CS 457/557: Functional Languages

Lecture 8: I/O Actions in Haskell

Mark P Jones and Andrew Tolmach
Portland State University

Side-effects considered harmful

- We define fst (x,y) = x
- But is fst (print 1, print 2) the same as print 1?
- Suppose that your C/C++ code calls a function int f(int n); What might happen?

Both purity and utility?

- Sometimes we need our programs to have effects on the real world
  - reading, printing
  - drawing a picture
  - controlling a robot
  - etc.
- But "effectful" operations don't mix well with Haskell's lazy evaluation
  - Evaluation order is complex and hard to predict

Example: Tracing

- The Debug.Trace module provides a way to wrap an expression with a string to be printed when that expression is evaluated
  - `trace :: String -> a -> a`
- Useful for “stick in a print statement” style of debugging
- Or is it?

What gets printed?

```
f x = trace "there" (x+1)
g x = x + trace "there" 1
h x = if x > 0
      then trace "there" 1
      else 2

f (trace "here" 1)
g (trace "here" 1)
h (trace "here" 1)
```

Violating assumptions of "computation by calculation"

```
c = x + x
 where x = trace "x" (length "abc")
```

```
c = trace "x" (length "abc") +
    trace "x" (length "abc")
```

Question

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

Answer

Functional program can evaluate to an IO **action** that performs IO when executed

We use the type system to separate “pure values” from “worldly actions”

IO Actions

action :: IO a

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

Built in IO Actions

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

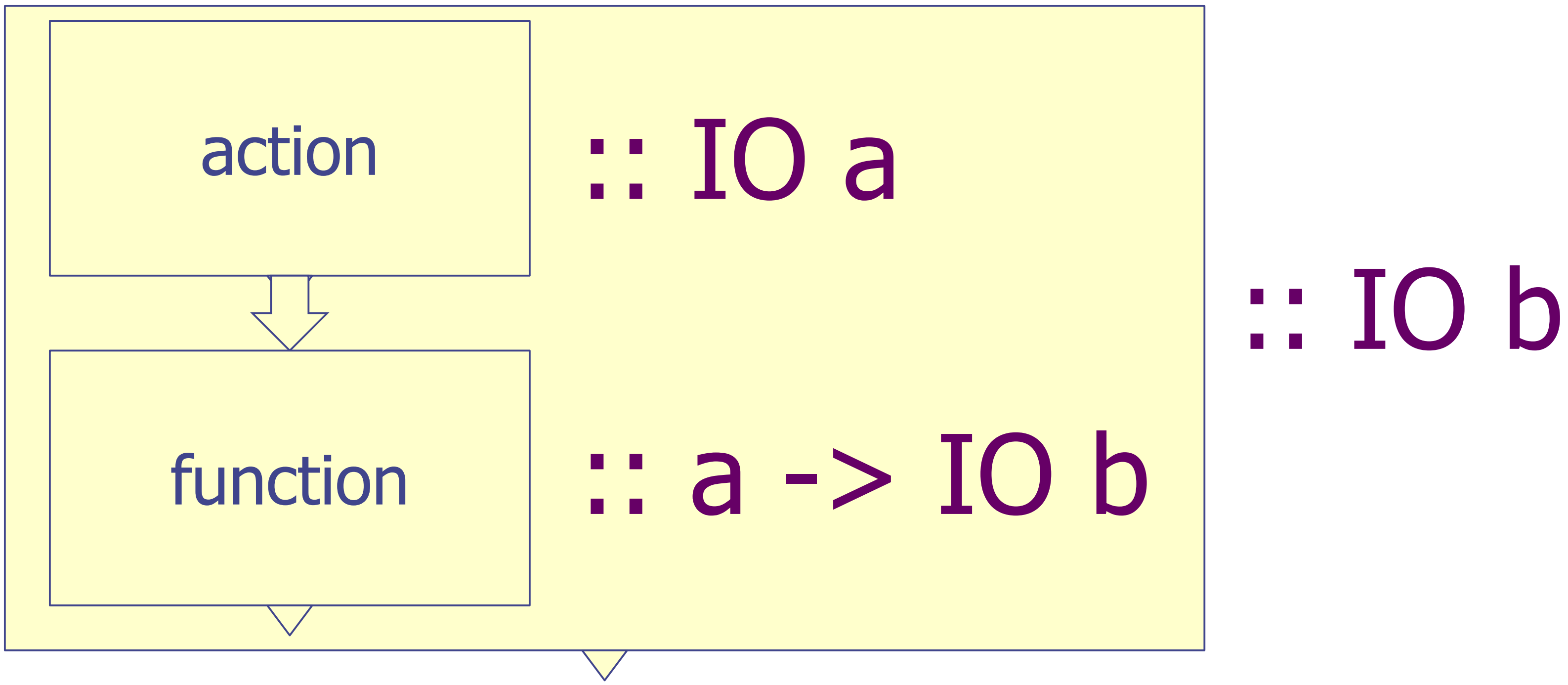
> The "unit" type:
>
> data () = ()

… and beyond the prelude…

```
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
```

etc., etc.

Combining IO Actions



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

then

```
do x <- action
   function x
```

:: IO b

Example

- Write code that reads and echoes a character, and return a Boolean indicating if it was a newline

- Primitives:

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

In scope in remainder of do block

- Code:

```
echo :: IO Bool
echo = do c <- getChar
          putChar c
          return (c == '\n')
```

introduces a sequence of IO actions

The return operator

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

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

Some laws:

```
do {x <- return e; f x} = f e
do {x <- e; return x} = e
```

Typing details for do

```
echo :: IO Bool
echo = do c <- getChar
          putChar c
          return (c == '\n')
```

:: Char

:: IO Char

:: IO ()

:: IO Bool

IO () is typical type for actions without a v<- binding

Type of last action determines type of entire do sequence

Do notation and Layout

- Haskell allows the programmer to use layout rather than explicit punctuation to indicate program structure.

- If you use layout:
  - All characters have the same width
  - Tab stops every 8 characters (but avoid tabs!)
  - All generators must start in the same column
  - Generators may be spread across multiple lines, but continuations require further indentation

Do notation and Layout

- Haskell allows the programmer to use layout rather than explicit punctuation to indicate program structure.

```
do x <- f y
     return (g x)
```

X

```
do x <- f y return (g x)
```

Last generator must be an expression

```
do x <- f y
   return (g x)
```

X

```
(do x <- f y) return (g x)
```

```
do x <- f y
   if x then g x
   else h x
```

X

```
do x <- f y
   (if x then g x)
   (else h x)
```

Syntax error(s)

Do notation and Layout

- Haskell allows the programmer to use layout rather than explicit punctuation to indicate program structure.

```
do x <- f y
   return (g x)
```

✓

```
do x <- f y
   if x then g x
        else h x
```

✓

Using Explicit Layout

- Haskell also allows the programmer to use **explicit punctuation** instead of layout.

  do { gen_1 ; gen_2 ; ... ; gen_n; expr }

```
do { x <- f y ;
     return (g x) }
```

```
do { x <- f y
   ; return (g x)
     }
```

```
do { x <- f y; return (g x) }
```

```
do { x <- f
         y ; return
      (g
     x)
        }
```

When are IO actions performed?

- A value of type IO a is an action, but it is still just a value; it will only have an effect when it is performed.
- The value of a Haskell program is the value of the variable main in the module Main. That value must have type IO t. The associated action will be performed when the whole program is run.
- There is **no** other way to perform an action (well, almost no other way)

Treatment of IO in GHCi

- If you write an expression e of type IO t at the ghci prompt, it will be performed immediately.

- In addition, the result value of type t will be displayed, provided that t is an instance of Show and t is not ().

- Example:

```
*Main> echo
a
aFalse
*Main>
```

Terminal Input

```
getChar  :: IO Char
```

The action getChar reads a single character from the terminal

Note that this action takes no parameters and does not look like a function (indeed, it is a **constant** action), but each time it is **performed** it will return a new character!

Recursive Actions

Action getLine reads characters up to (but not including) a newline.

```
getLine :: IO String
getLine =
  do c <- getChar          -- get a character
     if c == '\n' then     -- if it is a newline
       return ""           -- then return empty string
     else                  -- otherwise
       do l <- getLine     -- recurse for rest of line
          return (c:l)     -- and return entire line
```

Mapping IO Actions

```
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]

Mapping IO Actions

```
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

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)
```

Sequencing IO Actions

```
sequence :: [IO a] -> IO [a]
```

An action sequence as takes a list of IO actions IO a as its input, runs the actions in sequence, and returns the list of results as a single action

```
mapM f    = sequence . map f
mapM f xs = sequence [ f x | x <- xs ]
```

Terminal Output

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

An action putChar c takes a Char input and outputs it on the terminal producing a result of type ()

Example: `do {putChar ’h’; putChar 'i’}`

Terminal Output

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

An action putStr s takes a String input and outputs it on the terminal:

```
putStr = mapM_ putChar
```

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

```
print x = putStrLn (show x)
```

Side-effects considered harmful

- We define fst (x,y) = x
- But is fst (print 1, print 2) the same as print 1?
- Suppose that your C/C++ code calls a function int f(int n); What might happen?

Side-effects tamed!

- We define fst (x,y) = x
- But is fst (print 1, print 2) the same as print 1?
- Suppose that your Haskell code calls a function f :: Int -> Int. What might happen?

Side-effects tamed!

print 1

↓ Evaluation λ

values of type IO ()

↓ Execution

effects

Visualizing a File System

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

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

... continued

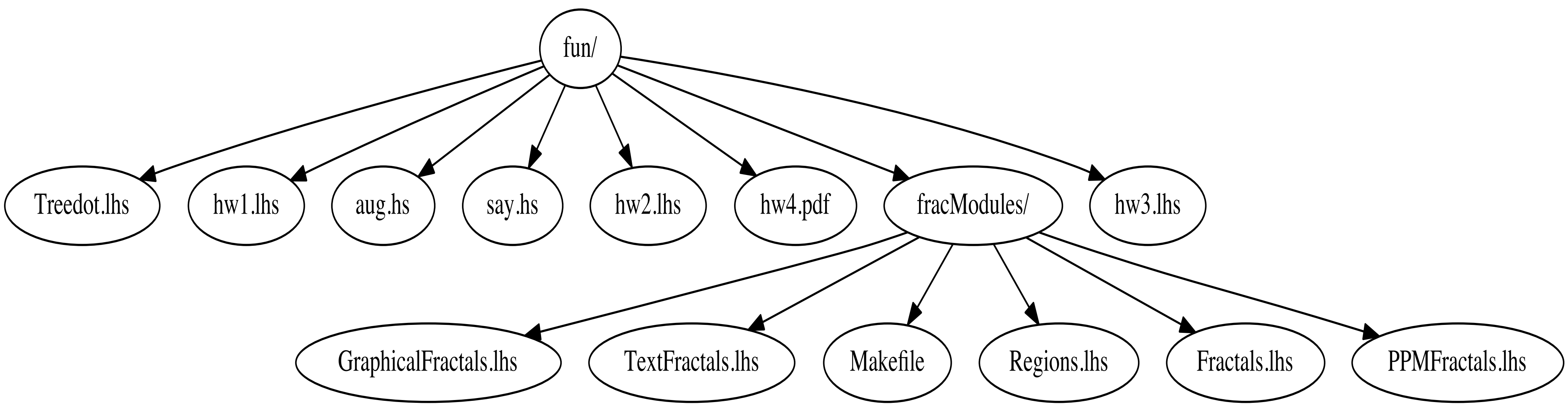
```
getFileSystemDir :: Int -> FilePath -> FilePath -> IO FileSystem
getFileSystemDir n path name
 | n < 1     = return $ Foldep name
 | otherwise = do fs <- getDirectoryContents path
                  let fs' = filter (not . dotFile) fs
                  fss <- mapM (getFileSystemIn (n-1) path) fs'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

getFileSystemIn :: Int -> FilePath -> FilePath -> IO FileSystem
getFileSystemIn n parent child
 = do b <- doesDirectoryExist path
      if b then
         getFileSystemDir n path child
      else
         return $ File child
   where path = parent </> child
```

Visualizing a FileSystem

```
dotFileSystem :: Int -> FilePath -> String -> IO ()
dotFileSystem n name dotfile =
      do fs <- getFileSystem n name
         toDot dotfile fs

getFileSystem :: Int -> FilePath -> IO FileSystem
getFileSystem n name = getFileSystemDir n name name
```



Simple File I/O

- Read contents of a text file:

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

- Write a text file:

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

- Example: Number lines

```
numLines inp out
  = do t <- readFile inp
       (writeFile out . unlines . f . lines) t
f = zipWith (\n s -> show n ++ s) [1..]
```

Handle-based File I/O

```
import IO

stdin, stderr, stdout :: Handle

openFile    :: FilePath -> IOMode -> IO Handle

hGetChar    :: Handle -> IO Char

hPutChar    :: Handle -> Char -> IO ()

hClose      :: Handle -> IO ()
```

Time

import Data.Time

getCurentTime :: IO UTCTime

getCurrentTimeZone :: IO TimeZone

getZonedTime :: IO ZonedTime

+ lots of pure operations for working with values of these types ...

For example: do { t <- getZonedTime; print t}

References

```haskell
import Data.IORef

data IORef a = ...

newIORef  :: a -> IO (IORef a)

readIORef :: IORef a -> IO a

writeIORef :: IORef a -> a -> IO ()
```

Just Because You Can …

```
gauss = do count <- newIORef 0
           total  <- newIORef 0
           let loop
                = do t <- readIORef total
                     c <- readIORef count
                     if (c>=11)
                       then return t
                       else do writeIORef total (t+c)
                               writeIORef count (c+1)
                               loop
           loop
```

It doesn't mean you should!

```
gauss :: IO Int
gauss = return (sum [1..10])
```

- You can write "C code" in Haskell
- But it's better to write C code in C and Haskell code in Haskell

Foreign Functions

A (now standard) Foreign Function Interface makes it possible to call C code from Haskell:

```
foreign import ccall
    putChar :: Char -> IO ()

foreign import ccall “putchar”
    putChar :: Char -> IO ()

foreign import ccall “intr.h enableInterrupts”
    enableInterrupts :: IO ()

foreign import ccall “io.h inb”
      inB :: Port -> IO Word8
```

… continued

- … or Haskell code from C:

  ```
  foreign export ccall foo :: Int -> Int
  ```

- Note that you can also import functions without assuming an IO result:

  ```
  foreign import ccall sin :: Float -> Float
  ```

- (But then there is an obligation on the programmer to justify/prove safety …)

Interfacing to Other Libraries

- Primitives for graphical programming:

```
mkWindow :: Int -> Int -> IO Window
setPixel :: Window -> (Int,Int) -> RGB -> IO ()
```

- Primitives for network programming:

```
socket   :: Family -> SocketType
               -> ProtocolNumber -> IO Socket
accept   :: Socket -> IO (Socket, SockAddr)
sendTo   :: Socket -> String -> SockAddr
              -> IO Int
recvFrom :: Socket -> Int
              -> IO (String, Int, SockAddr)
```

- Etc…

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

- The do notation is just "syntactic sugar" for a sequence of applications of a particular primitive function written >>= and called "bind"

- The fundamental primitives are:

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

These can be used just like any other functions

The "bind" operator

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

p >>= q is an I/O action that runs p, pipes the output into q, and runs the resulting action ...

A special case of bind

```
(>>) :: 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
```

```
(p >> q) >> r  =  p >> (q >>r)
```

“do-notation” expands to >>=

For example:

$$\begin{array}{l} \textbf{do}\ \ x_1 \text{ <- } p_1 \\ \quad\quad \ldots \\ \quad\ \ x_n \text{ <- } p_n \\ \quad\ \ q \end{array}$$

is equivalent to:

$$\begin{array}{l} p_1 \text{ >>= } \backslash x_1 \text{ -> } \\ \ldots \\ p_n \text{ >>= } \backslash x_n \text{ -> } \\ q \end{array}$$

"do-notation" without binders

- The sequence

  do p_1
  p_2
  …
  p_n

  is equivalent to:
  $p_1 >> p_2 >> \dots >> p_n$

  Of course, sequences with and without binders can be freely intermixed

IO Actions are monads

- IO actions turn out to be a special case of a more general structure called a **monad**
- Bind (>>=), return, and do-notation all work for arbitrary monads
  - via a type class!
- We will explore monads in more generality later in the course

The Haskell Logo

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