Monads

Dr. Mattox Beckman

Illinois Institute of Technology
Department of Computer Science

Objectives

- Describe the problem that monads attempt to solve.
- Know the three monad laws.
- Know the syntax for declaring monadic operations.
- Be able to give examples using the Maybe and List monads.

Motivation

- Monads are a way of defining computation.
- They are similar to continuations, but even more powerful.
- They are also related to the applicative functors from last time.
- Consider this program....

```haskell
inc1 a = a + 1
r1 = inc1 <$> Just 10 -- result: Just 11
r2 = inc1 <$> Nothing -- result: Nothing
```

But what if we have functions like this?

```haskell
inc2 a = Just (a+1)
recip a | a =/ 0     = Just (1/a)
        | otherwise = Nothing
```

How can we pass a Nothing to it? How can we use what we get from it?

Notice the pattern

- Applicatives take the values out of the parameters, run them through a function, and then repackage the result for us.
- The functions have no control: the applicative makes all the decisions.
- Monads let the functions themselves decide what should happen.

Introducing Monads

- A *monad* is a container type *m* along with two functions:
    - `return :: a -> m a`
    - `bind :: m a -> (a -> m b) -> m b`
    - In Haskell, `bind` is written as `>>=`
- These functions must obey three laws:

  Left identity `return a >>= f` is the same as `f a`

  Right identity `m >>= return` is the same as `m`

  Associativity `(m >>= f) >>= g` is the same as `m >>= (\x -> f x >>= g)`

Understanding Return

- The `return` keyword takes an element and puts it into a monad.
- This is a one way trip!
- Very much like `pure` in Applicative.

```
return a = Just a
```


Understanding Bind

- All the magic happens in bind.
- `bind :: m a -> (a -> m b) -> m b`
    - The first argument is a monad.
    - The second argument takes a monad, unpacks it, and repackages it with the help of the function argument.
        - Exactly *how* it does that is the magic part.

Bind for Maybe

```
Nothing  >>= f  = Nothing
(Just a) >>= f = f a
   -- Remember that f returns a monad
```


A calculator, with monads

```
minc x = x >>= (\xx -> return (xx + 1))
madd a b = a >>= (\aa ->
           b >>= (\bb -> return (aa+bb)))
-- but wait!!!
```

- Okay, the above code works, but here's a better way.
- First define functions `lift` to convert a function to monadic form for us!

These are part of Control.Monad

```
liftM f a = a >>= (\aa -> return (f aa))
liftM2 f a b = a >>= (\aa ->
                b >>= (\bb -> return (f aa bb)))
```

Continued

Lifting

```haskell
minc = liftM inc
madd = liftM2 add
msub = liftM2 sub
mdiv = a >>= (\aa ->
           b >>= (\bb ->
             if bb == 0 then fail "/0"
                        else return (dv aa bb)))
```

- `fail` is another useful monadic function.
- Here it's defined as `Nothing`.

A monad definition

- Here is the complete monad definition for Maybe

Maybe Monad

```haskell
instance Monad Maybe where
  return = Just

  (>>=) Nothing f  = Nothing
  (>>=) (Just a) f = f a

  fail s = Nothing
```


A list monad

- Lists can be monads too. The trick is deciding what bind should do.

List Monad

```haskell
instance Monad [] where
  return a = [a]

  (>>=) [] f  = []
  (>>=) xs f  = concatMap f xs

  fail s = []
```

- Note that we do not have to change *anything* in our lifted calculator example!

More capability

- What is the square root of 4?

Adding nondeterminism

```haskell
msqrt a = a >>= (\aa ->
      let sa = sqrt aa
       in [-sa,sa])


msqrt [4] >>= minc  -- becomes [-1,3]
```

Do notation

- Haskell has a special builtin syntax for monads.

```
mdiv a b = do aa <- a
              bb <- b
              if bb == 0 then fail "/0"
                         else return (dv aa bb)
```

- If you only need applicative, it's better to use that than monads.
- Avoid do notation if you can.