Intro to monads

Vladimir Alekseichenko

Agenda

- What is a monad?
- Funtions: (**>>==**), (**>>**), **return** and **fail**
- Monad laws
- **do** notation
- Monads: **Maybe**, **list**

Haskell

It is an advanced

purely-functional

programming language.

http://www.haskell.org

Haskell

It is an advanced

purely-functional

programming language.

http://www.haskell.org

Purity

That means you **cannot** **mutate** variables or introduce „**side effects**” anywhere in your program (short of I/O).

This is **good** and **bad**.

Laboratory is cool, but...

http://www.nanobeach.com/nanolab.php

Real life it isn't a laboratory!

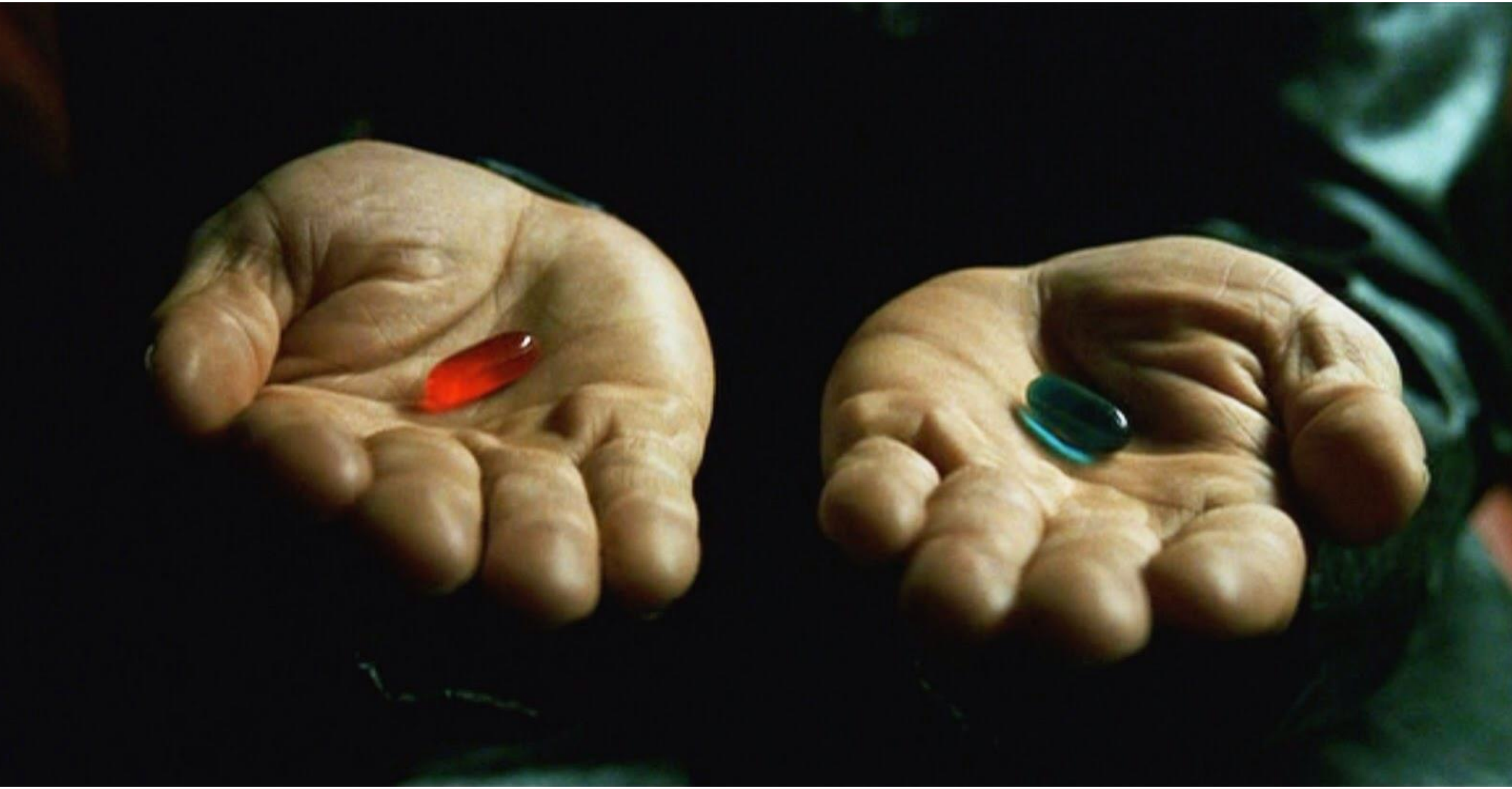
http://joyreactor.cc/post/915055

Impure computation

- May do file or terminal **input/output**.
- May raise **exceptions**.
- May read or write **shared state** (global or local).
- May sometimes fail to produce any results.
- …

http://mvanier.livejournal.com/3917.html

„Pure” pill or „impure” pill?



http://ospreydream.blogspot.com/2013/03/the-matrix-blue-pill-or-red-pill.html

„Pure” pill or „impure” pill?

- Pure + Pure = Pure
- Pure + Impure = Impure
- Impure + Impure = Impure

Pure

```
val = 10 :: Integer
resDouble = sqrt (fromIntegral val)
resInteger = round (sqrt (fromIntegral val))
```

Pure

```
val = 10 :: Integer
resDouble = sqrt $ fromIntegral val
resInteger = round $ sqrt $ fromIntegral val
```

Pure

```
val = 10 :: Integer
resDouble = sqrt $ fromIntegral val
resInteger = round $ sqrt $ fromIntegral val
```

Pure

Function composition

```haskell
sqrtInteger = round . sqrt . fromIntegral
resInteger' = sqrtInteger val
```

Pure

Function composition

```
sqrtInteger = round . sqrt . fromIntegral
resInteger' = sqrtInteger val
```

Pure

Function composition

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

```
sqrtInteger = round . sqrt . fromIntegral
resInteger' = sqrtInteger val
```

Pure

Function composition

Integer

- **fromIntegral** :: (Integral a, Num b) => a -> b

Double

- **sqrt** :: Floating a => a -> a

Integer

- **round** :: (Integral b, RealFrac a) => a -> b

Impure
Monadic functions, monadic values

f :: a -*[something else]*-> b

- f :: a -*[IO]*-> b
- f :: a -*[error]*-> b
- f :: a -*[State s]*-> b
- f :: a -[Maybe]-> b

http://mvanier.livejournal.com/3917.html

Impure

Monadic functions, monadic values

f :: a *IO* -> b

or

f :: a -> *IO* b ⟵ Haskell

http://mvanier.livejournal.com/3917.html

Monadic functions

They’re represented as
pure functions with a *funky*
„**monadic return type**”.

http://mvanier.livejournal.com/3917.html

Monadic functions

x :: *a*

x has type *a*

f x :: *m b*

f x is a „monadic value”

http://mvanier.livejournal.com/3917.html

g :: *a* -> *()* -> *a*

g x () = x

h = g 10

h () -- this will evaluate to 10

The unit type and value are both written as () in Haskell.

http://mvanier.livejournal.com/3917.html

So what is **g (f x)**?

f x :: *m b*

g :: *a -> () -> a*

g (f x) :: *() -> m b*

It's a function which takes a unit value as its argument and returns a monadic value.

http://mvanier.livejournal.com/3917.html

IO

- **getLine** :: *() -> String* -- not in Haskell
- **putStrLn** :: *String -> ()* -- not in Haskell

If we removed the ()s from the type signatures

- **getLine** :: *String*
- **putStrLn** :: *String*

http://mvanier.livejournal.com/3917.html

IO

getLine :: *IO String*

It is a monadic value.

putStrLn :: *String -> IO ()*

It is just a monadic function which happens to be in the IO monad.

http://mvanier.livejournal.com/3917.html

Two monadic functions

f :: *a -> m b*

g :: *b -> m c*

f :: *a -> IO b*

g :: *b -> IO c*

h :: *a -> IO c*

http://mvanier.livejournal.com/4305.html

Compose monadic functions

f :: *a -> IO b*

g :: *b -> IO c*

h :: *a -> IO c*

It doesn’t work!

h = g . f

(.) :: *(b -> c) -> (a -> b) -> a -> c*

http://mvanier.livejournal.com/4305.html

We are looking for solution

val :: *a*

fVal = **f** val *:: IO b*

extVal = **extract** fVal :: *b*

g extVal :: *c*

But itsn’t **IO c.**

http://mvanier.livejournal.com/4305.html

We are looking for solution

val :: *a*

fVal = **f** val *:: IO b*

mapply fVal **g** :: *IO c*

Nice ☺

mapply :: *m b -> (b -> m c) -> m c*

http://mvanier.livejournal.com/4305.html

Bind

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

```
(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
(>>=) :: IO a -> (a -> IO b) -> IO b
(>>=) :: State s a -> (a -> State s b) -> State s b
```

Compose monadic functions

f :: *a -> IO b*
g :: *b -> IO c*
h :: *a -> IO c*

```
h x = f x >>= g
```

```
h = \x -> f x >>= g
```

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

http://mvanier.livejournal.com/4305.html

bind & **compose**
monadic functions

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

(>>) f g = **f** x >>= **g**

Compose monadic functions

f :: *a -> IO b*

g :: *b -> IO c*

h :: *a -> IO c*

h = g >> f

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

http://mvanier.livejournal.com/4305.html

Value apply to monadic function

f :: *a -> m b*

g:: *b -> c*

g >> f

It doesn't work!

return g >> f

return :: a -> m c

It works ☺.

Monad

```
class Monad m where
   (>>=)  :: m a -> (a -> m b) -> m b
   (>>)   :: m a -> m b -> m b
   return :: a -> m a
   fail   :: String -> m a
```

Monad

```
class Monad m where
   (>>=)  :: m a -> (a -> m b) -> m b
   (>>)   :: m a -> m b -> m b
   return :: a -> m a
   fail   :: String -> m a
```

bind

Monad laws

- Left identity
- Right identity
- Associativity

Monad laws

The nice version

```
return >=> f   ==   f

f >=> return   ==   f

(f >=> g) >=> h  ==  f >=> (g >=> h)
```

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

http://mvanier.livejournal.com/4586.html

Monad laws

The ugly (formal) version

```
return x >>= f  ==   f x

m >>= return   ==   m

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

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

http://mvanier.livejournal.com/4586.html

The Maybe monad

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

data Maybe Int = Nothing | Just Int
```

Attempt to define Maybe

```
instance Monad Maybe where
    (>>=)  = {- definition of >>= for Maybe -}
    return = {- definition of return for Maybe -}
```

http://mvanier.livejournal.com/4647.html

The complete definition of the Maybe

```
instance Monad Maybe where
    return x  =  Just x

    Nothing >>= f  =  Nothing
    Just x  >>= f  =  f x
```

do notation

```haskell
main :: IO ()
main = do
   putStrLn "What's your name?"
   name <- getLine
   putStrLn $ "Hello, " ++ name ++ "!"
```

do notation

```haskell
main :: IO ()
main = do
    putStrLn "What's your name?"
    name <- getLine
    putStrLn $ "Hello, " ++ name ++ "!"
```

```haskell
main :: IO ()
main =
    putStrLn "What's your name?" >>
    getLine >>=
    \name -> putStrLn $ "Hello, " ++ name ++ "!"
```

do notation

```haskell
main :: IO ()
main = do
    putStrLn "What's your name?"
    name <- getLine
    putStrLn $ "Hello, " ++ name ++ "!"
```

```haskell
main :: IO ()
main =
    putStrLn "What's your name?" >>
    getLine >>=
    \name -> putStrLn $ "Hello, " ++ name ++ "!"
```

The list monad

f :: a -> *[b]*

f :: a -> *m b*

data List a = *Nil | Cons a (List a)*

f :: a -> *List b*

Using do-notation

```
run :: [(Integer, Integer)]
run =
    do
    x <- [1..6]
    y <- [1..6]
    if (x + y) == 7
        then return(x, y)
        else []
-- Result: [(1,6),(2,5),(3,4),(4,3),(5,2),(6,1)]
```

http://mvanier.livejournal.com/4647.html

Without the do-notation

```
run1 :: [(Integer, Integer)]
run1 =
    [1..6] >>= \x ->
        [1..6] >>= \y ->
            if (x+y) == 7
                then return (x, y)
                else []
-- Result: [(1,6),(2,5),(3,4),(4,3),(5,2),(6,1)]
```

http://mvanier.livejournal.com/4647.html

List comprehension

```
run2 :: [(Integer, Integer)]
run2 = [(x, y) | x <- [1..6], y <- [1..6], x + y == 7]
-- Result: [(1,6),(2,5),(3,4),(4,3),(5,2),(6,1)]
```

http://mvanier.livejournal.com/4647.html

Resources

- http://mvanier.livejournal.com
- https://www.fpcomplete.com/school/starting-with-haskell/basics-of-haskell
- http://adit.io
- http://learnyoua**haskell**.com