Imperative Programming in Haskell?

Armando Solar-Lezama

Computer Science and Artificial Intelligence Laboratory

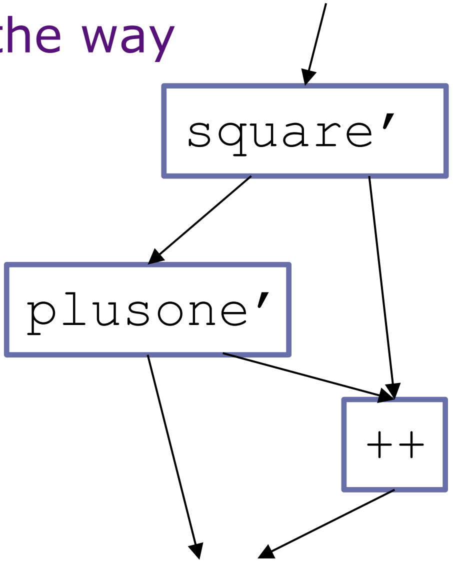
MIT

With content from Nirav Dave (used with permission) and
examples from Dan Piponi's great blog Post
"You could have invented Monads! And Maybe You Already Have"
http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html

October 7, 2015

Debuggable Functions

- Suppose you want your function to produce some debug output in addition to computation
  - e.g. plusone' x = (x+1 , "added one")
    square' x = (x*x , "squared")
  - convenient but…
- How do you compose two such functions?
  - (plusone' (square' x)) doesn't type check the way (plusone (square x)) would.
- Composition just became a pain
  - let (y,s) = square' x
    (z,t) = plusone' y in (z , s ++ t)



Debuggable Functions

- Make a function to facilitate this
  - bind:: (Int->(Int, String))->(Int, Sting)->(Int, String)
  - bind f (x,y) = let (u,v) = f(x) in (u, y++v)

- Ex.
  - (bind square')
  - (bind plusone')
  - (bind square') ((bind plusone') (x, “”))

Debuggable Functions

- Two more useful functions
  - unit x = (x, “”)
  - lift f x = unit (f x)
  - (*) f g = (bind f) . (g) = \x ((bind f) (g x))

- Some useful identities
  - unit * f = f * unit = f
  - lift f * lift g = lift (f . g)

Random Numbers

- Consider the “function” rand()
  - Not really a function, but you can make it a function
- rand: StdGen->(int, StdGen)
  - think of StdGen as the seed that gets updated (or as some infinitely long list of pre-generated random numbers)
- A randomized function a->a is really
  a -> StdGen -> (a, StdGen)

Composing Randomized Functions

- Again, composing randomized functions is a pain

- We can define a form of bind
  - Recall the pattern from before
    - bind :: (a -> something) -> something -> something
  - So we can do this for randomized functions
    - bind::
(a -> StdGen -> (a, StdGen))->
  (StdGen -> (a, StdGen))->(StdGen -> (a, StdGen))
    - bind f x seed = let (x',seed') = x seed in f x' seed'

Randomized Functions

- Ex.
  - plusrand x seed = let (rv, seed’) = random seed
    in (x + rv , seed’)
  - timesrand x seed = let (rv, seed’) = random seed
    in (x * rv , seed’)
- Let’s say I want 5 * rnd + rnd
  - (bind plusrand) ((bind timesrand) ??)
- unit
  - unit :: a -> something
  - unit :: a -> (StdGen -> (b, StdGen))
  - unit x g = (x,g)
  - (bind plusrand) ((bind timesrand) (unit 5))

Lift and composition

- We can again define
  - lift f x = unit (f x)
  - (*) f g = (bind f) . g

- And again it is true that
  - unit * f = f * unit = f
  - lift f * lift g = lift (f . g)

Monads as a type class

- Monad is a typeclass that requires
  - x >>= f
    - (>>=) :: something->(a->something)->something
    - (equivalent to bind f x)
  - return x
    - return :: a -> something
    - (equivalent to unit x)
  - etc.

- So in the rand example
  - (bind plusrand) ((bind timesrand) (unit 5)) becomes
  - (return 5) >>= timesrand >>= plusrand

Monad definition

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

- (m a), (m b) correspond to *something*
- Eg.
  - type MyRand a = StdGen -> (a, StdGen)

Monadic Laws

- They operators are expected to satisfy some rules
  - Left Identity:
    - return a >>= f ⇔ f a
    - i.e. unit * f = f
  - Right Identity
    - m >>= return ⇔ m
    - i.e. f*unit = f
  - Associativity
    - (m >>= f) >>= g ⇔ m >>= (\x -> f x >>= g)

do syntactic sugar for Monads

do e	⇒	***e***
do p <- e ; dostmts	⇒	***e >>= \p-> do dostmts***
do e ; dostmts	⇒	***e >>= _-> do dostmts***
do let p=e ; dostmts	⇒	***let p=e in do dostmts***

```
(return 5) >>= timesrand >>= plusrand
```

```
do x <- return 5;
   y <- timesrand x;
   plusrand y
```

do syntactic sugar for Monads

`do e`	⇒ `e`
`do p <- e ; dostmts`	⇒ `e >>= \p-> do dostmts`
`do e ; dostmts`	⇒ `e >>= \_-> do dostmts`
`do let p=e ; dostmts`	⇒ `let p=e in do dostmts`

```
do x <- return 5;
   y <- timesrand x;
   plusrand y
```

```
return 5 >>= \x (do y <- timesrand x;
                     plusrand y)
```

do syntactic sugar for Monads

do e	⇒ **e**
do p <- e ; dostmts	⇒ **e >>= \p-> do dostmts**
do e ; dostmts	⇒ **e >>= _-> do dostmts**
do let p=e ; dostmts	⇒ **let p=e in do dostmts**

```
return 5 >>= \x (do y <- timesrand x;
                    plusrand y)
```

```
return 5 >>= \x (timesrand x >>= \y do plusrand y)
```

do syntactic sugar for Monads

```
do e                          ⇒ e
do p <- e ; dostmts           ⇒ e >>= \p-> do dostmts
do e ; dostmts                ⇒ e >>= \_-> do dostmts
do let p=e ; dostmts          ⇒ let p=e in do dostmts
```

```
return 5 >>= \x (timesrand x >>= \y do plusrand y)
```

```
    return 5 >>= \x (timesrand x >>= \y plusrand y)
                                   Int->MyRand Int
```

do syntactic sugar for Monads

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

```
return 5 >>= \x (timesrand x >>= \y do plusrand y)
```

```
return 5 >>= \x (timesrand x >>= \y plusrand y)
```

MyRand Int

Int->MyRand Int

MyRand Int

Int -> MyRand Int

IO with Monads

Word Count Program

Flag to indicate we are inside a word

```
wc   :: String -> (Int,Int,Int)
wcs  :: String -> Bool -> (Int,Int,Int)
                                -> (Int,Int,Int)
wc cs = wcs cs False (0,0,0)

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)
```

Word Count Program

Flag to indicate we are inside a word

```
wc    :: String -> (Int,Int,Int)
wcs   :: String -> Bool -> (Int,Int,Int)
                              -> (Int,Int,Int)
wc cs = wcs cs False (0,0,0)

wcs []      inWord (nc,nw,nl) = (nc,nw,nl)
wcs (c:cs) inWord (nc,nw,nl) =
   if (isNewLine c) then
                                       +1))
   els
      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)
```

Suppose we want to read the string from an input file

File Handling Primitives

```
type Filepath = String
data IOMode = ReadMode | WriteMode | ...
data Handle = ... implemented as built-in type

openFile :: FilePath -> IOMode -> Handle
hClose   :: Handle -> () -- void
hIsEOF   :: Handle -> Bool
hGetChar :: Handle -> Char
```

These operations are not pure functions because each causes a side-effect. For example, **(hGetChar h)** should produce different answers if performed twice

Reading a File - First Attempt

```
getFileContents :: String -> String
getFileContents filename =
  let h = openFile filename ReadMode
      reversed_cs = readFileContents h []
      () = hClose h
  in (reverse reversed_cs)

readFileContents :: Handle -> String -> String
readFileContents h rcs =
  if (not (hIsEOF h))
   then ""
   else readFileContents h ((hGetChar h):rcs)
```

Bogus sequential code; no way to model or control side-effects

Ugly, yes, but may still be okay

- Issue: If we rely on strict execution, this cannot be simplified

```
let unused = bigComputation input
in 2
```

- To this…

```
2
```

Monads: A Review

Monad is a type class with the following operations

```
class (Monad m) where
  -- embedding
  return :: a -> m a
  -- sequencing
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
```

Monads let us lift a normal computation into a computational context and selectively access the context through primitive actions

Monadic I/O

- By embedding the concept of I/O in a monad we guarantee that there is a single sequence of the monadic I/O operations (no nondeterminism issues)

> **`IO a`**: computation which does some I/O, producing a value of type **`a`**.

- Unlike other monads, there is no way to make an **`IO a`** into an **`a`**

No operation to take a value out of an IO

Monadic I/O

```
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
```

Primitives

```
getFileContents :: String -> IO String
getFileContents filename =
  do h <- openFile filename ReadMode
     reversed_cs <- readFileContents h []
     hClose h
     return (reverse reversed_cs)
readFileContents :: Handle -> String -> IO String
readFileContents h rcs =
  do b <- hIsEOF h
     if (not b) then return []
                else do c <- hGetChar h
                        readFileContents h (c:rcs)
```

reading a file

Monadic vs bogus code

```
getFileContents :: String -> IO String
getFileContents filename =
  do h <- openFile filename ReadMode
     reversed_cs <- readFileContents h []
     hClose h
     return (reverse reversed_cs)
```

Contrast

```
getFileContents filename =
  let h = openFile filename ReadMode
      reversed_cs = readFileContents h []
      () = hClose h
  in (reverse reversed_cs)
```

Monadic printing: an example

```
print filename (nc,nw,nl) =
    do
      putStr "   "
      putStr (show nc)
      putStr "   "
      putStr (show nw)
      putStr "   "
      putStr (show nl)
      putStr "   "
      putStr filename
      putStr "\n"

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

no return !

Word Count using monads - version 1

```
main = do
   (filename:_) <- getArgs
   contents     <- getFileContents filename
   let (nc,nw,nl) = wc contents
   print filename (nc,nw,nl)
```

pure functional program

versus

```
main = do
   (filename:_) <- getArgs
   contents     <- getFileContents filename
   (nc,nw,nl)   <- return (wc contents)
   print filename (nc,nw,nl)
```

What if we wanted to compute wc as we read the file ?

Monadic Word Count Program
version 2

file name

```
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)

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) =
   do eof <- hIsEOF h
      if eof then return (nc,nw,nl)
      else do
        c <- hGetChar h
        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)
```

Beyond I/O

- Monadic I/O is a clever way to force meaningful interactions with the outside world. This is what most people think of when they think of monads

- But monads can do more
  - A mechanism to structure computation
  - A way to "thread information" through a program

Fib: Functional vs Monadic Style

```
fib :: Int -> Int
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 :: (Monad m) => Int -> m Int
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)
```

- monadic fib will work inside any other monadic computation!
- note the awkward style: everything must be named because computations cannot be inlined!

Limitations: The Modularity Problem

Inserting a print (say for debugging):

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

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

One print statement changes the whole structure of the program!