Monadic I/O in Haskell

Jim Royer

CIS 352

March 5, 2019

trapd in IO monad

plz help
References

- Chapter 9 of *Learn you a Haskell for Great Good* by Miran Lipovača
  
  http://learnyouahaskell.com/input-and-output

- “Tackling the Awkward Squad,” by Simon Peyton Jones
  
  http://research.microsoft.com/~simonpj/papers/marktoberdorff/

- Tutorials/Programming Haskell/String IO
  
  https://wiki.haskell.org/Tutorials/Programming_Haskell/String_IO

- Software Tools in Haskell
  
  https://crsr.net/Programming_Languages/SoftwareTools/
The program should have a module called `Main`, containing a function called `main`:

```haskell
module Main where

main :: IO ()
main = (...)```

The first line can be omitted, since the default module name is `Main`.

Here is a complete example: ...
module Main where

main :: IO ()
main = putStrLn "Hello, world!"

putStrLn :: String -> IO () -- prints a string to output.

How do we create our own IO actions?
Haskell is pure.
- Evaluating a Haskell expression just produces a value.
- It does not change anything!
- Ghci, not Haskell, handles printing results.

But the point of a program is to interact with the world — if only at the level of input & output.

∴ Doing input/output in Haskell requires a new idea.
Monadic I/O

- An I/O action has a type of the form \((\text{IO} \ a)\).
- An expression of type \((\text{IO} \ a)\) produces an action.
- When this action is performed:
  - it may do some input/output,
  - \text{and}
  - finally produces a value of type \(a\).
- Roughly: \(\text{IO} \ a \approx \text{World} \to (a, \text{World})\)

(Pictures from SPJ.)
Primitive I/O

**Stage Directions:** Open ghci in a window & play with these toys.

- `getChar` an action of type `IO Char`
- `putChar 'x'` an action of type `IO ()`

`getChar :: IO Char`

`putChar :: Char -> IO ()`

Main program is an action of type `IO ()`
Combining actions, I

Problem
We want to read a character and write it out again.

So we want something like:

Since this is Haskell: *when in need, introduce a new function.*
Combining actions, II

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

(built in)

--Sequentially compose two actions, passing any value
--produced by the first as an argument to the second.

Now we can define

\[
\text{echo} :: \text{IO } ()
\]

\[
\text{echo} = \text{getChar} \gg\gg \text{putChar}
\]

[Stage Directions: In a terminal window, load io.hs into ghci.]
Aside

$$(>>=) :: IO a \to (a \to IO b) \to IO b$$

--Sequentially compose two actions, passing any value
--produced by the first as an argument to the second.

You can tell that the Haskell community thinks $>>= $ is important since

is their logo.
Grab a character and print it twice

echoTwice :: IO ()

```haskell
echoTwice = getChar >>= (\c ->
putChar c >>= (\() ->
putChar c))
```

- As SPJ points out, the parens are optional.
  (Not that it helps readability much.)
- We drop the \( () \rightarrow \) stuff via another combinator:

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

\[
m \gg\gg n = m \gg\gg (\text{x } \rightarrow n)\\n\text{-- n ignores m's output.}
\]
So with

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

\[m \gg n = m \gg= (\lambda x \rightarrow n)\]

-- \(n\) ignores \(m\)'s output.

We can rewrite echoTwice as:

**Grab a character and print it twice (revised)**

\[
\begin{align*}
\text{echoTwice} &:: \text{IO } () \\
\text{echoTwice} &\equiv \text{getChar} \gg= \lambda c \rightarrow \\
&\quad \text{putChar } c \gg \\
&\quad \text{putChar } c
\end{align*}
\]

*(Still rather clunky! But we aren’t done yet.)*
Next problem:

Read two characters and return them

```
getTwoChars = getChar >>= (\c1 ->
                        getChar >>= (\c2 ->
                                      return (c1, c2)
                        ?? now what ??
```
Next problem:

**Read two characters and return them**

```haskell
getTwoChars = getChar >>= \c1 ->
getChar >>= \c2 ->
        return (c1, c2)
```

Another combinator:  

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

Jim Royer  

Monadic I/O in [1ex] Haskell  

13 / 33
Next problem:

Read two characters and return them

\[
\text{getTwoChars} = \text{getChar} >>= \lambda c1 -> \\
\text{getChar} >>= \lambda c2 -> \\
\text{return} (c1,c2)
\]

Another combinator: \( \text{return} :: \text{a} \to \text{IO a} \)
The do-notation

The clunky looking

getTwoChars
    = getChar >>= \c1 ->
      getChar >>= \c2 ->
      return (c1,c2)

Warning: \<- is not an assignment operator!!!!
The do-notation

The clunky looking

\[
\text{getTwoChars} = \text{getChar} >>= \lambda c_1 \rightarrow \\
\quad \text{getChar} >>= \lambda c_2 \rightarrow \\
\quad \text{return} (c_1,c_2)
\]

can be rewritten as:

\[
\text{getTwoChars} = \begin{aligned}
&\text{do} \{ \\
&\quad c_1 \leftarrow \text{getChar} ; \\
&\quad c_2 \leftarrow \text{getChar} ; \\
&\quad \text{return} (c_1,c_2) \\
&\}
\end{aligned}
\]

Warning: \texttt{\leftarrow} is not an assignment operator!!!!

Jim Royer
Monadic I/O in \[1ex\] Haskell 14 / 33
The do-notation

The clunky looking

```haskell
getcTwoChars
  = getChar >>= \c1 ->
    getChar >>= \c2 ->
    return (c1,c2)
```

can be rewritten as:

```haskell
getcTwoChars
  = do { c1 <- getChar;
        c2 <- getChar;
        return (c1,c2) }
```

and as

```haskell
getcTwoChars
  = do { c1 <- getChar
         ; c2 <- getChar
         ; return (c1,c2)
       }
```

Warning: `<-` is not an assignment operator!!!!
The do-notation

The clunky looking

```haskell
getTwoChars
  = getChar >>= \c1 ->
    getChar >>= \c2 ->
    return (c1,c2)
```

can be rewritten as:

```haskell
getTwoChars
  = do { c1 <- getChar;
    c2 <- getChar;
    return (c1,c2)
  }
```

and as

```haskell
getTwoChars
  = do { c1 <- getChar;
    ; c2 <- getChar;
    ; return (c1,c2)
  }
```

as well as

```haskell
getTwoChars
  = do { c1 <- getChar;
    c2 <- getChar;
    return (c1,c2)
  }
```

Warning: `<-` is not an assignment operator!!!!
The do-notation

The clunky looking

getTwoChars
    = getChar >>= \c1 ->
      getChar >>= \c2 ->
      return (c1,c2)

can be rewritten as:

getTwoChars
    = do { c1 <- getChar;
        c2 <- getChar;
        return (c1,c2) }

and as

getTwoChars
    = do { c1 <- getChar
        ; c2 <- getChar
        ; return (c1,c2) }

as well as

getTwoChars
    = do c1 <- getChar
        c2 <- getChar
        return (c1,c2)

Warning:  <- is not an assignment operator!!!
The do-notation is syntactic sugar

\[
\begin{align*}
do \{ x \leftarrow e; s \} & \equiv e \gg= \ x \rightarrow do \{ s \} \\
do \{ e; e \} & \equiv e \gg do \{ s \} \\
do \{ e \} & \equiv e
\end{align*}
\]

Some examples, I

- `putStr :: String -> IO ()` (built in)
  - outputs a string

- `putStrLn :: String -> IO ()` (built in)
  - outputs a string followed by a new line
  
  `putStrLn str = do { putStr str; putStr "\n" }`

- `print :: Show a => a -> IO ()` (built in)
  - outputs a Haskell value
  
  `print x = putStrLn (show x)`

- `put4times :: String -> IO ()`
  - print a string four times

  `put4times str = do putStrLn str
                     putStrLn str
                     putStrLn str
                     putStrLn str`
Some examples, II

Print a string \( n \) times

\[
\text{putNtimes :: Int} \to \text{String} \to \text{IO ()}
\]
\[
\text{putNtimes } n \text{ str} = \begin{cases} 
\text{putStrLn str} & \text{if } n \leq 1 \\
\text{else do putStrLn str} \\
\text{putNtimes } (\text{n-1}) \text{ str} 
\end{cases}
\]

Gets a line of input

\[
\text{getLine :: IO String} \text{ (built in)}
\]
\[
\text{getLine} = \begin{cases} 
\text{do c <- getChar} \\
\text{if c == } \backslash n' \\
\text{then return ''} \\
\text{else do cs <- getLine} \\
\text{return (c:cs)}
\end{cases}
\]
However, note that it is often easier to do the heavy lifting in the “functional” part of Haskell. E.g., in place of:

**Print a string \( n \) times**

```haskell
putNtimes :: Int -> String -> IO ()
putNtimes n str = if n <= 1
    then putStrLn str
    else do putStrLn str
           putNtimes (n-1) str
```

instead you can do this:

**Print a string \( n \) times**

```haskell
putNtimes' :: Int -> String -> IO ()
putNtimes' n str = putStrLn $ unlines $ replicate n str
```
Some examples, III

**copy a line from input to output**

```haskell
copy :: IO ()
copy = do { line <- getLine ; putStrLn line }
```

**read two lines, print them in reverse order and reversed**

```haskell
reverse2lines :: IO ()
reverse2lines = do line1 <- getLine
                   line2 <- getLine
                   putStrLn (reverse line2)
                   putStrLn (reverse line1)
```

**Convert a String to a Haskell value of type a**

```haskell
read :: Read a => String -> a
```

(read :: Read a => String -> a) is built in

**Read an Int from Input**

```haskell
getInt :: IO Int
getInt = do { item <- getLine ; return (read item :: Int) }
```
Problem
Read a series of positive integers from input and sum them up. Stop reading when an integer $\leq 0$ is found & then return the sum.

A simple version

```haskell
sumInts :: IO Int
sumInts = do n <- getInt
             if n <= 0
                then return 0
                else do { m <- sumInts ; return (m+n) }
```

A chatty version

```haskell
chattySum
    = do { putStrLn "Enter integers one per line"
          ; putStrLn "These will be summed until a 0 is entered."
          ; sum <- sumInts
          ; putStrLn "The sum is " ; print sum }
```
Some examples, V

More built-ins

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

type FilePath = String

copyFile :: FilePath -> FilePath -> IO ()
copyFile source target = do s <- readFile source
                            writeFile target s

sortFile :: FilePath -> FilePath -> IO ()
sortFile source target = do s <- readFile source
                           writeFile target (sortLines s)

sortLines = unlines . sort . lines
```
Haskell keeps pure and impure functions apart

- Pure ≡ no side-effects (easy to debug, get correct)
- In the above examples, IO types marks a function as impure.
- Roughly, you can only get access to an “outside” value inside of a do-block — where you may apply pure functions to it.
- Keep IO actions simple
- Do most of the serious work via pure functions.

**Example:** The sortFile example from before.
Things that **FAIL** to work in a do-block, I

**Problem**

In the context of this code, in:

```
x <- ??
```

the type of `??` must be `(IO Int)`.

**However, this works.**

```haskell
attempt2 :: IO Int
attempt2 = do { let x = 12
              ; return x
              }
```

**Do-blocks allow lets**

Use a `let` to introduce local variables.
The do-laws, revised

\[
\begin{align*}
\text{do } \{ x \leftarrow e; s \} & \equiv e \gg= \lambda x \rightarrow \text{do } \{ s \} \\
\text{do } \{ \text{let } x = e; s \} & \equiv \text{let } x = e \text{ in } \text{do } \{ s \} \\
\text{do } \{ e; e \} & \equiv e \gg \text{do } \{ s \} \\
\text{do } \{ e \} & \equiv e
\end{align*}
\]

**Fact:** When you are typing to the ghci prompt, you are in an \((\text{IO } (\_))\) do-block.
Things that **FAIL** to work in a do-block, II

**This will not compile.**

```haskell
attempt3 = do { return 12 }
```

**Problem**
Haskell does not have enough information to figure out the type of the result.

**However, this works.**

```haskell
attempt4 :: IO Int
attempt4 = do { return 12 }
```

**Also try:**

```haskell
attempt5 :: Maybe Int
attempt5 = do { return 12 }
```
Safer versions of getChar & putChar

getc :: IO (Maybe Char)
getc = do eof <- isEOF
    if eof
        then return Nothing
    else
        do c <- getChar
        return (Just c)

putc :: Char -> IO ()
putc = putChar

Copy std-in to std-out

copy :: IO ()
copy
    = do ch <- getc
    case ch of
        Nothing -> return ()
        Just c -> do putc c
                    copy
Count the number of characters in the input

```haskell
charcount :: IO ()
charcount = do nc <- cc 0
    putStrLn (show nc)

where
    cc nc = do ch <- getc
        case ch of
            Nothing -> return nc
            Just c  -> cc $! nc + 1
```

($!$) :: (a -> b) -> a -> b

Strict (call-by-value) application operator
Count the number of lines in the input

```haskell
linecount :: IO ()
linecount = do { nl <- lc 0 ; putStrLn (show nl) }
  where
    lc nl = do ch <- getc
      case ch of
        Nothing     -> return nl
        Just '\n'  -> lc $! nl + 1
        Just _      -> lc nl
```
interact :: (String -> String) -> IO ()
Standard input is passed to the (String -> String)-function as its argument; the resulting string is output to standard output.

Alternatives

\[
\begin{align*}
\text{showLn } v & = (\text{show } v)++"\n" \\
\text{copy’} & = \text{interact id} \\
\text{charcount’} & = \text{interact (showLn . length)} \\
\text{linecount’} & = \text{interact (showLn . length . lines)}
\end{align*}
\]

Often, using \text{interact} results in faster code than the imperative-style programs.
An IO-action is just another value to be passed around. So we can build our own control structures.

**repeat a particular IO-action forever**

```
forever :: IO () -> IO ()
forever a = do { a ; forever a }
```

**repeat a particular IO-action \( n \) times**

```
repeatN :: Int -> IO () -> IO ()
repeatN 0 a = return ()
repeatN n a = do { a ; repeatN (n-1) a }
```

**Do an IO action of each element of a list**

```
for :: [a] -> (a -> IO b) -> IO ()
for [] fa = return ()
for (x:xs) fa = do { fa x ; for xs fa }
```
Do an IO action of each element of a list

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

\[
\text{for } [] \text{ fa } = \text{return } ()
\]

\[
\text{for } (x:xs) \text{ fa } = \text{do } \{\text{fa x ; for xs fa}\}
\]

Alternative definition

\[
\text{for } xs \text{ fa } = \text{sequence_ } [\text{fa x } | x <\text{-} xs]
\]

where

\[
\text{sequence_ :: } [\text{IO } a] \rightarrow \text{IO } () \quad \text{(built in)}
\]

\[
\text{sequence_ as } = \text{foldr } (\gg\gg) (\text{return }()) \text{ as}
\]

or if you prefer

\[
\text{sequence_ } [] = \text{return } ()
\]

\[
\text{sequence_ } (a:as) = a \gg (\text{sequence_ as})
\]
Do a list of IO actions and return the list of results

```
sequence :: [IO a] -> IO [a] (built in)
sequence [] = return []
sequence (a:as) = do r <- a
                    rs <- sequence as
                    return (r:rs)
```

```
main = do
      a <- getLine
      b <- getLine
      c <- getLine
      print [a,b,c]

main = do
      rs <- sequence [getLine,getLine,getLine]
      print rs
```

...and so on.
Labeled Transitions

- $P \overset{!c}{\rightarrow} Q$  
  $P$ can move to $Q$ by writing character $c$ to standard output

- $P \overset{?c}{\rightarrow} Q$  
  $P$ can move to $Q$ by reading the character $c$ from standard output

First Two Transition Rules

\[
\begin{align*}
\{ \text{putChar } c \} & \overset{!c}{\rightarrow} \{ \text{return } () \} \\
\{ \text{getChar } \} & \overset{?c}{\rightarrow} \{ \text{return } c \}
\end{align*}
\]