THE FREE AND THE FURIOUS

@raichoo

What are OCamlers' critiques of Haskell?

Yoneda-crazy: I know Haskell, I know some category theory, but I am highly sceptical that teaching the Yoneda Lemma to C++ programmers is actually useful in any way. […] I'm worried that we may have a backslash at some point when, you know, people realize that unless your initials are E.K. you are ***wasting your time thinking about the co-density transformation***. […]

Appetizer

λ» runPure revEcho input

[…]

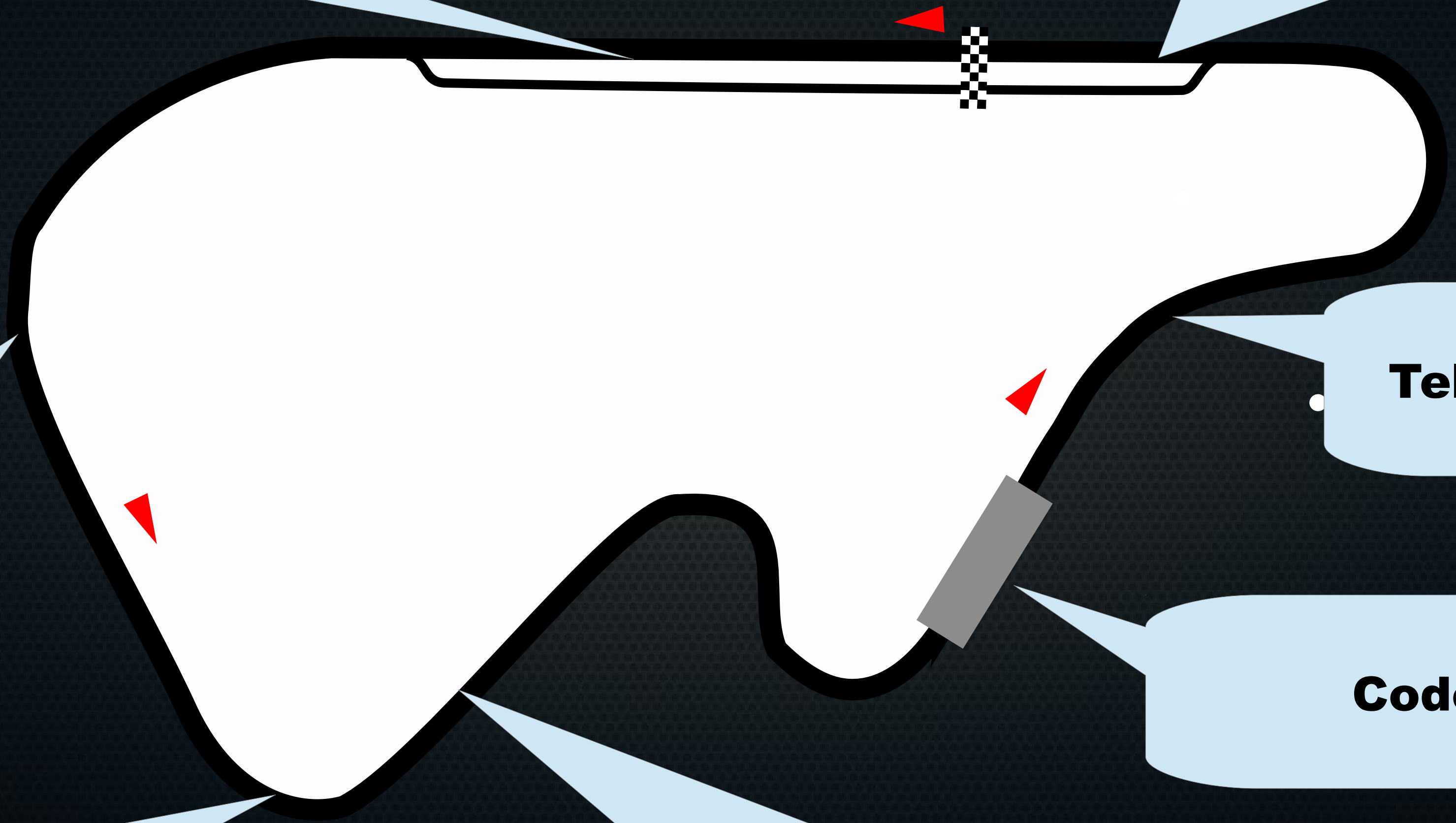
(71.49 secs, 38,538,732,696 bytes)

λ» runPure (improve revEcho) input

[…]

(0.72 secs, 233,224,296 bytes)

OUTLINE


Lists and Monoids
Improving Teletype performance
DLists
Teletype with Free
Codensity
Recap: Free Monads
Recovering Lists with Free

A MOTIVATING EXAMPLE

```haskell
type Log = [String]

prog :: String
prog = last (prog' [] 20000)
  where
    prog' :: Log -> Int -> Log
    prog' log 0 = log
    prog' log n = prog' (log ++ [show n]) (n - 1)
```

MONOIDS

```
class Monoid a where
  mempty :: a
  (<>)   :: a -> a -> a

-- xs <> mempty ≡ xs ≡ mempty <> xs
-- (xs <> ys) <> zs ≡ xs <> (ys <> zs)

instance Monoid [] where
  mempty = []
  (<>)   = (++)
```

DLIST

```haskell
newtype DList a = DList
    { runDList :: [a] -> [a] }

empty :: DList a
empty = DList id

singleton :: a -> DList a
singleton x = DList (x:)

append :: DList a -> DList a -> DList a
append (DList xs) (DList ys) = DList (xs . ys)
```

DLIST (CONT.)

```haskell
toList :: DList a -> [a]
toList (DList xs) = xs []

instance Monoid (DList a) where
  mempty = empty
  (<>)   = append
```

USING DLIST

```haskell
{-# LANGUAGE OverloadedLists #-}
import Data.Monoid ((<>))

import qualified Data.DList as DL

type Log = DL.DList String

prog :: String
prog = last (DL.toList (prog' [] 20000))
  where
    prog' :: Log -> Int -> Log
    prog' log 0 = log
    prog' log n = prog' (log <> [show n]) (n - 1)
```

RECAP: FREE MONAD

```haskell
data Free f a = Free (f (Free f a))
              | Pure a

instance Functor f => Monad (Free f) where
  return = pure

  Pure x >>= f = f x
  Free x >>= f = Free (fmap (>>= f) x)

liftF :: Functor f => f a -> Free f a
liftF = Free . fmap Pure
```

RECOVERING LIST

```haskell
type List a = Free ((,) a) ()

run :: Free ((,) a) () -> [a]
run (Pure _)       = []
run (Free (x, xs)) = x : run xs

empty :: List a
empty = pure ()

singleton :: a -> List a
singleton x = liftF (x, ())

append :: List a -> List a -> List a
append = (>>)
```

RECOVERING LIST

```
type Log = Free ((,) String) ()

prog :: String
prog = last . run $ prog' empty 20000
  where
    prog' :: Log -> Int -> Log
    prog' log 0 = log
    prog' log n =
      prog' (log `append` singleton (show n)) (n - 1)
```

SUBSTITUTE / NORMALIZE

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
m >>= f = join (fmap f m)

-- join :: Monad m => m (m a) -> m a
-- join (Pure x)  = x
-- join (Free xs) = Free (fmap join xs)
```

MONAD LAWS

```
return a >>= f ≡ f a

m >>= return ≡ m

(m >>= f) >>= g ≡ m >>= (\x -> f x >>= g)
```

ENTER CODENSITY

```haskell
{-# LANGUAGE RankNTypes #-}

newtype Codensity m a = Codensity
  { runCodensity :: forall b. (a -> m b) -> m b }

lowerCodensity :: Monad m => Codensity m a -> m a
lowerCodensity (Codensity c) = c return
```

IT'S A MONAD!

```
instance Functor (Codensity f) where
  fmap f (Codensity c) = Codensity (\k -> c (k . f))

instance Applicative (Codensity f) where
  pure x = Codensity (\k -> k x)

  Codensity f <*> Codensity x = Codensity (\k -> f (x . (k .)))

instance Monad (Codensity f) where
  Codensity x >>= f =
    Codensity (\k -> x (flip runCodensity k . f))
```

GENERALIZING THE MACHINERY

```
class (Functor f, Monad m) => MonadFree f m | m -> f where
  wrap :: f (m a) -> m a

instance Functor f => MonadFree f (Free f) where
  wrap = Free

instance MonadFree f m => MonadFree f (Codensity m) where
  wrap x =
    Codensity (\k -> wrap (fmap (flip runCodensity k) x))

liftF :: MonadFree f m => f a -> m a
liftF = wrap . fmap pure
```

RECOVERING DLIST

```
type List a = Codensity (Free ((,) a)) ()

run :: Free ((,) a) () -> [a]
run (Pure _)       = []
run (Free (x, xs)) = x : run xs

empty :: List a
empty = pure ()

singleton :: a -> List a
singleton x = liftF (x, ())

append :: List a -> List a -> List a
append = (>>)
```

RECOVERING DLIST

```
type Log = Codensity (Free ((,) a)) ()

prog :: String
prog = last . run . lowerCodensity $ prog' empty 20000
  where
    prog' :: Log -> Int -> Log
    prog' log 0 = log
    prog' log n =
      prog' (log `append` singleton (show n)) (n - 1)
```

TELETYPE

```haskell
{-# LANGUAGE DeriveFunctor #-}

data TeletypeF k = PutChar Char k
                 | GetChar (Char -> k)
                 deriving Functor

type Teletype a = Free TeletypeF a
```

TELETYPE

```haskell
{-# LANGUAGE DeriveFunctor #-}

data TeletypeF k = PutChar Char k
                 | GetChar (Char -> k)
                 deriving Functor

type Teletype a = forall m. MonadFree TeletypeF m => m a
```

TELETYPE

```haskell
import Control.Monad (when)

getChar :: Teletype Char
getChar = liftF (GetChar id)

putChar :: Char -> Teletype ()
putChar c = liftF (PutChar c ())

revEcho :: Teletype ()
revEcho = do
  c <- getChar
  when (c /= ' ') $ do
    revEcho
    putChar c
```

MONAD IMPROVEMENT

```
improve :: Functor f
        => (forall m. MonadFree f m => m a)
        -> Free f a
improve = lowerCodensity
```

FURTHER READING

- Hackage: *dlist, free, kan-extensions*
- Idris-hackers: *idris-free*
- Janis Voigtländer: *Asymptotic Improvement of Computations over Free Monads*
- Edward Kmett: *Monads for Less*

THANK YOU!

QUESTIONS?