CS-205 lecture 9: The IO monad

Cécilia Pradic

28/10/24

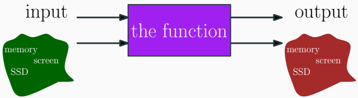
Autograder is live

Link to submit your coursework

`https://csautograder.swansea.ac.uk/web/project/69`

- Detailed submission instructions on canvas
- A bug was reported to me yesterday, should be fixed
- All tests re-ran after final submission
- 37 marks allocated automatically, the rest by handgrading
- The handgrader might compensate for harsh automated grading

Last week: interactive programming



What we have seen

How to

- write types for programs with IO side-effects in types `IO a`

```
print   :: Show a => a -> IO ()
getLine :: IO String
```

- combine them using bind `>>=` or the `do` notation
- a couple of examples
- compile haskell programs using `ghc`

Extended example: validating input

```haskell
getYN :: String -> IO Bool
getYN prompt = putStr (prompt ++ "[y/N]:") >>
               hFlush stdout >>
               getLine >>= \s ->
               if s `elem` answers then
                 return (s `elem` yanswers)
               else
                 putStrLn "Wrong input!" >> getYN prompt
       where yanswers = ["y","Y","yes","Yes"]
             answers  = yanswers ++ ["","n", "N","no","No"]
```

Extended example: validating input

```
getYN :: String -> IO Bool
getYN prompt = do {
                  putStr (prompt ++ "[y/N]:");
                  hFlush stdout;
                  s <- getLine;
                  if s `elem` answers then
                     return (s `elem` yanswers)
                  else
                      do {
                           putStrLn "Wrong input!";
                           getYN prompt
                         }
                 }
      where yanswers = ["y","Y","yes","Yes"]
            answers  = yanswers ++ ["","n", "N","no","No"]
```

Some tps/considerations for the lab

- I have not gone over all the `IO` primitives

  → use the online documentation (`hackage/hoogle`)
- You might need some import statement to import functions like `hFlush` or `isDigit` as in e.g.

```
import Data.Char (isDigit) -- imports only isDigit
import System.IO -- imports everything in the module
```

- `hFlush stdout` ≡ `fflush(stdout)`

  flushes the `stdout` buffer → forces printing

Some further topics

Warning

The rest of the lecture will survey some topics you could look into if you want to keep writing Haskell in the future/are curious

Before we move on, questions about Haskell/CW/etc?

Ofc you are free to ask at any later point :)

(more detailed explanation on the material below in `lecture11.hs`)

Further topic 1: monads

Extended do notation?

```
doList :: [(Int,Char)]
doList = do {
              x <- [1..5];
              y <- ['a','z'];
              return (x,y)
            }
```

Extended do notation?

```haskell
divMaybe :: Int -> Int -> Maybe Int
divMaybe x 0 = Nothing
divMaybe x y = Just (x `div` y)

doMaybe :: Int -> Int -> Int -> Maybe Int
doMaybe x y z = do {
                      a <- divMaybe x y;
                      b <- divMaybe z a;
                      return (a + b)
                   }
```

Workhorse behind this: the Monad typeclass

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

- `m ::  * -> *` is a variable, but not for a type
- Monads consist of a very generic, yet useful abstractions
- Typical instances: `Monad IO`, `Monad []`, `Monad Maybe`, `Monad (Cont r)`, `Monad (State s)`
- `Monad (State s)` = code “as if” we had mutable variables

Maybe the next example to look at if you are interested

The state monad

```
data State s a = Stateful (s -> a * s)

return :: a -> State s a
return x m = (x, m)

(>>=) :: State s a -> (a -> State s b) -> State s b
(Stateful c) >>= f = Stateful
                      \m -> let (x, m') = c m in
                            let Stateful g = f x in
                            g m'

escape :: State s a -> s -> a
escape (Stateful c) m = fst (c m)
```

- Usefuleness: one can go back to pure computations via `escape`

  (function of type `IO a -> a` named `unsafePerformIO`)

A monad you have already seen: Maybe

```
divMaybe :: Int -> Int -> Maybe Int
divMaybe x 0 = Nothing
divMaybe x y = Just (x `div` y)

doMaybe :: Int -> Int -> Int -> Maybe Int
doMaybe x y z = do {
                      a <- divMaybe x y;
                      b <- divMaybe z a;
                      return (a + b)
                   }
```

The list monad

```
doList :: [(Int,Int)]
doList = do {
              x <- [1..10];
              y <- [1..10];
              if even x /= even y then
                 return (x,y)
              else []
            }
```

- the list monad has further structure `[]`, ...
- special syntax for this in Haskell: **list comprehension**

```
doList1 = [(x,y) | x <- [1..10], y <- [1..10], even x /= even y]
```