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 ...

!!!

???
An IO action is a value of type IO T
T is the type of values that it produces
IO Actions:

If action :: IO a and function :: a -> IO b then action >>= function :: IO b
The New Haskell Logo:
Building Blocks:

\[( \ggg ) :: \text{IO } a \rightarrow \text{IO } b \rightarrow \text{IO } b \]

\( p \ggg q \) is an I/O action in which the output of \( p \) is ignored by \( q \)

\[ p \ggg q = \equiv p \gggg \lambda x \rightarrow q \]

(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:

\[
\text{inIO :: (a} \rightarrow \text{b)} \rightarrow \text{a} \rightarrow \text{IO b}
\]

An action \text{inIO f} applies the function \text{f} to each input of type \text{a} and produces outputs of type \text{b} as its results.
Building Blocks:

\[\text{mapM} :: (a \rightarrow \text{IO} \ b) \rightarrow [a] \rightarrow \text{IO} \ [b]\]

An action \text{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:

\[ \text{mapM}_\_ :: (a \rightarrow \text{IO } b) \rightarrow [a] \rightarrow \text{IO}() \]

An action \text{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:

\[ \text{putStr} :: \text{String} \rightarrow \text{IO} () \]
\[ \text{putStrLn} :: \text{String} \rightarrow \text{IO} () \]

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

\[ \text{putStrLn} \ s \ \text{does} \ \text{the} \ \text{same} \ \text{thing} \ \text{but} \ \text{adds} \ \text{a} \ \text{trailing} \ \text{new} \ \text{line} \]
Terminal Output:

\[
\text{print} :: \text{Show}\ a \Rightarrow a \rightarrow \text{IO}\ ()
\]

A \text{print} action takes a value whose type is in \text{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:
  
  ```haskell
  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:

```python
return url

>>> = getText

>>> = putStr
```
Counting Characters:

```python
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:

\[
\begin{align*}
\text{return url} \\
\text{>>> = getHrefs} \\
\text{>>> = inIO (filter (isSuffixOf "hs"))} \\
\text{>>> = mapM_ (downloadTo "source")}
\end{align*}
\]
Implementing downloadTo:

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

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

```python
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 (Folder 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:

```haskell
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:

\[
\text{do } p_1 \\
\quad p_2 \\
\quad \vdots \\
\quad p_n
\]

is equivalent to:

\[
p_1 \gg p_2 \gg \ldots \gg p_n
\]

and can also be written:

\[
\text{do } p_1; p_2; \ldots; p_n \quad \text{or} \quad \text{do } \{ p_1; p_2; \ldots; p_n \}
\]
Extending “do-notation”:

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

\[
\textbf{do } x_1 \leftarrow p_1 \\
... \\
x_n \leftarrow p_n \\
q
\]

is equivalent to:

\[
p_1 >>= \backslash x_1 \rightarrow \\
... \\
p_n >>= \backslash x_n \rightarrow \\
q
\]

- All “generators” should have the same indentation.
- The last item must be an expression.
- 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_:

\[
\begin{align*}
\text{mapM}_\_ & \quad :: (a \to \text{IO } b) \to [a] \to \text{IO } () \\
\text{mapM}_\_ f \; [] & \quad = \text{return } () \\
\text{mapM}_\_ f \; (x:xs) & \quad = f \; x \\
& \quad >\> \text{mapM}_\_ f \; xs
\end{align*}
\]

\[
\begin{align*}
\text{mapM} & \quad :: (a\to\text{IO } b) \to [a]\to\text{IO } [b] \\
\text{mapM } f \; [] & \quad = \text{return } [] \\
\text{mapM } f \; (x:xs) & \quad = f \; x \\
& \quad >\> \\text{mapM } f \; xs \\
& \quad >\> \\text{\textbackslash y } \to \\
& \quad \text{mapM } f \; xs \\
& \quad >\> \\text{\textbackslash ys } \to \\
& \quad \text{return } (y:ys)
\end{align*}
\]
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: \texttt{getChar}

- A simple primitive for reading a single character:
  \[
  \texttt{getChar} :: \texttt{IO Char}
  \]

- A simple example:
  \[
  \texttt{echo} :: \texttt{IO a}
  \]
  \[
  \texttt{echo} = \texttt{do} \ c \leftarrow \texttt{getChar} \\
  \quad \texttt{putChar} \ c \\
  \quad \texttt{echo}
  \]
Reading a Complete Line:

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

\texttt{getLine} :: \texttt{IO String}
\texttt{getLine} = \texttt{loop \ [\]}

\texttt{loop} :: \texttt{String -> IO String}
\texttt{loop \ cs} = \texttt{do \ c <- getChar}
  \texttt{case\ c\ of}
  \texttt{\'\\n\' -> return \ (reverse \ cs)}
  \texttt{\'\\b\' -> case\ cs\ of}
    \texttt{[] -> loop\ cs}
    \texttt{(c:cs) -> loop\ cs}
  \texttt{c -> loop\ (c:cs)}
There is No Escape!

- There are plenty of ways to construct expressions of type \( \text{IO} \ t \)

- Once a program is “tainted” with \( \text{IO} \), there is no way to “shake it off”

- For example, there is no primitive of type \( \text{IO} \ t \rightarrow t \) 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`

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

\[
\text{return} \quad :: \quad a \rightarrow \text{List} \ a \\
\text{return} \ x \quad = \quad [x] \\
\quad (\text{>>=}) \quad :: \quad \text{List} \ a \rightarrow (a \rightarrow \text{List} \ b) \rightarrow \text{List} \ b \\
x_\text{s} \text{ >>=} \ f \quad = \quad [y \mid x \leftarrow x_\text{s}, y \leftarrow f \ x] \\
\]

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