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: exp -> exp
  - state: (state, exp) -> (state, exp)
  - printing: (string, exp) -> (string, exp)
  - state and printing: (string, state, exp) -> (string, state, 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:

\[
\text{inc} :: [\text{Int}] \rightarrow [\text{Int}] \\
\text{inc} [] = [] \\
\text{inc} (n:\text{ns}) = n+1 : \text{inc} \text{ns}
\]

\[
\text{sqr} :: [\text{Int}] \rightarrow [\text{Int}] \\
\text{sqr} [] = [] \\
\text{sqr} (n:\text{ns}) = n^2 : \text{sqr} \text{ns}
\]

• What is the common functionality? How can we rewrite them to abstract out the commonality?
Abstracting Computation Patterns

• What is the common functionality?

\[
\text{inc} :: [\text{Int}] \rightarrow [\text{Int}]
\]
\[
\text{inc} [:] = []
\]
\[
\text{inc} (n:ns) = n+1 : \text{inc} ns
\]

\[
\text{sqr} :: [\text{Int}] \rightarrow [\text{Int}]
\]
\[
\text{sqr} [:] = []
\]
\[
\text{sqr} (n:ns) = n^2 : \text{sqr} ns
\]

• Both are instances of the map pattern:

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]
\[
\text{map} f [:] = []
\]
\[
\text{map} f (x:xs) = f x : \text{map} f xs
\]
\[
\text{inc} xs = \text{map} (+1) xs
\]
\[
\text{sqr} xs = \text{map} (^2) xs
\]
ABSTRACTING COMPUTATION PATTERNS: EVALUATORS
A Super-Simple Language

• A super-simple expression language:

```haskell
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:

```haskell
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:

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

• An evaluator:

```haskell
eval1 :: Expr1 -> Int

eval1 (Val1 n) = n
eval1 (Add e1 e2) = eval1 e1 + eval e2
eval1 (Div1 e1 e2) = eval1 e1 `div` eval1 e2
```

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!

raises exception on divide-by-zero
To avoid raising exceptions (and terminating the computation), let's redefine the evaluator so it uses the Maybe type to represent errors explicitly:

```haskell
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
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)

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

```haskell
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
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.
Let's consider a language with printing:

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

```haskell
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.
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)
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)

eval3 (PrintThen s e) =
    let (s', n) = eval3 e in (s ++ s', n)
Let's consider a language with mutable storage:

```haskell
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

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

compare with previous types:
- Int
- Maybe Int
- (String, Int)

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)
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) = \(s \rightarrow (s, x)\)

eval4(Read) = \(s \rightarrow (s, s)\)

eval4(Add e1 e2) =
  let f1 = eval4 e1
      f2 = eval4 e2 in
  \(s0 \rightarrow \text{let } (s1, n1) = f1 s0 \text{ in} \)
  \((s2, n2) = f2 s1 \text{ in} \)
  \((s2, n1 + n2)\)

core computation: addition

oh the plumbing!
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) = \(\lambda s \rightarrow (s, x)\)
eval4(Read) = \(\lambda s \rightarrow (s, s)\)

eval4(Add e1 e2) = 
  let f1 = eval4 e1
     f2 = eval4 e2 in
  \(\lambda s0 \rightarrow \) 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
  \(\lambda s0 \rightarrow \) let \(\_\), n1) = f1 s0 in
  f2 n1

ignore the state produced by f1; use expression value value
## 4 Examples

<table>
<thead>
<tr>
<th>Language</th>
<th>Result type (a == Int)</th>
<th>Plumbing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>a</td>
<td>None (identity function)</td>
</tr>
<tr>
<td>Safe Division (Languages with Errors)</td>
<td>Maybe a</td>
<td>pattern matching for Just and Nothing</td>
</tr>
<tr>
<td>Printing</td>
<td>(String, a)</td>
<td>pattern matching for pairs and String concatenation</td>
</tr>
<tr>
<td>Storage (Languages with State)</td>
<td>State -&gt; (State, a)</td>
<td>pattern matching, function composition</td>
</tr>
</tbody>
</table>
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
Naturally, Haskell has a Monad type class!

```haskell
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!

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

\[
eval \ (\text{Add1} \ e_1 \ e_2) = \quad \text{eval} \ e_1 \毗\ (\lambda x \rightarrow \text{eval} \ e_2 \毗\ (\lambda y \rightarrow \text{return} \ (x + y)))
\]

\[
eval \ (\text{Div1} \ e_1 \ e_2) = \quad \text{eval} \ e_1 \毗\ (\lambda x \rightarrow \text{eval} \ e_2 \毗\ (\lambda y \rightarrow \text{if} \ y == 0 \text{then} \text{Nothing} \text{else} \text{return} \ (x \ `\text{div}` \ y)))
\]

\[
safediv \ n \ m = \text{if} \ m == 0 \text{then Nothing} \text{else} \text{Just} \ (n \ `\text{div}` \ m)
\]

\[
eval2 \ (\text{Val1} \ v) = \text{Just} \ v
\]

\[
eval2 \ (\text{Add1} \ e_1 \ e_2) = \quad \text{eval2} \ e_1 \毗\ \text{case} \ eval2 \ e_2 \text{of}
\]
\[
\quad \text{Nothing} \rightarrow \text{Nothing}
\]
\[
\quad \text{Just} \ x \rightarrow \text{case} \ eval2 \ e_2 \text{of}
\]
\[
\quad \text{Nothing} \rightarrow \text{Nothing}
\]
\[
\quad \text{Just} \ y \rightarrow \text{Just} \ (\text{e1} + \text{e2})
\]

\[
eval2 \ (\text{Div1} \ e_1 \ e_2) = \quad \text{eval2} \ e_1 \毗\ \text{case} \ eval2 \ e_2 \text{of}
\]
\[
\quad \text{Nothing} \rightarrow \text{Nothing}
\]
\[
\quad \text{Just} \ x \rightarrow \text{case} \ eval2 \ e_2 \text{of}
\]
\[
\quad \text{Nothing} \rightarrow \text{Nothing}
\]
\[
\quad \text{Just} \ y \rightarrow \text{safediv} \ x \ y
\]
Safe Division: Not Satisfied!

\[
\begin{align*}
\text{eval (Val1 } \, v) &= \text{return } v \\
\text{eval (Add1 } \, e_1 \, e_2) &= \\
&= \text{ eval } e_1 \gg= (\lambda x \to \\
&\quad \text{ eval } e_2 \gg= (\lambda y \to \\
&\quad \quad \text{return } (x + y))) \\
\text{eval (Div1 } \, e_1 \, e_2) &= \\
&= \text{ eval } e_1 \gg= (\lambda x \to \\
&\quad \text{ eval } e_2 \gg= (\lambda y \to \\
&\quad \quad \text{if } y == 0 \text{ then Nothing} \\
&\quad \quad \text{else return } (x \, \text{`div` } y)))
\end{align*}
\]

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)
```
### In General

<table>
<thead>
<tr>
<th>Expression</th>
<th>Transformation</th>
</tr>
</thead>
</table>
| \[
\begin{align*}
\text{do} & \\
\quad x & \leftarrow e \\
\quad \text{computation} & \\
\end{align*}
\] | \[
\begin{align*}
\text{do} & \\
\quad e & \text{ >>= (} \lambda x. \text{do computation} \text{)} \\
\end{align*}
\] |
| \[
\begin{align*}
\text{do} & \\
\quad e & \\
\quad \text{computation} & \\
\end{align*}
\] | \[
\begin{align*}
\text{do} & \\
\quad e & \text{ >>= do computation} \\
\end{align*}
\] |
| \[
\begin{align*}
\text{do} & \\
\quad \text{let } x = e & \text{ in} \\
\quad \text{computation} & \\
\end{align*}
\] | \[
\begin{align*}
\text{do} & \\
\quad \text{let } x = e & \text{ in} \\
\quad \text{do computation} & \\
\end{align*}
\] |

- **A example:**

\[
\begin{align*}
\text{do} & \\
\quad x & \leftarrow \text{eval } e1 \\
\quad y & \leftarrow \text{eval } e2 \\
\quad \text{return } (x + y) & \\
\end{align*}
\] | \[
\begin{align*}
\text{do} & \\
\quad \text{eval } e1 & \text{ >>= (} \lambda x. \text{eval } e1 & \text{ >>= (} \lambda y. \text{return } (x + y)))) \end{align*}
\]
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