The State Monad and Friends

Nirav Dave
Computer Science and Artificial Intelligence Laboratory
M.I.T.

October 31, 2006

Outline

- What's a Monad?
- Some standard monads:
  - Id: The simplest monad
  - Maybe
  - State
- Some graph algorithms in monadic style
  - list insertion
  - graph traversal
- Composing monad transformers

What is a monad and Why should I care?

- A monad is an abstraction for linear compositions
  - Any linear composition can be represented via a monad
- So it restricts our parallelism?
  - Not always. It does provide a linear ordering for meaning, but it ***could*** *restrict the computational parallelism (much like a barrier could)*

Justifying Monadery

- So the monad typeclass is effectively sugar (doesn't add any power)?
  - **Yes:** the monad typeclass is effectively sugar
- I don't need abstractions. So why should I use monads?
  - Certain monads cannot be described inside the language (e.g. IO)
  - Because they allow us to **safely** add power to the language
- Why did we pick this abstraction?
  - Let's us keep determinism if desired

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

When we define our own Monads, we have to give the definition of **>>=** and **return** and make sure that those definitions follow these laws.

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

Maybe: Encapsulating failure

```
newtype Maybe a = Nothing
                | Just a

instance Monad Maybe where
  return a = Just a

  Nothing  >>= f = Nothing
  (Just a) >>= f = f a

  fail str = Nothing

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

Propagate failure

Prioritize valid results

Using Maybe: a micro-parser

```
readInt    :: Stream -> Maybe (Int,Stream)
readStr    :: Stream -> String -> Maybe Stream

readITuple :: Stream -> Maybe ((Int,Int), Stream)
readITuple s0 = do
    s1     <- readStr s0 "("
    (i,s2) <- readInt s1
    s3     <- readStr s2 ","
    (j,s4) <- readInt s3
    s5     <- readStr s4 ")"
    return ((i,j),s5)
```

If any step fails, the whole read fails

```
(readInt s) `mplus` (readITuple s)
```

The State Monad

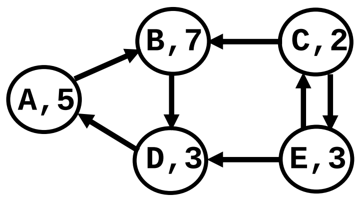
- Allow the use of a single piece of mutable state

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

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

instance Monad (State s)
```


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, ()))
evalState s (S c) = snd(c s)
```

Graph Traversal




```
data GNode =
      GNode {gid  :: Nodeid,
             gval :: Int,
             gnbrs:: [GNode] }
a = GNode "A" 5 [b]
b = GNode "B" 7 [d]
c = GNode "C" 2 [b]
d = GNode "D" 3 [a]
e = GNode "E" 3 [c,d]
```

Write function **rsum** to sum the nodes reachable from a given node.

rsum a ==> 15 **?**

rsum in a monadic style

```
rsum :: GNode -> Int
rsum gnode = let
     nb_sum = rsum' 0 gnode
   in
     evalState nb_sum empty_nb

rsum' :: Int -> GNode -> (SN Int)
rsum' n gnode = do
  let i = gid gnode
  b <- visitedN i
  if b then return n
   else do
    markN i
    n' <- foldM rsum' n (gnbrs gnode)
    return (n' + (gval gnode))
```

This is our state monad (points to: SN)

Check if we were here (points to: visitedN i)

Mark that we were here (points to: markN i)

SN: functional State

```
newtype Notebook = N [String]
type (SN i) = (State Notebook i)

new_nb = N []

markN :: String -> SN ()
markN i = do
  N ns <- get
  let ns' = i:ns
  _ <- put (N ns')
  return ()

visitedN :: String -> SN Bool
visitedN i = do
  N ns <- get
  return (elem i ns)
```


How “good” is this?

- Correctness?
  - Yes
- Readable?
  - Sure
- Parallelism
  - Same as a “normal” functional implementation
- Could a Notebook embedded in special monad be more efficient better than a functional one?

More on this a little later…

Combining Monads

- What do you do if you want to thread state and have a failure condition?
  - **(State Hash) Monad**
  - **Maybe Monad**

- Nest them?
  - type MS n = Maybe (State Hash n)
    (Maybe (Hash -> (Hash, n)))
  - type SM n = State Hash (Maybe n)
    (Hash -> Maybe (Hash, n))

Order Matters!

So let's give it a try…

Nesting Monads: SN and IO

```
printRSum :: GNode -> IO (Int)
printRSum gnode = do
  x <- printRSum' 0 gnode
  putStrLn "got " ++ (show x)

rsum' :: Int -> GNode -> IO (SN Int)
rsum' n gnode = do
  let i = gid gnode
  putStrLn "@Node" ++ (show i)
  return do
    b <- visitedN i
    if b then return n
         else do
           markN i
           n' <- foldM rsum' n (gnbrs gnode)
           return (n' + (gval gnode))
```

IO Monad (points to the first `do`)

SN Monad (points to `return do`)

Could we print here? (points to `n' <- foldM …`)

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

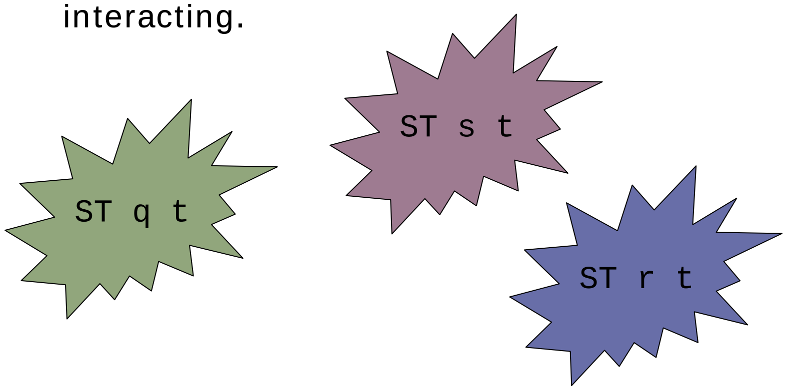
The State Transformer Monad

```
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 (list-based)

```
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
```

Once more ...

```
type Notebook s = (HashTable Nodeid ())

mkNotebook = new (==) HashString

memberAndInsert nb id = do
  munit <- lookup nb id
  case munit of
    Nothing   -> do
                   insert nb id
                   return False
    (Just _ ) -> return True
```

This type's accessor functions all return values in the IO monad

Problems with Monadic Style

- We need a new versions of common functions:

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

mapM’ :: (Monad m) -> (a -> m b) -> ([a] -> m [b])
mapM’ f []     = return []
mapM’ f (x:xs) = do
  as <- mapM’ f xs
  a  <- f x
  return (a:as)
```

What do these functions look like?

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-monadic execution.