Using Monads for Input and Output

Arvind
Jan-Willem Maessen
Laboratory for Computer Science
M.I.T.
Lecture 15

http://www.csg.lcs.mit.edu/6.827

Functional Languages and I/O

\[ z := f(x) + g(y); \]

In a functional language \( f \) and \( g \) can be evaluated in any order but not in a language with side-effects.

Consider inserting print statements (say for debugging) in \( f \) and \( g \).

An imperative language must take a position on evaluation order; if there is any doubt, must write it as

\[ a := f(x); b := g(y); z := a+b; \]

I/O is all about side-effects.

Is I/O incompatible with FL?

http://www.csg.lcs.mit.edu/6.827
What other languages do

- Execute programs in a fixed order:

```
(define (hello)
  (princ "Hello ")
  (princ "World "))
```

- Sequentiality simplifies the problem
- Weaker equational behavior:

```
(let ((a (f x)))
  (let ((b (g y)))
    (+ a b)))
```

Print string

```
printString :: String -> ()
printString "Hello World!"
```

but what about

```
let
  printString "Hello 
  printString "World!"
in ()
```

The string may be printed all jumbled up.

alternatives:

Output convention
Forced sequencing (Usually not available in pure FL’s)
Need for Sequencing

```haskell
echo :: () -> ()
echo () =
    let c = getChar()
    in if c=='\n'
    then ()
    else let putChar c
           >>>
               echo ()
           in ()
```

What about modularity?

Barriers are too coarse-grained:

```haskell
myProgram () =
    let input = produceAllTheInput()
        consumeAndOutput input
    in ()
```

Interleave producer and consumer
Very complex in general
Magic return value

getChar returns a magic value in addition to the character indicating that further I/O is safe.

echo :: World -> World
echo world0 =
    let (c, world1) = getChar world0
    in if c=='\n' then ()
        else let world2 = putChar c world1
                world3 = echo world2
                in world3

Used in Id and Clean

The Mind-Body Problem

RTS/OS provides the initial state of the world

main :: World -> World

Link Computation with Action:
  Computation: parallel, data constrains
  I/O Action: world imposes order
Role of Program Driver

Suppose by convention

\[
\begin{align*}
\text{main} &: \ [\text{string}] \\
\text{main} &= [\text{“Hello"}, \text{“world!”}] \\
\text{or} \\
\text{main} &= \text{let} \ a = \text{“Hello”} \\
&\quad \quad \quad b = \text{“World!”} \\
&\quad \quad \quad \text{in} \ [a,b]
\end{align*}
\]

Program is a specification of intended effect to be performed by the program driver

The driver, a primitive one indeed, takes a string and treats it as a sequence of commands to print.

---

Monadic I/O in Haskell and pH

Monadic I/O treats a sequence of I/O commands as a specification to interact with the outside world.

The program produces an actionspec, which the program driver turns into real I/O actions.

A program that produces an actionspec remains purely functional!

\[
\begin{align*}
\text{main} &: \ IO () \\
\text{putChar} &: \ Char \rightarrow IO () \\
\text{getChar} &: \ IO \ Char
\end{align*}
\]

\[
\text{main} = \text{putChar} \ ‘a’
\]

is an actionspec that says that character “a” is to be output to some standard output device

How can we sequence actionspecs?
Sequencing

We need a way to compose actionspecs:

\[(>>) \quad :: \quad IO() \rightarrow IO() \rightarrow IO()\]

Example:

\[
\begin{align*}
\text{putChar 'H' >> putChar 'i' >> putChar '!':} & \quad :: \quad IO() \\
\text{putString :: String \rightarrow IO() =} & \quad \text{done} \\
\text{putString (c:cs) =} & \quad \text{putChar c >> putString cs}
\end{align*}
\]

Monads: Composing Actionspecs

We need some way to get at the results of \(\text{getChar}\):

\[(\ggg) \quad :: \quad IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b\]

We read the "bind" operator as follows:

\[
x_1 \ggg \\lambda a \rightarrow x_2
\]

- Perform the action represented by \(x_1\), producing a value of type "a"
- Apply function \(\lambda a \rightarrow x_2\) to that value, producing a new actionspec \(x_2:: IO b\)
- Perform the action represented by \(x_2\), producing a value of type \(b\)

Example:

\[
\begin{align*}
\text{getChar >> getChar} & \quad :: \quad \text{IO c} \\
\text{the same as getChar >> getChar} & \quad :: \quad \text{IO c}
\end{align*}
\]

http://www.csg.lcs.mit.edu/6.827
An Example

main = 
    let
        islc c = putChar (if ('a'<=c)&&(c<='z')
                           then 'y'
                           else 'n')
    in
        getChar >>= islc

Turning expressions into actions

return :: a -> IO a
getLine :: IO String

getLine = getChar >>= \c ->
    if (c == '\n') then
        return ""
    else getLine >>= \s ->
        return (c:s)

where '\n' represents the newline character
Monadic I/O

Separate computation from sequencing

\[ \text{IO } a : \text{computation which does some I/O, then produces a value of type } a. \]

\[
\begin{align*}
(>>) & : \text{IO } a \rightarrow \text{IO } b \rightarrow \text{IO } b \\
(>>=) & : \text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b \\
\text{return} & : a \rightarrow \text{IO } a
\end{align*}
\]

Primitive actionspecs:

- getChar :: IO Char
- putChar :: Char -> IO ()
- openFile, hClose, ...

Monadic I/O is a clever, type-safe idea which has become very popular in the FL community.

---

Syntactic sugar: do

\[
\begin{align*}
do e & \rightarrow e \\
do e ; \text{dostmts} & \rightarrow e >>= \text{dostmts} \\
do p<-e ; \text{dostmts} & \rightarrow e >>= \text{if } (c == \text{`\n'}) \text{ then return } \text{""} \text{ else do s <- getline return (c:s)}}
\end{align*}
\]
Example: Word Count Program

type Filepath = String

data IOMode = ReadMode | WriteMode | ...

data Handle = ... implemented as built-in type

openFile :: FilePath -> IOMode -> IO Handle
hClose :: Handle -> IO ()

hIsEOF :: Handle -> IO Bool
hGetChar :: Handle -> IO Char

wc :: String -> IO (Int,Int,Int)
wcc filename =
do h <- openFile filename ReadMode
   (nc,nw,nl) <- wch h False 0 0 0
   hClose h
   return (nc,nw,nl)

Word Count Program cont.

wch :: Handle -> Bool -> Int -> Int -> Int
    -> IO (Int,Int,Int)
wch h inWord nc nw nl =
do eof <- hIsEOF h
   if eof then return (nc,nw,nl)
   else
      do c <- hGetChar h
         if (c=='\n') then
            wch h False (nc+1) nw (nl+1)
         else if (isSpace c) then
            wch h False (nc+1) nw nl
         else if (not inWord) then
            wch h True (nc+1) (nw+1) nl
         else
            wch h True (nc+1) nw nl
Calling WC

```haskell
main :: IO ()
main = do [filename] <- getArgs
         (nc,nw,nl) <- wc filename
         putStrLn ""
         putStrLn (show nc)
         putStrLn ""
         putStrLn (show nw)
         putStrLn ""
         putStrLn (show nl)
         putStrLn ""
         putStrLn filename
         putStrLn "\n"
```

Error Handling

Monad can abort if an error occurs. Can add a function to handle errors:

```haskell
catch :: IO a -> (IOError -> IO a) -> IO a
ioError :: IOError -> IO a
fail :: String -> IO a

catch echo (\err ->
    fail ("I/O error: "++show err))
```
An Example

```haskell
processFile fileName =  
    getContents fileName >>= \inp ->  
    print (processInput inp)

main =  
    putStrLn "Give me a file name" >>>
    getLine >>= \fileName ->
    catch (processFile f)  
        (\err ->
            print err >>
            main)
```

The Modularity Problem

Inserting a print (say for debugging):

```haskell
sqrt :: Float -> Float
sqrt x =  
    let ...  
        a = (putStrLn ...) :: IO String  
    in result
```

The binding does nothing!  
The I/O has to be exposed to the caller:

```haskell
sqrt :: Float -> IO Float
sqrt x =  
    let ...  
        a = (putStrLn ...) :: IO String  
    in a >>= return result
```
Monadic I/O is Sequential

```haskell
do (nc1,nw1,nl1) <- wc filename1
   (nc2,nw2,nl2) <- wc filename2
return (nc1+nc2, nw1+nw2, nl1+nl2)
```

The two wc calls are totally independent but the IO they perform must be sequentialized! We can imagine doing them in parallel:

```haskell
parIO :: IO a -> a

let (nc1,nw1,nl1) = parIO (wc filename1)
    (nc2,nw2,nl2) = parIO (wc filename2)
in (nc1+nc2, nw1+nw2, nl1+nl2)
```

Overcoming the Problems

The limitations are fundamental and can be overcome only by abandoning the purely functional character of the language.

```haskell
let (nc1,nw1,nl1) = doIO (wc filename)
    writeFile filename "Hello World!\n" 
    (nc2,nw2,nl2) = doIO (wc filename)
in (nc1+nc2, nw1+nw2, nl1+nl2)

let (nc1,nw1,nl1) = doIO (wc filename)
    writeFile filename "Hello World!\n" 
    (nc2,nw2,nl2) = (nc1,nw1,nl1)
in (nc1+nc2, nw1+nw2, nl1+nl2)
```

Suddenly program semantics are much more fuzzy!
**Monadic sequencing**

\[
\text{return } a \mathbin{\gg\gg} \lambda x \to m \equiv (\lambda x \to m) \ a
\]

\[
m \mathbin{\gg\gg} \lambda x \to \text{return } x \equiv m
\]

\[
(m \mathbin{\gg\gg} \lambda x \to n) \mathbin{\gg\gg} \lambda y \to o
\equiv m \mathbin{\gg\gg} \lambda x \to (n \mathbin{\gg\gg} \lambda y \to o)
\quad x \notin \text{FV}(o)
\]

True in every monad by definition.

A derived axiom:

\[
m \mathbin{\gg} (n \mathbin{\gg} o) \equiv (m \mathbin{\gg} n) \mathbin{\gg} o
\]

**Monads and Let**

Monadic binding behaves like let:

\[
\text{return } a \mathbin{\gg\gg} \lambda x \to m \equiv (\lambda x \to m) \ a
\]

\[
m \mathbin{\gg\gg} \lambda x \to \text{return } x \equiv m
\]

\[
(m \mathbin{\gg\gg} \lambda x \to n) \mathbin{\gg\gg} \lambda y \to o
\equiv m \mathbin{\gg\gg} \lambda x \to (n \mathbin{\gg\gg} \lambda y \to o)
\quad x \notin \text{FV}(o)
\]

\[
\text{let } x = a \text{ in } m \equiv (\lambda x \to m) \ a
\]

\[
\text{let } x = m \text{ in } x \equiv m
\]

\[
\text{let } y = (\text{let } x = m \text{ in } n) \text{ in } o
\equiv \text{let } x = m \text{ in } (\text{let } y = n \text{ in } o)
\quad x \notin \text{FV}(o)
\]

http://www.csg.lcs.mit.edu/6.827