Programming in Haskell
Aug–Nov 2015

LECTURE 23

NOVEMBER 12, 2015

S P Suresh
Chennai Mathematical Institute

Summary of IO

- Actions of type IO t1, t1 -> IO t2, t1 -> t2 -> IO t3 etc.
- 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 actions
- Actions cannot be embedded inside pure functions

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
- 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 ()`
  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

getLine

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

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 (that are 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 utilizes this in applying its lazy strategy

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!";
            }
  ```

- The two actions print different things on the screen, depending on the name that is input by the user

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, even though 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

- Read a line and print it out as many times as its length

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

- What if the user inputs the empty string?

return

- What if the user inputs the empty string?
- How do we define `printOften 0 str`?
- Can we just define it to be `()`?
- But then the output type would be `()`, 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, fixed

- Read a line and print it out as many times as its length

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

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 act
   where
      act = do {
                inp <- getLine;
                putStrLn inp;
               }
```

Reading other types

- The function `readLn` reads the value of any type `a` that is instance of the typeclass `Read`
- `readLn :: Read a => IO a`
- All basic types (`Int`, `Bool`, `Char`, ...) are instances of `Read`
- Basic type constructors also preserve readability – `[Int]`, `(Int, Char, Bool)`, etc are also instances of `Read`
- Syntax to read an integer
  `inp <- readLn :: IO Int`

Another IO example

- Read a list of integers (one on each line and terminated by -1) into a list, and print the list

```
main = do {
            ls <- readList [];
            putStrLn (show (reverse ls));
          }
readList :: [Int] -> IO [Int]
readList l = do {
                 inp <- readLn :: IO Int;
                 if (inp == -1) then
                     return l;
                 else readList (inp:l);
                }
```

The bind operator

- Two fundamental functions used to construct and combine actions
- `return :: a -> IO a`
  `(>>=)  :: IO a -> (a -> IO b) -> IO b`
- Execution of `act1 >>= act2`
  - executes `act1`
  - unboxes and extracts the return value (of type `a`)
  - executes `act2`, perhaps using the previously extracted value
  - the result value of the combined action is the result of `act2`

The bind operator

- Actually, `return` and `(>>=)` are functions common to all **monads**
- IO is an example of a monad
- Many other type constructs we have already seen produce monads – [], Maybe etc.
- More on monads in the next lecture
- Functions like readLn, putStrLn, print etc. are specific to the IO monad

Using bind

- Read a line and print it

```
getLine >>= putStrLn
```

- Read a line and print its length

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

- The result value of `getLine` has to be used by `print`

```
getLine >>= (\str ->
             print (length str)
             )
```

Using bind

- Read a line and print its length twice

```
getLine >>= (\str ->
             print (length str) >>=
             print (length str)
             )
```

- This produces a type error

- The second `(>>=)` expects a second argument of type `() -> IO c`, since `print x` is of type `IO ()`

-
```
getLine >>= (\str -> print (length str) >>=
                     (\str' -> print (length str)))
```

Bind without arguments

- A simpler version of the previous action:

```
getLine >>= (\str ->
              print (length str) >>
              print (length str)
            )
```

- If we do not want to unbox and use the result of the preceding action, we use `(>>)`

- The following are equivalent:

  `act1 >> act2`

  `act1 >>= (\n -> act2)`, where the name `n` is not used in `act2`

Bind without arguments

- Given the definitions

  `f x = exp1`

  `g y = exp2` (`y` does not occur in `exp2`)

  `g (f 10)` does not evaluate `f 10`

- But given actions `act1` and `act2`, `act1 >> act2` does execute `act1` and `act2` in that order, even though its return value is not used further

- The operators `(>>=)` and `(>>)` force the execution of both the arguments, the left one first and then the right one

do is syntactic sugar

- The do blocks introduced earlier can be translated in terms of (>>=) and (>>)

- A single action needs no do

```
do {
   act;          translates to          act
   }
```

do is syntactic sugar

- If there is no `<-` in the first action, we use `>>`

```
do {                          act1 >> do {
   act1;          translates to                S
   S                                           }
   }
```

- If there is `<-` in the first action, we use `>>=`

```
do {                                  act1 >>= \n ->
   n <- act1;     translates to                 do {
   S                                             S
   }                                             }
```

Rudimentary file IO

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

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

- Write output to file (rather than to standard output, which is the screen output) by output redirection

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

- Can combine the two: `./myprogram < inputfile > outputfile`

Summary

- Actions are used to interact with the real world and perform input/output
- IO is an example of a **Monad**, about which we will see more later
- The functions `return` and `(>>=)` are common to all monads
- All `do` blocks can be translated using `(>>=)` and `(>>)`
- Rudimentary file IO is done using input/output redirection