Monads

Tutorial FP week 12

Ruben Holubek

Radboud University Nijmegen

8th of December, 2021

Table of contents

Table of contents

Introduction I

- I'm Ruben Holubek, Master student Software Science, focus on education of Computing Science
- Did some research on teaching programming
- Putting it in practice here for 2 weeks with the some challenging topics in FP
- Last week, I presented Functors and Applicatives, this week I will present Monads
- At the end, I hope you have a better idea how Functors, Applicatives and Monads work

Introduction II

- I will make use of an online quiz during the lecture for small exercises
- Please go to `www.Socrative.com`
- Click on login (upper right corner)
- Click on Student login
- Insert room name HOLUBEK60

Table of contents

Introduction I

- Suppose we defined a function `half`:
- ```
  half :: Int -> Maybe Int
  half x = if (even x)
              then Just (x `div` 2)
              else Nothing
  ```
- If the given integer is even, it simply divides by 2, but if the integer is odd, it returns Nothing
- half 2 = Just 1
- half 1 = Nothing
- Great! We can use it on Integers!
- But what if we want to use it on a Maybe Integers?
  ```

Introduction II

- First, lets write it out:
- ```
  halfMaybe :: Maybe Int -> Maybe Int
  halfMaybe Nothing = Nothing
  halfMaybe (Just x) = half x
  ```
- halfMaybe Nothing = Nothing
- halfMaybe (Just 3) = Nothing
- halfMaybe (Just 4) = Just 2
- It works, but it is not maintainable when applying more functions...
- And we have these case distinctions again
- These has to be an easier way!
- Cannot we use a Functor or Applicative to solve this problem?
  ```

Introduction III

- Recall that Maybe is a Functor with fmap:
- ```
  -- fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)
  ```
- fmap half (Just 2) = Just (Just 1)
- There is an extra Just, but why does this happen?
- fmap applies the function half on the 2 in (Just 2), thus resulting in (Just (Just 1))
- We are on the right track, but this won't work...
- What about an Applicative?
  ```

Introduction IV

- Recall that Maybe is also an Applicative:
- ```
  -- pure :: a -> Maybe a
  pure x = Just x
  -- (<*>) :: f (a -> b) -> Maybe a -> Maybe b
  Nothing <*> _ = Nothing
  (Just f) <*> x = fmap f x
  ```
- pure half <*> (Just 2) = Just (Just 1)
- We have a similar problem here with the double Just...
- Lets see in more detail why the current type classes are not sufficient
  ```

Introduction V

- `fmap :: (a -> b) -> f a -> f b`
- fmap gets a direct function `(a -> b)`
- `(<*>) :: f (a -> b) -> f a -> f b`
- <*> gets a function completely wrapped inside a container: `f (a -> b)`
- However, our `half` function has type `(a -> f b)`, which does not occur in fmap or <*>
- Even though it is a function type that occurs a lot!
- So a function with the following type would be helpful:
- `:: f a -> (a -> f b) -> f b`
- The idea would be to "extract" the element of type `a` from the `f a` and apply the function to it
- This is exactly what a **Monad** does!

Monad definition I

- A Monad is another class for containers which can "extract" values from a container and apply a function on it:
-
```
1. class (Applicative m) => Monad m where
2.     return :: a -> m a
3.     (>>=) :: m a -> (a -> m b) -> m b
```
- Line 1 states that every container that is a Monad is also an Applicative
  - Thus we can use pure and <*>
  - But also that Monads are Functors, so we can use fmap
- Line 2 states the type of the function return, which puts a value in a "default" container
  - Remember pure? return does exactly the same (therefore, it is automatically derived by Haskell!)
- Line 3 states the type of >>= (Bind) takes a "wrapped" value and a function that takes a "normal" value and applies it on each other
  - It "shoves" our wrapped value inside the function

Intuition

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

Monad definition II

- ```
  class (Applicative m) => Monad m where
      return :: a -> m a
      (>>=) :: m a -> (a -> m b) -> m b
  ```
- (>>=) can be seen as a special function application
- Instead of taking a **normal** value and feeding it to a **normal** function
- We take a **wrapped** value and feed it to a **normal** function
- But we have to **unwrap**/extract it before we can use it
- For which we need a *plunger*, which is the (>>=) operator
- Main question within Monads: "If we have a fancy value and a function that takes a normal value but returns a fancy value, how do we feed that fancy value into the function?"
  ```

Maybe example I

- Lets take a look at the definition of Maybe:
- ```
  instance Monad Maybe where
      -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
      Nothing >>= _ = Nothing
      (Just x) >>= f = f x
  ```
- If the first value is Nothing, it doesn't matter what the function is, we simply return Nothing (similar to Applicative)
- If the first value is (Just x), we
  1. "Extract" the x by pattern matching
  2. And apply f `(::a -> Maybe b)` to x `(::a)` to get the desired type `Maybe b`
- As can be seen, the "extracting" is done via pattern matching
- The (>>=) indeed looks a lot like a function application
- Lets see how we can use this
  ```

Maybe example II

- Lets investigate the following expression:
- `Just 3 >>= (\x -> Just(x*2)) = Just 6`
- The following is happening:
  1. The 3 is extracted from Just 3 by >>=
  2. The extracted 3 is then send forward in variable x
  3. Which is evaluated in Just(x*2)
  4. Which yields the result Just 6
- Please note that:
  - `Just 3 :: Maybe Int`
  - `Just(x*2) :: Maybe Int`
  - `x :: Int`
  - `(>>=) :: Maybe Int -> (Int -> Maybe Int) -> Maybe Int`
  - `(>>=) :: m a -> (a -> m b) -> m b`
- The types all match!

Maybe example III

- `Just 3 >>= (\x -> Just(x*2)) = Just 6`
- return (or pure) could also be used instead of Just:
- `Just 3 >>= (\x -> return(x*2)) = Just 6`
- Lets investigate the other case:
- `Nothing >>= (\x -> return (x*2)) = Nothing`
- The following is happening:
  1. There is nothing to extract from Nothing by >>=
  2. So Nothing is returned
- Even though there are a lot of case distinctions that should be handled, the programmer doesn't have to think about these anymore
- Once used to the notation, binds are a very powerful tool!
- But very useful!

Quiz

3. What is the result of the following expression:

```
Just 3 >>= (\x → x+2)
```

A 5

B Just 3

C Just 5

D Error, as the types are incorrect

Quiz

4. What is the result of the following expression:

```
Just 3 >>= (\x → return (x+1))
```

A 4

B Just 4

C Just (Just 4)

D Error, as the types are incorrect

Quiz

5. What is the result of the following expression:

```
3 >>= (\x → Just 6)
```

A Just 3

B Just 6

C Just (Just 6)

D Error, as the types are incorrect

Quiz

6. What is the result of the following expression:

```
Just 3 >>= pure
```

A 3

B Just 3

C Just (Just 3)

D Error, as the types are incorrect

Bind in detail I

- Similar to an Applicative, we can also chain these operations:
- `Just 3 >>= (\x -> return (x*2)) >>= (\y -> return (y+1)) = Just 7`
- The following is happening:
  1. First 3 is extracted from Just 3
  2. This 3 is put in the red function `(\x -> return (x*2))`, which produces Just 6
  3. With the second >>=, the 6 is extracted from Just 6
  4. This 6 is put in the orange function `(\y -> return (y+1))`, which produces Just 7
- Remember, if at one point the operation results in Nothing, the whole expression returns Nothing:
- `Just 3 >>= (\x -> if x == 3 then Nothing else Just x) >>= (\y -> return( y+2)) = Nothing`
- You can chain as many bind operators as you want!

Bind in detail II

- `Just 3 >>= (\x -> return (x*2)) >>= (\y -> return (y+1))`
- We explicitly wrote brackets around the different function, thus limiting the scopes of their variables
- These brackets can be removed, which results in something interesting
- `Just 3 >>= \x -> return (x*2) >>= \y -> return (y+1)`
- The scope of `x` is extended to the second function!
- Which means we can use `x` in the orange function as well!
- `Just 3 >>= \x -> return (x*2) >>= \y -> return (y+x)`

Bind in detail III

- `Just 3 >>= \x -> return (x*2) >>= \y -> return (y+x)=Just 9`
- 1. 3 is extracted from Just 3 and put in x
  2. The red function evaluates to Just 6
  3. The 6 is extracted from Just 6 and put in y
  4. The orange function then evaluates to Just 9 (with the previously stored 3 in x)
- So, the chaining of (>>=) can be seen as smaller **(sub)computations**, of which the **extraction** is then stored in a **variable**, which is later re-used:
- 1. 3 is extracted from Just 3, stored in x
  2. 6 is extracted from return (x*2), stored in y
  3. return (y+x) is the result
- An intermediate result is **bound** to a variable
- This syntax is getting quite long when using multiple (>>=), but luckily Haskell has a special notation for this!

Do notation I

- A typical Monad operation looks like this:
- ```
  m1 >>= \x1 ->
  m2 >>= \x2 ->
  ...
  mn >>= \xn ->
  f x1 x2 ... xn
  ```
- Haskell provides the special do-notation for this:
- ```
  do  x1 <- m1
      x2 <- m2
      ...
      xn <- mn
      f x1 x2 ... xn
  ```
- Note that do is layout sensitive, so use tabs/spaces precisely!

Do notation II

- Recall our operation:
- ```
  Just 3          >>= \x ->
  return (x*2)    >>= \y ->
  return (y+x)
  ```
- 1. 3 is extracted from Just 3, stored in x
  2. 6 is extracted from return (x*2), stored in y
  3. return (y+x) is the result (Just 9)
- This can be written in the do-notation:
- ```
  do  x   <- Just 3
      y   <- return (x*2)
      return (y+x)
  ```
- This looks very similar to an imperative language, where we have variables and other function calls!
- It is also pretty easy to read!

Lookup example I

- Lets take a look at another example
- Suppose a course where students make exercises in groups
- We could have a directory `groupgrades` which contains the grade for each group (small on purpose):
- `groupgrades = [("group 1",7),("group 2",9)]`
- We also have a directory `studentsgroup` which contains a students name and their group:
- `studentsgroup = [("Alice","group 1"),("Bob","group 2")]`
- We also have access to the function `lookup :: Eq a => a -> [(a,b)] -> Maybe b`
- E.g. lookup "Alice" studentsgroup = Just "group 1"
- E.g. lookup "Eve" studentsgroup = Nothing
- The idea is to make a function nameToGrade, which returns the grade for the given name if it exists

Lookup example II

- `groupgrades = [("group 1",7),("group 2",9)]`
- `studentsgroup = [("Alice","group 1"),("Bob","group 2")]`
- The function nameToGrade returns the grade for a given name if it exists
- This can be solved by using (>>=):
- ```
  nameToGrade :: String -> Maybe Int
  nameToGrade name =
      lookup name studentsgroup >>= \group ->
      lookup group groupgrades
  ```
- `nameToGrade "Alice" = Just 7`
- `nameToGrade "Eve" = Nothing`
  ```

Lookup example III

- ```
  nameToGrade :: String -> Maybe Int
  nameToGrade name =
      lookup name studentsgroup >>= \group ->
      lookup group groupgrades
  ```
- This can also be written by the do notation:
- ```
  nameToGrade name = do
      group <- lookup name studentsgroup
      lookup group groupgrades
  ```
- It indeed looks like we are storing intermediate results!
- Now suppose we have another directory snumbersNames with s-numbers linked to names:
- ```
  snumbersNames =
  [("s123456","Alice"),("s987654","Bob")]
  ```
  ```

Lookup example IV

- `groupgrades = [("group 1",7),("group 2",9)]`
- `studentsgroup = [("Alice","group 1"),("Bob","group 2")]`
- `snumbersNames = [("s123456","Alice"),("s987654","Bob")]`
- The idea is to make a function snumberToGrade that returns the grade of the given s-number if possible:
- ```
  snumberToGrade :: String -> Maybe Int
  snumberToGrade snumber = do
      name <- lookup snumber snumbersNames
      group <- lookup name studentsgroup
      lookup group groupgrades
  ```
- The do notation is really useful for intermediate results and structured computations
- All the case distincation are solved by the Monad!
  ```

Quiz

7. What is the result of the following expression:

```
Just 3 >>= \x → 3 + x >>= \y → return (x+y)
```

A Just 3

B Just 6

C Just 9

D Just 12

E Error, as the types are incorrect

Quiz

8. What is the result of the following expression:

```
Just 3 >>= \x → return (3 + x) >>= \y → return (x+y)
```

A Just 3

B Just 6

C Just 9

D Just 12

E Error, as the types are incorrect

Quiz

9. What is an equivalent do notation of the following expression:
`Just 3 >>= \x → return (3 + x) >>= \y → return (x+y)`

A
```
do
  x ← Just 3
  y ← Just (3 + x)
  return (x+y)
```

B
```
do
  y ← Just (3 + x)
  x ← Just 3
  return (x+y)
```

C
```
do
  return (x+y)
  x ← Just 3
  y ← Just (3 + x)
```

D
```
do
  return (x+y)
  x ← Just (3 + x)
  y ← Just 3
```

Quiz

10. What is the result of the following expression:

```
1. do
2.    x ← (+) <$> Just 4 <*> Nothing
3.    y ← (+1) <$> Just 3
4.    return (y+1)
```

A Nothing

B Just 3

C Just 4

D Just 5

E Error, as the types are incorrect in line 2

F Error, as the types are incorrect in line 3

G Error, as the types are incorrect in line 4

H I have no idea

Table of contents

IO Monad

- You have seen the do notation before in the week about IO
- That is because IO is also a Monad!
- You have worked with this, so I continue to the next Monad

State Monad I

- The State Monad can look very scary, but is very useful
- I will give a brief overview what you can do with a concrete example
- The State Monad keeps track of the current state of your program, see it as global variables
- E.g. the map from variable names to values in the previous assignment
- But instead of passing it around as argument, we keep it in a Monad!
- Note that the State Monad takes 2 extra types instead of one:
- State st rt, where:
  - st is the type of the state of the program (e.g. Map String Int, the map from variable names to values)
  - rt is the type of the end result of the program (e.g. Int, to integer value of the computation)

State Monad II

- State st rt, where:
  - st is the type of the state of the program (e.g. Map String Int, the map from variable names to values)
  - rt is the type of the end result of the program (e.g. Int, to integer value of the computation)
- There are 3 important functions for this State Monad: get, put and evalState
  - `x <- get` gets the state from the State Monad and puts it in x
    - x has type st! (type of the state)
    - It is like reading from a global variable
  - `put x` sets the state in the State Monad to x
    - x has type st! (type of the state)
    - It is like writing to a global variable
- `evalState p ss` evaluates the given program p (type State st rt) with the starting state ss (type rt)
- The next example clarifies these definitions!

Recap Monads

- The Monad is a class that focuses on "extracting" values outside containers
- ```
  class (Applicative m) => Monad m where
      return :: a -> m a
      (>>=) :: m a -> (a -> m b) -> m b
  ```
- With these Monads it is possible to "cheat" imperative programming into a functional language
- For Haskell, the introducted do-notation is very useful
- Practical Monads in Haskell are the IO and State Monads
  ```

Hierarchy I

- We have seen 3 classes: Functors, Applicatives and Monads
- Note that Applicatives are more powerful than Functors and Monads are even more powerful than Applicatives
- Intuitively, a Functor can only apply a function inside a container with the fmap
- But this is only possible for one container, while the Applicative can apply functions which take more arguments
- Note that the fmap can be expressed with the Applicative functions:
- ```
  -- fmap :: (a -> b) -> c a -> c b
  -- (<*>) :: c (a -> b) -> c a -> c b
  fmap f cx = pure f <*> cx
  ```
  ```

Hierarchy II

- Applicatives can apply a function on multiple arguments
- However, Applicatives can only curry a function and will evaluate it from left to right
- With the Monad you have more control in the evaluation sequence and the intermediate results
- Note that you can define the function of an Applicative with the Monad functions:
- ```
  pure x = return x
  -- (<*>) :: c (a -> b) -> c a -> c b
  cf <*> cx = do
      f <- cf
      x <- cx
      return (f x)
  ```
  ```

Hierarchy III

- Similar to other programming issues, problems can be solved by many different strategies
- There is always discussion which is the best option (which is also personal taste)
- This comes with experience
- An example would be the evaluator of last week, which can be done both by Monads or Applicatives
- Which style you prefer is completely personal, but probably the argument would be that Monads are a bit overkill for this problem
- Knowing that these tools are available to you is the most important

Conclusion

- What should you take away from these 2 weeks?
- For Haskell specifically, how to use Functors, Applicatives and Monads
- However, in a general sense, take away that by seeing patterns in data structures, it is possible to define classes to generalise code
- ```
  fmap :: (a -> b) -> c a -> c b
  (<*>) :: c (a -> b) -> c a -> c b
  (>>=) :: c a -> (a -> c b) -> c b
  ```
- Seeing and using these patterns to write more generalized code is probably the most important take away
  ```

Questionnaire

References

- A site that explains the basics of Functors, Applicatives and Monads in pictures: `https://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html`
- A site that explains Monads with easy-to-follow examples: `https://ishantheperson.github.io/posts/haskell-monads/`
- A great explanation of Monads in a bit more detail: `http://learnyouahaskell.com/a-fistful-of-monads`
- An explanation about the State Monad (including the discussed example): `https://wiki.haskell.org/State_Monad`