Monads in Haskell

An Introduction

With drawings!

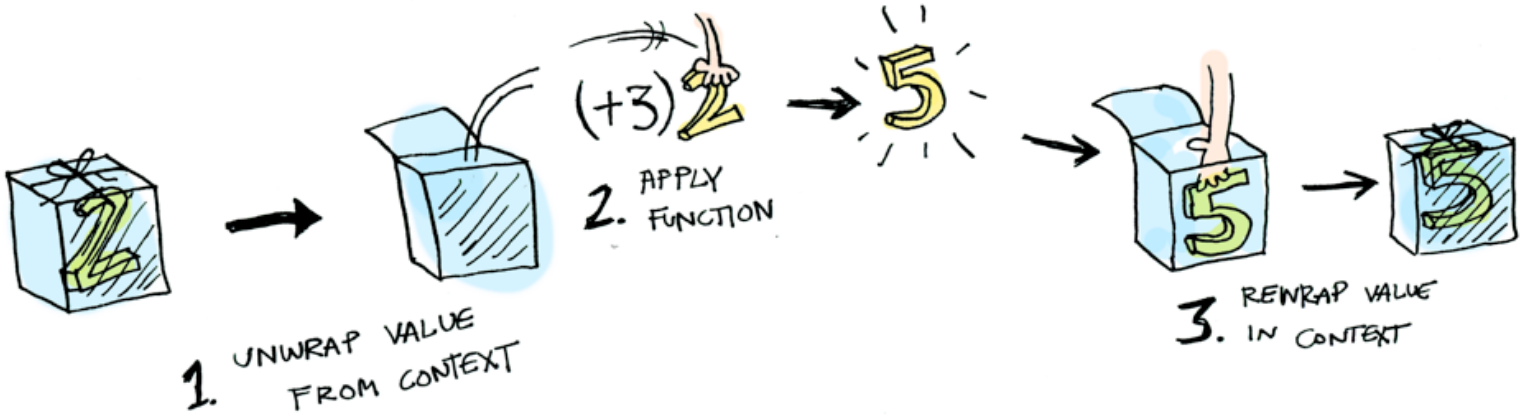

image: adit.io

Agenda

- Some Haskell basics
- The type system and Typeclasses
- Functors
- Monads
- The Real World

Motivation

- Haskell (and most functional languages) have:
  - No side effects
  - No concept of global state
- How do we do things that inherently have side effects or need state?

Some Haskell basics

- *Functions* are the most important construct
- Function application is just the name of the function followed by parameters:

```
ghci> head [2,4,5]
2
ghci> (\x -> x * x) 2
4
ghci> let square x = x * x
```

Types

```
square:: Int -> Int
square = x * x

head’:: [a] -> a
head’ (x:xs) = x

take:: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ []         = []
take n (x:xs) = x : take (n-1) xs
```

Created and Parametric Types

```
data Maybe a = Just a | Nothing
```

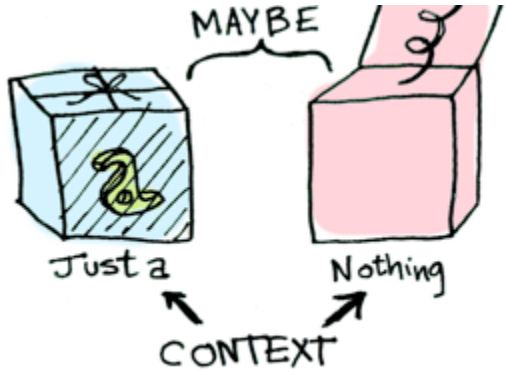

image: adit.io

Typeclasses

- NOT classes in OOP sense
- More like interfaces: define common behavior

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

Functors

```haskell
data Maybe a = Just a | Nothing
```

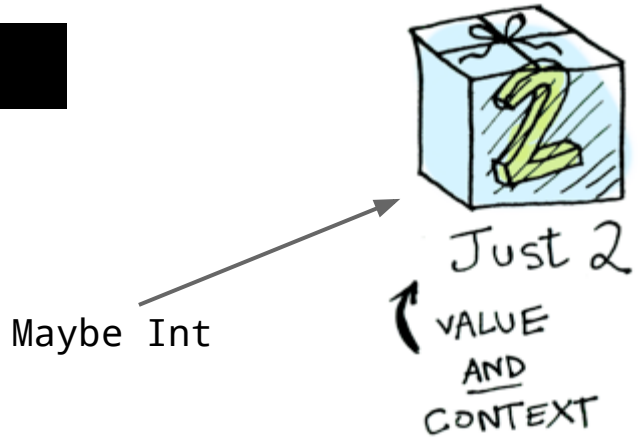

image: adit.io

The Functor Typeclass

```haskell
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

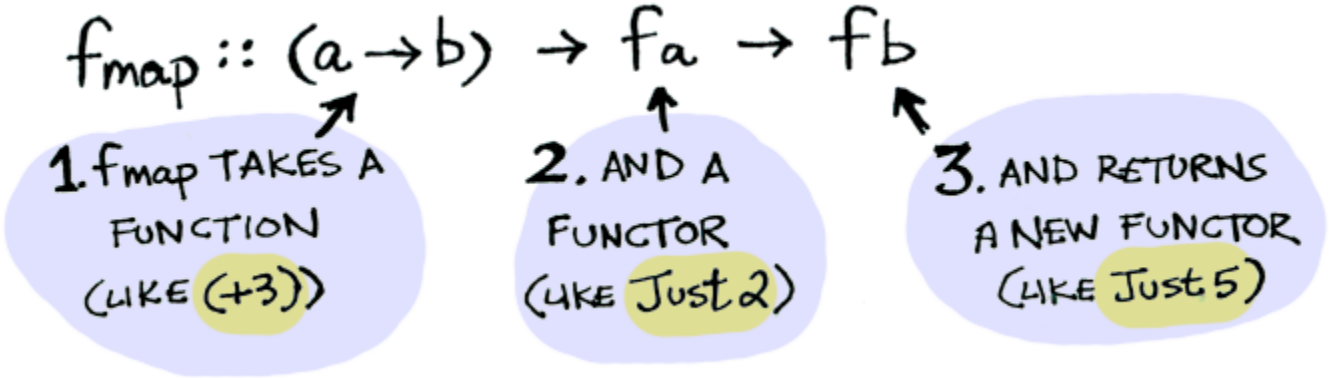

image: adit.io

Example - the `Maybe a` type

```haskell
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

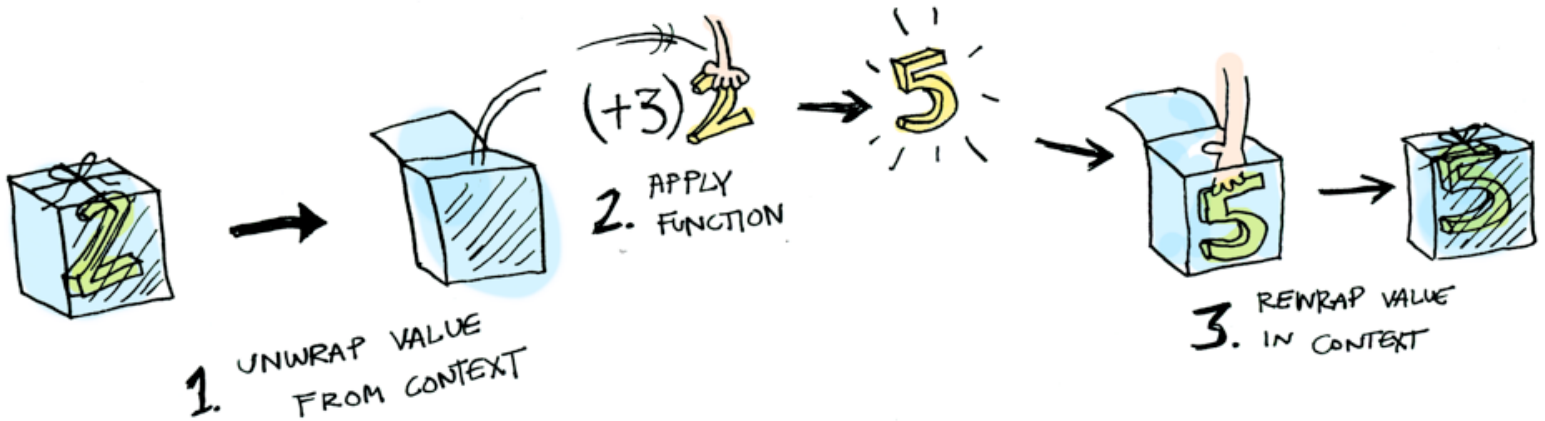

image: adit.io

Example - the `Maybe a` type

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

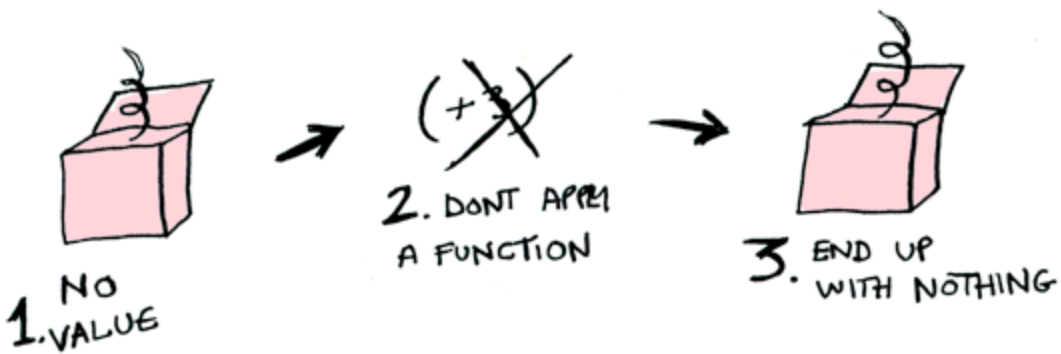

image: adit.io

So what's a Monad?

```
class Monad m where
    return :: a -> m a

    (>>=) :: m a -> (a -> m b) -> m b

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y

    fail :: String -> m a
    fail msg = error msg
```

Monads

```haskell
class Monad m where
    return :: a -> m a

    (>>=) :: m a -> (a -> m b) -> m b

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y

    fail :: String -> m a
    fail msg = error msg
```

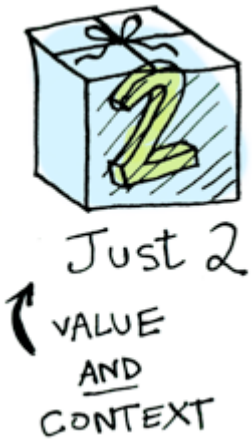

image: adit.io

Monads

```
class Monad m where
    return :: a -> m a

    (>>=) :: m a -> (a -> m b) -> m b

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y

    fail :: String -> m a
    fail msg = error msg
```

image: adit.io

Monads

```haskell
class Monad m where
    return :: a -> m a

    (>>=) :: m a -> (a -> m b) -> m b
```

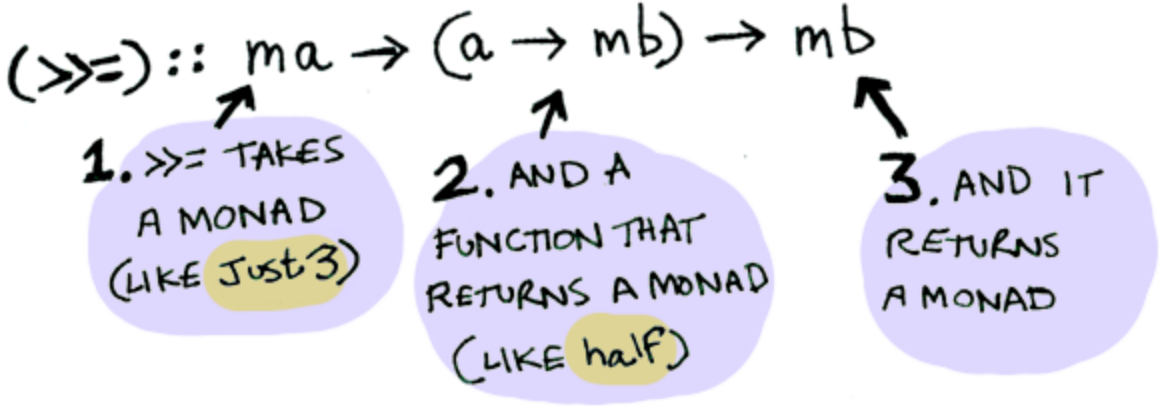

image: adit.io

Just an Example

```
half :: Int -> Maybe Int
ghci> half x = if even x
                 then Just x/2
                 else Nothing

gchi> half 3
Nothing

ghci> half 12 >>= half
Just 3

ghci> half 50 >>= half >>= half
Nothing
```

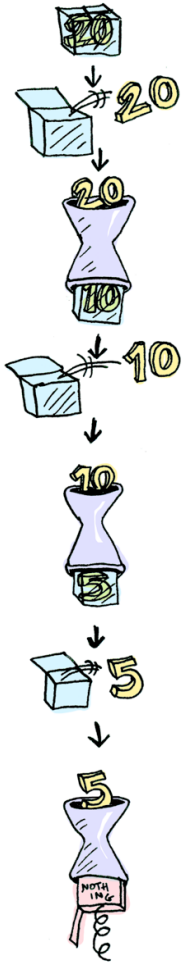

image: adit.io

Do Notation

- What if you need to pass more than just the previous result?

```
ghci> Just 2 >>= (\x -> Just 3 >>= (\y -> Just (x*y)))
Just 6       (Equivalent to: (λx . (λy . x *  y)) Just 3 Just 2)
```

```
routine = do
    x <- Just 2
    y <- Just 3
    Just (x * y)
```

Do Notation

- What if you need to pass more than just the previous result?

```
ghci> Just 2 >>= (\x -> Just 3 >>= (\y -> Just (x*y)))
Just 6
```

```
routine = do
    x <- Just 3
    y <- Just 2
    return (x * y)
```

The Writer Monad

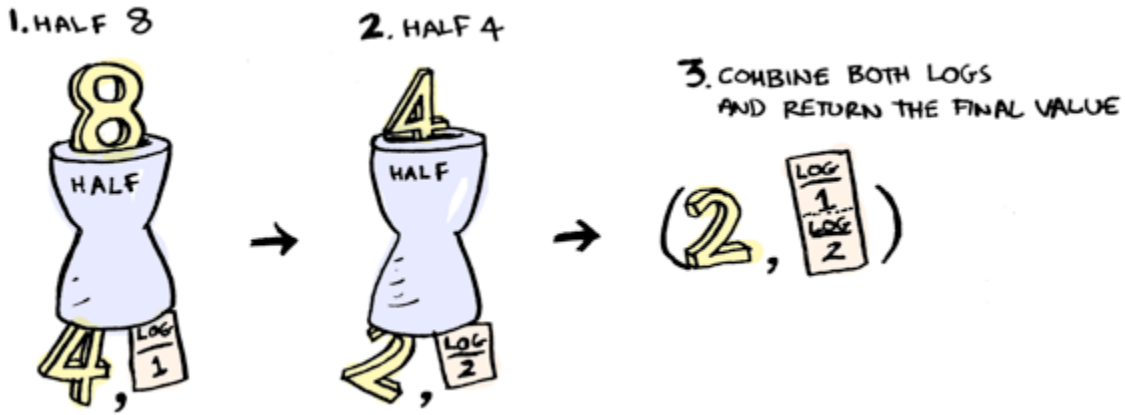

image: adit.io

The Writer Monad

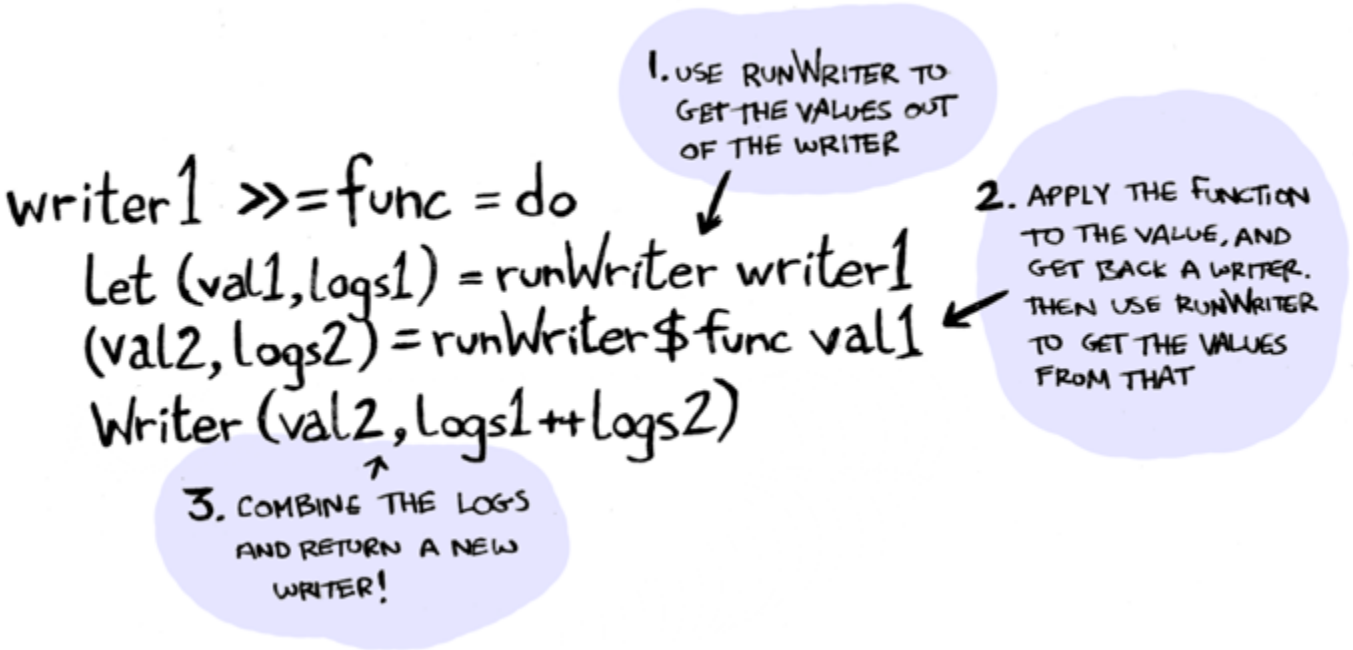

image: adit.io

The Writer Monad

```
import Control.Monad.Writer

mult2 :: Int -> Writer String Int
mult2 x = do
    tell("Multiplying " ++ show x ++ " by 2! ")
    return (x * 2)

add1 :: Int -> Writer String Int
add1 x = do
    tell("Adding 1 to " ++ show x ++ "! ")
    return (x + 1)
```

The Writer Monad

```
ghci> let x = return 2 >>= mult2 >>= add1 >>= mult2
(10,"Multiplying 2 by 2! Adding 1 to 4! Multiplying 5 by 2! ")
```

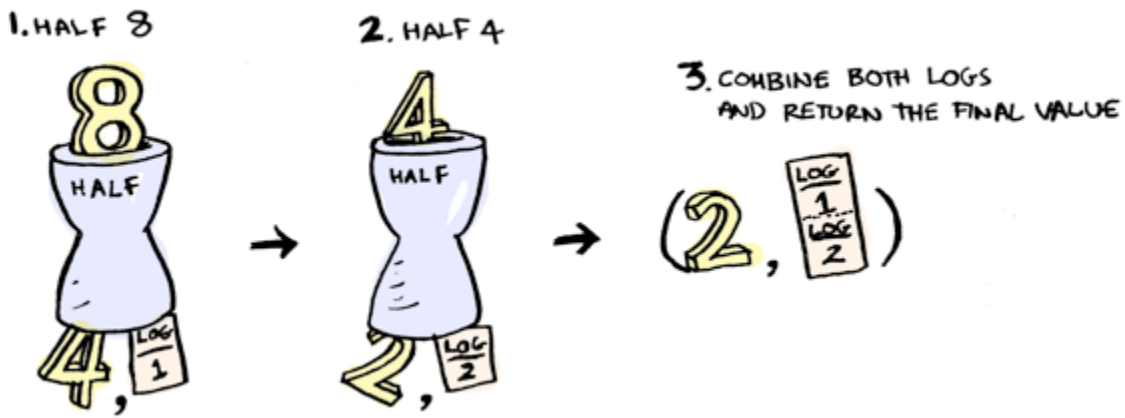

image: adit.io

The Real World

- I/O Monad
- State Monad
- Arrows

Sources

- Learn You a Haskell
- Functors, Applicatives, And Monads In Pictures - adit.io
- Three Useful Monads - adit.io
- http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html
- Hoogle - Haskell Search Engine