Fluokitten: Functors, Applicatives and Monads in Pictures - in Clojure

This tutorial is a Clojure version of the article Functors, Applicatives and Monads in Pictures (https://web.archive.org/web/20180218211616/http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html). The original article was written in Haskell, and is an excellent introduction to the very basics of functors, applicatives, and monads. This article is not self-contained: it is ment to be read side-by-side with the original article (https://web.archive.org/web/20180218211616/http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html), and used as a commentary and reference for the Clojure version of the examples. Due to the differences in typing, data structures, support for varargs and currying, there are differences in how the concepts that the original article explains are implemented in Haskell and Clojure.

To be able to follow this article, you'll have to have Clojure installed and Fluokitten library included as a dependency in your project, as described in Getting Started Guide (/articles/getting_started.html). Obviously, you'll need a reasonable knowledge of Clojure (you don't have to be an expert, though), and being familiar with the basics of Haskell is helpful, but not a necessity. So, after checking out Getting Started Guide (/articles/getting_started.html), start up Clojure REPL and open this article and the original article (https://web.archive.org/web/20180218211616/http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html) and we are ready to go.

The complete source code of the examples used in this article is available here (https://github.com/uncomplicate/fluokitten/blob/master/test/uncomplicate/fluokitten/articles/functors_applicatives_monads_pictures_test.clj) in the form of midje (https://github.com/marick/Midje) tests.

The concept of a box as a simple context

When we are working with some values as they are, the function is applied to a value, resulting in some new value as a result:

```
2
;=> 2

((partial + 3) 2)
;=> 5
```

There should be nothing confusing here; the only difference is that, since Clojure does not support automatic currying, `(+3)` would result in applying `+` to `3`, resulting with number `3` instead of a function that adds `3` as in Haskell. Therefore, in Clojure we use `partial` that enables the equivalent behavior. Note: Fluokitten has a `curry` function that can transform any function to its automatically curried version, but it is not necessary for this article, so we opted for the mainstream Clojure version.

Now, about boxes. Lists, vectors, other data structures are kinds of "boxes" that contain other values. There is an even simpler kind of a box - Maybe. That is an existing box that contains some value - `Just something`, or a box that indicates that something went wrong with the process of putting a value in a box, or something was wrong with the value itself - `Nothing`. In Clojure, nil is used as a value that indicates non-existence, so in clojure we have `(just 2)` as a simple box that contains value `2` or nil:

```
(just 2)
;=> (just 2)

nil
;=> nil
```

Box as a box, whether it is a `Just` or a list, tree, or something more complex has its own utility and functionality, which is something that is not of interest for us here. Category theory applied to programming is interested in seeing what is common to all these boxes, what is general enough that can be done to all these kinds of boxes, in a same way without concern for the actual type of the box. So a particular box can be used in a number of different general ways, and depending on the ways of usage that it supports, the box can be given one or more of the general names: functor, applicative functor, monad etc. (there are more, but this tutorial cover these three).

Functor

The simplest of these general concepts is functor. Here is a problem that it solves:

```
((partial + 3) (just 2))
;=> (throws ClassCastException)
```

A plain function that accepts number cannot accept a box with a number. Straightforward solution would be to create a special function that extracts the value and then applies the original function, but this solution is not general enough - we would have to check the type of box, find a relevant extractor function, etc.

This is where functors come in: if some box can accept functions to be applied to the context of a box (and most, if not all boxes can do this) we can say that the box is a functor and use `fmap` function just for that purpose:

```
(fmap (partial + 3) (just 2))
;=> (just 5)
```

`fmap` is very convenient: it not only applies a normal function to the box, it return its result conveniently boxed!

Just what is a Functor really?

Haskell has its own machinery for implementing this, as sketched in the original article. Clojure uses its own, different mechanism - protocols. In Clojure, `Functor` is a protocol that defines the `fmap` method. If you want a box you are creating to be a functor, you'll implement the `Functor` protocol, and make sure (by examining the code, testing, etc) that it satisfies functor laws, which ensure that your `fmap` implementation is a well behaved `fmap`. Clojure `fmap` also supports varargs (multiple arguments) so it can apply a multiargument function to many boxes, while in Haskell can accept only one box. When we `fmap` a function (even the one that would normaly break its teeth on a broken value and raise an exception) to a broken Maybe box, we get a broken box:

```
(fmap (partial + 3) nil)
;=> nil
```

The next example is specific to Maybe, but shows how functors can save us working on particulars and concentrate on the generalities:

```
(def posts {1 {:title "Apples"}})

(defn find-post [post-id]
  (if-let [post (posts post-id)]
    (just post)
    nil))

(defn get-post-title [post]
  "This example is intentionally silly to correspond to the
   intention of the original example. Clojure maps handle
   the nil-map case well, but the real database title fetching
   function might not."
  (post :title))

(fmap get-post-title (find-post 1))
;=> (just "Apples")

(fmap get-post-title (find-post 2))
;=> nil)
```

And, let's see a slightly more complex box - Clojure vector:

```
(fmap (partial + 3) [2 4 6]) => [5 7 9]
```

Functions are functors, too:

```
(def foo (fmap (partial + 3) (partial + 2)))

(foo 10)
;=> 15)
```

Applicatives

A slightly more powerful kind of box is a box that can accept function(s) to be applied not only bare, but also boxed. Every box that can be applicative can also be a functor.

```
(def add3 (partial + 3))

(just add3)
;=> (just add3)

(fapply (just add3) (just 2))
;=> (just 5)

(<*> (just add3) (just 2))
;=> (just 5))
```

Why `fmap` instead of `<*>` and what is the difference?

First, the difference between Clojure's `fapply` and Haskell's `<*>`. Haskell `<*>` is an infix operator, so it is more convenient to use the symbol, than its (mostly used but not that widely known) name `ap`. In Clojure, we do not use operators, and function application is written in a prefix style, so using an a-z name is more readable. `ap` is too short and indistinctive and `apply` is already taken by Clojure core. Additionally, `fapply` is consistently named with `fmap`.

Second, the difference between `fapply` and `<*>` functions in Clojure. In this example, they are equivalent, but when more parameters are provided, `(<*> x y z)` in Clojure is equivalent to `(fapply (fapply x y) z)` in Clojure and `x <*> y <*> z` in Haskell.

Vectors are also applicative, with an interesting twist that a vector can carry many functions to be applied:

```
(<*> [(partial * 2) (partial + 3)] [1 2 3])
;=> [2 4 6 4 5 6]
```

The following example in the original article is relevant only for Haskell's `fmap`. The goal was to apply a function that takes two arguments to two boxed values. Haskell's `fmap` only accepts one boxed value at a time. Therefore, applicative is needed for the application for the second box, because the result of the first application is a curried `(+3)` function in a box. Clojure's `fmap` does not have such problems, since it can accept as many arguments as the function that is to be applied needs.

```
(fmap * (just 5) (just 3))
;=> (just 15)
```

Monads

Maybe monads seemed bizarre if you tried to understand them right away, but after you familliarize yourself with functors and applicatives, they should be much simpler to grasp. Let's translate the original article's examples to Clojure code.

Here's the `half` function in Clojure:

```
(defn half [x]
  (if (even? x)
    (just (quot x 2))
    nil))
```

It takes a number and maybe produces a boxed number. If the number cannot be halved to two equal integers, it returns `nil` (remember, `nil` is Clojure's `Nothing`)

```
(half 4)
;=> (just 2)

(half 5)
;=> nil
```

Try to feed it a boxed value, and it complains with a nasty exception:

```
(half (just 4))
;=> (throws IllegalArgumentException)
```

Fortunately, since our box (`Just`) is also a monad, it can handle even such cases with the help of the `bind` function:

```
(bind (just 3) half)
;=> nil
```

Similarly to the case with `fapply` and `<*>` , `bind` has its own twin, `>>=` . When only one box is supplied (like in this example) they do the same thing. Here's the rest of the code for the original example, which, by now, you should be able to follow easily:

```
(>>= (just 3) half)
;=> nil

(>>= (just 4) half)
;=> (just 2)

(>>= nil half)
;=> nil

(>>= (just 20) half half half)
;=> nil

(>>= (just 20) half half)
;=> (just 5)
```

The last example in the original article (reading a file and printing) is not really relevant for Clojure, which is not a strictly pure language. In Clojure, we would simply use impure functions for working with input/output, and we are not forced to use IO Monad for that. Therefore, we skip this part.

Conclusion

- A functor in Clojure is a type that implements the `Functor` protocol.
- An applicative is a type that implements the `Applicative` protocol.
- A monad is a type that implements the `Monad` protocol.
- Maybe monad is implemented with the `Just` and `nil` types, which implement all three protocols, so they are functors, applicatives, and monads.

Where to go next

To continue with the spirit of the original article, we continue by providing the Clojure version (/articles/learnyouahaskell.html) of the Learn you a Haskell for Great Good (https://learnyouahaskell.com) book. Continue here (/articles/learnyouahaskell.html) Of course, while reading these guides, you should check Fluokitten reference documentation and tutorials (/articles/guides.html#fluokitten_documentation_and_tutorials) out as needed.

Tell Us What You Think!

Please take some time to tell us about your experience with the library and this site. Let us know (/articles/community.html) what we should be explaining or is not clear enough. If you are willing to contribute improvements, even better!

Please support my work on Patreon (https://patreon.com/draganrocks) by adopting a pet Neanderthal function in your name! (https://dragan.rocks/articles/18/Patreon-Announcement-Adopt-a-Function) I'll intive you to a dedicated Discord (https://discordapp.com) discussion server. Can't afford to donate? Ask for a free invite.