# CS 350 2024-25 Sem | Lecture 9

Satyadev Nandakumar

August 31, 2024

Satyadev Nandakumar

CS 350 2024-25 Sem | Lecture 9

August 31, 2024

### Applicatives (continued)

- 2 Monads introduction
- 3 Monads Input output in Haskell

## The do notation

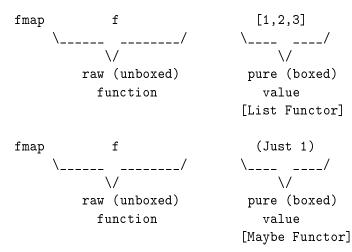
## Applicatives (continued)

### 2 Monads - introduction

### 3 Monads - Input output in Haskell

### The do notation

# One way to think about Functors versus Applicative



August 31, 2024

(pure add3) <\*> (MyJust 1) <\*> (MyJust 2) <\*> (MyJust 3) can be written in prefix form as as

# One way to think about Functors versus Applicative

```
(<*>)
  ((<*>)
    ((<*>)
               (pure add3) (MyJust 1))
                                             unlike in fmap,
                                             function is also
                 boxed
                                boxed
                                             boxed
                function
                                value
                                              return value is
      boxed, curried function
                                              a boxed function, so
    (MyJust 2))
                                              form same as
    \ /
                                              the innermost (<*>)
    boxed value
    boxed, curried function
  (MyJust 3))
  hoved walu
  Satvadev Nandakumar
                        CS 350 2024-25 Sem | Lecture 9
                                                    August 31, 2024
                                                                 6 / 20
```

### Applicatives (continued)

## 2 Monads - introduction

3 Monads - Input output in Haskell

### The do notation

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

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

- 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.

## Applicatives (continued)

### 2 Monads - introduction

### 3 Monads - Input output in Haskell

### 4) The do notation

main = putStrLn "Hello, World!"

Satyadev Nandakumar

CS 350 2024-25 Sem | Lecture 9

August 31, 2024

3

main = putStrLn "Hello, World!"

Satyadev Nandakumar

CS 350 2024-25 Sem | Lecture 9

August 31, 2024

10 / 20

э

main = putStrLn "Hello, World!"

• We can compile this using ghc and run the executable (OR)

Satyadev Nandakumar

CS 350 2024-25 Sem | Lecture 9

August 31, 2024

main = putStrLn "Hello, World!"

We can compile this using ghc and run the executable (OR)
Load io\_1.hs interactively in ghci, and call main

Satyadev Nandakumar

CS 350 2024-25 Sem | Lecture 9

August 31, 2024

• 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 ()~

## Applicatives (continued)

### 2 Monads - introduction

### 3 Monads - Input output in Haskell

## The do notation

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

□ ▶ ▲ 臣 ▶ ▲ 臣 ▶ □ 臣 □ - のへ()~

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

### Type of main

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

### Type of main



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

## Type of main

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

< 47 > <

B A B A B A A A

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

### Type of main

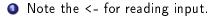
- 1 main :: IO ()
- 2 returns an IO action with an empty tuple
- O Can be seen as returning "void" in C

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

### Type of main

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

= ~ ~ ~



э

- One the <- for reading input.</p>
- We saw that last in the *list comprehension* notation

- One the <- for reading input.</p>
- We saw that last in the list comprehension notation
- There, it denoted, reading from a list in sequence

- Note the <- for reading input.</p>
- We saw that last in the list comprehension notation
- There, it denoted, reading from a list in sequence
- Similarly, it denotes reading from an IO stream here

- Note the <- for reading input.</p>
- We saw that last in the list comprehension notation
- There, it denoted, reading from a list in sequence
- Similarly, it denotes reading from an IO stream here
- In this case, getLine:: IO String

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

3

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

## Type of main

Satyadev Nandakumar

3

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

## Type of main

1 main:: IO String

3

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

## Type of main

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

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

## Type of main

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

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

- 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)

- 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 <u>generators</u> (similar as in list comprehension)

- 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 <u>generators</u> (similar as in list comprehension)
- The right side of every <- inside do will be an <u>action</u> (i.e. expression of type IO)

- 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 <u>generators</u> (similar as in list comprehension)
- The right side of every <- inside do will be an <u>action</u> (i.e. expression of type IO)
- statements within do must be laid out according to Haskell indentation rules.

- 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)</p> 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 ++ "!"
   Satyadev Nandakumar
```

CS 350 2024-25 Sem | Lecture 9

August 31, 2024

```
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
```

### 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
```



э

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

- Ishows that other functions can also use the do notation
- Inote the coercion operator (::) to convert string to Float
- What is the type of read ?