Give me freedom!

Or let me forget

Joseph Tel Abrahamson / @sdbo / `github.com/tel`

May 21, 2015

Synopsis

- Ways of seeing Freedom: a noun, an adjective, a verb
- Thinking through Freedom as a process
- Using Category Theory as a tool of insight

A first glimpse of Freedom

Free is a noun

Free is a noun

```
data Free f a
  = Return a
  | Free (f (Free f a))
```

What are Free Monads anyway?

What are Free Monads anyway?

```haskell
newtype Identity a = Identity a
newtype Fix f = Fix (f (Fix f))

-- Free f a ~= Identity a + Fix f
```

What are Free Monads anyway?

```haskell
newtype Identity a = Identity a
newtype Fix f = Fix (f (Fix f))

-- Free f a ~= Identity a + Fix f

-- I don't know, something like that, not quite kind of?
```

What's going on here?

```
lift     :: Functor f ⇒ f a → Free f a
foldFree :: Monad m ⇒ (∀ x . f x → m x) → (Free f a → m a)
```

Free Monads as ”interpreters”

```haskell
data TeletypeF a
  = PutStrLn String a
  | GetLine (String → a)
    deriving ( Functor )

type Teletype = Free TeletypeF

putStrLnTT :: String → Teletype ()
putStrLnTT line = lift (PutStrLn line ())

getLineTT :: Teletype String
getLineTT = lift (GetLine id)
```

Very nice embedded DSLs… for *less*!

```
echoTT :: Teletype ()
echoTT = forever $ do
  line ← getLineTT
  putStrLineTT line
```

Very nice embedded DSLs... for *less*!

```
interp :: TeletypeF a → IO a
interp x = case x of
  PutStrLn line a → putStrLn line  »  return a
  GetLine next → do
    line ← getLine
    return (next line)

echoIO :: IO ()
echoIO = fold interp echoTT
```

“Less” $\neq$ Free

What's going on here?

Free is an adjective

”Free” things!

- `Free` makes free **`Monad`**s!

”Free” things!

- `Free` makes free **`Monad`**s!
- Free `Monoid`s are lists?

Free Monoids are lists

```
pure    :: a → [a]
foldMap :: Monoid m ⇒ (a → m) → ([a] → m)
```

Free `Monoid`s are lists

```
pure    :: a → [a]
foldMap :: Monoid m ⇒ (a → m) → ([a] → m)
```

"`Foldable` just means `toList`"

Free Monoids are lists

```
lift     :: Functor f ⇒ f a → Free f a
foldFree :: Monad m ⇒ (∀ x . f x → m x) → (Free f a → m a)
```

”Free” things!

- `Free` makes free **`Monad`**s!

”Free” things!

- `Free` makes free **`Monad`**s!
- Free `Monoid`s are lists!

”Free” things!

- `Free` makes free `Monad`s!
- Free `Monoid`s are lists!
- We can make free `Applicative`s, I hear

”Free” things!

- `Free` makes free `Monad`s!
- Free `Monoid`s are lists!
- We can make free `Applicative`s, I hear
- Can there be free things of any kind? Sure looks like it!

”Free” things!

- `Free` makes free `Monad`s!
- Free `Monoid`s are lists!
- We can make free `Applicative`s, I hear
- Can there be free things of any kind? Sure looks like it!
- Let’s free all the things!

- Lists are the “largest” `Monoid`s
- Lists are the “simplest” `Monoid`s

- Lists are the “largest” `Monoid`s
- Lists are the “simplest” `Monoid`s
- What are the largest and simplest examples of other things?

More questions than answers

- `Free f` is the “largest” **Monad**?

More questions than answers

- `Free f` is the “largest” **`Monad`**?
- Does that mean that both `Free TeletypeF` and `Free []` are *both* the “largest” **`Monad`**?

More questions than answers

- `Free f` is the “largest” **`Monad`**?
- Does that mean that both `Free TeletypeF` and `Free []` are *both* the “largest” **`Monad`**?
- I hear that `Free f` and `Operational f` are *both* free monads?

More questions than answers

- `Free f` is the “largest” **`Monad`**?
- Does that mean that both `Free TeletypeF` and `Free []` are *both* the “largest” **`Monad`**?
- I hear that `Free f` and `Operational f` are *both* free monads? But they’re not isomorphic.

Wat.

What's going on here?

Freedom is a process

What is `Free`, really?

```
> :kind Free
```

What is `Free`, really?

$>$: `kind Free`

`Free` :: $(\star \to \star) \qquad \to (\star \to \star)$

What is `Free`, really?

But really more like...

$$> : \texttt{kind Free}$$

$$\texttt{Free} :: (\star \to \star)_{\texttt{Functor}} \to (\star \to \star)_{\texttt{Monad}}$$

What is `Free`, really?

But really more like...

$$> : \texttt{kind Free}$$

$$\texttt{Free} :: (\star \to \star)_{\texttt{Functor}} \to (\star \to \star)_{\texttt{Monad}}$$

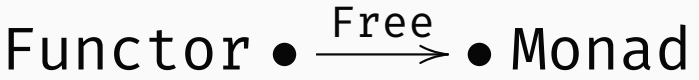
```
-- remember...
instance Functor f ⇒ Monad (Free f)
```

Oh, an *arrow*!

Oh, an *arrow*! Time to use some category theory!

A picture of “Free monads”

$\texttt{Free}_{\texttt{Monad}}$



A picture of “Free monads”

$\texttt{Free}_{\texttt{Monad}}$

$$\texttt{Functor} \bullet \xrightarrow{\texttt{Free}} \bullet \texttt{Monad}$$

$\texttt{List}$

$$\texttt{Hask} \bullet \xrightarrow{\texttt{Free}} \bullet \texttt{Monoid}$$

A picture of “Free monads”

$\texttt{Free}_{\texttt{Monad}}$

$$\texttt{Functor} \bullet \xrightarrow{\texttt{Free}} \bullet \texttt{Monad}$$

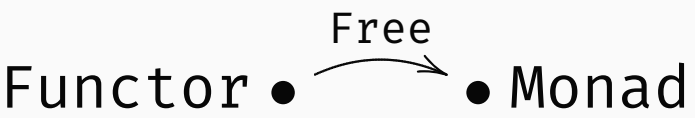
$\texttt{List}$

$$\texttt{Hask} \bullet \xrightarrow{\texttt{Free}} \bullet \texttt{Monoid}$$

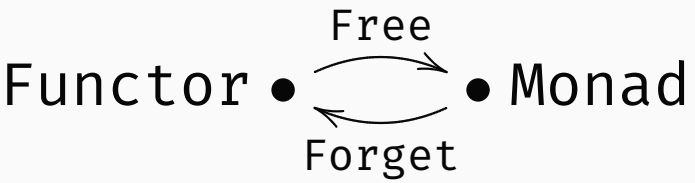
$\texttt{Coyoneda}$

$$\texttt{Hask}_{(\star \to \star)} \bullet \xrightarrow{\texttt{Free}} \bullet \texttt{Functor}$$

Dualize!



Dualize!



- If `Forget :: Monad` $\rightarrow$ `Functor` *forgets* that some type is a **Monad**...

- If `Forget :: Monad → Functor` *forgets* that some type is a **Monad**...
- Is `Free :: Functor → Monad` remembering it?

Not quite

$$(\texttt{Free} \circ \texttt{Forget})(M) \neq M$$

$$(\texttt{Forget} \circ \texttt{Free})(F) \neq F$$

Not quite

$$(\texttt{Free} \circ \texttt{Forget})(M) \neq M$$

$$(\texttt{Forget} \circ \texttt{Free})(F) \neq F$$

For good reason!

Just right

If

$$M = \mathtt{Free}(F)$$

for some $\mathtt{Functor}$ F, then

$$(\mathtt{Free} \circ \mathtt{Forget})(M) = M$$

Just right

If

$$F = \mathtt{Forget}(M)$$

for some **Monad** M, then

$$(\mathtt{Forget} \circ \mathtt{Free})(F) = F$$

Adjunctions

$$Free \dashv Forget$$

$$Free \circ Forget \circ Free = Free$$

$$Forget \circ Free \circ Forget = Forget$$

What does an Adjunction buy us?

$$F : \mathcal{C} \to \mathcal{D}$$

$$G : \mathcal{D} \to \mathcal{C}$$

What does an Adjunction buy us?

$$F : \mathcal{C} \to \mathcal{D}$$

$$G : \mathcal{D} \to \mathcal{C}$$

$$\forall c : \mathcal{C}, d : \mathcal{D}, \mathcal{D}(Fc, d) \equiv \mathcal{D}(c, Gd)$$

What does an Adjunction buy us?

```
type Forget f a = f a

-- "Natural transformations"
type f :→ g = ∀ x . f x → g x

fwd :: Monad m ⇒ (Free f :→ m) → (f :→ Forget m)
bwd :: Monad m ⇒ (f :→ Forget m) → (Free f :→ m)
```

What does an Adjunction buy us?

```
fwd :: Monad m ⇒ (∀ x . Free f x → m x) → (f a → m a)
bwd :: Monad m ⇒ (∀ x . f x → m x) → (Free f a → m a)
```

```
foldFree :: Monad m ⇒ ( ∀ x . f x → m x) → (Free f a → m a)
foldFree = bwd
```

```
idFree :: Free f x → Free f x
idFree = id

-- m ~ Free f
lift :: f a → Free f a
lift = fwd idFre
```

What does an Adjunction buy us?

Everything we need.

Bonus: Freedom for everyone

Definition by elimination

Definition by elimination

```
curious :: _
curious = flip bwd
```

Definition by elimination

```
curious :: Monad m ⇒ Free f a → (f :→ m) → m a
curious = flip bwd
```

Definition by elimination

```
newtype Free f a
  = Free { runFree :: ∀ m . Monad m ⇒ (f :→ m) → m a }
```

Definition by elimination

```
{-# LANGUAGE ConstraintKinds #-}

newtype Free c f a
  = Free { runFree :: ∀ m . c m ⇒ (f :→ m) → m a }
```

Definition by elimination

```
{-# LANGUAGE ConstraintKinds #-}

newtype HFree c f a
  = HFree { runHFree :: ∀ m . c m ⇒ (f :→ m) → m a }
```

Definition by elimination

```haskell
{-# LANGUAGE ConstraintKinds #-}

newtype Free c a
  = Free { runFree :: ∀ r . c r ⇒ (a → r) → r }
```

Definition by elimination

```
free :: [a] → Free Monoid a
free as = Free $ \ar → foldMap ar as

unfree :: Free Monoid a → [a]
unfree f = runFree f (\x → [x])
```

Thanks!

Tweet at me!

@sdbo