CS 457/557: Functional Languages

Lecture 8: I/O Actions in Haskell

Mark P Jones and Andrew Tolmach
Portland State University
We define \( \text{fst} (x, y) = x \)

But is \( \text{fst} (\text{print } 1, \text{print } 2) \) the same as \( \text{print } 1 \)?

Suppose that your C/C++ code calls a function \( \text{int f(int n);} \) 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

\[ \text{trace} :: \text{String} \rightarrow a \rightarrow a \]

Useful for “stick in a print statement” style of debugging

Or is it?
What gets printed?

\[
\begin{align*}
f \ x &= \ \text{trace} \ "\text{there}" \ (x+1) \\
g \ x &= \ x + \ \text{trace} \ "\text{there}" \ 1 \\
h \ x &= \ \text{if} \ x > 0 \\
    &\quad\quad\text{then} \ \text{trace} \ "\text{there}" \ 1 \\
    &\quad\quad\text{else} \ 2 \\
\end{align*}
\]

\[
\begin{align*}
f \ (\ \text{trace} \ "\text{here}" \ 1) \\
g \ (\ \text{trace} \ "\text{here}" \ 1) \\
h \ (\ \text{trace} \ "\text{here}" \ 1)
\end{align*}
\]
Violating assumptions of “computation by calculation”

\[ c = x + x \]

where \( x = \text{trace } "x" \text{ (length } "abc" \text{)} \)

\[ c = \text{trace } "x" \text{ (length } "abc" \text{)} + \text{trace } "x" \text{ (length } "abc" \text{)} \]
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 \textbf{action} that performs IO when executed.

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

An “IO action” is a value of type \( \text{IO} \ a \)

- \( a \) 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

e 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 ()
  ```

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

  introduces a sequence of IO actions
The return operator

\texttt{return :: a \rightarrow IO a}

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

Some laws:

\texttt{do \{x \leftarrow \text{return } e; f x\} = f e}
\texttt{do \{x \leftarrow e; \text{return } x\} = e}
Typing details for do

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

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

Last generator must be an expression

```
  do x <- f y
      return (g x)
  do x <- f y
      return (g 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) }
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 \( \text{IO} \ t \) 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 \texttt{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.

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

-- get a character
-- if it is a newline
-- then return empty string
-- otherwise
-- recurse for rest of line
-- and return entire line
Mapping IO Actions

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

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 \([b]\)
Mapping IO Actions

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

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.
Defining \( \text{mapM} \) and \( \text{mapM} \_ \)

\[
\begin{align*}
\text{mapM} \_ & :: (a \to \text{IO } b) \to [a] \to \text{IO } () \\
\text{mapM} \_ f \ [] & = \text{return } () \\
\text{mapM} \_ f \ (x:xs) & = \text{do } f \ x \\
& \quad \text{mapM} \_ f \ xs \\
\text{mapM} & :: (a \to \text{IO } b) \to [a] \to \text{IO } [b] \\
\text{mapM} \ f \ [] & = \text{return } [] \\
\text{mapM} \ f \ (x:xs) & = \text{do } y <- f \ x \\
& \quad y:ys <- \text{mapM} \ f \ xs \\
& \quad \text{return } (y:ys)
\end{align*}
\]
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

\texttt{putChar} :: \texttt{Char} \rightarrow \texttt{IO} ()

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

Example: \texttt{do \{putChar 'h'; putChar 'i';\}}
Terminal Output

\[
\begin{align*}
\text{putStr} &: \text{ String} \rightarrow \text{ IO } () \\
\text{putStrLn} &: \text{ String} \rightarrow \text{ IO } () \\
\end{align*}
\]

An action \texttt{putStr s} takes a \texttt{String} input and outputs it on the terminal:
\[
\text{putStr} = \text{mapM}\_\text{putChar}
\]

\texttt{putStrLn s} does the same thing but adds a trailing new 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

\[
\text{print } x = \text{putStrLn } (\text{show } x)
\]
Side-effects considered harmful

- We define \( \text{fst} (x, y) = x \)

- But is \( \text{fst} (\text{print 1, print 2}) \) the same as \( \text{print 1} \)?

- Suppose that your C/C++ code calls a function \( \text{int f(int n);} \); What might happen?
Side-effects tamed!

- We define \( \text{fst} (x,y) = x \)

- But is \( \text{fst} (\text{print } 1, \text{print } 2) \) the same as \( \text{print } 1 \)?

- Suppose that your Haskell code calls a function \( f :: \text{Int} \rightarrow \text{Int} \). What might happen?
Side-effects tamed!

print 1

values of type IO ()

effects

Evaluation

Execution

λ
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 <- getFileFileSystem n name
toDot dotfile fs

getFileSystem :: Int -> FilePath -> IO FileSystem
getFileSystem n name = getFileFileSystemDir n name name
Simple File I/O

- **Read contents of a text file:**
  
  ```haskell
  readFile :: FilePath -> IO String
  ```

- **Write a text file:**
  
  ```haskell
  writeFile :: FilePath -> String -> IO ()
  ```

- **Example: Number lines**
  
  ```haskell
  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

```haskell
import IO

stdin, stderr, stdout :: Handle

openFile :: FilePath -> IOMode -> IO Handle

hGetChar :: Handle -> IO Char

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

hClose :: Handle -> IO ()
```
import Data.Time
getCurrentTime :: 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}
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:

```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:

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

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

```haskell
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:
- \( \text{mkWindow} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{IO Window} \)
- \( \text{setPixel} :: \text{Window} \rightarrow (\text{Int},\text{Int}) \rightarrow \text{RGB} \rightarrow \text{IO ()} \)

Primitives for network programming:
- \( \text{socket} :: \text{Family} \rightarrow \text{SocketType} \rightarrow \text{ProtocolNumber} \rightarrow \text{IO Socket} \)
- \( \text{accept} :: \text{Socket} \rightarrow \text{IO (Socket, SockAddr)} \)
- \( \text{sendTo} :: \text{Socket} \rightarrow \text{String} \rightarrow \text{SockAddr} \rightarrow \text{IO Int} \)
- \( \text{recvFrom} :: \text{Socket} \rightarrow \text{Int} \rightarrow \text{IO (String, Int, SockAddr)} \)

Etc…
There is No Escape!

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

- Once a program is “tainted” with IO, there is no way to “shake it off”

- For example, there is no primitive of type \( \text{IO} \ t \rightarrow t \) 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 \( \gg= \) and called “bind”

- The fundamental primitives are:
  
  \[
  (\gg=) :: \text{IO } a \to (a \to \text{IO } b) \to \text{IO } b
  \]
  
  \[
  \text{return} :: a \to \text{IO } a
  \]

  These can be used just like any other functions
The “bind” operator

\[(>>=) \quad :: \quad \text{IO} \ a \rightarrow (a \rightarrow \text{IO} \ b) \rightarrow \text{IO} \ b\]

\[p \ >>\ = q\] is an I/O action that runs \(p\), pipes the output into \(q\), and runs the resulting action ...
A special case of bind

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

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

\[p \gg q = p \gg\gg \lambda x \rightarrow q\]

\[(p \gg q) \gg r = p \gg (q \gg r)\]
“do-notation” expands to $\gg=$

For example:

```
do  x_1 <- p_1
    ...
    x_n <- p_n
    q
```

is equivalent to:

```
p_1 \gg= \backslash x_1 ->
    ...
    p_n \gg= \backslash x_n ->
    q
```
“do-notation” without binders

The sequence

\[
\textbf{do } p_1 \\
p_2 \\
\ldots \\
p_n
\]

is equivalent to:

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

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 (\(\gg\gg=\))\(^a\), 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