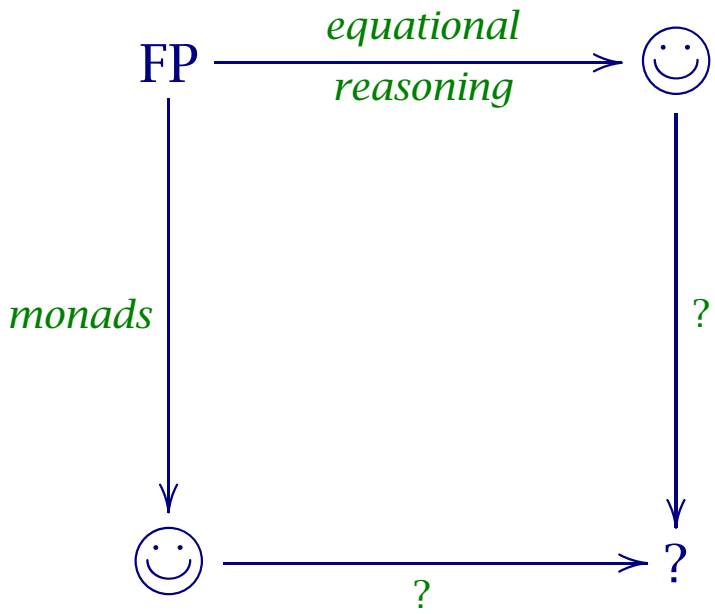
# The Art of Effectful Reasoning

*Jeremy Gibbons*
*University of Oxford*
*WG2.1#66, September 2010*

# 1. Reasoning with effects?



## 1.1. Seeing the wood through the trees

At TFP 2008, Hutton & Fulger discuss the ‘correctness’ of

$$\mathit{relabel} :: \mathit{Tree}\ a \to \mathit{Tree}\ \mathit{Int}$$

I think they miss two opportunities for abstraction:

- from the specific *effects* (they expand the *State* monad to state-transforming functions), and
- from the *pattern of computation* (they use explicit induction on trees).

Can we do better?

# 2. Idioms

'Strong lax-monoidal functors'—or, more perspicuously,

$$
\begin{aligned}
&\textbf{class}\ \mathit{Functor}\ m \Rightarrow \mathit{Idiom}\ m\ \textbf{where}\\
&\quad \mathit{pure} :: a \to m\ a\\
&\quad (\circledast) \quad :: m\ (a \to b) \to m\ a \to m\ b
\end{aligned}
$$

satisfying four laws:

$$
\begin{aligned}
\mathit{pure}\ \mathit{id} \circledast u &= u && \text{-- identity}\\
\mathit{pure}\ (\circ) \circledast u \circledast v \circledast w &= u \circledast (v \circledast w) && \text{-- composition}\\
\mathit{pure}\ f \circledast \mathit{pure}\ x &= \mathit{pure}\ (f\ x) && \text{-- homomorphism}\\
u \circledast \mathit{pure}\ x &= \mathit{pure}\ (\lambda f \to f\ x) \circledast u && \text{-- interchange}
\end{aligned}
$$

## 2.1. Monadic and monoidal idioms

The *State* monad is an idiom (as indeed is every monad):

```
instance Idiom (State s) where
  pure a     = return a
  mf ⊛ mx = do { f ← mf; x ← mx; return (f x) }
```

The constantly-lists functor is an idiom (as indeed for any monoid):

```
newtype KL a b = K{ unK :: [ a ] }

instance Functor (KL a) where
  fmap f (K a) = K a

instance Idiom (KL a) where
  pure a          = K [ ]
  K mf ⊛ K mx = K (mf ++ mx)
```

## 2.2. Composition of idioms

Idioms compose nicely, in exactly the way that monads don't:

```
data (m·n) a = C{ unC :: m (n a) }
instance (Functor m, Functor n) ⇒ Functor (m·n) where
  fmap f (C x) = C (fmap (fmap f) x)
instance (Idiom m, Idiom n) ⇒ Idiom (m·n) where
  pure a            = C (pure (pure a))
  C mnf ⊛ C mnx = C (pure (⊛) ⊛ mnf ⊛ mnx)
```

So we have a composition operator for idiomatic computations:

```
(⊙) :: (Idiom m, Idiom n) ⇒ (b → n c) → (a → m b) → a → (m·n) c
f ⊙ g = C ∘ fmap f ∘ g
```

## 2.3. Identity idiom

Of course, there's an identity idiom too:

```
newtype Id a = I{ unI :: a}
instance Functor Id where
  fmap f (I x) = I (f x)
instance Idiom Id where
  pure a   = I a
  I f ⊛ I x = I (f x)
```

# 3. Traversal

Traversable datatypes support *traverse* and *dist*, which are interdefinable.

```
class Functor t ⇒ Traversable t where
  traverse :: Idiom m ⇒ (a → m b) → t a → m (t b)
  traverse f = dist ∘ fmap f
  dist :: Idiom m ⇒ t (m a) → m (t a)
  dist = traverse id
```

## 3.1. Traversal respects composition of idioms

Traversal ought to respect the compositional structure of idioms:

$$traverse_{Id}\ (I \circ f) = I \circ fmap\ f$$

and (for $f :: b \to n\ c$ and $g :: a \to m\ b$)

$$traverse_{m \cdot n}\ (C \circ fmap\ f \circ g) = C \circ fmap\ (traverse_n\ f) \circ traverse_m\ g$$

the latter being equivalent to

$$traverse_{m \cdot n}\ (f \odot g) = traverse_n\ f \odot traverse_m\ g$$

## 3.2. Traversal is natural in the idiom

Idiom morphism from idiom $m$ to idiom $n$ is natural transformation $\phi : m \dot{\to} n$ such that

$$\begin{aligned} \phi\ (pure_m\ a) &= pure_n\ a \\ \phi\ (mf \circledast_m mx) &= \phi\ mf \circledast_n \phi\ mx \end{aligned}$$

Traversal ought also to be natural in the idiom: for an idiom morphism $\phi : m \dot{\to} n$, we expect the laws

$$\begin{aligned} \phi \circ dist_m &= dist_n \circ fmap\ \phi \\ \phi \circ traverse_m\ f &= traverse_n\ (\phi \circ f) \end{aligned}$$

to hold (each of which implies the other).

## 3.3. Two consequences

In particular, $pure_m \circ unI : Id \dot{\to} m$ is an idiom morphism, so

$$traverse_m\ pure_m = pure_m$$

Moreover, when $m$ is the idiom arising from a *commutative* monad, then $join \circ unC : m \cdot m \dot{\to} m$ is also an idiom morphism, so

$$traverse_m\ f \bullet traverse_m\ g = traverse_m\ (f \bullet g)$$

where '$\bullet$' is Kleisli composition, $join :: m\ (m\ a) \to m\ a$ is monad multiplication, and $f :: b \to m\ c$ and $g :: a \to m\ b$ both use the same monad.

# 4. Tree labelling, idiomatically

```
data Tree a = Tip a | Bin (Tree a) (Tree a)
```

Trees are traversable:

```
instance Functor Tree where
  fmap f (Tip a)   = Tip (f a)
  fmap f (Bin t u) = Bin (fmap f t) (fmap f u)
instance Traversable Tree where
  traverse f (Tip a)   = pure Tip ⊛ f a
  traverse f (Bin t u) = pure Bin ⊛ traverse f t ⊛ traverse f u
```

## 4.1. Streams

$$
\begin{aligned}
&\mathit{adorn} :: a \to \mathit{State}\ [b]\ (a, b) \\
&\mathit{adorn}\ x = \mathbf{do}\ \{\, y : \mathit{ys} \leftarrow \mathit{get}; \mathit{put}\ \mathit{ys}; \mathit{return}\ (x, y)\,\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{label} :: \mathit{Tree}\ a \to \mathit{State}\ [b]\ (\mathit{Tree}\ (a, b)) \\
&\mathit{label} = \mathit{traverse}\ \mathit{adorn}
\end{aligned}
$$

This annotates rather than overwriting, so that we can talk about inversion.

And it is polymorphic in the annotations; so our essential property is that labels are drawn without repetition from the given stream.

The latter follows if the labels chosen form an initial segment of the stream; or equivalently, if returning them to the final stream reconstructs the initial stream.

## 4.2. Unlabelling

$$
\begin{aligned}
&\mathit{strip} :: (a, b) \to \mathit{State}\ [b]\ a \\
&\mathit{strip}\ (x, y) = \mathbf{do}\ \{\mathit{ys} \leftarrow \mathit{get}; \mathit{put}\ y : \mathit{ys}; \mathit{return}\ x\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unlabel} :: \mathit{Tree}\ (a, b) \to \mathit{State}\ [b]\ (\mathit{Tree}\ a) \\
&\mathit{unlabel} = \mathit{traverse}\ \mathit{strip}
\end{aligned}
$$

Now one might expect that *label*—which adorns a tree with labels from a stream—and *unlabel*—which strips those labels off again, and returns them to the stream—would be each other's inverses in some sense.

## 4.3. Idiomatic composition of tree traversals

In particular, suppose that $m, n$ are idioms, that $f :: b \to n\ a$ and $g :: a \to m\ b$, and that $f \odot g = pure_{m \cdot n}$. Then

$$
\begin{aligned}
& traverse_n\ f \odot traverse_m\ g \\
= \quad & [\![\ \text{traversal respects composition}\ ]\!] \\
& traverse_{m \cdot n}\ (f \odot g) \\
= \quad & [\![\ \text{assuming } f \odot g = pure\ ]\!] \\
& traverse_{m \cdot n}\ (pure_{m \cdot n}) \\
= \quad & [\![\ \text{purity}\ ]\!] \\
& pure_{m \cdot n}
\end{aligned}
$$

So, can the assumption

$$strip \odot adorn = pure$$

be discharged? Sadly, no!

## 4.4. Monadic composition of tree traversals

*label* and *unlabel* use the same monadic idiom, *State*, so we can use Kleisli composition. And indeed,

$$\mathit{strip} \bullet \mathit{adorn} = \mathit{pure}$$

Sadly, the *State* monad is the epitome of noncommutativity, so the fusion rule on Slide 3.3 does not apply.

(Incidentally, our conjecture (Gibbons & Oliveira, 2008) that

$$\mathit{traverse}\ f \bullet \mathit{traverse}\ g = \mathit{traverse}\ (f \bullet g)$$

holds even for a non-commutative monad, provided that $f$ and $g$ themselves commute, is false.)

# 5. Backwards idioms (with RSB)

Unlike monads, idioms flip:

$$
\begin{aligned}
\mathit{pure}_{\tilde{m}}\ a &= \mathit{pure}_m\ a \\
\mathit{mf} \circledast_{\tilde{m}} \mathit{mx} &= \mathit{pure}_m\ (\mathit{flip}\ (\$))\ \circledast_m \mathit{mx} \circledast_m \mathit{mf}
\end{aligned}
$$

Define a simple wrapper type for converting an idiom to run backwards:

```
newtype Back m a = B{ unB :: m a }

instance Functor m ⇒ Functor (Back m) where
  fmap f (B x) = B (fmap f x)

instance Idiom m ⇒ Idiom (Back m) where
  pure a          = B (pure a)
  B mf ⊛ B mx = B (pure (flip ($)) ⊛ mx ⊛ mf)
```

Then $B : \tilde{m} \dot{\to} \mathit{Back}\ m$ is another idiom morphism.

## 5.1. Backwards traversal

A consequence of $B$ being an idiom morphism is that it respects traversal:

$$B \circ traverse_{\tilde{m}}\ f = traverse_{Back\ m}\ (B \circ f)$$

or equivalently:

$$traverse_{\tilde{m}}\ f = unB \circ traverse_{Back\ m}\ (B \circ f)$$

Therefore, let us define

$$\begin{aligned} &revert :: (Idiom\ m, Traversable\ t) \Rightarrow (a \to m\ b) \to t\ a \to m\ (t\ b) \\ &revert\ f = unB \circ traverse\ (B \circ f) \end{aligned}$$

so that

$$revert_m\ f = traverse_{\tilde{m}}\ f$$

## 5.2. Unlabelling backwards

Of course: socks and shoes!

When $n = \tilde{m}$, we have $traverse_n = revert_m$, and when $f :: b \to \tilde{m}\ c$ and $g :: a \to m\ b$, if $f \odot g = pure_{m \cdot \tilde{m}}$ the composition of traversals reduces to:

$$revert_m\ f \odot traverse_m\ g = pure_{m \cdot \tilde{m}}$$

We should use the backwards idiom for strip:

$$strip :: (a, b) \to \widetilde{State}\ [b]\ a$$

But still,

$$strip \odot adorn \not\equiv pure$$

## 5.3. Kleisli composition, backwards

For $f :: b \to \tilde{m}\ c$ and $g :: a \to m\ c$ such that

$$f \bullet g = return_m = return_{\tilde{m}}$$

I conjecture (another 'ought'?) that

$$revert_m\ f \bullet traverse_m\ g = traverse_m\ return = return_m$$

It's not difficult to show for pairs, which are in some sense 'the only interesting case'.

But I don't know how to show it in general.
(In particular, I don't see that it follows from naturality.)

# 6. Naturality in the data structure (due to OR)

Naturality in the data structure would be: for any $\phi : t \dot{\to} u$,

$$\begin{tikzcd}
t\ a \arrow[r, "traverse^t\ f"] \arrow[d, "\phi"'] & m\ (t\ b) \arrow[d, "m\ \phi"] \\
u\ a \arrow[r, "traverse^u\ f"] & m\ (u\ b)
\end{tikzcd}$$

That's clearly too strong.

$traverse^t$ determines an element ordering, which $\phi$ need not respect; besides, $\phi$ may drop or duplicate elements.

## 6.1. Coherence with contents

Morally,

$$\begin{tikzcd}
t\ a \arrow[r, "traverse^t\ f"] \arrow[d, "contents"] & m\ (t\ b) \arrow[d, "m\ contents"] \\
[a] \arrow[r, "traverse^{[\,]}\ f"] & m\ [\,b\,]
\end{tikzcd}$$

where $contents = traverse\ (\lambda a \to K\ [\,a\,])$.

'The essence of traversal is the ordering it imposes on elements.'

In fact, I reckon that the previous square commutes for any $\phi$ with

$$contents \circ \phi = contents$$

## 6.2. Coherent relabelling

If we can accept coherence, correctness of relabelling on an arbitrary data structure reduces to correctness on lists.

$$
\begin{aligned}
&\mathit{traverse}^{[\,]}\ \mathit{adorn}\ \mathit{xs}\ \mathit{bs} = (\mathit{zip}\ \mathit{xs}\ (\mathit{take}\ l\ \mathit{bs}), \mathit{drop}\ l\ \mathit{bs}) \\
&\quad \textbf{where}\ l = \#\mathit{xs}
\end{aligned}
$$

or, in terms of replacing and naturals,

$$
\begin{aligned}
&\mathit{traverse}^{[\,]}\ (\mathit{const}\ \mathit{fresh})\ \mathit{xs}\ n = (\mathit{zip}\ \mathit{xs}\ [\,n\,..\,n+l-1\,], n+l) \\
&\quad \textbf{where}\ \mathit{fresh}\ n = (n, n+1) \\
&\qquad\qquad\ \ l = \#\mathit{xs}
\end{aligned}
$$

This is straightforward to prove, since both sides are folds.