Functors, Applicative Functors

Recalls

Mathematics

Mathematical Concept of Conservation of Structure.

The notion of structure refers to algebraic structures such as groups, monoids, etc.

A group morphism or group homomorphism is a mapping between two groups that respects/conserves the group structure.

Let (E, ⊤) and (F, ⊥) two groups. Let *f* and application from E to F.

f is a group morphism if and only if:

$\forall\, x,\ y\ \in\ E,\ f(x \top y)\ =\ f(x)\ \perp\ f(y)$

Example:

Let consider ℂ the set of complex numbers and ℂ* = ℂ\{0} the set of complex numbers without 0. (ℂ, +) is a group and (ℂ*, ×) is also a group.

We define the application *f* from ℂ to ℂ* as :

$f(z)\ =\ e^{z}$ *f* thus defined is a group morphism

Property:

Let *f* a group morphism from (E, ⊤) to (F, ⊥). Let e the neutral element of (E, ⊤) and e' the neutral element of (F, ⊥). then:

- $f(e)\ =\ e'$
- $Let\ x\ \in\ E\ \ sx\ the\ symetric\ of\ x\ in\ (E,\ \top),\ y\ the\ image\ of\ x\ from\ f\ and\ sy\ the\ symetric\ of\ y\ in\ (F,\ \ \perp)\ \ thn\ \ sy\ =\ f(sx)$

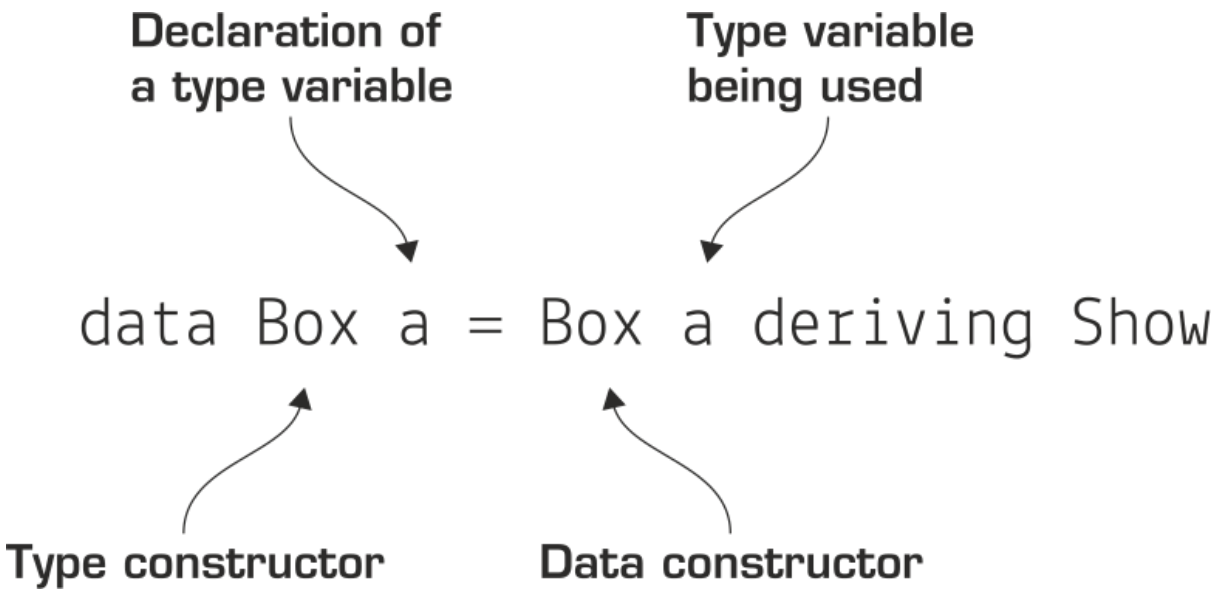
wada

Parameterized types

In previous chapters so far, we've discussed how types can be added and multiplied, like data. Like functions, types can also take arguments. Types take arguments by using type variables in their definitions (so their arguments are always other types). Types defined using parameters are called parameterized types. Parameterized types perform an important role in Haskell, as they allow you to define generic data structures that work with a wide range of existing data. When a type takes à parameter, it works as à function. Its inputs are types and the outputs are type. Type constructor becomes the function.

Example

```
data Maybe a = Nothing | Just a
data List a = EmptyList | Cons a (List a)
data Either a b = Left a | Right b
data BinaryTree a  = Leaf | Node a (BinaryTree a)
(BinaryTree a)
```



wada

Kind

Maybe, List, Either, BinaryTree are parameterized type constructor. They construct new types from other types. like functions, we can determine their types. In this case we are talking about the type of a type. This is called the kind of a type. In Haskell this can be known by running the command `:kind TypeName or :k TypeName`. When a type does not take any parameter, it is called concrete type. Conversely, when a type takes parameters, it is called parameterized type or abstract type or polymorphic type.

Map

We saw in Higher Order functions the map function on list ([]). I is defined as follow:

```
map :: (a -> b) -> [a] -> [b]
```

[a] is a concrete type obtained from type constructor [] giving it the type a as an argument.
[b] is a concrete type obtained from type constructor [] giving it the type b as an argument.
These types are obtained using the same type constructor.
How can we generalize this concept of map to other type constructors like BinaryTree, Maybe, Either a, etc.? (parameterized types that take only one parameter as input)
This is where Functors come in place.

Typeclasses are open, which means that we can define our own data type, think about what it can act like and connect it with the typeclasses that define its behaviors. Because of that and because of Haskell's great type system that allows us to know a lot about a function just by knowing its type declaration, we can define typeclasses that define behavior that's very general and abstract.

wada

Functors

Question: How can we define a function as a map over BinaryTree? e.i

```
mymap :: (a -> b) -> (BinaryTree a) -> (BinaryTree b)
```

Solution:

```
mymap _ Leaf = Leaf
mymap f (Node x left right) = Node (f x) (mymap f left)
(mymap f right)
```

The concept behind map is generalized in the typeclass Functor.

Definition
A functor is a way to apply a function over or around some structure that we don't want to alter (structure conservation). That is, we want to apply the function to the value that is "inside" some structure and leave the structure alone.

In Haskell, they're described by the typeclass Functor, which has only one typeclass method, namely fmap, which has a type of
fmap :: (a -> b) -> f a -> f b.

```
class Functor f where
   fmap :: (a -> b) -> f a -> f b
```

Note that in this typeclass declaration f is a type constructor which the kind is * -> *

wada

Usage

```
Prelude> map (\x -> x > 3) [1..6]
[False,False,False,True,True,True]
Prelude> fmap (\x -> x > 3) [1..6]
[False,False,False,True,True,True]
```

Some instances of Functor typeclass

IO

```
instance Functor IO where
   fmap f action = do
      result <- action
      return (f result)
```

BinaryTree

```
instance Functor BinaryTree where
   fmap _ Leaf = Leaf
   fmap f (Node x left right) = Node (f x) (mymap f left)
(mymap f right)
```

Functions type Constructor (->) r

(->) r is a type constructor, when is given a type a as input, return the type of function r -> a (type of function from r to a)
(->) is a type constructor that takes two parameters as input. (->) r is (->) type constructor partially applied.

```
instance Functor ((->) r) where
   fmap f g = (\x -> f (g x))
```

wada

Exercise:
Make Maybe, [], and Either x instances of Functor

Functor laws

In order for something to be a functor, it should satisfy some laws. All functors are expected to exhibit certain kinds of functor-like properties and behaviors. They should reliably behave as things that can be mapped over. Calling fmap on a functor should just map a function over the functor, nothing more. This behavior is described in the functor laws. There are two of them that all instances of Functor should abide by. They aren't enforced by Haskell automatically, so we have to test them out ourselves.

Identity

The first functor law states that if we map the id function over a functor, the functor that we get back should be the same as the original functor.

```
fmap id = id
```

Composition conservation

The second law says that composing two functions and then mapping the resulting function over a functor should be the same as first mapping one function over the functor and then mapping the other one. Formally written, that means that:

```
fmap (f . g) = fmap f . fmap g
Or
fmap (f . g) F = fmap f (fmap g F)
```

wada

Applicative functors

As you know, functions in Haskell are curried by default, which means that a function that seems to take several parameters actually takes just one parameter and returns a function that takes the next parameter and so on. If a function is of type a -> b -> c, we usually say that it takes two parameters and returns a c, but actually it takes an a and returns a function b -> c. That's why we can call a function as f x y or as (f x) y. This mechanism is what enables us to partially apply functions by just calling them with too few parameters, which results in functions that we can then pass on to other functions.
So far, when we were mapping functions over functors, we usually mapped functions that take only one parameter. But what happens when we map a function like a->b ->c, which takes two parameters, over a functor?
Let's take a look at a couple of concrete examples of this.

```
ghci> :t fmap (++) [1, 2, 3]
fmap (++) [1, 2, 3] :: Num [a] => [[a] -> [a]]
ghci> :t fmap compare "A LIST OF CHARS"
fmap compare "A LIST OF CHARS" :: [Char -> Ordering]
ghci> :t fmap (\x y z -> x + y / z) [3,4,5,6]
fmap (\x y z -> x + y / z) [3,4,5,6] :: (Fractional a) =>
[a -> a -> a]
ghci> :t fmap (++) (Just "hey")
fmap (++) (Just "hey") :: Maybe ([Char] -> [Char])
```

If we map compare, which has a type of (Ord a) => a -> a -> Ordering over a list of characters, we get a list of functions of type Char -> Ordering, because the function compare gets partially applied with the characters in the list. It's not a list of (Ord a) => a -> Ordering function,

because the first a that got applied was a Char and so the second a has to decide to be of type Char.
We see how by mapping "multi-parameter" functions over functors, we get functors that contain functions inside them. So now what can we do with them? Well for one, we can map functions that take these functions as parameters over them, because whatever is inside a functor will be given to the function that we're mapping over it as a parameter.

```
ghci> let a = fmap (*) [1,2,3,4]
ghci> :t a
a :: [Integer -> Integer]
ghci> fmap (\f -> f 9) a
[9,18,27,36]
```

But what if we have a functor value of Just (3 *) and a functor value of Just 5 and we want to take out the function from Just (3 *) and map it over Just 5? With normal functors, we're out of luck, because all they support is just mapping normal functions over existing functors. Even when we mapped \f -> f 9 over a functor that contained functions inside it, we were just mapping a normal function over it. But we can't map a function that's inside a functor over another functor with what fmap offers us. We could pattern-match against the Just constructor to get the function out of it and then map it over Just 5, but we're looking for a more general and abstract way of doing that, which works across functors.
Meet the Applicative typeclass. It lies in the Control.Applicative module and it defines two methods, pure and <*>. It doesn't provide a default implementation for any of them, so we have to define them both if we want something to be an applicative functor. The class is defined like so:

wada

```haskell
class (Functor f) => Applicative f where
   pure :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b
```

This simple three line class definition tells us a lot! Let's start at the first line. It starts the definition of the Applicative class and it also introduces a class constraint. It says that if we want to make a type constructor part of the Applicative typeclass, it has to be in Functor first. That's why if we know that if a type constructor is part of the Applicative typeclass, it's also in Functor, so we can use fmap on it.
The first method it defines is called pure. Its type declaration is pure :: a -> f a. f plays the role of our applicative functor instance here. Because Haskell has a very good type system and because everything a function can do is take some parameters and return some value, we can tell a lot from a type declaration and this is no exception. pure should take a value of any type and return an applicative functor with that value. But the a -> f a type declaration is still pretty descriptive. We take a value and we wrap it in an applicative functor that has that value as the result inside it. A better way of thinking about pure would be to say that it takes a value and puts it in some sort of default (or pure) context—a minimal context that still yields that value.

The <*> function is really interesting. It has a type declaration of f (a -> b) -> f a -> f b. Does this remind you of anything? Of course, fmap : : (a -> b) -> f a -> f b. It's a sort of a beefed up fmap. Whereas fmap takes a function and a functor and applies the function inside the functor, <*> takes a functor that has a function in it and another functor and sort of extracts that function from the first functor and then maps it over the second one. When I say extract, I actually sort of mean run and then extract, maybe even sequence. We'll see why soon.
Let's take a look at the Applicative instance implementation for Maybe.

wada

```haskell
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
```

So for Maybe, <*> extracts the function from the left value if it's a Just and maps it over the right value. If any of the parameters is Nothing, Nothing is the result.

```haskell
ghci> Just (+3) <*> Just 9
Just 12
ghci> pure (+3) <*> Just 10
Just 13
ghci> pure (+3) <*> Just 9
Just 12
ghci> Just (++"hahah") <*> Nothing
Nothing
ghci> Nothing <*> Just "woot"
Nothing
```

What's going on here? Let's take a look, step by step. <*> is left-associative, which means that pure (+) <*> Just 3 <*> Just 5 is the same as (pure (+) <*> Just 3) <*> Just 5. First, the + function is put in a functor, which is in this case a Maybe value that contains the function. So at first, we have pure (+) , which is Just (+) . Next, Just (+) <*> Just 3 happens. The result of this is Just (3+) . This is because of partial application. Only applying 3 to the + function results in a function that takes one parameter and adds 3 to it. Finally, Just (3+) <*> Just 5 is carried out, which results in a Just 8.

wada

Function (<$>)

Applicative functors and the applicative style of doing pure f <*> x <*> y <*> allow us to take a function that expects parameters that aren't necessarily wrapped in functors and use that function to operate on several values that are in functor contexts. The function can take as many parameters as we want, because it's always partially applied step by step between occurrences of <*>.
This becomes even more handy and apparent if we consider the fact that pure f <*> x equals fmap f x. This is one of the applicative laws. We'll take a closer look at them later, but for now, we can sort of intuitively see that this is so. Think about it, it makes sense. Like we said before, pure puts a value in a default context. If we just put a function in a default context and then extract and apply it to a value inside another applicative functor, we did the same as just mapping that function over that applicative functor. Instead of writing pure f <*> x <*> y <*> ..., we can write fmap f x <*> y <*> .... This is why Control.Applicative exports a function called <$>, which is just fmap as an infix operator. Here's how it's defined:

```
(<$>) :: (Functor F) => (a -> b) -> F a -> F b
f <$> x = fmap f x
```

Let's take a closer look at how this works. We have a value of Just "johntra" and a value of Just "volta" and we want to join them into one String inside a Maybe functor. We do this:

```
ghci> (++) <$> Just "johntra" <*> Just "volta"
Just "johntravolta"
```

Before we see how this happens, compare the above line with this:

```
ghci> (++) "johntra" "volta"
"johntravolta"
```

To use a normal function on applicative functors, just sprinkle some <$> and <*> about and the function will operate on applicatives and return an applicative. How cool is that? Anyway, when we do (++) <$> Just "johntra" <*> Just "volta", first (++), which has a type of (++) : : [a] -> [a] -> [a] gets mapped over Just "johntra", resulting in a value that's the same as Just ("johntra"++) and has a type of Maybe ([Char] -> [Char]) . Notice how the first parameter of (++) got eaten up and how the as turned into Chars. And now Just ("johntra"++) <*> Just "volta" happens, which takes the function out of the Just and maps it over Just "volta", resulting in Just "johntravolta". Had any of the two values been Nothing, the result would have also been Nothing.

The list type constructor []

Lists (actually the list type constructor, []) are applicative functors. What a surprise! Here's how [] is an instance of Applicative:

```
instance Applicative [] where
   pure x = [x]
   fs <*> xs = [f x | f <- fs, x <- xs]
```

Earlier, we said that pure takes a value and puts it in a default context. Or in other words, a minimal context that still yields that value. The minimal context for lists would be the empty list, [ ] , but the empty list represents the lack of a value, so it can't hold in itself the value that we used pure on. That's why pure takes a value and puts it in a singleton list. Similarly, the minimal context for the Maybe applicative functor would be a Nothing, but it represents the lack of a value instead of a value, so pure is implemented as Just in the instance implementation for Maybe.

```
ghci> pure "Hey" :: [String]
```

wada

```
["Hey"]
ghci> pure "Hey" :: Maybe String
Just "Hey"
```

What about <*>? If we look at what <*>'s type would be if it were limited only to lists, we get
(<*>) :: [a -> b] -> [a] -> [b]. It's implemented with a list comprehension. <*> has to somehow extract the function out of its left parameter and then map it over the right parameter. But the thing here is that the left list can have zero functions, one function, or several functions inside it.
The right list can also hold several values. That's why we use a list comprehension to draw from both lists. We apply every possible function from the left list to every possible value from the right list. The resulting list has every possible combination of applying a function from the left list to a value in the right one.

```
ghci> [(*0),(+100),(^2)] <*> [1,2,3]
[0,0,0,101,102,103,1,4,9]
```

The left list has three functions and the right list has three values, so the resulting list will have nine elements. Every function in the left list is applied to every function in the right one. If we have a list of functions that take two parameters, we can apply those functions between two lists.

```
ghci> [(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
```

```
ghci> (++) <$> ["ha","heh","hmm"] <*> ["?","!","."]
["ha?","ha!","ha.","heh?","heh!","heh.","hmm?","hmm!","hm
m."]
```

Again, see how we used a normal function that takes two strings between two applicative functors of strings just by inserting the appropriate applicative operators.

List as Non-Deterministic

You can view lists as non-deterministic computations. A value like 100 or "what" can be viewed as a deterministic computation that has only one result, whereas a list like [1,2,3] can be viewed as a computation that can't decide on which result it wants to have, so it presents us with all of the possible results. So when you do something like (+) <$> [1,2,3] <*> [4,5,6] , you can think of it as adding together two non-deterministic computations with +, only to produce another non-deterministic computation that's even less sure about its result.

Using the applicative style on lists is often a good replacement for list comprehensions. In the à previous chapter, we wanted to see all the possible products of [2,5,10] and [8,10,11], so we did this:

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

We're just drawing from two lists and applying a function between every combination of elements. This can be done in the applicative style as well:

```
ghci> (*) <$> [2,5,10] <*> [8,10,11]
[16,20,22,40,50,55,80,100,110]
```

This seems clearer to me, because it's easier to see that we're just calling * between two non-deterministic computations. If we wanted all possible products of those two lists that are more than 50, we'd just do:

```
ghci> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
[55,80,100,110]
```

wada

IO is applicative functor

Another instance of Applicative that we've already encountered is IO. This is how the instance is implemented:

```
instance Applicative IO where
   pure = return
   a <*> b = do
      f <- a
      x <- b
      return (f x)
```

(->) r is an applicative functor

```
instance Applicative ((->) r) where
pure x = (\_ -> x)
f <*> g = \x -> f x (g x)
```

```
ghci> (pure 3) "blah"
3
```

Because of currying, function application is left-associative, so we can omit the parentheses.

```
ghci> pure 3 "blah"
3
```

The instance implementation for <*> is a bit cryptic, so it's best if we just take a look at how to use functions as applicative functors in the applicative style.

```
ghci> :t (+) <$> (+3) <*> (*100)
(+) <$> (+3) <*> (*100) :: (Num a) => a -> a
ghci> (+) <$> (+3) <*> (*100) $ 5
508
```

wada

Calling <*> with two applicative functors results in an applicative functor, so if we use it on two functions, we get back a function. So what goes on here? When we do `(+) <$> (+3) <*> (*100)`,
we're making a function that will use + on the results of (+3) and (*100) and return that. To demonstrate on a real example, when we did (+) <$> (+3) <*> (*100) $ 5, the 5 first got applied to (+3) and (*100), resulting in 8 and 500. Then, + gets called with 8 and 500, resulting in 508.

```
ghci> (\x y z -> [x,y,z]) <$> (+3) <*> (*2) <*> (/2) $ 5
[8.0,10.0,2.5]
```

liftA2

Control.Applicative defines a function that's called liftA2, which has a type of liftA2 : (Applicative f) => (a -> b -> c) -> f a -> f b -> f c . It's defined like this:

```
liftA2 :: (Applicative F) => (a -> b -> c) -> F a -> F b
-> F c
liftA2 f a b = f <$> a <*> b
```

```
ghci> liftA2 (:) (Just 3) (Just [4])
Just [3,4]
ghci> (:) <$> Just 3 <*> Just [4]
Just [3,4]
```

sequenceA

Remember, (:) is a function that takes an element and a list and returns a new list with that element at the beginning. Now that we have Just [3,4], could we combine that with Just 2 to produce Just [2,3,4]? Of course we could. It seems that we can combine any amount of

wada

applicatives into one applicative that has a list of the results of those applicatives inside it. Let's try implementing a function that takes a list of applicatives and returns an applicative that has a list as its result value. We'll call it sequenceA.

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA [] = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs
```

First, we look at the type. It will transform a list of applicatives into an applicative with a list. From that, we can lay some groundwork for an edge condition. If we want to turn an empty list into an applicative with a list of results, well, we just put an empty list in a default context. Now comes the recursion. If we have a list with a head and a tail (remember, x is an applicative and xs is a list of them), we call sequenceA on the tail, which results in an applicative with a list. Then, we just prepend the value inside the applicative x into that applicative with a list.
So if we do sequenceA [Just 1, Just 2], that's (:) <$> Just 1 <*> sequenceA [Just 2] . That equals (:)
<$> Just 1 <*>((:) <$> Just 2 <*> sequenceA[]). We know that sequenceA[] ends up as being Just [], so this expression is now (:) <$> Just 1 <*> ((:) <$> Just 2 <*> Just []), which is (:) <$> Just 1 <*> Just [2], which is Just [1,2]!
Another way to implement sequenceA is with a fold. Remember, pretty much any function where we go over a list element by element and accumulate a result along the way can be implemented with a fold.

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA = foldr (liftA2 (:)) (pure [])
```

Example

```
ghci> sequenceA [Just 3, Just 2, Just 1]
Just [3,2,1]
```

wada

```
ghci> sequenceA [Just 3, Nothing, Just 1]
Nothing
ghci> sequenceA [(+3),(+2),(+1)] 3
[6,5,4]
ghci> sequenceA [[1,2,3],[4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
ghci> sequenceA [[1,2,3],[4,5,6],[3,4,4],[]]
[]
```

Note:
When used on Maybe values, sequenceA creates a Maybe value with all the results inside it as a list. If one of the values was Nothing, then the result is also a Nothing. This is cool when you have a list of Maybe values and you're interested in the values only if none of them is a Nothing.
Using sequenceA is cool when we have a list of functions and we want to feed the same input to all of them and then view the list of results.

For instance, we have a number and we're wondering whether it satisfies all of the predicates in a list. One way to do that would be like so:

```
ghci> map (\f -> f 7) [(>4),(<10),odd]
[True,True,True]
ghci> and $ map (\f -> f 7) [(>4),(<10),odd]
True
```

When used with [], sequenceA takes a list of lists and returns a list of lists. Hmm, interesting. It actually creates lists that have all possible combinations of their elements. For illustration, here's

the above done with sequenceA and then done with a list comprehension:

```
ghci> sequenceA [[1,2,3],[4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
ghci> [[x,y] | x <- [1,2,3], y <- [4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
ghci> sequenceA [[1,2],[3,4]]
[[1,3],[1,4],[2,3],[2,4]]
ghci> [[x,y] | x <- [1,2], y <- [3,4]]
[[1,3],[1,4],[2,3],[2,4]]
ghci> sequenceA [[1,2],[3,4],[5,6]]
[[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],
[2,4,6]]
ghci> [[x,y,z] | x <- [1,2], y <- [3,4], z <- [5,6]]
[[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],
[2,4,6]]
```

Applicative functors laws

All Applicative functors must respect the following laws:

- **pure f <*> x = fmap f x**
- **pure id <*> v = v**
- **pure (.) <*> u <*> v <*> w = u <*> (v <*> w)**
- **pure f <*> pure x = pure (f x)**
- **u <*> pure y = pure ($ y) <*> u**

Ref:

- http://learnyouahaskell.com/functors-applicative-functors-and-monoids

wada

- https://mmhaskell.com/monads/functors
-