Monads Part 2

November 4, 2019

Monads Part 2

November 4, 2019

Source

Monads Part 2

November 4, 2019

Source

do notation

- `do {                  x} = x`

do notation

- `do {              x} = x`
- `do {let y = a; x} = let y = a in do {x}`

do notation

- `do {             x} = x`
- `do {let y = a; x} = let y = a in do {x}`
- `do {a <- y;     x} = y >>=  \a -> do {x}`

do notation

- `do {            x} = x`
- `do {let y = a; x} = let y = a in do {x}`
- `do {a <- y;    x} = y >>=  \a -> do {x}`
- `do {y;         x} = y >>=  \_ -> do {x}`

do notation

- `do {            x} = x`
- `do {let y = a; x} = let y = a in do {x}`
- `do {a <- y;     x} = y >>=  \a -> do {x}`
- `do {y;         x} = y >>         do {x}`

do notation

- ```
  myAction = do
      a <- getLine
      b <- getLine
      print $ a ++ b
  ```
  ```

do notation

- ```
  myAction = do
      a <- getLine
      b <- getLine
      print $ a ++ b

          =

      getLine >>= \a -> do
          b <- getLine
          print $ a ++ b
  ```
  ```

do notation

- ```
  myAction = do
      a <- getLine
      b <- getLine
      print $ a ++ b

            =

      getLine >>= \a ->
      getLine >>= \b -> do
         print $ a ++ b
  ```
  ```

do notation

- ```
  myAction = do
      a <- getLine
      b <- getLine
      print $ a ++ b

            =

      getLine >>= \a ->
      getLine >>= \b ->
      print $ a ++ b
  ```
  ```

do notation

- Why `(>>=)` instead of `(=<<)`?

do notation

- Why (>>=) instead of (=<<)?
- ```
  dog = do
      print "the"
      print "dog"
      print "barked"
  ```
  ```

do notation

- Why `(>>=)` instead of `(=<<)`?
-
```
dog = do
    print "the"
    print "dog"
    print "barked"

    = print "the" >>= \_ ->
      print "dog" >>= \_ ->
      print "barked"
```

do notation

- Why `(>>=)` instead of `(=<<)`?
- `dog = do`

```
dog = do
      print "the"
      print "dog"
      print "barked"

    = print "the" >>= \_ ->
      print "dog" >>= \_ ->
      print "barked"

    = (\_ -> print "barked") =<<
      (\_ -> print "dog"   ) =<<
               print "the"
```

Lists are Monads

- ```
  instance Monad [] where
      return x = [x]
      xs >>= f = concat (map f xs)
      fail _ = []
  ```
  ```

Lists are Monads

- ```
  instance Monad [] where
      return x = [x]
      xs >>= f = concat (map f xs)
      fail _ = []
  ```
  - `join = concat`
  ```

Lists are Monads

- `instance Monad [] where`
  ```
  return x = [x]
  xs >>= f = concat (map f xs)
  fail _ = []
  ```
  - `join = concat`
    - `concat = foldr (++) []`

Lists are Monads

- ```
  instance Monad [] where
      return x = [x]
      xs >>= f = concat (map f xs)
      fail _ = []
  ```
  - `join = concat`
    - `concat = foldr (++) []`
    - ```
      (foldr (++) "")
        <$> sequenceA [getLine, getLine, getLine]
      ```

Lists are Monads

- ```
  instance Monad [] where
      return x = [x]
      xs >>= f = concat (map f xs)
      fail _ = []
  ```
  - `join = concat`
    - `concat = foldr (++) []`
    - ```
      concat
        <$> sequenceA [getLine, getLine, getLine]
      ```

Lists are Monads

- `[1,2,3,4] = (+) <$> [0,2] <*> [1,2]`

Lists are Monads

- `[1,2,3,4] = (+) <$> [0,2] <*> [1,2]`

```
          = [0,2] >>= \a ->
            [1,2] >>= \b ->
            return $ a + b
```

Lists are Monads

- `[1,2,3,4] = (+) <$> [0,2] <*> [1,2]`

```
= [0,2] >>= \a ->
  [1,2] >>= \b ->
  return $ a + b

= [0,2] >>= \a ->
  [1,2] >>= \b ->
  [a + b]
```

Lists are Monads

- ```
  [1,2,3,4] = (+) <$> [0,2] <*> [1,2]

            = [0,2] >>= \a ->
              [1,2] >>= \b ->
              return $ a + b

            = [0,2] >>= \a ->
              concat (map (\b -> [a + b]) [1,2])
  ```
  ```

Lists are Monads

- `[1,2,3,4] = (+) <$> [0,2] <*> [1,2]`

```
          = [0,2] >>= \a ->
            [1,2] >>= \b ->
            return $ a + b

          = [0,2] >>= \a ->
            concat [[a+1], [a+2]]
```

Lists are Monads

- ```
  [1,2,3,4] = (+) <$> [0,2] <*> [1,2]

            = [0,2] >>= \a ->
              [1,2] >>= \b ->
              return $ a + b

            = [0,2] >>= \a ->
              [a+1, a+2]
  ```
  ```

Lists are Monads

- `[1,2,3,4] = (+) <$> [0,2] <*> [1,2]`

```
          = [0,2] >>= \a ->
            [1,2] >>= \b ->
            return $ a + b

          = concat (map (\a -> [a+1, a+2]) [0,2])
```

Lists are Monads

- ```
  [1,2,3,4] = (+) <$> [0,2] <*> [1,2]

            = [0,2] >>= \a ->
              [1,2] >>= \b ->
              return $ a + b

            = concat [[0+1, 0+2], [2+1, 2+2]]
  ```
  ```

Lists are Monads

- `[1,2,3,4] = (+) <$> [0,2] <*> [1,2]`

```
            = [0,2] >>= \a ->
              [1,2] >>= \b ->
              return $ a + b

            = [1,2,3,4]
```

Lists are Monads

- ```
  [1,2,3,4] = [0,2] >>= \a ->
              [1,2] >>= \b ->
              return $ a + b
  ```
  ```

Lists are Monads

- ```
  [1,2,3,4] = [0,2] >>= \a ->
              [1,2] >>= \b ->
              return $ a + b

            =
              [0,2] >>= \a ->
              [1,2] >>= \b -> do
              return $ a + b
  ```
  ```

Lists are Monads

- ```
  [1,2,3,4] = [0,2] >>= \a ->
              [1,2] >>= \b ->
              return $ a + b

            =
              [0,2] >>= \a -> do
              b <- [1,2]
              return $ a + b
  ```
  ```

Lists are Monads

- ```
  [1,2,3,4] = [0,2] >>= \a ->
              [1,2] >>= \b ->
              return $ a + b

            = do
              a <- [0,2]
              b <- [1,2]
              return $ a + b
  ```
  ```

Lists are Monads

- ```
  [1,2,3,4] = [0,2] >>= \a ->
              [1,2] >>= \b ->
              return $ a + b

            = [a + b |
               a <- [0,2],
               b <- [1,2]]
  ```
  ```

Lists are Monads

- ```
  [1,2,3,4] = [0,2] >>= \a ->
              [1,2] >>= \b ->
              return $ a + b

            = [a + b |
               a <- [0,2],
               b <- [1,2]]
  ```
  - List comprehensions are syntactic sugar for monadic computations!
  ```

Monad Laws

- Definitions of `id` and `.`:
  - Identity: `id $ v = v`
  - Composition: `(.) u v $ w = u $ (v $ w)`

Monad Laws

- Definitions of `id` and `.`:
  - Identity: `id $ v = v`
  - Composition: `(.) u v $ w = u $ (v $ w)`
- Functor laws:
  - Identity: `id <$> v = v`
  - Composition: `(.) u v <$> w = u <$> (v <$> w)`

Monad Laws

- Definitions of `id` and `.`:
  - Identity: `id $ v = v`
  - Composition: `(.) u v $ w = u $ (v $ w)`
- Functor laws:
  - Identity: `id <$> v = v`
  - Composition: `(.) u v <$> w = u <$> (v <$> w)`
- Applicative laws:
  - Identity: `pure id <*> v = v`
  - Composition: `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`

Monad Laws

- Definitions of `id` and `.`:
  - Identity: `id $ v = v`
  - Composition: `(.) u v $ w = u $ (v $ w)`
- Functor laws:
  - Identity: `id <$> v = v`
  - Composition: `(.) u v <$> w = u <$> (v <$> w)`
- Applicative laws:
  - Identity: `pure id <*> v = v`
  - Composition: `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
- Monad laws:
  - Identity:
  - Composition:

Monad Laws

- Definitions of `id` and `.`:
  - Identity: `id $ v = v`
  - Composition: `(.) u v $ w = u $ (v $ w)`
- Functor laws:
  - Identity: `id <$> v = v`
  - Composition: `(.) u v <$> w = u <$> (v <$> w)`
- Applicative laws:
  - Identity: `pure id <*> v = v`
  - Composition: `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
- Monad laws:
  - Left Identity: `return w >>= v = v w`
  - Right Identity: `v >>= return = v`
  - Composition:

Monad Laws

- Definitions of `id` and `.`:
  - Identity: `id $ v = v`
  - Composition: `(.) u v $ w = u $ (v $ w)`
- Functor laws:
  - Identity: `id <$> v = v`
  - Composition: `(.) u v <$> w = u <$> (v <$> w)`
- Applicative laws:
  - Identity: `pure id <*> v = v`
  - Composition: `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
- Monad laws:
  - Left Identity: `return w >>= v = v w`
  - Right Identity: `v >>= return = v`
  - Composition: `(w >>= v) >>= u = w >>= (\x -> v x >>= u)`

Monad Laws

- Left Identity: `return w >>= v = v w`

Monad Laws

- Left Identity: `return w >>= v = v w`
  - Let `v :: a -> m b` and `w :: a`

Monad Laws

- Left Identity: `return w >>= v = v w`
  - Let `v :: a -> m b` and `w :: a`
  - Then `return w :: m a`

Monad Laws

- Left Identity: `return w >>= v = v w`
  - Let `v :: a -> m b` and `w :: a`
  - Then `return w :: m a`
  - `(>>=)` extracts `w` from `return w` and feeds it to `v`

Monad Laws

- Left Identity: `return w >>= v = v w`
  - Let `v :: a -> m b` and `w :: a`
  - Then `return w :: m a`
  - `(>>=)` extracts `w` from `return w` and feeds it to `v`
    - i.e. computes `v w`

Monad Laws

- Left Identity: `return w >>= v = v w`
  - Let `v :: a -> m b` and `w :: a`
  - Then `return w :: m a`
  - `(>>=)` extracts `w` from `return w` and feeds it to `v`
    - i.e. computes `v w`
- Right Identity: `v >>= return = v`

Monad Laws

- Left Identity: `return w >>= v = v w`
  - Let `v :: a -> m b` and `w :: a`
  - Then `return w :: m a`
  - `(>>=)` extracts `w` from `return w` and feeds it to `v`
    - i.e. computes `v w`
- Right Identity: `v >>= return = v`
  - Let `v :: m a` and `return :: a -> m a`

Monad Laws

- Left Identity: `return w >>= v = v w`
  - Let `v :: a -> m b` and `w :: a`
  - Then `return w :: m a`
  - `(>>=)` extracts `w` from `return w` and feeds it to `v`
    - i.e. computes `v w`
- Right Identity: `v >>= return = v`
  - Let `v :: m a` and `return :: a -> m a`
  - `(>>=)` extracts a value of type `a` from `v` and feeds it to `return`

Monad Laws

- Left Identity: `return w >>= v = v w`
  - Let `v :: a -> m b` and `w :: a`
  - Then `return w :: m a`
  - `(>>=)` extracts `w` from `return w` and feeds it to `v`
    - i.e. computes `v w`
- Right Identity: `v >>= return = v`
  - Let `v :: m a` and `return :: a -> m a`
  - `(>>=)` extracts a value of type `a` from `v` and feeds it to `return`
    - i.e. puts the value of type `a` in a box

Monad Laws

- Left Identity: `return w >>= v = v w`
  - Let `v :: a -> m b` and `w :: a`
  - Then `return w :: m a`
  - `(>>=)` extracts `w` from `return w` and feeds it to `v`
    - i.e. computes `v w`
- Right Identity: `v >>= return = v`
  - Let `v :: m a` and `return :: a -> m a`
  - `(>>=)` extracts a value of type `a` from `v` and feeds it to `return`
    - i.e. puts the value of type `a` in a box
    - i.e. computes `v`

Monad Laws

- Left Identity: `v =<< return w = v w`
  - Let `v :: a -> m b` and `w :: a`
  - Then `return w :: m a`
  - `(>>=)` extracts `w` from `return w` and feeds it to `v`
    - i.e. computes `v w`
- Right Identity: `return =<< v = v`
  - Let `v :: m a` and `return :: a -> m a`
  - `(>>=)` extracts a value of type `a` from `v` and feeds it to `return`
    - i.e. puts the value of type `a` in a box
    - i.e. computes `v`

Monad Laws

- Composition: `(w >>= v) >>= u = w >>= (\x -> v x >>= u)`
  - Consider regular function composition:

Monad Laws

- Composition: `(w >>= v) >>= u = w >>= (\x -> v x >>= u)`
  - Consider regular function composition:
    - `(.)  u v $ w = u $ (v $ w)`

Monad Laws

- Composition: `(w >>= v) >>= u = w >>= (\x -> v x >>= u)`
  - Consider regular function composition:
    - `(u . v) w = u (v w)`

Monad Laws

- Composition: `(w >>= v) >>= u = w >>= (\x -> v x >>= u)`
  - Consider regular function composition:
    - `(u . v) w = u (v w)`
    - ```
      u :: b -> c
      v :: a -> b
      w :: a
      ```
      ```

Monad Laws

- Composition: `(w >>= v) >>= u = w >>= (\x -> v x >>= u)`
  - Consider regular function composition:
    - `(u . v) w = u (v w)`
    - ```
      u :: b -> c
      v :: a -> b
      w :: a
      ```
  - Now consider monadic function composition:
      ```

Monad Laws

- Composition: `(w >>= v) >>= u = w >>= (\x -> v x >>= u)`
  - Consider regular function composition:
    - `(u . v) w = u (v w)`
    - ```
      u :: b -> c
      v :: a -> b
      w :: a
      ```
  - Now consider monadic function composition:
    - `(u <=< v) x = v x >>= u`
      ```

Monad Laws

- Composition: `(w >>= v) >>= u = w >>= (\x -> v x >>= u)`
  - Consider regular function composition:
    - `(u . v) w = u (v w)`
    - `u :: b -> c`
      `v :: a -> b`
      `w :: a`
  - Now consider monadic function composition:
    - `(u <=< v) x = v x >>= u`
    - `u :: b -> m c`
      `v :: a -> m b`
      `x :: a`

Monad Laws

- Composition: `(w >>= v) >>= u = w >>= (\x -> v x >>= u)`
  - Consider regular function composition:
    - `(u . v) w = u (v w)`
    - ```
      u :: b -> c
      v :: a -> b
      w :: a
      ```
  - Now consider monadic function composition:
    - `(u <=< v) x = v x >>= u`
    - ```
      u :: b -> m c
      v :: a -> m b
      x :: a
      ```

    - Apply `x` to `v` to get a monadic value of type `m b`

Monad Laws

- Composition: `(w >>= v) >>= u = w >>= (\x -> v x >>= u)`
  - Consider regular function composition:
    - `(u . v) w = u (v w)`
    - ```
      u :: b -> c
      v :: a -> b
      w :: a
      ```
  - Now consider monadic function composition:
    - `(u <=< v) x = v x >>= u`
    - ```
      u :: b -> m c
      v :: a -> m b
      x :: a
      ```

    - Apply `x` to `v` to get a monadic value of type `m b`
    - Then extract a value of type `b` and feed it to `u` to get a monadic value of type `m c`

Monad Laws

- Composition: `(w >>= v) >>= u = w >>= (u <=< v)`
  - Consider regular function composition:
    - `(u . v) w = u (v w)`
    - ```
      u :: b -> c
      v :: a -> b
      w :: a
      ```
  - Now consider monadic function composition:
    - `(u <=< v) x = v x >>= u`
    - ```
      u :: b -> m c
      v :: a -> m b
      x :: a
      ```

    - Apply `x` to `v` to get a monadic value of type `m b`
    - Then extract a value of type `b` and feed it to `u` to get a monadic value of type `m c`

Monad Laws

- Composition: `(<=<) u v =<< w = u =<< (v =<< w)`
  - Consider regular function composition:
    - `(u . v) w = u (v w)`
    - ```
      u :: b -> c
      v :: a -> b
      w :: a
      ```
  - Now consider monadic function composition:
    - `(u <=< v) x = v x >>= u`
    - ```
      u :: b -> m c
      v :: a -> m b
      x :: a
      ```
    - Apply `x` to `v` to get a monadic value of type `m b`
    - Then extract a value of type `b` and feed it to `u` to get a monadic value of type `m c`

Monad Laws

- (Right) Identity:
  - `       id  $  v = v`
  - `       id <$> v = v`
  - `pure   id <*> v = v`
  - `return   =<< v = v`

Monad Laws

- (Right) Identity:
  - `       id  $  v = v`
  - `       id <$> v = v`
  - `pure   id <*> v = v`
  - `return    =<< v = v`
- Left Identity:
  - `v =<< return w = v w`

Monad Laws

- (Right) Identity:
  - `       id  $  v = v`
  - `       id <$> v = v`
  - `pure   id <*> v = v`
  - `return   =<< v = v`
- Left Identity:
  - `v =<< return w = v w`
- Composition:
  - `     (.)         u     v  $  w = u  $  (v  $  w)`
  - `     (.)         u     v <$> w = u <$> (v <$> w)`
  - `pure (.)    <*> u <*> v <*> w = u <*> (v <*> w)`
  - `     (<=<)       u     v =<< w = u =<< (v =<< w)`

Monads

- Other examples of monads:

Monads

- Other examples of monads:
  - `Maybe`

Monads

- Other examples of monads:
  - `Maybe`
  - Functions (`(->) r`)

Monads

- Functors are boxes
  - That implement maps that lift normal functions (of type `a -> b`) to functions over boxes (of type `F a -> F b`)

Monads

- Functors are boxes
  - That implement maps that lift normal functions (of type `a -> b`) to functions over boxes (of type `F a -> F b`)
- Applicative functors are boxes that support function application
  - If you have a normal function (`a -> b`), you can put it in a box (`F (a -> b)`), and apply it to a box (`F a`) to get another box (`F b`)

Monads

- Functors are boxes
  - That implement maps that lift normal functions (of type `a -> b`) to functions over boxes (of type `F a -> F b`)
- Applicative functors are boxes that support function application
  - If you have a normal function (`a -> b`), you can put it in a box (`F (a -> b)`), and apply it to a box (`F a`) to get another box (`F b`)
- Monads are boxes that support functions that create their own boxes

Monads

- Functors are boxes
  - That implement maps that lift normal functions (of type `a -> b`) to functions over boxes (of type `F a -> F b`)
- Applicative functors are boxes that support function application
  - If you have a normal function (`a -> b`), you can put it in a box (`F (a -> b)`), and apply it to a box (`F a`) to get another box (`F b`)
- Monads are boxes that support functions that create their own boxes
  - If you have a monadic function (`a -> F b`), you can apply it to a value (`a`) in a box (`F a`) to get another box (`F b`)

Monads

- Functors represent context
  - That implement maps that lift normal functions (of type `a -> b`) to functions over context (of type `F a -> F b`)
- Applicative functors represent contexts that support function application
  - If you have a normal function (`a -> b`), you can put it in a context (`F (a -> b)`), and apply it to a context (`F a`) to get another context (`F b`)
- Monads represent contexts that support functions that create their own contexts
  - If you have a monadic function (`a -> F b`), you can apply it to a value (`a`) in a context (`F a`) to get another context (`F b`)

Monads

- Functors represent context
  - That implement maps that lift normal functions (of type `a -> b`) to functions over context (of type `F a -> F b`)
- Applicative functors represent contexts that support function application
  - If you have a normal function (`a -> b`), you can put it in a context (`F (a -> b)`), and apply it to a context (`F a`) to get another context (`F b`)
- Monads represent contexts that can be joined together
  - If you have a context in another context (`F (F a)`), you can join the two contexts into one (`F a`)