Cats Documentation

Andrey Antukh & Alejandro Gómez
2.2.0
Table of Contents

Introduction

Category Theory and algebraic abstractions for Clojure.

Rationale

The main motivations for writing this library are:

- The existing libraries do not have support for ClojureScript.
- We do not intend to write a little Haskell inside Clojure. We have adopted a practical and Clojure like pragmatic approach, always with correctness in mind.
- We do not like viral/copyleft like licenses and in contrast to other libraries cats is licensed under the BSD (2 clauses) license.
- We do not intend to only implement monads. Other category theory and algebraic abstractions are also first class in cats.

Install

The simplest way to use *cats* in a Clojure project is by including it as a dependency in your ***project.clj***:

```
[funcool/cats "2.2.0"]
```

And it works with the following platforms: **jdk7**, **jdk8**, **node** (4.2.0, 5.10.1 and 6.2.0).

User Guide

This section introduces almost all the category theory and algebraic abstractions that the *cats* library supports.

We will use *Maybe* for the example snippets, because it has support for all the abstractions and is very easy to understand. You can read more about it in the next section of this documentation.

Semigroup

A semigroup is an algebraic structure with an associative binary operation (`mappend`). Most of the builtin collections form a semigroup because their associative binary operation is analogous to Clojure's `into`.

```
(require '[cats.core :as m])
(require '[cats.builtin])

(m/mappend [1 2 3] [4 5 6])
;; => [1 2 3 4 5 6]
```

Given that the values it contains form a Semigroup, we can `mappend` multiple *Maybe* values.

```
(require '[cats.core :as m])
(require '[cats.builtin])
(require '[cats.monad.maybe :as maybe])

(m/mappend (maybe/just [1 2 3])
           (maybe/just [4 5 6]))
;; => #<Just [1 2 3 4 5 6]>
```

Monoid

A Monoid is a Semigroup with an identity element (`mempty`). For the collection types the `mempty` function is analogous to Clojure's `empty`.

Given that the values it contains form a Semigroup, we can `mappend` multiple *Maybe*, with Nothing being the identity element.

```
(require '[cats.core :as m])
(require '[cats.builtin])
(require '[cats.monad.maybe :as maybe])

(m/mappend (maybe/just [1 2 3])
           (maybe/nothing)
           (maybe/just [4 5 6])
           (maybe/nothing))
;; => #<Just [1 2 3 4 5 6]>
```

Functor

Let's dive into the functor. The Functor represents some sort of "computational context", and the abstraction consists of one unique function: **fmap**.

Signature of **fmap** function

```
(fmap [f fv])
```

The higher-order function **fmap** takes a plain function as the first parameter and a value wrapped in a functor context as the second parameter. It extracts the inner value, applies the function to it and returns the result wrapped in same type as the second parameter.

But what is the **functor context**? It sounds more complex than it is. A Functor wrapper is any type that acts as "Box" and implements the `Context` and `Functor` protocols.

One good example of a functor is the **Maybe** type:

```
(require '[cats.monad.maybe :as maybe])

(maybe/just 2)
;; => #<Just 2>
```

The `just` function is a constructor of the Just type that is part of the Maybe monad.

Let's see one example of using **fmap** over a **just** instance:

Example using fmap over **just** instance.

```
(require '[cats.core :as m])
```

```
(m/fmap inc (maybe/just 1))
;; => #<Just 2>
```

The **Maybe** type also has another constructor: `nothing`. It represents the absence of a value. It is a safe substitute for `nil` and may represent failure.

Let’s see what happens if we perform the same operation as the previous example over a **nothing** instance:

Example using fmap over **nothing**.

```
(m/fmap inc (nothing))
;; => #<Nothing>
```

Oh, awesome, instead of raising a `NullPointerException`, it just returns **nothing**. Another advantage of using the functor abstraction, is that it always returns a result of the same type as its second argument.

Let’s see an example of applying fmap over a Clojure vector:

Example using fmap over **vector**.

```
(require '[cats.builtin])

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

The main difference compared to the previous example with Clojure’s map function, is that map returns lazy seqs no matter what collection we pass to it:

```
(type (map inc [1 2 3]))
;; => clojure.lang.LazySeq (cljs.core/LazySeq in ClojureScript)
```

But why can we pass vectors to the `fmap` function? Because some Clojure container types like vectors, lists and sets, also implement the functor abstraction. See the section on built-in types for more information.

Applicative

Let’s continue with applicative functors. The Applicative Functor represents some sort of "computational context" like a plain Functor, but with the ability to execute a function wrapped in the same context.

The Applicative Functor abstraction consists of two functions: **fapply** and **pure**.

Signature of **fapply** function

```
(fapply [af av])
```

Note the **pure** function will be explained later.

The use case for Applicative Functors is roughly the same as for plain Functors: safe evaluation of some computation in a context.

Let’s see an example to better understand the differences between functor and applicative functor:

Imagine you have some factory function that, depending on the language, returns a greeter function, and you only support a few languages.

```
(defn make-greeter
  [^String lang]
  (condp = lang
    "es" (fn [name] (str "Hola " name))
```

```
    "en" (fn [name] (str "Hello " name))
    nil))
```

Now, before using the resulting greeter you should always defensively check if the returned greeter is a valid function or a nil value.

Let’s convert this factory to use the Maybe type:

```
(defn make-greeter
  [^String lang]
  (condp = lang
    "es" (just (fn [name] (str "Hola " name)))
    "en" (just (fn [name] (str "Hello " name)))
    (nothing)))
```

As you can see, this version of the factory differs only slightly from the original implementation. And this tiny change gives you a new superpower: you can apply the returned greeter to any value without a defensive nil check:

```
(fapply (make-greeter "es") (just "Alex"))
;; => #<Just "Hola Alex">

(fapply (make-greeter "en") (just "Alex"))
;; => #<Just "Hello Alex">

(fapply (make-greeter "it") (just "Alex"))
;; => #<Nothing>
```

Moreover, the applicative functor comes with the **pure** function, which allows you to put some value in side-effect-free context of the current type.

Examples:

```
(require '[cats.monad.maybe :as maybe])

(pure maybe/maybe-monad 5)
;; => #<Just 5>
```

If you do not understand the purpose of the **pure** function, the next sections should clarify its purpose.

Foldable

The **Foldable** is a generic abstraction for data structures that can be folded. It consists mainly on two functions: `foldl` and `foldr`. `foldl` is also known as `reduce` or `inject` in other mainstream programming languages.

Both function have an identical signature and differ in how they traverse the data structure. Let’s look at a little example using `foldl`:

```
(foldl (fn [acc v] (+ acc v)) 0 [1 2 3 4 5])
;; => 15
```

You can observe that `foldl` is identical to the clojure `reduce` function:

```
(reduce (fn [acc v] (+ acc v)) 0 [1 2 3 4 5])
;; => 15
```

And the same operation can be done using `foldr`:

```
(foldr (fn [v wc] (+ v wc)) 0 [1 2 3 4 5])
;; => 15
```

The main difference between `foldl` and `reduce` is that `foldl` has a fixed arity so all parameters are mandatory and `foldl` is a generic abstraction that can work with other types apart from collections.

As we said previously, the `foldl` and `foldr` differ mainly on how they traverse the data structure. Then, for understanding better how they work internally, let's see a graphical representation of the `foldl` execution model:

```
((((acc⊕1)⊕2)⊕3)⊕4)⊕5
```

In contrast to the `foldr` internal execution model that looks like that:

```
1⊕(2⊕(3⊕(4⊕(5⊕(wc)))))
```

In languages with strict argument evaluation, `foldr` does not have many applications because when the data structure to fold grows it tends to consume all the stack (causing the well known stack overflow). In case of Clojure, the unique obvious case of using foldr is for small datastructures.

```
(m/foldr #(cons (inc %1) %2) '() (range 100000))
;; => StackOverflowError
```

The **Foldable** abstraction is already implemented for Clojure vectors, lazy seqs and ranges plus the cats maybe, either and validation types. Let see an example how it behaves with maybe:

```
(m/foldl #(m/return (+ %1 %2)) 1 (maybe/just 1))
;; => #<Just 2>

(m/foldl #(m/return (+ %1 %2)) 1 (maybe/nothing))
;; => 1
```

It there also other fold functions that are implemented in terms of the basic `foldl` or `foldr` that can be **foldm** and **foldmap**. At this moment, cats comes only with **foldm**.

The **foldm** function in analgous to the `foldl` in terms of how it does the fold operation, with the difference that is aware of the monad context. Or in other terms, it works with reducing function that return monad types.

Let see an example:

```
(defn m-div
  [x y]
  (if (zero? y)
    (maybe/nothing)
    (maybe/just (/ x y))))

(m/foldm m-div 1 [1 2 3])
;; => #<Just 1/6>


(m/foldm m-div 1 [1 0 3])
;; => #<Nothing>
```

Traversable

The **Traversable** is a generic abstraction for data structures that can be traversed from left to right, running an Applicative action for each element. Traversables must also be Functors and Foldables.

Note that, since Traversables use the Applicative's `pure` operation, the context of the applicative must be set when using the `traverse` function.

Let's look at an example: we have a vector with numbers that we want to map to a Maybe value, and we want to aggregate the result in a Maybe. If any of the actions fails (is Nothing) the resulting aggregate will be Nothing, but if all succeed we preserve the vector's structure inside a Just value.

First of all, we define the function that will transform a number to a Maybe. Our function will wrap the value in a Just if it's even and in a Nothing if it's not:

```
(require '[cats.monad.maybe :as maybe])

(defn just-if-even
  [n]
  (if (even? n)
    (maybe/just n)
    (maybe/nothing)))
```

Now that we have a function that maps a value to the Maybe Applicative, we can traverse a vector of numbers and aggregate a Maybe value. The applicatives will be evaluated from left to right using the applicative's `fapply`.

```
(require '[cats.core :as m])
(require '[cats.context :as ctx])

(ctx/with-context maybe/context
  (m/traverse just-if-even []))
;; => #<Just []>

(ctx/with-context maybe/context
  (m/traverse just-if-even [2 4]))
;; => #<Just [2 4]>

(ctx/with-context maybe/context
  (m/traverse just-if-even [1 2]))
;; => #<Nothing>

(ctx/with-context maybe/context
  (m/traverse just-if-even [2 3]))
;; => #<Nothing>
```

Monad

Monads are the most discussed programming concept to come from category theory. Like functors and applicatives, monads deal with data in contexts.

Additionally, monads can also transform contexts by unwrapping data, applying functions to it and putting new values in a completely different context.

The monad abstraction consists of two functions: **bind** and **return**

Bind function signature.

```
(bind [mv f])
```

As you can see, bind works much like a Functor but with inverted arguments. The main difference is that in a monad, the function is responsible for wrapping a returned value in a context.

Example usage of the bind higher-order function.

```
(m/bind (maybe/just 1)
        (fn [v] (maybe/just (inc v))))
;; => #<Just 2>
```

One of the key features of the bind function is that any computation executed within the context of bind (monad) knows the context type implicitly. With this, if you apply some computation over some monadic value and you want to return the result in the same container context but don't know what that container is, you can use `return` or `pure` functions:

Usage of return function in bind context.

```
(m/bind (maybe/just 1)
        (fn [v]
          (m/return (inc v))))
;; => #<Just 2>
```

The `return` or `pure` functions, when called with one argument, try to use the dynamic scope context value that's set internally by the `bind` function. Therefore, you can't use them with one argument outside of a `bind` context.

We now can compose any number of computations using monad **bind** functions. But observe what happens when the number of computations increases:

Composability example of bind function.

```
(m/bind (maybe/just 1)
        (fn [a]
          (m/bind (maybe/just (inc a))
                  (fn [b]
                    (m/return (* b 2))))))
```

This can quickly lead to callback hell. To solve this, *cats* comes with a powerful macro: **mlet**

Previous example but using **mlet** macro.

```
(m/mlet [a (maybe/just 1)
         b (maybe/just (inc a))]
  (m/return (* b 2)))
```

MonadZero

Some monads also have the notion of an identity element analogous to that of Monoid. When calling `bind` on a identity element for a monad, the same value is returned. This means that whenever we encounter the identity element in a monadic composition it will short-circuit.

For the already familiar Maybe type the identity element is Nothing:

```
(require '[cats.core :as m])
(require '[cats.monad.maybe :as maybe])

(m/mzero maybe/maybe-monad)
;; => #<Nothing>
```

Having an identity element we can make a monadic composition short-circuit using a predicate:

```
(require '[cats.core :as m])
(require '[cats.monad.maybe :as maybe])

(m/bind (maybe/just 1)
        (fn [a]
          (m/bind (if (= a 2)
                    (m/return nil)
                    (m/mzero))
                  (fn [_]
```

```
                         (m/return (* a 2))))))
;; => #<Nothing>
```

As you can see in the above example the predicate `(= a 2)` returns either a monadic value `(m/return nil)` or the identity value for the maybe monad. This can be captured in a function, which is available in `cats.core` namespace:

```
(defn guard
  [b]
  (if b
    (return nil)
    (mzero)))
```

The above example could be rewritten as:

```
(require '[cats.core :as m])
(require '[cats.monad.maybe :as maybe])

(m/bind (maybe/just 1)
        (fn [a]
          (m/bind (m/guard (= a 2))
                  (fn [_]
                    (m/return (* a 2))))))
;; => #<Nothing>
```

Or, using mlet:

```
(require '[cats.core :as m])
(require '[cats.monad.maybe :as maybe])

(m/mlet [a (maybe/just 1)
         :when (= a 2)]
   (m/return (* a 2)))
;; => #<Nothing>
```

MonadPlus

MonadPlus is a complementary abstraction for Monads that support an associative binary operation, analogous to that of a Semigroup. If the monad implements the MonadZero and MonadPlus protocols it forms a monoid.

For the Maybe type, `mplus` acts similarly to a logical OR that treats `Nothing` values as falsey.

```
(require '[cats.core :as m])
(require '[cats.monad.maybe :as maybe])

(m/mplus (maybe/nothing))
;; => #<Nothing>

(m/mplus (maybe/nothing) (maybe/just 1))
;; => #<Just 1>

(m/mplus (maybe/just 1) (maybe/just 2))
;; => #<Just 1>
```

Types

This section will take a tour over the types exposed in cats library and explain how they can be used in the previously explained abstractions.

Maybe

This is one of the two most used monad types (also known as Optional in other programming languages).

The Maybe monad represents encapsulation of an optional value; e.g. it is used as the return type of functions which may or may not return a meaningful value when they are applied. It consists of either an empty constructor (called None or Nothing), or a constructor encapsulating the original data type A (e.g. Just A or Some A).

cats, implements two types:

- `Just` that represents a value in a context.
- `Nothing` that represents the abscense of value.

Example creating instances of `Just` and `Nothing` types:

```
(maybe/just 1)
;; => #<Just 1>

(maybe/nothing)
;; => #<Nothing>
```

There are other useful functions for working with maybe monad types in the same namespace. See the API documentation for a full list of them. But here we will explain a little relevant subset of them.

We mentioned above that **fmap** extracts the value from a functor context. You will also want to extract values wrapped by **just** and you can do that with **from-maybe**.

As we said previously, the Just or Nothing instances act like wrappers and in some circumstances you will want extract the plain value from them. cats offers the `from-maybe` function for that.

Example using **from-maybe** to extract values wrapped by **just**.

```
(maybe/from-maybe (maybe/just 1))
;; => 1

(maybe/from-maybe (maybe/nothing))
;; => nil

(maybe/from-maybe (maybe/nothing) 42)
;; => 42
```

The `from-maybe` function is a specialized version of a more generic one: `cats.core/extract`. The generic version is a polymorphic function and will also work with different types of different monads.

For interoperability with Clojure and ClojureScript's `IDeref` abstraction, maybe values are derrefable.

Example using **deref** to extract values wrapped by **just**.

```
(deref (maybe/just 1))
;; => 1

(deref (maybe/nothing))
;; => nil
```

Either

Either is another type that represents a result of a computation, but (in contrast with maybe) it can return some data with a failed computation result.

In *cats* it has two constructors:

- `(left v)`: represents a failure.
- `(right v)`: represents a successful result.

Usage example of **Either** constructors.

```
(require '[cats.monad.either :refer :all])

(right :valid-value)
;; => #<Right [:valid-value :right]>

(left "Error message")
;; => #<Either [Error message :left]>
```

Note Either is also (like Maybe) a Functor, Applicative Functor and Monad.

Like Maybe, Either values can be dereferenced returning the value they contain.

Exception

Also known as the Try monad, as popularized by Scala.

It represents a computation that may either result in an exception or return a successfully computed value. Is very similar to the Either monad, but is semantically different.

It consists of two types: Success and Failure. The Success type is a simple wrapper, like Right of the Either monad. But the Failure type is slightly different from Left, because it always wraps an instance of Throwable (or any value in cljs since you can throw arbitrary values in the JavaScript host).

The most common use case of this monad is to wrap third party libraries that use standard Exception based error handling. Under normal circumstances, however, you should use Either instead.

It is an analogue of the try-catch block: it replaces try-catch's stack-based error handling with heap-based error handling. Instead of having an exception thrown and having to deal with it immediately in the same thread, it disconnects the error handling and recovery.

Usage example of **try-on** macro.

```
(require '[cats.monad.exception :as exc])

(exc/try-on 1)
;; => #<Success [1]>

(exc/try-on (+ 1 nil))
;; => #<Failure [#<NullPointerException java.lang.NullPointerException>]>
```

cats comes with other syntactic sugar macros: `try-or-else` that returns a default value if a computation fails, and `try-or-recover` that lets you handle the return value when executing a function with the exception as first parameter.

Usage example of `try-or-else` macro.

```
(exc/try-or-else (+ 1 nil) 2)
;; => #<Success [2]>
```

Usage example of `try-or-recover` macro.

```
(exc/try-or-recover (+ 1 nil)
                    (fn [e]
                      (cond
                        (instance? NullPointerException e) 0
                        :else 100)))
;; => #<Success [0]>
```

The types defined for the Exception monad (Success and Failure) also implement the Clojure IDeref interface, which allows library development using monadic composition without forcing a user of that library to use or understand monads.

That is because when you dereference the failure instance, it will reraise the enclosed exception.

Example dereferencing a failure instance

```
(def f (exc/try-on (+ 1 nil)))

@f
;; => NullPointerException   clojure.lang.Numbers.ops (Numbers.java:961)
```

Built in types

Some of the abstractions in *cats* are implemented for built-in types but you can’t use them directly. First, you must load the `cats.builtin` namespace:

```
(require '[cats.builtin])
(require '[cats.core :as m])

(m/fmap inc [1 2 3 4])
;; => [2 3 4 5]
```

nil

Given the fact that `nil` is both a value and a type, we have extended the `nil` type to be equivalent to Maybe monad’s `Nothing`. This means that you can use `nil` as if were a `Just` instance like in the following example:

```
(use 'cats.builtin)
(require '[cats.core :as m])
(require '[cats.monad.maybe :as maybe])

(m/mlet [x (maybe/just 42)
         y nil]
  (m/return (+ x y)))
;; => nil
```

As you can see, the `mlet` short-circuits when encountering a `nil` value.

Vector

Clojure vectors also participate in several of the abstractions implemented in *cats*, most notably as a monad. Compare the following `for` comprehension:

```
(for [x [1 2]
      y [3 4 5]]
```

```
  (+ x y))
;; => (4 5 6 5 6 7)
```

with the equivalent using *mlet*:

```
(use 'cats.builtin)
(require '[cats.core :as m])

(m/mlet [x [1 2]
         y [3 4 5]]
  (m/return (+ x y)))
;; => [4 5 6 5 6 7]
```

Note the symmetry between `for` and `mlet`. This is not accidental, both are what is called a monad comprehension, the difference is that `for` is limited to sequences and `mlet` can work with arbitrary monads.

Also, since `mlet` desugars into calls to the Monad's `bind` function, its result keeps the type of the monadic values.

Lazy sequences

Lazy sequences implement the same abstractions as vectors with practically an identical implementation. If you don't need the results right away or are interested in a subset of the final results, you can use lazy sequence comprehensions.

Using `mlet` with lazy sequences yields exactly the same result as using `for`:

```
(use 'cats.builtin)
(require '[cats.core :as m])

(m/mlet [x (lazy-seq [1 2])
         y (lazy-seq [3 4 5])]
  (m/return (+ x y)))
;; => (4 5 6 5 6 7)
```

Set

Sets implement almost every abstraction in *cats*, from Semigroup to Monad.

```
(use 'cats.builtin)
(require '[cats.core :as m])

(m/pure set-context 42)
;; => #{42}

(m/fmap inc #{1 2 3 4})
;; => #{4 3 2 5}

(m/bind #{1 2 3}
        (fn [v] #{v (inc v)}))
;; => #{1 4 3 2}
```

Map

Maps implement the *Semigroup* protocol, since we can use `merge` as their associative binary operation. Using `mappend` on maps is a way to merge them together:

```
(use 'cats.builtin)
(require '[cats.core :as m])
```

```
(m/mappend {:a "A"} {:b "B"})
;; => {:a "A", :b "B"}
```

Since we can consider the empty map an identity element for the `mappend` associative binary operation maps also implement *Monoid* and the `mempty` function gives an empty map.

Syntax sugar

Additionally to the abstractions and types, **cats** exposes some powerful syntax abstractions that surelly will make the usage of that abstractions in a more familiar way.

mlet

The `mlet` syntactic abstraction intends to facilitate the composition of monadic operations.

If you've followed along with the documentation you've seen many examples of its usage already, let's see what can `mlet` do. First of all, mlet turns this let-like bindings:

```
(m/mlet [a (maybe/just 1)
         b (maybe/just (inc a))]
  (m/return (* a b)))
```

into a chain of calls to bind:

```
(m/bind (maybe/just 1)
        (fn [a]
          (m/bind (maybe/just (inc a))
                  (fn [b]
                    (m/return (* a b))))))
```

That makes a lot more natural to write code that uses monads and gives a very familiar `let` like syntax abstraction that makes the clojure code that uses monads less "strange".

If you are coming from Haskell, mlet is mostly analogous to the **do notation**.

Since the bindings in the mlet macro run the monadic effects of the right-hand values we cannot just put any value in there and expect to be bound to its left symbol. For cases where we want the regular behavior of let we can inline a `:let` clause, just like with Clojure's `for`:

```
(m/mlet [a (maybe/just 1)
         b (maybe/just (inc a))
         :let [z (+ a b)]]
  (m/return (* z 2)))
```

`mlet` has support for using guards using a `:when` clause, analogous to the one used in `for`. We can filter out values using `bind` with `mlet` and `:when` like the following:

```
(require '[cats.core :as m])
(require '[cats.monad.maybe :as maybe])

(m/mlet [a (maybe/just 1)
         :when (= a 2)]
   (m/return (* a 2)))
;; => #<Nothing>
```

Any monadic type that implements `MonadZero` can be combined with guards inside `mlet` bindings. Here is an example with vectors:

```
(require '[cats.builtin]
(require '[cats.core :as m])

(m/mlet [a [1 2 3 4]
         :when (odd? a)]
   (m/return (* a 2)))
;; => [2 6]
```

alet

One limitation of monadic bind is that all the steps are strictly sequential and happen one at a time. This piece of code illustrates the usage of monadic bind:

```
(require '[cats.core :refer [bind return]])
(require '[cats.monad.maybe :refer [just]])

(bind (just 1)
      (fn [a]
        (bind (just 41)
              (fn [b]
                (return (+ a b))))))
;; => #<Just 42>
```

In the first call to `bind`, `(just 1)` and the anonymous function will be evaluated. The call of the anonymous function performed by the first `bind` will cause the evaluation of the `(just 41)` and the next anonymous function, which will be also called to create the final result. Note that `(just 1)` and `(just 41)` are independent and thus could be evaluated at the same time.

Here is the `mlet` version for reference and clarity:

```
(mlet [a (just 1)
       b (just 41)]
  (return (+ a b)))
;; => #<Just 42>
```

Now let’s see the equivalent using `alet`:

```
(require '[cats.core :refer [alet]])

(alet [a (just 1)
       b (just 41)]
  (+ a b))
;; => #<Just 42>
```

Note that no `return` is used, this is because the `alet` body runs inside the applicative context with `fapply`. This is roughly what `alet` desugars to:

```
(fapply (fn [a]
          (fn [b]
            (do
              (+ a b))))
        (just 1)
        (just 41))
;; => #<Just 42>
```

Note that now `(just 1)` and `(just 41)` are evaluated at the same time. This use of `fapply` can be called "applicative bind" and in some cases is more efficient than monadic bind. Furthermore, the `alet` macro splits the bindings into batches that have dependencies only in previous values and evaluates all applicative values in the batch at the same time.

This makes no difference at all for Maybe, but applicatives that have latency in their calculations (for example promises that do an async computation) get a pretty good evaluation strategy, which can minimize overall latency. In the next examples we use the promesa clj/cljs library for emulate asynchronous behavior:

```
(require '[cats.core :as m])
(require '[cats.labs.promise])
(require '[promesa.core :as p])

(defn sleep-promise
  "A simple function that emulates an
  asynchronous operation."
  [wait]
  (p/promise (fn [resolve reject]
               (future
                 (Thread/sleep wait)
                 (resolve wait)))))

;; note: deref-ing for blocking the current thread
;; waiting for the promise being delivered

(time
 @(m/mlet [x (sleep-promise 42)
           y (sleep-promise 41)]
    (m/return (+ x y))))
;; "Elapsed time: 84.328182 msecs"
;; => 83

(time
 @(m/alet [x (sleep-promise 42)
           y (sleep-promise 41)]
    (+ x y)))
;; "Elapsed time: 44.246427 msecs"
;; => 83
```

Another example for illustrating dependencies between batches:

```
(time
 @(m/mlet [x (sleep-promise 42)
           y (sleep-promise 41)
           z (sleep-promise (inc x))
           a (sleep-promise (inc y))]
   (m/return  (+ z a))))
;; "Elapsed time: 194.253182 msecs"
;; => 85

(time
 @(m/alet [x (sleep-promise 42)
           y (sleep-promise 41)
           z (sleep-promise (inc x))
           a (sleep-promise (inc y))]
    (+ z a)))
;; "Elapsed time: 86.20699 msecs"
;; => 85
```

Higher-order functions

curry

The first combinator that *cats* provides is a `curry` macro. Given a function, it can convert it to a curried versions of itself. The generated function will accept parameters until all the expected parameters are given.

Let's see some examples of a curried function in action:

```
(require '[cats.core :as m])

(defn add [a b c]
  (+ a b c))

(def curried-add (m/curry add))

(= curried-add (curried-add))
;; => true

(= (curried-add 1 2 3) 6)
;; => true

(= ((curried-add 1) 2 3) 6)
;; => true

(= ((curried-add 1 2) 3) 6)
;; => true
```

As you can see above, since the original `add` has a single arity (3) and is fixed (i.e. it doesn't accept a variable number of arguments), the `curry` macro was able to generate a curried function with the correct number of parameters.

This doesn't mean that functions with multiple arities or variadic arguments can't be curried but an arity for the curried function must be given:

```
(require '[cats.core :as m])

(def curried+ (m/curry 3 +))

(= curried+ (curried+))
;; => true

(= (curried+ 1 2 3) 6)
;; => true

(= ((curried+ 1) 2 3) 6)
;; => true

(= ((curried+ 1 2) 3) 6)
;; => true
```

Curried functions are very useful in combination with the applicative's `fapply` operation, since we can curry a function and use applicatives for building up results with context-specific effects.

```
(require '[cats.core :as m])
(require '[cats.monad.maybe :refer [just nothing]])

(def curried+ (m/curry 3 +))

(m/fapply (just curried+) (just 1) (just 2) (just 3))
;; => #<Just 6>

(m/fapply (just curried+) (just 1) (just 2) (nothing))
;; => #<Nothing>

(m/fapply (just curried+) (just 1) nil (just 3))
;; => nil

(m/fapply (m/fmap curried+ (just 1)) (just 2) (just 3))
;; => #<Just 6>
```

```
(m/<*> (m/<$> curried+ (just 1)) (just 2) (just 3))
;; => #<Just 6>
```

lift-m

The `lift-m` macro is a combinator for promoting functions that work on regular values to work on monadic values instead. It uses the monad's bind operation under the hood and, like `curry`, can be used without specifying arity if the function we are lifting has a fixed and a single arity:

```
(require '[cats.core :as m])
(require '[cats.monad.maybe :refer [just nothing]])

(defn add [a b c]
  (+ a b c))

(def add-m (m/lift-m add))

(add-m (just 1) (just 2) (just 3))
;; => #<Just 6>

(add-m (just 1) (nothing) (just 3))
; => #<Nothing>

(add-m (just 1) nil (just 3))
;; => nil
```

Like with `curry`, we must provide an arity in case we are lifting a function that has multiple arities or is variadic:

```
(require '[cats.core :as m])
(require '[cats.monad.maybe :refer [just nothing]])

(def add-m (m/lift-m 3 +))

(add-m (just 1) (just 2) (just 3))
;; => #<Just 6>

(add-m (just 1) (nothing) (just 3))
; => #<Nothing>

(add-m (just 1) nil (just 3))
;; => nil
```

Note that you can combine both `curry` and `lift-m` to get curried functions that work on monadic types using the `curry-lift-m` macro. The arity is mandatory when using this macro:

```
(require '[cats.core :as m])
(require '[cats.monad.maybe :refer [just nothing]])

(def curried-add-m (m/curry-lift-m 3 +))

(curried-add-m (just 1) (just 2) (just 3))
;; => #<Just 6>

((curried-add-m (just 1)) (just 2) (just 3))
;; => #<Just 6>

((curried-add-m (just 1) (just 2)) (just 3))
;; => #<Just 6>
```

Labs

This section intends to explain different kind of extra features that can be found under **cats.labs** namespace. The fact that they are here because they are experimental, requires external dependencies or simply does not have much application in clojure(script).

In any case the state of each module will be notified on the start of the each section.

test.check

Status: Experimental

The `cats.labs.test` namespace implements monad and applicative instances for generators, which lets you use the `cats.core/alet` and `cats.core/mlet` macros for writing generators:

```
(require '[cats.core :as m])
(require '[cats.labs.test])
(require '[clojure.test.check.generators :as gen])

(def color
  (m/alet [r gen/int
           g gen/int
           b gen/int]
    [r g b]))

(gen/sample color 1)
;; => ([0 0 0])

(def mcolor
  (m/mlet [r gen/int
           g gen/int
           b gen/int]
    (m/return [r g b])))

(gen/sample mcolor 1)
;; => ([0 0 0])
```

Apart from that, the namespace contains multiple functions for generating test.check properties that verify the laws of Semigroup, Monoid, Functor, Applicative, Monad, MonadZero and MonadPlus.

The implementation of cats' abstractions are tested using generative testing and the `cats.labs.test` property generation functions.

Channel

Status: Experimental

This namespace exposes the ability to use the **core.async** channel as monadic type and in consequence use it in `mlet` or `alet` macros.

Before use it, you should add core.async to your dependencies:

```
[org.clojure/core.async "0.2.385"]
```

Now, let see some code. This will allow you understand how it can be used and why this integration between cats and core.async matters. At first step we will go to define a function that emulates whatever asynchronous task, that for our case it’s consist in a just sleep operation:

```
(require '[clojure.core.async :as a])
(require '[cats.labs.channel])
```

```
(defn async-call
  "A function that emulates some asynchronous call."
  [n]
  (a/go
    (println "---> sending request" n)
    (a/<! (a/timeout n))
    (println "<--- receiving request" n)
    n))
```

Now, instead of using the `go` macro, just use a `let` like bindings with the help of the **mlet** macro for bind values to asyncrhonous calls:

```
(time
 (<!! (m/mlet [x (async-call 200)
               y (async-call 100)]
        (m/return (+ x y)))))
;; ---> sending request 200
;; <--- receiving request 200
;; ---> sending request 100
;; <--- receiving request 100
;; "Elapsed time: 302.236804 msecs"
;; => 300
```

Here we can observe few things:

- The asynchronous calls are made serially.
- We are calling a function that return a channel and bind its value to a symbol.
- At the end, an operation is performed with the `mlet` bindings.
- The `mlet` macro also returns a channel.

The main difference with the default clojure `let`, is that the bindings are already plain values (not channels). The take! operation is already performed automatically by the `mlet`. This kind of behavior will make you fully asynchronous code looks like synchronous code.

But, cats also comes with `alet` that has identical aspect to the previously used `mlet` macro, but it has some advantages over it. Let see an example:

```
(time
  (a/<!! (m/alet [x (async-call 100)
                  y (async-call 100)]
            (+ x y)))))

;; ---> sending request 100
;; ---> sending request 100
;; <--- receiving request 100
;; <--- receiving request 100
;; "Elapsed time: 101.06644 msecs"
;; => 200
```

And here we can observe few things:

- The asynchronous calls are made in parallel.
- The total time of processing is half less of if we use `mlet`.
- The `return` function is not used because `alet` evaluates the body in the context of the applicative.

The alet is a powerful macro that analyzes the dependencies between bindings and executes the expressions in batches resultin in a very atractive feature for asynchronous calls.

Here an other examples that shows in a clearly way how the batches are executed:

```
(time
 (a/<!! (m/alet [x (async-call 120)
                 y (async-call 130)
                 z (async-call (- x 100))
                 u (async-call (- y 100))
                 t (async-call (inc u))]
          z))))
;; ---> sending request  130
;; ---> sending request  120
;; <--- receiving request  120
;; <--- receiving request  130
;; ---> sending request  20
;; ---> sending request  30
;; <--- receiving request  20
;; <--- receiving request  30
;; ---> sending request  31
;; <--- receiving request  31
;; "Elapsed time: 194.536235 msecs"
;; => 20
```

Manifold Deferred

Status: Experimental

This namespace exposes the ability to use the **manifold** deferred as monadic type and in consequence use it in `mlet` or `alet` macros.

Before use it, you should add manifold to your dependencies:

```
[manifold "0.1.1"]
```

Now, let see some code. This will allow you understand how it can be used and why this integration between cats and manifold matters. At first step we will go to define a function that emulates whatever asynchronous task, that for our case it's consist in a just sleep operation:

For demostration purposes, let's define a function that emulates the asyncrhonous call:

```
(require '[cats.labs.manifold :as mf]
         '[manifold.deferred :as d])

(defn async-call
  "A function that emulates some asynchronous call."
  [n]
  (d/future
    (println "---> sending request" n)
    (Thread/sleep n)
    (println "<--- receiving request" n)
    n))
```

Now, the manifold deferreds can participate in the monad/applicative abstractions using `mlet` and `alet` respectivelly.

Example using manifold deferred with `mlet`.

```
(time
  @(m/mlet [x (async-call 200)
```

```
            y (async-call 100)]
     (m/return (+ x y)))))
;; ---> sending request 200
;; <--- receiving request 200
;; ---> sending request 100
;; <--- receiving request 100
;; "Elapsed time: 302.236804 msecs"
;; => 200
```

If you are familiar with manifold's `let-flow` macro, the cats `alet` serves for almost identical purpose, with difference that `alet` is defined as generic abstraction instread of a specific purpose macro.

Example using manifold deferred with `alet`.

```
(time
  @(m/alet [x (async-call 100)
            y (async-call 100)]
     (+ x y)))))

;; ---> sending request 100
;; ---> sending request 100
;; <--- receiving request 100
;; <--- receiving request 100
;; "Elapsed time: 101.06644 msecs"
;; => 200
```

Complementary libraries

- Promise monad: https://github.com/funcool/promesa
- Concurrent data fetching: https://github.com/funcool/urania
- Pattern matching for the Cats' types: https://github.com/zalando/cats.match

FAQ

What Clojure types implement some of the Category Theory abstractions?

In contrast to other similar libraries in Clojure, *cats* doesn't intend to extend Clojure types that don't act like containers. For example, Clojure keywords are values but can not be containers so they should not extend any of the previously explained protocols.

Table 1. Summary of Clojure types and implemented protocols

Name	Implemented protocols
sequence	Semigroup, Monoid, Functor, Applicative, Monad, MonadZero, MonadPlus, Foldable
vector	Semigroup, Monoid, Functor, Applicative, Monad, MonadZero, MonadPlus, Foldable
hash-set	Semigroup, Monoid, Functor, Applicative, Monad, MonadZero, MonadPlus

Name	Implemented protocols
hash-map	Semigroup, Monoid
function	Semigroup, Monoid, Functor, Applicative, Monad

Developers Guide

Philosophy

Five most important rules:

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Readability counts.

All contributions to *cats* should keep these important rules in mind.

Contributing

Unlike Clojure and other Clojure contributed libraries, *cats* does not have many restrictions for contributions. Just open an issue or pull request.

Editor integration

For making Emacs' clojure-mode treat `alet`, `mlet` et al like a `let` and indent them correctly, you can use `define-clojure-indent` like in the following example:

```
(require 'clojure-mode)

(define-clojure-indent
  (alet 'defun)
  (mlet 'defun))
```

Source Code

cats is open source and can be found on github.

You can clone the public repository with this command:

```
git clone https://github.com/funcool/cats
```

Run tests

For running tests just execute this for clojure:

```
lein test
```

And this for clojurescript:

```
./scripts/build
node ./out/tests.js
```

License

Last updated 2018-01-11 09:53:43 CET