

CS 350 2024-25 Sem I Lecture 9

Satyadev Nandakumar

August 31, 2024

- 1 Applicatives (continued)
- 2 Monads - introduction
- 3 Monads - Input output in Haskell
- 4 The do notation

- 1 Applicatives (continued)
- 2 Monads - introduction
- 3 Monads - Input output in Haskell
- 4 The do notation

One way to think about Functors versus Applicative

```
(pure add3) <*> (MyJust 1) <*> (MyJust 2) <*> (MyJust 3)
```

can be written in prefix form as as

```
(<*>) ((<*>) ((<*>) (pure add3) (MyJust 1))  
      (MyJust 2))  
      (MyJust 3))
```

One way to think about Functors versus Applicative

(<*>)

((<*>)

((<*> (pure add3) (MyJust 1))

_____/
boxed boxed

function value

unlike in fmap,
function is also
boxed

_____/
boxed, curried function

return value is
a boxed function, so
form same as
the innermost (<*>)

(MyJust 2))

_____/
boxed value

_____/
boxed, curried function

(MyJust 3))

_____/
boxed value

boxed value

Outline

- 1 Applicatives (continued)
- 2 Monads - introduction**
- 3 Monads - Input output in Haskell
- 4 The do notation

- The boxed curried functions in `Applicative` "carry a context" (previous arguments)

Motivation

- The boxed curried functions in `Applicative` "carry a context" (previous arguments)
- in some sense, they carry a state

Motivation

- The boxed curried functions in `Applicative` "carry a context" (previous arguments)
- in some sense, they carry a state
- Can we have mutable state in Haskell?

- The boxed curried functions in `Applicative` "carry a context" (previous arguments)
- in some sense, they carry a state
- Can we have mutable state in Haskell?
- Important in practical applications - input/output, random number generation etc.

Outline

- 1 Applicatives (continued)
- 2 Monads - introduction
- 3 Monads - Input output in Haskell**
- 4 The do notation

Hello, World in Haskell

```
io_1.hs
```

```
main = putStrLn "Hello, World!"
```

Hello, World in Haskell

```
io_1.hs
```

```
main = putStrLn "Hello, World!"
```

Hello, World in Haskell

```
io_1.hs
```

```
main = putStrLn "Hello, World!"
```

- 1 We can compile this using `ghc` and run the executable (OR)

Hello, World in Haskell

```
io_1.hs
```

```
main = putStrLn "Hello, World!"
```

- 1 We can compile this using `ghc` and run the executable (OR)
- 2 Load `io_1.hs` interactively in `ghci`, and call `main`

- reading from the console, displaying a GUI, outputting to the terminal all change the state of the world.

- reading from the console, displaying a GUI, outputting to the terminal all change the state of the world.
- An interactive program: a pure function taking

- reading from the console, displaying a GUI, outputting to the terminal all change the state of the world.
- An interactive program: a pure function taking
 - the state of the world as an argument

- reading from the console, displaying a GUI, outputting to the terminal all change the state of the world.
- An interactive program: a pure function taking
 - the state of the world as an argument
 - returning a modified state of the world as output.

- reading from the console, displaying a GUI, outputting to the terminal all change the state of the world.
- An interactive program: a pure function taking
 - the state of the world as an argument
 - returning a modified state of the world as output.
- The modified state will have the side-effects of interaction

- reading from the console, displaying a GUI, outputting to the terminal all change the state of the world.
- An interactive program: a pure function taking
 - the state of the world as an argument
 - returning a modified state of the world as output.
- The modified state will have the side-effects of interaction

- reading from the console, displaying a GUI, outputting to the terminal all change the state of the world.
- An interactive program: a pure function taking
 - the state of the world as an argument
 - returning a modified state of the world as output.
- The modified state will have the side-effects of interaction

A first attempt is type `IO = World -> World`

- but I/O also may return a value (for example, reading a character).

- but I/O also may return a value (for example, reading a character).
- Hence, type `IO a = World -> (a,World)` may be better

- but I/O also may return a value (for example, reading a character).
- Hence, type `IO a = World -> (a,World)` may be better
- Expressions whose type is `IO` are called *actions*.

- but I/O also may return a value (for example, reading a character).
- Hence, type `IO a = World -> (a,World)` may be better
- Expressions whose type is `IO` are called *actions*.
e.g. `~ getChar IO Char ~`

- but I/O also may return a value (for example, reading a character).
- Hence, type `IO a = World -> (a,World)` may be better
- Expressions whose type is `IO` are called *actions*.
 - e.g. `~ getChar IO Char ~`
 - e.g. If your code has a `main` function doing I/O, then `~main IO ()~`

Outline

- 1 Applicatives (continued)
- 2 Monads - introduction
- 3 Monads - Input output in Haskell
- 4 The do notation

io_2.hs : an interactive Hello

```
main = do
  putStrLn "Hi there, what's your name?"
  name <- getLine
  putStrLn $ "Hello, " ++ name ++ "!"
```

io_2.hs : an interactive Hello

```
main = do
  putStrLn "Hi there, what's your name?"
  name <- getLine
  putStrLn $ "Hello, " ++ name ++ "!"
```

Type of main

io_2.hs : an interactive Hello

```
main = do
  putStrLn "Hi there, what's your name?"
  name <- getLine
  putStrLn $ "Hello, " ++ name ++ "!"
```

Type of main

```
1 main :: IO ()
```


io_2.hs : an interactive Hello

```
main = do
  putStrLn "Hi there, what's your name?"
  name <- getLine
  putStrLn $ "Hello, " ++ name ++ "!"
```

Type of main

- 1 main :: IO ()
- 2 returns an IO action with an empty tuple

io_2.hs : an interactive Hello

```
main = do
  putStrLn "Hi there, what's your name?"
  name <- getLine
  putStrLn $ "Hello, " ++ name ++ "!"
```

Type of main

- 1 main :: IO ()
- 2 returns an IO action with an empty tuple
- 3 Can be seen as returning "void" in C

io_2.hs : an interactive Hello

```
main = do
  putStrLn "Hi there, what's your name?"
  name <- getLine
  putStrLn $ "Hello, " ++ name ++ "!"
```

Type of main

- 1 main :: IO ()
- 2 returns an IO action with an empty tuple
- 3 Can be seen as returning "void" in C
- 4 happens usually when the last statement in main is an output function.

Explanation (continued)

- 1 Note the `<-` for reading input.

Explanation (continued)

- 1 Note the `<-` for reading input.
- 2 We saw that last in the *list comprehension* notation

Explanation (continued)

- 1 Note the `<-` for reading input.
- 2 We saw that last in the *list comprehension* notation
- 3 There, it denoted, reading from a list in sequence

Explanation (continued)

- 1 Note the `<-` for reading input.
- 2 We saw that last in the *list comprehension* notation
- 3 There, it denoted, reading from a list in sequence
- 4 Similarly, it denotes reading from an IO stream here

Explanation (continued)

- 1 Note the `<-` for reading input.
- 2 We saw that last in the *list comprehension* notation
- 3 There, it denoted, reading from a list in sequence
- 4 Similarly, it denotes reading from an IO stream here
- 5 In this case, `getLine:: IO String`

main returning values

io_2.hs modified

```
main = do
  putStrLn "Hi there, what's your name?"
  name <- getLine
  return name
```

main returning values

io_2.hs modified

```
main = do
  putStrLn "Hi there, what's your name?"
  name <- getLine
  return name
```

Type of main

main returning values

io_2.hs modified

```
main = do
  putStrLn "Hi there, what's your name?"
  name <- getLine
  return name
```

Type of main

① `main:: IO String`

main returning values

io_2.hs modified

```
main = do
  putStrLn "Hi there, what's your name?"
  name <- getLine
  return name
```

Type of main

- 1 main:: IO String
- 2 To return a value, use return (this is actually a function inside Monad, not a keyword.)

main returning values

io_2.hs modified

```
main = do
  putStrLn "Hi there, what's your name?"
  name <- getLine
  return name
```

Type of main

- 1 main:: IO String
- 2 To return a value, use return (this is actually a function inside Monad, not a keyword.)
- 3 See :info Monad

More about the do notation

- A sequence of IO actions can be combined into a single expression using `do`.

More about the do notation

- A sequence of IO actions can be combined into a single expression using `do`.
- lines are executed one after the other (succeeding lines can use variables in preceding lines)

More about the do notation

- A sequence of IO actions can be combined into a single expression using `do`.
- lines are executed one after the other (succeeding lines can use variables in preceding lines)
- Statements of the form `v <- foo` are called generators (similar as in list comprehension)

More about the do notation

- A sequence of IO actions can be combined into a single expression using `do`.
- lines are executed one after the other (succeeding lines can use variables in preceding lines)
- Statements of the form `v <- foo` are called generators (similar as in list comprehension)
- The **right** side of every `<-` inside `do` will be an action (i.e. expression of type IO)

More about the do notation

- A sequence of IO actions can be combined into a single expression using `do`.
- lines are executed one after the other (succeeding lines can use variables in preceding lines)
- Statements of the form `v <- foo` are called generators (similar as in list comprehension)
- The **right** side of every `<-` inside `do` will be an action (i.e. expression of type IO)
- statements within `do` must be laid out according to Haskell indentation rules.

More about the do notation

- A sequence of IO actions can be combined into a single expression using `do`.
- lines are executed one after the other (succeeding lines can use variables in preceding lines)
- Statements of the form `v <- foo` are called generators (similar as in list comprehension)
- The **right** side of every `<-` inside `do` will be an action (i.e. expression of type IO)
- statements within `do` must be laid out according to Haskell indentation rules.

```
io_3.hs : let inside do
```

```
main = do
  putStrLn "Hi there, what's your name?"
  let nameM = getLine          -- new
      name <- nameM
  putStrLn $ "Hello, " ++ name ++ "!"
```

Example program : reverse every word in the input line

Reverse every word in every line (io_4.hs)

```
main = do
  line <- getLine
  if null line
  then return () -- see :t return
  else do
    putStrLn $ reverseWords line
    main -- recurse

reverseWords = unwords . map reverse . words
```

Example program : adding input integers

Adding numbers input by user (empty line to terminate)

```
main = add_numbers 0
```

```
add_numbers init_val = do
  next_number_string <- getLine
  if null next_number_string
  then putStrLn $ show init_val
  else do
    let
      next_number = read next_number_string::Float
    add_numbers $ init_val + next_number
```

- 1 shows that other functions can also use the do notation

Explanation

- 1 shows that other functions can also use the do notation
- 2 note the coercion operator (`::`) to convert string to Float

Explanation

- 1 shows that other functions can also use the do notation
- 2 note the coercion operator (`::`) to convert string to Float
- 3 What is the type of `read` ?