I/O in Haskell

Prof. Susan Older

13 November 2019

Our Haskell Experience to Date: Pure Functions

We've seen pure functions:

- Behavior of a function depends *only* on its arguments
  For example: `foo "abc"` will always return the same result.
- No observable side effects, such as using I/O devices, mutation to state, etc.

Benefit: Supports assurance: it's much easier to reason mathematically about behavior of programs.

Drawback: Limits functionality: some computations may require side effects.

The Challenge: How to accommodate computational side effects while preserving benefits of pure functions?

I/O in Haskell

IO types:

For each Haskell type `t`, there is a type `IO t` whose values are:

I/O actions (or programs) that yield a result of type `t`.

When an I/O action is executed, it does two things:

1. It (possibly) performs some input/output or other side effects.
2. It produces/yields a result.

- `IO Bool`: action that may do some I/O, and then yields a `Bool` value
- `IO String`: action that may do some I/O, then yields a `String` value
- `IO ()`: action that may do some I/O, and then yields a `()` value[a]

[a] `()` is the type with the single value `()` (i.e., a 0-ary tuple).

Some Simple I/O Actions for Output

- `putChar :: Char -> IO ()`

  `putChar ch` is an I/O action that, when executed, writes `ch` to standard output

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

  `putStr cs` is an I/O action that, when executed, writes `cs` to standard output

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

  `putStrLn cs` is an I/O action that, when executed, writes `cs` (followed by newline) to standard output

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

  `print x` is an I/O action that, when executed, displays `x` to standard output

Some More Simple I/O Actions

- `getChar :: IO Char`

  `getChar` is an I/O action that, when executed, reads a character from standard input

- `getLine :: IO String`

  `getLine` is an I/O action that, when executed, reads a line (of characters) from standard input

- `return :: a -> IO a`

  `return x` is an I/O action that, when executed, performs no actual I/O but yields result `x`

Disclaimer:

Ghci will tell you that `return` has type `Monad m => a -> m a`.

I/O is an instance of a more general notion called a monad, but you don't need to know about monads to do I/O.

How to Sequence Actions

The bind operator `>>=`:

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

Informally, executing `act >>= f` proceeds as follows:

1. Perform the action `act`, which yields a result (say, `r`)
2. Apply the function `f` to `r`, which returns an action (say, `act2`)
3. Perform the action `act2`

What happens when `doubleLine` is executed?

```
doubleLine :: IO ()
doubleLine = getLine >>= (\w -> putStrLn (w++w))
```


More Examples of Sequencing

- `getChar >>= putChar :: IO ()`
- `getLine >>= (print.length) :: IO ()`
- `getLine >>= (putStrLn.reverse) :: IO ()`
- `getLine >>= (return.reverse) :: IO String`

Recall: `.` is function composition:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
g . f = \x -> g (f x)
```

- `print.length :: [a] -> IO ()`
- `putStrLn.reverse :: [Char] -> IO ()`
- `return.reverse :: [a] -> IO [a]`

Chaining: A Special Case of Sequencing

Suppose we define the following:

```
echo :: IO ()
echo = getChar >>= putChar
```

What is the type of `echo >>= echo`? Type Error

A fix:

`echo >>= (\ _ -> echo)` has type `IO ()`

Hence, Haskell's chain operator `>>` is syntactic sugar:

```
(>>) :: IO a -> IO b -> IO b
act1 >> act2 = act1 >>= (\ _ -> act2)
```

`echo >> echo` has type `IO ()`.

Chaining and do Notation

```
putNTimes :: Int -> String -> IO ()
putNTimes n str
   | n <= 1    = putStrLn str
   | otherwise = putStrLn str >> putStrLn "*****"
                              >> putNTimes (n-1) str
```

An equivalent formulation, using do notation:

```
putNTimes :: Int -> String -> IO ()
putNTimes n str
   | n <= 1    = putStrLn str
   | otherwise = do putStrLn str
                    putStrLn "*****"
                    putNTimes (n-1) str
```


One More Example

```
getAndPut :: IO ()
getAndPut = do line <- getLine
               putStrLn line
```

What's going on here?

1. When getLine is executed, it yields a String value.
2. That value gets bound to (a newly scoped) variable line, which putStrLn can use.

An equivalent formulation, using >>= instead of do:

```
getAndPut :: IO ()
getAndPut = getLine >>= (\ line -> putStrLn line)
```


Let's Write a Few Programs

(This slide intentionally left blank.)

The main Program

The main program is an I/O action:

```
main :: IO ()
main = do putStr "Enter a string: "
          resp <- getLine
          putStrLn ("Your string was: " ++ resp)
          putStrLn "\n\nGoodbye!"
```

Two options for executing main (in file io.hs) outside of Ghci:

1. You can use runhaskell or runghc:

```
runhaskell io
```

2. Create an executable file and then run it:

```
ghc --make io
./io
```

Our Own Word-Count Program

Let's write a file `ourWC.hs`:

```
wordCount :: String -> (Int, Int, Int)
wordCount inp = (l,w,c)
  where
    l = length (lines inp)
    w = length (words inp)
    c = length inp

main :: IO ()
main = do inp <- getContents
          print (wordCount inp)
```

To try it out:

```
runhaskell ourWC < sampleFile
```


Some Built-In Control Structures

`sequence`

```
sequence :: [IO a] -> IO [a]
sequence_ :: [IO a] -> IO ()
```

Compare the following:

```
sequence  (map print [1..10])
sequence_ (map print [1..10])

sequence  (replicate 5 getChar)
sequence_ (replicate 5 getChar)
```


Putting it All Together

Let's write a program that does the following:

1. Prompts the user to enter an integer (say, `n`)
2. Reads `n` floating-point numbers
3. Yields the sum of those float-point numbers

The power of `sequence`:

```
sumN :: IO Float
sumN = do putStrLn "How many numbers do you want to sum? "
          n <- getInteger
          vals <- sequence [ getFloat | x <- [1..n] ]
          return (sum vals)
```