FUNCTIONAL PEARL

Why walk when you can take the tube?

LUCAS DIXON
University of Edinburgh

PETER HANCOCK and CONOR MCBRIDE
University of Nottingham

Abstract

Mornington Crescent

1 Introduction

The purpose of this paper is not only self-citation (McBride, 2001; McBride & Paterson, 2006), but also to write a nice wee program.

2 Traversable Polynomial Functors

We're going to work generically with recursive datatypes representing terms in some syntax. The syntax will be determined by a functor $f :: * \rightarrow *$, representing the choice of expression forms. The parameter is used to indicate the places for subterms within terms, and the resulting type of terms is given by taking the least fixpoint of f, so that the parameter gets instantiated with the very type of terms we are defining.

newtype $\mu\, f = \mathsf{In}\ (f\ (\mu\, f))$

To work with data in this style, we'll need a kit for building functors. Let's start with the *polynomials*, given as follows—in each definition, x represents the type of subterms:

newtype $\mathsf{Id}\quad x = \mathsf{Id}\ x$ -- 'identity' for a subterm
newtype $\mathsf{C}\ c\quad x = \mathsf{C}\ c$ -- 'constant' for non-recursive data of type c
data $(p \boxplus q)\ x = \mathsf{InL}\ (p\ x) \mid \mathsf{InR}\ (q\ x)$ -- 'sum' for choice
data $(p \boxtimes q)\ x = p\ x \boxtimes q\ x$ -- 'product' for pairing

For a simple example, consider the syntax of expressions with numeric constants and addition, directly defined thus:

data Expr = Num Int | Add Expr Expr

We can build this from our kit by describing the basic choice of expression forms—a non-recursive Int or pair of subterms:

$$\textbf{type}\ \mathsf{ExprF} = \mathsf{C\ Int} \boxplus \mathsf{Id} \boxtimes \mathsf{Id}$$

Now, the resulting fixpoint, μ ExprF, is isomorphic to Expr, and we can define the analogues of Expr's constructors.

$$\begin{array}{l} \mathsf{num} :: \mathsf{Int} \to \mu\ \mathsf{ExprF} \\ \mathsf{num}\ i = \mathsf{In}\ (\mathsf{InL}\ (\mathsf{C}\ i)) \\ \mathsf{add} :: \mu\ \mathsf{ExprF} \to \mu\ \mathsf{ExprF} \to \mu\ \mathsf{ExprF} \\ \mathsf{add}\ e_1\ e_2 = \mathsf{In}\ (\mathsf{InR}\ (\mathsf{Id}\ e_1 \boxtimes \mathsf{Id}\ e_2)) \end{array}$$

This method of constructing datatypes as fixpoints of functors is entirely standard: a comprehensive account can be found in Bird and de Moor's *Algebra of Programming* (Bird & de Moor, 1997). The point of casting datatypes in this uniform mould is to capture *patterns* of operations over a class of datatypes, once and for all. Paradigmatically, for each functor f, $\mu\, f$ has an iteration operator, or *catamorphism*, which explains how to compute recursively over whole terms, by applying at each node an *algebra* ϕ to compute the output for a term, given the outputs from its subterms.

$$\begin{array}{l} \mathsf{cata} :: \mathsf{Functor}\ f \Rightarrow (f\ t \to t) \to \mu\, f \to t \\ \mathsf{cata}\ \phi\ (\mathsf{In}\ fm) = \phi\ (\mathsf{fmap}\ (\mathsf{cata}\ \phi)\ fm) \end{array}$$

For example, we can write an evaluator as a catamorphism whose algebra explains how each of the expression forms operates on *values*:

$$\begin{array}{l} \mathsf{eval} :: \mu\ \mathsf{ExprF} \to \mathsf{Int} \\ \mathsf{eval} = \mathsf{cata}\ \phi\ \textbf{where} \\ \quad \phi :: \mathsf{ExprF\ Int} \to \mathsf{Int} \\ \quad \phi\ (\mathsf{In}\ (\mathsf{InL}\ (\mathsf{C}\ i))) \qquad\qquad = i \\ \quad \phi\ (\mathsf{In}\ (\mathsf{InR}\ (\mathsf{Id}\ v_1 \boxtimes \mathsf{Id}\ v_2))) = v_1 + v_2 \end{array}$$

Of course, we must show that the polynomial type constructors are indeed functorial. We do this by writing four instances of the Functor class, showing that Id and C c *are* functors, whilst $\cdot \boxplus \cdot$ and $\cdot \boxtimes \cdot$ *preserve* functoriality. Recall that a Functor in Haskell is just a type constructor f which supports the overloaded 'map' operation, $\mathsf{fmap} :: (s \to t) \to f\ s \to f\ t$.

$$\begin{array}{l} \textbf{instance}\ \mathsf{Functor\ Id}\ \textbf{where} \\ \quad \mathsf{fmap}\ f\ (\mathsf{Id}\ x) = \mathsf{Id}\ (f\ x) \\ \textbf{instance}\ \mathsf{Functor}\ (\mathsf{C}\ c)\ \textbf{where} \\ \quad \mathsf{fmap}\ f\ (\mathsf{C}\ c) = (\mathsf{C}\ c) \\ \textbf{instance}\ (\mathsf{Functor}\ p, \mathsf{Functor}\ q) \Rightarrow \mathsf{Functor}\ (p \boxplus q)\ \textbf{where} \\ \quad \mathsf{fmap}\ f\ (\mathsf{InL}\ px) = \mathsf{InL}\ (\mathsf{fmap}\ f\ px) \\ \quad \mathsf{fmap}\ f\ (\mathsf{InR}\ qx) = \mathsf{InR}\ (\mathsf{fmap}\ f\ qx) \\ \textbf{instance}\ (\mathsf{Functor}\ p, \mathsf{Functor}\ q) \Rightarrow \mathsf{Functor}\ (p \boxtimes q)\ \textbf{where} \\ \quad \mathsf{fmap}\ f\ (px \boxtimes qx) = \mathsf{fmap}\ f\ px \boxtimes \mathsf{fmap}\ f\ qx \end{array}$$

But we can have more than that! In (McBride & Paterson, 2006), McBride and Paterson introduced the notion of a Traversable functor, delivering a version of 'map' which performs an *effectful* computation for each element, combining the effects:

```
class Functor f ⇒ Traversable f where
  traverse :: Applicative a ⇒ (s → a t) → f s → a (f t)
```

Recall that an Applicative functor has the following operations (hence any Monad can be made Applicative):

```
class Functor a ⇒ Applicative a where
  pure :: x → a x                      -- values become effectless a-computations
  (⊛) :: a (s → t) → a s → a t         -- a-application, combining effects
```

Implementing traverse is just like implementing fmap, except that we use pure to lift the constructors and we replace the ordinary application with a-application. We think of working in this lifted way as 'programming in the *idiom* of a'.

```
instance Traversable Id where
  traverse f (Id x) = pure Id ⊛ f x
instance Traversable (C c) where
  traverse f (C c) = pure (C c)
instance (Traversable p, Traversable q) ⇒ Traversable (p ⊞ q) where
  traverse f (InL px) = pure InL ⊛ traverse f px
  traverse f (InR qx) = pure InR ⊛ traverse f qx
instance (Traversable p, Traversable q) ⇒ Traversable (p ⊠ q) where
  traverse f (px ⊠ qx) = pure (⊠) ⊛ traverse f px ⊛ traverse f qx
```

We shall be making particular use of traversability in the Maybe idiom, lifting a *failure-prone* function f on elements to a traversal which succeeds only if f succeeds at every element.

[*lookup* example.] [traverse with Maybe is strict, so doesn't work on streams.]

Remark. Every Traversable functor must be an instance of Functor. We can easily implement fmap by using traverse with a = Id.

[You don't get closure under functoriality from closure under traversability.] [A pair of streams is still functorial.]

2.1 Composition

```
newtype (p ⊡ q) x = Comp (p (q x))
instance (Traversable p, Traversable q) ⇒ Traversable (p ⊡ q) where
  traverse f (Comp xqp) = pure Comp ⊛ traverse (traverse f) xqp
```

3 Free Monads

The *free monad* construction lifts any functorial *signature* p of operations to a *syntax* of expressions constructed from those operations and from free variables x.

```
data Term p x = Con (p (Term p x)) | Var x
```

The return of the Monad embeds free variables into the syntax. The ≫= is exactly the simultaneous substitution operator. Below, f takes variables in x to expressions in Term p y; (≫=f) delivers the corresponding action on expressions in Term p x.

```
instance Functor p ⇒ Monad (Term p) where
  return = Var
  Var x  ≫= f = f x
  Con tp ≫= f = Con (fmap (≫=f) tp)
```

Correspondingly, Term p is also Applicative and a Functor. Moreover, if p is Traversable, then so is Term p.

```
instance Traversable p ⇒ Traversable (Term p) where
  traverse f (Var x)  = pure Var ⊛ f x
  traverse f (Con tp) = pure Con ⊛ traverse (traverse f) tp
```

By way of example, we choose a simple signature with constant integer values and a binary operator[1]. As one might expect, $\cdot \boxplus \cdot$ delivers choice and $\cdot \boxtimes \cdot$ delivers pairing. Meanwhile Id marks the spot for each subexpression.

```
type Sig = C Int ⊞ Id ⊠ Id
```

Now we can implement the constructors we first thought of, just by plugging Con together with the constructors of the polynommial functors in Sig.

[Relate this to the direct presentation of expressions.]

```
val :: Int → Term Sig x
val i = Con (InL (C i))

add :: Term Sig x → Term Sig x → Term Sig x
add x y = Con (InR (Id x ⊠ Id y))
```

4 The ∅ Type

We can recover the idea of a *closed* term by introducing the ∅ type, beloved of logicians but sadly too often spurned by programmers.

```
data ∅
```

Bona fide elements of ∅ are hard to come by, so we may safely offer to exchange them for anything you might care to want: as you will be paying with bogus currency, you cannot object to our shoddy merchandise.

```
naughtE :: ∅ → a
naughtE _ = ⊥
```

More crucially, naughtE lifts functorially. The type f ∅ represents the 'base cases' of f which exist uniformly regardless of f's argument. For example, [ ] :: [∅], Nothing ::

[1] Hutton's Razor strikes again!

Maybe ∅ and C 3 :: Sig ∅. We can map these terms into any $f\ a$, just by turning all the elements of ∅ we encounter into elements of a.

```
inflate :: Functor f ⇒ f ∅ → f a
inflate = unsafeCoerce #    -- fmap naughtE – could be unsafeCoerce
```

Thus equipped, we may take Term p ∅ to give us the *closed* terms over signature p. Modulo the usual fuss about bottoms, Term p ∅ is just the usual recursive datatype given by taking the fixpoint of p. The general purpose 'evaluator' for closed terms is just the usual notion of *catamorphism*.

```
fcata :: (Functor p) ⇒ (p v → v) → Term p ∅ → v
fcata operate (Var nonsense)   = naughtE nonsense
fcata operate (Con expression) = operate (fmap (fcata operate) expression)
```

Following our running example, we may take

```
sigOps :: Sig Int → Int
sigOps (InL (C i)) = i
sigOps (InR (Id x ⊠ Id y)) = x + y
```

and now

```
cata sigOps (add (val 2) (val 2)) = 4
```

We shall also make considerable use of ∅ in a moment, when we start making *holes* in polynomials.

5 Differentiating Polynomials

[Need the usual pictures, and some examples.]

```
class (Traversable p, Traversable p′) ⇒ ∂p ↦ p′ | p → p′ where
  (<·) :: p′ x → x → p x
  down :: p x → p (p′ x, x)
```

```
downright                  fmap snd (down xf)  =  xf
downhome  fmap (uncurry (<·)) (down xf)  =  fmap (const xf) xf
```

```
instance ∂(C c) ↦ C ∅ where
  C z <· _ = naughtE z
  down (C c) = C c
```

```
instance ∂Id ↦ C () where
  C () <· x = Id x
  down (Id x) = Id (C (), x)
```

$$
\begin{array}{l}
\textbf{instance}\ (\partial p \mapsto p', \partial q \mapsto q') \Rightarrow \partial(p \boxplus q) \mapsto p' \boxplus q'\ \textbf{where} \\
\quad \mathsf{InL}\ p' \lessdot x = \mathsf{InL}\ (p' \lessdot x) \\
\quad \mathsf{InR}\ q' \lessdot x = \mathsf{InR}\ (q' \lessdot x) \\
\quad \mathsf{down}\ (\mathsf{InL}\ p) = \mathsf{InL}\ (\mathsf{fmap}\ (\mathsf{InL} \times \mathsf{id})\ (\mathsf{down}\ p)) \\
\quad \mathsf{down}\ (\mathsf{InR}\ q) = \mathsf{InR}\ (\mathsf{fmap}\ (\mathsf{InR} \times \mathsf{id})\ (\mathsf{down}\ q))
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ (\partial p \mapsto p', \partial q \mapsto q') \Rightarrow \partial(p \boxtimes q) \mapsto p' \boxtimes q \boxplus p \boxtimes q'\ \textbf{where} \\
\quad \mathsf{InL}\ (p' \boxtimes q) \lessdot x = (p' \lessdot x) \boxtimes q \\
\quad \mathsf{InR}\ (p \boxtimes q') \lessdot x = p \boxtimes (q' \lessdot x) \\
\quad \mathsf{down}\ (p \boxtimes q) = \\
\quad\quad \mathsf{fmap}\ ((\mathsf{InL} \cdot (\boxtimes q)) \times \mathsf{id})\ (\mathsf{down}\ p) \boxtimes \mathsf{fmap}\ ((\mathsf{InR} \cdot (p \boxtimes)) \times \mathsf{id})\ (\mathsf{down}\ q)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ (\partial p \mapsto p', \partial q \mapsto q') \Rightarrow \partial(p \circledcirc q) \mapsto (p' \circledcirc q) \boxtimes q'\ \textbf{where} \\
\quad (\mathsf{Comp}\ p' \boxtimes q') \lessdot x = \mathsf{Comp}\ (p' \lessdot q' \lessdot x) \\
\quad \mathsf{down}\ (\mathsf{Comp}\ xqp) = \mathsf{Comp}\ (\mathsf{fmap\ outer}\ (\mathsf{down}\ xqp))\ \textbf{where} \\
\quad\quad \mathsf{outer}\ (p', xq) = \mathsf{fmap\ inner}\ (\mathsf{down}\ xq)\ \textbf{where} \\
\quad\quad\quad \mathsf{inner}\ (q', x) = (\mathsf{Comp}\ p' \boxtimes q', x)
\end{array}
$$

6 Differentiating Free Monads

A one-hole context in the syntax of Terms generated by the free monad construction is just a *sequence* of one-hole contexts for subterms in terms, as given by differentiating the signature functor.

$$\textbf{newtype}\ \partial p \mapsto p' \Rightarrow \mathsf{Tube}\ p\ p'\ x = \mathsf{Tube}\ [p'\ (\mathsf{Term}\ p\ x)]$$

Tubes are Traversable Functors. They also inherit a Monoid structure from their underlying representation of sequences. Exactly which sequence structure you should use depends on the operations you need to support. As in (McBride, 2001), we are just using good old [] for pedagogical simplicity. At the time, Ralf Hinze, Johan Jeuring and Andres Löh pointed out (2004), this choice does not yield constant-time *navigation* operations in the style of Huet's 'zippers' (1997), and I am sure they would not forgive us this time if we failed to mention that replacing [] by 'snoc-lists' which grow on the right restores this facility.

Let us give an interface to contexts. We shall need the Monoid structure:

$$
\begin{array}{l}
\textbf{instance}\ \partial p \mapsto p' \Rightarrow \mathsf{Monoid}\ (\mathsf{Tube}\ p\ p'\ x)\ \textbf{where} \\
\quad \varepsilon = \mathsf{Tube}\ [\,] \\
\quad ctxt \oplus \mathsf{Tube}\ [\,] = ctxt \\
\quad \mathsf{Tube}\ ds \oplus \mathsf{Tube}\ ds' = \mathsf{Tube}\ (ds +\!\!+ ds')
\end{array}
$$

We may construct a one-step context for Term p from a one-hole context for subterms in a p.

$$
\begin{array}{l}
\mathsf{step} :: \partial p \mapsto p' \Rightarrow p'\ (\mathsf{Term}\ p\ x) \rightarrow \mathsf{Tube}\ p\ p'\ x \\
\mathsf{step}\ d = \mathsf{Tube}\ [d]
\end{array}
$$

Plugging a Term into a Tube just iterates $\lessdot$ for p.

```
(≪·) :: ∂p ↦ p′ ⇒ Tube p p′ x → Term p x → Term p x
Tube [ ] ≪· t = t
Tube (d : ds) ≪· t = Con (d <· Tube ds ≪· t)
```

Moreover, anyplace you can plug a subterm is certainly a place you can plug a variable, and *vice versa*. We shall also have

```
instance ∂p ↦ p′ ⇒ ∂(Term p) ↦ Tube p p′ where
  ctxt <· x = ctxt ≪· Var x
  down (Var x)   = Var (ε, x)
  down (Con tp) = Con (fmap outer (down tp)) where
    outer (p′, t) = fmap inner (down t) where
      inner (ctxt, x) = (step p′ ⊕ ctxt, x)
```

7 Going Underground

```
data ∂p ↦ p′ ⇒ Underground p p′ x
  = Ground (Term p ∅)
  | Tube p p′ ∅ :≺Node p p′ x
data ∂p ↦ p′ ⇒ Node p p′ x
  = Terminus x
  | Junction (p (Underground p p′ x))
```

```
var :: ∂p ↦ p′ ⇒ x → Underground p p′ x
var x = ε :≺Terminus x
```

```
con :: ∂p ↦ p′ ⇒ p (Underground p p′ x) → Underground p p′ x
con psx = case traverse compressed psx of
  Just pt0 → Ground (Con pt0)
  Nothing → case foldMap tubing (down psx) of
    Just sx → sx
    Nothing → ε :≺Junction psx
  where
    compressed :: ∂p ↦ p′ ⇒ Underground p p′ x → Maybe (Term p ∅)
    compressed (Ground pt0) = Just pt0
    compressed _            = Nothing
    tubing (p′sx, bone :≺node) = case traverse compressed p′sx of
      Just p′t0 → Just (step p′t0 ⊕ bone :≺node)
      Nothing  → Nothing
    tubing _ = Nothing
```

```
underground :: ∂p ↦ p′ ⇒ Underground p p′ x → (x → t) → (p (Underground p p′ x) → t) → t
underground (Ground (Con pt0))          v c = c (fmap Ground pt0)
underground (Tube [ ] :-< Terminus x)   v c = v x
underground (Tube [ ] :-< Junction psx) v c = c psx
underground (Tube (p′t0 : tube) :-< station) v c =
  c (fmap Ground p′t0 <: (Tube tube :-< station))

tunnel :: ∂p ↦ p′ ⇒ Term p x → Underground p p′ x
tunnel (Var x)   = var x
tunnel (Con ptx) = con (fmap tunnel ptx)

untunnel :: ∂p ↦ p′ ⇒ Underground p p′ x → Term p x
untunnel sx = underground sx
  (λ {-var -} x   → Var x)
  (λ {-con -} psx → Con (fmap untunnel psx))

(-<) :: ∂p ↦ p′ ⇒ Tube p p′ ∅ → Underground p p′ x → Underground p p′ x
tube  -< Ground pt0 = Ground (tube <<: pt0)
tube₀ -< tube₁ :-< node = tube₀ ⊕ tube₁ :-< node

instance ∂p ↦ p′ ⇒ Monad (Underground p p′) where
  return = var
  Ground pt0                 >>= σ = Ground pt0
  (tube :-< Junction psx)    >>= σ = tube -< con (fmap (>>=σ) psx)
  (tube :-< Terminus x)      >>= σ = tube -< σ x
```

References

Bird, Richard, & de Moor, Oege. (1997). *Algebra of Programming*. Prentice Hall.

Hinze, Ralf, Jeuring, Johan, & Löh, Andres. (2004). Type-indexed data types. *Science of computer programmming*, **51**, 117–151.

Huet, Gérard. (1997). The Zipper. *Journal of Functional Programming*, **7**(5), 549–554.

McBride, Conor. (2001). *The Derivative of a Regular Type is its Type of One-Hole Contexts*. Available at `http://www.cs.nott.ac.uk/~ctm/diff.pdf`.

McBride, Conor, & Paterson, Ross. (2006). Applicative programming with effects. *Journal of Functional Programming*. to appear.