Functors and Applicatives

Tutorial FP week 11

Ruben Holubek

Radboud University Nijmegen

1st 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 most challenging topics of FP
- 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

- You have used the `map` function often in Haskell for lists
- ```
  map :: (a -> b) -> [a] -> [b]
  map _ [] = []
  map f (x:xs) = (f x):(map f xs)
  ```
- E.g. `map (+1) [1,2,3] = [2,3,4]`
- You can imagine such a map function for other datatypes as well
  ```

Introduction II

- For example, a binary tree
- `data Btree a = Tip a | Bin (Btree a) (Btree a)`
- Note that the a in `Btree a` is another type, e.g. `Int`
- A map function would look something like this:
- ```
  btMap :: (a -> b) -> (Btree a) -> (Btree b)
  btMap f (Tip e) = Tip (f e)
  btMap f (Bin l r) = Bin (btMap f l) (btMap f r)
  ```
- E.g. `BTMap (*3) (Bin (Tip 3) (Tip 4)) = (Bin (Tip 9) (Tip 12))`
  ```

Introduction III

- Do you see the similarities?

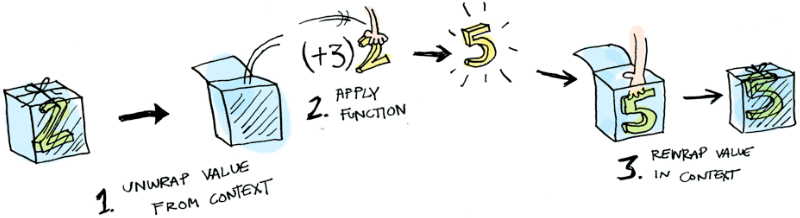
```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = (f x):(map f xs)
BTMap :: (a -> b) -> (Btree a) -> (Btree b)
BTMap f (Tip e) = Tip (f e)
BTMap f (Bin l r) = Bin (BTMap f l) (BTMap f r)
```

- Such a map function can be defined for many different datatypes
- For example, tuples, general trees, Maybe types...
- So a generic map function for different datatypes could be quite useful
- Which is the usage of a functor!

Functor definition

- A Functor is a class for containers which contains (among others) the fmap function
  - Where a container is a datatype containing other types, e.g. a list, tree, tuple, Maybe type etc..
  - Where fmap is the generic map function for the specified container.
- ```
  class Functor f where
      fmap :: (a -> b) -> f a -> f b
  ```
- So, to make from a specific container a Functor, we only have to define the fmap function
- Loosely saying, you are giving a container the label Functor
- Benefit is automatically getting the properties of a functor, e.g. extra functions!
  ```

Intuition


1. UNWRAP VALUE FROM CONTEXT
(+3) 2
2. APPLY FUNCTION
5
3. REWRAP VALUE IN CONTEXT

List example

- ```
  class Functor f where
      fmap :: (a -> b) -> f a -> f b
  ```
- Lists already have the map function, but this can also be specified with the fmap
- ```
  instance Functor [] where
      -- fmap :: (a -> b) -> [a] -> [b]
      fmap f [] = []
      fmap f (x:xs) = (f x):(fmap f xs)
  ```
- Note that
  - f is of type a → b
  - (x:xs) is of type [a], or List a
  - the result of op type [b], or List b
  - so, the type of this fmap is (a → b) → List a → List b
- This fmap could be used like:
  fmap (+1) [1,2,3,4] = [2,3,4,5]
  fmap ord ["a","b","c"] = [97,98,99]

Btree example

- ```
  class Functor f where
      fmap :: (a -> b) -> f a -> f b
  ```
- We can do something similar for the binary tree
- ```
  instance Functor Btree where
      -- fmap :: (a -> b) -> Btree a -> Btree b
      fmap f (Tip e) = Tip (f e)
      fmap f (Bin l r) = Bin (fmap f l) (fmap f r)
  ```
- Examples:
- fmap (*3) (Bin (Tip 3) (Tip 2)) = Bin (Tip 9) (Tip 6)
- fmap chr (Bin (Tip 97) (Tip 98)) = Bin (Tip "a") (Tip "b")

Maybe example

- Another useful example is the Maybe type:
- `data Maybe a = Just a | Nothing`
- `instance Functor Maybe where`

```
    -- fmap :: (a -> b) -> Maybe a -> Maybe b
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

- Which could be used like this:
- fmap (+1) Nothing = Nothing
- fmap (+1) (Just 3) = Just 4

Maybe continued

- Assume we have a function maybeInt, that sometimes returns an integer or else nothing
- If it returns something, we want to negate this number
- This can be solved by pattern matching:
- ```
  case (maybeInt) of
       (Just a) -> (Just (-a))
       Nothing -> Nothing
  ```
- Which is quite a lot of code, but this can be much simplified with the fmap:
- `fmap (*(-1)) maybeInt`
- The fmap already handles the case distinctions!
- The programmer can focus on the important code and all edge cases are covered automatically!
  ```

Other notes

- There is also an infix synonym for fmap, $(<\$>)$, which is also widely used
- $(+3) <\$> [1,2,3] = [3,4,5]$
- These are simply easier to write and to combine:
- $(+3) <\$> (*2) <\$> [1,2,3] = [5,7,9]$

Answers exercises

3. What would be the result of the following code:
fmap (+2) [1,2,3]

A [3,4,5]

B [1,2,3]

C [(+2),(+2),(+2)]

D Error, because the types are incorrect

Answers exercises

4. Suppose the following datatype:
data Tree2 a = Leaf | Node a (Tree2 a) (Tree2 a)
(a tree with elements as nodes, but no element in the tip)
What is the type of the fmap implementation?

- A fmap :: (a → a) → Tree2 a → Tree2 a
- B fmap :: (a → b) → a → Tree2 b
- C fmap :: (a → b) → Tree2 a → Tree2 b
- D fmap :: Tree2 (a → b) → Tree2 a → Tree2 b
- E fmap :: Tree2 (a → a) → Tree2 a → Tree2 a

Answers exercises

5. Suppose the following datatype:
data Tree2 a = Leaf | Node a (Tree2 a) (Tree2 a)
(a tree with elements as nodes, but no elements in the leafs)
What is the implementation of the fmap?

```
instance Functor Tree2 where
    -- fmap :: (a → b) → Tree2 a → Tree2 b
    fmap = ...
```

A fmap f Leaf = Leaf
fmap f (Node e lt rt) = Node (f e) (fmap f lt) (fmap f rt)

B fmap f Leaf = Leaf
fmap f (Node e lt rt) = Node (fmap f e) (fmap f lt) (fmap f rt)

C fmap f Leaf = Leaf
fmap f (Node e lt rt) = Node (f e) (f lt) (f rt)

D fmap f Leaf = Leaf
fmap f (Node e lt rt) = Node (fmap f e) (f lt) (f rt)

Summary Functor

- A Functor is a class for containers which contains the fmap function
- ```
  class Functor f where
      fmap :: (a -> b) -> f a -> f b
  ```
- Defining the fmap for a container, and thus making it a Functor has benefits:
- You do not have to write out case distinctions every time
- It makes your code more readable and editable
- It is also needed for the following class: Applicatives
  ```

Table of contents

Introduction I

- Suppose we want to add up two Maybe Int types
- E.g. (Just 3) + (Just 4) = Just 7
- We could do that with the following code:

```
addMaybe :: (Maybe Int) -> (Maybe Int) -> Maybe Int
addMaybe Nothing _ = Nothing
addMaybe _ Nothing = Nothing
addMaybe (Just x) (Just y) = Just (x+y)
```

- This is a lot of code for such a simple operation
- Note that Maybe is a Functor, so can't we use that fmap?
- That would automatically take care of the case distinctions!

Introduction II

- Lets try the following code:

```haskell
addMaybe :: (Maybe Int) -> (Maybe Int) -> Maybe Int
addMaybe x y = fmap (+) x y
```

- However, this gives an error in Haskell...
- Lets take a closer look at the types:
-
```haskell
fmap :: (a -> b) -> Maybe a -> Maybe b
(+) :: Int -> (Int -> Int)
fmap (+) :: Maybe Int -> Maybe (Int -> Int)
x :: Maybe Int
fmap (+) x :: Maybe (Int -> Int)
y :: Maybe Int
```

- It makes sense; we cannot access the function right now!
- But we can solve this!

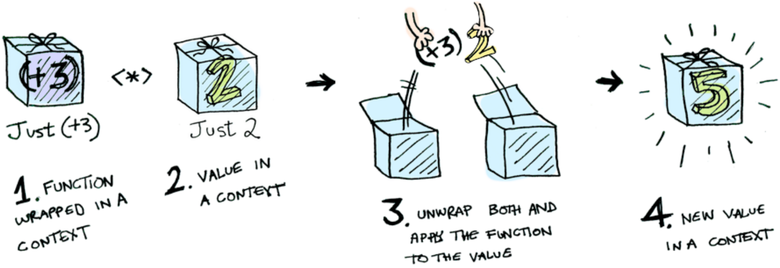
Introduction III

- ```haskell
  -- fmap (+) x :: Maybe (Int -> Int)
  -- y :: Maybe Int
  addMaybe :: (Maybe Int) -> (Maybe Int) -> Maybe Int
  addMaybe x y =
      case (fmap (+) x) of
          Nothing -> Nothing
          (Just f) ->
              case y of
                  Nothing -> Nothing
                  (Just i) -> Just(f i)
  ```
- Please note that we can shorten the second case a lot!
  ```

Introduction IV

- ```
  -- fmap (+) x :: Maybe (Int -> Int)
  -- y :: Maybe Int
  addMaybe :: (Maybe Int) -> (Maybe Int) -> Maybe Int
  addMaybe x y =
      case (fmap (+) x) of
          Nothing -> Nothing
          (Just f) -> fmap f y
  ```
- This looks already a lot better!
- However, there should be an easier way to handle this function inside a container, right?
- Especially if you have a function on even more arguments
- The solution for this is the **Applicative**!
  ```

Applicative definition I

- An Applicative is a class for containers which contains (among others) the pure and <*> function:
-
```haskell
1. infixl 4 <*>
2. class (Functor f) => Applicative f where
3.     pure :: a -> f a
4.     (<*>) :: f (a -> b) -> f a -> f b
```
- Lets investigate every line here:
  1. This simply states that the <*> symbol is left-associative and has precedence value 4
  2. Here, `(Functor f) =>` states that an Applicative should also be a Functor
     - So you can automatically use the fmap in your Applicative definition!
  3. pure takes a value and puts it in the desired container
  4. <*> (apply) takes a function in a container, an element in a container, applies these things to each other and returns a new element in the container

Intuition


Just (+3)
<*>
Just 2
1. FUNCTION WRAPPED IN A CONTEXT
2. VALUE IN A CONTEXT
3. UNWRAP BOTH AND APPLY THE FUNCTION TO THE VALUE
4. NEW VALUE IN A CONTEXT

Applicative definition II

- ```haskell
  class (Functor f) => Applicative f where
        pure :: a -> f a
        (<*>) :: f (a -> b) -> f a -> f b
  ```
- Pure can be seen as putting an element into a "default" container
  - E.g. for lists, pure 3 = [3]
  - For Maybe, pure "a" = Just "a"
- Maybe <*> looks a bit familiar?
  - ```haskell
    (<*>) :: f (a -> b) -> f a -> f b
    fmap :: (a -> b) -> f a -> f b
    ```
  - It is exactly the fmap, except the function is put in a container!
- But why would we want this? Isn't it more convenient to have the function directly?
- Well... Lets take a look at a few examples

Maybe example I

- ```
  class (Functor f) => Applicative f where
      pure :: a -> f a
      (<*>) :: f (a -> b) -> f a -> f b
  ```
- Suppose we want to define the Applicative for Maybe:
- ```
  instance Applicative Maybe where
      -- pure :: a -> Maybe a
      pure x = Just x
      -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
      Nothing <*> _ = Nothing
      _ <*> Nothing = Nothing
      (Just f) <*> (Just x) = Just (f x)
  ```
- The pure function simply wraps the value inside a Just, indeed what you would expect

Maybe example II

- ```haskell
  instance Applicative Maybe where
      -- pure :: a -> Maybe a
      pure x = Just x
      -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
      Nothing <*> _ = Nothing
      _ <*> Nothing = Nothing
      (Just f) <*> (Just x) = Just (f x)
  ```
- The <*> function has a few cases:
  - If the function in the Maybe container is Nothing, then return Nothing
  - If the value in the Maybe container is Nothing, then return Nothing as well
  - If both the function and value are Just constructs, then unwrap these, apply them to each other and wrap them again with Just
    - Note that pure would have worked here as well!
- However, this is not the official definition, which uses fmap
  ```

Maybe example III

- ```
  instance Applicative Maybe where
        -- pure :: a -> Maybe a
        pure x = Just x
        -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
        Nothing <*> _ = Nothing
        _ <*> Nothing = Nothing
        (Just f) <*> (Just x) = Just (f x)
  ```
- Using the fmap, we get the same result, but with cleaner code:
- ```
  instance Applicative Maybe where
        -- pure :: a -> Maybe a
        pure x = Just x
        -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
        Nothing <*> _ = Nothing
        (Just f) <*> x = fmap f x
  ```
- See for yourself that this is indeed the same!

Maybe example IV

- A few examples:
- pure 2 = Just 2
- Just (+1) <*> Just 3 = Just 4
- Often pure is used for the formula:
- pure (+1) <*> Just 3 = Just 4
- pure (+1) <*> Nothing = Nothing
- However, Applicatives really shine when used on more arguments:
- pure (+) <*> Just 1 <*> Just 2 = Just 3
- pure (+) <*> Nothing <*> Just 2 = Nothing
- pure (\a b c →a+b-c) <*> Just 1 <*> Just 2 <*> Just 3 = Just 0
- After implementing Applicative, we can write really clean code!

Answers exercises

6. What would be the result of the following code:
pure (+) <*> Just 7 <*> Nothing

- **A** Just 7
- **B** Nothing
- **C** Just 8
- **D** Error, because the types are incorrect

Answers exercises

7. What would be the result of the following code:
`(++) <*> Just "F" <*> Just "P"`

A Just "FP"

B Just "PF"

C Nothing

D Error, because the types are incorrect

Answers exercises

8. What would be the result of the following code (ignoring the printing problems):
Just (+) <*> Just 2

- **A** Just 2
- **B** Just (+2)
- **C** Nothing
- **D** Error, because the types are incorrect

Applicative style I

- First, lets take a closer look at what is happening here:
- pure (+) <*> Just 1 <*> Just 2
- <*> is left associative, so the brackets are like this:
- (pure (+) <*> Just 1) <*> Just 2
- (pure (+) <*> Just 1) is simply (Just (+) <*> Just 1), which returns the partial function Just(1+) (remember currying?)
- Exactly a function inside a container, which we expect in the Applicative!
- Just(1+) <*> Just 2 then evaluates to Just 3
- This is why we can use functions that need more arguments, which is not possible with fmap!
- All by using the Applicative style: pure f <*> x <*> y <*> z <*> ...

Applicative style II

- Please note the following (this holds in general!):
- `pure f <*> x = fmap f x`
- With `<*>`, we first wrap the function, but with fmap we apply it immediately, which is exactly the same
- Recall that `fmap f x = f <$> x`
- But that means we can do this:
- `pure f <*> x <*> y <*> ... =`
- `((pure f <*> x) <*> y) <*> ... =`
- `fmap f x <*> y <*> ... =`
- `f <$> x <*> y <*> ...` !
- This looks really clean!

Applicative style III

- Recall how to concatenate 2 strings in Haskell:
- (++) "Hello " "World!" = "Hello World!"
- Now suppose that the strings are in Maybe types, so that complicates things right? (No!)
- (++) <$> (Just "Hello ") <*> (Just "World!")
- We can completely handle the problems with Maybe types by using some <$> and <*>!
- Recall the stated problem of addMaybe that adds 2 Maybe Ints:
- addMaybe x y = (+) <$> x <*> y
- This looks really clean!

Applicative style IV

- So the general (informal) idea is to
  1. First write down the function
  2. Then an fmap on the first element (<$>)
  3. Then all the arguments combined with <*>
- E.g. (\a b c →a+b-c) <$> Just 1 <*> Just 2 <*> Just 3 = Just 0
- Ofcourse, feel free to simply use pure in the exercises combined with <*>
- But once used to this notation, you do not want to go back

List example I

- Lists are also an instance of an Applicative:
-
```
instance Applicative [] where
    -- pure :: a -> [a]
    pure x = [x]
    -- (<*>) :: [a -> b] -> [a] -> [b]
    fs <*> xs = [f x | f <- fs, x <- xs]
```
- The pure function simply puts the element in a singleton list
- The $<*>$ gets a list of functions and a list of elements and produces a list where all functions in fs are applied on all elements of xs
- It can be seen as 2 for loops where the outer loop loops over functions and the inner loop over elements
- This definition automatically handles the case where the lists are not of the same length or if one of lists is empty

List example II

- ```
  instance Applicative [] where
      pure x = [x]
      fs <*> xs = [f x | f <- fs, x <- xs]
  ```
- A few examples:
- pure chr = [chr]
- pure chr <*> [97,98] = ["a","b"]
- [(+1),(+3)] <*> [1,2] = [2,3,4,5]
- [(+),(*)] <*> [1,2] <*> [3,4] =
- [(1+),(2+),(1*),(2*)] <*> [3,4] =
- [4,5,5,6,3,4,6,8]
  ```

IO example I

- IO is also in instance of an Applicative:
-
```haskell
instance Applicative IO where
    -- pure :: a -> IO a
    pure = return
    -- (<*>) :: IO (a -> b) -> IO a -> IO b
    a <*> b = do
        f <- a
        x <- b
        return (f x)
```
- pure is the return function, as this simply puts the element in an IO object
- <*> first extracts the function f from the IO container, then it extracts the element x and returns f applied on x in an IO container
- The IO Applicative is useful for sequences of actions

IO example II

- Consider the following function:
- ```haskell
  myAction :: IO String
  myAction = do
        a <- getLine
        b <- getLine
        return (a ++ b)
  ```
- First, the function gets a line from the input, the another line, and concatenates these.
- This can also be written with an Applicative:
- (++) <$> getLine <*> getLine
- The informal idea is that the IO actions are performed in sequence, and the results are combined with the function.
  ```

Answers exercises

9. Suppose we want to add 2 Maybe integers, as in the slides.
Which of the following code fragments is an **incorrect** way to do this?

- **A** addMaybe x y = pure (+) <*> x <*> y
- **B** addMaybe x y = Just (+) <*> x <*> y
- **C** addMaybe x y = pure (+) <$> x <*> y
- **D** addMaybe x y = (+) <$> x <*> y

Answers exercises

10. What would be the result of the following code:
`[(+1),(*2)] <*> [2,4]`

A [3,8]

B [3,5,4,8]

C [3,4,5,8]

D Error, because the types are incorrect

Summary Applicative

- An Applicative is a class for containers which contains the pure and <*> (apply) function
-
```haskell
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```
- In contrast to a Functor, an Applicative can apply a function to more arguments using the Applicative style:
- pure f <*> x <*> y <*> ...
- Alternatively, f <$> x <*> y <*> ...

References

- Honestly, all the things you learn these weeks, I use them regularly in (programming) courses
- I took images and inspiration from these two sources; if you still struggle with the topic (also monads next week), I would suggest to check these out:
  - http://learnyouahaskell.com/functors-applicative-functors-and-monoids#applicative-functors
  - https://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html