

Monads

COS 441 Slides 16

Reminder

- Assignment 4 is due Tuesday Nov 29

Agenda

- Last week we discussed operational semantics
 - lambda calculus basics
 - first-order abstract syntax
 - higher-order abstract syntax
 - recursive functions, if statements & booleans, numbers, pairs, sums (aka simple data types)
 - imperative languages with state and printing
- Recall, when it came to representing "state," we had to make the operational semantics more complicated:
 - pure: $\text{exp} \rightarrow \text{exp}$
 - state: $(\text{state}, \text{exp}) \rightarrow (\text{state}, \text{exp})$
 - printing: $(\text{string}, \text{exp}) \rightarrow (\text{string}, \text{exp})$
 - state and printing: $(\text{string}, \text{state}, \text{exp}) \rightarrow (\text{string}, \text{state}, \text{exp})$
- Today: monads: another technique for implementing operational semantics (and other stuff!)

ABSTRACTING COMPUTATION PATTERNS

Abstracting Computation Patterns

- Monads are another example of an abstraction of a very common computation pattern
- Let's review the idea. Consider the following two programs:

```
inc :: [Int] -> [Int]
inc [] = []
inc (n:ns) = n+1 : inc ns
```

```
sqr :: [Int] -> [Int]
sqr [] = []
sqr (n:ns) = n^2 : sqr ns
```

- What is the common functionality? How can we rewrite them to abstract out the commonality?

Abstracting Computation Patterns

- What is the common functionality?

```
inc :: [Int] -> [Int]
inc [] = []
inc (n:ns) = n+1 : inc ns
```

```
sqr :: [Int] -> [Int]
sqr [] = []
sqr (n:ns) = n^2 : sqr ns
```

- Both are instances of the map pattern:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

```
inc xs = map (+1) xs
sqr xs = map (^2) xs
```

ABSTRACTING COMPUTATION PATTERNS: EVALUATORS

A Super-Simple Language

- A super-simple expression language:

```
data Expr0 =  
    Val0 Int          -- integer values  
  | Add0 Expr0 Expr0 -- addition
```

A Super-Simple Language

- A super-simple expression language:

```
data Expr0 =  
    Val0 Int          -- integer values  
  | Add0 Expr0 Expr0 -- addition
```

- An evaluator:

```
eval0 :: Expr0 -> Int
```

```
eval1 (Val0 n) = n
```

```
eval1 (Add0 e1 e2) = eval0 e1 + eval0 e2
```

A Simple Language

- A simple expression language:

```
data Expr1=  
    Val1 Int          -- integer values  
  | Add1 Expr1 Expr1 -- addition  
  | Div1 Expr1 Expr1 -- division
```

A Simple Language + Evaluator

- A simple expression language:

```
data Expr1=  
    Val1 Int          -- integer values  
  | Add1 Expr1 Expr1 -- addition  
  | Div1 Expr1 Expr1 -- division
```

- An evaluator:

```
eval1 :: Expr1 -> Int
```

```
eval1 (Val1 n) = n
```

```
eval1 (Add e1 e2) = eval1 e1 + eval e2
```

A Simple Language + Evaluator

- A simple expression language:

```
data Expr1 =  
    Val1 Int          -- integer values  
  | Add1 Expr1 Expr1 -- addition  
  | Div1 Expr1 Expr1 -- division
```

- An evaluator:

```
eval1 :: Expr1 -> Int
```

```
eval1 (Val1 n) = n
```

```
eval1 (Add e1 e2) = eval1 e1 + eval e2
```

```
eval1 (Div1 e1 e2) = eval1 e1 `div` eval1 e2
```

raises exception on
divide-by-zero

why did all my languages last
week include +, -, * but not divide?
because I didn't want to have to
worry about managing the errors
that come from divide-by-zero!

A "Safe" Evaluator for Division

- To avoid raising exceptions (and terminating the computation), let's redefine the evaluator so it uses the Maybe type to represent errors explicitly

```
data Maybe a = Nothing | Just a
```

```
safediv :: Int -> Int -> Maybe Int
safediv n m = if m == 0 then Nothing
              else Just (n `div` m)
```

A "Safe" Evaluator for Division

```
data Maybe a = Nothing | Just a
```

```
safediv :: Int -> Int -> Maybe Int
```

```
safediv n m = if m == 0 then Nothing  
               else Just (n `div` m)
```

```
eval2 :: Expr1 -> Maybe Int
```

compare with eval1
which had type
Expr1 -> Int

only an Int result

A "Safe" Evaluator for Division

```
data Maybe a = Nothing | Just a
```

```
safediv :: Int -> Int -> Maybe Int
safediv n m = if m == 0 then Nothing
              else Just (n `div` m)
```

```
eval2 :: Expr1 -> Maybe Int
```

```
eval2 (Val1 n) = Just n
```

A "Safe" Evaluator for Division

```
data Maybe a = Nothing | Just a
```

```
safediv :: Int -> Int -> Maybe Int
```

```
safediv n m = if m == 0 then Nothing  
               else Just (n `div` m)
```

```
eval2 :: Expr1 -> Maybe Int
```

```
eval2 (Val1 n) = Just n
```

```
eval2 (Add1 e1 e2) =  
  case eval2 e1 of  
    Nothing -> Nothing  
    Just x -> case eval2 e2 of  
      Nothing -> Nothing  
      Just y -> Just (x + y)
```

the "interesting"
part of the computation

the rest is just
"plumbing" -- extracting
the interesting bits from
the Maybe data structure
via pattern matching

A "Safe" Evaluator for Division

```
data Maybe a = Nothing | Just a
```

```
safediv :: Int -> Int -> Maybe Int
```

```
safediv n m = if m == 0 then Nothing  
               else Just (n `div` m)
```

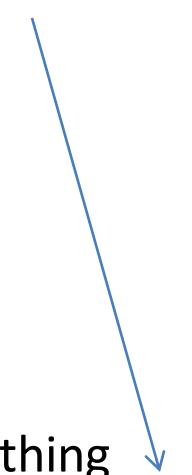
```
eval2 :: Expr1 -> Maybe Int
```

```
eval2 (Val1 n) = Just n
```

```
eval2 (Add1 e1 e2) =  
  case eval2 e1 of  
    Nothing -> Nothing  
    Just x -> case eval2 e2 of  
      Nothing -> Nothing  
      Just y -> Just (e1 + e2)
```

another "interesting"
part of the computation

```
eval2 (Div1 x y) =  
  case eval2 x of  
    Nothing -> Nothing  
    Just xn -> case eval2 y of  
      Nothing -> Nothing  
      Just yn -> safediv xn yn
```



A Simple Language with Printing

- Let's consider a language with printing:

```
data Expr3=
    Val3 Int          -- integer values
  | Add3 Expr3 Expr3 -- addition
  | PrintThen String Expr3 -- print String then return Expr
```

A Simple Language with Printing

- Let's consider a language with printing:

```
data Expr3 =  
    Val3 Int          -- integer values  
  | Add3 Expr3 Expr3 -- addition  
  | PrintThen String Expr3 -- print String then return Expr
```

eval3 :: Expr3 -> (String, Int)

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

compare with
previous types:
• Int
• Maybe Int

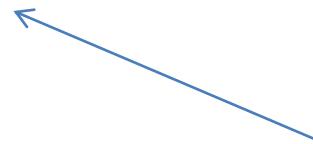
A Simple Language with Printing

- Let's consider a language with printing:

```
data Expr3 =  
    Val3 Int          -- integer values  
  | Add3 Expr3 Expr3 -- addition  
  | PrintThen String Expr3 -- print String then return Expr
```

eval3 :: Expr3 -> (String, Int)

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



Evaluation of a value prints nothing; it just returns the value

A Simple Language with Printing

- Let's consider a language with printing:

```
data Expr3 =  
    Val3 Int          -- integer values  
  | Add3 Expr3 Expr3 -- addition  
  | PrintThen String Expr3 -- print String then return Expr
```

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)

the heart of the computation
is doing the addition

more plumbing: extracting
the strings from evaluating
subexpressions and putting
them together

A Simple Language with Printing

- Let's consider a language with printing:

```
data Expr3=  
    Val3 Int          -- integer values  
  | Add3 Expr3 Expr3 -- addition  
  | PrintThen String Expr3 -- print String then return Expr
```

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

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

```
eval3 (Add 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)
```

A Simple Language with Printing

- Let's consider a language with printing:

```
data Expr3 =  
    Val3 Int          -- integer values  
  | Add3 Expr3 Expr3 -- addition  
  | PrintThen String Expr3 -- print String then return Expr
```

```
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)
```

more plumbing

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

A Simple Language with Storage

- Let's consider a language with mutable storage:

```
data Expr4 =  
    Val4 Int          -- integer values  
  | Add4 Expr4 Expr4 -- addition  
  | StoreThen Expr4 Expr4 -- Store e1 e2 stores e1 and returns e2  
  | Read            -- Read returns whichever integer has been  
                      stored last
```

A Simple Language with Storage

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

```
type State = Int
```

```
type Result a = State -> (State, a)
```

```
eval4 :: Expr4 -> Result Int
```

compare with
previous types:

- Int
- Maybe Int
- (String, Int)

an evaluator for
a language with storage
can be implemented
as a function that
takes an initial storage
state and returns a storage
state and a value

here, our values are
integers as before

A Simple Language with Storage

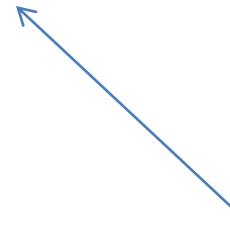
```
data Expr4 = Val4 | Add4 Expr4 Expr4 | StoreThen Expr4 Expr4 | Read
```

```
type State = Int
```

```
type Result a = State -> (State, a)
```

```
eval4 :: Expr4 -> Result a
```

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



no state change

A Simple Language with Storage

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

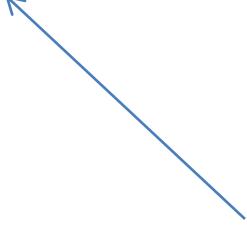
```
type State = Int
```

```
type Result a = State -> (State, a)
```

```
eval4 :: Expr4 -> Result a
```

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

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



read returns whatever
is in the current state

A Simple Language with Storage

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

```
type State = Int
```

```
type Result a = State -> (State, a)
```

```
eval4 :: Expr4 -> Result a
```

```
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)
```

oh the plumbing!

core computation:
addition

A Simple Language with Storage

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

```
type State = Int
```

```
type Result a = State -> (State, a)
```

```
eval4 :: Expr4 -> Result a
```

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

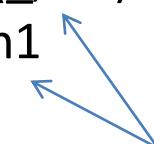
```
eval4(Read) = \s -> (s, s)
```

```
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
```



ignore the state
produced by $f1$;
use expression value

4 Examples

Language	Result type (<code>a == Int</code>)	Plumbing
Addition	<code>a</code>	None (identity function)
Safe Division (Languages with Errors)	<code>Maybe a</code>	pattern matching for Just and Nothing
Printing	<code>(String, a)</code>	pattern matching for pairs and String concatenation
Storage (Languages with State)	<code>State -> (State, a)</code>	pattern matching, function composition

MONADS

Monads

- Monads are abstractions that can help you write evaluators
 - and Haskell programs you may not consider "evaluators" will often use monads as well ... as we will see
- When using monads, you need to worry about 3 things:
 - what is the type of the result?
 - how to create the "null" evaluator that does nothing but return a value?
 - how to define the "plumbing" that allows you to compose evaluation of two subexpressions?
- From now on, we will call the evaluation of an expression, a "computation"
 - monads can be used, generally speaking, to structure computations and **compose** (ie: put together) computations

The Details

- Naturally, Haskell has a Monad type class!

m is the type you need to define

```
class Monad m where  
    return :: a -> m a
```

-- the null computation

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

given a computation/evaluation x that returns a value v with type a and a function f that generates a computation m b from v, compose x and f to create a new computation/evaluation

The Details

- Naturally, Haskell has a Monad type class!

```
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 uses the Maybe type to keep track of whether an error has happened.

instance Monad Maybe where

return v = Just v -- an error-free computation that
 -- does nothing but return a value v

Just v >>= f = f v -- compose an error-free computation with f

Nothing >>= f = Nothing -- compose an error-full computation with f

The Safe Division Evaluator Revisited

instance Monad Maybe where

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

```
(Just v) >>= f = f v      -- compose an error-free computation with f
```

```
Nothing >>= f = Nothing  -- compose an error-full computation with f
```

```
data Expr1 =
```

```
    Val1 Int           -- integer values
```

```
  | Add1 Expr1 Expr1 -- addition
```

```
  | Div1 Expr1 Expr1 -- division
```

```
eval :: Expr1 -> Maybe Int
```

The Safe Division Evaluator Revisited

instance Monad Maybe where

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

```
Just v >>= f = f v      -- compose an error-free computation with f
```

```
Nothing >>= f = Nothing -- compose an error-full computation with f
```

```
data Expr1 =
```

```
  Val1 Int           -- integer values  
  | Add1 Expr1 Expr1 -- addition  
  | Div1 Expr1 Expr1 -- division
```

```
eval :: Expr1 -> Maybe Int
```

```
eval (Val1 v) = return v
```

The Safe Division Evaluator Revisited

instance Monad Maybe where

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

```
Just v >>= f = f v      -- compose an error-free computation with f
```

```
Nothing >>= f = Nothing -- compose an error-full computation with f
```

```
data Expr1 =
```

```
  Val1 Int           -- integer values  
  | Add1 Expr1 Expr1 -- addition  
  | Div1 Expr1 Expr1 -- division
```

```
eval :: Expr1 -> Maybe Int
```

```
eval (Val1 v) = return v
```

```
eval (Add1 e1 e2) =
```

```
  eval e1 >>= (\x ->
```

```
  eval e2 >>= (\y ->
```

```
    return (x + y)))
```

The Safe Division Evaluator Revisited

instance Monad Maybe where

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

```
Just v >>= f = f v      -- compose an error-free computation with f
```

```
Nothing >>= f = Nothing -- compose an error-full computation with f
```

data Expr1 =

```
  Val1 Int           -- integer values  
  | Add1 Expr1 Expr1 -- addition  
  | Div1 Expr1 Expr1 -- division
```

eval :: Expr1 -> Maybe Int

```
eval (Val1 v) = return v
```

eval (Add1 e1 e2) =

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

eval (Div1 e1 e2) =

```
  eval e1 >>= (\x ->  
  eval e2 >>= (\y ->  
  if y == 0 then Nothing  
  else return (x `div` y)))
```

The Safe Division Evaluator Revisited

eval (Val1 v) = return v

eval (Add1 e1 e2) =
eval e1 >>= (\x ->
eval e2 >>= (\y ->
return (x + y)))

eval (Div1 e1 e2) =
eval e1 >>= (\x ->
eval e2 >>= (\y ->
if y == 0 then Nothing
else return (x `div` y)))

safediv n m = if m == 0 then Nothing
else Just (n `div` m)

eval2 (Val1 v) = Just v

eval2 (Add1 e1 e2) =
case eval2 e1 of
Nothing -> Nothing
Just x -> case eval2 e2 of
Nothing -> Nothing
Just y -> Just (e1 + e2)

eval2 (Div1 e1 e2) =
case eval2 e1 of
Nothing -> Nothing
Just x -> case eval2 e2 of
Nothing -> Nothing
Just y -> safediv x y

Safe Division: Not Satisfied!

```
eval (Val1 v) = return v
```

```
eval (Add1 e1 e2) =  
  eval e1 >>= (\x ->  
  eval e2 >>= (\y ->  
    return (x + y)))
```

```
eval (Div1 e1 e2) =  
  eval e1 >>= (\x ->  
  eval e2 >>= (\y ->  
    if y == 0 then Nothing  
    else return (x `div` y)))
```

Still not satisfied! Ugly. 9 characters plus some spaces to implement the concept of "composition."

A Surprise

- Haskell's do notation is just special built-in syntax for using monads!

eval :: Expr1 -> Maybe Int

eval (Val1 v) = return v

eval (Add1 e1 e2) = do
x <- eval e1
y <- eval e2
return (x + y)

eval (Div1 e1 e2) = do
x <- eval e1
y <- eval e2
if y == 0 then Nothing
else return (x `div` y)

eval :: Expr1 -> Maybe Int

eval (Val1 v) = return v

eval (Add1 e1 e2) =
eval e1 >>= (\x ->
eval e2 >>= (\y ->
return (x + y)))

eval (Div1 e1 e2) =
eval e1 >>= (\x ->
eval e2 >>= (\y ->
if y == 0 then Nothing
else return (x `div` y)))

In General

do
 x <- e
 computation \longrightarrow $e >>= (\lambda x. \text{do computation})$

do
 e
 computation \longrightarrow $e >> \text{do computation}$

do
 let x = e
 computation \longrightarrow $\text{let } x = e \text{ in}$
 do computation

- A example:

do
 x <- eval e1
 y <- eval e2
 return (x + y) \longrightarrow eval e1 >>= (\lambda x.
 eval e2 >>= (\lambda y.
 return (x + y)))

SUMMARY

Summary

- We can simply implementation of evaluators using monads
- There are monads for handling errors, printing, storage and more
- Defining a monad involves three parts:
 - What is the type of the monad?
 - How do we evaluate a pure value and do nothing else?
 - ie: how do we implement “return”
 - resembles "pure" from an applicative functor
 - How do we compose evaluation of two subexpressions
 - ie: how do we implement “bind”: $e >= f$