On the expressive power of Constructor Classes

Luc Duponcheel
Vakgroep Informatika, Rijks Universiteit Utrecht
Utrecht, The Netherlands `luc@cs.ruu.nl`

Erik Meijer
Vakgroep Informatika, Rijks Universiteit Utrecht
Utrecht, The Netherlands `erik@cs.ruu.nl`

September 12, 1995

Abstract

The aim of this paper is to explore the expressive power of constructor classes, a generalisation of type classes. We present a categorical `prelude` for `Gofer`, a programming environment which supports constructor classes. The `prelude` contains much more code than the one which is dealt with in the paper. We do not go into all the details of the `prelude`. We explain *why* we wrote it and *how* certain language constructs have made it possible to do so.

1 Introduction

Every programming language comes with its standard libraries. If a new release of a programming language introduces new constructs, then it is challenging to make these libraries more useful by making use of the expressive power of those constructs. The introduction of *type classes* in `Haskell` (see [3]) has had an influence on the design of its standard `prelude`. The `prelude` contains, among others, type classes `Text`, `Eq` and `Num` which make it possible to overload standard function and operator notation for a variety of type class instances. `Gofer` is the first programming language which supports *constructor classes* (see [4]). Constructor classes are a generalisation of type classes. Thus, as stated above, it is challenging to make use of the expressive power offered by them. This paper tries to give an idea about which kind of things are possible.

In the paper we show how to write an categorical `prelude` for `Gofer` from scratch. It turns out that a considerable amount of useful concepts from category theory can be encoded in `Gofer` by making use of its constructor classes. Such an encoding makes it possible to use of a uniform notation for a variety of similar programming problems. Well-known general purpose functions like `map` and `foldr` (plus a whole range of functions which are defined in terms of them) can be used in different (but similar) situations. The categorical encoding also encourages a modular approach to solving programming problems.

Non-trivial pieces of code can be obtained by combining and/or transforming other, conceptually simpler, pieces of code.

The main part of the paper simply consists of a thourough study of (part of) the categorical `prelude`. The style of the paper is sometimes a little bit unusual: some (sub)sections start with a piece of code. Do not try to understand the code at once: the aim of the (sub)section is precisely to explain that piece of code. The `prelude` contains much more code than the one which is explained in the paper. The number of classes (and the corresponding categorical concepts) which occur in the `prelude` may, at first reading, be a little bit overwhelming. Therefore we decided *not* to go into the details of all of them. In the introduction we explain *why* these classes are introduced, i. e. which constructs they are generalisations of. In the main part of the paper we show *how* to introduce the classes. We concentrate on the language features of `Gofer` which make it possible to encode categorical concepts in a convenient way. For more details we refer to a forthcoming report [2]. In that report we show (among other things) how to write a generic categorical lambda-calculus interpreter. It is based upon the following standard result: any reflexive object of any cartesian closed category can be used as a denotational model for lambda-calculus. Specific reflexive objects can be chosen to obtain call-by-value, call-by-name resp. call-by-need semantics. The forthcoming report does not explain all the details either ; explaining all the details of the `prelude` would probably lead to a book rather than a report or a paper. In fact, many researchers have suggested us that we write a book, similar to the one of Burstall and Rydeheard (see [1]), about the contents and usage of the categorical `prelude`.

1.1 Functions

Functional programming is based upon lambda-calculus. This means that, if we start `Gofer` with a minimal non-empty `prelude` e. g.

```
> bottom = bottom
```

then we can:

- *define* functions (lambda-abstraction), and
- *use* functions (lambda-application).

One of the main ideas of the paper is to generalise functions by defining a `Category` class which introduces morphisms[1]. Functional programs use functions as first class citizens. We model this fact in our categorical framework by

[1] The possibility to encode a `Category` class in `Gofer` was first recognised by Mark P. Jones. A first definition of such a class appeared in the file `fancycat` of the `demo` directory the `Gofer` `2.28` release.

defining a `FunCategory` class which introduces internal functions. A considerable amount of useful code dealing with functions can be generalised to the framework of morphisms and internal functions.

1.2 Data Definitions

Data types can be introduced in `Gofer` by making use of `data` definitions. Lists can, for example, be defined as follows:

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

Even a simple example like this one makes use of three important data structuring facilities:

- **Products** : the `Cons` data constructor constructs compound data out of constituents `a` and `List a`.
- **Sums** : the `|` construct splits up the `List a` data type in two parts: a `List a` expression is either `Nil` or an expression of the form `Cons a as`, where `as` is a `List a` expression.
- **Recursion** : `List a` occurs both at the left and at the right hand side of its definition. This means that `List a` is defined recursively (in terms of itself).

One of the main ideas of the paper is to generalise data definitions by defining a class `ProdCategory` which introduces internal products, a class `SumCategory` which introduces internal sums and a class `RecCategory` which introduces internal recursion.

By the way, in [6] yet another useful data structuring facility is introduced: the subtyping mechanism. Subypes can e. g. be defined as follows:

```
> class SubTypeOf sup sub where
>   inj :: sub -> sup
>   prj :: sup -> sub
```

Any type is a subtype of itself.

```
> data Only x = Only x

> instance SubTypeOf (Only x) x where
>   inj x = Only x
>   prj (Only x) = x
```

We have introduced a dummy data constructor `Only` to avoid overlapping instances. By making use of the `|` construct, it is possible to define an important range of `SubType` instances as follows:

```
> data AddTo x y = Old x | New y

> msg = "don't know where to go to"

> instance SubTypeOf sup sub
>         => SubTypeOf (AddTo sup any) sub where
>   inj x = Old (inj x)
>   prj (Old x) = prj x
>   prj (New _) = error msg
```

In other words: adding `any` data to `sup` preserves the `SubTypeOf` relation with `sub`. Given types `One`, `Two`, `Three`, ..., one can now construct towers of types:

```
type First  = Only  One
type Second = AddTo First  Two
type Third  = AddTo Second Three
...
```

Subtyping is one of the facilities which are used in [6] for writing complex programs such as interpreters in a modular way. The facilities (subtyping, monad transformers, ...) which are used are precisely the ones which we have tried to implement in our categorical `prelude`.

1.3 Restricted Type Synonyms

As explained in 1.2, one of the main ideas of the paper is to replace data definitions by appropriate class definitions. But how can we define data types if we are not allowed to use any data definitions at all? `Gofer` offers an alternative to data definitions: restricted type synonyms. Type synonyms are (as the name suggests) aliases for other type expressions[2]. Restricted type synonyms are aliases only for a particular list of functions. Outside the scope of those functions they behave as (untagged) data definitions. Restricted type synonyms can be used to define abstract data types (adts). The particular list of functions contains the functions which are allowed to access the internal repesentation of the adt. Here is a (simplified) typical example:

```
> class EuclidSpace v where
>  (<*>) :: v -> v -> Float

> angle :: EuclidSpace v => v -> v -> Float
> angle v w = acos (v<*>w / sqrt (v<*>v + w<*>w))

> type Vector = [Float] in inprod
```

[2] Constructor synonyms would be a better name since they can also be used to define aliases for constructor expressions.

```
> inprod :: Vector -> Vector -> Float
> xs `inprod` ys = sum (zipWith (*) xs ys)

> instance EuclidSpace Vector where
>   (<*>) = inprod
```

The `inprod` function is allowed to access the internal representation of the `Vector` adt. One can now define `Vector` as an instance of the `EuclidSpace` class and, as a consequence, use the `angle` function for computing the angle between two vectors. Adts can, of course, also be introduced using data definitions:

```
> data Vector = Vec [Float]

> instance EuclidSpace Vector where
>   Vec xs <*> Vec ys = sum (zipWith (*) xs ys)
```

However, there is an important difference. Suppose that the source code of an adt is included in a library package as a *read-only* file. If the adt code is written using a restricted type synonym, then it is not possible to access the internals of the adt since the list of access-functions cannot be extended in another file. If the adt code is written using a data definition, then one can, in any other file, make use of the data constructor to open the adt. Here is the adt-opening code for vectors:

```
> openVec :: Vector -> [Float]
> openVec (Vec xs) = xs
```

As can be seen from the `Vector` example, there is a (minor) programming inconvenience related to the usage of restricted type synonyms when defining class instances: one has to introduce auxiliary access-functions. When working with data definitions, class instances can be defined without introducing auxiliary functions. One final remark about resticted type synonyms: it is often needed to associate an access-function with more than one adt (this corresponds to the friend construct in C^{++}) . Here is an example:

```
> class Apply m v where
>   (#) :: m -> v -> v

> type Matrix = [[Float]] in app

> app :: Matrix -> Vector -> Vector
> xss `app` ys = [ xs <*> ys | xs <- xss ]

> instance Apply Matrix Vector where
>   (#) = app
```

The types of this code are correct only if `app` is added to the list of access-functions of `Vector`.

1.4 Constructor Classes

Constructor classes are a generalisation of type classes. Constructor classes were introduced in `Gofer` starting from release `2.28` (see [4]). We have already presented a type class `EuclidSpace` in subsection 1.3. In this subsection we present a typical example of a constructor class.

```
> class Stack s where
>   empty :: s a
>   push :: a -> s a -> s a
>   pop :: s a -> s a
>   top :: s a -> a
```

In contrast with the example of subsection 1.3, where we defined a class of types, the example of this subsection defines a class of constructors. A (unary) constructor `s` is a `Stack` instance if the usual operations are defined on *any* type `s a`. Lists can be defined as `Stack` instances as follows:

```
> instance Stack List where
>   empty = Nil
>   push = Cons
>   pop (Cons _ xs) = xs
>   top (Cons x _) = x
```

Constructor classes are, in our opinion, one of the most important tools for structuring functional programs. One of the main ideas of the `prelude` is to make use of constructor classes to encode data type structuring facilities.

1.5 Squiggol Syntax

Squiggol syntax is introduced as a notation for supporting the Bird-Meertens programming style. We have chosen the syntax of the class members of the categorical `prelude` in such a way that it is possible to make use of Squiggol-like syntax in `Gofer`. To give an idea about the kind of notation that will enter the picture in the main body of the paper, we now show how one can quickly introduce Squiggol notation in `Gofer` (in the main body of the paper we will do this in a more structured way).

```
> type Mor = (->)
> type Prod = (,)
```

```
> infixl 8 <*>, >*<
> infixr 7 <+>, >+<

> exl (x,y) = x
> exr (x,y) = y

> f <*> g = \x -> (f x,g x)
> f >*< g = \(x,y) -> (f x,g y)

> data Sum x y = L x | R y

> inl x = L x
> inr y = R y

> f <+> g = \s -> case s of L x -> f x ; R y -> g y
> f >+< g = \s -> case s of L x -> L (f x) ; R y -> R (g y)

> data RecR f x = InR (f x (RecR f x))

> isoR x = InR x
> osiR (InR x) = x

> class FunctorR f where
>   mapR :: Mor u v -> Mor (f x u) (f x v)

> cataR :: FunctorR f => Mor (f a x) x -> Mor (RecR f a) x
> cataR phi = phi . mapR (cataR phi) . osiR
```

The `Sum` and `RecR` data definitions are generic. The idea is to define *all* other data definitions in terms of them (and, of course, also `Mor` and `Prod`). The function `cataR` can be used to define a whole range of recursive functions on recursive types `RecR f a`. This function generalises the `foldr` function on lists. Using recursion in a structured way (i. e. via `cata`) makes it possible to write reusable programs.

1.5.1 Binary Trees

In this subsection we show how trees can be defined using the Bird-Meertens formalism (lists are defined in a similar way). In subsection 2.11.2 we will show how to define trees in our categorical framework without using any data definitions at all.

```
> type T a x = Sum a (Prod x x) in leafT, nodeT,
>                                  mapRT, mapLT, revT
```

```
> type Tree = RecR T

> leafT :: Mor a (T a x)
> leafT = inl

> nodeT :: Mor (Prod x x) (T a x)
> nodeT = inr

> leaf :: Mor a (Tree a)
> leaf = isoR . leafT

> node :: Mor (Prod (Tree a) (Tree a)) (Tree a)
> node = isoR . nodeT
```

Notice that we have started with a binary (non-recursive) constructor `T` and ended up with a unary (recursive) constructor `Tree`. The type `Tree a` is essentially the same as the one which is defined using the data definition:

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

Up to now we have only given the code for functions `leaf` and `node` which can be used to construct trees. Next we turn `T` into a `FunctorR` instance. This makes it possible to define recursive functions on trees using `cataR`.

```
> mapRT :: Mor x y -> Mor (T a x) (T a y)
> mapRT g = id >+< g >*< g

> instance FunctorR T where
>   mapR = mapRT
```

Here comes a simple example of a recursive function on trees: the function `revTree` which reverses a tree.

```
> revT :: Mor (T a (Tree a)) (Tree a)
> revT = leaf <+> node . (exr <*> exl)

> revTree :: Mor (Tree a) (Tree a)
> revTree = cataR revT
```

The function `revT` swaps the top level subtrees (if any). The full power of `revTree` comes from `cataR` which, when going down into the subtrees, recursively keeps on swapping the top level subtrees. It is also possible to make `Tree` an instance of useful classes. A typical example is the `Functor` class which makes it possible to overload the `map` notation when iterating through a tree:

```
> mapLT :: Mor a b -> Mor (T a x) (T b x)
> mapLT g = g >+< id >*< id

> instance Functor Tree where
>   map g = cataR (isoR . mapLT g)
```

2 The `prelude`

Now we are ready for the main part of the paper: the actual code of the `prelude`. The code contains a lot of restriced type synonyms. The access-functions for those restricted type synonms will (just as in the tree example of subsection 1.5.1) be introduced gradually. Therefore we will often write ... after their list of access-functions to indicate that there may be other access-functions following.

2.1 Identity and Composition

Together with abstraction and application, composition and identity are perhaps the most important concepts one can make use of when writing functional programs. Composition and identity are so essential for functional programming that it is worth the pain to generalise them from functions to morphisms. For example: in many programmig problems substitutions and/or transformations play a central role. A substitution and/or transformation from `x` to `y` is not really a function from `x` to `y`[3]. However, substitutions and/or transformations *do* have an identity and *can* be composed. The experimental `prelude` establishes a programming environment in which one can, to a large extent, treat substitutions and/or transformations as if they are functions. This does not only simplifies our perception of programming problems which make use of them, it also offers us notational convenience for programming with them. The situation is very similar to the one we encounter when writing technical documents. If we write a document which deals with substitutions and/or transformations, then we deliberately overload the words identity and composition (notice that we actually did so a few lines ago!). This is just a convenient abuse of notation: the context in which we use the words makes it evident which kind of identity and composition we are talking about.

2.2 Categories

```
> class Category mor where
```

[3] Substitutions are functions from `x` to `Term x`, where `Term` is a unary constructor which constructs terms of type `Term x` having variables of type `x`. Transformations are functions from `x` to `Maybe(y,x)`. An input of type `x` *may* be transformed to an output of type `y` (and the rest of the input, of type `x`).

```
>   id :: mor x x
>   (.) :: mor y z -> mor x y -> mor x z
```

In this subsection we deal with the basic properties of the function space constructor `(->)` and generalise these properties to `Category` instances. A binary constructor `mor` (standing for morphism) is a `Category` instance if an appropriate identity morphism `id` and composition meta-operator `(.)` on morphisms are defined. Notice that the objects of a category necessarily correspond to types. The morphims from `x` to `y`, however, do not necessarily need to correspond to functions from `x` to `y`. The most important `Category` instance is, of course, the category of funtions:

```
> type Mor = (->)

> instance Category Mor where
>   id = \x -> x
>   g . f = \x -> g (f x)
```

The identity function does nothing with its argument. Function composition is defined by making explicit use of association to the right (application associates to the left). Notice that, when dealing with functions, we have introduced `Mor` as a type synonym for `(->)`. We will use `(->)` as a meta-notation: morphism composition associates with a morphism from `y` to `z` and a morphism from `x` to `y` a morphism from `x` to `z`. The type of the composition meta-operator for functions instance is:

```
> Mor y z -> Mor x y -> Mor x z
```

We could have written this type differently, for example:

```
> Mor (y -> z) (Mor (x -> y) (x -> z))
```

but this would only lead to confusion. In subsection 2.4 we will, much in the same spirit, introduce a type synonym `Prod` for `(,)` when dealing with products[4].

2.3 Opposite Categories

```
> type OpM mor x y = mor y x in idOpM,compOpM, ...

> idOpM :: Category mor => OpM mor x x
> idOpM = id
```

[4] We will not use `(,)` as a meta-notation. Meta-functions using `(,)` can be converted to meta-functions using `(->)`.

```
> compOpM :: Category mor
>           => OpM mor y z -> OpM mor x y -> OpM mor x z
> g `compOpM` f = f . g

> instance Category mor => Category (OpM mor) where
>   id = idOpM
>   (.) = compOpM
```

With every `Category` instance corresponds an opposite `Category` instance. The introduction of opposite categories reduces the number of classes of the `prelude` considerably: instead of introducing opposite classes over and over again, we simply make use of existing classes, instantiated with opposite categories. The `OpM mor x y` resticted type synonym has a very long list of access-functions: every time we introduce a new class instance which somehow has to deal with opposite categories, we need another access-function.

2.4 Product Categories

```
> class Category mor => FinalCategory mor where
>   final :: mor x ()
>
> class FinalCategory mor => ProdCategory mor prod where
>   exl :: mor (prod x y) x
>   exr :: mor (prod x y) y
>   (<*>) :: mor z x -> mor z y -> mor z (prod x y)
>   (>*<) :: mor u x -> mor v y -> mor (prod u v) (prod x y)

>   f >*< g = f . exl <*> g . exr
```

In subsection 2.2 we have dealt with the basic properties of the function space constructor and we have generalised these properties to `Category` instances. We now deal with the basic properties of the combination of the function space constructor `(->)`, with the product space constructor `(,)` which can be used to define compound data, and generalise these properties to `ProdCategory` instances. Two binary constructors `mor` and `prod` (standing for product) are a `ProdCategory` instance if appropriate extraction morphisms `exl` and `exr` and an appropriate meta-operator `(<*>)` on morphisms are defined. We have also added a meta-operator `(>*<)` to the `ProdCategory` class together with a default definition of it in terms of other members of the class. This gives us the possiblity to override the default definition by a more efficient one whenever we define a particular `ProdCategory` instance. Notice that we have limited the `FinalCategory` instances to those for which the final object is the type `()`. The standard `ProdCategory` instance is:

```
> type Prod = (,)

> instance FinalCategory Mor where
>   final = \_ -> ()

> instance ProdCategory Mor Prod where
>   exl = \(x,_) -> x
>   exr = \(_,y) -> y
>   f <*> g = \x -> (f x,g x)
>   f >*< g = \(x,y) -> (f x,g y)
```

One can now, for example, represent the commutativity property of products as a meta-function:

```
> swap :: ProdCategory mor prod
>        => mor (prod x y) z -> mor (prod y x) z
> swap f = f . (exr <*> exl)
```

2.5 Sum Categories

```
> instance Category mor => InitialCategory mor where
>   initial :: String -> mor a x

> class InitialCategory mor => SumCategory mor sum where
>  inl :: mor x (sum x y)
>  inr :: mor y (sum x y)
>  (<+>) :: mor x z -> mor y z -> mor (sum x y) z
>  (>+<) :: mor x u -> mor y v -> mor (sum x y) (sum u v)

>   f >+< g = inl . f <+> inr . g

> instance InitialCategory Mor where
>   initial = error

> data Sum x y = L x | R y

> instance SumCategory Mor Sum where
>   inl = L
>   inr = R
>   f <+> g = \s -> case s of L x -> f x ; R y -> g y
>   f >+< g = \s -> case s of L x -> L (f x) ; R y -> R (g y)
```

We now deal with the basic properties of the combination of the function space constructor `(->)` with the `|` construct, which can be used to define union data, and generalise these properties to `SumCategory` instances. Two binary constructors `mor` and `sum` (standing for sum) are a `SumCategory` instance if appropriate injection morphisms `inl` and `inr` and an appropriate meta-operator `(<+>)` on morphisms are defined. We have limited the `InitialCategory` instances to those for which the initial object is the polymorphic type `a`. We have also added an extra `String` argument to the `initial` meta-function. This argument is useful for generating error messages.

2.6 Function Categories

```
> class Category mor => FunCategory mor fun where
>   before :: mor x y -> mor (fun y z) (fun x z)
>   flip :: mor x (fun y z) -> mor y (fun x z)
>   bnd :: mor x (fun (fun x y) y)

>   bnd = flip id
```

We now deal with the basic properties of the combination of the function space constructor `(->)` with itself (the function space constructor `(->)` can also be used to define function-like data) and generalise these properties to `FunCategory` instances. Two binary constructors `mor` and `fun` (standing for function) are a `FunCategory` instance if appropriate meta-functions `before` and `flip` are defined. For many instances `mor` and `fun` will be the same binary constructor. The meta-function `before` is a contravariant iternal function composition (notice that, in its signature, `y` and `z` are swapped). The meta-function `flip` is a `FunCategory` equivalent of the `swap` meta-function of subsection 2.4. One can define a `bnd` morphism in terms of `flip`. It is a `FunCategory` equivalent for function argument binding. The standard `FunCategory` instance is:

```
> type Fun = (->)

> instance FunCategory Mor Fun where
>   flip f = \x y -> f y x
>   f `before` g = \x -> g (f x)
>   x `bnd` f = f x
```

Categories with internal functions turn out to be the framework in which it is possible to deal with continuations.

2.7 Closed Categories

```
> class (ProdCategory mor prod, FunCategory mor fun)
>      => ClosedCategory mor prod fun where
>   curry :: mor (prod x y) z -> mor x (fun y z)
>   uncurry :: mor x (fun y z) -> mor (prod x y) z
```

```
>   eval :: mor (prod (fun x y) x) y

>   eval = uncurry id
```

In this subsection we combine the properties of subsection 2.4 and subsection 2.6. There is an important relationship between products and functions: a function from a type `(x,y)` to a type `z` is essentially the same as function from the type `x` to the type `y -> z`. We now generalise this property to `ClosedCategory` instances (using meta-functions `curry` and `uncurry`). One can define a morphism `eval` in terms of `uncurry`. It is a `ClosedCategory` equivalent for function application. The standard `ClosedCategory` instance is:

```
> instance ClosedCategory Mor Prod Fun where
>   curry f = \x y -> f (x,y)
>   uncurry f = \(x,y) -> f x y
>   eval (f,x) = f x
```

Closed categories turn out to be the framework in which it is possible to deal with state. Notice that it is possible to define the `FunCategory` members in terms of the `ClosedCategory` members:

```
> before g = curry (eval . (id >*< g))
> flip = curry . swap . uncurry
```

2.8 Recursion Categories

```
> class Category mor => RecCategory mor rec where
>  iso :: mor (f (rec f)) (rec f)
>  osi :: mor (rec f) (f (rec f))
```

In this subsection we deal with recursive data definitions and generalise them to `RecCategory` instances. A binary constructor `mor` and a unary constructor `rec` (standing for recursion) are a `RecCategory` instance if all objects `rec f` and `f (rec f)` are isomorphic (by means of isomorphisms `iso` and `osi`). The standard `RecCategory` instance is:

```
> data Rec f = Rec (f (Rec f))

> instance RecCategory Mor Rec where
>   iso x = Rec x
>   osi (Rec x) = x
```

The unary constructor `rec` is exactly what we need to define recursive types (i. e. nullary constructors) `rec f`. For dealing with recursive unary constructors, another class (with a binary constructor `recR`) is needed:

```
> class Category mor => RecRCategory mor recR where
>  isoR :: mor (f x (recR f x)) (recR f x)
>  osiR :: mor (recR f x) (f x (recR f x))
```

2.9 Remark

There is nothing magic about the meta-functions and morphisms we have introduced so far: they correspond to the very basic functions which can be found in any other `prelude`. Instead of defining them as functions we have defined them as class members. This gives us the possibility to use them for a variety of class instances. We finish this overview of categories and special categories by showing how to deal with subtypes.

```
> class SumCategory mor sum => SubTypeOf mor sup sub where
>   inj :: mor sub sup
>   prj :: mor sup sub

> msg = "don't know where to go to"

> instance SubTypeOf mor sup sub
>         => SubTypeOf mor (sum sup any) sub where
>   inj = inl . inj
>   prj = prj <+> initial msg
```

2.10 Functors

```
> class (Category mor, Category mor') => Functor mor mor' f where
>   fun :: mor x y -> mor' (f x) (f y)
```

In subsection 2.2 and the ones following it we have defined the `Category` class and some of its specialisations. The next natural step consists of defining a `Functor` class. A `Functor` instance is a unary constructor `f` for which an appropriate meta-function `fun` on morphisms is defined.

2.10.1 Examples

A whole range of general `Functor` instances can be defined. Trying to define those instances both as general and as simple as possible leads to type problems: the `Gofer` type checker will soon complain because of overlapping instances. There is an easy way out of this: one can introduce restricted type synonyms. This makes the code a bit more complex but it also has its advantages: one can introduce meaningful names. Here are some useful `Functor` instances:

```
> type Rd mor r x = mor r x in funRd, ...
```

```
> funRd :: Category mor
>         => mor y z -> Mor (Rd mor x y) (Rd mor x z)
> funRd g f = g . f

> instance Category mor => Functor mor Mor (Rd mor x) where
>   fun = funRd

> type Gn prod x y = prod x y in funGn, ...

> funGn :: ProdCategory mor prod
>         => mor y z -> mor (Gn prod x y) (Gn prod x z)
> funGn f = id >*< f

> instance ProdCategory mor prod
>         => Functor mor mor (Gn prod x) where
>   fun = funGn

> type Ch sum x y = sum x y in funCh, ...

> funCh :: SumCategory mor sum
>         => mor y z -> mor (Ch sum x y) (Ch sum x z)
> funCh f = id >+< f

> instance SumCategory mor sum
>         => Functor mor mor (Ch sum x) where
>   fun = funCh
```

The `Rd mor r` functor (standing for read-functor) can be used for programs which need to read data (e. g. from an environment). The `Gn prod s` functor (standing for generate-functor) can be used for programs which need to generate data (e. g. write trace information). The `Ch sum x` functor (standing for choice-functor) can be used for programs which, depending on some condition, choose a different way to proceed (e. g. for error handling).

```
> type OpP prod x y = prod y x in funOpP ...

> funOpP :: ProdCategory mor prod
>           => mor y z -> mor (OpP prod x y) (OpP prod x z)
> funOpP f = f >*< id

> instance ProdCategory mor prod
>         => Functor mor mor (OpP prod x) where
>   fun = funOpP

> type OpF fun a b = fun b a in funOpF1, funOpF2, ...
```

```
> funOpF1 :: FunCategory mor fun
>           => OpM mor x y -> mor (OpF fun a x) (OpF fun a y)
> funOpF1 = before

> instance FunCategory mor fun
>         => Functor (OpM mor) mor (OpF fun a) where
>   fun = funOpF1

> funOpF2 :: FunCategory mor fun
>           => mor x y -> OpM mor (OpF fun a x) (OpF fun a y)
> funOpF2 = before

> instance FunCategory mor fun
>         => Functor mor (OpM mor) (OpF fun a) where
>   fun = funOpF2
```

The `OpP prod x` functor is not really useful on its own (it is isomorphic to the `Gn prod s` functor). However, it turns out that flipping the constructor arguments of `prod` is exactly what we need to use this functor as a building block for defining a state monad. Similarly, the two `OpF fun a` functors are not really useful on their own. They can be used as building blocks in for defining a continuation monad.

2.10.2 Auxiliary Functions

The `Gofer` type checker complains from time to time when it is offered `fun` (possibly dealing with two underlying `Category` instances `mor` and `mor'`). On the other hand we often do not need the full generality of the `Functor` class: the underlying categories will, as in the examples above, somehow be related. Therefore we introduce some auxiliary meta-functions which provide the type checker with the extra type information it sometimes needs.

```
> endofun :: Functor mor mor f
>           => mor x y -> mor (f x) (f y)
> endofun = fun

> opfunL :: Functor (OpM mor) mor f
>          => OpM mor x y -> mor (f x) (f y)
> opfunL = fun

> opfunR :: Functor mor (OpM mor) f
>          => mor x y -> OpM mor (f x) (f y)
> opfunR = fun
```

2.11 Recursion

Many unary constructors, even the simplest ones, are of a recursive nature. Proving that recursive unary constructors are `Functor` instances (given appropriate recursive definitions for `fun`) usually proceeds by using structural induction on the specific structure of the constructor. In this section we will show how one can deal with recursion in a general way.

2.11.1 Recursive Morphisms

```
> class (RecCategory mor rec, Functor mor mor f)
>       => Banana mor f rec where
>   cata :: mor (f x) x -> mor (rec f) x
>   ana :: mor x (f x) -> mor x (rec f)

>   cata phi = phi . endofun (cata phi) . osi
>   ana psi = iso . endofun (ana psi) . psi
```

The aim of the `Banana` class is to offer its user the possibility to define a whole range of recursive morphisms for the recursive type `rec f`. The meta-function `cata` is used for morphisms *from* the recursive type `rec f` to any other type `x`. The meta-function `ana` is used for morphisms from a type `x` *to* the recursive type `rec f`. The name `Banana` comes from the Squiggol community. The Squiggol syntax for `cata phi` is the Greek letter ϕ between banana brackets.

The `Banana` class can be used to define recursive morphisms from and to a recursive type (i. e. nullary constructor) `rec f`. If our aim is to define recursive morphisms from and to a type which is constructed by a recursive unary constructor, then another class is needed:

```
> class Category mor => FunctorR mor mor' f where
>   funR :: mor y z -> mor' (f x y) (f x z)

> class (RecRCategory mor recR, FunctorR mor mor f)
       => BananaR mor f recR where
>   cataR :: mor (f x y) y -> mor (recR f x) y
>   anaR :: mor y (f x y) -> mor y (recR f x)

>   cataR phiR = phiR . endofunR (cataR phiR) . osiR
>   anaR psiR = isoR . endofunR (anaR psiR) . psiR
```

2.11.2 Binary Trees Revisited

In this subsection we return to the binary tree example of subsection 1.5.1. The idea is to replace the constructors `Mor`, `Sum`, `Prod` and `RecR` by constructor variables `mor`, `sum`, `prod` and `recR`. Notice that our definition of trees does not use any data definitions at all.

```
> type T sum prod a x = sum a (prod x x) in leafT, nodeT,
>                                           funRT, funLT, revT

> type Tree sum prod = recR (T sum prod)

> leafT :: SumCategory mor sum => mor a (T sum prod a x)
> leafT = inl

> nodeT :: SumCategory mor sum => mor (prod x x) (T sum prod a x)
> nodeT = inr

> leaf :: (SumCategory mor sum, RecRCategory mor recR)
>        => mor a (Tree sum prod a)
> leaf = isoR . leafT

> node :: (SumCategory mor sum, RecRCategory mor recR)
>        => mor (prod (Tree sum prod a) (Tree sum prod a))
>               (Tree sum prod a)
> node = isoR . nodeT
```

This finishes the code for the constructors `leaf` and `node`. Our next goal is to define recursive morphisms on `Tree sum prod a`. Therefore we need a `FunctorR` instance.

```
> funRT :: (SumCategory mor sum, ProdCategory mor prod)
>         => mor x y -> mor (T sum prod a x) (T sum prod a y)
> funRT g = id >+< g >*< g

> instance (SumCategory mor sum, ProdCategory mor prod)
>         => FunctorR (T sum prod) where
>   funR = funRT

> revT :: (SumCategory mor sum, ProdCategory mor prod)
>        => mor (T sum prod a (Tree sum prod a))
>               (Tree sum prod a)
> revT = leaf <+> node . (exr <*> exl)

> rev :: (SumCategory mor sum, ProdCategory mor prod,
>         BananaR mor (T sum prod) recR)
>        => mor (Tree sum prod a) (Tree sum prod a)
> rev = cataR revT
```

The code of the morphisms and meta-functions is exactly the same as the code of the functions of subsection 1.5.1. The types of the morphisms and meta-functions are more general. They are also more verbose: the type qualifiers show the categorical framework which is needed to define them.

2.11.3 Recursive Functors

```
> class Category mor => FunctorL mor mor f where
>   funL :: mor x y -> mor (f x z) (f y z)

> instance (BananaR mor f recR, FunctorL mor mor f)
>         => Functor mor mor (recR f) where
>   fun f = cataR (isoR . endofunL f)
```

In subsection 2.11.1 we have showed that for defining recursive morphisms on `recR f x` the binary constructor `f` has to be a `FunctorR` instance. In this subsection we introduce a `FunctorL` class which is exactly what we need for turning `recR f` into a `Functor` instance. The meta-function `endofunL` is defined in a similar way as the meta-function `endofun` of subsection 2.10 for keeping the `Gofer` type checker happy. Here is the `FunctorL` instance code for trees:

```
> funLT :: (SumCategory mor sum, ProdCategory mor prod)
>         => mor a b -> mor (T sum prod a x) (T sum prod b x)
> funLT g = g >+< id >*< id

> instance (SumCategory mor sum, ProdCategory mor prod)
>         => FunctorL mor mor (T sum prod) where
>   funL = funLT
```

Notice that the definition of `funLT` is very simple. However, combined with `cataR` it becomes a very powerful meta-function

3 Conclusion

We have shown that many concepts from category theory which turn out to be useful for functional programming can be encoded in a natural way in `Gofer` by making use of its constructor classes. Many of the classes and class instances of the `prelude` are used (in a less general setting) successfully for writing modular interpreters (see, for example, [6]). Combining subtypes, catamorphisms and monad transformers makes it possible to write reusable building blocks for modular interpreters in `Gofer`. In subsection 2.10.1 we have presented some functors which were claimed to be useful as building blocks for defining the state and the continuation monad. We have not shown *how* this can be done. The functors can be seen as the left and right functor of an adjunction. For every `Adjunction` instance with opposite underlying categories `OpM mor` and `mor` one can define a composed `Triple` instance:

```
> instance Adjunction (OpM mor) mor l r => Triple mor (CompOp r l)

> type Cnt fun a = CompOp (OpF fun a) (OpF fun a)
```

The `Cnt` composed `Triple` instance can be defined for any `FunCategory` instance. For every `Adjunction` instance with equal underlying categories `mor` one can define a composed `Triple` instance:

```
> instance Adjunction mor mor l r => Triple mor (Comp r l)

> type St fun prod s = Comp (Rd fun s) (OpP prod s)
```

The `St` composed `Triple` instance which can be defined for any `ClosedCategory` instance.

References

[1] D.E. Rydeheard, R.M. Burstall, *Computational Category Theory.* Prentice Hall, 1988.

[2] L. Duponcheel, E. Meijer, *On the expressive power of constructor classes.* Research Report RUU, 1994 (to appear).

[3] P. Hudak, S. Peyton Jones, P.Wadler, *Report on the programming language Haskell, version 1.2.* ACM SIGPLAN Notices, Vol. 27(5), May 1992.

[4] M.P. Jones, *A system of constructor classes: overloading and implicit higher-order polymorphism.* Proceedings of the ACM conference on Functional Programming Languages and Computer Architecture, 1993.

[5] M.P. Jones, L. Duponcheel, *Composing monads.* Research Report YALEU/DCS/RR-1004, 1993.

[6] S. Liang, P. Hudak, M.P. Jones, *Monad transformers and modular interpreters.* Research Report, YALEU/DCS, 1994.

[7] G.L. Steele Jr., *Building interpreters by composing monads.* Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, 1994.

[8] P. Wadler, *The essence of functional programming.* Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, 1992.

[9] P. Wadler, S.P. Jones, *Imperative functional programming.* Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, 1993.