Monads

COS 441 Slides 16

Agenda

- Last time:
  - We looked at implementation strategies for languages with errors, with printing and with storage
  - We introduced the concept of a monad, which involves 3 things:
    - What is the type of the result of evaluation?
      - ie: what type defines the monad type class instance
    - How do we evaluate a pure value and do nothing else?
      - ie: how do we implement "return"
    - How do we compose evaluation of two subexpressions
      - ie: how do we implement "bind": e >>= f

- This time:
  - How do we implement monads for printing and for storage?
  - Can we use monads more generally?

REVIEW:
THE ERROR MONAD

The Monad Typeclass

the type

```
class Monad m where
  return :: a -> m a                          -- the null computation

  (>>=) :: m a -> (a -> m b) -> m b           -- “bind” ie: composition
```

A useful derived operator:

```
  >> :: m a -> m b -> m b                     -- sequencing

  x >> y = (x >>= f)    where f _ = y
```

The Error Monad

- The error monad:

```
instance Monad Maybe where
  return v = Just v               -- an error-free computation that
                                  -- does nothing but return a value v

                                  -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  (Just v) >>= f  = f v           -- compose an error-free computation with f
  Nothing >>= f = Nothing         -- compose an error-full computation with f
```

The Error Monad

- The error monad:

```
instance Monad Maybe where
  return v = Just v              -- an error-free computation that
                                 -- does nothing but return a value v

                                 -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  (Just v) >>= f  = f v          -- compose an error-free computation with f
  Nothing >>= f = Nothing        -- compose an error-full computation with f
```

- Using the error monad:

```
eval (Val v) = return v

eval (Add e1 e2) = do
    x <- eval e1
    y <- eval e2
    return (x + y)
```

do
block

The Error Monad

- The error monad:

```
instance Monad Maybe where
  return v = Just v            -- an error-free computation that
                               -- does nothing but return a value v

                               -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  (Just v) >>= f  = f v        -- compose an error-free computation with f
  Nothing >>= f = Nothing      -- compose an error-full computation with f
```

- Using the error monad:

```
eval (Val v) = return v

eval (Add e1 e2) = do
    x <- eval e1
    y <- eval e2
    return (x + y)
```

do block

→

```
eval (Add e1 e2) =
 eval e1 >>= (\x.
 eval e1 >>= (\y.
 return (x + y)))
```

THE PRINTING MONAD

Recall Evaluation of Printing Expressions

```
data Expr3=
     Val3 Int
   | Add3 Expr3 Expr3
   | PrintThen String Expr3


eval3 :: Expr3 -> (String, Int)

eval3 (Val3 x) = ("", x)

eval3 (Add3 e1 e2) =
   let (s1,n1) = eval3 e1
       (s2,n2) = eval3 e2 in
   (s1 ++ s2, n1 + n2)

eval3 (PrintThen s e) =
   let (s', n) = eval3 e in (s ++ s', n)
```

null computation

plumbing that arises from composing computations that manipulate strings

Recall Evaluation of Printing Expressions

```
data Expr3=
      Val3 Int
    | Add3 Expr3 Expr3
    | PrintThen String Expr3
```

instance Monad $\underbrace{\text{(String, a)}}$ where

```
eval3 :: Expr3 -> (String, Int)

eval3 (Val2 x) = ("", x)

eval3 (Add3 e1 e2) =
    let (s1,n1) = eval3 e1
        (s2,n2) = eval3 e2 in
    (s1 ++ s2, n1 + n2)

eval3 (PrintThen s e) =
    let (s', n) = eval3 e in (s ++ s', n)
```

intuitively, we would like to make a monad from just a pair of a String and a return value a

however, we can only attach type classes to abstract types created using the data or newtype keywords

Recall Evaluation of Printing Expressions

```
data Expr3=
      Val3 Int
    | Add3 Expr3 Expr3
    | PrintThen String Expr3



eval3 :: Expr3 -> (String, Int)

eval3 (Val2 x) = ("", x)

eval3 (Add3 e1 e2) =
    let (s1,n1) = eval3 e1
         (s2,n2) = eval3 e2 in
    (s1 ++ s2, n1 + n2)

eval3 (PrintThen s e) =
    let (s', n) = eval3 e in (s ++ s', n)
```

```
newtype Output a = Out (String, a)

instance Monad Output where
```

Recall Evaluation of Printing Expressions

```
data Expr3=
      Val3 Int
    | Add3 Expr3 Expr3
    | PrintThen String Expr3


eval3 :: Expr3 -> (String, Int)

eval3 (Val2 x) = ("", x)

eval3 (Add3 e1 e2) =
    let (s1,n1) = eval3 e1
         (s2,n2) = eval3 e2 in
    (s1 ++ s2, n1 + n2)

eval3 (PrintThen s e) =
    let (s', n) = eval3 e in (s ++ s', n)
```

```
newtype Output a = Out (String, a)

instance Monad Output where
  return v = Out ("", v)

  (Out (s,v)) >>= f  =
     let (s',v') = f v in (s ++ s', v')
```

Recall Evaluation of Printing Expressions

```
data Expr3=
      Val3 Int
    | Add3 Expr3 Expr3
    | PrintThen String Expr3


eval3 :: Expr3 -> (String, Int)

eval3 (Val2 x) = ("", x)

eval3 (Add3 e1 e2) =
    let (s1,n1) = eval3 e1
         (s2,n2) = eval3 e2 in
    (s1 ++ s2, n1 + n2)

eval3 (PrintThen s e) =
    let (s', n) = eval3 e in (s ++ s', n)
```

```
newtype Output a = Out (String, a)

instance Monad Output where
  return v = Out ("", v)

  (Out (s,v)) >>= f  =
     let (s',v') = f v in (s ++ s', v')

  printme s = Out (s, ())
```

Recall Evaluation of Printing Expressions

```
data Expr3=
     Val3 Int
   | Add3 Expr3 Expr3
   | PrintThen String Expr3


eval3 :: Expr3 -> (String, Int)

eval3 (Val2 x) = ("", x)

eval3 (Add3 e1 e2) =
   let (s1,n1) = eval3 e1
       (s2,n2) = eval3 e2 in
   (s1 ++ s2, n1 + n2)

eval3 (PrintThen s e) =
   let (s', n) = eval3 e in (s ++ s', n)
```

```
newtype Output a = Out (String, a)

instance Monad Output where
  return v = Out ("", v)

  (Out (s,v)) >>= f  =
     let (s',v') = f v in (s ++ s', v')

  printme s = Out (s, ())

  eval3 (Val3 x) = return x

  eval3 (Add3 e1 e2) = do
     n1 <- eval3 e1
     n2 <- eval3 e2
     return (n1 + n2)

  eval3 (PrintThen s e) = do
     printme s
     eval3 e
```

Comparing Implementations of add

```
eval :: Expr1 -> Maybe Int

eval (Val1 x) = return x

eval (Add1 e1 e2) = do
   n1 <- eval e1
   n2 <- eval e2
   return (n1 + n2)
```

```
eval3 :: Expr3 -> Output Int

eval3 (Val3 x) = return x

eval3 (Add3 e1 e2) = do
    n1 <- eval3 e1
    n2 <- eval3 e2
    return (n1 + n2)
```

- The only difference is the type, which controls which monad we are evaluating inside of, using the type class mechanism
- We have isolated the essence of evaluating addition, independently of other side-effects such as errors or printing!
- This is deep! What an abstraction!

THE STATE MONAD

Recall Evaluation of a Stateful Language

```
data Expr4 =  Val4 | Add4 Expr4 Expr4 | StoreThen Expr4 Expr4 | Read

type State = Int
type Result a = State -> (State, a)
eval4 :: Expr4 -> Result a
```

implementing reading and writing requires a state transformer

```
eval4 (Val4 x) = \s -> (s, x)

eval4(Read) = \s -> (s, s)

eval4(Add e1 e2) =
   let f1 = eval4 e1
       f2 = eval4 e2 in
   \s0 -> let (s1, n1) = f1 s0
              (s2, n2) = f2 s1 in
          (s2, n1 + n2)
```

```
eval4 (StoreThen e1 e2) =
    let f1 = eval4 e1
        f2 = eval4 e2 in
    \s0 -> let (_, n1) = f1 s0 in
           f2 n1
```

The State Monad

```
type State = Int
newtype SM a = Transform (State -> (State, a))

instance Monad (SM) where
```

The State Monad

```
type State = Int
newtype SM a = Transform (State -> (State, a))

instance Monad (SM) where
   return x =                                          -- return :: a -> SM a
```

The State Monad

```
type State = Int
newtype SM a = Transform (State -> (State, a))

instance Monad (SM) where
    return x = Transform (\s -> (s,x))                -- return :: a -> SM a
```

The State Monad

```
type State = Int
newtype SM a = Transform (State -> (State, a))

instance Monad (SM) where
    return x = Transform (\s -> (s,x))             -- return :: a -> SM a

    (Transform t) >>= f =                          -- (>>=) :: SM a  -> (a -> SM b) -> SM b
                                                   -- t :: State -> (State, a), f :: a -> SM b
```

The State Monad

```
type State = Int
newtype SM a = Transform (State -> (State, a))

instance Monad (SM) where
   return x = Transform (\s -> (s,x))            -- return :: a -> SM a

   (Transform t) >>= f =                          -- (>>=) :: SM a  -> (a -> SM b) -> SM b
      Transform                                   -- t :: State -> (State, a) , f :: a -> SM b
         (\s -> ...                               -- s :: State
```

The State Monad

```
type State = Int
newtype SM a = Transform (State -> (State, a))

instance Monad (SM) where
   return x = Transform (\s -> (s,x))            -- return :: a -> SM a

   (Transform t) >>= f =                          -- (>>=) :: SM a  -> (a -> SM b) -> SM b
      Transform                                   -- t :: State -> (State, a) , f :: a -> SM b
         (\s ->                                   -- s :: State
             let (s', x) = t s in                 -- x :: a
             let (Transform g) = f x in           -- g :: State -> (State, b)
             g s')
```

The State Monad

```
type State = Int
newtype SM a = Transform (State -> (State, a))

instance Monad (SM) where
   return x = Transform (\s -> (s,x))                  -- return :: a -> SM a

   (Transform t) >>= f =                               -- (>>=) :: SM a  -> (a -> SM b) -> SM b
      Transform                                        -- t :: State -> (State, a) , f :: a -> SM b
         (\s ->                                        -- s :: State
             let (s', x) = t s in                      -- x :: a
             let (Transform g) = f x in                -- g :: State -> (State, b)
             g s')

getState :: SM State
getState = Transform (\s -> (s, s))

setState :: State -> SM ()
setState s' = Transform (\s -> (s', ()))
```

The State Monad

```
type State = Int
newtype SM a = Transform (State -> (State, a))

instance monad (SM) where
   return x = Transform (\s -> (s,x))

   (Transform t) >>= f =
      Transform
         (\s ->
              let (s', x) = t s in
              let (Transform g) = f x in
              g s')

getState :: SM State
getState = Transform (\s -> (s, s))

setState :: State -> SM ()
setState s' = Transform (\s -> (s', ()))
```

```
eval4 (Val4 x) = return x

eval4 (Read) = ...
```

The State Monad

```
type State = Int
newtype SM a = Transform (State -> (State, a))

instance monad (SM) where
    return x = Transform (\s -> (s,x))

    (Transform t) >>= f =
        Transform
            (\s ->
                let (s', x) = t s in
                let (Transform g) = f x in
                g s')

getState :: SM State
getState = Transform (\s -> (s, s))

setState :: State -> SM ()
setState s' = Transform (\s -> (s', ()))
```

```
eval4 (Val4 x) = return x

eval4 (Read) = getState
```

The State Monad

```
type State = Int
newtype SM a = Transform (State -> (State, a))

instance monad (SM) where
    return x = Transform (\s -> (s,x))

    (Transform t) >>= f =
        Transform
            (\s ->
                let (s', x) = t s in
                let (Transform g) = f x in
                g s')

getState :: SM State
getState = Transform (\s -> (s, s))

setState :: State -> SM ()
setState s' = Transform (\s -> (s', ()))
```

```
eval4 (Val4 x) = return x

eval4 (Read) = getState

eval4 (Add e1 e2) =
    n1 <- eval4 e1
    n2 <- eval4 e2
    return (n1 + n2)

eval4 (StoreThen e1 e2) = ...
```

The State Monad

```
type State = Int
newtype SM a = Transform (State -> (State, a))

instance monad (SM) where
    return x = Transform (\s -> (s,x))

    (Transform t) >>= f =
        Transform
            (\s ->
                let (s', x) = t s in
                let (Transform g) = f x in
                g s')

getState :: SM State
getState = Transform (\s -> (s, s))

setState :: State -> SM ()
setState s' = Transform (\s -> (s', ()))
```

```
eval4 (Val4 x) = return x

eval4 (Read) = getState

eval4 (Add e1 e2) =
    n1 <- eval4 e1
    n2 <- eval4 e2
    return (n1 + n2)

eval4 (StoreThen e1 e2) =
    n1 <- eval4 e1
    setState n1
    eval e2
```

The State Makeover: Before and After

Before the state monad:

```
eval4 (Val4 x) = \s -> (s, x)

eval4(Read) = \s -> (s, s)

eval4(Add e1 e2) =
    let f1 = eval4 e1
        f2 = eval4 e2 in
    \s0 -> let (s1, n1) = f1 s0
               (s2, n2) = f2 s1 in
           (s2, n1 + n2)

eval4 (StoreThen e1 e2) =
    let f1 = eval4 e1
        f2 = eval4 e2 in
    \s0 -> let (_, n1) = f1 s0 in
           f2 n1
```

After the state monad:

```
eval4 (Val4 x) = return x

eval4 (Read) = getState

eval4 (Add e1 e2) =
    n1 <- eval4 e1
    n2 <- eval4 e2
    return n1 + n2

eval4 (StoreThen e1 e2) =
    n1 <- eval4 e1
    setState n1
    eval e2
```

USING MONADS IN GENERAL-PURPOSE COMPUTATIONS

Why Did Haskell Implementers Bother?

- Did the Haskell implementers really invent monads and a special syntax just to make it easier to implement expression evaluators?
  - No! Of course not.
- Monads are used more generally to compose computations

Why Did Haskell Implementers Bother?

inputs

"Walker, David, Prof"

"Monsanto, Chris, TA"

"Kid, Wiz, Student"

"Wiz, Not"

""

Why Did Haskell Implementers Bother?

inputs

"Walker, David, Prof"

"Monsanto, Chris, TA"

"Kid, Wiz, Student"

"Wiz, Not"

""

Consider a simple string processing application:

```
csvConvert s = aux "" s
  where
    aux "" []         = []
    aux s  []         = [reverse s]
    aux s  (',' : cs) = reverse s : aux "" cs
    aux s  (c : cs)   = aux (c:s) cs
```

By the way, what is the type of csvConvert?

Why Did Haskell Implementers Bother?

inputs

```
"Walker, David, Prof"

"Monsanto, Chris, TA"

"Kid, Wiz, Student"

"Wiz, Not"

""
```

Consider a simple string processing application:

```
csvConvert s = aux "" s
  where
    aux "" []        = []
    aux s  []        = [reverse s]
    aux s  (',' : cs) = reverse s : aux "" cs
    aux s  (c : cs)  = aux (c:s) cs
```

Add a list indexing function:

```
index :: [a] -> Int -> Maybe a
index []       i    = Nothing
index (x:xs)   0    = Just x
index (x:xs)   i    = index xs (i-1)
```

Why Did Haskell Implementers Bother?

inputs

"Walker, David, Prof"

"Monsanto, Chris, TA"

"Kid, Wiz, Student"

"Wiz, Not"

""

```
csvConvert :: String -> [String]
index :: [a] -> Int -> Maybe a

getall :: String -> Maybe (String, String, String)
getall s =
```

Why Did Haskell Implementers Bother?

inputs

```
"Walker, David, Prof"

"Monsanto, Chris, TA"

"Kid, Wiz, Student"

"Wiz, Not"

""
```

```
csvConvert :: String -> [String]
index :: [a] -> Int -> Maybe a

getall :: String -> Maybe (String, String, String)
getall s =
  let items = csvConvert s in
  case index items 0 of
    Nothing -> Nothing
    Just last ->
      case index items 1 of
        Nothing -> Nothing
        Just first ->
          case index items 2 of
            Nothing -> Nothing
            Just role -> Just (last, first, role)
```

red is useful computation
blue is plumbing
so much plumbing!!!
plumbing just like the
plumbing in our evaluator!

Why Did Haskell Implementers Bother?

inputs

"Walker, David, Prof"

"Monsanto, Chris, TA"

"Kid, Wiz, Student"

"Wiz, Not"

""

```
csvConvert :: String -> [String]
index :: [a] -> Int -> Maybe a

getall :: String -> Maybe (String, String, String)
getall s = do
    let items = csvConvert s
    last <- index items 0
    first <- index items 1
    role <- index items 2
    return (first, last, role)
```

the Maybe monad takes care of the error-propagation plumbing!

Programming with Errors

- In general, we might have a whole bunch of functions that can produce errors:

```
getHead :: [a]  -> Maybe a
getTail :: [a] -> Maybe [a]
getStart :: [a] -> Maybe [a]
getLast :: [a] -> Maybe a
```

- We can string them together inside a monad that does error-propagation for us :

```
exchange :: [a] -> Maybe [a]

exchange xs =do
  x    <- getHead xs
  ys  <- getTail  xs
  zs  <- getStart ys
  z    <- getLast ys
  return (z ++ zs ++ x)
```

PROGRAMMING WITH STATE IN HASKELL

Programming with State

- In Java, a unique string generator:

global variable

no parameters

```
static int  n = 0;

String gen () {
       String s = "x" + Integer.toString(n);
       n = n + 1;
       return s;
}
```

- An analogous functional Haskell program:

global variable
becomes
parameter
and result

```
gen :: Int -> (String, Int)
gen n = ("x" ++ show n, n + 1)
```

Using State

- In Java (or C or most other imperative languages):

```
String s1, s2, s3;

s1 = gen()
s2 = gen()
s3 = gen()
```

- In Haskell:

```
let n0 = 0 in
let (s1, n1) = gen n0 in
let (s2, n2) = gen n1 in
let (s3, n3) = gen n3 in
....
```

plumbing!

- In functional languages, you have to manually thread the state through the program. Yuck! No wonder no one uses them!

A Generic State Monad

```
data ST s a = S (s -> (a, s))

apply        :: ST s a -> s -> (a, s)
apply (S f) x = f x
```

A Generic State Monad

```
data ST s a = S (s -> (a, s))

apply        :: ST s a -> s -> (a, s)
apply (S f) x = f x

instance Monad (ST s) where                    -- type State = ST s
  return x   = S (\s -> (x,s))                 -- return :: a -> State a


  st >>= f   = S transform                     -- (>>=)  :: State a -> (a -> State b) -> State b
                 where transform s =
                   let (x,s') = apply st s in
                   apply (f x) s'
```

Using the State Monad

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

tree :: Tree Char
tree = Node (Node (Leaf 'a') (Leaf 'b')) (Leaf 'c')

type State = ST Int        -- Int is the kind of state we'll use
gen :: State String
gen = S (\n -> ("x" ++ show n, n+1))

treeLabel :: Tree a -> State (Tree (String, a))
```

Using the State Monad

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

tree :: Tree Char
tree = Node (Node (Leaf 'a') (Leaf 'b')) (Leaf 'c')

type State = ST Int        -- Int is the kind of state we'll use
gen :: State String
gen = S (\n -> ("x" ++ show n, n+1))

treeLabel :: Tree a -> State (Tree (String, a))

treeLabel (Leaf x) = do
   s <- gen
   return (Leaf (s, x))

treeLabel (Node l r) = do
   l' <- mlabel l
   r' <- mlabel r
   return (Node l' r')
```

Using the State Monad

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

tree :: Tree Char
tree = Node (Node (Leaf 'a') (Leaf 'b')) (Leaf 'c')

type State = ST Int        -- Int is the kind of state we'll use
gen :: State String
gen = S (\n -> ("x" ++ show n, n+1))

treeLabel :: Tree a -> State (Tree (String, a))

treeLabel (Leaf x) = do
    s <- gen
    return (Leaf (s, x))

treeLabel (Node l r) = do
    l' <- mlabel l
    r' <- mlabel r
    return (Node l' r')
```

```
runST :: State a -> a
runST s = fst (apply s 0)

treelab :: Tree a -> Tree (String, a)
treelab t =
  runST (treeLabel t)
```

BUILT-IN STATE

IO Monad

- Haskell uses monads itself to structure its own evaluation:

```
main :: IO Char

main = do
   putStr “Hello”
   c <- getChar
   putStr
```

IO is a complex monad combining:

- printing to standard out
- reading files
- writing files
- reading command-line args
- writing mutable references (state)
- reading mutable references
- errors
- exceptions
- …
- all of the "effects" you find in ordinary languages

IO Monad

- Intuitively:

```
data ST s a = S (s -> (a, s))

type World = …

newtype IO a = ST World a
```

- Using the IO monad allows us to modify the "world"
  - files
  - stdout
  - …

Built-in Mutable References

```
import Data.IORef as R

new       = R.newIORef           -- create a new mutable object with 1 field
get       = R.readIORef          -- read out the stored object
r != n    = R.writeIORef r n     -- store n into reference r
```

Built-in Mutable References

```
import Data.IORef as R

new       = R.newIORef                 -- create a new mutable object with 1 field
get       = R.readIORef                -- read out the stored object
r != n    = R.writeIORef r n           -- store n into reference r

mkgenerator :: IO (IO String)

mkgenerator = do
  r <- new 0
  let gen = do n <- get r
               r != (n+1)
               return ("x" ++ show n))
  return gen
```

Built-in Mutable References

```
mkgenerator = do
  r <- new 0
  let gen = do n <- get r
               r != (n+1)
               return ("x" ++ show n))
  return gen
```

```
mynames = do
  gen <- mkgenerator
  x1 <- gen
  y1 <- gen
  gen' <- mkgenerator
  x2 <- gen'
  y2 <- gen'
  z1 <- gen
  return ([x1,y1,z1], [x2,y2])
```

in ghci:

```
*Main> mynames
(["x0","x1","x2"],["x0","x1"])
```

Mutable Data Structures

- A pure, functional binary search tree (not mutable):

```
BST key val = Null | Node key val (Tree a) (Tree a)
```

- A tree with mutable leaves:

```
import Data.IORef

MBST key val = Null | Node key (IORef val) (Tree a) (Tree a)

single key val = do
  r <- new v
  return (Node k r Null Null)
```

Mutable Data Structures

- A pure, functional binary search tree (not mutable):

```
BST key val = Null | Node key val (Tree a) (Tree a)
```

- A tree with mutable leaves:

```
import Data.IORef

MBST key val = Null | Node key (IORef val) (Tree a) (Tree a)

update :: (Ord key) => MBST key val -> key -> val -> IO Bool
update Null k v = return False
update (Node k' r' left right) k v =
  if k' == k then
    do {r' != v; return True}
  else if k' < k then
    update left k v
  else
    update right k v
```

return true if update succeeds
return false if not

Get me out of here!

- When we built our own state monad, we could extract the final value and throw away the state:

```
runST :: State a -> a
runST s = fst (apply s 0)
```

- When we work within the state monad, we can't do that.
- There is no safe function:

```
performIO :: IO a -> a
```

- But you can use *unsafe* IO:

```
import System.IO.Unsafe

unsafePerformIO :: IO a -> a
```

See: http://www.haskell.org/ghc/docs/latest/html/libraries/base/System-IO-Unsafe.html

SUMMARY

Summary

- We learned several things:
  - We can simplify the implementation of evaluators by using monads
  - We can simplify implementation and composition of more general computations using monads
    - errors using maybe
    - string creation (ie "printing")
    - state
    - …
  - Haskell has some built-in monads for handling effects, the most common being the IO monad
- The latter is at the core of how Haskell handles effects and yet still acts like a functional program and preserves powerful reasoning principles involving substitution of equals for equals