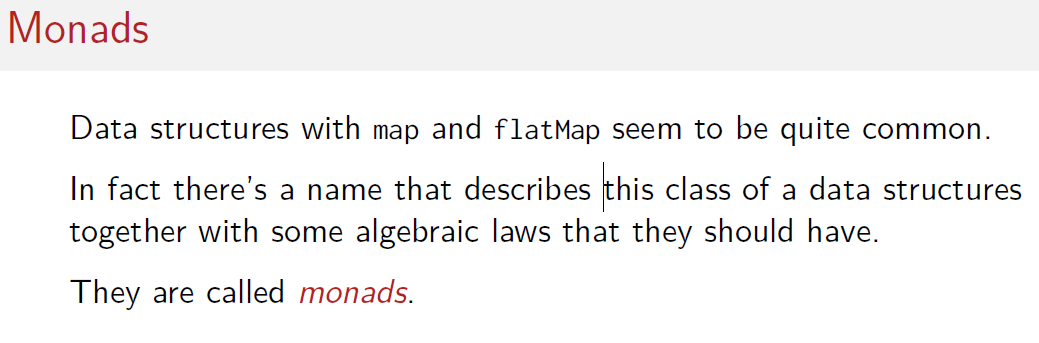
In the final unit of this week, we'll cover some of the general theory that underlies what we have seen before.

We're going to study a very general class of design patterns that come up almost everywhere in functional programming and in reactive programming.

The theoretical foundation of this class of design patterns, is embodied in the concept of a monad.

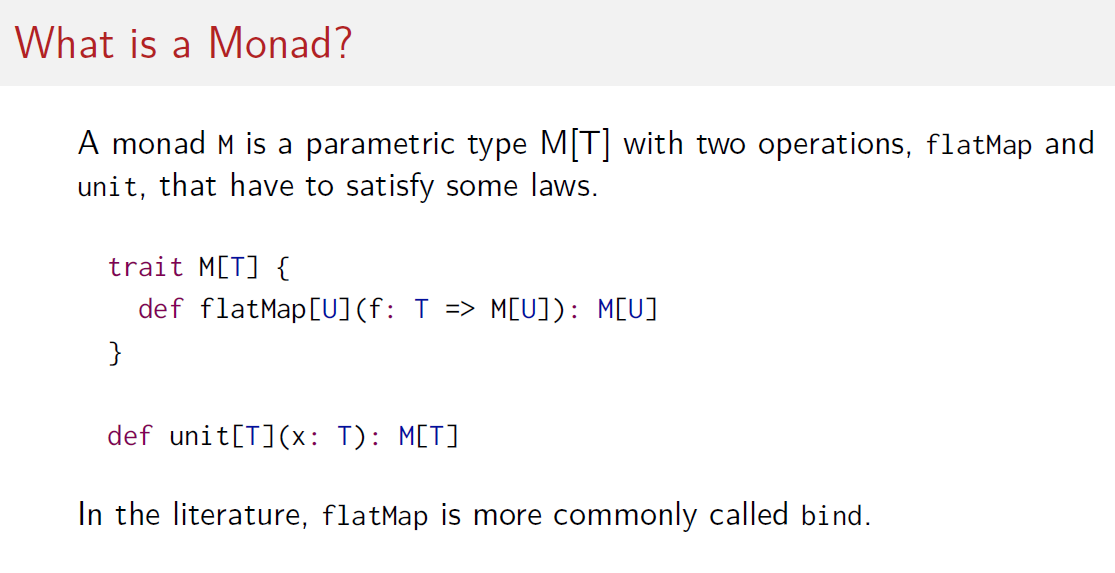
As we've seen, data structures with map and flatMap, are quite common. And in fact, there's a name that describes this class of data structure, together with some algebraic laws, that they should have.

These data structures are called monads



A monad is a parametric type M with a parameter type T that has two operations:

* flatMap
* unit



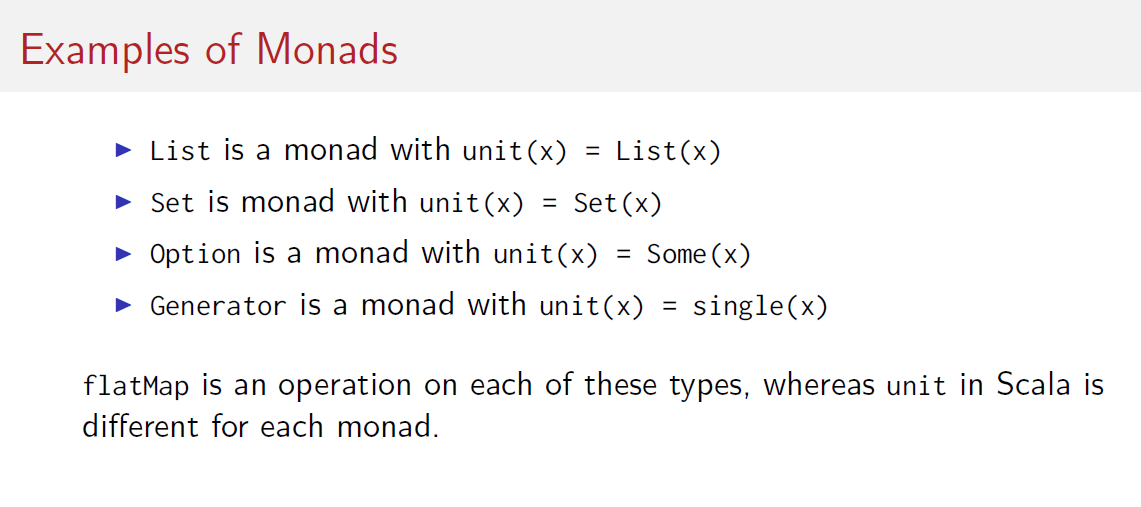
And these operations also need to satisfy some laws.

So, you could can see it in Scala as trait M by M is T, is type parameter it would have a flatMap method of the type that you see here.

So, flatMap takes an arbitrary type U as a parameter and a function that Maps of T to a monad of U and it gives you back the same monad applied to you

And besides flatMap, there would also be the unit method, that unit method takes an element of type of T and gives you an instance of the monad of T.

In the literature flatMap is more commonly called bind but in Scala we have flatMap already an established name so I will continue to use that.



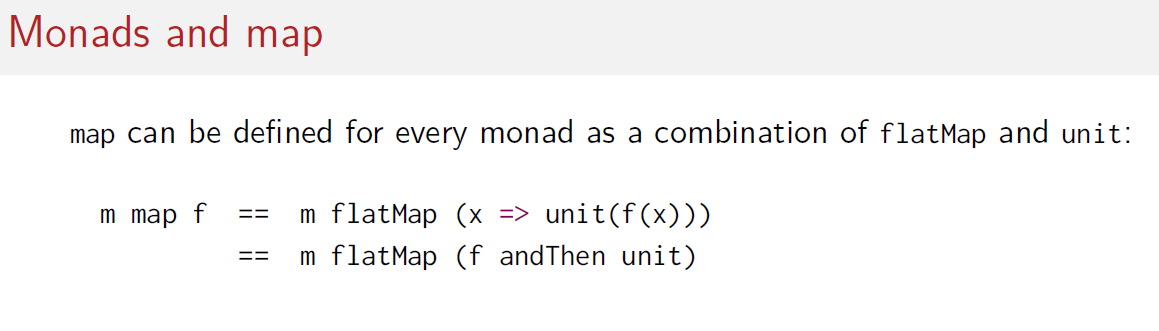
As you've seen, flatMap was available as an operation on each of these types, whereas the unit in Scala is different for each monad.

Quite often it's just the name of the monad type applied to an argument, but sometimes it's different.

For instance, for generator we use single(), for option we use Some()

So, we've seen monads have the flatMap operation. What about map?

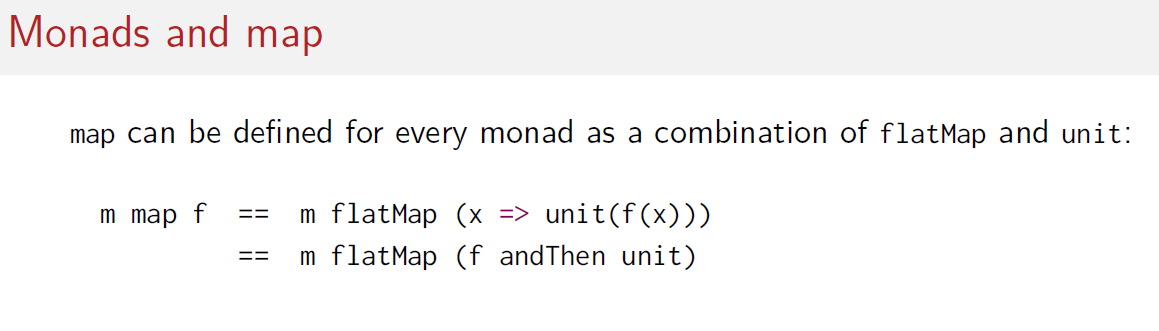
Well in fact map can be defined for every monad as a combination of flatMap and unit. So, the map applied to a monad with the function f() would be flatMap of. First apply f() to the argument x and then reinject into the m map using unit.

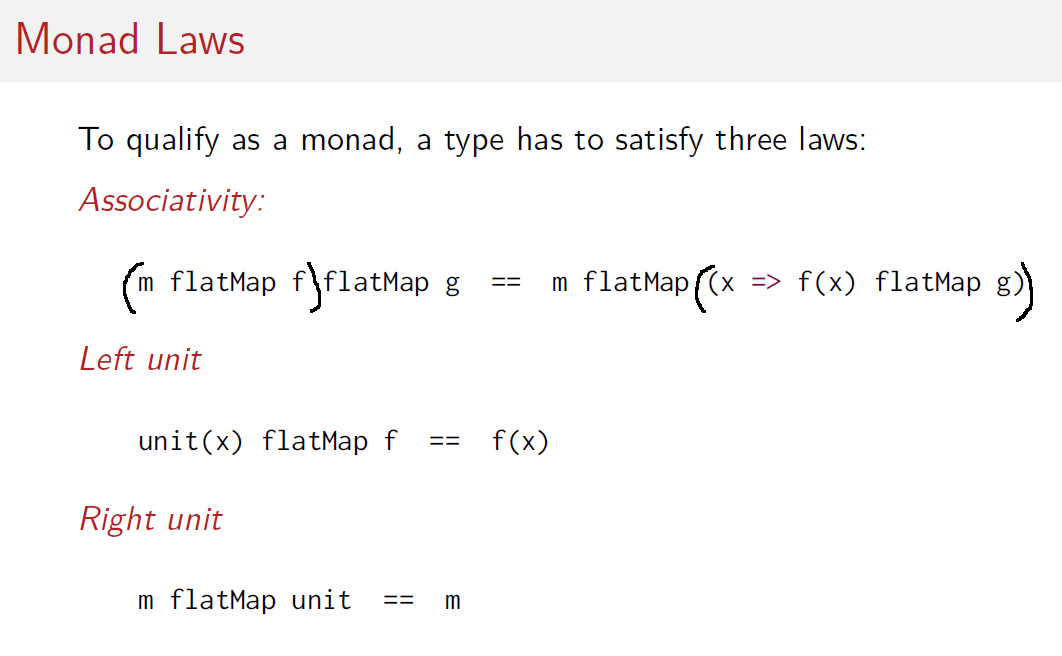


Another way to write this expression would be to use the andThen Combinator instead of the f for function composition.

So you could say m map f is m flatMap of f() and then unit. So, you first apply the m function and then you apply the unit function to the result of that.

Now in Scala, we do not have a unit that we can call here because every monad has a different expression that gives the unit value. Therefore, map is in Scala a primitive function that is also defined on every monad.





To qualify as a monad, a type must satisfy three laws that connect flatMap and unit.

Associativity is as usual a law about placing parentheses, the parenthesis would be placed on the left-hand side, and here you see them placed on the right-hand side here.

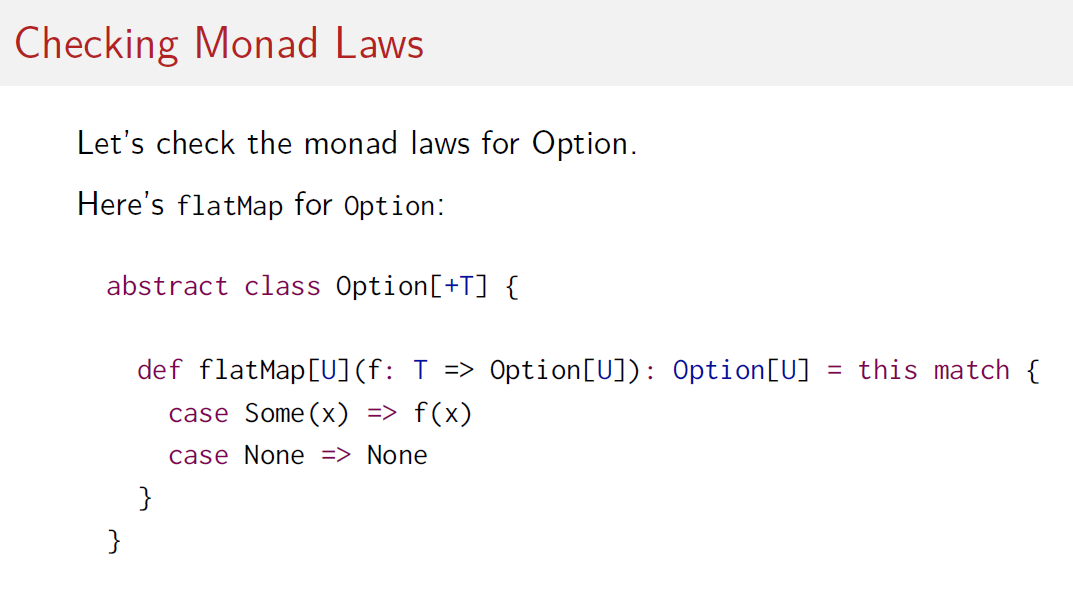
So, we can alternatively either do the flatMap here first or combine the two functions in the flatMap and apply to the monad.

Domain is a bit easier to express if we go from monads to monoids.

Monoids is a simpler form of monads that doesn't bind anything. So, for instance, integers are a monoid, and they're associative because (x + y) + z is the same thing as x + (y +z). So, again, I can put the parentheses either to the left or to the right.

The second law that needs to hold for a monad is called left-unit. It says that if I inject into the monad using unit, and then flatMap with f() and the result is simply the same as simply applying f(x).

The last law is right-unit. It says that if I flatMap with the unit constructor, I end up with the same monad value as before.



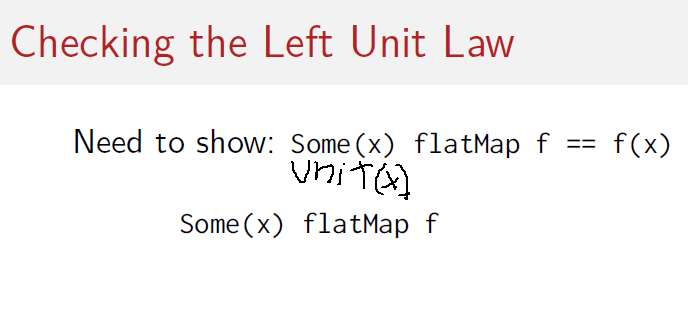
Let's check the monad laws for flatMap type. I pick Option for that.

First thing we must do is look at flatMap for Option.

So, what flatMap should do is it should take an optional value. If that optional value is so

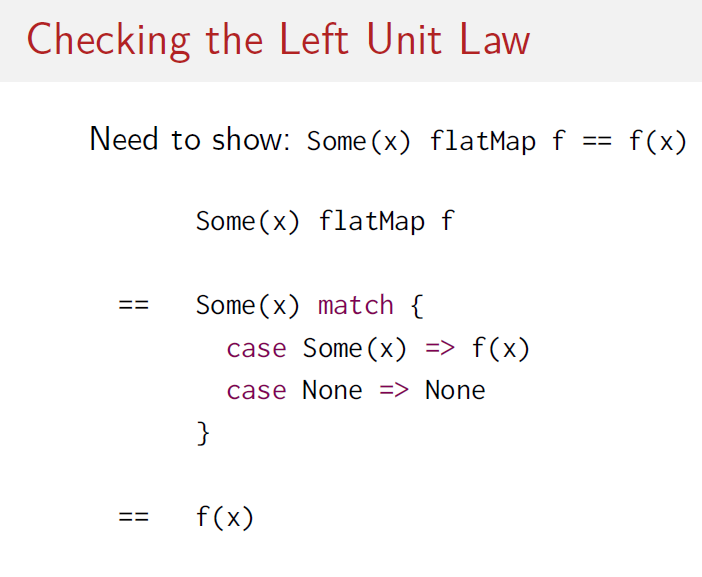
we have nothing and we keep none.

If the optional value is something with a value x, we apply a given function to that value. And that will give us another optional value.



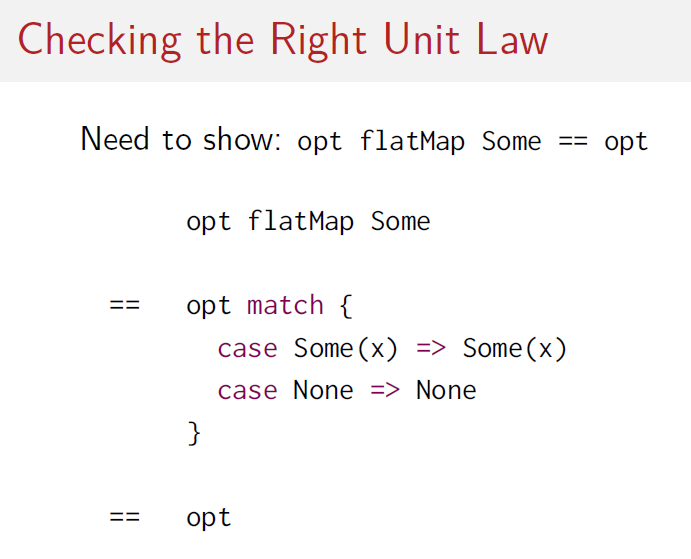
Let's start to prove the left-unit law. What we need to show here is that unit of x for option that was some of x, as you know, flatMap f is the same thing as f(x).

So, let's start with that expression, Some(x) flatMap f and expand what flatMap means.



So, flatMap is this pattern match that says well, if it's Some(x), then apply f to it.

If it's None, keep on having None. Now that we can simplify obviously, because we have a Some(x) here, so it clearly matches the first pattern. And that would give us f(x) as the result. So, the left-unit law holds.

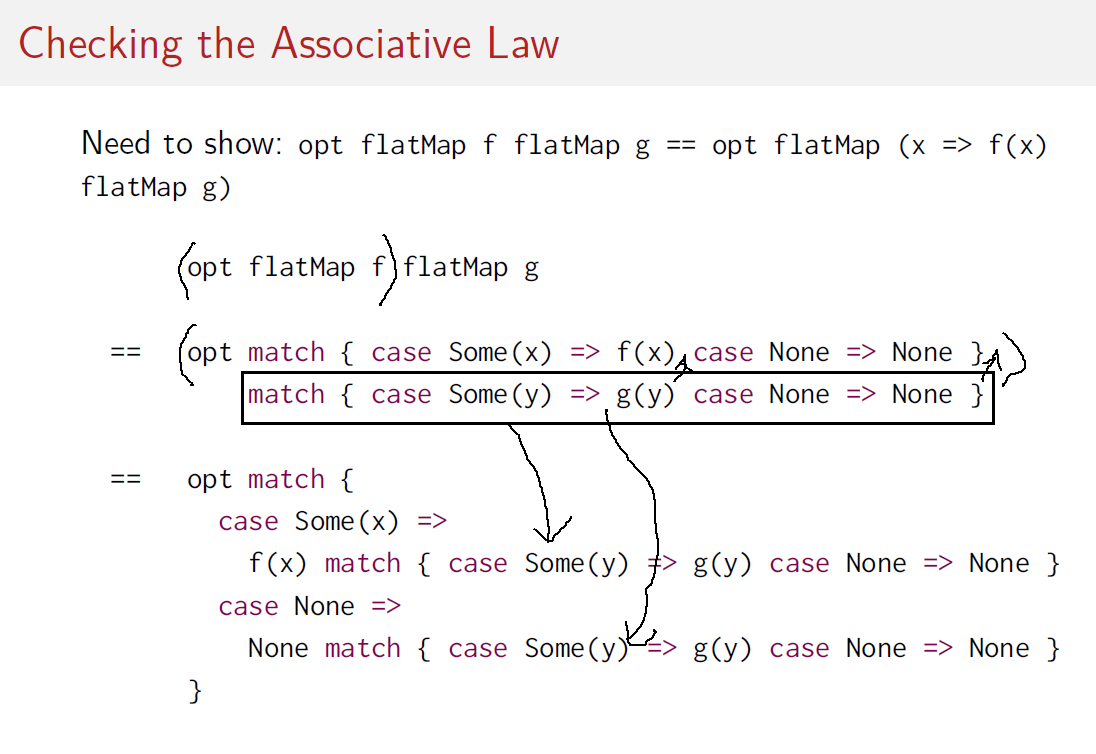


Let's look at the right-unit law now. So, the right-unit law says that, some optional value flatMap with some which is the unit constructor is the same thing as that optional value.

So, we start with the left hand side, optional value flatMap Some. We expand what flatMap means, so we again have this pattern match which says will match if it's Some of x then we turn now, our function f is Some.

So the function f() here gets inserted here so we return sum of x, and if it's none we return none.

Then again simplifies to just opt because we see that in each of the two branches of the pattern match we turn exactly the thing we started with. So, the right-unit law holds also.

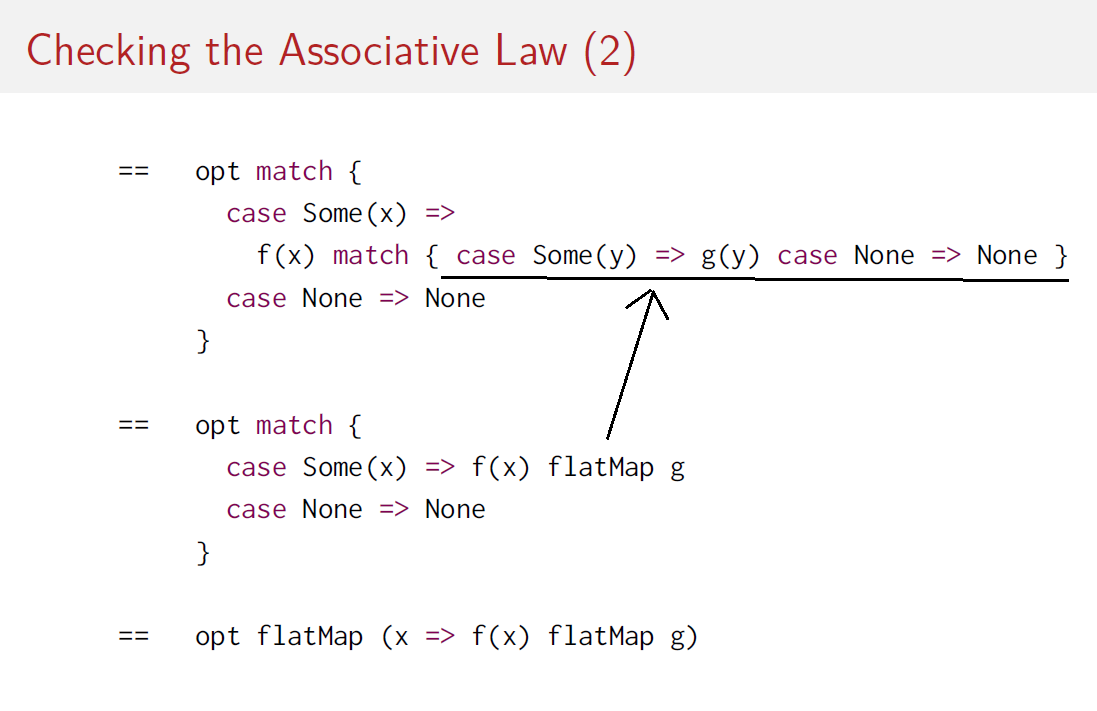


Finally, we must check the associate law. We need to show that the sequence of the two flat maps with the parentheses to the left is the same thing as a flatMap of a flatMap with the parentheses to the right.

So, let's start with the left again. Here we have the sequence of the two flatMaps here is what it expands to. So that line is the first flatMap with the two pattern matches now on one line. And the parentheses go to this That's its expansion. So that result must be subjected to the second flatMap, so here we immediately follow the first match with the second one which now implements the flatMap called with g instead of f before. So that's the expansion of this expression.

Now what I do is I take the second pattern matched this one here. And I move it inside the two branches of the first one. So, I know that the result of the first pattern match will be the selector of the second one. All I did here was to say well let's take each branch of the first pattern match and make it a selector of the second one. So, I pulled this second selector in here, and in here.

That gives me this expression here and that expression now in turn we can simplify so let's look at the case None here first so if the optional value is None Then we have the match None match case some y => g of y case None => none. So obviously it would be the second pair on match that applies.



And the whole expression simplifies to this one that you see here. So, the second case is just if get a None, we keep a None. Let's turn to the first case.

So, in the first case if we say well if we got a some(x) then we match f(x) in turn. Again, if we get a some(y) we give a g(y) and we keep a none.

So what that is, if we look at things in reverse it's just f(x) flatMap g. Because if we expand flatMap g then that expansion in turn gives us exactly that pattern match here.

And if we look at that expression in turn and we see there's just another instance of a flatMap. This time the flatMap is with this function here.

So we say well it's the function that says given an x it will return this expression here. So we get opt flatMap x arrow f of x flatMap g. That's exactly the right-hand side of our original equation we wanted to prove. So, option is a monad because those three laws hold as we have just shown.