HASKELL MONADS

Curt Clifton
Rose-Hulman Institute of Technology

SVN update. We'll be working in the *HaskellMonads* folder later.

MONADS

- Ooh, scary!
- Not really, just an extremely useful example of *generalization*
- Goal: recognize monads as a general solution to lots of problems

Lon Chaney, Jr. as The Wolf Man

GENERAL IDEA

- A computation with a certain type of result
  - e.g., Integer
- A certain type of structure in its result
  - e.g., Nothing, [], [2, -2]
- Need to pass the result of one of these computations to another

Monads let us build up these computations as static entities without necessarily running them

MONAD TYPECLASS

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

return takes a value of the inner type and wraps it in a computation

binding operator takes a computation

and feeds its value to a function

that makes a another computation
  ```

MAYBE AS A MONAD

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

- **instance Monad Maybe where**
  ```
  return x = Just x

  Nothing >>= f = Nothing
  Just x  >>= f = f x
  ```

return takes a value of the inner type and wraps it in a computation

binding operator takes a computation

and feeds its value to a function

that makes a another computation

INTEGER SQUARE ROOT

```
isqrt :: Integer -> Maybe Integer
isqrt x = isqrt' x (0,0)
  where isqrt' x (s,r)
        | s > x     = Nothing
        | s == x    = Just r
        | otherwise = isqrt' x (s + 2*r + 1, r+1)
```

Maybe computation

```
i4throot :: Integer -> Maybe Integer
i4throot x = case isqrt x of
           Nothing -> Nothing
           Just y  -> isqrt y
```

i4throot x = isqrt x >>= isqrt

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

Maybe computation made of Maybe computations

LIST AS A MONAD

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

- **instance Monad [] where**
  return x = [x]

  xs >>= f = concat (map f xs)

return takes a value of the inner type and wraps it in a computation

binding operator takes a computation

and feeds its value to a function

that makes a another computation

INTEGER SQUARE ROOT

```
isqrtL :: Integer -> [Integer]
isqrtL x = isqrt' x (0,0)
   where isqrt' x (s,r)
         | s > x     = []
         | s == x    = [r, -r]
         | otherwise = isqrt' x (s + 2*r + 1, r+1)
```

List computation

```
i4throotL :: Integer -> [Integer]
```

~~i4throotL x = case isqrtL x of~~
~~[] -> []~~
~~[y, _] -> isqrtL y~~

i4throotL x = isqrtL x >>= isqrtL

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

List computation made of
List computations

TRAPPED IN A MONAD

- How do we get results from computation?
  - Pattern match
  - Could use support functions if provided
- Without these the result is trapped!

http://www.flickr.com/photos/snugglepup/

THE STATE MONAD

PASSING STATE IMPLICITLY

Type of the state passed around

- `newtype State s a ...`

Type of the return value

- For any type `s`, `State s` is a monad
  - `State (Map Char Integer)` is a monad that passes around a `Map` implicitly
  - `State Integer` passes an `Integer` implicitly

Q2

PASSING STATE IMPLICITLY

- `newtype State s a ...`
- For any type `s`, `State s` is a monad
  - `State (Map Char Integer)` is a monad that passes around a `Map` implicitly
- Helper functions:
  - `get :: State s s`
  - `put :: s -> State s ()`

Takes implicit state and "shifts" it to result position

Replaces implicit state with a new state

Q3

THREE MORE STATE HELPERS

- `runState :: State s a -> s -> (a, s)`
- `evalState :: State s a -> s -> a`
- `execState :: State s a -> s -> s`

Takes a "State s" computation with result type a and an initial state, produces a pair of the result and the final state

Just yields the result

Just yields the final state

MONAD TYPECLASS EXTENDED

- class Monad m where

```
return :: a -> m a
(>>=)  :: m a -> (a -> m b) -> m b
(>>) :: m a -> m b -> m b
c >> d = c >>= \_ -> d
```

Convenience operator for chaining two computations together, ignoring result of the first

```
countDownBy n = get >>= \ctr -> put (ctr - n) >> return (ctr - n <= 0)
```

IMPLEMENTING AN INTERPRETER USING MONADS

THE LANGUAGE: EDDIE

- Syntax:
  - 42
  - 30 + 12
  - 6 * 7
  - 85 / 2
  - x
  - x := 2; y := x * 3; x := y * 7; x

Typical semantics, except integer division

imperative (non-functional) assignment

Q4

IMPLEMENTING EDDIE

- *EddieTypes.hs*:
  - Defines the data types
- *EddieParse.hs*:
  - Defines a parser for Eddie using the Parsec module
- *EddieEval.hs*:
  - Where we'll define an interpreter for Eddie