CS 457/557: Functional Languages

I/O Actions in Haskell

Mark P Jones
Portland State University

Question:

If functional programs don’t have any side-effects, then how can we ever do anything useful?

I/O: A quick overview

Computing by calculating:

- 1 + 3
- take 32 (iterate (2*) 1)
- color red (translate (1,2) (circle 3))
- (leftTree `beside` rightTree)
- getChar >>= putChar

Demo:

… of Mac OS X Automator …

!!!

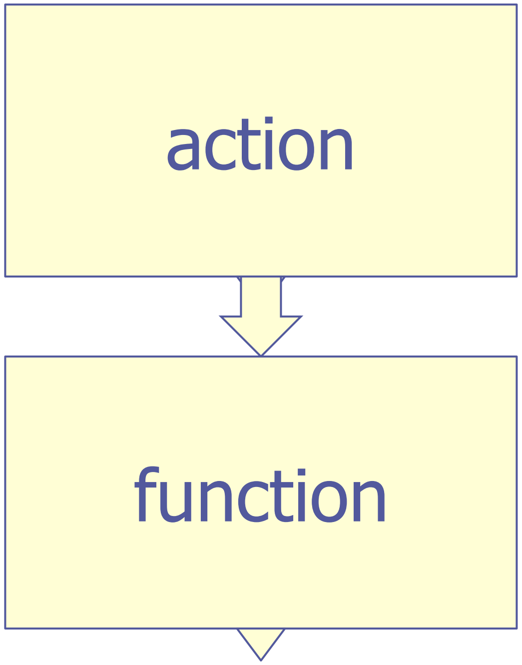
???

IO Actions:

action :: IO a

- An IO action is a value of type IO T
- T is the type of values that it produces

IO Actions:



action :: IO a

function :: a -> IO b

If action :: IO a and function :: a -> IO b

then action >>= function :: IO b

The New Haskell Logo:

Building Blocks:

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

p >> q is an I/O action in which the output of p is ignored by q

p >> q == p >>= \x -> q

(where x does not appear in q)

Building Blocks:

`return :: a -> IO a`

An I/O action that returns its input with no actual I/O behavior

Building Blocks:

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

An action inIO f applies the function f to each input of type a and produces outputs of type b as its results

Building Blocks:

```
mapM :: (a -> IO b)
                 -> [a] -> IO [b]
```

An action **mapM f** takes a list of inputs of type **[a]** as its input, runs the action **f** on each element in turn, and produces a list of outputs of type **[b]**

Building Blocks:

```
mapM_  ::  (a  ->  IO  b)
                          ->  [a]  ->  IO()
```

An action mapM_ f takes a list of inputs of type [a] as its input, runs the action f on each element in turn, and produces a result of type () as output

Terminal Output:

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

An action putStr s takes a String input and outputs it on the terminal producing a result of type ()

putStrLn s does the same thing but adds a trailing new line

Terminal Output:

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

A print action takes a value whose type is in Show and outputs a corresponding String on the terminal

Special Treatment of IO:

- The main function in every Haskell program is expected to have type IO ()
- If you write an expression of type IO t at the Hugs prompt, it will be evaluated as a program and the result discarded
- If you write an expression of some other type at the Hugs prompt, it will be turned in to an IO program using:

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

- If you write an expression e of type IO t at the GHCi prompt, it will treat it as e >>= print

Web Actions:

The WebActions module provides the following I/O actions:

```
getText         :: URL -> IO String
getByteString   :: URL -> IO ByteString
writeByteString :: String -> ByteString -> IO ()
downloadTo      :: FilePath -> URL -> IO ()
getTags         :: URL -> IO [Tag]
getHrefs        :: URL -> IO [URL]
getHTML         :: URL -> IO [TagTree]
getXML          :: URL -> IO [Content]
```

Viewing a Webpage:

```
return url
     >>= getText
     >>= putStr
```

Counting Characters:

```
return url

    >>= getText

    >>= inIO length

    >>= print
```

Counting Lines:

```
return url
    >>= getText
    >>= inIO (length . lines)
    >>= print
```

Viewing a Webpage as Tags:

```
return url

    >>= getTags

    >>= inIO (unlines . map show)

    >>= putStr
```

Extracting Hyper-references:

```
getHrefs   :: URL -> IO [URL]
getHrefs url
 = getTags url >>= \ts ->
    return [ link |
                (TagOpen "a" attrs) <- ts,
                ("href", link) <- attrs ]
```

Downloading From a Webpage:

```
return url
    >>= getHrefs
    >>= inIO (filter (isSuffixOf "hs"))
    >>= mapM_ (downloadTo "source")
```

Implementing downloadTo:

```
downloadTo :: FilePath -> URL -> IO ()
downloadTo dir url
 = getByteString url
   >>= writeByteString (dir </> urlName url)

urlName    :: String -> String
urlName     = reverse
            . takeWhile ('/'/=)
            . reverse
```

Visualizing a Webpage:

```
return url
    >>= getTags
    >>= inIO tagTree
    >>= inIO (listToDot "root")
    >>= writeFile "tree.dot"
```

IOActions Primitives:

```
putChar     :: Char   -> IO ()
putStr      :: String -> IO ()
putStrLn    :: String -> IO ()
print       :: Show a => a -> IO ()
getChar     :: IO Char
getLine     :: IO String
getContents :: IO String
readFile    :: String -> IO String
writeFile   :: String -> IO ()
```

… continued:

```
getDirectoryContents :: FilePath -> IO [FilePath]
getDirectoryPaths    :: FilePath -> IO [FilePath]
getCurrentDirectory  :: IO FilePath
getHomeDirectory     :: IO FilePath
doesFileExist        :: FilePath -> IO Bool
doesDirectoryExist   :: FilePath -> IO Bool
createDirectory      :: FilePath -> IO ()
getFiles             :: FilePath -> IO [FilePath]
getDirectories       :: FilePath -> IO [FilePath]
getArgs              :: IO [String]
getProgName          :: IO String
getEnv               :: String -> IO String
runCommand :: String -> FilePath -> IO ExitCode
```

Exercises:

- Load up IOActions.hs, and write IO Actions to answer the following:
  - How many Haskell source files are there in the current directory?
  - How many lines of Haskell source code are in the current directory?
  - What is the largest Haskell source file in the current directory
  - Copy the largest Haskell source file in the current directory into Largest.hs

Visualizing a File System:

```
data FileSystem = File FilePath
                | Folder FilePath [FileSystem]
                | Foldep FilePath
                  deriving Show

instance Tree FileSystem where …
Instance LabeledTree FileSystem where …
```

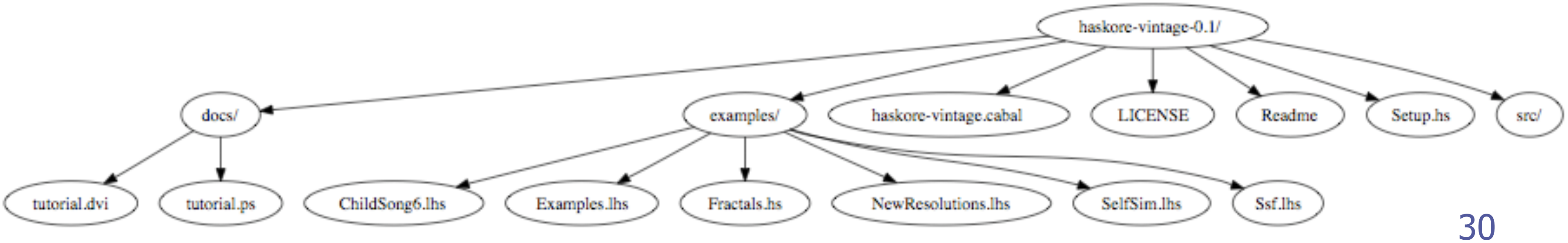
… continued:

```
getFileSystemDir :: Int -> FilePath -> FilePath -> IO FileSystem
getFileSystemDir n path name
 | n < 1     = return (Foldep name)
 | otherwise = getDirectoryContents path
               >>= inIO (filter (not . dotFile))
               >>= mapM (getFileSystemIn (n-1) path)
               >>= inIO (Folder name)

getFileSystemIn :: Int -> FilePath -> FilePath -> IO FileSystem
getFileSystemIn n parent child
 = doesDirectoryExist path
   >>= \b-> case b of
        True  -> getFileSystemDir n path child
        False -> return (File child)
   where path = parent </> child
```

Visualizing a FileSystem:

```
return "haskore-vintage-0.1"
  >>= getFileSystem 4
  >>= inIO toDot
  >>= writeFile "tree.dot”
```



Alternative Notation:

- The pipelined style for writing IO Actions isn’t always so convenient:
  - Need to refer to an input at multiple stages of a pipeline?
  - Non-linear flow (error handling)?
  - Recursion? Loops?
  - Shorter lines?

"do-notation":

- Syntactic sugar for writing IO actions:

```
do  p1
    p2
    …
    pn
```

is equivalent to:

p_1 >> p_2 >> … >> p_n

and can also be written:

do p_1; p_2; …; p_n or **do** { p_1; p_2; …; p_n }

Extending “do-notation”:

We can bind the results produced by IO actions variables using an extended form of do-notation. For example:

do x_1 <- p_1
$\ldots$
x_n <- p_n
q

all “generators” should have the same indentation

last item must be an expression

is equivalent to:

p_1 >>= \ x_1 ->
$\ldots$
p_n >>= \ x_n ->
q

variables introduced in a generator are in scope for the rest of the expression

The “v <-” portion of a generator is optional and defaults to “_ <-” if

Defining mapM and mapM_:

```
mapM_          :: (a -> IO b) -> [a] -> IO ()
mapM_ f []      = return ()
mapM_ f (x:xs)  = f x
                  >> mapM_ f xs

mapM           :: (a->IO b) -> [a]->IO [b]
mapM f []       = return []
mapM f (x:xs)   = f x            >>= \y ->
                  mapM f xs      >>= \ys->
                  return (y:ys)
```

Defining mapM and mapM_:

```
mapM_            :: (a -> IO b) -> [a] -> IO ()
mapM_ f []        = return ()
mapM_ f (x:xs)    = do f x
                       mapM_ f xs

mapM             :: (a->IO b) -> [a]->IO [b]
mapM f []         = return []
mapM f (x:xs)     = do y  <- f x
                       ys <- mapM f xs
                       return (y:ys)
```

More examples: getChar

- A simple primitive for reading a single character:

```
getChar :: IO Char
```

- A simple example:

```
echo :: IO a
echo = do c <- getChar
          putChar c
          echo
```

Reading a Complete Line:

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

Alternative:

```
getLine     :: IO String
getLine     = loop []

loop        :: String -> IO String
loop cs     = do c <- getChar
                 case c of
                   '\n' -> return (reverse cs)
                   '\b' -> case cs of
                             []     -> loop cs
                             (c:cs) -> loop cs
                   c    -> loop (c:cs)
```

There is No Escape!

- There are plenty of ways to construct expressions of type IO t
- Once a program is “tainted” with IO, there is no way to “shake it off”
- For example, there is no primitive of type IO t -> t that runs a program and returns its result

The Real Primitives:

- Many of the I/O functions that we’ve introduced can be defined in terms of other I/O functions
- The fundamental primitives are:

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

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

Generalizing …

- We can define versions of return and (>>=) for other types:

```
return    :: a -> List a
return x  = [x]

(>>=)     :: List a -> (a -> List b) -> List b
xs >>= f = [ y | x <- xs, y <- f x ]
```

- I can feel a type class coming on …

Further Reading:

- “Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell” Simon Peyton Jones, 2005
- “Imperative Functional Programming” Simon Peyton Jones and Philip Wadler, POPL 1993