Session 10

Programming with actions

This session

- at last: input and output in a functional language
  - resolving the paradox using a type of *actions*, which are passed to and executed by the runtime system
- programming with actions
  - `do`-notation as a convenient notation
  - so we can write Haskell code that looks like imperative code (but we prefer not to)
- control structures
  - since actions are values, functions can operate on them
  - a peek at the next level of abstraction (type constructors)

The problem

- In a pure functional language:
  - the only meaning of an expression is its value
  - functions always produce the same results for the same inputs
- There can be no value **`getInt`** of an integer read from the console: there can be no meaning for

```
getInt + getInt
```

- There can be no function **`putInt`** that writes an integer: there can be no meaning for

```
(putInt 3, putInt 4)
```

- similarly for any other interaction with the environment.

Actions

The solution: actions

- At the top level, supply an expression of the abstract type `IO a`, which consists of instructions to the system.
- Primitives for construction such actions, e.g.

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

- Normally GHCi prints the value of any expression typed at the prompt, but if the expression has type `IO t`, GHCi carries out the action.

interpreter
```
"Hello\nworld\n"
:t putStr
:t putStr "Hello\nworld\n"
putStr "Hello\nworld\n"
```

Some output actions

Prelude
```
putStr  :: String -> IO ()
putChar :: Char -> IO ()

type FilePath = String

writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

Things to try:

interpreter
```
writeFile "out.txt" "Hello\nworld\n"
appendFile "out.txt" (unlines (map show [1..100]))
```

These actions produce no output in the interpreter. To see the effects, you'll have to switch out of GHCi.

Actions on files and directories

These functions are in the `System.Directory` library module:

System.Directory
```
removeFile :: FilePath -> IO ()
renameFile :: FilePath -> FilePath -> IO ()

createDirectory :: FilePath -> IO ()
removeDirectory :: FilePath -> IO ()
renameDirectory :: FilePath -> FilePath -> IO ()
```

Things to try:

interpreter

```
:m + System.Directory
writeFile "out.txt" "Hello\n"
renameFile "out.txt" "new.txt"
removeFile "new.txt"
```

Again, you'll have to monitor these actions outside the interpreter.

Sequencing of actions

An operation to construct actions:

Prelude

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

x **>>** y is an action that says: perform x, then perform y.

interpreter

```
putStr "Hello" >> putChar '\n' >> putStr "world\n"
```

From the **Prelude**:

Prelude

```
putStrLn   :: String -> IO ()
putStrLn s =  putStr s >> putChar '\n'

print      :: Show a => a -> IO ()
print x    =  putStrLn (show x)
```

interpreter

```
putStrLn "Hello" >> putStrLn "world"
print (2+3)
```

Input actions

Actions that produce values

For example, input actions:

Prelude

```
getChar :: IO Char
getLine :: IO String
readFile :: FilePath -> IO String
```

We would like to pass these values to other functions, like

Prelude

```
putStrLn :: String -> IO ()
putStrLn . reverse :: String -> IO ()
putChar :: Char -> IO ()
```

For this, we use another operation

Prelude

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

x **>>=** f says: perform x, and if this yields v, then perform f v. (f is a callback)

Examples of sequencing

`>>` is a special case:

Prelude

```
x >> y = x >>= \ _ -> y
```

More examples:

interpreter

```
getLine >>= putStrLn
getLine >>= (putStrLn . reverse)
getLine >>= \s -> putStrLn (reverse s)
```

Alternative syntax using layout:

```
do
   s <- getLine
   putStrLn (reverse s)
```

or (using explicit markers to get a one-liner):

interpreter

```
do { s <- getLine; putStrLn (reverse s) }
```

Do notation

`do`-notation is just an abbreviation:

```
do { p <- e ; A }   =   e >>= \ p -> do { A }
do { e ; A }        =   e >> do { A }
do { e }            =   e
```

But it is very convenient:

```
main :: IO ()
main = do
    s1 <- getLine
    s2 <- getLine
    putStrLn (reverse s1)
    putStrLn (reverse s2)
```

Note: `<-` is not assignment; it's more like declaration + initialization.

Typical top level of a Haskell program

```
module Main where

import Maze

main :: IO ()
main = do
    input <- readFile "maze.txt"
    putStrLn ("Minimum steps: " ++ show (solve (readMaze input)))
```

Notes:

- the body of a **do** expression is a sequence of *statements*.
- a **<-** statement introduces a variable and gives it the value produced by running an action.
- good practice is to do as much as possible in non-**IO** code.

Programming with actions

Defining new actions

The **Prelude** includes an action

Prelude

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

that reads a line, parsing it according to the type involved, e.g.

```
getInt :: IO Int
getInt = readLn
```

A program using **getInt**:

```
main :: IO ()
main = do
    putStrLn "Enter two numbers"
    x <- getInt
    y <- getInt
    putStrLn ("The sum is " ++ show (x+y))
```

Actions that return a value

The **Prelude** includes a function

Prelude

```
return :: a -> IO a
```

It is useful when building actions:

```
getSum :: IO Int
getSum = do
    x <- getInt
    y <- getInt
    return (x+y)
```

An imperative program

The guessing game from the C++ module in Haskell:

```
maxValue :: Int
maxValue = 100

main :: IO ()
main = do
    n <- randomIO    -- from System.Random
    guessingGame (n `mod` maxValue + 1)
```

```
guessingGame :: Int -> IO ()
guessingGame target = do
    putStr $ "Guess a number between 1 and " ++ show maxValue ++ ":
        "
    guesses target 1
```

The `guesses` function

This recursive function gives the effect of a loop:

```
guesses :: Int -> Int -> IO ()
guesses target nguesses = do
    guess <- getInt
    if guess == target then
        putStrLn $ "Correct in " ++ show nguesses ++ " guesses"
    else do
        putStr $ if guess > target
            then "Too high! " else "Too low! "
        putStr "Guess again: "
        guesses target (nguesses+1)
```

Here we have `if` expressions on `IO` actions and on strings.

Control structures

Building control structures

We can have a list of actions, and execute them in order:

Prelude

```
sequence_ :: [IO a] -> IO ()
sequence_ [] = return ()
sequence_ (a:as) = a >> sequence_ as
```

For example:

Prelude

```
putStr :: String -> IO ()
putStr s = sequence_ [putChar c | c <- s]
```

Generalize:

Prelude

```
mapM_ :: (a -> IO b) -> [a] -> IO ()
mapM_ f as = sequence_ (map f as)
```

so that `putStr = mapM_ putChar`

Generalizing from `IO`

There are two kinds of built-in operations on the `IO` type:

- I/O primitives like `getChar`, `putChar`, etc
- Glue primitives:

Prelude

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

(>>)   :: IO a -> IO b -> IO b
x >> y  = x >>= \ _ -> y
```

These are used in building the general control structures, but other types support these operations too, representing other kinds of action. Mathematicians call such types *monads*.

Type constructors

Type constructors like **`Maybe`**, **`[]`**, **`LTree`**, **`Parser`** and **`IO`** can be placed in a hierarchy of classes:

Functor: types with a map-like operation

Prelude

```
(<$>) :: (a -> b) -> f a -> f b
```

Applicative: **`Functor`** plus

Prelude

```
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

Monad: **`Applicative`** plus **`return`** = **`pure`** and

Prelude

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

There are useful functions parameterized by each of these.

Don't be intimidated

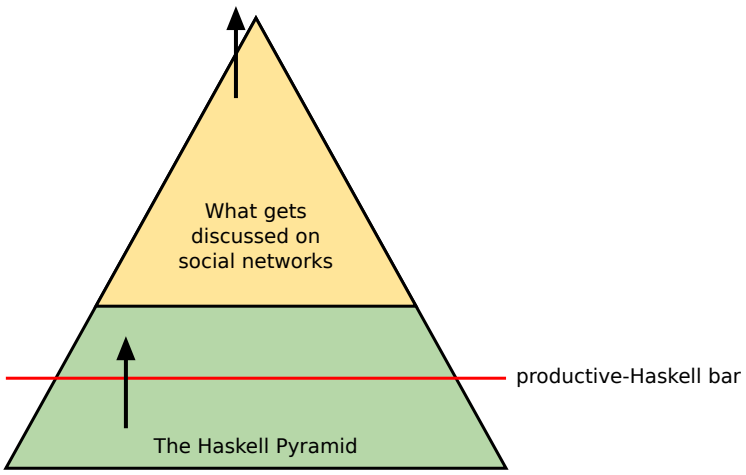

Diagram by Lucas Di Cioccio

That's all folks

Functional languages use a very different programming model, with powerful abstraction (generalization) mechanisms:

- higher-order functions
- parametric polymorphism
- constrained parametric polymorphism (Haskell's type classes)
- algebraic data types

but they can also be used for general purpose programming, by wrapping **`IO`** actions around the outside.

Exercises

1. Write a program that will prompt the user, read a line of input, and then report on whether the line is a palindrome.
2. This library function reads the entire contents of a named file:

   `System.Directory`

   ```
   readFile :: FilePath -> IO String
   ```

   Use this function to write an IO function that prints the lines of a file in reverse order.
3. The following function in the **`System.Directory`** library module gets the contents of a directory:

   `System.Directory`

   ```
   getDirectoryContents :: FilePath -> IO [FilePath]
   ```

   Write an IO action to print the contents of the current directory.
4. Using the **`getDirectoryContents`** library function, write an action to print the contents of all the files in the current directory. Try to do this without using recursion.
5. Write a function

   ```
   repeatIO :: Int -> IO a -> IO ()
   ```

   that repeats an action n times.
6. Write a program that reads lines until a palindrome is entered. Add prompts and explanations for the user.