Getting Started

This is a brief introductory guide to Fluokitten that aims to give you the necessary information to get up and running, as well as a brief overview of some available resources for learning key category theory concepts and how to apply them in Clojure with Fluokitten.

How to get started

- Walk through this guide, set up your development environment, and try the examples.
- Read some of the introductory tutorials (/articles/guides.html#tutorials_on_category_theory_in_programming) while working through the examples.
- As you go, familiarize yourself with Fluokitten's more advanced guides (/articles/guides.html#fluokitten_documentation_and_tutorials) and API documentation.
- Try your hand at writing your own implementations of Fluokitten protocols (/articles/guides.html).

Overview

Fluokitten is a Clojure library that enables programming paradigms derived from category theory (CT). It provides:

- A core library of CT functions `uncomplicate.fluokitten.core` ;
- Protocols for many CT concepts `uncomplicate.fluokitten.protocols` ;
- Implementations of these protocols for standard Clojure constructs (collections, functions, etc.) `uncomplicate.fluokitten.jvm` ;
- Macros and functions to help you write custom protocol implementations.
- Accompanying website with learning resources.

Installation

The most straightforward way to include Fluokitten in your project is with Leiningen. Add the following dependency to nyour `project.clj` :

```
[uncomplicate/fluokitten "0.9.1"]                    @clojars.org
```

Requirements

Fluokitten requires Java `1.8` , and at least Clojure `1.7` , as it uses transducers.

Usage

See the this tutorial's source as a midje test. (https://github.com/uncomplicate/fluokitten/blob/master/test/uncomplicate/fluokitten/articles/getting_started_test.clj)

First `use` or `require` `uncomplicate.fluokitten.core` and `uncomplicate.fluokitten.jvm` in your namespace, and you'll be able to call appropriate functions from the Fluokitten library.

```
(ns example
  (:use [uncomplicate.fluokitten core jvm]))
```

What functions does this make available? Not many (which is a good thing), because a key point of CT programming is to utilize a small set of highly generalized and well defined operators and combinators that can be mixed and combined simply to achieve desired outcomes. The following demonstrates simple examples, mostly using Fluokitten extensions to Clojure built-in constructs.

Functors and fmap

The basic categorical concepts concern contexts for data. One familiar data context is a sequence, for example one that represents several numbers. Suppose we have a vanilla function that operates on numbers and we want to apply it to data without regard to the wrapper (context) that contains it. Of course, Clojure already has a function that can reach inside the sequence and apply a function to each element – the ubiquitous `map` function:

```
(map inc [1 2 3])
;=> (2 3 4)
```

The only, usually minor, problem with `map` is that it outputs a lazy sequence, so our context is damaged a bit – we start with a vector and end with a sequence.

Let's try an alternative - the `fmap` function:

```
(fmap inc [1 2 3])
;=> [2 3 4])
```

It's similar to `map` , but the starting context is preserved. `fmap` reaches inside any context that implements the `Functor` protocol (in this case, a Clojure vector), applies a plain function (here, `inc`) to the data inside and produces a result inside the same type of context. Fluokitten extends all standard Clojure collections with the `Functor` protocol, and provides specific implementations of fmap for each, as we see below. Note that, depending on how many arguments the function can accept, we may provide many contexts to `fmap` .

```
(fmap + [1 2 3] [1 2 3 4])
;=> [2 4 6]

(fmap + (list 1 2 3) [1 2] (sorted-set  5 3 1 2))
;=> (3 6)

(fmap + {:a 1 :b 2} {:a 3 :c 4} {:d 5})
;=> {:a 4 :b 2 :c 4 :d 5}
```

Of course, Clojure collections are not the only implementations of the `Functor` protocol. Fluokitten extends most of the Clojure types with the appropriate implementations of `Functor` protocol. For example:

```
(fmap ∗ (atom 2) (ref 3) (atom 4))
;=> (atom 24)

((fmap inc ∗) 2 3)
;=> 7
```

Of course, you can also build your own implementations, which is covered in detailed guides (/articles/guides.html).

Applicative functors: pure and fapply

Starting with the same idea of data inside a context, we can extend it to the function part: what if we want to apply plain functions that are wrapped in context to data in similar contexts? For example, a vector of functions applied to vector(s) of data. For this purpose Fluokitten provides `fapply`:

```
(fapply [inc dec (partial * 10)] [1 2 3])
;=> [2 3 4 0 1 2 10 20 30]
```

`fapply` reaches inside any `Applicative` context (in this case, vector), applies function(s) wrapped in the same type of context (vector) and produces a similarly wrapped result. As an `Applicative`, vector produces a context wrapping results of applying all wrapped functions to all wrapped data. Simpler examples:

```
(fapply [+ -] [1 2] [3 4])
;=> [4 6 -2 -2]

(fapply {:a + :b *} {:a 1 :b 2} {:a 3 :b 3 :c 4} {:d 5})
;=> {:a 4, :b 6, :c 4, :d 5}
```

`Applicative`s also support a function that wraps any data into minimal typed context – `pure`:

```
(pure [] 3)
;=> [3]

(pure (atom nil) 5)
;=> (atom 5)
```

Are these function definitions and implementations arbitrary? NO! All these functions have to satisfy mathematical laws, ensuring they mesh seamlessly with the rest of the framework. Discussion of these laws is well beyond our scope, but you may rest assured that they are rigorously followed in the Fluokitten implementation. When you move beyond using the provided contexts and functions to writing your own CT implementations, you'll have to become familiar with these laws, which are covered in the advanced guides.

Monads and bind

Monads are certainly the most discussed programming concept to come from category theory. Like functors and applicatives, monads deal with data in contexts. However, in addition to applying functions contextually, monads can also transform context by unwrapping data, applying functions to it and rewrapping it, possibly in a completely different context. Sound confusing? Until you gain some practical experience, it is – that is why monad tutorials are written every day. Don't be daunted, however. If you take a step-by-step approach and don't try to swallow everything in one sitting, it's really not hard at all. This tutorial only scratches the surface; please check out the further reading for deeper comprehension. The core monad function is `bind`, and in the case of vector, it is trivially used as follows.

```
(bind [1 2 3] #(return (inc %) (dec %)))
;=> [2 0 3 1 4 2]
```

If the function produces minimal context, it does not need to know which context it is. The return function is going to create the right context for the value, in this case atom.

```
(bind (atom 1) (comp return inc))
;=> (atom 2)
```

Fluokitten implements the `Monad` protocol for many Clojure core types. Please check out the tutorials and docs and be patient until it clicks for you.

Monoids

`Monoid` is a protocol that offers a default operation `op` on some type, and an identity element, `id` for that operation. `op` has to be closed, meaning (op x y) must have the same type as x and y, and it has to be associative. For example, the default operation for numbers in Fluokitten is +, with the identity element 0, while for lists it is concat, with the default element empty list.

```
(id 4)
;=> 0

(op 1 2)
;=> 3

(id [4 5 6])
;=> []

(op [1 2] [3])
;=> [1 2 3]
```

Foldables and fold

Having seen the some manipulation of contexts and data, we'd like some methods to get it back out, without writing custom, context-specific code. If we implement the `Foldable` protocol, which Fluokitten does for many Clojure types, we can use `fold` function to get a summary of the contextual data:

```
(fold (atom 3))
;=> 3
```

With more than one value though, fold aggregates. If the data are subject to a monoid, the accumulating `op` will produce the folded value:

```
(fold [])
;=> nil

(fold [1 2 3])
;=> 6

(fold [[1 2 3] [3 4 5 4 3] [3 2 1]])
;=> [1 2 3 3 4 5 4 3 3 2 1]

(fold (fold [[1 2 3] [3 4 5 4 3] [3 2 1]]))
;=> 31
```

Fold is similar to Clojure's reduce, but more powerful. It can work without explicitly supplied function, and it can accept multiple foldables. If you need to transform entries before folding, there is the `foldmap` function:

```
(fold (fold [[1 2 3] [3 4 5 4 3] [3 2 1]]))
;=> 31

(fold op [] [[1] [2] [3 4]] [[4] [5 6] [6 66]] [[22 33]])
;=> [1 4 22 33 2 5 6 3 4 6 66]

(foldmap + 0 count [[1] [2] [3 4]])
;=> 4
```

Where to go next

Hopefully this guide got you started and now you'd like to learn more. I expect to build a comprehensive base of articles and references for exploring this daunting topic, so please check the All Guides (/articles/guides.html) page from time to time. More importantly, I will post articles with Clojure code for related articles, tutorials and videos, which use another reference language (Haskell) to discuss category theory. Of course, you should also check Fluokitten API for specific details, and feel free to take a gander at the source while you are there.

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!

```
[uncomplicate/fluokitten "0.9.1"]
```
@clojars.org

This website was developed by Dragan Djuric (https://github.com/blueberry).