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 Polynomial Traversable Functors

```
newtype C c x = C c
instance Traversable (C c) where
  traverse f (C c) = pure (C c)

newtype X x = X x
instance Traversable X where
  traverse f (X x) = pure X ⊛ f x

data (p ⊞ q) x = InL (p x) | InR (q x)
instance (Traversable p, Traversable q) ⇒ Traversable (p ⊞ q) where
  traverse f (InL xp) = pure InL ⊛ traverse f xp
  traverse f (InR xq) = pure InR ⊛ traverse f xq

data (p ⊠ q) x = p x ⊠ q x
instance (Traversable p, Traversable q) ⇒ Traversable (p ⊠ q) where
  traverse f (xp ⊠ xq) = pure (⊠) ⊛ traverse f xp ⊛ traverse f xq

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.

$$\textbf{data}\ \mathsf{Term}\ p\ x = \mathsf{Con}\ (p\ (\mathsf{Term}\ p\ x))\ |\ \mathsf{Var}\ x$$

The return of the Monad embeds free variables into the syntax. The $\gg\!=$ is exactly the simultaneous substitution operator. Below, f takes variables in x to expressions in Term p y; $(\gg\!=f)$ 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 X marks the spot for each subexpression.

```
type Sig = C Int ⊞ X ⊠ X
```

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

```
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 (X x ⊠ X 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.

[1] Hutton's Razor strikes again!

$$\mathsf{naughtE} :: \emptyset \rightarrow a$$
$$\mathsf{naughtE}\ \_ = \bot$$

More crucially, naughtE lifts functorially. The type $f\ \emptyset$ represents the 'base cases' of f which exist uniformly regardless of f's argument. For example, $[\,] :: [\emptyset]$, $\mathsf{Nothing} :: \mathsf{Maybe}\ \emptyset$ and $\mathsf{C}\ 3 :: \mathsf{Sig}\ \emptyset$. We can map these terms into any $f\ a$, just by turning all the elements of $\emptyset$ we encounter into elements of a.

$$\mathsf{inflate} :: \mathsf{Functor}\ f \Rightarrow f\ \emptyset \rightarrow f\ a$$
$$\mathsf{inflate} = \mathit{unsafeCoerce}\ \#\quad \text{-- fmap naughtE -- could be } \mathit{unsafeCoerce}$$

Thus equipped, we may take $\mathsf{Term}\ p\ \emptyset$ to give us the *closed* terms over signature p. Modulo the usual fuss about bottoms, $\mathsf{Term}\ p\ \emptyset$ 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*.

$$\mathsf{cata} :: (\mathsf{Functor}\ p) \Rightarrow (p\ v \rightarrow v) \rightarrow \mathsf{Term}\ p\ \emptyset \rightarrow v$$
$$\mathsf{cata}\ \mathit{operate}\ (\mathsf{Var}\ \mathit{nonsense}) \quad = \mathsf{naughtE}\ \mathit{nonsense}$$
$$\mathsf{cata}\ \mathit{operate}\ (\mathsf{Con}\ \mathit{expression}) = \mathit{operate}\ (\mathsf{fmap}\ (\mathsf{cata}\ \mathit{operate})\ \mathit{expression})$$

Following our running example, we may take

$$\mathsf{sigOps} :: \mathsf{Sig}\ \mathsf{Int} \rightarrow \mathsf{Int}$$
$$\mathsf{sigOps}\ (\mathsf{InL}\ (\mathsf{C}\ i)) \qquad = i$$
$$\mathsf{sigOps}\ (\mathsf{InR}\ (\mathsf{X}\ x \boxtimes \mathsf{X}\ y)) = x + y$$

and now

$$\mathsf{cata}\ \mathsf{sigOps}\ (\mathsf{add}\ (\mathsf{val}\ 2)\ (\mathsf{val}\ 2)) = 4$$

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

5 Differentiating Polynomials

$$\textbf{class}\ (\mathsf{Traversable}\ p, \mathsf{Traversable}\ p') \Rightarrow \partial p \mapsto p' \mid p \rightarrow p'\ \textbf{where}$$
$$\quad (\lessdot) :: p'\ x \rightarrow x \rightarrow p\ x$$
$$\quad \mathsf{down} :: p\ x \rightarrow p\ (p'\ x, x)$$

$$\textbf{downright} \qquad \mathsf{fmap}\ \mathsf{snd}\ (\mathsf{down}\ \mathit{xf}) \;=\; \mathit{xf}$$
$$\textbf{downhome}\quad \mathsf{fmap}\ (\mathsf{uncurry}\ (\lessdot))\ (\mathsf{down}\ \mathit{xf}) \;=\; \mathsf{fmap}\ (\mathit{const}\ \mathit{xf})\ \mathit{xf}$$

$$\textbf{instance}\ \partial(\mathsf{C}\ c) \mapsto \mathsf{C}\ \emptyset\ \textbf{where}$$
$$\quad \mathsf{C}\ z \lessdot \_ = \mathsf{naughtE}\ z$$
$$\quad \mathsf{down}\ (\mathsf{C}\ c) = \mathsf{C}\ c$$

$$\textbf{instance}\ \partial\mathsf{X} \mapsto \mathsf{C}\ ()\ \textbf{where}$$
$$\quad \mathsf{C}\ () \lessdot x = \mathsf{X}\ x$$
$$\quad \mathsf{down}\ (\mathsf{X}\ x) = \mathsf{X}\ (\mathsf{C}\ (), x)$$

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

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

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

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:

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

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

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

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 crush 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

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.