Programming in Haskell: Lecture 17

S P Suresh

October 14, 2019

Summary of `IO`

- **Actions** of type `IO t1`, `t1 -> IO t2`, `t1 -> t2 -> IO t3` &c.

Summary of `IO`

- **Actions** of type `IO t1`, `t1 -> IO t2`, `t1 -> t2 -> IO t3` &c.
- As opposed to **pure functions** whose type does not involve `IO`

Summary of `IO`

- **Actions** of type `IO t1`, `t1 -> IO t2`, `t1 -> t2 -> IO t3` &c.
- As opposed to **pure functions** whose type does not involve `IO`
- Actions have side effects – reading input from user and printing output to screen

Summary of `IO`

- **Actions** of type `IO t1`, `t1 -> IO t2`, `t1 -> t2 -> IO t3` &c.
- As opposed to **pure functions** whose type does not involve `IO`
- Actions have side effects – reading input from user and printing output to screen
- Actions and pure functions can be embedded inside other actions

Summary of `IO`

- **Actions** of type `IO t1`, `t1 -> IO t2`, `t1 -> t2 -> IO t3` &c.
- As opposed to **pure functions** whose type does not involve `IO`
- Actions have side effects – reading input from user and printing output to screen
- Actions and pure functions can be embedded inside other actions
- Actions cannot be embedded inside pure functions

Summary of `IO`

- Actions can be chained inside a do block

```
bigact = do {
    act1;
    act2;
    ...
    actn;
}
```

Summary of `IO`

- Actions can be chained inside a do block

```
bigact = do {
    act1;
    act2;
    ...
    actn;
}
```

- The actions are executed in order, one after the other

Summary of `IO`

- Actions can be chained inside a do block

```
bigact = do {
    act1;
    act2;
    ...
    actn;
}
```

- The actions are executed in order, one after the other
- There can be recursive calls to `bigact` inside the `do` block

Summary of `IO`

- Actions can be chained inside a do block

```
bigact = do {
    act1;
    act2;
    ...
    actn;
}
```

- The actions are executed in order, one after the other
- There can be recursive calls to `bigact` inside the `do` block
- The return type of `bigact` is the return type of `actn`

Summary of `IO`

- `main` is a distinguished action where computation begins

Summary of `IO`

- `main` is a distinguished action where computation begins
- Standalone programs should have a `main` action

Summary of `IO`

- `main` is a distinguished action where computation begins
- Standalone programs should have a `main` action
- Compiled using `ghc` and run on the terminal, outside `ghci`

Summary of `IO`

- `main` is a distinguished action where computation begins
- Standalone programs should have a `main` action
- Compiled using `ghc` and run on the terminal, outside `ghci`
- Binding the return value of an action to a name is achieved using `<-`

Summary of `IO`

- `main` is a distinguished action where computation begins
- Standalone programs should have a `main` action
- Compiled using `ghc` and run on the terminal, outside `ghci`
- Binding the return value of an action to a name is achieved using `<-`
- We use `return` to promote a value of type `a` to an action of type `IO a`

More actions

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

More actions

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

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 ()`

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 character argument to the screen

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 character argument to the screen
- `getLine :: IO String`

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

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 character 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

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 character 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
  - The return value is a string

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 character 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
  - The return value is a string
- `getChar :: IO Char`

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 character 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
  - The return value is a string
- `getChar :: IO Char`
  - Read the next character from the standard input

Functions vs. Actions

- A function that takes an integer as argument and returns an integer as result has type `Int -> Int`

Functions vs. Actions

- A function that takes an integer as argument and returns an integer as result has type `Int -> Int`
- An action that has a side effect in addition has type `Int -> IO Int`

Functions vs. Actions

- A function that takes an integer as argument and returns an integer as result has type `Int -> Int`
- An action that has a side effect in addition has type `Int -> IO Int`
- This is in contrast to a language like C or Java, where the type signatures are just `int -> int`, and any function can produce a side effect

Functions vs. Actions

- The functions we have seen till now (free of side effects) are called **pure functions**

Functions vs. Actions

- The functions we have seen till now (free of side effects) are called **pure functions**
- Their type gives all the information we need about them

Functions vs. Actions

- The functions we have seen till now (free of side effects) are called **pure functions**
- Their type gives all the information we need about them
- Invoking a function on the same arguments always yields the same result

Functions vs. Actions

- The functions we have seen till now (free of side effects) are called **pure functions**
- Their type gives all the information we need about them
- Invoking a function on the same arguments always yields the same result
- The order of evaluation of the subcomputations does not matter – Haskell takes advantage this in applying its **lazy strategy**

Functions vs. Actions

- The presence of `IO` in the type indicates that actions potentially have side effects

Functions vs. Actions

- The presence of `IO` in the type indicates that actions potentially have side effects
- External state is changed

Functions vs. Actions

- The presence of `IO` in the type indicates that actions potentially have side effects
- External state is changed
- Order of computation is important – **sequencing**

Functions vs. Actions

- Performing the same action on the same arguments twice might have different results

```
greetUser :: String -> IO ()
greetUser greeting = do {
    putStrLn "Please enter your name";
    name <- getLine;
    putStrLn ("Hi " ++ name ++ ". " ++ greeting);
}
main = do {
    greetUser "Welcome!";
    greetUser "Welcome!";
}
```

Combining pure functions and `IO` *actions*

- Haskell type system allows us to combine pure functions and actions in a safe manner

Combining pure functions and `IO` *actions*

- Haskell type system allows us to combine pure functions and actions in a safe manner
- No mechanism to execute an action inside a pure function

Combining pure functions and `IO` *actions*

- Haskell type system allows us to combine pure functions and actions in a safe manner
- No mechanism to execute an action inside a pure function
- But pure functions can be used as subroutines inside actions

Combining pure functions and `IO` *actions*

- Haskell type system allows us to combine pure functions and actions in a safe manner
- No mechanism to execute an action inside a pure function
- But pure functions can be used as subroutines inside actions
- IO is performed by an action only if it is executed from within another action

Combining pure functions and `IO` *actions*

- Haskell type system allows us to combine pure functions and actions in a safe manner
- No mechanism to execute an action inside a pure function
- But pure functions can be used as subroutines inside actions
- IO is performed by an action only if it is executed from within another action
- `main` is where all the action begins

IO example

- First item

```
main = do {
    putStrLn "Enter your name: ";
    name <- getLine;
    putStrLn $ "Hi " ++ name ++ "! Welcome to Haskell!";
}
```

IO example

- First item

```
main = do {
    putStrLn "Enter your name: ";
    name <- getLine;
    putStrLn $ "Hi " ++ name ++ "! Welcome to Haskell!";
}
```

- We would like to let the user enter their name on the same line as the prompt

IO example

- First item

```
main = do {
    putStrLn "Enter your name: ";
    name <- getLine;
    putStrLn $ "Hi " ++ name ++ "! Welcome to Haskell!";
}
```

- We would like to let the user enter their name on the same line as the prompt
- Use `putStr` instead of `putStrLn`

IO example

- First item

```
main = do {
    putStrLn "Enter your name: ";
    name <- getLine;
    putStrLn $ "Hi " ++ name ++ "! Welcome to Haskell!";
}
```

- We would like to let the user enter their name on the same line as the prompt
- Use `putStr` instead of `putStrLn`
- Works as expected in `ghci`

IO example – buffering

- But IO is usually **buffered**

IO example – buffering

- But IO is usually **buffered**
- Means that output is printed on screen only on user pressing **Enter**

IO example – buffering

- But IO is usually **buffered**
- Means that output is printed on screen only on user pressing **Enter**
- On compiling the above using `ghc` and running on the command line, we see this:

```
Suresh
Enter your name: Hi Suresh! Welcome to Haskell!
```

IO example – buffering

- But IO is usually **buffered**
- Means that output is printed on screen only on user pressing **Enter**
- On compiling the above using `ghc` and running on the command line, we see this:

```
Suresh
Enter your name: Hi Suresh! Welcome to Haskell!
```

- Solution – explicitly prohibit buffering

IO example

- First item

```
import System.IO
main = do {
    hSetBuffering stdout NoBuffering;
    putStr "Enter your name: ";
    name <- getLine;
    putStrLn $ "Hi " ++ name ++ "! Welcome to Haskell!";
}
```

IO example

- First item

```
import System.IO
main = do {
    hSetBuffering stdout NoBuffering;
    putStr "Enter your name: ";
    name <- getLine;
    putStrLn $ "Hi " ++ name ++ "! Welcome to Haskell!";
}
```

- `stdout` refers to the standard output, typically the screen

IO example

- First item

```
import System.IO
main = do {
    hSetBuffering stdout NoBuffering;
    putStr "Enter your name: ";
    name <- getLine;
    putStrLn $ "Hi " ++ name ++ "! Welcome to Haskell!";
}
```

- `stdout` refers to the standard output, typically the screen
- `import System.IO` required for `hSetBuffering`

IO example, repetition

- Read a line and print it out ten times

IO example, repetition

- Read a line and print it out ten times
- Use recursion

```
main = do {
    inp <- getLine;
    printOften 10 inp;
}
printOften :: Int -> String -> IO ()
printOften 1 str = putStrLn str
printOften n str = do {
    putStrLn str;
    printOften (n-1) str;
}
```

IO example, repetition

- How do we define `printOften 0 str`?

IO example, repetition

- How do we define `printOften 0 str`?
- Can we just define it to be `()`?

IO example, repetition

- How do we define `printOften 0 str`?
- Can we just define it to be `()`?
- But then the output would be of type `()`, not `IO ()`

IO example, repetition

- How do we define `printOften 0 str`?
- Can we just define it to be `()`?
- But then the output would be of type `()`, not `IO ()`
- Need a way to promote `()` to an object of type `IO ()`

IO example, repetition

- How do we define `printOften 0 str`?
- Can we just define it to be `()`?
- But then the output would be of type `()`, not `IO ()`
- Need a way to promote `()` to an object of type `IO ()`
- Achieved by the `return` function

IO example, repetition

- How do we define `printOften 0 str`?
- Can we just define it to be `()`?
- But then the output would be of type `()`, not `IO ()`
- Need a way to promote `()` to an object of type `IO ()`
- Achieved by the `return` function
- If `v` is a value of type `a`, `return v` is of type `IO a`

IO example, repetition

- How do we define `printOften 0 str`?
- Can we just define it to be `()`?
- But then the output would be of type `()`, not `IO ()`
- Need a way to promote `()` to an object of type `IO ()`
- Achieved by the `return` function
- If `v` is a value of type `a`, `return v` is of type `IO a`
- Not to be confused with `return` in languages like C, Java &c.

IO example, repetition

- How do we define `printOften 0 str`?
- Can we just define it to be `()`?
- But then the output would be of type `()`, not `IO ()`
- Need a way to promote `()` to an object of type `IO ()`
- Achieved by the `return` function
- If `v` is a value of type `a`, `return v` is of type `IO a`
- Not to be confused with `return` in languages like C, Java &c.
  - In imperative languages, `return` is used to return control to the caller

IO example, repetition

- How do we define `printOften 0 str`?
- Can we just define it to be `()`?
- But then the output would be of type `()`, not `IO ()`
- Need a way to promote `()` to an object of type `IO ()`
- Achieved by the `return` function
- If `v` is a value of type `a`, `return v` is of type `IO a`
- Not to be confused with `return` in languages like C, Java &c.
  - In imperative languages, `return` is used to return control to the caller
  - Here it just wraps an action around a value

IO example, repetition

- How do we define `printOften 0 str`?
- Can we just define it to be `()`?
- But then the output would be of type `()`, not `IO ()`
- Need a way to promote `()` to an object of type `IO ()`
- Achieved by the `return` function
- If `v` is a value of type `a`, `return v` is of type `IO a`
- Not to be confused with `return` in languages like C, Java &c.
  - In imperative languages, `return` is used to return control to the caller
  - Here it just wraps an action around a value
  - `x <- return e; act;` is the same as `let x = e in act;`

IO example, repetition

- Read a line and print it out ten times

```
main = do {
    inp <- getLine;
    printOften 10 inp;
}
printOften :: Int -> String -> IO ()
printOften 0 str = return ()
printOften n str = do {
    putStrLn str;
    printOften (n-1) str;
}
```

IO example, repetition – `getLine`

- Here is a possible implementation of `getLine`

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

IO example, repetition – `getLine`

- Here is a possible implementation of `getLine`

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

- Note the use of `return` and the recursion

Repetition and IO, another example

- Repeat an IO action n times

```
ntimes :: Int -> IO () -> IO ()
ntimes 0 a = return ()
ntimes n a = do { a; ntimes (n-1) a;}
```

Repetition and IO, another example

- Repeat an IO action n times

```
ntimes :: Int -> IO () -> IO ()
ntimes 0 a = return ()
ntimes n a = do { a; ntimes (n-1) a;}
```

- Read and print 100 lines

```
main = ntimes 100 $ do {
    inp <- getLine;
    putStrLn inp;
}
```

Reading other types

- The function `readLn` reads the value of any type `a` that is an instance of the typeclass `Read`

```
readLn :: Read a => IO a
```

Reading other types

- The function `readLn` reads the value of any type `a` that is an instance of the typeclass `Read`

```
readLn :: Read a => IO a
```

- All basic types (`Int`, `Bool`, `Char`, &c.) are instances of `Read`

Reading other types

- The function `readLn` reads the value of any type `a` that is an instance of the typeclass `Read`

```
readLn :: Read a => IO a
```

- All basic types (`Int`, `Bool`, `Char`, &c.) are instances of `Read`
- Basic type constructors also preserve readability

Reading other types

- The function `readLn` reads the value of any type `a` that is an instance of the typeclass `Read`

```
readLn :: Read a => IO a
```

- All basic types (`Int`, `Bool`, `Char`, &c.) are instances of `Read`
- Basic type constructors also preserve readability
- `[Int]`, `(Int, Bool, Char)`, &c. are also instances of `Read`

Reading other types

- The function `readLn` reads the value of any type `a` that is an instance of the typeclass `Read`

```
readLn :: Read a => IO a
```

- All basic types (`Int`, `Bool`, `Char`, &c.) are instances of `Read`
- Basic type constructors also preserve readability
- `[Int]`, `(Int, Bool, Char)`, &c. are also instances of `Read`
- Syntax to read an integer

```
inp <- readLn :: IO Int
```

Reading integers, example

- Read a list of non-negative integers (one on each line and terminated by a negative integer)

Reading integers, example

- Read a list of non-negative integers (one on each line and terminated by a negative integer)
- Print out the list at the end

```
main = do {ls <- readIntList; print ls;}

readIntList :: IO [Int]
readIntList = do {
    inp <- readLn :: IO Int;
    if (inp < 0) then return [];
    else do {l <- readIntList; return (inp:l);}
}
```

Rudimentary file IO

- Simplest way to read from files and write into files is by **input/ output redirection**

Rudimentary file IO

- Simplest way to read from files and write into files is by **input/ output redirection**
- Usual input is from **standard input** (which is the keyboard)

Rudimentary file IO

- Simplest way to read from files and write into files is by **input/ output redirection**
- Usual input is from **standard input** (which is the keyboard)
- Read input from a file by input redirection

```
$ ./myprogram < inputfile
```

Rudimentary file IO

- Simplest way to read from files and write into files is by **input/ output redirection**
- Usual input is from **standard input** (which is the keyboard)
- Read input from a file by input redirection

```
$ ./myprogram < inputfile
```

- Usual output is to **standard output** (which is the screen)

Rudimentary file IO

- Simplest way to read from files and write into files is by **input/ output redirection**
- Usual input is from **standard input** (which is the keyboard)
- Read input from a file by input redirection

```
$ ./myprogram < inputfile
```

- Usual output is to **standard output** (which is the screen)
- Send output to a file by output redirection

```
$ ./myprogram > outputfile
```

Rudimentary file IO

- Simplest way to read from files and write into files is by **input/ output redirection**
- Usual input is from **standard input** (which is the keyboard)
- Read input from a file by input redirection

```
$ ./myprogram < inputfile
```

- Usual output is to **standard output** (which is the screen)
- Send output to a file by output redirection

```
$ ./myprogram > outputfile
```

- Can combine the two:

```
$ ./myprogram < inputfile > outputfile
```