CIS 352

Monadic I/O in Haskell

Jim Royer
March 19, 2015

The conflict

- **Haskell is pure.**
  - Evaluating a Haskell expression just produces a value.
  - It does not change anything!
  - Ghci, not Haskell, handles the printing of the result.
- But the point of a program is to interact with the world, if only at the level of input & output.
- So, doing input/output in Haskell requires a new idea.

References

- Chapter 9 of *Learn you a Haskell for Great Good* by Miran Lipovača
  
- “Tackling the Awkward Squad,” by Simon Peyton Jones
  

Monadic I/O

- An I/O action has a type of the form `(IO a)`.
- 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: \( \text{IO } a \approx \text{World } \rightarrow (a, \text{World}) \)

(Pictures from SPJ.)
**Combining actions, I**

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

So we want something like:

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

Main program is an action of type `IO ()`

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.

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

We have connected two actions to make a new, bigger action.

**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
```

--n ignores m's output.

```
m >> n = m >>= \x -> n
```
Combining actions, IV

So with

\[ (>>) :: \text{IO } a \to \text{IO } b \to \text{IO } b \]
\[ m >> n = m >>= (\lambda x \to n) \]
\[-n \text{ ignores } m\text{'s output.} \]

We can rewrite `echoTwice` as:

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

echoTwice :: \text{IO } ()
echoTwice = getChar >>= \(\lambda c \to
  \text{putChar } c \>
  \text{putChar } c
```

(Still rather clunky!)

The do-notation

The clunky looking

```
getTwoChars
  = getChar >>= \(\lambda c1 \to
  getChar >>= \(\lambda c2 \to
  \text{return } (\text{c1},\text{c2})
```

can be rewritten as:

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

and as

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

as well as

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

The do-laws

The do-notation is syntactic sugar*

\[ \text{do } \{ x <- e ; s \} \equiv e >>= \lambda x \to \text{do } \{ s \} \]
\[ \text{do } \{ e ; e \} \equiv e >> \text{do } \{ s \} \]
\[ \text{do } \{ e \} \equiv e \]

Some examples, I

- `putStrLn :: String -> IO ()` (built in)
  outputs a string followed by a new line
  `putStrLn str = do putStrLn 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 == '
' then return ""
                 else do cs <- getLine
                        return (c:cs)

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)

Some examples, IV

**Problem**
Read a series of positive integers from input and sum them up. Stop reading when an integer \(\leq 0\) is found and 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
               putStrLn "The sum is "
               print sum`
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 }
```

Jim Royer :: Monadic I/O in [1ex] Haskell

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 ns fa = sequence_ [fa n | n <- ns]
where
  sequence_ :: [IO a] -> IO () (built in)
  sequence_ as = foldr (>>) (return ()) as
or if you prefer
  sequence_ [] = return ()
  sequence_ (a:as) = a >> (sequence_ as)
```

Jim Royer :: Monadic I/O in [1ex] Haskell

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.