Haskell Monads

CSC 131

Kim Bruce

Monads

The ontological essence of a **monad** is its ***irreducible simplicity***. Unlike atoms, monads possess no material or spatial character. They also differ from atoms by their complete mutual independence, so that interactions among monads are only apparent. Instead, by virtue of the principle of pre-established harmony, **each monad follows a preprogrammed set of "instructions" peculiar to itself, so that a monad "knows" what to do at each moment**.

-*wikipedia*

Monads

In category theory, a branch of mathematics, a *monad*, or triple is an (endo-)functor, together with two natural transformations. Monads are used in the theory of pairs of adjoint functors, and they generalize closure operators on partially ordered sets to arbitrary categories.

-*wikipedia*

Defining Monads

part of Standard Prelude ←

- class Monad m where
  (>>=) :: m a → (a → m b) → m b
  return :: a → m a

  >>= allows a kind of composition of wrapped values or computations -- called *bind*

  - return wraps an unwrapped value.

Maybe Monad

- ```
  instance Monad Maybe where
        (>>=) Nothing f = Nothing
        (>>=) (Just x) f = f x
        return x = Just x
  ```

- >>= preserves “Nothing”,
- >>= unwraps argument to compute w/ a Just’ed value
- Second arg of >>= is function applied to unwrapped value
- *Abbreviate* `compu >>= \x → exp` *as*

  ```
  do x <- compu
        exp
  ```
  ```

Tuesday Example

- Expression
  - ```
    getPFN name rooms phones =
        do rm <- getDormFor name rooms
           num <- getPhoneForRoom rm phones
           return num
    ```
  - *abbreviates*
  - ```
    getPFN name rooms phones =
        getDormFor name rooms >>=
           (\rm -> getPhoneForRoom rm phones)
    ```

Monads

- Provide operations to compose wrapped values
- Operations obey laws:
  - return x >>= f == f x *left identity*
  - c >>= return == c *right identity*
  - c >>= (\x -> f x >>= g) == (c >>= f) >>= g *associativity*

In "do" notation

- Left identity:

```
do { x' <- return x;
       f x'                    ≡      do { f x }
     }
```

- Right identity:

```
do { x <- m;
       return x                ≡      do { m }
   }
```

- Associativity:

```
do { y <- do { x <- m;                    do { x <- m;
                 f x                           do { y <- f x;
               }                   ≡                  g y
       g y                                         }
}                                         }
```

Application of Laws

- Program:

```
skip_and_get = do
        unused <- getLine
        line <- getLine
        return line
```

- is equivalent to:

```
skip_and_get = do
        unused <- getLine
        getLine
```

by right identity

See http://www.haskell.org/haskellwiki/Monad_laws for more info

Other Monad Examples

- Error handling $M(a) = a \cup \{error\}$
  - Add a special "error value" to a type
  - Define bind operator ">>=" to propagate error
- Information-flow tracking $M(a) = a \times Labels$
  - Add information flow label to each value
  - Define bind to check and propagate labels
- State $M(a) = a \times States$
  - Computation produces value and new state
  - Define bind to make output state of first go to input state of second

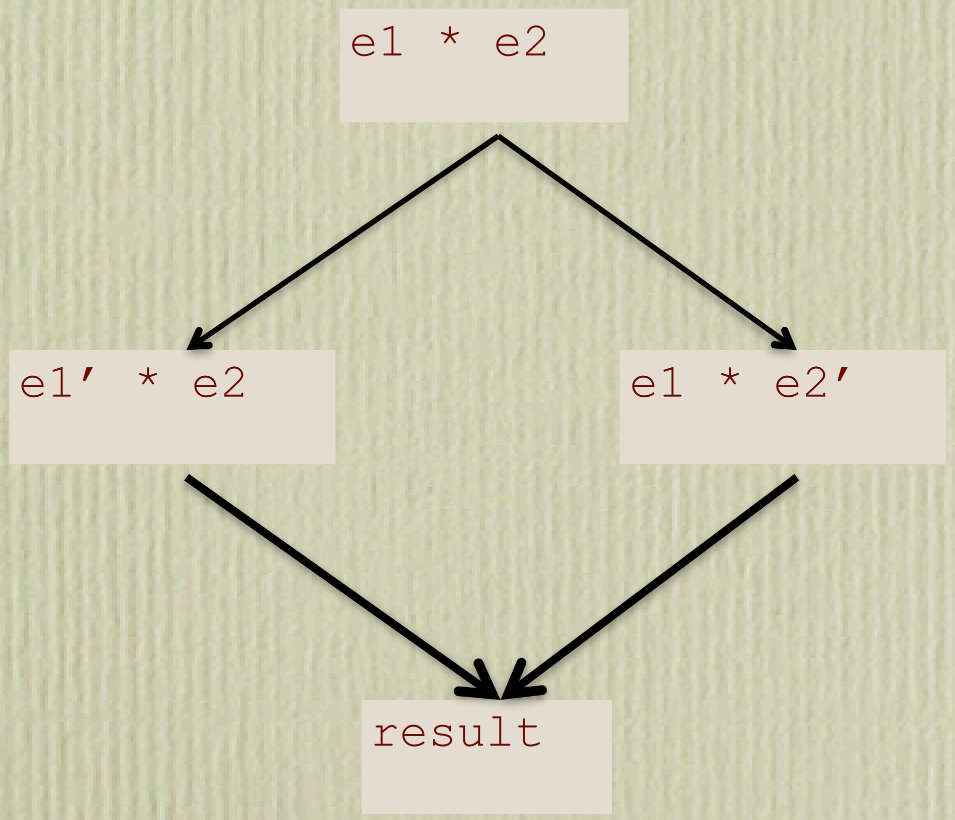
Big Idea

- Write code as though computing on a, but actually run it on M a.
  - That's what we did with Maybe monad!
  - Can think of monad as representing a suspended or pending computation.
  - Difference between having a cake and having a recipe for a cake.

Beauty

- Functional programming is beautiful:
  - Concise and powerful abstractions
    - higher-order functions, algebraic data types, parametric polymorphism, principled overloading, ...
  - Close correspondence with mathematics
    - Semantics of a code function is the mathematical function
    - Equational reasoning: if x = y, then f x = f y
  - Independence of order-of-evaluation
    - Confluence, aka Church-Rosser

Confluence means ...



- The compiler can choose the best sequential or parallel evaluation order!

... and the Beast

- But to be useful as well as beautiful, a language must manage the "Awkward Squad":
  - Input/Output
  - Imperative update
  - Error recovery (eg, timeout, divide by zero, etc.)
  - Foreign-language interfaces
  - Concurrency control

The whole point of a running a program is to interact with the external environment and affect it

The Direct Approach

- Just add imperative constructs "the usual way"
  - I/O via "functions" with side effects:
    - putChar 'x' + putChar 'y'
  - Imperative operations via assignable reference cells:
    - z = ref 0; z := z + 1; ...
  - Error recovery via exceptions
  - Foreign language procedures mapped to "functions"
  - Concurrency via operating system threads
- Can work if language determines eval order

Examples: ML, OCAML, Scheme/Racket

What if Lazy?

- Order of evaluation deliberately undefined.
- Example:
  - ls = [putChar ‘x’, putChar ‘y’]
  - if only use (length ls), then nothing printed!!

Fundamental Question

- Can you add imperative features with changing the meaning of pure Haskell expressions?
  - Even though laziness and side-effects are incompatible!!

History

- Big embarrassment to lazy functional programming community
  - ML, Scheme/LISP/Racket didn't care about being purely functional
- Alternatives:
  - Streams ⟵ *Haskell 1.0 adopted, essentially lazy lists*
  - Continuations
    - pure functions passed to IO routines to process input
  - Pass state of world as parameter
    - Hard to make single-threaded

Monads to Rescue!

- Value of type (IO a) is an action
  - that may perform some input/output
  - and deliver result of type a

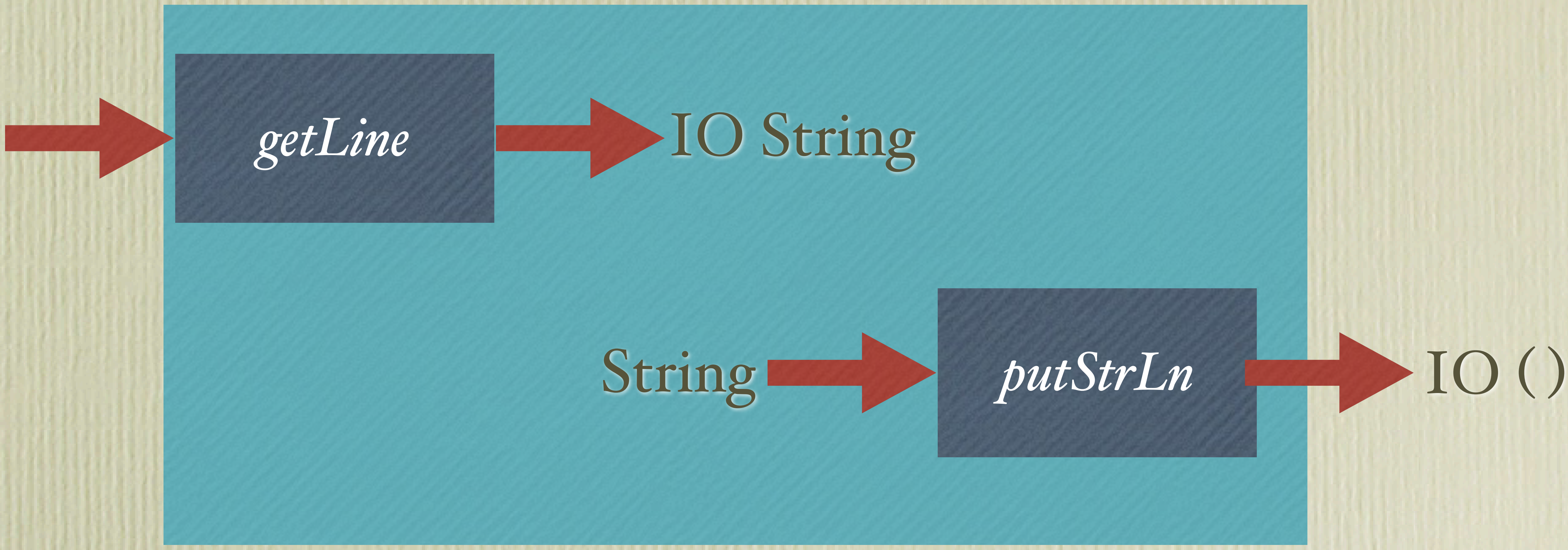
I/O

- main :: IO( )   -- "IO action"
- main = putStrLn "Hello World!"
- where putStrLn:: String → IO( )
- getLine :: IO String   -- "IO action" returning string

• Want echo = putStrLn getLine
  - Types don't match
  - Need >> = for IO monad!!
  - echo = do str <- getLine
    putStrLn str

See monad.hs

Connecting Actions



Glued together with >>=

Combining IO Operations

- If don't need result of first, even easier:
  - Remember that f >> g means run f, ignore result and run g.
  - putStrLn "Hello" >> putStrLn "world"

Executing program in Haskell

- Put main program in function **main** in myFile.hs
  - e.g. main = putStrLn “Hello world!”
  - main must have type IO( )
- Compile it: ghc myFile.hs
- Run it: ./myFile

Combining IO

- main = putStrLn "Please enter a number: " >> (readLn >>= (\n -> putStrLn (show (n+1))))

- main = do

  putStrLn "Please enter a number: "

  n <- readLn

  putStrLn (show (n+1))))

More IO

```
ask :: String -> String -> IO()
ask prompt ansPrefix =
                do putStr (prompt++" ")
                   response <- getLine
                   putStrLn (ansPrefix ++ " " ++ response)

getInteger :: IO Integer
getInteger = do putStr "Enter an integer: "
                line <- getLine
                return (read line)
              -- converts string to Integer then to IO Integer
```

IO & Ref Transparency

- Main program is IO action w/type IO( )
- Perform IO in IO actions & call pure functions from inside there
- Can never escape from IO! *Unlike Maybe.*
  - *No constructors for IO, so can't pattern match to escape!!!*
- IO impure in that successive calls of getLine return different values.

Using IO in Haskell

- Can build language at IO monad level:

```
ifIO :: IO Bool -> IO a -> IO a -> IO a
ifIO b tv fv = do { bv <- b;
                    if bv then tv else fv}

whileIO :: IO Bool -> IO( ) -> IO( )
whileIO b m = ifIO b
                (do {m; whileIO b m})
                (return( ))
```

Parsing Monad

Claim need parser to be monad to parse 4 78 19 3 44 3 1 7 5 2 3 2
into [[78 19 3 44] [1 7 5] [3 2]]

```
sequence :: Monad m => [m a] -> m [a]
sequence [] = return []
sequence (ma:mas) =
  ma >>= \a ->
  sequence mas >>= \as ->
  return (a:as)
```

Imagine m a = Maybe a. Then

```
sequence [(Just 5), (Just 3), (Just 6)]
      =>  Just [5, 3, 6]

sequence [(Just 5), (Just 3), Nothing, (Just 6)]
      =>  Nothing
```

How to write using do?

Parsing Tough Languages

```
sequence :: Monad m => [m a] -> m [a]
sequence [] = return []
sequence (ma:mas) = do
  a <- ma
  as <- sequence mas
  return (a:as)
```

Parsing Tough Languages

```haskell
replicateM :: Monad m => Int -> m a -> m [a]
replicateM n m = sequence (replicate n m)
```

replicateM 4 (Just 7) => Just [7,7,7,7]

Parsing Tough Languages

And now we are finally in a position to write the parser we wanted to write: it is simply

```
parseFile :: Parser [[Int]]
parseFile = many parseLine

parseLine :: Parser [Int]
parseLine = parseInt >>= \i -> replicateM i parseInt
```

Monad allows access to result of first parse to use in the next one.

Stateful computations

- Random number generator:
  - nextRand seed = (value, newSeed)
- Mirror stateful computation
  - Carry state around as parameter, perhaps as list of pairs of (locn,value)
  - Painful to have to thread state everywhere
- Perhaps monad can hide it

State Monad

- data **State** s a = State(s → (a,s))
  - Values are of form State f, where f provides one step of computation from state s, returning value-state pair (a,s')
  - define runState:: **State** s a → s → (a,s) by runState (State f) s = f s
  - provides a step of threaded computation returning an a.
  - evalState (State f) s = first (f s)
    - Just provides answer, ignoring new state
  - execState (State f) s = second (f s) -- gives just new state

State Monad

- data **State** s a = State(s → (a,s))
- Define >>=, return
  - return av = State(\s → (av,s))
    - value is always av, doesn't affect state
  - c >>= f = State(\s → let (a, s') = runState c s
    in runState (f a) s')
    - Given s, calculates state value pair (a,s') from running c on s. Then runs (f a) on new state s', providing value,state pair

State Monad

- Inside Monad State class have defs:
  - get :: **State** s s
  - get = State (\s → (s,s))
  - *returns current state as value*
  - put :: s → **State** s ( )
  - put s = State(_ → (( ),s))
  - *replace current state w/ new one*

Using randoms

```
type LCGState = Word32
lcg :: LCGState -> (Integer, LCGState)
lcg s0 = (output, s1)  where ... s0 ..

getRandom :: State LCGState Integer        <- State Monad instance!
getRandom = get >>= \s0 -> let (x,s1) = lcg s0
                           in put s1 >> return x

-- do something with randoms
addTwoRandoms = do a <- getRandom
                   b <- getRandom
                   return (a+b)
```

See Monad.hs for full code

Actually Computing ...

Start up with initial state

```
*Main> runState addTwoRandoms 109573
  (85805, 2066785931)
*Main> evalState addTwoRandoms 109573
   85805
```