

Functional Programming

Lecture 10: Input and output

Twan van Laarhoven

21 November 2022

Outline

- Side-effects and I/O
- The I/O interface
- Case study: Haskinator
- References
- Summary



IO in Haskell

Side-effects

- A function has a side effect if it
 - modifies a mutable data structure or variable
 - draws something on the screen
 - asks for keyboard input
 - wipes out your hard disk
 - launches a cruise missile
 - ...
- Functions with side-effects (sometimes called procedures) can be unpredictable depending on the state of the system.
- Functions without side-effects can be executed anytime. They will always give the same result when given the same input.



Side-effects in Haskell?

- A pure functional language such as Haskell is *referentially transparent*: A reference can be replaced by its definition anywhere.
- Imagine a function

```
putStr :: String → ()
```

- What is the value and what is the effect of

```
let x = putStr "hello" in [x,x]
```

- What about

```
[putStr "hello", putStr "hello"]
```

- Or

```
let x = putStr "hello" in take 0 [x,x]
```



To-do lists of actions

Idea:

- New type `IO a` of *actions*
- `putStr "hello"` is an action, but has *no side effect*
- `IO a` is type of computation that, when executed, may do I/O, then returns an element of type `a`.
- Executing `putStr "hello"` does I/O, then returns `()`

Execute an action:

- With the interactive prompt in `Ghci`
- The main function
- As part of a larger action



Basic I/O actions

- Console I/O
 - `putStr` :: `String` \rightarrow `IO ()`
 - `putStrLn` :: `String` \rightarrow `IO ()`
 - `getLine` :: `IO String`
- Console I/O via `Show` and `Read`
 - `print` :: (`Show` `a`) \Rightarrow `a` \rightarrow `IO ()`
 - `readLn` :: (`Read` `a`) \Rightarrow `IO a`
- File I/O
 - `type` `FilePath` = `String`
 - `writeFile` :: `FilePath` \rightarrow `String` \rightarrow `IO ()`
 - `readFile` :: `FilePath` \rightarrow `IO String`
- Many, many more



Combining I/O actions

- Sequence of actions can be combined into a composite action using the do-notation.

```
welcome :: IO ()  
welcome = do  
    putStr "Please enter your name.\n"  
    s ← getLine  
    putStr ("Welcome " ++ s ++ "!\n")
```

- $x \leftarrow a$ gives a name to the result of an action a .
Read as “Execute a and call the result x ”, or “bind x to the result of a ”
Note that a has type $\text{IO } t$, whereas x has type t .



Combining I/O actions

- Sequence of actions can be combined into a composite action using the do-notation.

```
welcome :: IO ()  
welcome =  
  do { putStr "Please enter your name.\n";  
    s ← getLine; putStr  
      ("Welcome " ++ s ++ "!\n") }
```

- $x \leftarrow a$ gives a name to the result of an action a .
Read as “Execute a and call the result x ”, or “bind x to the result of a ”
Note that a has type $\text{IO } t$, whereas x has type t .
- Syntax: layout-sensitive versus braces and semicolons



Combining I/O actions (continued)

- $m \gg n$ first executes m and then n
 $(\gg) :: IO\ a \rightarrow IO\ b \rightarrow IO\ b$
- $m \gg= n$ additionally feeds the result of the first computation into the second
 $(\gg=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$

- Example

welcome :: IO ()

welcome

```
= putStr "Please enter your name.\n" >>  
  getLine >>= \s ->  
  putStr ("Welcome " ++ s ++ "!\n")
```



Combining I/O actions (continued)

- $m \gg n$ first executes m and then n
 $(\gg) :: IO\ a \rightarrow IO\ b \rightarrow IO\ b$
- $m \gg= n$ additionally feeds the result of the first computation into the second
 $(\gg=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$
- Example, with sugar
welcome $:: IO\ ()$
welcome = **do**
 putStr "Please enter your name.\n"
 s \leftarrow getLine
 putStr ("Welcome " ++ s ++ "!\n")



Pure actions

- Turning *pure* values into impure actions

`pure :: a → IO a`

As a to do list: “do nothing, just return this value”.

- Note: `pure` used to be called `return`.

Note 2: `return` does *not* end the execution like in other languages.

- Example: reading a string from the keyboard:

```
getLine :: IO String
getLine = do
  x ← getChar
  if x == '\n' then
    pure []
  else do xs ← getLine
          pure (x:xs)
```



Example: file I/O

```
processFile :: (String → String) → FilePath → FilePath → IO ()  
processFile f inFile outFile = do  
  s ← readFile inFile  
  let s' = f s  
  writeFile outFile s'
```

Pattern: Keep your logic and computation pure



Example: file I/O

```
processFile :: (String → String) → FilePath → FilePath → IO ()
processFile f inFile outFile = do
  s ← readFile inFile
  let s' = f s
  writeFile outFile s'
```

Pattern: Keep your logic and computation pure

A pure core in a small impure shell



Actions are not values

Given $x :: \text{IO Int}$, how do you get the Int out?



Actions are not values

Given $x :: \text{IO Int}$, how do you get the Int out?

This question makes no sense!

Consider

```
grandma'sRecipe :: IO Cake
grandma'sRecipe = do
  whisk eggs
  add flour (Gram 250)
  add sugar (Gram 175)
  bake (Celsius 180) (Minutes 45)
```

A recipe doesn't "contain" a cake.

Actions vs. values

Which of these is correct?

$f_1 = \text{putStrLn } x \text{ where } x = \text{getLine}$

$f_2 = \text{putStrLn } (\text{getLine} ++ \text{getLine})$

$f_3 = \text{getLine}$

$f_4 = \text{do}$

$x \leftarrow \text{getLine}$

$y \leftarrow \text{getLine}$

$z \leftarrow x ++ y$

$\text{putStrLn } z$

$f_5 = \text{length} . \text{getLine}$

$f_6 = \text{getLine} \gg= \text{length}$

I/O actions as first-class citizens

You can freely mix IO actions with, say, lists

```
printNumbers :: [IO ()]  
printNumbers = [print i | i <- [0..9]]
```

This is a list of actions.

We can combine the actions in sequence

```
main :: IO ()  
main = sequence_ printNumbers
```

Use the familiar list design pattern

```
sequence_ :: [IO ()] → IO ()  
sequence_ [] = pure ()  
sequence_ (a : as) = a >> sequence_ as
```



I/O actions as first-class citizens (cont.)

IO actions as function arguments:

```
when :: Bool → IO () → IO ()  
when True  act = act  
when False _  = pure ()
```

Usage:

```
main = do  
  args ← getArgs  
  when (length args < 2) $ do  
    putStrLn "This program needs 2 arguments"  
    exitFailure  
  doThings
```



map for IO

`mapM` :: (a → IO b) → [a] → IO [b]

`mapM_` :: (a → IO b) → [a] → IO ()



map for IO

`mapM` :: (a → IO b) → [a] → IO [b]

`mapM_` :: (a → IO b) → [a] → IO ()

With recursion

`mapM` f [] = pure []

`mapM` f (x:xs) = **do**

 y ← f x

 ys ← `mapM` f xs

 pure (y:ys)



map for IO

`mapM` :: (a → IO b) → [a] → IO [b]

`mapM_` :: (a → IO b) → [a] → IO ()

With recursion

`mapM` f [] = pure []

`mapM` f (x:xs) = **do**

 y ← f x

 ys ← `mapM` f xs

 pure (y:ys)

With sequence

`mapM` f xs = `sequence` (`map` f xs)



Utilities

`liftM` :: $(a \rightarrow b) \rightarrow \text{IO } a \rightarrow \text{IO } b$

`liftM` f `act` = `do`

`x` \leftarrow `act`

`pure` (`f act`)

`liftM2` :: $(a \rightarrow b \rightarrow c) \rightarrow \text{IO } a \rightarrow \text{IO } b \rightarrow \text{IO } c$

`foldM` :: $(b \rightarrow a \rightarrow \text{IO } b) \rightarrow b \rightarrow [a] \rightarrow \text{IO } b$

`filterM` :: $(a \rightarrow \text{IO } \text{Bool}) \rightarrow [a] \rightarrow \text{IO } [a]$

`replicateM` :: $\text{Int} \rightarrow \text{IO } a \rightarrow \text{IO } [a]$



Stand-alone programs

- Entry point: `main :: IO ()`
- Compilation: `ghc --make MyProgram.hs`
- Typically multiple modules, main is in **module** `Main`
- In real-life you use a package/build manager (cabal, stack)
 - `cabal install FancyLibrary`
 - `cabal init`
 - `cabal build`



Our first real program

Case study: Akinator

Think about a real or fictional character...
I will try to guess who it is.

iGuessTheCelebrity :: IO ()

Think of number between a and b...
I will try to guess the number.

iGuessTheNumber :: Integer → Integer → IO ()



A game tree

Goal: separate the game logic from the underlying data.

```
data Tree a b = Tip a | Node b (Tree a b) (Tree a b)
  deriving (Show)
```

The type is parametric in the type of labels of external nodes (i.e. tips) and in the type of labels of internal nodes.

```
bimap :: (a1 → a2) → (b1 → b2) → (Tree a1 b1 → Tree a2 b2)
bimap f g (Tip a)      = Tip (f a)
bimap f g (Node b l r) = Node (g b) (bimap f g l) (bimap f g r)
```

The function `bimap` is a binary variant of `map`.



The game logic

```
guess :: Tree String String → IO ()  
guess (Tip s) =  
    putStrLn s  
guess (Node q l r) = do  
    b ← yesOrNo q  
    if b then guess l  
        else guess r
```

```
yesOrNo :: String → IO Bool  
yesOrNo question = do  
    putStrLn question  
    answer ← getLine  
    pure (map toLower answer `isPrefixOf` "yes")
```



I guess the celebrity

```
iGuessTheCelebrity = do
  putStrLn ("Think of a celebrity.")
  guess (bimap (\s → s ++ "!")
               (\q → q ++ "?") celebrity)
```

```
celebrity :: Tree String String
celebrity = Node "Female"
  (Node "Actress"
    (Tip "Emilia Clarke")
    (Tip "Adele"))
  (Node "Actor"
    (Tip "Peter Dinklage")
    (Tip "Max Verstappen"))
```



I guess the number

```
iGuessTheNumber a b = do
  putStrLn ("Think of number between " ++ show a ++ " and "
           ++ show b ++ ".")
  guess (bimap (\n → show n ++ "!")
              (\m → "≤ " ++ show m ++ "?")
        (nest a b))
```

`nest :: Integer → Integer → Tree Integer Integer`

`nest a b`

```
| a == b    = Tip a
| otherwise = Node m (nest a m) (nest (m + 1) b)
where m = (a + b) `div` 2
```



Other effects

More IO goodies

The **IO** type offers a lot more:

- exception handling
- threads
- environment variables
- updatable variables (aka references or pointers)
- updatable arrays
- ...

IO is Haskell's *sin bin*



Random numbers

Is generating a random number a side-effect?

Random numbers

Is generating a random number a side-effect?

A pure function always gives the same answer!

`random` :: `Int`

`random` = 4 — *Chosen by a fair dice roll, guaranteed to be random.*



Random numbers

Is generating a random number a side-effect?

A pure function always gives the same answer!

`random` :: `Int`

`random` = 4 — *Chosen by a fair dice roll, guaranteed to be random.*

As an (IO) action

`class Random a`

`randomIO` :: `Random a` \Rightarrow `IO a`

`randomRIO` :: `Random a` \Rightarrow `(a, a)` \rightarrow `IO a`



References

Updatable variables live in the IO world

```
type IOREf a
newIORef    :: a → IO (IORef a)
readIORef   :: IORef a → IO a
writeIORef  :: IORef a → a → IO ()
modifyIORef :: IORef a → (a → a) → IO ()
```

- `newIORef` creates a new `IORef` and initializes it
- `readIORef` reads the value of an `IORef`
- `writeIORef` writes a new value into an `IORef`
- `modifyIORef` applies a function to the value of an `IORef`, and writes the output



References: examples

Copying “variables”

```
copy :: IORef a → IORef a → IO ()  
copy x y = do val ← readIORef y  
             writeIORef x val
```

Swapping “variables”

```
swap :: IORef a → IORef a → IO ()  
swap x y = do a ← readIORef x  
              b ← readIORef y  
              writeIORef x b  
              writeIORef y a
```



Example: Mutable linked lists

Singly-linked lists

```
type ListRef elem = IOREf (List elem)
data List elem    = Nil | Cons elem (ListRef elem)
```

- **IORefs** are first class citizens; they can mix and mingle
- The two types are mutually recursive
- Operations on singly-linked lists often also come in pairs defined by mutual recursion



Linked lists: length

The length of a singly-linked list (definition style)

```
length :: ListRef elem → IO Int
```

```
length ref = do { list ← readIORef ref; length' list }
```

```
length' :: List elem → IO Int
```

```
length' Nil = pure 0
```

```
length' (Cons x next) = do { n ← length next; pure (n + 1) }
```

```
type ListRef elem  
  = IORef (List elem)  
data List elem  
  = Nil  
  | Cons elem (ListRef elem)
```



Linked lists: length

The length of a singly-linked list (expression style)

```
length :: ListRef elem → IO Int
length ref = do
  list ← readIORef ref
  case list of
    Nil      → pure 0
    Cons x next → do { n ← length next; pure (n + 1) }
```

```
type ListRef elem
  = IORef (List elem)
data List elem
  = Nil
  | Cons elem (ListRef elem)
```

Note: layout-sensitive syntax and syntax using braces and semicolons can be mixed



Linked lists: concatenation

Rear of a list (last reference cell)

```
rear :: ListRef elem → IO (ListRef elem)
rear ref = do
  list ← readIORef ref
  case list of
    Nil      → pure ref
    Cons a next → rear next
```

Concatenating two singly-linked lists

```
append :: ListRef elem → ListRef elem → IO ()
append xref yref = do
  ref ← rear xref
  copy ref yref
```



Linked lists: a puzzle

What is printed?

```
puzzle = do
  x ← fromList [0..14]
  y ← fromList [15..19]
  append x y
  n1 ← length x
  print n1
  append x y
  n2 ← length x
  print n2
```



Take away

Summary

- “lazyness forces you to be pure”
- I/O actions are first-class citizens!
- In general, try to minimize the I/O part of your program

