

Functional Programming

Lecture 10

Rostislav Horčík
Niklas Heim

Czech Technical University in Prague
Faculty of Electrical Engineering
xhorcik@fel.cvut.cz
heimnikl@fel.cvut.cz

Mutating the world

I/O operations are fundamentally mutating:

```
> putStrLn "Hello World"    | stdout: Hello World
```

```
> putStrLn "Hello World"    | stdout: Hello World  
                             Hello World
```

Unless the whole world is an argument to our functions:

```
putStrLn :: String -> World -> World  
getLine  :: World -> (String, World)
```

Mutating the world

```
putStrLn :: String -> World -> World  
getLine  :: World -> (String, World)
```

With the definitions above we can write **pure** I/O functions:

```
helloworld :: World -> World  
helloworld w1 = w4 where  
    w2          = putStrLn "What is your name?" w1  
    (name, w3) = getLine w2  
    w4          = putStrLn ("Hello " ++ name) w3
```

Mutating the world

In practice we don't mutate the world, but we use **monads**.

```
helloworld :: IO ()  
helloworld = do  
    putStrLn "What is your name?"  
    name <- getLine  
    putStrLn ("Hello " ++ name)
```

Monads extend far beyond I/O and mutation. Common examples are: **Maybe**, **list []**, **State**, **Either**.

IO actions

IO actions

Haskell separates the part of the program with side effects using values of special types

IO is a functor satisfying further properties (**monad**) such that a value of type **IO a** is an **action**, which when executed produces a value of type **a**

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

```
putStrLn :: String -> IO ()
```

```
getLine  :: IO String
```

The IO actions can be composed to build up more complex actions.

Hello World

Haskell executes only one IO action in a program, the action returned by the function `main`.

```
main :: IO ()
```

```
main = putStrLn "Hello, World!"
```

```
$ ghc <filename.hs>; ./<filename>
```

```
$ runghc <filename.hs>
```

IO actions (not only the one returned by `main`) can be also executed in GHCi by evaluating them.

Hello World

We can try to rewrite `helloworld` in terms of `IO`:

```
helloworld :: IO ()
helloworld =
    let ac_name = getLine    -- IO String
    -- This fails! We cannot ++ with an action!
    in putStrLn ("Hello " ++ ac_name)
```

What we need is:

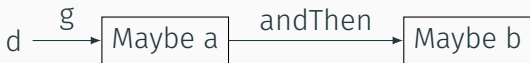
```
??? :: IO String -> (String -> IO ()) -> IO ()
```


Sequencing computations

Last time we saw how to compose failing computations by

`andThen :: Maybe a -> (a -> Maybe b) -> Maybe b`

$$a \xrightarrow{h} \text{Maybe } b$$



`andThen` is in fact the **bind operator** `>>=` making the functor `Maybe` into a monad.

Monads

Monads

Maybe, IO, [] are instances of a type class called **monad**:

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

`>>=` pulls out the result stored in `m a` and pass it to another computation represented by a function of type `a -> m b`.

`>>` composes two computations and the second one ignores the result of the first one.

`return` allows to embed “pure” values into the computational context.

Monads are functors

`>>` can be implemented in terms of `>>=` as follows:

```
x >> f = x >>= \_ -> f
```

Every monad is a functor since

```
fmap f x = x >>= return . f
```

Note that

```
x :: m a
```

```
f :: a -> b
```

```
return :: b -> m b
```

```
return . f :: a -> m b
```

`(>>) :: IO a -> IO b -> IO b`

composes two IO actions (the first action is performed only for its side-effect), e.g.

```
putStrLn "Hello" >> putStrLn "World"
```

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

`x >>= f` is the action performing first `x`, passing its result to `f` that returns a second action to be performed, e.g.

```
getLine >>= putStrLn
```

Hello World again

helloworld in terms of >>=

```
helloworld :: IO ()
```

```
helloworld =
```

```
    putStrLn "What is your name?" >>
```

```
    getLine >>=
```

```
    \name -> putStrLn ("Hello " ++ name)
```

return

```
return :: a -> IO a
```

creates an IO action that does nothing except produces the given value.

Useful when we need to combine results of previous actions by a pure function:

```
getSquare :: IO Int
getSquare = putStrLn "Enter number:"
           >> getLine
           >> \line -> let n = read line
                       in return (n*n)
```

Maybe monad

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

```
x >>= f = case x of  
    Nothing -> Nothing  
    Just y -> f y
```

```
return x = Just x
```

```
safeSecond :: [a] -> Maybe a
```

```
safeSecond xs = safeTail xs >>= safeHead
```

```
sumFirstTwo :: Num a => [a] -> Maybe a
```

```
sumFirstTwo xs = safeHead xs  
    >>= \first -> safeSecond xs  
    >>= \second ->  
        return (first + second)
```


Separation of IO side-effects

We can access the value stored in **Maybe** monad via the data constructor **Just**.

```
getMaybe :: Maybe Int -> Int
getMaybe (Just x) = x
getMaybe _ = 0
```

However, we cannot do the same for **IO**. There is no accessible data constructor allowing to do pattern matching on values of type **IO a**. Thus there is no function of type

```
unsafe :: IO a -> a
```

Consequently, all the values obtained as results of impure actions with side-effects have to be closed inside **IO**.

We can manipulate them only via **>>=**.

Do notation

`do` is a syntax block, such as `where` and `let`

- action on a separate line gets executed
- `v <- x` runs action `x` and binds the result to `v`
- `let a = b` defines `a` to be the same as `b` until the end of the block (no `in` is used)

```
getSquare2 :: IO Int
```

```
getSquare2 = do putStrLn "Enter number:"  
               line <- getLine  
               let n = read line  
               return (n*n)
```

```
(>>=) :: [a] -> (a -> [b]) -> [b]
```

```
xs >>= f = concat $ map f xs  
return x = [x]
```

```
> [1,2,3] >>= \x -> [x,10*x,100*x]  
[1,10,100,2,20,200,3,30,300]
```

Monadic sequencing

Suppose we have a list of monadic actions and we want to evaluate all of them.

```
sequence :: Monad m => [m a] -> m [a]
```

```
sequence_ :: Monad m => [m a] -> m ()
```

```
ioActions :: [IO ()]
```

```
ioActions = [ print "Hello!"  
              , putStrLn "just kidding",  
              , getLine >> print]
```

```
> sequence_ ioActions
```

Monadic maps

Monadic analogs of `map`

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

```
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
```

```
> mapM putStrLn ["a", "b", "c"]
```

```
a
```

```
b
```

```
c
```

```
[(),(),()]
```

Homework assignment 4 — Parser of λ -programs

Aim: To practice monadic parsing in Haskell, together with HW3 build a complete λ -calculus interpreter

$0 := (\backslash s. (\backslash z. z))$

$S := (\backslash w. (\backslash y. (\backslash x. (y ((w y) x)))))$

$1 := (S\ 0)$

$2 := (S\ 1)$

$((2\ S)\ 1)$

Points: 13

Deadline: June 8

Penalty: after deadline -1 points every day (at most -12)

Description: all details can be found in CW

Summary

- Results of IO actions are enclosed in **IO** monad.
- We can manipulate them only via monadic operators.
- **Monads** are special functors allowing to sequence monadic actions/computations via the bind operator `>>=`.
- Other monads are e.g. **Maybe**, `[]`.
- Action sequencing can be also done in **do-blocks**.
- There are monadic variants of `map`: `mapM`, `mapM_`.
- A list of actions can be composed by **sequence** or `sequence_`.

Functors as computational context

Functors as computational context

Functor is a type constructor `f` having an implementation of

```
fmap :: (a -> b) -> f a -> f b
```

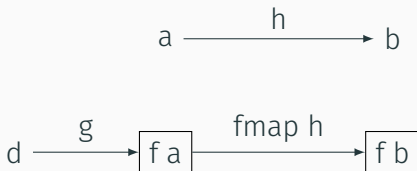
It can be also viewed as computational context. `f a` stores the result of a computation.

- **Maybe** `a` — result of a possibly failing computation
- `[a]` — all possible results of a non-deterministic computation
- **IO** `a` — result of a computation having an IO side-effect

IO functor allows haskell programs to execute computations having IO side-effects. It has to satisfy further properties than being a functor. It must be a monad.

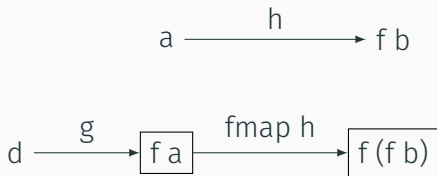
Computational contexts

Suppose we have a computation $g :: d \rightarrow f\ a$ whose result is stored in $f\ a$. Then we can transform its result by any “pure” function h .



Computational contexts

Suppose $h :: a \rightarrow f\ b$ is not “pure” and we want to chain both computations g followed by h . Now `fmap` alone does not suffice.



We don't want results of type **Maybe** (**Maybe** a) or **IO** (**IO** a).