Monadic I/O in Haskell

Jim Royer

CIS 352

March 5, 2019

References

- Chapter 18 of *Haskell: the Craft of Functional Programming* by Simon Thompson, Addison-Wesley, 2011.
- 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/marktoberdorf/
- Tutorials/Programming Haskell/String IO
  https://wiki.haskell.org/Tutorials/Programming_Haskell/String_IO
- Software Tools in Haskell
  https://crsr.net/Programming_Languages/SoftwareTools/

Digression: Creating stand-alone Haskell Programs

- The program should* have a module called `Main`, containing a function called `main`:

```
module Main where

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

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

- Here is a complete example: . . .

Digression, continued

```
module Main where

main :: IO ()
main = printStrLn "Hello, world!"
```

```
[Post:pl/code/IO] jimroyer% cat hello.hs
module Main where

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

[Post:pl/code/IO] jimroyer% ghc --make hello.hs
[Post:pl/code/IO] jimroyer% ./hello
Hello, world!
```

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

How do we create our own IO actions?

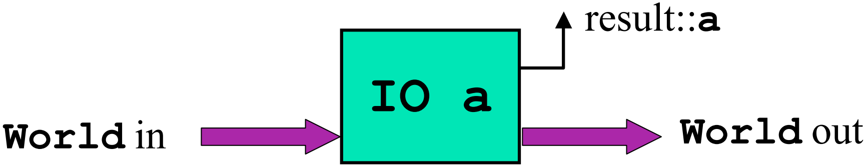
The conflict

- 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.

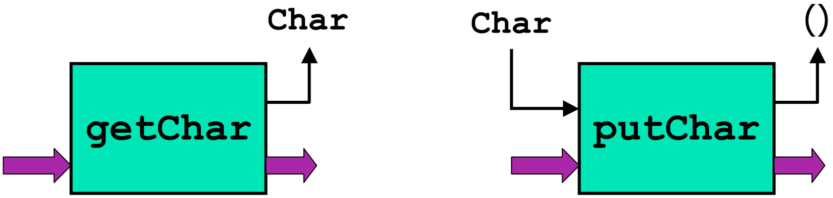
∴ Doing input/output in Haskell requires a new idea.

Monadic I/O

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



(Pictures from SPJ.)




```
getChar :: IO Char
```

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

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

what is ()?

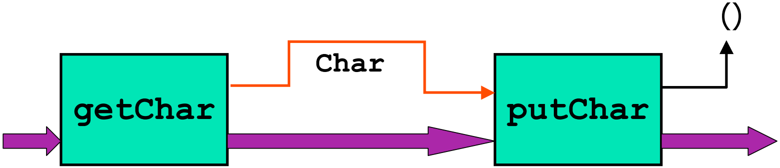
[**Stage Directions:** Open ghci in a window & play with these toys.]

Combining actions, I

Problem

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

So we want something like:



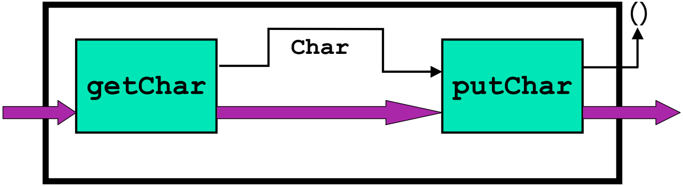
Since this is Haskell: when in need, introduce a new function.

Combining actions, II

```
(>>=) :: IO a -> (a -> IO b) -> IO b
--Sequentially compose two actions, passing any value
--produced by the first as an argument to the second.
```
(built in)

Now we can define

```
echo :: IO ()
echo  =  getChar >>= putChar
```



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

Aside

```
(>>=) :: IO a -> (a -> IO b) -> IO b
  --Sequentially compose two actions, passing any value
  --produced by the first as an argument to the second.
```

(built in)

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

is their logo.

Combining actions, III

Grab a character and print it twice

```
echoTwice :: IO ()
echoTwice  =  getChar   >>= (\c ->
              putChar c >>= (\() ->
              putChar c))
```

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

`(>>) ::  IO a -> IO b -> IO b` **(built in)**

```
m >> n  =  m >>= (\x -> n)
--n ignores m's output.
```

So with

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

(built in)

```
m >> n  =  m >>= (\x -> n)
--n ignores m's output.
```

We can rewrite `echoTwice` as:

Grab a character and print it twice (revised)

```
echoTwice :: IO ()
echoTwice  =  getChar   >>=   \c ->
              putChar c >>
              putChar c
```

(Still rather clunky! But we aren't done yet.)

Combining actions, V

Next problem:

Read two characters and return them

```
getTwoChars  =  getChar >>= \c1 ->
                getChar >>= \c2 ->
                ?? now what ??
```

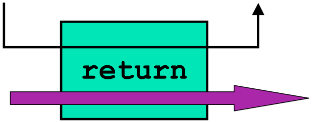
Combining actions, V

Next problem:

Read two characters and return them

```
getTwoChars  =  getChar >>= \c1 ->
                getChar >>= \c2 ->
                ?? now what ??
```

Another combinator: `return :: a -> IO a`



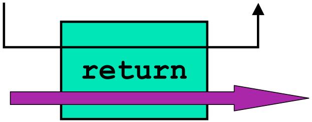
Combining actions, V

Next problem:

Read two characters and return them

```
getTwoChars  =  getChar >>= \c1 ->
                getChar >>= \c2 ->
                ?? now what ??
```

Another combinator: `return :: a -> IO a`



Read two characters and return them

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

The do-notation

The clunky looking

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

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)
        }
```

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)
        }
```

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)
```

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-laws

The do-notation is syntactic sugar*

```
do { x <- e; s }  ≡  e >>= \x -> do { s }
do { e; e }       ≡  e >> do { s }
do { e }          ≡  e
```

* http://en.wikipedia.org/wiki/Syntactic_sugar

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

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

Gets a line of input

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

Aside

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

```
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

```
putNtimes' :: Int -> String -> IO ()
putNtimes' n str  =  putStr $ unlines $ replicate n str
```

Some examples, III

copy a line from input to output

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

read two lines, print them in reverse order and reversed

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

Convert a String to a Haskell value of type a

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

Read an Int from Input

```
getInt :: IO Int
getInt  =  do { item <- getLine ; return (read item :: Int) }
```

Some examples, IV

Problem

Read a series of positive integers from input and sum them up.
Stop reading when an integer ≤ 0 is found & then return the sum.

A simple version

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

A chatty version

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

Some examples, V

More built-ins

```
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 $\equiv$ 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

This will not compile.

```
attempt1 :: IO Int
attempt1 = do { x <- 12
              ; return x
              }
```

Problem

In the context of this code, in:

```
x <- ??
```

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

However, this works.

```
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

```
do { x <- e; s }        ≡  e >>= \x -> do { s }
do { let x = e; s }     ≡  let x = e in do { s }
do { e; e }             ≡  e >> do { s }
do { e }                ≡  e
```

Fact: When you are typing to the ghci prompt, you are in an `(IO ())` do-block.

Things that FAIL to work in a do-block, II

This will not compile.

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

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

However, this works.

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

Also try:

```
attempt5 :: Maybe Int
attempt5 = do { return 12 }
```

More examples: Module IOTools, I

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
```

More examples: Module IOTools, II

Count the number of characters in the input

```
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

More examples: Module IOTools, III

Count the number of lines in the input

```
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
```

More examples: Module IOTools, IV

`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

```
showLn v    = (show v)++"\n"

copy'       = interact id

charcount'  = interact (showLn . length)

linecount'  = interact (showLn . length . lines)
```

Often, using `interact` results in faster code that the imperative-style programs.

Control structures, I

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 }
```

Control structures, II

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 }
```

Alternative definition

```
for xs fa  =  sequence_ [fa x | x <- xs]
```

where

```
sequence_ :: [IO a] -> IO ()                    (built in)
sequence_ as  =  foldr (>>) (return ()) as
```

or if you perfer

```
sequence_ []      =  return ()
sequence_ (a:as)  =  a >> (sequence_ as)
```

Control structures, III

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.

Operational semantics of IO (too briefly)

Labeled Transitions

- $P \xrightarrow{!c} Q$ — P can move to Q by writing character c to standard output
- $P \xrightarrow{?c} Q$ — P can move to Q by reading the character c from standard output

First Two Transition Rules

$$\{\texttt{putChar}\ c\} \xrightarrow{!c} \{\texttt{return ()}\}$$

$$\{\texttt{getChar}\} \xrightarrow{?c} \{\texttt{return}\ c\}$$