CS 350 2024-25 Sem I Lecture 11

Satyadev Nandakumar

September 4 2024

Outline

Outline

Motivation

- used for side-effects in various settings

Motivation

- used for side-effects in various settings
- the various Monads have different uses.

Motivation

- used for side-effects in various settings
- the various Monads have different uses.
- concept best understood through concrete examples

Motivation

- used for side-effects in various settings
- the various Monads have different uses.
- concept best understood through concrete examples
- We see the most important monads in the next 2 lectures.

Definition

The Monad class

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

Definition

The Monad class

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

More

Definition

The Monad class

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

More

- `return` is usually `pure` from `Applicative`

Definition

The Monad class

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

More

- `return` is usually `pure` from `Applicative`
- We focus on the >>= (bind) function.

Definition

The Monad class

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

More

- `return` is usually `pure` from `Applicative`
- We focus on the >>= (bind) function.
- Its first input is a "boxed a" value

Definition

The Monad class

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

More

- `return` is usually `pure` from `Applicative`
- We focus on the >>= (bind) function.
- Its first input is a "boxed a" value
- Its second input is a function that takes "unboxed a" and returns "boxed b".

Definition

The Monad class

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

More

- `return` is usually `pure` from `Applicative`
- We focus on the >>= (bind) function.
- Its first input is a "boxed a" value
- Its second input is a function that takes "unboxed a" and returns "boxed b".
- The return value of bind is "boxed b" - the output of the function which is the second argument

Outline

The bind operator

- the bind operator is `(>>=) :: m a -> (a -> m b) -> m b`

The bind operator

- the bind operator is `(>>=)::m a -> (a -> m b) -> m b`
- The first argument is of boxed type

The bind operator

- the bind operator is `(>>=)::m a -> (a -> m b) -> m b`
- The first argument is of boxed type
- The function requires an unboxed value

The bind operator

- the bind operator is `(>>=)::m a -> (a -> m b) -> m b`
- The first argument is of boxed type
- The function requires an unboxed value
- Point to ponder: how do we unbox the first argument?

The bind operator

- the bind operator is `(>>=)::m a -> (a -> m b) -> m b`
- The first argument is of boxed type
- The function requires an unboxed value
- Point to ponder: how do we unbox the first argument?
  - in the definition of `(>>=)` through pattern-matching, for example.

The bind operator

- the bind operator is `(>>=)::m a -> (a -> m b) -> m b`
- The first argument is of boxed type
- The function requires an unboxed value
- Point to ponder: how do we unbox the first argument?
  - in the definition of `(>>=)` through pattern-matching, for example.
- let's look at a few examples

Outline

Maybe as a Monad

code

```
instance Monad MyMaybe where
  return = pure
  MyNothing >>= f = MyNothing
  (MyJust x) >>= f = f x
```

Example usage for 1 argument functions

Suppose we have a "safe" version of integer division which can handle division by 0 in a safe manner, by returning Nothing.

safe reciprocal

```
safeReciprocal 0 = MyNothing
safeReciprocal x = MyJust (1/x)
```

Example usage for 1 argument functions

Suppose we have a "safe" version of integer division which can handle division by 0 in a safe manner, by returning Nothing.

safe reciprocal

```
safeReciprocal 0 = MyNothing
safeReciprocal x = MyJust (1/x)
```

Calling `safeDiv` using bind

```
MyJust 0.5 >>= safeReciprocal >>= safeReciprocal
-- equals MyJust 0.5
```

```
MyJust 0 >>= safeReciprocal >>= safeReciprocal
-- equals MyNothing
```

Pipes with types

- Monads are often called "pipes with types"

Pipes with types

- Monads are often called "pipes with types"
- Specifically, this refers to the use of >>= analogous to Unix pipes

Example usage for 2 argument functions (important)

- for multiargument functions, we must use currying

Example usage for 2 argument functions (important)

- for multiargument functions, we must use currying
- let us see how a 2 argument function, `safeDiv` can be written - it is integer division which safely handles division by 0.

Example usage for 2 argument functions (important)

- for multiargument functions, we must use currying
- let us see how a 2 argument function, `safeDiv` can be written - it is integer division which safely handles division by 0.

safe integer division

```
safeDiv m 0 = MyNothing
safeDiv m n = MyJust (m `div` n)
```

Example usage for 2 argument functions (important)

- There are two arguments, so there must be two occurrences of >>= in the expression. How is this done?

Story:

Example usage for 2 argument functions (important)

- There are two arguments, so there must be two occurrences of >>= in the expression. How is this done?

Story:

1. The left operand of the first >>= is the boxed first argument, say `mx`

Example usage for 2 argument functions (important)

- There are two arguments, so there must be two occurrences of >>= in the expression. How is this done?

Story:

1. The left operand of the first >>= is the boxed first argument, say `mx`
2. The second argument to the right operand of the first >>= must be a function : a unary function.

Example usage for 2 argument functions (important)

- There are two arguments, so there must be two occurrences of >>= in the expression. How is this done?

Story:

1. The left operand of the first >>= is the boxed first argument, say `mx`
2. The second argument to the right operand of the first >>= must be a function : a unary function.
    1. it should take the unboxed first operand, say `x`

Example usage for 2 argument functions (important)

- There are two arguments, so there must be two occurrences of >>= in the expression. How is this done?

Story:

1. The left operand of the first >>= is the boxed first argument, say `mx`
2. The second argument to the right operand of the first >>= must be a function : a unary function.
   1. it should take the unboxed first operand, say `x`
   2. it should return an expression of the form `my >>= foo` where

Example usage for 2 argument functions (important)

- There are two arguments, so there must be two occurrences of >>= in the expression. How is this done?

Story:

1. The left operand of the first >>= is the boxed first argument, say `mx`
2. The second argument to the right operand of the first >>= must be a function : a unary function.
   1. it should take the unboxed first operand, say `x`
   2. it should return an expression of the form `my >>= foo` where
      1. `my` is the boxed second operand to `safeDiv`

Example usage for 2 argument functions (important)

- There are two arguments, so there must be two occurrences of >>= in the expression. How is this done?

Story:

1. The left operand of the first >>= is the boxed first argument, say `mx`
2. The second argument to the right operand of the first >>= must be a function : a unary function.
   1. it should take the unboxed first operand, say `x`
   2. it should return an expression of the form `my >>= foo` where
      1. `my` is the boxed second operand to `safeDiv`
      2. `foo` is a function that takes the unboxed second operand, say `y`, and evaluates to `safeDiv x y`

Example usage for 2 argument functions (important)

Calling `safeDiv` using `>>=`

```
(MyJust 4) >>= (\m ->
        (MyJust 2) >>= (\n ->
                safeDiv m n))
```

Outline

The do notation as an alternative to >>=

using the >>= notation

```
(MyJust 4) >>= (\m ->
        (MyJust 2) >>= (\n ->
                safeDiv m n))
```

The do notation as an alternative to >>=

using the >>= notation

```
(MyJust 4) >>= (\m ->
        (MyJust 2) >>= (\n ->
                safeDiv m n))
```

using the do notation

```
do
  m <- MyJust 4
  n <- MyJust 2
  safeDiv m n
```

The do notation

```
m1 >>= \x1 ->
m2 >>= \x2 ->
   ...
mn >>= \xn ->
f x1 x2 ... xn
```

```
do
   x1 <- m1
   x2 <- m2
   ...
   xn <- mn
   f x1 x2 ... xn
```

Outline

The List monad

code

```
instance Monad [] where
  [] >>= f = []
  xs >>= f = [y | x <- xs, y <- f x]
```

The List monad

code

```
instance Monad [] where
  [] >>= f = []
  xs >>= f = [y | x <- xs, y <- f x]
```

Example usage

```
[1,2,3] >>= (\x ->[1+x])
```

Note that the function returns a *singleton list* -why?

The List monad

code

```
instance Monad [] where
  [] >>= f = []
  xs >>= f = [y | x <- xs, y <- f x]
```

Example usage

```
[1,2,3] >>= (\x ->[1+x])
```

Note that the function returns a *singleton list* -why?

The List monad

code

```
instance Monad [] where
  [] >>= f = []
  xs >>= f = [y | x <- xs, y <- f x]
```

Example usage

```
[1,2,3] >>= (\x ->[1+x])
```

Note that the function returns a *singleton list* -why?
to be type compatible for >>=

Outline

Outline

Generic functions

- work on all Monads

Generic functions

- work on all Monads

Monadic map

Example code