


isomorphism) as the least fixed point \( \text{Rec} (\text{Square} (\text{MyRec} \ f)) \) of the covariant functor \( \text{Square} (\text{MyRec} \ f) \).

Another way of showing that covariant functors suffice is to first eliminate the use of mutual recursion in the definitions of \text{cata} and \text{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 \text{cata} and \text{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 \( \text{Twist} \ a \ b \) of lists of alternating elements of type \( a \) and type \( b \), and the datatype \( \text{Nest} \ a \) of lists of nested lists:

\[
data \text{Twist} \ a \ b = \text{Nil} \mid \text{Cons} a (\text{Twist} b \ a)
data \text{Nest} \ a = \text{Block} a (\text{Nest} [a])
\]

To our knowledge, it is not in general known how to express non-regular datatypes as fixed points of functors (or difunctions). Note however that it is possible, with some effort, to express non-regular datatypes as fixed points of type constructors, by using a \( \text{Rec} \) of kind \((\ast \rightarrow \ast) \rightarrow (\ast \rightarrow \ast) \) instead of kind \((\ast \rightarrow \ast) \rightarrow \ast\).

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 \( \text{Rec} \), leading to the introduction of the fictitious constructor \( \text{In} \). 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

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

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

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

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 f (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
or in diagrammatic form,

or in diagrammatic form,

\[
\begin{array}{cccc}
\text{a} & \text{f} & \text{b} & \phi \\
\text{g} & \text{dimap} & \text{h} & \text{h} \\
\text{c} & \text{f} & \text{d} & \phi \\
\end{array}
\]

and

\[
\begin{array}{cccc}
\text{d} & \text{f} & \text{c} & \psi \\
\text{h} & \text{dimap} & \text{g} & \text{g} \\
\text{b} & \text{f} & \text{a} & \psi \\
\end{array}
\]

\[\Rightarrow\]

\[
\begin{array}{cccc}
\text{Rec f} & \text{cata }\phi & \psi & \text{b} \\
\text{cata }\phi & \psi & \text{d} & \text{h} \\
\text{Rec f} & \text{ana }\phi & \psi & \text{c} \\
\text{ana }\phi & \psi & \text{a} & \text{g} \\
\end{array}
\]

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 \(\text{up} :: b \rightarrow a\) and \(\text{down} :: a \rightarrow b\) such that \(\text{up}.\text{down} = \text{id} :: a \rightarrow a\). In other words, \(\text{down}\) is an injective function with \(\text{up}\) as a left-inverse. In Gofer, the notion of a retract can be encapsulated as a type class, as follows:

\[
\text{class Retract } a \rightarrow a \text{ where}
\]

\[
\text{up} :: b \rightarrow a
\]

\[
\text{down} :: a \rightarrow b
\]

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

\[
\text{instance (Difunctor } f, \text{ Retract } (f a a) a \Rightarrow \text{ Retract } a (\text{Rec } f) \text{ where}}
\]

\[
\text{up} = \text{ana (up :: Retract } (f a a) a \Rightarrow f a a \rightarrow a) \quad \text{down} :: \text{Retract} (f a a) a \Rightarrow f a a \rightarrow a
\]

\[
\text{instance Functor Env where}
\]

\[
\text{map} = \text{mapEnv}
\]

The standard (call by name) interpreter for the untyped \(\lambda\)-calculus is obtained by taking the reflexive datatype \(\\text{Scott}\) of section 3.3 as the semantic domain:

\[
\text{instance Retract } \text{Scott } (\text{Scott} \rightarrow \text{Scott}) \text{ where}
\]

\[
\text{up} := \text{In} (\text{Func } f) \Rightarrow f \quad \text{down } f := \text{In} (\text{Func } f)
\]

\[
\text{instance Reflexive } \text{Scott}
\]

\[
\text{instance LambdaModel } \text{Scott}
\]

An appropriate reflexive type \(\\text{Cont}\) for a (call by name) continuation-based semantics for \(\lambda\)-expressions is defined by:
### 4.1 Difunctors and recursive datatype

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

\[
\text{data} \ \text{Rec} \ f = \ \text{In} \ (f \ (\text{Rec} \ f) \ (\text{Rec} \ f)) \ (- \ \#\text{STRICT}\# -)
\]

### 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 \( \text{In} :: f \ (\text{Rec} \ f) \ (\text{Rec} \ f) \rightarrow \text{Rec} \ f \). It is also the minimal \( f \)-invariant, in the sense that \( \text{copy = id} \), where

\[
\text{copy} :: \text{Difunctor} \ f \Rightarrow (\text{Rec} \ f \rightarrow \text{Rec} \ f)
\]

This definition can be expressed in diagrams by

\[
\begin{array}{c}
\text{(Rec} \ f \text{)} \ (f) \ \text{In} \ \rightarrow \text{Rec} \ f \\
\text{copy} \ \text{dimap} \ \text{copy} \\
(\text{Rec} \ f) \ (f) \ \text{In} \ \rightarrow \text{Rec} \ f
\end{array}
\]

Note that by drawing the arrows \( g :: a \rightarrow b \) and \( h :: c \rightarrow d \) of a difunctor \( g \ (\text{dimap} \ h) :: (b \ (f \ c) \rightarrow (a \ (f \ d)) \) separately, both the contravariance and typing assumptions of \( \text{dimap} \) are made explicit.

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

\[
\text{cata phi (In} \ x) = \phi (((\text{cata phi} \ (\text{dimap} \ (\text{cata phi})) \ x)
\]

However this definition is too restrictive, since it forces the argument function \( \phi \) to have type \( (\text{Rec} \ f) \rightarrow \text{Rec} \ f \), and \( \text{cata phi} \) itself to have type \( \text{Rec} f \rightarrow \text{Rec} f \).

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

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

\[
\begin{array}{c}
\text{Rec} \ f \ (f) \ \text{In} \ \rightarrow \text{Rec} \ f \\
\text{g} \ \text{dimap} \ \text{cata phi} \\
\text{b} \ (f) \ \text{In} \ \rightarrow \text{Rec} \ f \\
\text{phi a}
\end{array}
\]

A similar problem occurs with the naive generalisation of \( \text{copy} \) to obtain an anamorphism functional:

\[
\text{ana psi x} = \text{In} (((\text{ana psi} \ (\text{dimap} \ (\text{ana psi})) \ (\text{psi} \ x)))
\]

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

\[
\begin{array}{c}
a \ (f) \ b \\
\psi \\
\text{ans psi}
\end{array}
\]

\[
\begin{array}{c}
\text{Rec} \ f \ (f) \ (\text{Rec} f) \ \text{out} \ (\text{Rec} f)
\end{array}
\]

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

\[
\begin{array}{c}
\text{cata psi (In} \ x) = \phi (((\text{cata psi} \ (\text{dimap} \ (\text{cata psi}))) \ x)
\end{array}
\]

\[
\begin{array}{c}
\text{ana psi x} = \text{In} (((\text{ana psi} \ (\text{dimap} \ (\text{ana psi})) \ (\text{psi} \ x)))
\end{array}
\]

Note that the difunctor versions of \( \text{cata psi} \) and \( \text{ana psi} \) 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 \( \text{cata psi} \) and \( \text{ana psi} \) functional for difunctors satisfy a fusion law, which arises as a free theorem. Because the difunctors versions of \( \text{cata psi} \) and \( \text{ana psi} \) 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 \( h \),

\[
\begin{array}{c}
h,\phi \ = \ \phi',(g \ (\text{dimap} \ h)) \\
\wedge \psi \ . g \ = \ (h \ (\text{dimap} \ g)),\psi'
\end{array}
\]

\[
\begin{array}{c}
\Rightarrow \ h,((\text{cata psi} \ (\psi)) \ = \ \text{cata psi}',\psi') \\
\wedge \ (\text{ana psi} \ (\psi)),g \ = \ (\text{ana psi}',\psi')
\end{array}
\]
An example where \((-\to\)) is used contravariantly is in the definition of a type \(\text{Scott}\) for modelling the untyped (lazy) \(\lambda\)-calculus [1]:

\[
\text{data } S \ s = \text{Func } (s \to s)
\]

\[\text{type } \text{Scott} = \text{Rec } S\]

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 \((-\to\)), and covariant otherwise. The argument \(a\) to \(S\) above occurs both covariantly \((a \to g)\) and contravariantly \((g \to a)\). 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 \(a\) in the definition of \(S\) explicit by defining a binary type constructor \(S'\):

\[
\text{data } S' \ s' = \text{Func } (s \to s')
\]

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 \(g = (a \to b)\) and \(h = (c \to d)\) to a function \(g \cdot \dimap\ h = (f \ a\ b \to f\ c\ d)\). A difunctor must also preserve the identity function and distribute over function composition in the following way:

\[
\text{id} \cdot \dimap = \text{id} \quad (g \cdot \dimap) \cdot (i \cdot j) = (h \cdot \dimap) \cdot (g \cdot \dimap)\]

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

\[
\text{class } \text{Difunctor } f \text{ where}
\dimap :: (a \to b) \to (c \to d) \to (f \ a\ b \to f\ c\ d)
\]

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

\[
\text{instance } \text{Difunctor } S' \text{ where}
\dimap :: (a \to b) \to (c \to d) \to (f \ a\ b \to f\ c\ d)
\]

In the above, the \(\text{Rec}\) constructor only plays an auxiliary rôle. In fact, \(S'\) is a difunctor because the function-space constructor \((-\to\)) is itself a difunctor:

\[
\text{instance } \text{Difunctor } (-\to) \text{ where}
\dimap :: (a \to b) \to (c \to d) \to (f \ a\ b \to f\ c\ d)
\]

In general, by separating the covariant and contravariant occurrences of the argument \(a\) in the body of a non-recursive datatype declaration data \(F\ a = \ldots\), 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.
In this case there is no strictness requirement on h, since dualising h, \texttt{bet \rightarrow bot} gives \texttt{bet \rightarrow bot}, which is true for all functions h. Using fusion for anamorphisms, together with the fact that \texttt{ana out} is the identity function on \texttt{Rec f}, we can show that \texttt{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 \texttt{ana} followed by a \texttt{cata}. Let us briefly show how paramorphisms can be implemented in Gofer. The first step is to define a family of functors \texttt{P f}, one for each functor \texttt{f}:

```haskell
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 (\(\langle x, a \rangle \rightarrow (x, g a)\))

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

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

```haskell
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 (\(\langle a -> (a, a) \rangle \)) x
```

Again for technical reasons concerning types, the definition for \texttt{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 \texttt{ana} followed by a \texttt{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 \texttt{cata} and \texttt{ana} as the only means to define recursive functions are not limited in expressive power.

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

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

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

```haskell
hylo :: Functor 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 fixed together to give a direct recursive definition that avoids building an intermediate [37] (or virtual [33]) value:

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

For example, if we express \texttt{fix} as a hylomorphism rather than the composition of a \texttt{cata} and an \texttt{ana}.

```haskell
fix f = hylo (\(\langle Cons f x \rangle \rightarrow f x \)) (\f -> Cons f f)
```

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

### 3 Problems with exponentials

In the previous section we reviewed how the functionals \texttt{foldr} and \texttt{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 \((\rightarrow)\) is contravariant in its first argument. The effect is that certain type constructors defined using \((\rightarrow)\) cannot be made into functors, and as a result, functionals such as \texttt{cata} and \texttt{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 \((\rightarrow)\)

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

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

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

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

```haskell
instance Functor (\(\rightarrow a\)) where
    map = (\cdot)
```

Now \texttt{S a b} can be made into a functor, as follows:

```haskell
instance (Functor (\(\rightarrow a\)), 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 \texttt{S a b} is not recursive; the uses of \texttt{map} in its definition are those for \((\rightarrow a)\) and lists \([]\). A function \texttt{exec} that forces evaluation of a state to its set of final values can now be defined as a catamorphism:
functor E for arithmetic expressions. The following function (which replaces the constructors \texttt{Num} and \texttt{Add} by the functions \texttt{id} and \texttt{(+)} ) is an E-algebra of type \texttt{E Int} -> \texttt{Int}: 
\[
\begin{align*}
(\text{x} & \mapsto \text{x of Num n} 
\text{ Add } e e' & \mapsto e + e')
\end{align*}
\]
Applying \texttt{cata} to the above algebra gives the standard evaluator for arithmetic expressions:
\[
\begin{align*}
\text{eval} :: \text{Expr} -> \text{Int} \\
\text{eval} = \text{cata } (\lambda x \mapsto \text{case x of Num n} \mapsto \text{id n} \\
\text{ Add } e e' \mapsto (\text{eval e}) + (\text{eval e}')
\end{align*}
\]

This definition says that expressions can be evaluated by simultaneously replacing all \texttt{Num} constructors by the \texttt{id} function on integers, and all \texttt{Add} constructors by \texttt{(+)} on integers. Unfolding the definition to eliminate the use of \texttt{cata} and \texttt{map} makes clear that it has the expected behaviour:
\[
\begin{align*}
\text{eval } (\text{In } x) = \\
\text{ case x of Num n \mapsto \text{id n} \\
\text{ Add } e e' \mapsto (\text{eval e}) + (\text{eval e}')
\end{align*}
\]

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 \texttt{cata} :: \texttt{Functor f} \Rightarrow (f a -> a) -> (Rec f a -> a) is the well-known fusion law [27]: for strict functions h,
\[
\begin{align*}
\text{h phi} = \text{phi'.(map h)} \Rightarrow \text{h.(cata phi)} = \text{cata phi'}
\end{align*}
\]

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{align*}
\begin{tikzpicture}[node distance=2cm,auto,>=latex]
  \node (a) {a}; \\
  \node (b) [right of=a] {b}; \\
  \node (c) [below of=a] {h}; \\
  \node (d) [below of=b] {h'}; \\
  \node (e) [below of=c] {\text{cata phi}}; \\
  \node (f) [below of=d] {\text{cata phi'}}; \\
  \node (g) [below of=e] {\text{Rec f}};

  \draw[->] (a) -- node[auto] {\text{map h}} (b);
  \draw[->] (a) -- (c);
  \draw[->] (b) -- (d);
  \draw[->] (c) -- node[auto] {\phi} (e);
  \draw[->] (d) -- node[auto] {\phi'} (f);
  \draw[->] (e) -- node[auto] {} (g);
  \draw[->] (f) -- node[auto] {} (g);
\end{tikzpicture}
\end{align*}
\]

Fusion captures a common pattern of inductive proof on programs expressed as catamorphisms, in a similar way to that in which \texttt{cata} itself captures a common pattern of recursion over polynomial datatypes. Minimality and fusion can together be used to show that \texttt{cata phi} satisfies a universal property, namely that \texttt{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 \texttt{[Int]} to store intermediate values. Such a stack-based evaluator can be defined as follows:
\[
\begin{align*}
\text{eval1} :: \text{Expr} -> (\text{[Int]} -> \text{[Int]}) \\
\text{eval1} = \text{cata } (\lambda x \mapsto \text{case x of Num n} \mapsto \text{push n} \\
\text{ Add } e e' \mapsto (\text{eval e}) (\text{eval e}')
\end{align*}
\]
where \texttt{push a} as \texttt{a:as} pushes a number onto the stack and \texttt{add (a:b:cs)} = \texttt{(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 :: \texttt{Expr} we have \texttt{push (eval e)} = \texttt{eval1 e}, can easily be proved using fusion and the distribution of \texttt{push} over addition: \texttt{push (a+b)} = \texttt{add (push a) (push b)} [25].

2.4 Coalgebras and anamorphisms
Using \texttt{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
\[
\begin{align*}
\text{copy :: Functor f} \Rightarrow (\text{Rec f} \rightarrow \text{Rec f}) \\
\text{copy x = In (map copy (out x))}
\end{align*}
\]
where \texttt{out (In x) = x} is the inverse of the isomorphism \texttt{In}. If we now generalise this version of copy by replacing the occurrence of \texttt{out} :: \texttt{Rec f -> f (Rec f)} in its definition by an arbitrary function \texttt{psi :: a -> f (an f-coalgebra)}, we obtain the notion of an \textit{anamorphism} [27]:
\[
\begin{align*}
\text{ana :: Functor f} \Rightarrow (a -> f a) -> (a -> \text{Rec f}) \\
\text{ana psi x = In (map (ana psi) (psi x))}
\end{align*}
\]
The functional \texttt{ana}—written as ‘lens’ brackets \texttt{[ ]} in the Squiggol literature—is the generic version of the recursion functional \texttt{unfold} [8, p173] on lists. The Greek preposition \texttt{ana} means upwards, and its use here reflects the fact that \texttt{ana psi} recursively builds up its result by decomposing its argument using the function \texttt{psi}.

We illustrate the notion of an anamorphism by defining a function \texttt{n2b} that converts natural numbers to their binary representation. The first step is to define a type \texttt{Bin} of binary numbers as the fixed point of a functor \texttt{B}:
\[
\begin{align*}
\text{data B b = Empty } | \text{ Zero b } | \text{ One b}
\end{align*}
\]

\[
\begin{align*}
\text{instance Functor B where}
\text{ map g } = \lambda x \mapsto \text{case x of Empty } \mapsto \text{Empty}
\text{ Zero b } \mapsto \text{Zero (g b)}
\text{ One b } \mapsto \text{One (g b)}
\end{align*}
\]

\[
\begin{align*}
\text{type Bin } = \text{Rec B}
\end{align*}
\]

The binary representation of a natural number is built by recursively splitting off its least significant bit:
\[
\begin{align*}
\text{n2b :: Int} \rightarrow \text{Bin} \\
\text{n2b = ana } (\lambda x \mapsto \text{case x of 0} \mapsto \text{Empty} \\
\text{2*n} \mapsto \text{Zero n} \\
\text{2*n+1} \mapsto \text{One n})
\end{align*}
\]

For example, \texttt{n2b 2 = In (Zero (In (One (In Empty))))}.

The dual function \texttt{b2n} that converts a binary number back to a natural number can be defined as a catamorphism:
\[
\begin{align*}
\text{b2n :: Bin} \rightarrow \text{Int} \\
\text{b2n = cata } (\lambda x \mapsto \text{case x of Empty } \mapsto 0 \\
\text{ Zero b } \mapsto 2*b \\
\text{ One b } \mapsto 1+2*b)
\end{align*}
\]

2.5 Free theorems and fusion

The free theorem for the functional \texttt{ana} :: \texttt{Functor f} \Rightarrow (a -> f a) -> (a -> \text{Rec f}) is also a fusion theorem:
\[
\begin{align*}
\text{psi.h = (map h).psi'} \Rightarrow (\text{ana psi}).h = \text{ana psi'}
\end{align*}
\]
or in diagrammatic form,
\[
\begin{align*}
\begin{tikzpicture}[node distance=2cm,auto,>=latex]
  \node (a) {a}; \\
  \node (b) [right of=a] {b}; \\
  \node (c) [below of=a] {h}; \\
  \node (d) [below of=b] {h'}; \\
  \node (e) [below of=c] {\text{ana psi}}; \\
  \node (f) [below of=d] {\text{ana psi'}};

  \draw[->] (a) -- node[auto] {\text{map h}} (b);
  \draw[->] (a) -- node[auto] {} (c);
  \draw[->] (b) -- node[auto] {} (d);
  \draw[->] (c) -- node[auto] {\text{psi}} (e);
  \draw[->] (d) -- node[auto] {\text{psi'}} (f);
  \draw[->] (e) -- node[auto] {} (f);
\end{tikzpicture}
\end{align*}
\]
map g :: f a -> f b. In Gofer, the concept of a functor can be encapsulated as a constructor class, as follows:

```haskell
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 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:

```haskell
instance Functor [] where
    map f x = [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:

```haskell
map id = id
map (g . h) = (map g) . (map h)
```

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:

```haskell
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:

```haskell
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:

```haskell
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`:

```haskell
type Expr = Rec E
```

Some illustrative values of type `Expr` are:

```haskell
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 role, 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

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

do not permit lists with elements of type `a` to be defined as follows:

```haskell
data L a 1 = Nil | Cons a 1
```

```haskell
instance Functor (L a) where
    map g = \x -> case x of
        Nil    -> Nil
        Cons a 1 -> Cons a (g 1)
```

It (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 `a` and `b` 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

```haskell
copy :: Functor f => (Rec f -> Rec f)
copy (In x) = In (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` by itself but by an arbitrary function `phi :: f a -> a`. In this way we obtain the notion of a `catamorphism` [27]:

```haskell
cata :: Functor f => (f a -> a) -> (Rec f -> a)
cata phi (In x) = phi (copy (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 σαντιμ, 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
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 of cpos and embedding-projection pairs, on which category the exponential functor can be made covariant [34]. But while the setting of cpo 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 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