EDAF95/EDAN40 - Functional Programming Lecture Notes
L6: Functors, Applicatives, and Monads

Simon Kristoffersson Lind
simon.kristoffersson_lind@cs.lth.se

2024

In this lecture we're going to explore Haskell's most common dataclasses: `Functor`, `Applicative` and `Monad`. Recall from the previous lecture on types, a *dataclass* is basically just an interface in Java (or an abstract class in C++). Despite this, it's very common for people to struggle with grasping Monads. Functors and Applicatives are usually easy enough, but a lot of people struggle with Monads.

Why, then, are Monads particularly difficult?

They're not, really. Personally I think the main problem with Monads is the way people talk about them. But more on that later.

With this lecture, I hope to convince you that Monads really aren't that complicated. **Monad. Is. Just. An. Interface.**

Anyway, let's get into it.

1 Functor

We'll begin with the `Functor` dataclass, the simplest of the three. All it does is define the `fmap` function:

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

If you haven't gotten used to reading Haskell's type signatures yet, this might look quite confusing. Let's look at the `Maybe` type to get a better idea of what this means:

```haskell
instance Functor Maybe where
    --fmap :: (a -> b) -> Maybe a -> Maybe b
    fmap _ Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

Here we've declared that `Maybe` *is a* `Functor`. Or, in Java terminology: `Maybe` *implements* the `Functor` interface. This example also makes it much more clear how to interpret the type of `fmap`.

In my opinion, `list` is the type that perfectly captures the `Functor` typeclass:

```haskell
instance Functor [] where
    --fmap :: (a -> b) -> [a] -> [b]
    fmap = map
```

Yes, `fmap` is just the `map` function for lists.

So, why should we care about the `Functor` typeclass, when we already have lists and the `map` function?

It's the exact same reason you use interfaces in Java: This allows you to define more generic functions that work on *any* type that implements `Functor`. To illustrate:

```haskell
increment_list_elements :: Num a => [a] -> [a]
increment_list_elements xs = map (+1) xs

increment_elements :: (Functor f, Num a) => f a -> f a
increment_elements xs = fmap (+1) xs
```

2 Applicative

The `Applicative` typeclass is a generalization of `Functor`. As such, applicatives are a subset of functors:

$$\text{Applicatives} \subset \text{Functors}$$

In other words, all applicatives are also functors.

So what are applicatives?

`Applicative` generalizes `Functor` to use functions of several arguments.

Recall, the `fmap` function allows us to take a "regular" function of one argument, and map it over a datatype (e.g. the contents of a list). However, sometimes we might want to map "regular" function that take more than one argument. In this case we could define several versions of `fmap`:

```
instance NaiveApplicative f where
    fmap2 :: (a -> b -> c) -> f a -> f b -> f c
    fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
    fmap4 :: (a -> b -> c -> d -> e) -> f a -> f b -> f c -> f d -> f e
    ...
```

Obviously, this becomes tedious very quickly, and it only allows us to get a fixed number of arguments. Can we define a general `fmap` that allows functions with any number of arguments?

Of course the answer is yes, and this is where the `Applicative` typeclass makes use of some clever tricks.

Recall that all functions in Haskell are *Curried*, which means that all functions *take only one argument*. And functions that take several arguments really only take one argument, and return a new function. As such, this:

```
a -> b -> c -> d
```

is really the same as:

```
a -> (b -> (c -> d))
```

This is the trick to generalize `fmap` to any number of arguments. Essentially, we're going to create a function with this type of signature:

```
cool_fmap :: f (a -> b) -> f a -> f b
```

As you can see, this looks very similar to `fmap`. The magic here is that `b` is allowed to be another function! Let's say, as an example, that we have `b = (c -> d)`, then:

```haskell
cool_fmap :: f (a -> (c -> d)) -> f a -> f (c -> d)
```

In other words, this takes a function of two arguments, and returns a function of one argument.

The only remaining piece of this puzzle is how we get the *first* function to become wrapped in the type `f`. If we're working with lists, then we would have to do something like this:

```haskell
cool_fmap (cool_fmap [my_function] my_list1) my_list2
```

But what about other types?

`Applicative` defines a function called `pure`, which fulfills this purpose.

Now let's finally look at the real definition of the `Applicative` typeclass:

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

Let's look at the list implementation as an example:

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

As you can see, the `pure` function just puts whatever it's given into a list. Then, we have the *infix* operator `<*>` that acts just like our `cool_fmap`. This combination of operators gives us a pretty neat syntax for applying functions:

```haskell
list1 = [1, 2, 3, 4, 5]
list2 = [6, 7, 8, 9, 10]

pure (+) <*> list1 <*> list2
-- [7,8,9,10,11,8,9,10,11,12,9,10,11,12,13,10,11,12,13,14,11,12,13,14,15]
```

This resulting value might look a bit surprising when you first see it. Essentially, it has applied the `(+)` function for each possible combination of arguments from the lists (i.e. [1+6, 1+7, 1+8, 1+9, 1+10, 2+6, ...]). This comes from the fact that the list implementation of `<*>` uses list comprehension to iterate over both the list of functions and the list of arguments. Effectively, using set notation, we're applying `(+)` to the set of arguments $\text{List}_1 \times \text{List}_2$.

`Applicative` actually also defines an additional operator `<$>`, but we don't have to implement that one ourselves. It allows us to write the same thing like this instead:

```
(+) <$> list1 <*> list2
-- [7,8,9,10,11,8,9,10,11,12,9,10,11,12,13,10,11,12,13,14,11,12,13,14,15]
```

Which one you use is just a matter of preference. Personally, I prefer to use `pure` to signal that the function is being lifted into the `Applicative` type.

3 Monad

Finally we're here, Monads.

Just like Applicatives are a subset of Functors, Monads are a subset of Applicatives:

$$\text{Monads} \subset \text{Applicatives} \subset \text{Functors}$$

In other words, all Monads are also Applicatives.

Unlike `Applicative`, `Monad` isn't a direct generalization of the same pattern. I would rather call `Monad` an extension of `Applicative`. Specifically, `Monad` allows us to chain several functions in a neat way. Allow me create a simple example to illustrate.

Say you are working as a tax official, and you have a list that you use to map a person's name to their national ID-number (think Swedish personal number or US social security number):

```
people = [("Alice", 1), ("Bob", 2)]
```

You also have a list that you use to map a persons ID-number to the amount of tax they owe:

```
tax_owed = [(1, 1200), (3, -5400)]
```

It's tedious to manually look up a person's name, and then their owed tax, so you want to create a function that automates this process.

You immediately decide that the `lookup` function is a good candidate:

```haskell
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

Recall that it takes a key and and a list of key-value pairs. If the key exists in the list of key-value pairs, then the corresponding value is returned. Otherwise it returns `Nothing`.

Since `lookup` returns a `Maybe`, you create a helper function to take the value out of the `Maybe`:

```haskell
unwrap :: Maybe a -> a
unwrap (Just x) = x
```

Of course, this function will crash if it gets a `Nothing`, so we have to check the argument first.

With this, you implement your function:

```haskell
find_tax :: String -> Maybe Integer
find_tax p
    | identifier /= Nothing = lookup (unwrap identifier) tax_owed
    | otherwise = Nothing
    where
    identifier = lookup p people
```

Now, does this work? Yes.
But is it tedious? Also yes, because we have to manually check and unwrap the `Maybe`. Imagine if we had 3 calls to `lookup`, or even more. Then this type of function would quickly become unreasonably large.

So, what do we do instead?

Let's think about it for a second. The issue is that we have to check if it's `Nothing`, and then unwrap it. And if we had more `Lookup`s, we would have to check and unwrap before every new `lookup`. As it turns out, we can easily capture this pattern in a helper function:

```haskell
bind :: Maybe a -> (a -> Maybe b) -> Maybe b
bind Nothing _ = Nothing
bind (Just x) f = f x
```

And this allows us to rewrite our `find_tax` function much more cleanly:

```haskell
find_tax :: String -> Maybe Integer
find_tax p = bind (lookup p people) (\i -> lookup i tax_owed)
```

That's it, you've just discovered the `Monad`!

Lecture, done!

Just kidding, we still have a few more things to cover. But `Monad` is really that simple. Recall how `Functor` is just the `map` function. `Monad` is just the `bind` function.

However, just like with `Applicative`, Haskell gives us some nicer syntax for it. Here's what that looks like:

```haskell
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

Here we have the *infix* operator `>>=` which works exactly like our `bind` function.

We also have a function called `return`, which is really just another name for the `pure` function. You'll see why we have that soon. For now we focus on the bind operator.

Here's the `Monad` implementation for `Maybe`:

```haskell
instance Monad Maybe where
    return x = Just x
    x >>= f = bind x f
```

Of course, the official Haskell definition doesn't use our `bind` function. I just used it to highlight that they work exactly the same.

Now this allows us to rewrite `find_tax` even nicer:

```haskell
find_tax :: String -> Maybe Integer
find_tax p = lookup p people >>= (\i -> lookup i tax_owed)
```

But wait, it doesn't end there! Haskell also gives us the so-called `do`-notation, which allows us to mimic imperative programming. And this is where the `return` function comes in:

```haskell
find_tax :: String -> Maybe Integer
find_tax p = do
    i <- lookup p people
    owed <- lookup i tax_owed
    return owed
```

Really this is just syntactic sugar. In fact the compiler translates this to:

```haskell
find_tax :: String -> Maybe Integer
find_tax p = lookup p people >>= (\i -> lookup i tax_owed >>= (\owed ->
 ↪  return owed))
```

Thats it!

For real this time.
That's all it is.

In my opinion, `Maybe` is the quintessential `Monad`, because it perfectly captures this pattern of operation (the bind operator).

"But wait", you may be thinking, "that wasn't so bad".

Indeed, the `Monad` typeclass really isn't that complicated when you break it down.

However, next we'll look at an *application* of the `Monad` typeclass. In Java language, we'll look at a specific *implementation* of the `Monad` interface. And this is where things get confusing. I also think this is where people get confused about Monads, because the *concept* of a `Monad` gets mixed up with this specific *application* of `Monad`.

3.1 Stateful Computations using Monads

When you go anywhere to read about Monads, you often hear something like:

> "they allow us to perform stateful computations"

I think this is what creates the most confusion when learning Monads, because it sounds like there is something *special* about Monads, in that they magically enable us to do something that we *couldn't* do without them. This is entirely *false*. Just like I said above, `Monad` is really *just* the bind function

`(>>=)`.

So, what are people referring to when making statements like that?

Well, the bind function `(>>=)`, and especially `do`-notation allows us to do stateful computations in a *nicer way*.

Let me show you with another example. Bear with me, this one will be a bit more involved, because I want to show you all the details. When we're done here, there will (hopefully) be no more magic left in the `Monad`.

In this example, we're going to write a function that takes an `Integer`, let's call it x, and returns the prime factorization of x as a list of primes. To illustrate, we want to create a function like this:

```haskell
prime_factors :: Integer -> [Integer]
prime_factors x = -- TODO...

x = 360 -- This is 2*2*2*3*3*5
prime_factors x
-- [2, 2, 2, 3, 3, 5]
```

The general algorithm we're going to implement looks like this (Java-ish pseudo code):

```
list_of_primes = []
while (x > 1) {
    p = get_next_prime_number()
    while ((x mod p) == 0) {
        x = x / p
        list_of_primes.add(p)
    }
}
return list_of_primes
```

I'm sure most of you already see approximately how this will look in Haskell. So let's start writing:

```haskell
prime_factors :: Integer -> [Integer]
prime_factors 1 = []
prime_factors x = (replicate n p) ++ prime_factors x'
        where
        p = next_prime
        (n, x') = extract_prime_factor p x
```

Here I've used the `replicate` function to add `n` copies of `p` to the resulting list. Of course this isn't correct (yet). There are some crucial parts missing. First and foremost, I plugged in two functions that don't exist yet (`next_prime` and `extract_prime_factor`).

Let's start with `extract_prime_factor`. It should take our `x` and our current prime `p`, and it should return the number of time we can divide `x` by `p`, along with the new `x`-value after dividing by `p`.

Here we need to keep track of how many times we've divided by `p`, so I'll create a local function using a `where`-clause. This allows me to thread a counter variable through the recursive function calls:

```
-- We return (Int, Integer) because the repeat function takes an Int,
-- and not Integer, so n needs to be an Int.
extract_prime_factor :: Integer -> Integer -> (Int, Integer)
extract_prime_factor p x = extract_pfactor p 0 x
    where
    extract_pfactor p n x
        | mod x p == 0 = extract_pfactor p (n+1) (div x p)
        | otherwise = (n, x)
```

Easy enough. We can argue that this is already a stateful computation, since we have to keep track of the counter. However, the state is only internal, and thus we can hide it inside our `where`-clause.

The `next_prime` function is where the state becomes more explicit. In order to compute the *next* prime, we need all the *previous* primes. Obviously, we don't want to recompute all primes every time we call `next_prime`, so we'll need to explicitly handle a list of all our primes. Let's start by modifying `prime_factors`:

```
prime_factors :: Integer -> [Integer]
prime_factors x = pfactors [] x
    where
    pfactors _ 1 = []
    pfactors primes x = (replicate n p) ++ pfactors primes' x'
        where
        (p, primes') = next_prime primes
        (n, x') = extract_prime_factor p x
```

Here I've modified the function to take a list of all the previous primes we've computed. Once again, I've used a `where`-clause to hide this internal state inside a local function. Now the `next_prime` function looks more reasonable

as well, since it no longer magically produces the next prime from nothing.

Let's implement `next_prime`. First we'll need a helper function to simply check if a number is prime:

```
is_prime :: [Integer] -> Integer -> Bool
is_prime [] _ = True
is_prime (p:ps) x
    | mod x p == 0 = False
    | p*p > x = True
    | otherwise = is_prime ps x
```

We simply run through the list of primes that we already have computed, to see if `x` is divisible by any of them. If it is not, then `x` is prime.

Before we continue, just a quick word on the line:

```
    | p*p > x = True
```

This is just an optimization. We don't need to check *all* previous primes, only up to $\sqrt{x}$. As such, this tiny optimization makes our search for new primes **a lot** faster.

On to the actual implementation of `next_prime`:

```
next_prime :: [Integer] -> (Integer, [Integer])
next_prime [] = (2, [2])
next_prime [2] = (3, [2,3])
next_prime prev_primes = (p, prev_primes ++ [p])
    where
    p = until (is_prime prev_primes) (+2) (last prev_primes + 2)
```

Here we implement two base cases to get the search started. Then in the general case, we simply check every odd number by using `until` to loop until we find a new prime.

Okay, done!

Let's quickly summarize our final implementation:

```haskell
is_prime :: [Integer] -> Integer -> Bool
is_prime [] _ = True
is_prime (p:ps) x
    | mod x p == 0 = False
    | p*p > x = True
    | otherwise = is_prime ps x

next_prime :: [Integer] -> (Integer, [Integer])
next_prime [] = (2, [2])
next_prime [2] = (3, [2,3])
next_prime prev_primes = (p, prev_primes ++ [p])
    where
    p = until (is_prime prev_primes) (+2) (last prev_primes + 2)

extract_prime_factor :: Integer -> Integer -> (Int, Integer)
extract_prime_factor p x = extract_pfactor p 0 x
    where
    extract_pfactor p n x
        | mod x p == 0 = extract_pfactor p (n+1) (div x p)
        | otherwise = (n, x)

prime_factors :: Integer -> [Integer]
prime_factors x = pfactors [] x
    where
    pfactors _ 1 = []
    pfactors primes x = (replicate n p) ++ pfactors primes' next_x
        where
        (p, primes') = next_prime primes
        (n, next_x) = extract_prime_factor p x
```

The part that is really considered stateful here is the `next_prime` function, since it requires all the previous primes. As such, `next_prime` is the part that we're going to change by using a `Monad` instead.

I want to be *really really* clear here: the `Monad` typeclass doesn't magically allow us to do something new, that we couldn't do without it! Clearly we've already implemented a stateful function (`next_prime`), and it worked *just fine*. However, we'll see that using a `Monad` allows us to make stateful functions *nicer to work with*, by hiding the state. After all, we don't really care about the list of primes, so let's hide it inside a `Monad` ☺.

Introducing: the State Monad!

```haskell
newtype Stateful a b = ST (a -> (b, a))
```

Here `a` is our *state* type, and `b` is the type of the thing we actually care about.

In our case for `next_prime`, our state is our list of primes so the type `a` is `[Integer]`, and the thing we care about is our prime so the type `b` is `Integer`.

Now we're going to create our instance of the `Monad` typeclass. Or *implement* the `Monad` interface in Java terms. Recall that Monads ⊂ Applicatives ⊂ Functors, and therefore we also have to implement `Functor` and `Applicative`:

```
instance Functor (Stateful a) where
    -- fmap :: (b -> c) -> Stateful a b -> Stateful a c
    fmap f (ST gen) = ST (\state -> let (x, state') = gen state
                                        in (f x, state'))

instance Applicative (Stateful a) where
    -- pure :: b -> Stateful a b
    pure x = ST (\state -> (x, state))

    -- (<*>) :: Stateful a (b -> c) -> Stateful a b -> Stateful a c
    (ST f) <*> (ST gen) = ST (\state -> let (x, state') = f state
                                            (y, state'') = gen state'
                                            in (x y, state''))

instance Monad (Stateful a) where
    -- (>>=) :: Stateful a b -> (b -> Stateful a c) -> Stateful a c
    (ST gen) >>= f = ST (\state -> let (x, state') = gen state
                                       (ST gen2) = f x
                                       in gen2 state')
```

You'll notice that I didn't explicitly implement `return`. That is because we get the default implementation:

```
return = pure
```

entirely for free! ☺.

I know this is a lot to take in at once. Please do take the time to read the types for each of these functions, as the types really help with understanding what these functions do. Speficially, pay attention to the fact that all these functions *always act on the same state type* `a`. In other words, all of these manipulate only the *thing-we-care-about*.

Don't worry, I'll help you break down the bind function `(>>=)` to better understand it. (Since we won't really use `fmap` or `<*>` in our example, I'll leave those as an exercise ☺). But first I need an example to work with, so let's

implement our new-and-improved `next_prime`!

We'll begin by declaring a new type `PrimeGenerator`, to capture our state:

```
type PrimeGenerator b = Stateful [Integer] b
```

We use the `type` keyword here, so `PrimeGenerator b` is really just an alias for `Stateful [Integer] b`. In other words, we're really just specifying that our state-type is `[Integer]` (our list of primes).
Now we can use this to write our new version of `next_prime`:

```
next_prime :: PrimeGenerator Integer
next_prime = ST np
    where
    np [] = (2, [2])
    np [2] = (3, [2, 3])
    np ps = let p = until (is_prime ps) (+2) (last ps + 2)
                    in (p, ps ++ [p])
```

Okay, let's break it down.

First of all, I'm sure you can see that we have all the same components in this version, it's just structured a bit differently. Really the only difference is that *we no longer return the next prime directly*! Instead, we return *a function that generates the next prime*. And this is the "magic" that allows us to hide the state.

In order to really show you what's going on, I'll write a little helper function:

```
run :: PrimeGenerator a -> [Integer] -> (a, [Integer])
run (ST gen) state = gen state
```

Recall that our new version of `next_prime` returns `ST np`, where `np` is the function that actually generates the next prime. Now we can use this `run` function to actually generate primes:

```haskell
run next_prime []
-- (2, [2])

run next_prime [2]
-- (3, [2, 3])

run next_prime [2, 3]
-- (5, [2, 3, 5])

-- etc...
```

of course, we're not just going to do `run next_prime` in our `prime_factors` function. That would be pointless, since we'd still have to keep track of the list of primes.

Instead, we're going to let the bind operator `(>>=)` keep track of our state for us ☺.

To illustrate how powerful this function-return style is, ill show some examples. First, with `fmap` from `Functor`, we can do this:

```haskell
run (fmap (+1) next_prime) []
-- (3, [2])

run (fmap (+1) next_prime) [2]
-- (4, [2, 3])

run (fmap (+1) next_prime) [2, 3]
-- (6, [2, 3, 5])
```

And the `<*>` operator from `Applicative` allows us to do this:

```haskell
run (pure (+) <*> next_prime <*> next_prime) []
-- (5, [2, 3])

run (pure (+) <*> next_prime <*> next_prime) [2]
-- (8, [2, 3, 5])

run (pure (+) <*> next_prime <*> next_prime) [2, 3]
-- (12, [2, 3, 5, 7])
```

Here we start to see the power. We've made it generate *two* primes, but we only had to give it a list of previous primes *once*.
Admit it, that's pretty cool ☺.

Finally, the monad allows us to take it one step further:

```
example = do
    a <- next_prime
    b <- next_prime
    c <- next_prime
    return (a + b + c)

run example []
-- (10, [2, 3, 5])

run example [2]
-- (15, [2, 3, 5, 7])

run example [2, 3]
-- (23, [2, 3, 5, 7, 11])
```

Now we can make it generate any number of primes we want, and we can do whatever we want with the primes, and we still only have to give it a list of primes *once*.
Absolutely beautiful, isn't it? 😎

Let's break it down to see what's going on.

In order to break it down, I'll take a step back to a slightly simpler version:

```
example = do
    a <- next_prime
    b <- next_prime
    return (a + b)
```

Recall that this is compiled to:

```
example = next_prime >>= (\a ->
          next_prime >>= (\b ->
          ST (\state -> ((a + b), state))))
```

And if we unpack the bind operator in the `Monad` implementation (i.e. make it more readable) we have:

```
(ST gen) >>= f = ST bind
    where
    bind ps = gen2 ps'
        where
        (x, ps') = gen ps
        (ST gen2) = f x
```

In our `example` function, our furst argument to `>>=` is `next_prime`. The bind function unwraps our `np` function from inside `next_prime`, and applies it to the state `ps` (initially []). This causes us to generate one prime, and return it along with the new state `ps'` ([2]). It then takes that prime and passes it into the function f, which in our case is:

```
(\a -> next_prime >>= (\b -> ST (\state -> ((a + b), state))))
```

As such, `a` becomes the first generated prime number. In the next bind operator `(>>=)`, we have another `next_prime`, and so it repeats the same process for that one. As such, we get a second generated prime in `b` before the final function gives us `(a + b)`.

When it comes to the state (our list of primes), bind first gives the state `ps` to `next_state`. From there it gets the updated state `ps'`, which it simply passes along to the function on the right. And again, it repeats.

As you can see, this construction automatically chains the list of primes for us, while only giving us the individual prime numbers we care about.

So, let's finally look at the improved implementation of `prime_factors`:

```
prime_factors :: Integer -> [Integer]
prime_factors x = fst (run (pfactors x) [])
    where
    pfactors 1 = return []
    pfactors x = do
        p <- next_prime
        let (n, x') = extract_prime_factor p x
        rest <- pfactors x'
        return ((replicate n p) ++ rest)
```

Much more readable if I say so myself ☺.