Programming in Haskell Aug–Nov 2015

LECTURE 22

NOVEMBER 5, 2015

S P Suresh
Chennai Mathematical Institute

Till now …

- A program is a bunch of functions
- A function of type `a -> b` produces a result of type b on an argument of type `a`
- The programs are run in ghci – by invoking a function on some arguments
- ghci automatically displays the result on the screen (provided it can be shown)

User interaction

- Can we execute programs outside ghci?
- How do we let the programs interact with users?
    - Accept user inputs midway through a program execution
    - Print output and diagnostics on screen / to a file
- Can interaction with the outside world be achieved without violating the spirit of Haskell?

Standalone programs and main

- Execution of a Haskell program starts with the function `main`
- Every standalone Haskell program should have a `main` function

First program

- First compilable program

  ```
  main = putStr "Hello, world!\n"
  ```

- Save this into a file named `hw.hs`
- Compile it using the command `ghc hw.hs`
- This generates the files `hw.hi`, `hw.o` and `hw` (with execute permissions)
- Run the executable using `./hw`

ghc

- ghc is the Glasgow Haskell Compiler
- `ghci` is the interactive version of the compiler
- One can view `ghci` as an interpreter or a playground in which to test your programs
- Software intended for use by others is written as a standalone program, compiled using ghc and shipped

ghc

- Compiled versions of programs run much faster and use much less memory, compared to running them in `ghci`
- Check out commonly used compiler options using `ghc --help`
- Use `ghc --show-options` to know all options (a huge list!)
- The GHC Manual at `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/` is a comprehensive document about both `ghc` and `ghci`

Hello, world!

- `main = putStr "Hello, world!\n"`
- `putStr str` prints the string `str` on screen
- Clearly `putStr` is of type `String -> b`, for some type `b`
- The return value is not used at all, so perhaps it returns nothing of significance
- The type `()`, which consists of a single value, also denoted by `()`, can be used to model “nothing”
- So is its type `String -> ()`?

Hello, world!

- Is `putStr` of type `String -> ()`?

- But it does not return the value `()`!

- And how do we account for the **side effect** of printing something on screen?

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

-
  ```
  ghci> :t putStr "Hello, world!"
  putStr "Hello, world!" :: IO ()
  ```

IO a

- `IO` is a type constructor, just like `List` or `BTree` or `AVLTree` that we encountered in previous lectures

- `IO a` is a type whenever `a` is a type

- Recall that the value constructors and internal structure of `List`, `BTree` etc. are visible

- The internal structure and constructors of `IO` are not visible to the user

IO a

- One can understand `IO` as follows:
  `data IO a = IO (RealWorld -> (RealWorld,a))`

- So an object of `IO a` is a function which takes as input the current state of the real world, and produces a new state of the real world and a value of type `a`

- In other words, objects of `IO a` constitute both a value of type `a` and a side effect (the change in state of the world)

IO a and actions

- Technically, an object of type `IO a` is not a function but an **IO action**
- An IO action produces a side effect when its value is extracted
- Any function that produces a side effect will have return type `IO a`

putStr and main

- `putStr :: String -> IO ()`
- `putStr` takes a string as argument and returns `()`, producing a side effect when the return value is extracted
- The side effect is that of printing on screen the string provided as argument
- `main :: IO ()`
- `main` is always of type `IO a`

Side effects

- Kind of side effects
  - Printing on screen
  - Reading a user input from the terminal
  - Opening / closing a file
  - Changing a directory
  - Writing into a file
  - **Launching a missile**

putStr and putStrLn

- `putStr "Hello world!"` prints the string on the screen
- `putStrLn "Hello world!"` prints the string and a newline (`'\n'`) on the screen
- `putStrLn str` is equivalent to `putStr (str ++ "\n")`

Chaining actions

- We use the command do to chain multiple actions
-
```
main = do
    putStrLn "Hello!"
    putStrLn "What's your name?"
```
- do makes the actions take effect in sequential order, one after the other
- Indentation is important

Chaining actions

- Alternative, friendlier syntax

```
main = do {
    putStrLn "Hello!";
    putStrLn "What's your name?";
}
```

- Actions can occur inside `let`, `where` etc.

-
  ```
  main = do {act1; act2;}
    where
      act1 = putStr "Hello, "
      act2 = putStrLn "world!"
  ```

More actions

- `print :: Show a => a -> IO ()`
  Output a value of any printable type to the standard output (screen), and add a newline

- `putChar :: Char -> IO ()`
  Write the `Char` argument to the screen

- `getLine :: IO String`
  Read a line from the standard input and return it as a string

- The side effect of `getLine` is the consumption of a line of input, and the return value is a string

- `getChar :: IO Char`
  Read the next character from the standard input

Binding

- `getLine` is of type `IO String`, but is there a way to use the return value?

- We need to bind the return value to an object of type `String` and use it elsewhere

- The syntax for binding is `<-`

-
  ```
  main = do {
             putStrLn "Please type your name!";
             n <- getLine;
             putStrLn ("Hello, " ++ n);
            }
  ```

Binding

-
  ```
  main = do {
             putStrLn "Please type your name!";
             n <- getLine;
             putStrLn ("Hello, " ++ n);
            }
  ```

- This is **wrong!**

  ```
  putStrLn("Hello, " ++ getLine);
  ```

- `getLine` is not a `String`

- It is an action that returns `String`, that has to be extracted before use

getLine

-
  ```
  getLine :: IO String
  getLine =
      do {
          c <- getChar;
          if (c == '\n') then
              return "";
          else do {
                  cs <- getLine;
                  return (c:cs);
                  }
         }
  ```

Summary

- Haskell has a clean separation of pure functions and actions with side effects
- Actions are used to interact with the real world and perform input/output
- `main` is an action where the computation begins
- `ghc` can be used to compile and run programs