Using Monads for Input and Output

Arvind
Jan-Willem Maessen
Laboratory for Computer Science
M.I.T.
November 4, 2002

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?

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

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

What about modularity?

Barriers are too coarse-grained:

```
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

```
main :: [string]
main = ["Hello", "world!"]
```

or

```
main = let a = "Hello"
           b = "World!"
       in [a,b]
```

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!

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

main = 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:

```
(>>)     :: IO () -> IO () -> IO ()
```

Example:

```
putChar 'H' >> putChar 'i' >>
putChar '!'                    :: IO ()

putString :: String -> IO ()
putString ""      = done
putString (c:cs) =
  putChar c >> putString cs
```

Monads: Composing Actionspecs

We need some way to get at the results of `getChar`

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

We read the "bind" operator as follows:

x_1 `>>= \a ->` x_2

- Perform the action represented by x_1, producing a value of type "a"
- Apply function `\a ->` 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: `getChar >>= putChar`
the same as `getChar >>= \c -> putChar c`

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

IO a: computation which does some I/O,
then produces a value of type **a.**

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

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

```
do e                  -> e

do e ; dostmts        -> e >> do dostmts

do p<-e ; dostmts     -> e >>= \p-> do dostmts

do let p=e ; dostmts  -> let p=e in do dostmts


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

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)
wc 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

```
main :: IO ()

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

Error Handling

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

```
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

```
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):
```
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:

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

Monadic I/O is Sequential

```
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:

```
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 over-
come only by abandoning the purely functional
character of the language.

```
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

```
return a >>= \x -> m  ≡  (\x -> m) a

m >>= \x -> return x  ≡  m

(m >>= \x -> n) >>= \y -> o
        ≡ m >>= \x -> (n >>= \y -> o)
                            x ∉ FV(o)
```

True in every monad by definition.

A derived axiom:

```
m >> (n >> o) ≡ (m >> n) >> o
```

Monads and Let

Monadic binding behaves like let:

```
return a >>= \x -> m ≡ (\x -> m) a
m >>= \x -> return x ≡ m
(m >>= \x -> n) >>= \y -> o
      ≡ m >>= \x -> (n >>= \y -> o)
                              x ∉ FV(o)

let x = a in m       ≡  (\x -> m) a
let x = m in x       ≡  m
let y = (let x = m in n) in o
      ≡ let x = m in (let y = n in o)
                              x ∉ FV(o)
```