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 $\gg, \gg=$

```haskell
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)

```haskell
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

\[
\text{getchar} :: \text{Int} -> \text{Char}
\]

\[
\text{get2chars} _ = [\text{getchar} 1, \text{getchar} 2]
\]

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)

\[
\text{getchar} :: \text{Int} -> (\text{Char}, \text{Int})
\]

\[
\text{get2chars} i0 = [a,b] \text{ where } (a,i1) = \text{getchar} i0
\]
\[
(b,i2) = \text{getchar} i1
\]
Sequencing Through Data Dependency

The same sequencing problems would reoccur

\[
\text{get4chars} = [\text{get2chars 1, get2chars 2}]
\]

Hence we want

\[
\text{get2chars} :: \text{Int} \rightarrow (\text{String, Int})
\]

\[
\text{get2chars i0} = ([a,b], i2) \text{ where } (a,i1) = \text{getchar i0} \\
(b,i2) = \text{getchar i1}
\]

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

Good intuition for how IO works

```haskell
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 ()

```haskell
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:

<table>
<thead>
<tr>
<th>getChar :: RealWorld -&gt; (Char, RealWorld)</th>
<th>IO Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>main :: RealWorld -&gt; (() , RealWorld)</td>
<td>IO ()</td>
</tr>
<tr>
<td>main world0 = let (a, world1) = getChar world0</td>
<td></td>
</tr>
<tr>
<td>(b, world2) = getChar world1</td>
<td></td>
</tr>
<tr>
<td>in (() , world2)</td>
<td></td>
</tr>
</tbody>
</table>

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

\[
(\gg\gg=) :: \text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b
\]

\[
(\text{action1 } \gg\gg= \text{action2}) \text{world}0 = \\
\quad \text{let}\ (a, \text{world}1) = \text{action1 } \text{world}0 \\
\quad (b, \text{world}2) = \text{action2 } a \text{ world}1 \\
\quad \text{in } (b, \text{world}2)
\]

\[
\text{return} :: a \rightarrow \text{IO } a \\
\text{return } x \text{ world}0 = (x, \text{world}0)
\]

Monad is just a convenient abstraction to do something like this!
Maybe Monad

\[ (\gg\gg=) \:: \text{Maybe } a \to (a \to \text{Maybe } b) \to \text{Maybe } b \]
\[ \text{Nothing} \gg\gg= _ = \text{Nothing} \]
\[ (\text{Just } x) \gg\gg= g = g \; x \]

\[ \text{return} \:: a \to \text{Maybe } a \]
\[ \text{return } x = \text{Just } x \]
Since it is a monad, we can use the do notation:

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

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

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

```haskell
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

```haskell
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

\[
(\gg\gg=) :: [a] \rightarrow (a \rightarrow [b]) \rightarrow [b] \\
x \cdot \gg\gg= k = \text{concat} (\text{map } k \ x) \\
\]

return :: a \rightarrow [a] \\
return x = [x]
List Comprehensions

\[
\text{squares \ lst} = \text{do}
\]
\[
x \leftarrow \text{lst}
\]
\[
\text{return} \ (x \times x)
\]

\[
\text{squares \ lst} = \text{lst} \gg= \ x \rightarrow \text{return} \ (x \times x)
\]

\[
\text{squares \ lst} = \text{concat} \ \$ \ \text{fmap} \ k \ \text{lst}
\]
\[
\text{where} \ k = \ x \rightarrow \ [x \times x]
\]
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 number

- \text{randomR} :: \text{(RandomGen g, Random a)} \to (a, a) 
  \to g \to (a, g)

Range can be inferred from output type

- \text{random} :: \text{(RandomGen g, Random a)} \to g \to (a, g)

Using the standard generator in the IO monad

- \text{randomRIO} (0,1)
- \text{randomRIO} (0,1::Float)
- \text{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

```haskell
import System.Random

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

*Main> :t getStdGen
getStdGen :: IO StdGen

*Main> :t random
random :: (RandomGen g, Random a) => g -> (a, 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