Fluokitten Extensions of Clojure Core

This article is a guide to using Clojure core artifacts as implementations of Fluokitten protocols. By requiring `org.uncomplicate.fluokitten.jvm` namespace in your namespace, you activate Fluokitten's extensions to Clojure core types so they can act as functors, applicatives, monads etc.

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). So, after checking out Getting Started Guide (/articles/getting_started.html), start up Clojure REPL and open this article 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/fluokitten_extensions_clojure_core_test.clj) in the form of midje (https://github.com/marick/Midje) tests.

Data structures

The following is a list of common Clojure core data structures and their behavior as various Fluokitten protocols implementations. Generally, any data structure can be considered as a specific context that can hold zero, one or more values.

Functor

`fmap` behaves in the same way as `map` for most data structures, but takes care that the resulting data structure is of the same type as the first argument after the function. The number of arguments that the function can accept have to match the number of functors.

```
(fmap inc [])
;=> []

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

(fmap + [1 2] [3 4 5] [6 7 8])
;=> [10 13]

(fmap inc (list))
;=> (list)

(fmap inc (list 1 2 3))
;=> (list 2 3 4)

(fmap + (list 1 2) (list 3 4 5) (list 6 7 8))
;=> (list 10 13)

(fmap inc (empty (seq [1]))) => (empty (seq [2]))

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

(fmap + (seq [1 2]) (seq [3 4 5]) (seq [6 7 8]))
;=> (seq [10 13])

(fmap inc (lazy-seq []))
;=> (lazy-seq [])

(fmap inc (lazy-seq [1 2 3]))
;=> (lazy-seq [2 3 4])

(fmap + (lazy-seq [1 2]) (lazy-seq [3 4 5]) (lazy-seq [6 7 8]))
;=> (lazy-seq [10 13])

(fmap inc #{})
;=> #{}

(fmap inc #{1 2 3})
;=> #{2 3 4}

(fmap + #{1 2} #{3 4 5} #{6 7 8})
;=> #{10 13}
```

Clojure maps (data structure) have a richer structure than the previously described structures. While `map` would treat a map structure as a sequence of entries, `fmap` treats is as a context and just applies a function to the values, while taking care that keys are preserved and matched in the case of a modifying function that takes more than one argument:

```
(fmap inc {})
;=> {}

(fmap inc {:a 1 :b 2 :c 3})
;=> {:a 2 :b 3 :c 4}

(fmap + {:a 1 :b 2} {:a 3 :b 4 :c 5} {:a 6 :b 7 :c 8})
;=> {:a 10 :b 13 :c 13}
```

Map entries are functors, too. The modifying function is applied to the value(s) of map entry or entries, and the key of the first entry will be the key of the result:

```
(fmap inc (first {:a 1}))
;=> (first {:a 2})

(fmap + (first {:a 1})
        (first {:a 3})
        (first {:b 6}))
;=> (first {:a 10})
```

Finally, Clojure's reducibles (https://clojure.com/blog/2012/05/15/anatomy-of-reducer.html) are also properly handled by `fmap` (but it can only accept one reducible at a time):

```
(into [] (fmap inc (r/map identity  [1 2 3])))
;=> [2 3 4]

(into [] (fmap + (r/map identity [1 2])
                 (r/map identity [3 4 5])
                 (r/map identity [6 7 8])))
;=> (throws UnsupportedOperationException)
```

Applicative

For most core structures, `pure` produces a collection of the required type, with the supplied value(s) as its element(s):

```
(pure [4 5] 1)
;=> [1]

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

(pure (list) 1)
;=> (list 1)

(pure (seq [3]) 1)
;=> (seq [1])

(pure (lazy-seq [4]) 1)
;=> (lazy-seq [1])

(pure #{} 1)
;=> #{1}
```

For maps and map entries, `pure` uses nil to generate the default key for the pure entry:

```
(pure {} 1)
;=> {nil 1}

(pure (first {1 1}) 1)
;=> (first {nil 1})
```

If you provide enough elements to pure, it will put them in the appropriate structure.

```
(pure {} 1 2 3 4)
;=> {1 2 3 4}
```

`pure` handles Clojure reducibles (https://clojure.com/blog/2012/05/15/anatomy-of-reducer.html):

```
(into (list) (pure (r/map identity [2 3]) 1))
 => (list 1)
```

We can establish an implicit context with the `utils/with-context macro`, and use `return` or `unit` functions (they are the same) instead of `pure`. `return` ant `unit` call `pure` with the implicit context.

```
(with-context []
  (return 1))
;=> [1]

(with-context {}
  (return 1))
;=> {nil 1}
```

`fapply` is pretty straightforward for most structures, except for maps and map entries, where it has a more complex behavior. It expect two structures of the same type as arguments: a function(s) structure and a data structure. Then it applies the functions to the data. The actual matching of the functions and the data depends on the data structure. For most core structures, it applies all combinations of functions and data:

```
(fapply [] [])
;=> []

(fapply [] [1 2 3])
;=> []

(fapply [inc dec] [])
;=> []

(fapply [inc dec] [1 2 3])
;=> [2 3 4 0 1 2]

(fapply [+ *] [1 2 3] [4 5 6])
;=> [5 7 9 4 10 18]

(fapply (list) (list))
;=> (list)

(fapply (list) (list 1 2 3))
;=> (list)

(fapply (list inc dec) (list))
;=> (list)

(fapply (list inc dec) (list 1 2 3))
;=> (list 2 3 4 0 1 2)

(fapply (list + *) (list 1 2 3) (list 4 5 6))
;=> (list 5 7 9 4 10 18)

(fapply (empty (seq [2])) (empty (seq [3])))
;=> (empty (seq [1]))

(fapply (empty (seq [33])) (seq [1 2 3]))
;=> (empty (seq [44]))

(fapply (seq [inc dec]) (empty (seq [1])))
;=> (empty (seq [3]))

(fapply (seq [inc dec]) (seq [1 2 3]))
;=> (seq [2 3 4 0 1 2])

(fapply (seq [+ *]) (seq [1 2 3]) (seq [4 5 6]))
;=> (seq [5 7 9 4 10 18])

(fapply (lazy-seq []) (lazy-seq []))
;=> (lazy-seq [])

(fapply (lazy-seq []) (lazy-seq [1 2 3]))
;=> (lazy-seq [])

(fapply (lazy-seq [inc dec]) (lazy-seq []))
;=> (lazy-seq [])

(fapply (lazy-seq [inc dec]) (lazy-seq [1 2 3]))
;=> (lazy-seq [2 3 4 0 1 2])

(fapply (lazy-seq [+ *])
        (lazy-seq [1 2 3])
        (lazy-seq [4 5 6]))
;=> (lazy-seq [5 7 9 4 10 18])

(fapply #{} #{})
```

```
;=> #{}

(fapply #{} #{1 2 3})
;=> #{}

(fapply #{inc dec} #{})
;=> #{}

(fapply #{inc dec} #{1 2 3})
;=> #{2 3 4 0 1}

(fapply #{+ *} #{1 2 3} #{4 5 6})
;=> #{5 7 9 4 10 18}
```

In the case of maps and map entries, it matches the functions and the data by map keys, while the `nil` key serves as a universal key, whose function applies to any data that does not have the specific matching function with the same key:

```
(fapply {} {})
;=> {}

(fapply {} {:a 1 :b 2 :c 3})
;=> {:a 1 :b 2 :c 3}

(fapply {:a inc} {})
;=> {}

(fapply {:a inc :b dec nil (partial * 10)}
        {:a 1 :b 2 :c 3 :d 4 nil 5})
;=> {:a 2 :b 1 :c 30 :d 40 nil 50}

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

(fapply (first {:a inc}) (first {:b 1}))
;=> (first {:b 1})

(fapply (first {:a inc}) (first {:a 1}))
;=> (first {:a 2})

(fapply (first {nil inc}) (first {:a 1}))
;=> (first {:a 2})

(fapply (first {nil inc}) (first {nil 1}))
;=> (first {nil 2})
```

Monad

Similarly to the previous examples, `bind` works as expected for most core data structures, except maps and map entries that are a bit specific. It requires one or more data structures, and a function as the last parameter. That function should accept elements extracted from all provided data structures (one by one), and wrap the result(s) in the same type of structure. The bind function is automatically and transparently aware of the implicit context, so we can use `return` or `unit` to create agnostic monadic functions that can be applied to any monad. In the following example, the `increment` and `add` functions' results will be packaged in appropriate contexts transparently.

Here are typical examples of sequences and friends:

```
(def increment (comp return inc))
(def add (comp return +))

(bind [] increment)
;=> []

(bind [1 2 3] increment)
;=> [2 3 4]

(bind [1 2 3] [4 5 6] add)
;=> [5 7 9]

(bind (list) increment)
;=> (list)

(bind (list 1 2 3) increment)
;=> (list 2 3 4)

(bind (list 1 2 3) (list 4 5 6) add)
;=> (list 5 7 9)

(bind (empty (seq [2])) increment)
;=> (empty (seq [3]))

(bind (seq [1 2 3]) increment) => (seq [2 3 4])

(bind (seq [1 2 3]) (seq [4 5 6]) add)
;=> (seq [5 7 9])

(bind (lazy-seq []) increment)
;=> (lazy-seq [])

(bind (lazy-seq [1 2 3]) increment)
;=> (lazy-seq [2 3 4])

(bind (lazy-seq [1 2 3]) (lazy-seq [4 5 6]) add)
;=> (lazy-seq [5 7 9])

(bind #{} increment)
;=> #{}

(bind #{1 2 3} increment)
;=> #{2 3 4}

(bind #{1 2 3} #{4 5 6} add)
;=> #{5 7 9}
```

With maps and map entries, `bind` treats keys as parts of the context, and feeds only the values of the corresponding entries to the function. Since the function returns a map for each entry, the results have to be flattened, so the final result is a map. It is easier to understand what happens with a few examples:

```
(bind {} #(hash-map :x %)) => {}

(bind {:a 1} #(hash-map :increment (inc %)))
;=> {[:a :increment] 2}

(bind {:a 1  :b 2 :c 3} #(hash-map :increment (inc %)))
;=> {[:a :increment] 2 [:b :increment] 3 [:c :increment] 4}

(bind {:a 1} {:a 2 :b 3} {:b 4 :c 5} (fn [& args] {:sum (apply + args)}))
;=> {[:a :sum] 3 [:b :sum] 7 [:c :sum] 5}
```

So, the function takes the corresponding values for each key from each map, feed it to the function, which returns the result wrapped in a context of a map. Then, all the resulting maps are joined by pairing the key of the input and the key of the output for a particular value.

Map entries do not care about the keys when feeding the values to the function. Only the key of the first map entry is used as the key of the result:

```
(bind (first {:a 1}) #(first {:increment (inc %)}))
=> (first {[:a :increment] 2})

(bind (first {:a 1}) (first {:a 2}) (first {:b 4}) (fn [& args] (first {:sum (apply + args)})))
=> {[:a :sum] 8}
```

The `join` function flattens one nesting level of the the data structure if it contains nested data structures of the same type, in a similar way as clojure's `flatten` function (but only one level deep), for all collections except maps. For maps, it have to take account of the nesting of the map's keys as parts of the context, by creating a vector of all the keys that were flattened and using it as the key for the value of the flattened entry, as shown in the following examples:

```
(join [[1 2] [3 [4 5] 6]])
;=> [1 2 3 [4 5] 6]

(join (list (list 1 2) (list 3 (list 4 5) 6)))
;=> (list 1 2 3 (list 4 5) 6)

(join (seq (list (list 1 2) (list 3 (list 4 5) 6))))
;=> (seq (list 1 2 3 (list 4 5) 6))

(join (lazy-seq (list (list 1 2) (list 3 (list 4 5) 6))))
;=> (lazy-seq (list 1 2 3 (list 4 5) 6))

(join #{#{1 2} #{3 #{4 5} 6}})
;=> #{1 2 3 #{4 5} 6}

(join {:a 1 :b {:c 2 :d {:e 3}}})
;=> {:a 1 [:b :c] 2 [:b :d] {:e 3}}

(join (first {:a (first {:b 1})}))
;=> (first {[:a :b] (first {:c 1})})
```

Magma

`op` is a pretty straightforward for Clojure core data structures - it is similar to the `concat` function. The difference is that `concat` turns everything into lazy sequences while `op` preserves the type of the data structure. `op` is associative for all Clojure core data structures, so they form semigroups.

```
(op [1 2 3] [4 5 6])
;=> [1 2 3 4 5 6]

(op [1 2 3] [4 5 6] [7 8 9] [10 11 12])
;=> [1 2 3 4 5 6 7 8 9 10 11 12]

(op (list 1 2 3) (list 4 5 6))
;=> (list 1 2 3 4 5 6)

(op (list 1 2 3) (list 4 5 6) (list 7 8 9) (list 10 11 12))
;=> (list 1 2 3 4 5 6 7 8 9 10 11 12)

(op (lazy-seq [1 2 3]) (lazy-seq [4 5 6]))
;=> (lazy-seq [1 2 3 4 5 6])

(op (lazy-seq [1 2 3]) (lazy-seq [4 5 6])
    (lazy-seq [7 8 9]) (lazy-seq [10 11 12]))
;=> (lazy-seq [1 2 3 4 5 6 7 8 9 10 11 12])

(op (seq [1 2 3]) (seq [4 5 6]))
;=> (seq [1 2 3 4 5 6])

(op (seq [1 2 3]) (seq [4 5 6])
    (seq [7 8 9]) (seq [10 11 12]))
;=> (seq [1 2 3 4 5 6 7 8 9 10 11 12])

(op #{1 2 3 6} #{4 5 6})
;=> #{1 2 3 4 5 6}

(op #{1 2 3 6} #{4 5 6} #{7 8 9} #{10 11 12})
;=> #{1 2 3 4 5 6 7 8 9 10 11 12}

(op {:a 1 :b 2} {:a 3 :c 4})
;=> {:a 3 :b 2 :c 4}

(op {:a 1 :b 2} {:a 3 :c 4} {:d 5} {:e 6})
;=> {:a 3 :b 2 :c 4 :d 5 :e 6}

(op (first {:a 1}) (first {:b 2}))
;=> (first {:ab 3})

(op (first {:a 1}) (first {:b 2}) (first {:b 3}))
;=> (first {:abb 6})
```

Monoid

`id` for Clojure's data structures' `op` is an empty structure of the same type as the `id`'s argument:

```
(id [2])
;=> []

(id (list 4 5 6))
;=> (list)

(id (seq [1 2]))
;=> (empty (seq [2]))

(id (lazy-seq [1 23]))
;=> (lazy-seq [])

(id #{2 3})
;=> #{}

(id {:1 2})
;=> {}

(id (first {:a 1}))
;=> [(keyword "") 0]
```

Foldable

The `fold` function aggregates the content of the core data structures into one value. All elements in the data structure must belong to the same monoid, i.e. they have the same type that implements Monoid protocol, or of types compatible with the first element's type for the Magma's `op` function. It is similar to Clojure's reduce, but does not require the reduction function (since Monoid's `op` is used).

```
(fold [1 2 3 4 5 6])
=> 21

(fold (list "a" "b" "c"))
;=> "abc"

(fold (seq [:a :b :c]))
;=> :abc

(fold (lazy-seq [[1] (list 2) (seq [3]])))
;=> [1 2 3]

(fold #{1 2 3})
;=> 6

(fold {:a 1 :b 2 :c 3})
;=> 6

(fold (first {:a 1}))
;=> 1
```

The `fold` function does not have to use `op` for folding (reducing). If you supply the function and an init value, it works as you'd expect from `reduce` .

```
(fold + 3 [1 2 3])
;=> 9
```

`fold` can handle multiple foldable data structures. In that case, it works as Clojure's `reduce` on steroids - the folding (reduction) function is applied to the result of applying `op` to the corresponding elements of supplied collections.

```
(fold + 0 [1 2 3] [4 5 6])
;=> 21

(fold * 1 [1 2 3] [4 5 6])
;=> 315

(fold * 2 [1 2 3] [4 5 6] [7 8 9] [1 2 1] [3 2 1])
;=> 12160
```

Sometimes the collection(s) that we would like to fold hold elements that are not Magmas - they do not have `op`. In that case, we can fold them with `foldmap`, which is a variant of `fold` that accept an additional function, which is applied to the elements of the collection being folded before folding. Of course, foldmap is useful even when elements have `op` - when we need to use some alternative function instead of `op`.

```
(foldmap str [1 2 3 4 5 6])
;=> "123456"

(foldmap (fn [^String s] (.toUpperCase s))
         (list "a" "b" "c"))
;=> "ABC"

(foldmap str (seq [:a :b :c]))
;=> ":a:b:c"

(foldmap str {:a 1 :b 2 :c 3})
;=> "123"

(foldmap str (first {:a 1}))
;=> "1"

(foldmap + 3 inc [1 2 3])
;=> 12

(foldmap * 5 / [1 2 3] [4 5 6])
;=> 1/4

(foldmap * 2 / [1 2 3] [4 5 6] [7 8 9] [1 2 1] [3 2 1])
;=> 1/60480
```

Objects

Each Java object is a trivial implementation of `Functor` and `Foldable`, where the object is its own trivial context, unless there is a more specific implementation.

Functor

`fmap` simply applies the provided function to the object.

```
(fmap str (Object.))
;=> (str (Object.))
```

Foldable

`fold` returns the object itself:

```
(fold (Object.))
;=> (Object.)
```

String

String is an implementation of several Fluokitten protocols. The implementations help with many string functions that treat strings as sequences, so they assume that the string is in the sequence context.

Functor

`fmap` applies the function on the string, and then makes sure that the result is converted to string.

```
(fmap reverse "loWercase")
;=> "esacreWol"
```

Magma

`op` is equivalent to `str` function.

```
(op "some" "thing" " " "else")
;=> "something else"
```

Monoid

`id` returns an empty string.

```
(id "something")
;=> ""
```

Foldable

`fold` returns the string itself, as with any plain object.

```
(fold "something")
;=> "something"
```

Keyword

Keywords are functors, magmas, monoids, and foldables.

Functor

`fmap` applies the function to the keyword's name, and then coerces the result to a keyword.

```
(fmap reverse :something)
;=> :gnihtemos
```

Magma

`op` aggregates names of all its keyword arguments to a new keyword.

```
(op :some :thing :else)
;=> :somethingelse
```

Monoid

`id` is an empty keyword.

```
(id :something)
;=> (keyword "")
```

Foldable

`fold` returns the keyword itself, as with plain objects.

```
(fold :something)
;=> :something
```

Numbers

Numbers are functors, magmas, monoids and foldables.

Functor

`fmap` works as with any plain object, by applying the function directly to the argument(s).

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

Magma

`op` for numbers is the addition (+) function.

```
(op 1 2 3)
;=> 6
```

Monoid

`id` for addition is `0` .

```
(id 4)
;=> 0
```

Foldable

`fold` simply returns its argument.

```
(fold 5)
;=> 5
```

Functions

In Haskell, functions are functors, monads, and many other categorical types. However, with Clojure functions, there is one big caveat: ordinary Clojure functions are not automatically curried (https://en.wikipedia.org/wiki/Currying), so the well known implementations of most categorical types, when translated to Clojure, do not obey all laws that monads, applicatives, etc. have to observe. Fortunately, Fluokitten provides the `curry` function that creates a curried version of any Clojure function.

While both ordinary Clojure functions and those curried by `curry` implement Fluokitten protocols, and can be used with all appropriate functions from Fluokitten core, only curried functions satisfy all appropriate laws. Most of the time, you can use fluokitten on ordinary functions regardless. However, be warned that those are not “kosher” and “halal” from the point of view of Haskellers, and may not be able to give the full benefit of “monadic” programming. The most obvious case would be if you translate examples that rely on Haskell’s currying.

Functor

Both plain and curried functions implement the `Functor` protocol. `fmap` composes its function arguments:

```
((fmap inc +) 1 2 3)
;=> ((comp inc +) 1 2 3)
```

What's more, when used with multiple arguments, fmap offers a more advanced composition unavailable in Clojure:

```
((fmap + * /) 1 2 3)
;=> (+ (* 1 2 3) (/ 1 2 3))
```

Curried functions work in the following way, compared to plain functions:

```
(inc)
;=> clojure.lang.ArityException

((curry inc))
;=> a curried inc function
```

In the following example, we create two functions by using `fmap`. `inc+` is a plain Clojure function created by applying the `inc` function to the result of applying the `+` function, in context. It first sums its arguments and then increments the sum. `cinc+` does the same thing, but if called with insufficient number of arguments (two in this example, may be created with any arity) it fixes the provided arguments and creates a new curried function that expects the missing ones. This is in contrast with the plain function, which either accepts any number of arguments, or throws a Clojure compiler exception if called with less than minimal number of arguments.

```
(let [inc+ (fmap inc +)
      cinc+ (fmap inc (curry +))]

   (inc+ 1 2 3)
   ;=> 7

   (inc+ 1)
   ;=> 2

   (cinc+ 1 2 3)
   ;=> 7

   ((cinc+ 1) 2)
   ;=> 4

   ((inc+ 1) 2)
   ;=> (throws ClassCastException))
```

Called with multiple arguments, `fmap` creates appropriate compositions, depending of whether it is called with plain or curried argumens:

```
(let [inc*3+2 (fmap inc (partial * 3) (partial + 2))
      cinc*3+2 (fmap inc ((curry *) 3) ((curry +) 2))]

   (inc*3+2 7 3 1)
   ;=> 40

   (cinc*3+2 7 3 1)
   ;=> 40

   ((inc*3+2) 2)
   ;=> (throws ClassCastException)

   ((cinc*3+2) 2)
   ;=> 13
```

Applicative

`pure` creates a function that ignores its arguments and returns the content of its context.

```
(((pure identity inc)) 1)
;=> 2

((pure curried c+) 13 2)
;=> c+
```

The behavior of funciton as Applicatives is a bit tricky to understand, so we give a simple example that is clear once you understand how `fapply` works with (curried) functions, but if this is the first time you are reading about this, we recommend that you check out a chapter 11 from Learn You a Haskell for Great Good (/articles/learnyouahaskell.html), which is dedicated to this topic.

```
((<*> (pure identity inc) (pure identity 1)))
;=> 2

(((pure curried inc) 100) 1)
;=> 2

((fapply (fapply (pure curried c+) (c+ 3)) (c* 100)) 6)
;=> 609

((<*> (pure curried c+) (c+ 3) (c* 100)) 6)
;=> 609
```

Monad

As in the case of applicatives, the behavior of bind, you need some experience to understand how it works with (curried) functions, so we recommend that you check out Learn You a Haskell for Great Good (/articles/learnyouahaskell.html), and just give an example here for the reference.

```
((bind (curry +) (curry *)) 3 4)
;=> 84
```

Magma

Both plain and curried functions are magmas - `op` composes its arguments.

Monoid

`id` for plain functions is the `identity` function, while for curried functions it is curried identity function `cidentity`.

Foldable

Plain functions are foldable just as any object is - `fold` returns its argument. With curried functions, `fold` returns the plain function that was curried by the `curry` function.

References

Clojure synchronous references (atoms and refs) implement all categorical protocols. The reference is the (mutable!) context that hold an immutable value. Keep in mind that in this case, the preferred functions that work on the content are mutable (`fmap!`, `fapply!` etc.) , since they change the reference, instead of producing a new one (`fmap` etc.).

First, we describe regular `fmap`, `fapply` etc., but keep in mind that these do not make much sense with references, since the point of references is that they should mutate instead of producing new instances.

Functor

With refs and atoms, `fmap` is analogous to `alter` and `swap!`. The main difference from these, reference specific methods is that it returns the updated reference (context) itself, not the new value, consistently with other `fmap` implementations described earlier.

```
(fmap inc (atom 3))
;=> (atom 4)

(dosync (fmap inc (ref 5)))
;=> (ref 6)

(fmap + (atom 3) (atom 4) (ref 5))
;=> (atom 12)

(dosync (fmap + (ref 5) (atom 6) (ref 7)))
;=> (ref 18))
```

Applicative

`pure` creates new minimal reference context:

```
(pure (atom 8) 1)
;=> (atom 1)

(pure (ref 8) 2)
;=> (ref 2)
```

`fapply` expects a function and its argument to be inside references, extracts their contents, and updates the first value reference with the result of applying the function to the values.

```
(fapply (atom inc) (atom 1))
;=> (atom 2)

(dosync (fapply (ref inc) (ref 2)))
;=> (ref 3)

(fapply (atom +) (atom 1) (ref 2) (atom 3))
;=> (atom 6)

(dosync (fapply (ref +) (ref 1) (atom 2) (atom 3)))
;=> (check-eq (ref 6))
```

Monad

`bind` accepts variable number of arguments of which all are references except the last, which is a function that takes plain values and produces a reference. `bind` updates the first reference using this function.

```
(bind (atom 8) increment)
;=> (atom 9)

(dosync (bind (ref 8) increment))
;=> (ref 9)

(bind (atom 8) (ref 9) (atom 10) add)
;=> (atom 27)

(dosync (bind (ref 18) (ref 19) (atom 20) add))
;=> (ref 57)
```

`join` flattens the nested reference one level deep.

```
(join (atom (ref (atom 33))))
;=> (atom (atom 33))

(dosync (join (ref (ref (atom 33)))))
;=> (ref (atom 33))
```

Magma

Both refs and atoms form a semigroup with `op` working on their contents, as in the following example:

```
(op (atom 3) (atom 4))
;=> (atom 7)

(op (ref "some") (ref "thing"))
;=> (ref "something")

(op (atom 1) (ref 2) (atom 3))
;=> (atom 6)
```

Monoid

`id` produces a reference that holds the identity element of the monoid provided in the prototype argument:

```
(id (atom nil))
;=> (atom nil)

(id (ref 8))
;=> (ref 0)

(id (ref "something"))
;=> (ref "")
```

Foldable

`fold` extracts the content of the reference.

```
(fold (atom "something"))
;=> "something"

(fold (ref 2))
;=> 2
```

Mutable functions

```
(let [a (atom 3)]
  (fmap! inc a) => a)

(let [r (ref 5)]
  (dosync
   (fmap! inc r) => r))

(fmap! + (atom 3) (atom 4) (ref 5)) => (check-eq (atom 12))

(dosync
 (fmap! + (ref 5) (atom 6) (ref 7)) => (check-eq (ref 18)))

(fapply! (atom inc) (atom 1)) => (check-eq (atom 2))

(dosync (fapply! (ref inc) (ref 2))) => (check-eq (ref 3))

(fapply! (atom +) (atom 1) (ref 2) (atom 3)) => (check-eq (atom 6))

(dosync (fapply! (ref +) (ref 1) (atom 2) (atom 3)))
=> (check-eq (ref 6))

(join! (atom (ref (atom 33)))) => (check-eq (atom (atom 33)))

(dosync (join! (ref (ref (atom 33)))))
=> (check-eq (ref (atom 33)))

(bind! (atom 8) increment) => (check-eq (atom 9))

(dosync (bind! (ref 8) increment))
=> (check-eq (ref 9))

(bind! (atom 8) (ref 9) (atom 10) add)
=> (check-eq (atom 27))

(dosync (bind! (ref 18) (ref 19) (atom 20) add))
=> (check-eq (ref 57)
```

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).