Using Monads to Structure Computation

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

November 6, 2002

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

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 actions:

```
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.

Monadic sequencing

```
a >> b                    ≡  a >>= (\_ -> b)

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)
```

A derived axiom:

```
m >> (n >> o) ≡ (m >> n) >> o
```


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

do a ; b                ≡  do _ <- a ; b
do x <- return a ; b ≡  (\x -> do b) a
do x <- m ; return x ≡  m
do y <- (do x <- m ; n) ; o
                ≡  do x <- m; (do y <- n; o)
```


Monads and Let

Monadic binding behaves like let:

```
do a ; b                ≡  do _ <- a ; b
do x <- return a ; b    ≡  (\x -> do b) a
do x <- m ; return x    ≡  m
do y <- (do x <- m ; n) ; o
      ≡  do x <- m; (do y <- n; 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)
```


Monads and Let

- Relationship between monads and let is deep
- Use this 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              --*
```


Outline

- Monadic operations and their properties
- Reasoning about monadic programs
- Creating our own monads:
  - Id: The simplest monad
  - State
  - Supplying unique names
  - Emulating simple I/O
  - Exceptions
- Composing monad transformers
- IO and ST: two very special monads
- Using ST for imperative computation
- Ordering issues

Proving simple properties

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

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

Show:

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


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)
```


Representation Independence

- Our proof did not depend on the behavior of I/O!
- Uses properties of Monads
- Requires *some* function

```
putChar :: Char -> m ()
```

A monadic computation has two sets of operations:

- The monadic operations, with general properties
- Specific operations with unique properties

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 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!

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

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)
```


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
```


State

```
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
```


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!")   ([],"!")
```


Monad Transformers

- State and error handling are separate features
- We can plug them together in multiple ways
- Other monads have a similar flavor
- Monad Transformer: add a feature to a Monad.

```
instance (Monad m) => Monad (ErrorT m)
instance (Monad m) => Monad (StateT s m)

type ErrorM = ErrorT Id
type StateM s = StateT s Id
type SM s a = StateT s (ErrorT Id)
type MS s a = ErrorT (StateT s Id)
```


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

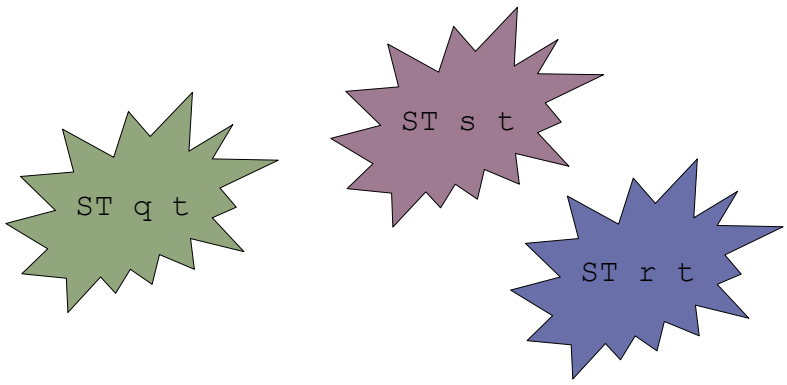
The State Transformer Monad

```
instance Monad (ST s)

newSTRef   :: a -> ST s (STRef s a)
readSTRef  :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()

runST :: (∀s. ST s a) -> a
```

- The special type of `runST` guarantees that an `STRef` will not escape from its computation.

Independent State Transformers

- In `ST s t`, the type `s` represents the "world."
- We can have multiple independent worlds.
- The type of `runST` keeps them from interacting.



Mutable lists using ST

We can create as many mutable references as we like, allowing us to build mutable structures just as we would with I- and M-cells.

```
data RList s t = RNil
               | RCons t (STRef s t)

rCons :: t-> RList s t-> ST s (RList s t)
rCons t ts = do r <- newSTRef ts
                return (RCons t r)
```


Insert using RList

```
insertr RNil            x = rCons x RNil
insertr ys@(RCons y yr) x =
  if x==y then return ys
  else do ys' <- readSTRef yr
          ys'' <- insertr ys' x
          writeSTRef yr ys''
          return ys
```


Graph traversal: *ST notebook*

```
data GNode = GNode NodeId Int [GNode]

rsum node = do
  nb <- mkNotebook
  let rsum' (GNode x i nbs) = do
        seen <- memberAndInsert nb x
        if seen
        then return 0
        else do nbs' <- mapM rsum' nbs
                return (i + sum nbs')
```


A traversal notebook

```
type Notebook s = STRef s (RList s Nodeid)

mkNotebook = newSTRef RNil

memberAndInsert nb id = do
  ids <- readSTRef nb
  case ids of
    MNil -> do t <- rCons id MNil
               writeSTRef nb t
               return False
    MCons id' nb'
      | id==id'   = return True
      | otherwise = memberAndInsert nb' id
```


Problems with Monadic Style

- We need a new versions of common functions:

```
mapM f []     = return []
mapM f (x:xs) = do
  a <- f x
  as <- mapM f xs
  return (a:as)

mapM' f []     = return []
mapM' f (x:xs) = do
  as <- mapM' f xs
  a  <- f x
  return (a:as)
```


Monads and Ordering

- Monads aren't inherently ordered (`Id`)
- But stateful computations must be ordered
- For ST and IO, at least the side-effecting computations are ordered.
- The `unsafeInterleaveIO` construct relaxes this ordering, but is impure.

- On the other hand, barriers order *all* computation, including non-mondic execution.

There is still room for experimentation!