Functional Programming
Lecture 12: Haskell Monads

Viliam Lisý

Artificial Intelligence Center
Department of Computer Science
FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

Monad

IO is a special case of generally useful pattern

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

Based on category theory

Way of meaningfully sequencing computations

1. Creating a (separated) boxed value
2. Creating functions for modifying them within the boxes

do Notation

Using monads leads to long sequences of operations chained by operators >>, >>=

```
main = putStrLn "Hello, what is your name?" >>
       getLine >>= \name ->
       putStrLn ("Hello, " ++ name ++ "!")
```

Do notation just makes these sequences more readable

(it is rewritten to monad operators before compilation)

```
main = do putStrLn "Hello, what is your name?"
          name <- getLine
          putStrLn ("Hello, " ++ name ++ "!")
```

Understanding IO Monad

Assume we are implementing getchar in Haskell

what type should it have?

```
getchar :: Char
```

We can then implement

```
get2chars :: String
get2chars = [getchar, getchar]
```

Haskell functions are pure, hence the compiler will

- remove the double call by caching the return value
- if it called the function twice, it would be in arbitrary order

How to solve caching?

Adding a (fake) parameter makes the calls different

```
getchar :: Int -> Char
```

```
get2chars _ = [getchar 1, getchar 2]
```

The calls can still be executed in an arbitrary order

Data dependency can order function execution

(if a result of one function is used by another function)

```
getchar :: Int -> (Char, Int)
```

```
get2chars i0 = [a,b]  where (a,i1) = getchar i0
                            (b,i2) = getchar i1
```

Sequencing Through Data Dependency

The same sequencing problems would reoccur

```
get4chars = [get2chars 1, get2chars 2]
```

Hence we want

```
get2chars :: Int -> (String, Int)

get2chars i0 = ([a,b], i2)  where (a,i1) = getchar i0
                                  (b,i2) = getchar i1
```

We are forcing a specific sequence of executing functions using data dependencies

RealWorld

Good intuition for how IO works

```
type IO a  =  RealWorld -> (a, RealWorld)

> :i IO
```

RealWorld is a fake type serving as the Int from above

The main function is of type IO ()

```
main :: RealWorld -> ((), RealWorld)
```

All IO functions take the real world as an argument and return (a possibly modified) new version of the world

Example

Function main calling getChar two times:

```
getChar :: RealWorld -> (Char, RealWorld)

main :: RealWorld -> ((), RealWorld)
main world0 = let (a, world1) = getChar world0
                  (b, world2) = getChar world1
              in ((), world2)
```

IO Char

IO ()

Only main gets the RealWord. Therefore only main can execute IO actions.

Example

Conditional execution of actions

```
when :: Bool -> IO () -> IO ()
when condition action world =
    if condition
      then action world
      else ((), world)
```

Lazy evaluation will make the action not to be executed

IO Monad

Hides passing of the RealWorld value from the programmer

a->RealWorld->(b,RealWorld)

```
(>>=) :: IO a -> (a -> IO b) -> IO b
(action1 >>= action2) world0 =
   let (a, world1) = action1 world0
       (b, world2) = action2 a world1
   in (b, world2)
```

```
return :: a -> IO a
return x world0  =  (x, world0)
```

Monad is **just** a convenient abstraction to do <u>something</u> like this!

Maybe Monad

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>= _ = Nothing
(Just x) >>= g = g x
```

```
return :: a -> Maybe a
return x  = Just x
```

```haskell
f::Int -> Maybe Int
f 0 = Nothing
f x = Just x

g :: Int -> Maybe Int
g 100 = Nothing
g x = Just x
```

Since it is a monad, we can use the do notation:

```haskell
h :: Int -> Maybe Int
h x = do n <- f x
         g n
```

Exception Handling

Exceptions in Haskell are represented by special types

such as Maybe, Either

Explicit handling of errors makes code hard to read

the special values of the types must be handled everywhere

```
lookUp :: Char -> Either String Int
lookUp name = case M.lookup name vars of
  Just x  -> Right x
  Nothing -> Left ("Variable not found: " ++ show name)
```

```
eval (Add l r) = case eval l of
     m@(Left msg) -> m
     Right x -> case eval r of
         m@(Left msg) -> m
         Right y -> Right (x + y)
```

Exception Handling

Use of monads can hide the error handling

```
data Evaluator a = Ev (Either String a)

instance Monad Evaluator where
    (Ev ev) >>= k =
        case ev of
          Left msg -> Ev (Left msg)
          Right v -> k v
    return v = Ev (Right v)
    fail msg = Ev (Left msg)
```

```
eval :: Expr -> Evaluator Int
eval (Mul l r) = do lres <- eval l
                    rres <- eval r
                    return (lres*rres)
```

https://www.schoolofhaskell.com/user/bartosz/basics-of-haskell/10_Error_Handling

List Monad

Suitable for combining non-deterministic computations

can return multiple results and we want to continue with all

```
(>>=) :: [a] -> (a -> [b]) -> [b]
xs >>= k = concat (map k xs)
```

```
return :: a -> [a]
return x  = [x]
```

List Comprehensions

```
squares lst = do
    x <- lst
    return (x * x)
```

```
squares lst = lst >>= \x -> return (x * x)
```

```
squares lst = concat $ fmap k lst
  where k = \x -> [x * x]
```

List Comprehensions

```
pairs l1 l2 = do
    x <- l1
    y <- l2
    return (x, y)
```

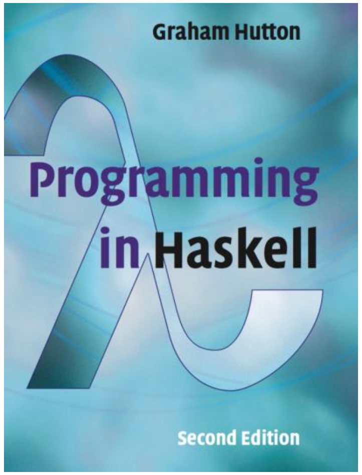
```
pairs l1 l2 = [(x, y) | x <- l1, y <- l2]
```

```
pairs l1 l2 = l1 >>= \x -> l2 >>= \y -> return (x,y)
```

Guards can also be added, but it requires MonadPlus, for more advanced combinations of computations.

Acknowledgements

- https://wiki.haskell.org/Introduction_to_IO
- https://wiki.haskell.org/IO_inside



Monads Summary

IO in pure functional programming is problematic

- it prevents optimization possible with pure functions
- it requires explicit ordering of pseudo-function calls

Haskell encloses these operations to IO actions

– no result of pseudo-function can leave the IO "container"

Monads are a useful abstraction for

– sequencing operations on containers
– making operation within containers

Build-in Monads

– Maybe, Either e, [], IO

Random numbers

- decent random numbers
  - System.Random
- cryptographically secure random numbers
  - Crypto.Random
- Getting random numbers generator
  - mkStdGen <seed>
  - getStdGen

Random numbers

- Getting a random nubmer
  - randomR :: (RandomGen g, Random a) => (a, a) -> g -> (a, g)
- Range can be infered from output type
  - random :: (RandomGen g, Random a) => g -> (a, g)
- Using the standard generator in the IO monad
  - randomRIO (0,1)
  - randomRIO (0,1::Float)
  - randomIO :: IO Float

Random sequence

```
myRnds :: Int -> [Float]
myRnds seed = randSeq (mkStdGen seed)
  where randSeq gen = let (v,g2) = random gen
                                     in v:randSeq g2
```

Build-in variant

- randoms <generator>
- randomRs <range> <generator>

Random with IO

```
*Main> :t getStdGen
getStdGen :: IO StdGen
*Main> :t random
random :: (RandomGen g, Random a) => g -> (a, g)
```

```
import System.Random

main = do
  g <- getStdGen
  print . take 10 $ (randomRs ('a', 'z') g)
  print . take 10 $ (randomRs ('a', 'z') g)
```

Random values of custom type

Type must be an instance of class Random

```
data Coin = Heads | Tails deriving (Show, Enum, Bounded)

instance Random Coin where
  randomR (a, b) g =
      let (x, g') = randomR (fromEnum a, fromEnum b) g
      in  (toEnum x, g')
  random g = randomR (minBound, maxBound) g
```