor f =>
   (f a -> a) -> (b -> f b) -> (b -> a)
hylo phi psi = cata phi . ana psi
```

A straightforward fixed point induction shows that the two constituents of a hylomorphism can be fused together to give a direct recursive definition that avoids building an intermediate [37] (or virtual [35]) value:

```
hylo phi psi = phi . map (hylo phi psi) . psi
```

For example, if we express `fix` as a hylomorphism rather than the composition of a `cata` and an `ana`,

```
fix f =
   hylo (\(Cons f x) -> f x) (\f -> Cons f f)
```

then by unfolding using the more efficient definition of `hylo` we find that `fix f` = `f (fix f)`, as expected.

3 Problems with exponentials

In the previous section we reviewed how the functionals `foldr` and `unfold` are generalised from lists to polynomial datatypes. While such datatypes are sufficient for many programming tasks, a central aspect of functional programming is that functions are first-class values.

However, exponentials (function-spaces) are problematic because the type constructor `(->)` is contravariant in its first argument. The effect is that certain type constructors defined using `(->)` cannot be made into functors, and as a result, functionals such as `cata` and `ana` cannot always be used to define functions on recursive datatypes involving exponentials. This section gives a number of examples of recursive datatypes involving exponentials, and elaborates on the problems with such datatypes.

3.1 Covariant uses of `(->)`

An example in which function-spaces are used covariantly is that of non-deterministic computations [32]. An element of datatype `State a b` is either a final value of type `b`, or an intermediate state of type `a` together with a non-deterministic continuation of type `a -> [State a b]`:

```
data S a b s = Done b | Pause a (a -> [s])

type State a b = Rec (S a b)
```

To make the type constructor `S a b` into a functor, we first observe that the sub-component `(a ->)` can itself be made into a functor. That is, fixing the first argument of `(->)` to a specific type `a` yields a functor `((->) a)`. The required type of the `map` functional for `((->) a)` is `(b -> c) -> ((->) a b) -> ((->) a c)`. Using familiar infix notation we recognise `(b -> c) -> (a -> b) -> (a -> c)` as the type of function composition `(.)`. One can easily verify that `(.)` indeed makes `((->) a)` into a functor.

```
instance Functor ((->) a) where
   map = (.)
```

Now `S a b` can be made into a functor, as follows:

```
instance (Functor ((->) a), Functor []) =>
   Functor (S a b) where
      map g =
         \x -> case x of
                  Done n    -> Done n
                  Pause n h ->
                     Pause n (map (map g) h)
```

Note that `map` for `S a b` is not recursive; the uses of `map` in its definition are those for `((->) a)` and lists `[]`.

A function `exec` that forces evaluation of a state to its set of final values can now be defined as a catamorphism:

```
exec :: Functor (S a b) => State a b -> [b]
exec = cata (\x -> case x of
                     Done n    -> [n]
                     Pause n h -> concat (h n))
```

As a simple application, `exec` and `ana` can be used to define a function `n2d` that extracts the list of digits from a number:

```
n2d :: Int -> [Int]
n2d = exec . ana (\x ->
         if x <= 9 then Done x
         else Pause x (\n -> [n `div` 10,
                                n `mod` 10]))
```

For example, `n2d 1234` $= [1, 2, 3, 4]$.

A more practical use of the type `State` is the extension of a library of parsing combinators [15] with a combinator for parallel composition of parsers [9].

3.2 Contravariant uses of `(->)`

An example where `(->)` is used contravariantly is in the definition of a fixed point combinator by using recursion on types rather than recursion on functions. We first define a type `Inf a` of functions that yield a result of type `a` from an infinitely nested argument of such functions:

```
type I a i = (i -> a) in inI, outI

inI  :: (i -> a) -> I a i
outI :: I a i -> (i -> a)

inI  = id
outI = id

type Inf a = Rec (I a)
```

The functions `inI` and `outI` above play the rôle of constructor and destructor functions for the type `I a i`.

Using `Inf a` we can define Curry's fixed point combinator $Yf = gg \ where \ g = \lambda h.f(hh)$ from the untyped λ-calculus in a typed functional language:

```
fix :: (a -> a) -> a
fix f = g (In (inI g))
        where g (In h) = f (outI h (In h))
```

We would like to be able to express recursive functions on `Inf a` using recursion functionals such as `cata`, but it is not possible to define a `map` that makes `I a` into a covariant functor. However, the following definition of a functional `comap` makes `I a` into a contravariant functor:

```
comap :: (b -> c) -> (I a c -> I a b)
comap g h = inI.(h.g).outI
```

A *contravariant* functor is like a covariant functor, except that the functional `comap` lifts a function `g :: a -> b` to a function `comap g :: f b -> f a` where the argument and result types have been interchanged. As a consequence, such functors must distribute contravariantly over function composition: `comap` (g.h) = (`comap h`).(`comap g`).

In Gofer, the concept of a contravariant functor can be encapsulated as a constructor class, as follows:

```
class Cofunctor f where
   comap :: (a -> b) -> (f b -> f a)
```

It is possible, with some effort, to define versions of `cata`, `ana`, `para`, and `hylo` on datatypes expressed as fixed points of contravariant functors, but this would only be a partial solution to the problem. In general, a type constructor involving function-spaces can be of mixed variance.

3.3 Mixed variant uses of `(->)`

An example where `(->)` is used both covariantly and contravariantly is in the definition of a type `Scott` for modelling the untyped (lazy) λ-calculus [1]:

```
data S s = Func (s -> s)

type Scott = Rec S
```

An occurrence of a type variable in a type expression is said to be contravariant if it occurs to the left of an odd number of nested arrows `(->)`, and covariant otherwise. The argument `s` to `S` above occurs both covariantly `(s -> s)` (second `s` underlined) and contravariantly `(s -> s)` (first `s` underlined). The effect is that `S` cannot be made into a functor, either covariant or contravariant.

We can however make the distinction between the two kinds of occurrences of the argument `s` in the definition of `S` explicit by defining a binary type constructor `S'`:

```
data S' s s' = Func (s -> s')
```

By fixing its first argument, `S'` can be made into a covariant functor; by fixing its second argument, `S'` can be made into a contravariant functor. In general, a binary type constructor with this property is called a difunctor.

Formally, a *difunctor* [12] is a binary type constructor `f` that assigns to each pair of types `a` and `b` a type `f a b`, together with a polymorphic functional `dimap` that lifts a pair of functions `g :: a -> b` and `h :: c -> d` to a function ``g `dimap` h :: f b c -> f a d``. A difunctor must also preserve the identity function and distribute over function composition in the following way:

```
  id   `dimap`   id   =  id
(g.h)  `dimap`  (i.j) =  (h `dimap` i).(g `dimap` j)
```

In Gofer the concept of a difunctor can be encapsulated as a constructor class, as follows:

```
class Difunctor f where
   dimap :: (a -> b) -> (c -> d) ->
               (f b c -> f a d)
```

One can verify now that the following definition for `dimap` makes the type constructor `S'` into a difunctor:

```
instance Difunctor S' where
   (f `dimap` g) (Func h) = Func (g.h.f)
```

In the above, the `Func` constructor only plays an auxiliary rôle. In fact, `S'` is a difunctor because the function-space constructor `(->)` is itself a difunctor:

```
instance Difunctor (->) where
   (f `dimap` g) h = g.h.f
```

In general, by separating the covariant and contravariant occurrences of the argument `a` in the body of a non-recursive datatype declaration `data F a = ...`, every such type constructor `F` induces a difunctor `F'`, such that `F` can be recovered from `F'` by diagonalising, i.e. `F a` = `F' a a`.

4 General datatypes

We have seen in the previous section that (non-recursive) type constructors involving exponentials do not in general induce functors, but do induce difunctors. Freyd [12] presents a categorical theory of recursive datatypes modelled as fixed points of difunctors. In this section we explain how Freyd's work shows how to generalise the recursion functionals `cata` and `ana`, together with their associated fusion rules. As was the case previously, `cata` and `ana` are obtained by suitably generalising a simple `copy` function.

4.1 Difunctors and recursive datatypes

Given a difunctor `f`, its induced recursive datatype `Rec f` is defined as the simultaneous fixed point of `f` in both arguments. In Gofer this definition for `Rec f` can be implemented as follows (as previously, strictness of the constructor `In` is necessary to obtain an isomorphism):

```
data Rec f =
   In (f (Rec f) (Rec f)) {- #STRICT# -}
```

4.2 Catamorphisms and anamorphisms

An isomorphism between types `f a a` and `a` is called an `f`-invariant. An example of an `f`-invariant is `In :: f (Rec f) (Rec f) -> Rec f`. It is also the minimal `f`-invariant, in the sense that `copy` = `id`, where

```
copy :: Difunctor f => (Rec f -> Rec f)
copy (In x) = In ((copy `dimap` copy) x)
```

This definition can be expressed in diagrams by

$$\begin{tikzcd}
\texttt{(Rec f)} & \texttt{`f`} \arrow[d, phantom, "\texttt{`dimap`}"] & \texttt{(Rec f)} \arrow[r, "\texttt{In}"] \arrow[d, "\texttt{copy}"] & \texttt{Rec f} \arrow[d, "\texttt{copy}"] \\
\texttt{(Rec f)} \arrow[u, "\texttt{copy}"] & \texttt{`f`} & \texttt{(Rec f)} \arrow[r, "\texttt{In}"'] & \texttt{Rec f}
\end{tikzcd}$$

Note that by drawing the arrows `g :: a -> b` and `h :: c -> d` of a difunctor ``g `dimap` h :: (b `f` c) -> (a `f` d)`` separately, both the contravariance and typing assumptions of `dimap` are made explicit.

For datatypes expressed as fixed points of functors, the notion of a catamorphism arose by abstracting on `In` in the body of the definition of `copy`. Let us now try to play the same game for the difunctors version of `copy`. As a first attempt, abstracting (naively) on `In` in the body of the difunctors version of `copy` gives the definition

```
cata phi (In x) =
   phi (((cata phi) `dimap` (cata phi)) x)
```

However this definition is too restrictive, since it forces the argument function `phi` to have type `f (Rec f) (Rec f) -> Rec f`, and `cata phi` itself to have type `Rec f -> Rec f`.

The problem is the use of `cata phi` as *both* the covariant and contravariant argument of `dimap` in the definition. The covariant use of `cata phi` requires that the argument function `phi` have type `f b a -> a`; a function of this type is called an `f`-*dialgebra* [12]. The additional contravariant use of `cata phi` then requires that `a` = `b` = `Rec f`, i.e. that `phi` have type `f (Rec f) (Rec f) -> Rec f`.

As a first step to solving this problem, let us assume the existence of a function `g :: b -> Rec f` to use as the contravariant argument of `dimap` in the body of `cata phi`, rather than `cata phi` itself. This assumption leads to a definition for `cata phi` with sufficiently general typing requirements, as illustrated by the following diagram:

$$\begin{tikzcd}
\texttt{(Rec f)} & \texttt{`f`} \arrow[d, phantom, "\texttt{`dimap`}"] & \texttt{(Rec f)} \arrow[r, "\texttt{In}"] \arrow[d, "\texttt{cata phi}"] & \texttt{Rec f} \arrow[d, "\texttt{cata phi}"] \\
\texttt{b} \arrow[u, "\texttt{g}"] & \texttt{`f`} & \texttt{a} \arrow[r, "\texttt{phi}"'] & \texttt{a}
\end{tikzcd}$$

A similar problem occurs with the naive generalisation of `copy` to obtain an anamorphism functional:

```
ana psi x =
   In (((ana psi) `dimap` (ana psi)) (psi x))
```

The covariant use of `ana psi` here requires that `psi` have type `b -> f a b`; a function of this type is called an `f`-*codialgebra*. The additional contravariant use of `ana psi` then requires that `a` = `b` = `Rec f`, i.e. that `psi` have type `Rec f -> f (Rec f) (Rec f)`, and hence that `ana psi` have type `Rec f -> Rec f`. However, a definition for `ana psi` with sufficiently general typing requirements can be obtained by assuming the existence of a function `h :: Rec f -> a` to use as the contravariant argument of `dimap`:

$$\begin{tikzcd}
\texttt{a} & \texttt{`f`} \arrow[d, phantom, "\texttt{`dimap`}"] & \texttt{b} \arrow[d, "\texttt{ana psi}"] & \texttt{b} \arrow[l, "\texttt{psi}"'] \arrow[d, "\texttt{ana psi}"] \\
\texttt{(Rec f)} \arrow[u, "\texttt{h}"] & \texttt{`f`} & \texttt{(Rec f)} & \texttt{(Rec f)} \arrow[l, "\texttt{out}"]
\end{tikzcd}$$

Let us now consider the above diagrams for `cata phi` and `ana psi` simultaneously. We observe that a function `g :: b -> Rec f` required to define `cata phi` can be obtained simply as `g` = `ana psi`, and similarly, a function `h :: Rec f -> a` required to define `ana psi` can be obtained as `h` = `cata phi`. Thus we are naturally led to the following mutually recursive definitions for `cata` and `ana` on datatypes expressed as fixed points of difunctors:

```
cata :: Difunctor f =>
   (f b a -> a) -> (b -> f a b) -> (Rec f -> a)
ana :: Difunctor f =>
   (f b a -> a) -> (b -> f a b) -> (b -> Rec f)

cata phi psi (In x) =
   phi (((ana phi psi)
      `dimap` (cata phi psi)) x)
ana phi psi x =
   In (((cata phi psi)
      `dimap` (ana phi psi)) (psi x))
```

Note that the difunctor versions of `cata` and `ana` above are proper generalisations of the functor versions from Section 2, in the sense that if the difunctor `f` is independent of its contravariant argument, the definitions reduce to the standard definitions for functors.

4.3 Free theorems and fusion

Just as was the case for functors, the `cata` and `ana` functionals for difunctors satisfy a fusion law, which arises as a free theorem. Because the difunctors versions of `cata` and `ana` are defined mutually recursively, we get a simultaneous fusion law for the two functionals, rather than two separate laws as was the case previously: for strict functions `h`,

```
        h.phi = phi'.(g `dimap` h)
     ∧
        psi.g = (h `dimap` g).psi'
  ⇒
        h.(cata phi psi) = cata phi' psi'
     ∧
        (ana phi psi).g = ana phi' psi'
```

or in diagrammatic form,

$$
\begin{tikzcd}
\mathtt{a} \arrow[r, phantom, "\text{`f`}"] \arrow[dr, phantom, "\text{`dimap`}"] & \mathtt{b} \arrow[r, "\mathtt{phi}"] \arrow[d, "\mathtt{h}"] & \mathtt{b} \arrow[d, "\mathtt{h}"] \\
\mathtt{c} \arrow[u, "\mathtt{g}"] \arrow[r, phantom, "\text{`f`}"] & \mathtt{d} \arrow[r, "\mathtt{phi'}"'] & \mathtt{d}
\end{tikzcd}
$$

$$\wedge$$

$$
\begin{tikzcd}
\mathtt{d} \arrow[r, phantom, "\text{`f`}"] \arrow[dr, phantom, "\text{`dimap`}"] & \mathtt{c} \arrow[d, "\mathtt{g}"] & \mathtt{c} \arrow[l, "\mathtt{psi'}"'] \arrow[d, "\mathtt{g}"] \\
\mathtt{b} \arrow[u, "\mathtt{h}"] \arrow[r, phantom, "\text{`f`}"] & \mathtt{a} & \mathtt{a} \arrow[l, "\mathtt{psi}"]
\end{tikzcd}
$$

$$\Rightarrow$$

$$
\begin{tikzcd}
\mathtt{Rec\ f} \arrow[r, "\mathtt{cata\ phi\ psi}"] \arrow[dr, "\mathtt{cata\ phi'\ psi'}"'] & \mathtt{b} \arrow[d, "\mathtt{h}"] \\
& \mathtt{d}
\end{tikzcd}
$$

$$\wedge$$

$$
\begin{tikzcd}
\mathtt{Rec\ f} & \mathtt{c} \arrow[l, "\mathtt{ana\ phi'\ psi'}"'] \arrow[d, "\mathtt{g}"] \\
& \mathtt{a} \arrow[ul, "\mathtt{ana\ phi\ psi}"]
\end{tikzcd}
$$

It is interesting to note that the above fusion law turns out to be the specialisation to functions of Pitt's relational induction principle for recursive datatypes [29, Prop 2.10].

Let us consider an example of the use of fusion. A *retract* from a type `b` to a type `a` is a pair of functions `up :: b -> a` and `down :: a -> b` such that `up.down` = `id :: a -> a`. In other words, `down` is an injective function with `up` as a left-inverse. In Gofer, the notion of a retract can be encapsulated as a type class, as follows:

```
class Retract b a where
   up   :: b -> a
   down :: a -> b
```

Using fusion it can be shown that given a difunctor `f`, if `up` and `down` form a retract from `(f a a)` to `a`, then `(ana up down)` and `(cata up down)` form a retract from `a` to `(Rec f)`. In Gofer, this can be implemented as follows:

```
instance (Difunctor f, Retract (f a a) a) =>
   Retract a (Rec f) where
      up = ana (up :: Retract (f a a) a =>
              f a a -> a) (down :: Retract
                 (f a a) a => a -> f a a)
      down = cata (up :: Retract (f a a) a =>
                f a a -> a) (down :: Retract
                   (f a a) a => a -> f a a)
```

This result will be used in the next section.

4.4 Interpreters for the λ-calculus

We illustrate the generalised theory by defining a class of interpreters for the untyped λ-calculus, and taking some steps towards formally relating such interpreters. We begin by defining a datatype `Expr` of λ-expressions:

```
data Expr = Var String
          | Lambda String Expr
          | Apply Expr Expr
```

The datatype `Expr` could of course be expressed as the fixed point of a functor, but we don't do this here, preferring instead to concentrate on the use of fixed points in defining the semantic domains for λ-interpreters.

A datatype can serve as such a semantic domain if it is reflexive [4]. Formally, a type `a` is *reflexive* if there is a retract from `a` to `(a -> a)`. (The notion of a retract was defined in the previous section.) Intuitively then, a type is reflexive if it is large enough to faithfully represent its own function-space. In Gofer, the notion of a reflexive type can be encapsulated as a type class, as follows:

```
class Retract a (a -> a) => Reflexive a where
   apply :: a -> (a -> a)
   abstr :: (a -> a) -> a

   f `apply` a = up f a
   abstr f     = down f
```

We can now define a class of interpreters (one for each reflexive type `a`) that map a λ-expression to its value in the semantic domain `a`; as usual, an environment carries the values of the free variables in the expression:

```
class Reflexive a => LambdaModel a where
   eval :: Expr -> Env a -> a

   eval (Var x) env =
      env `lookup` x
   eval (Lambda x b) env =
      abstr (\a -> eval b (env `update` (x,a)))
   eval (Apply f a) env =
      (eval f env) `apply` (eval a env)
```

Environments are represented as functions from identifiers to values, and are equipped with two operations:

```
type Env a = String -> a in
                lookup, update, mapEnv

lookup :: Env a -> String -> a
update :: Env a -> (String,a) -> Env a

env `lookup` x = env x

env `update` (x,a) =
   \y -> if y==x then a else env `lookup` y
```

Later on in this section, we will use the fact that `Env` can be extended to a functor, as follows:

```
mapEnv :: (a -> b) -> (Env a -> Env b)
mapEnv = (.)

instance Functor Env where
   map = mapEnv
```

The standard (call by name) interpreter for the untyped λ-calculus is obtained by taking the reflexive datatype `Scott` of section 3.3 as the semantic domain:

```
instance Retract Scott (Scott -> Scott) where
   up   (In (Func f)) = f
   down f             = In (Func f)

instance Reflexive Scott

instance LambdaModel Scott
```

An appropriate reflexive type `Cont` for a (call by name) continuation-based semantics for λ-expressions is defined by:

```
type Cont = (Closure -> Closure) -> Closure

instance Retract Cont (Cont -> Cont) where
   up f =
      \a -> \cont ->
         f (\(In (Clos f)) -> f a cont)
   down f =
      \cont -> cont (In (Clos f))

instance Reflexive Cont

instance LambdaModel Cont
```

where `Closure` is the fixed point of difunctor `C`:

```
data C c c' =
   Clos (((c -> c') -> c)
      -> ((c' -> c) -> c'))

instance Difunctor (->) => Difunctor C where
   (f `dimap` g) (Clos h) =
      Clos ((((f `dimap` g) `dimap` f)
         `dimap` ((g `dimap` f) `dimap` g)) h)

type Closure = Rec C
```

Suppose now that we want to show that the continuation-based interpreter is correct with respect to the standard interpreter [30]. An interpreter based upon a reflexive type `b` is correct with respect to an interpreter based upon a reflexive type `a` if there exists a retract from `b` to `a` such that the following diagram commutes for all expressions `e`:

$$\begin{tikzcd} \texttt{Env a} \arrow[r, "\texttt{eval e}"] & \texttt{a} \\ \texttt{Env b} \arrow[u, "\texttt{map up}"] \arrow[r, "\texttt{eval e}"'] & \texttt{b} \arrow[u, "\texttt{up}"'] \end{tikzcd}$$

That is, evaluating an expression using an environment binding variables to `b`-values and then converting the result to an `a`-value is equivalent to first converting the bindings to `a`-values and then interpreting the expression.

It turns out now that the free theorem for the polymorphic λ-interpreter `eval :: Reflexive a => Expr -> Env a -> a` gives tractable conditions that establish the correctness of one interpreter with respect to another:

$$\begin{array}{l} \qquad \texttt{h (f `apply` a)} = \texttt{(h f) `apply` (h a)} \\ \quad \wedge \\ \qquad \texttt{h.g} = \texttt{f.h} \;\Rightarrow\; \texttt{h (abstr g)} = \texttt{abstr f} \\ \Rightarrow \\ \quad \texttt{h.(eval e)} = \texttt{(eval e).map h} \end{array}$$

(See [26] for a general discussion of free theorems in the context of class restrictions.) This result is a generalisation to the untyped λ-calculus of Reynolds' (functional) *abstraction theorem* for the typed λ-calculus [31].

Appealing to the abstraction theorem with `a` = `Scott`, `b` = `Cont` and `h` = `up`, we are required to find a retract from `Cont` to `Scott` satisfying the preconditions of the theorem. Using the fact that `Scott` is of the form `Rec S'`, together with the fact that `cata` and `ana` preserve retracts (previous section), we are motivated to look for a retract from `Cont` to `S' Cont Cont`. It is easy to prove that the following definitions for `up` and `down` give such a retract, thus establishing a retract from `Cont` to `Scott`:

```
instance Retract Cont (S' Cont Cont) where
   up f          = Func (up f)
   down (Func f) = down f

instance Retract Cont Scott
```

It remains to show that `up :: Cont -> Scott` satisfies the preconditions of the abstraction theorem. Verifying the second condition is straightforward. However, we have not yet been successful in establishing the first condition, namely that `up (f `apply` a)` = `(up f) `apply` (up a)`.

4.5 Covariant functors suffice

Freyd [12] shows that, somewhat surprisingly, the generalisation from functors to difunctors is not technically necessary to handle exponentials: fixed points of difunctors can be expressed in terms of fixed points of covariant functors. The result is mainly of theoretical interest, but it is instructive to see how the translation from difunctors to functors works. The present class system of Gofer isn't quite powerful enough to let us implement all aspects of the translation directly, so here we just give an outline.

As we have seen previously for the case of `(->)`, a difunctor `f` can be made into a covariant functor `(f a)` by fixing its contravariant argument to a specific type `a`:

```
instance Difunctor f => Functor (f a) where
   map g = id `dimap` g
```

Consider now the mapping on types `MyRec f` which sends a type `a` to the fixed point of the covariant functor `(f a)`:

```
type MyRec f a = Rec (f a)
```

By using the `cata` operator for functors of section 2.2, the mapping `MyRec f` can be extended to a mapping on functions, such that `MyRec f` is then a contravariant functor:

```
mycomap :: (Difunctor f, Functor (f b)) =>
    (a -> b) -> (MyRec f b -> MyRec f a)
mycomap g = cata (In . (g `dimap` id))
```

For technical reasons concerning type classes in Gofer, `MyRec f` cannot directly be made into an instance of Gofer class `Cofunctor` of contravariant functors.

A contravariant functor `f` can be made into a covariant functor `Square f` by composing `f` with itself:

```
type Square f a = f (f a) in sqrmap

sqrmap :: Cofunctor f =>
   (a -> b) -> (Square f a -> Square f b)
sqrmap g = comap (comap g)

instance Cofunctor f => Functor (Square f)
   where map = sqrmap
```

Again for technical reasons, `sqrmap` cannot be defined directly within the instance declaration above.

Freyd shows now that fixed points of difunctors can be reduced to fixed points of covariant functors in two steps. First of all, the least fixed point `Rec f` of a difunctor `f` is isomorphic to the least fixed point `Rec (MyRec f)` of the contravariant functor `MyRec f` (viewed as a difunctor independent of its second argument). And secondly, the least fixed point `Rec f` of a contravariant functor is isomorphic to the least fixed point `Rec (Square f)` of the covariant functor `Square f`. Combining the two steps, we see that the least fixed point `Rec f` of a difunctor `f` can be obtained (up to isomorphism) as the least fixed point `Rec (Square (MyRec f))` of the covariant functor `Square (MyRec f)`.

Another way of showing that covariant functors suffice is to first eliminate the use of mutual recursion in the definitions of `cata` and `ana` (using standard techniques), and then construct datatypes on which the resulting functions are catamorphisms and anamorphisms.

5 Discussion

In this paper we have explained how the recursion functionals `cata` and `ana` can be generalised from polynomial datatypes to those involving exponentials. An important area for future research is in experimenting with the use of the generalised operators and laws in writing and transforming programs. Another interesting topic for study is non-regular datatypes [28], i.e. datatypes in which the recursive calls in the body are not all of the form of the head of a definition. Examples are the datatype `Twist a b` of lists of alternating elements of type `a` and type `b`, and the datatype `Nest a` of lists of nested lists:

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

data Nest a = Block a (Nest [a])
```

To our knowledge, it is not in general known how to express non-regular datatypes as fixed points of functors (or difunctors). Note however that it is possible, with some effort, to express non-regular datatypes as fixed points of type constructors, by using a `Rec` of kind `((* -> *) -> (* -> *)) -> (* -> *)` instead of kind `(* -> *) -> *`.

Our final remarks concern the Gofer type system. In a number of places we had to hack around the limitations of type synonyms. First of all, since standard type synonyms cannot be partially applied we were forced in some cases to make use of restricted type synonyms, which can be partially applied. Secondly, since type synonyms cannot be recursive, we were forced to use a data declaration in defining `Rec`, leading to the introduction of the fictitious constructor `In`. Both these problems don't seem to be inherent to type synonyms, but are rather artifacts of the treatment of type synonyms as macros in earlier functional languages; see [19] for further discussion on this point.

Acknowledgements

Utrecht University and Chalmers University provided funding for mutual visits by the two authors, during which time part of this paper was written. It was Ross Paterson who originally suggested to Meijer that Freyd's paper might have applications in functional programming. Thanks to Luc Duponcheel, Johan Jeuring, Mark Jones, and the FPCA referees for useful comments and suggestions.

References

[1] Samson Abramsky. The lazy lambda calculus. In David Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[2] Lennart Augustsson. The Haskell B. compiler. Chalmers University of Technology, 1994.

[3] Roland Backhouse, Ed Voermans, and Jaap van der Woude. A relational theory of datatypes. In preparation, 1994.

[4] Henk Barendregt. *The Lambda Calculus – it's Syntax and Semantics*. North-Holland, 1984. Revised edition.

[5] Hans Bekič. *Programming Languages and their Definition*, volume 177 of *LNCS*. Springer-Verlag, 1984.

[6] Richard Bird. Constructive functional programming. In *Proc. Marktoberdorf International Summer School on Constructive Methods in Computer Science*. Springer-Verlag, 1989.

[7] Richard Bird and Oege de Moor. The algebra of programming. In preparation, 1994.

[8] Richard Bird and Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988.

[9] Gert Florijn. Modelling office processes with functional parsers. University of Utrecht, The Netherlands, 1994.

[10] Maarten Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, March 1992.

[11] Peter Freyd. Algebraically complete categories. In A. Carboni et al, editor, *Proc. 1990 Como Category Theory Conference*, volume 1488 of *Lecture Notes in Math*, pages 95–104. Springer-Verlag, Berlin, 1990.

[12] Peter Freyd. Recursive types reduced to inductive types. In *Proc. LICS 90*. IEEE Computer Society Press, 1990.

[13] Jeremy Gibbons. *Algebras for Tree Algorithms*. PhD thesis, Oxford University, September 1991.

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

[15] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[16] Mark Jones. Gofer 2.28 release notes. February 1993.

[17] Mark Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proc. FPCA 93*. Springer, 1993.

[18] Mark Jones. Personal communication, May 1994.

[19] Mark Jones and Erik Meijer. Gofer goes bananas. In preparation, 1994.

[20] Grant Malcolm. Algebraic data types and program transformation. *Science of Computer Programming*, 14(2-3):255–280, September 1990.

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

[22] Lambert Meertens. Algorithmics: Towards programming as a mathematical activity. In *Proc. CWI Symposium*, Centre for Mathematics and Computer Science, Amsterdam, November 1983.

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

[24] Erik Meijer. *Calculating Compilers*. PhD thesis, Nijmegen University, February 1992.

[25] Erik Meijer. More advice on proving a compiler correct: Improve a correct compiler. Submitted for Publication, September 1994.

[26] Erik Meijer. Type classes for better free theorems. In preparation, 1994.

[27] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Proceedings FPCA 91*, number 523 in LNCS. Springer-Verlag, 1991.

[28] Ross Paterson. Control structures from types. Submitted for publication, 1994.

[29] Andrew M. Pitts. Relational properties of recursively defined domains. In *Proc. LICS 93*. IEEE Computer Society Press, 1993.

[30] John C. Reynolds. On the relation between direct and continuation semantics. In Jacques Loeckx, editor, *Proc. 2nd Colloquium on Automata, Languages and Programming*, number 14 in LNCS, pages 141–156. Springer-Verlag, 1974.

[31] John C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing*, 83:513–523, 1983.

[32] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

[33] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proc. FPCA 93*. Springer, 1993.

[34] Mike B. Smyth and Gordon D. Plotkin. The category theoretic solution of recursive domain equations. *SIAM Journal of Computing*, pages 761–783, 1982.

[35] Doaitse Swierstra and Oege de Moor. Virtual data structures. Technical Report RUU-CS-92-16, Utrecht University, The Netherlands, 1992.

[36] Philip Wadler. Theorems for free! In *Proc. FPCA 89*. Springer, 1989.

[37] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.