Monadic Programming

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

October 26, 2006

Outline

- IO Monad
  - Example: wc program
  - Some properties
- Monadic laws
- Creating our own monads:
  - Id: The simplest monad
  - State
  - Unique name generator
  - Emulating simple I/O
  - Exceptions

Monadic I/O

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 a rage in the FL community.

Word Count Program

```
wcs  :: String -> Bool -> (Int,Int,Int)
                          -> (Int,Int,Int)

wcs []     inWord (nc,nw,nl) = (nc,nw,nl)
wcs (c:cs) inWord (nc,nw,nl) =
  if (isNewLine c) then
     wcs cs False ((nc+1),nw,(nl+1))
  else if (isSpace c) then
     wcs cs False ((nc+1),nw,nl)
  else if (not inWord) then
     wcs cs True ((nc+1),(nw+1),nl)
  else
     wcs cs True ((nc+1),nw,nl)
```

Can we read the string from an input file as needed?

File Handling Primitives

```
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
```


Monadic Word Count Program

file name

```
wc       :: String -> IO (Int,Int,Int)

wc filename =
     openFile filename ReadMode >>= \h ->
     wch h False (0,0,0) >>= \(nc,nw,nl) ->
     hClose h >>
     return (nc,nw,nl)

wch  :: Handle -> Bool -> (Int,Int,Int)
                       -> IO (Int,Int,Int)
wcs  :: String -> Bool -> (Int,Int,Int)
                       ->  (Int,Int,Int)
```

Monadic Word Count Program *cont.*

```
wch  :: Handle -> Bool -> (Int,Int,Int)
                             -> IO (Int,Int,Int)
wch h inWord (nc,nw,nl) =
  hIsEOF h >>= \eof ->
     if eof then return (nc,nw,nl)
     else
       hGetChar h >>= \c ->
          if (isNewLine c) 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 =    getArgs >>= \[filename] ->
          wc filename >>= \(nc,nw,nl) ->
          putStr “   ” >>
          putStr (show nc) >>
          putStr “   ” >>
          putStr (show nw) >>
          putStr “   ” >>
          putStr (show nl) >>
          putStr “   ” >>
          putStr filename >>
          putStr “\n”
```

Once a value enters the IO monad it cannot leave it!

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))
```

October 26, 2006 http://www.csg.csail.mit.edu/6.827 L13-9

The Modularity Problem

Inserting a print (say for debugging):

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

Without the binding has no effect; the I/O has to be exposed to the caller:

One print statement changes the whole structure of the program!

October 26, 2006 http://www.csg.csail.mit.edu/6.827 L13-10

Monadic I/O is Sequential

```
wc filename1 >>= \(nc1,nw1,nl1) ->
wc filename2 >>= \(nc2,nw2,nl2) ->
return (nc1+nc2, nw1+nw2, nl1+nl2)!
```

The two wc calls are totally independent but the IO they perform must be sequentialized!

Monadic I/O is not conducive for parallel operations

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
```

```
wc filename1 >>= \(nc1,nw1,nl1) ->
wc filename2 >>= \(nc2,nw2,nl2) ->
return (nc1+nc2, nw1+nw2, nl1+nl2)
```

versus

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

Are these program meaningful?

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

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


Monadic Laws

```
1. do x <- return a ; m ≡ (\x -> do m) a

2. do x <- m ; return x ≡  m

3. do y <- (do x <- m ; n) ; o
                ≡  do x <- m; (do y <- n; o)
                     x ∉ FV(o)
```

True for all monads. Only primitive operations distinguish monads from each other

```
do m ; n     ≡  do _ <- m ; n
```

A derived axiom:

```
m >> (n >> o) ≡ (m >> n) >> o
```

Properties of programs involving IO

```
putString []     = return ()
putString (c:cs) = putChar c >> putString cs

[]     ++ bs     = bs
(a:as) ++ bs     = a : (as ++ bs)
```

```
putString as >> putString bs
                  ≡ putString (as++bs)
```

One can prove this just using monadic laws without involving I/O properties

Monads and Let

```
1. do x <- return a ; m ≡ (\x -> do m) a
2. do x <- m ; return x ≡  m
3. do y <- (do x <- m ; n) ; o
              ≡  do x <- m; (do y <- n; o)
                                   x ∉ FV(o)
```

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

Monadic binding behaves like let

Monads and Let

- Relationship between monads and let is deep
- This is used to embed languages inside Haskell
- IO is a special sublanguage with side effects

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a            --*
```


Fib in Monadic Style

```
fib n =
  if (n<=1) then n
  else
    let
      n1 = n - 1
      n2 = n - 2
      f1 = fib n1
      f2 = fib n2
    in f1 + f2
```

```
fib n =
  if (n<=1) then return n
  else
    do
     n1 <- return (n-1)
     n2 <- return (n-2)
     f1 <- fib n1
     f2 <- fib n2
     return (f1+f2)
```

Note the awkward style: everything must be named!

Outline

- IO Monad
  - Example: wc program
  - Some properties
- Monadic laws
- Creating our own monads: ←
  - Id: The simplest monad
  - State
  - Unique name generator
  - Emulating simple I/O
  - Exceptions

Id: The Simplest Monad

```
newtype Id a = Id a

instance Monad Id where
  return a    = Id a
  Id a >>= f =  f a

runId (Id a) = a
```

- This monad has no special operations!
- Indeed, we could just have used **let**
- The **runId** operation runs our computation

  For IO monad **run** was done outside the language

The State Monad

- Allow the use of a single piece of mutable state

```
put :: s -> State s ()
get :: State s s

runState :: s -> State s r -> (s,r)

instance Monad (State s)
```

October 26, 2006 http://www.csg.csail.mit.edu/6.827 L13-21

The State Monad: Implementation

```
newtype State s r = S (s -> (s,r))

instance Monad (State s) where
  return r  = S (\s -> (s,r))
  S f >>= g = S (\s -> let (s’, r) = f s
                           S h     = g r
                       in  h s’)

get              = S (\s -> (s,s))
put s            = S (\o -> (s,()))
runState s (S c) = c s
```

October 26, 2006 http://www.csg.csail.mit.edu/6.827 L13-22

Generating Unique Identifiers

```
type Uniq = Int
type UniqM = State Int

runUniqM :: UniqM r -> r
runUniqM comp = snd (runState 0 comp)

uniq :: UniqM Uniq
uniq = do u <- get
          put (u+1)
          return u
```


Poor Man's I/O

```
type PoorIO a = State (String, String)

putChar :: Char -> PoorIO ()
putChar c = do (in, out) <- get
               put (in, out++[c])

getChar :: PoorIO Char
getChar = do (in, out) <- get
             case in of
               a:as -> do put (as, out)
                          return a
               []   -> fail "EOF"
```

Error Handling using Maybe

```
instance Monad Maybe where
  return a = Just a
  Nothing >>= f = Nothing
  Just a  >>= f = f a
  fail _   = Nothing

Just a  `mplus` b = Just a
Nothing `mplus` b = b

do m' <- matrixInverse m
   y  <- matrixVectMult m x
   return y
```


Combining Monads

- To simulate I/O, combine State and Maybe.
- There are two ways to do this combination:

```
newtype SM s a = SM (s -> (s, Maybe a))
newtype MS s a = MS (s -> Maybe (s, a))
```

	SM	MS
	([],"")	([],"")
do putChar 'H'	([],"H")	([],"H")
a <- getChar	([],"H")	Nothing
putChar 'I'	*skipped*	
`mplus` putChar '!'	([],"H!")	([],"!")

Special Monads

- Operations inexpressible in pure Haskell
- IO Monad
  - Primitives must actually call the OS
  - Also used to embed C code
- State Transformer Monad
  - Embeds *arbitrary* mutable state
  - Alternative to M-structures + barriers

Extras

Monadic Laws

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

2. m >>= \x -> return x  ≡  m

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

True in every monad by definition. Primitive monadic operators distinguish one monad from another

A derived axiom:

```
m >> (n >> o) ≡ (m >> n) >> o
```


Base case

```
putString []     = return ()

[]     ++ bs     = bs

putString [] >> putString bs
 ≡ return () >> putString bs
 ≡ putString bs
 ≡ putString ([]++bs)
```

Inductive case

```
putString (a:as) = putChar a >> putString as

(a:as) ++ bs     = a : (as ++ bs)

putString (a:as) >> putString bs
 ≡ (putChar a >> putString as) >> putString bs
 ≡ putChar a >> (putString as>>putString bs)
 ≡ putChar a >> (putString (as ++ bs))
 ≡ putString (a : (as ++ bs))
 ≡ putString ((a:as) ++ bs)
```