BAHUG 101 - Lecture 7

3rd February 2016

Outline of Today's Lecture

- Monads!
- Monad combinators

Monad Tutorial Warning

Warning: Please play around with monads yourself!

Monads are an inherently tricky topic because they are *abstract*.

Please play around with any code in this lecture (or the homework, or your own) to get a better understanding.

Monads are sometimes described as burritos

Figure 1:Cat Burrito meme by Brent Yorgey

And there are a lot of monad tutorials

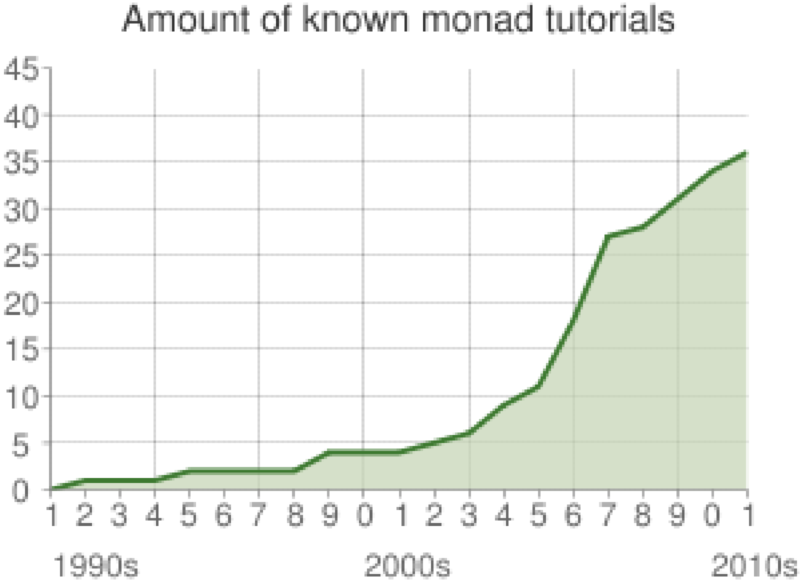

Figure 2:

What monads are not

The What a Monad is not site is quite helpful for dispelling what a monad is not.

What monads are not

The What a Monad is not site is quite helpful for dispelling what a monad is not.

Monads are not

What monads are not

The What a Monad is not site is quite helpful for dispelling what a monad is not.

Monads are not

- a language feature

What monads are not

The What a Monad is not site is quite helpful for dispelling what a monad is not.

Monads are not

- a language feature
- required for Haskell (I/O has been done differently before)

What monads are not

The What a Monad is not site is quite helpful for dispelling what a monad is not.

Monads are not

- a language feature
- required for Haskell (I/O has been done differently before)
- impure

What monads are not

The What a Monad is not site is quite helpful for dispelling what a monad is not.

Monads are not

- a language feature
- required for Haskell (I/O has been done differently before)
- impure
- about state (although they can help with that)

What monads are not

The What a Monad is not site is quite helpful for dispelling what a monad is not.

Monads are not

- a language feature
- required for Haskell (I/O has been done differently before)
- impure
- about state (although they can help with that)
- about strictness

What monads are not

The What a Monad is not site is quite helpful for dispelling what a monad is not.

Monads are not

- a language feature
- required for Haskell (I/O has been done differently before)
- impure
- about state (although they can help with that)
- about strictness
- values (this is a kind error)

What monads are not

The What a Monad is not site is quite helpful for dispelling what a monad is not.

Monads are not

- a language feature
- required for Haskell (I/O has been done differently before)
- impure
- about state (although they can help with that)
- about strictness
- values (this is a kind error)
- about ordering/sequencing (although you can use them this way)

Motivation for Monads

Motivating Example

Let's write a function that zips together two trees, but only if the trees have the same structure.

```
data Tree a = Node (Tree a) a (Tree a)
            | Empty
            deriving (Show)
```

Motivating Example

Let's write a function that zips together two trees, but only if the trees have the same structure.

```
data Tree a = Node (Tree a) a (Tree a)
            | Empty
            deriving (Show)

zipTree1 :: (a -> b -> c) -> Tree a -> Tree b -> Maybe (Tree c)
```

Motivating Example

Let's write a function that zips together two trees, but only if the trees have the same structure.

```
data Tree a = Node (Tree a) a (Tree a)
            | Empty
            deriving (Show)

zipTree1 :: (a → b → c) → Tree a → Tree b → Maybe (Tree c)

zipTree1 _ (Node _ _ _) Empty = Nothing
zipTree1 _ Empty (Node _ _ _) = Nothing
zipTree1 _ Empty Empty        = Just Empty
```

Motivating Example

Let's write a function that zips together two trees, but only if the trees have the same structure.

```haskell
data Tree a = Node (Tree a) a (Tree a)
            | Empty
              deriving (Show)

zipTree1 :: (a -> b -> c) -> Tree a -> Tree b -> Maybe (Tree c)

zipTree1 _ (Node _ _ _) Empty = Nothing
zipTree1 _ Empty (Node _ _ _) = Nothing
zipTree1 _ Empty Empty        = Just Empty

zipTree1 f (Node l1 x r1) (Node l2 y r2) =
    case zipTree1 f l1 l2 of
      Nothing -> Nothing
      Just l  -> case zipTree1 f r1 r2 of
                   Nothing -> Nothing
                   Just r  -> Just $ Node l (f x y) r
```

What can we factor out?

Notice when comparing trees that we know have nodes, we have a similar structure in the case statements

```
zipTree1 f (Node l1 x r1) (Node l2 y r2) =
    case zipTree1 f l1 l2 of
      Nothing -> Nothing
      Just l  -> case zipTree1 f r1 r2 of
                   Nothing -> Nothing
                   Just r  -> Just $ Node l (f x y) r
```

What can we factor out?

Notice when comparing trees that we know have nodes, we have a similar structure in the case statements

```
zipTree1 f (Node l1 x r1) (Node l2 y r2) =
    case zipTree1 f l1 l2 of
      Nothing -> Nothing
      Just l  -> case zipTree1 f r1 r2 of
                   Nothing -> Nothing
                   Just r  -> Just $ Node l (f x y) r
```

We are only doing computation if we hit a **Just**. As pseudocode, we are doing the following

```
f :: Maybe (Tree a) -> (a -> Maybe Tree b) -> Maybe (Tree b)
f = case Maybe (Tree a) =
    Nothing -> Nothing
    Just l -> do something with l, returning Maybe (Tree b)
```

What can we factor out?

We are only doing computation if we hit a **Just**. As pseudocode, we are doing the following

```
f :: Maybe (Tree a) -> (a -> Maybe Tree b) -> Maybe (Tree b)
f = case Maybe (Tree a) =
    Nothing -> Nothing
    Just l -> do something with l, returning Maybe (Tree b)
```

What can we factor out?

We are only doing computation if we hit a **Just**. As pseudocode, we are doing the following

```
f :: Maybe (Tree a) -> (a -> Maybe Tree b) -> Maybe (Tree b)
f = case Maybe (Tree a) =
    Nothing -> Nothing
    Just l -> do something with l, returning Maybe (Tree b)
```

We can write this in Haskell as

```
bindMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
bindMaybe mx f = case mx of
                   Nothing -> Nothing
                   Just x  -> f x
```

Refactor **zipTree**, first attempt

Using **bindMaybe** makes the code quite a bit simpler.

```
zipTree2 :: (a -> b -> c) -> Tree a -> Tree b -> Maybe (Tree c)
zipTree2 _ (Node _ _ _) Empty = Nothing
zipTree2 _ Empty (Node _ _ _) = Nothing
zipTree2 _ Empty Empty        = Just Empty
zipTree2 f (Node l1 x r1) (Node l2 y r2) =
    bindMaybe (zipTree2 f l1 l2) $ \l ->
        bindMaybe (zipTree2 f r1 r2) $ \r ->
            Just (Node l (f x y) r)
```

We already used monads in `zipTree2`

The `zipTree2` function uses Monads! The `Monad` type class is defined as follows:

```
class Monad m where
  return :: a -> m a

  -- pronounced "bind"
  (>>=) :: m a -> (a -> m b) -> m b

  (>>)  :: m a -> m b -> m b
  m1 >> m2 = m1 >>= \_ -> m2
```

We already used monads in zipTree2

The **zipTree2** function uses Monads! The **Monad** type class is defined as follows:

```haskell
class Monad m where
  return :: a -> m a

  -- pronounced "bind"
  (>>=) :: m a -> (a -> m b) -> m b

  (>>)  :: m a -> m b -> m b
  m1 >> m2 = m1 >>= \_ -> m2
```

- **return** takes a value and turns it into a "monadic value".
- **(>>=)** (or "bind", written >>=) does the same as our **bindMaybe**.
- **(>>)** is the same as **(>>=)** but ignores the value in the monadic value.

We already used monads in `zipTree2`

The `zipTree2` function uses Monads! The `Monad` type class is defined as follows:

```haskell
class Monad m where
  return :: a -> m a

  -- pronounced "bind"
  (>>=) :: m a -> (a -> m b) -> m b

  (>>)  :: m a -> m b -> m b
  m1 >> m2 = m1 >>= \_ -> m2
```

- `return` takes a value and turns it into a “monadic value”.
- `(>>=)` (or “bind”, written >>=) does the same as our `bindMaybe`.
- `(>>)` is the same as `(>>=)` but ignores the value in the monadic value.

The `Monad` class has a fourth method, `fail`, but this should never be used (it is unsafe; `fail s = error s`).

Looking more closely at `(>>=)`

`(>>=)` (pronounced “bind”) has the following type:

```
(>>=) :: m a -> (a -> m b) -> m b
```

Looking more closely at (>>=)

(>>=) (pronounced "bind") has the following type:

```
(>>=) :: m a -> (a -> m b) -> m b
```

The first argument is a *monadic value*, `m a`. Here are some examples

Looking more closely at `(>>=)`

`(>>=)` (pronounced "bind") has the following type:

```
(>>=) :: m a -> (a -> m b) -> m b
```

The first argument is a *monadic value*, `m a`. Here are some examples

- `c1 :: Maybe a` is a computation which might fail but results in an `a` if it succeeds.

Looking more closely at `(>>=)`

`(>>=)` (pronounced "bind") has the following type:

```
(>>=) :: m a -> (a -> m b) -> m b
```

The first argument is a *monadic value*, `m a`. Here are some examples

- `c1 :: Maybe a` is a computation which might fail but results in an `a` if it succeeds.
- `c2 :: [a]` is a computation which results in (multiple) `a`s.

Looking more closely at (>>=)

(>>=) (pronounced “bind”) has the following type:

```
(>>=) :: m a -> (a -> m b) -> m b
```

The first argument is a *monadic value*, `m a`. Here are some examples

- `c1 :: Maybe a` is a computation which might fail but results in an `a` if it succeeds.
- `c2 :: [a]` is a computation which results in (multiple) `a`s.
- `c3 :: Rand StdGen a` is a computation which may use pseudo-randomness and produces an `a`.

Looking more closely at $(\gg=)$

$(\gg=)$ (pronounced "bind") has the following type:

```
(>>=) :: m a -> (a -> m b) -> m b
```

The first argument is a *monadic value*, `m a`. Here are some examples

- `c1 :: Maybe a` is a computation which might fail but results in an `a` if it succeeds.
- `c2 :: [a]` is a computation which results in (multiple) `a`s.
- `c3 :: Rand StdGen a` is a computation which may use pseudo-randomness and produces an `a`.
- `c4 :: IO a` is a computation which potentially has some I/O effects and then produces an `a`.

Second argument to (>>=)

(>>=) (pronounced "bind") has the following type:

```
(>>=) :: m a -> (a -> m b) -> m b
```

[1]you can branch using this function

Second argument to `(>>=)`

`(>>=)` (pronounced "bind") has the following type:

```
(>>=) :: m a -> (a -> m b) -> m b
```

The second argument is a function that acts on the non-monadic component of the first argument. This offers the *choice* of what to do with `a`[1].

[1]you can branch using this function

Second argument to `(>>=)`

`(>>=)` (pronounced "bind") has the following type:

```
(>>=) :: m a -> (a -> m b) -> m b
```

The second argument is a function that acts on the non-monadic component of the first argument. This offers the *choice* of what to do with `a`[1].

`(>>=)` puts together two actions to produce a larger one.

[1]you can branch using this function

Second argument to `(>>=)`

`(>>=)` (pronounced "bind") has the following type:

```
(>>=) :: m a -> (a -> m b) -> m b
```

The second argument is a function that acts on the non-monadic component of the first argument. This offers the *choice* of what to do with `a`[1].

`(>>=)` puts together two actions to produce a larger one.

The all-important twist is that we get to decide which action to run second based on the output from the first.

[1]you can branch using this function

Revisiting (>>)

(>>) should make more sense in the context of stitching together components

```
(>>)  :: m a -> m b -> m b
m1 >> m2 = m1 >>= \_ -> m2
```

Revisiting (>>)

(>>) should make more sense in the context of stitching together components

```
(>>)   :: m a -> m b -> m b
m1 >> m2 = m1 >>= \_ -> m2
```

`m1 >> m2` does `m1` and then `m2`, ignoring the result of `m1`.

Examples

Let's start by writing a **Monad** instance for **Maybe**:

```
instance Monad Maybe where
  return  = Just
```

Examples

Let's start by writing a **Monad** instance for **Maybe**:

```
instance Monad Maybe where
  return  = Just

  Nothing >>= _ = Nothing
```

Examples

Let's start by writing a **Monad** instance for **Maybe**:

```
instance Monad Maybe where
  return   = Just

  Nothing >>= _ = Nothing

  Just x  >>= k = k x
```

[2] Incidentally, it is common to use the letter k for the second argument of (>>=) because k stands for "continuation".

Examples

Let's start by writing a `Monad` instance for `Maybe`:

```
instance Monad Maybe where
  return   = Just

  Nothing >>= _ = Nothing

  Just x  >>= k = k x
```

This is the exact same as `bindMaybe` but with pattern matching instead of a case statement. [2]

[2] Incidentally, it is common to use the letter `k` for the second argument of `(>>=)` because `k` stands for "continuation".

Refactor zipTree, second attempt

Rewriting with (>>=) instead of bindMaybe.

```
zipTree3 :: (a -> b -> c) -> Tree a -> Tree b -> Maybe (Tree c)
zipTree3 _ (Node _ _ _) Empty = Nothing
zipTree3 _ Empty (Node _ _ _) = Nothing
zipTree3 _ Empty Empty        = Just Empty
zipTree3 f (Node l1 x r1) (Node l2 y r2) =
    zipTree3 f l1 l2 >>= \l ->
        zipTree3 f r1 r2 >>= \r ->
            return (Node l (f x y) r)
```

`do` notation works for any Monad

The **`do`** notation for working with IO can work with *any* monad.

The backwards arrows that we use in a **`do`** block are just syntactic sugar for binds.

do notation works for any Monad

The **do** notation for working with IO can work with *any* monad.

The backwards arrows that we use in a **do** block are just syntactic sugar for binds.

As an example, the following function

```
addM :: Monad m => m Int -> m Int -> m Int
addM mx my = do
  x <- mx
  y <- my
  return $ x + y
```

do notation works for any Monad

The **do** notation for working with IO can work with *any* monad.

The backwards arrows that we use in a **do** block are just syntactic sugar for binds.

As an example, the following function

```
addM :: Monad m => m Int -> m Int -> m Int
addM mx my = do
  x <- mx
  y <- my
  return $ x + y
```

desugars in GHC to

```
addM' :: Monad m => m Int -> m Int -> m Int
addM' mx my = mx >>= \x -> my >>= \y -> return (x + y)
```

Refactor **zipTree**, third attempt

Now with **do** notation!

```
zipTree :: (a → b → c) → Tree a → Tree b → Maybe (Tree c)
zipTree _ (Node _ _ _) Empty = Nothing
zipTree _ Empty (Node _ _ _) = Nothing
zipTree _ Empty Empty        = Just Empty
zipTree f (Node l1 x r1) (Node l2 y r2) = do
    l ← zipTree f l1 l2
    r ← zipTree f r1 r2
    return $ Node l (f x y) r
```

More Examples

Here are some more examples. Let's start with the `k` functions given to bind.

```
check :: Int → Maybe Int
check n | n < 10    = Just n
        | otherwise = Nothing
```

More Examples

Here are some more examples. Let's start with the `k` functions given to bind.

```
check :: Int -> Maybe Int
check n | n < 10     = Just n
        | otherwise = Nothing

halve :: Int -> Maybe Int
halve n | even n     = Just $ n `div` 2
        | otherwise = Nothing
```

More Examples

Here are some more examples. Let's start with the `k` functions given to bind.

```
check :: Int -> Maybe Int
check n | n < 10    = Just n
        | otherwise = Nothing

halve :: Int -> Maybe Int
halve n | even n    = Just $ n `div` 2
        | otherwise = Nothing

ex01 = return 7 >>= check >>= halve
```

More Examples

Here are some more examples. Let's start with the **k** functions given to bind.

```
check :: Int → Maybe Int
check n | n < 10    = Just n
        | otherwise = Nothing

halve :: Int → Maybe Int
halve n | even n    = Just $ n `div` 2
        | otherwise = Nothing

ex01 = return 7 >>= check >>= halve

ex02 = return 12 >>= check >>= halve
```

More Examples

Here are some more examples. Let's start with the `k` functions given to bind.

```
check :: Int → Maybe Int
check n | n < 10    = Just n
        | otherwise = Nothing

halve :: Int → Maybe Int
halve n | even n    = Just $ n `div` 2
        | otherwise = Nothing

ex01 = return 7 >>= check >>= halve

ex02 = return 12 >>= check >>= halve

ex03 = return 12 >>= halve >>= check
```

Examples using **do** notation

The prior examples can be rewritten using **do** notation

```
ex01 = return 7  >>= check >>= halve
ex02 = return 12 >>= check >>= halve
ex03 = return 12 >>= halve >>= check

ex04 = do
  checked <- check 7
  halve checked
```

Examples using **do** notation

The prior examples can be rewritten using **do** notation

```
ex01 = return 7  >>= check >>= halve
ex02 = return 12 >>= check >>= halve
ex03 = return 12 >>= halve >>= check

ex04 = do
  checked <- check 7
  halve checked

ex05 = do
  checked <- check 12
  halve checked
```

Examples using **do** notation

The prior examples can be rewritten using **do** notation

```
ex01 = return 7 >>= check >>= halve
ex02 = return 12 >>= check >>= halve
ex03 = return 12 >>= halve >>= check

ex04 = do
  checked <- check 7
  halve checked

ex05 = do
  checked <- check 12
  halve checked

ex06 = do
  halved <- halve 12
  check halved
```

Monad for [] constructor

The **Monad** instance for a list is

```
instance Monad [] where
  return x = [x]
  xs >>= k = concatMap k xs
```

Monad for [] constructor

The **Monad** instance for a list is

```
instance Monad [] where
  return x = [x]
  xs >>= k = concatMap k xs

ex07 = [10,20,30] >>= \x -> [x+1, x+2]
ex08 = do
  num <- [10, 20, 30]
  (\x -> [x+1, x+2]) num
```

Monad for [] constructor

The **Monad** instance for a list is

```
instance Monad [] where
  return x = [x]
  xs >>= k = concatMap k xs

ex07 = [10,20,30] >>= \x -> [x+1, x+2]
ex08 = do
  num <- [10, 20, 30]
  (\x -> [x+1, x+2]) num
```

You can also use the (=<<) operator, which is (>>=) with the arguments flipped.

```
ex09 = addOneOrTwo =<< [10,20,30]
```

Monad for [] constructor

The **Monad** instance for a list is

```
instance Monad [] where
  return x = [x]
  xs >>= k = concatMap k xs

ex07 = [10,20,30] >>= \x -> [x+1, x+2]
ex08 = do
  num <- [10, 20, 30]
  (\x -> [x+1, x+2]) num
```

You can also use the `(=<<)` operator, which is `(>>=)` with the arguments flipped.

```
ex09 = addOneOrTwo =<< [10,20,30]
```

The list monad encodes non-determinism (it attempts every possibility of applying a function to every value).

MonadPlus

Lists are also part of a class called **MonadPlus** which allows for computation to abort if some predicate is met (aka failure).

```
ex10 = do
  num ← [1..20]
  guard (even num)
  guard (num `mod` 3 == 0)
  return num
```

or

```
ex11 =             [1..20] >>= \num →
  guard (even num)         >>
  guard (num `mod` 3 == 0) >>
  return num
```

sequence

With just **`return`** and `(>>=)` we can build up quite a few common actions.

sequence

With just **return** and **(>>=)** we can build up quite a few common actions.

sequence takes a list of monadic values and produces a single monadic value which collects the results.

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

Other monad combinators

Using **`sequence`** we can also write other combinators, such as

```
replicateM :: Monad m ⇒ Int → m a → m [a]
replicateM n m = sequence (replicate n m)
```

Other monad combinators

Using **sequence** we can also write other combinators, such as

```
replicateM :: Monad m ⇒ Int → m a → m [a]
replicateM n m = sequence (replicate n m)

void :: Monad m ⇒ m a → m ()
void ma = ma >> return ()
```

Other monad combinators

Using **sequence** we can also write other combinators, such as

```
replicateM :: Monad m ⇒ Int → m a → m [a]
replicateM n m = sequence (replicate n m)

void :: Monad m ⇒ m a → m ()
void ma = ma >> return ()

join :: Monad m ⇒ m (m a) → m a
join mma = do
  ma ← mma
  ma
```

Other monad combinators

Using **sequence** we can also write other combinators, such as

```
replicateM :: Monad m ⇒ Int → m a → m [a]
replicateM n m = sequence (replicate n m)

void :: Monad m ⇒ m a → m ()
void ma = ma >> return ()

join :: Monad m ⇒ m (m a) → m a
join mma = do
  ma ← mma
  ma

when :: Monad m ⇒ Bool → m () → m ()
when b action =
  if b
  then action
  else return ()
```

List comprehensions

List comprehensions make it convenient to build certain lists.

```
evensUpTo100 :: [Int]
evensUpTo100 = [ n | n ← [1..100], even n ]
```

List comprehensions

List comprehensions make it convenient to build certain lists.

```
evensUpTo100 :: [Int]
evensUpTo100 = [ n | n ← [1..100], even n ]
```

In turns out that there is a straightforward translation from list comprehensions to **do** notation:

```
[ a | b ← c, d, e, f ← g, h ]
```

is exactly equivalent to

```
do b ← c
   guard d
   guard e
   f ← g
   guard h
   return a
```

List comprehensions

List comprehensions make it convenient to build certain lists.

```
evensUpTo100 :: [Int]
evensUpTo100 = [ n | n ← [1..100], even n ]
```

In turns out that there is a straightforward translation from list comprehensions to **do** notation:

```
[ a | b ← c, d, e, f ← g, h ]
```

is exactly equivalent to

```
do b ← c
   guard d
   guard e
   f ← g
   guard h
   return a
```

It is possible to use this syntax for any monad using the GHC language extension `MonadComprehensions`.