netguru

Introduction to Monads in Ruby

Do not fear, Monads are here

Introduction to Monads in Ruby

Do not fear, Monads are here

netguru

Kamil Kula

Ruby on Rails Developer

kamil.kula@netguru.com

netguru

Functors

Functors

Functor is simply something that can be mapped over and returns something else of the same type.

Functors

Ruby's example of Functor is an Array.

```ruby
[1,2,3,4,5].map { |i| i.to_s } # => ['1','2','3','4','5']
```

Mapping over array returns new array with computed values.

```
[Integer] -> [String]
```

In more generic way we can read this as *"apply a function from x to y, to an array of x, which will result in array of y"*

```
[x] -> [y]

(x -> y) -> [x] - [y]
```

Functors

`Hash` in Ruby on the other hand is **not** a `Functor` as it may return other data type.

```ruby
id = -> x { x }

hash = { dog_age: 3, cat_age: 6 }

hash.map(&id) # => [[:dog_age, 3], [:cat_age, 6]]

hash.map(&id) == hash # => false

Hash -> Array
```

This time given `Hash` class `Array` is returned instead

```
Hash -> Array
```

Identity law

Mapping an identity function (function that returns value passed to it) over a Functor should not change the data Functor encapsulates.

In Ruby example of identity function is method itself.

```ruby
[1,2,3].map(&:itself) # => [1,2,3]

Array -> Array # Array is a functor!
```

```ruby
{ dog_age: 3, cat_age: 6 }.map(&:itself) # => [[:dog_age, 3], [:cat_age, 6]]

Hash -> Array # Hash is not a functor
```

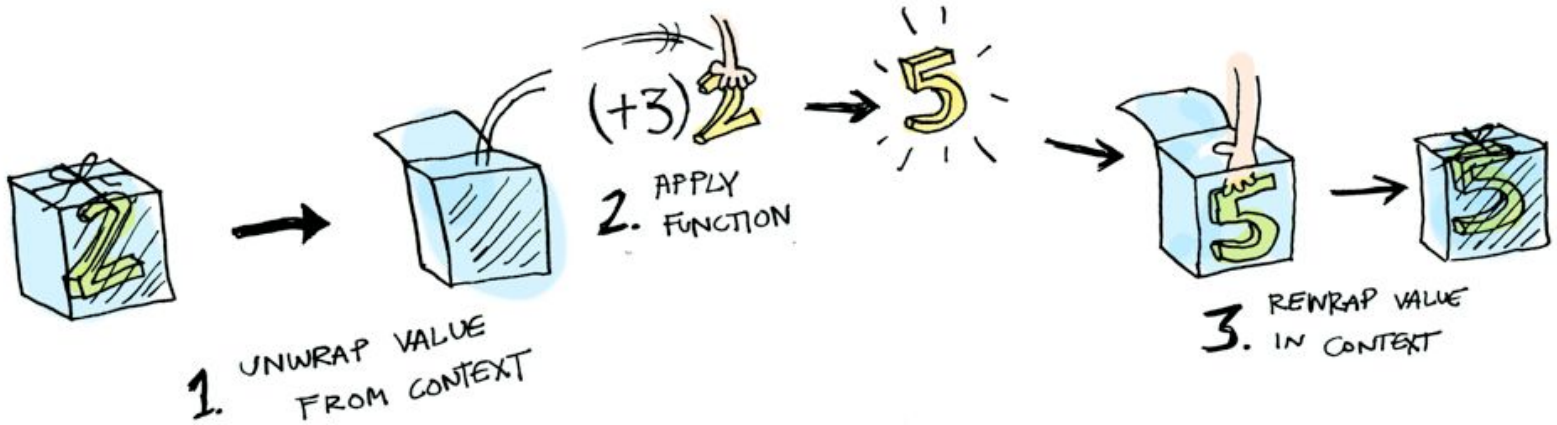
Associative law

Mapping f (g (x)) over a Functor should give the same result as mapping g (x) followed by f (x) over the Functor (function composition).

```ruby
array = [1, 2, 3]

f = -> (x) { x + 1 }
g = -> (x) { x * 2 }

array.map(&f).map(&g)                # => [4, 6, 8]

array.map { |x| g.call(f.call(x)) } # => [4, 6, 8]
array.map { |x| g.(f.(x)) }          # => [4, 6, 8]
```

Functors

All that comes down to with Functors is that they always should stay that same type of Functor.



http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

netguru

Monads

Monads

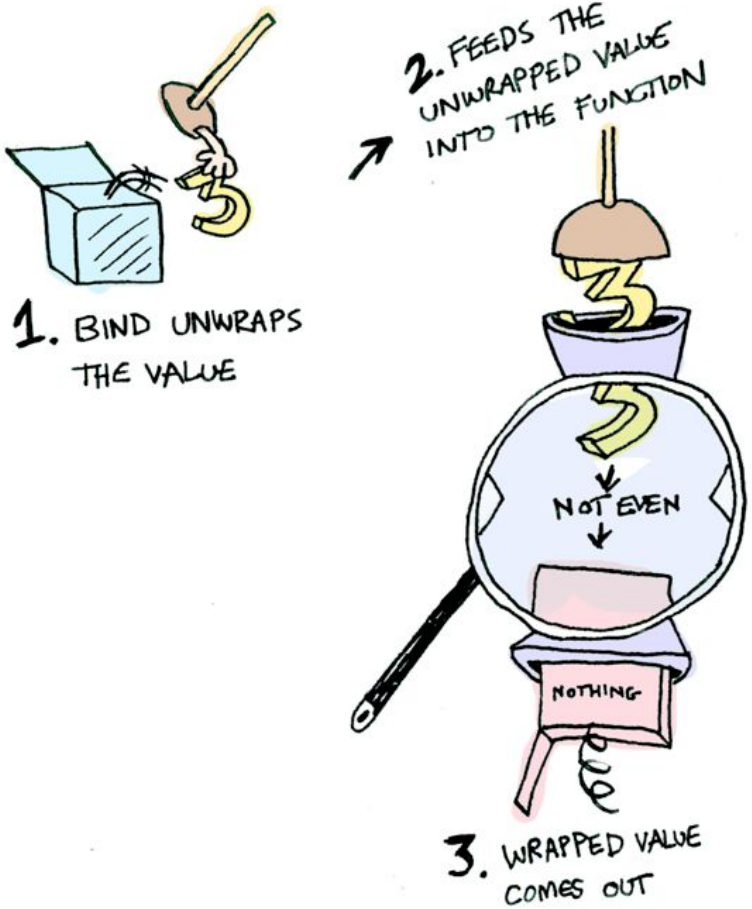
“A monad is just a monoid in the category of endofunctors, what's the problem?” - James Iry

Monads

Monads apply a function that returns a wrapped value to a wrapped value.

Monads have a function >>= (bind) that allows to do that.

Monads



http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

netguru

Maybe Monad

Maybe

Maybe addresses very common pattern where a given method call might return a value or may return a nil.

```
def lowest_number(array)
  sorted = array.sort { |x, y| x <=> y }
  sorted.first
end

[Comparable] -> Maybe Comparable
```

Maybe


```ruby
class Maybe
  def initialize(value)
    @value = value
  end

  # Unwraps the value if it's not nil, does some work and return
  # result wrapped in new context
  def fmap
    return self if @value.nil?
    self.class.new(yield @value)
  end

  # Unwraps the value if not nil, does some work and return the
  # result itself (without context). Block passed to bind have to
  # wrap result in new context
  def bind
    return self if @value.nil?
    yield @value
  end

  # Useful method for irb/pry to print Some/None
  def inspect
    return 'None' if @value.nil?
    'Some ' + @value.inspect
  end
end
```

Maybe

Simple Maybe implementation in action:

```ruby
Maybe.new(1).fmap { |x| x + 1 }   # => Some 2

Maybe.new(1).bind { |x| x + 1 }   # => 2

Maybe.new(nil).bind { |x| x + 1 } # => None

Maybe.new(nil).fmap { |x| x + 1 } # => None
```

Monads

- fmap is similar to map in ruby - unwraps, applies block (function) and wrapps again
- bind like fmap unwraps, applies block but it returns unwrapped value (so other bind cannot be chained on to it)

Monads

In order for a data type to be a Monad it has to obey three "*monad laws*":

- Left identity: `return a >>= f ≡ f a`
- Right identity `m >>= return ≡ m`
- Associativity: `(m >>= f) >>= g ≡ m >>= ( \x -> f x >>= g)`

Left identity

```
return a >>= f ≡ f a
```

Putting a value in the default context (e.g. Maybe) and feeding it to a function is the same as applying the function to the value.

```ruby
f = -> (x) { Maybe(x ** 3) }

Maybe(2).bind(&f) #=> Some(8)

f.(2) #=> Some(8)

Maybe(2).bind(&f) == f.(2) # => true
```

Right identity

```
m >>= return ≡ m
```

If we have a Maybe object and try to bind it to another Maybe, this operation will not change anything

```ruby
Maybe(5).bind(&method(:Maybe)) # => Maybe(5)
```

Associativity

```
(m >>= f) >>= g ≡ m >>= ( \x -> f x >>= g)
```

```ruby
a = Maybe(4)

f = -> (x) { Maybe(x ** 3) }
g = -> (x) { x > 10 ? Maybe(x) : None() }

# (m >>= f) >>= g

(a.bind(&f)).bind(&g) # => Some(64)

# m >>= ( \x -> f x >>= g)

a.bind { |x| f.(x).bind(&g) } # => Some(64)
```

Given a chain of computations it does not matter how they are nested - the outcome must always be the same.

netguru

dry-monads

dry-monads

`dry-monads` is a set of common monads for Ruby that helps execute code in more elegant, free of if/else way.

`dry-monads` are a basis for some other gems implemented by `dry` team.

`gem install dry-monads` (or add `gem 'dry-monads'` to gemfile and bundle)

dry-monads

netguru

Monads implemented by dry-monads gem:

- Maybe
- Result
- Try
- List
- Task

Dry::Monads::Maybe

The `Maybe` monad is used when a series of computations could return `nil` at any point, returning `Some(value)` or `None()`.

Methods:

- `bind`
- `fmap`
- `value!`
- `value_or`
- `or`

Dry::Monads::Maybe


```ruby
require 'dry/monads/maybe'

M = Dry::Monads

maybe_company = M.Maybe(company).bind do |company|
  M.Maybe(company.address).bind do |address|
    M.Maybe(address.street)
  end
end

# If company with address exists
# => Some("Street Address")
# If company or address is nil
# => None()
```

Dry::Monads::Result

The Result monad is used when a series of computations might return an error object with additional data. Result uses two constructors Success(value) and Failure(value).

Methods:

- bind
- fmap
- value!
- value_or
- or
- failure
- failure?
- success
- success?
- to_maybe

Dry::Monads::Result


```ruby
require 'dry/monads/result'

class SimpleCalculator
  include Dry::Monads::Result::Mixin

  attr_accessor :integer


  def calc
    i = Integer(integer)

    Success(i).bind do |val|
      if val >= 0
        Success(val + 1)
      else
        Failure(:value_less_than_zero)
      end
    end.bind do |val|
      if val % 2 != 0
        Success(val * 3)
      else
        Failure(:value_should_be_odd)
      end
    end
  end
end
```

Dry::Monads::Result


```ruby
# instantiate new SimpleCalculator
calculator = SimpleCalculator.new

# full Success path
c.integer = 2
calculator.calc # => Success(9)

# first Failure
c.integer = -1
calculator.calc # => Failure(:value_less_than_zero)

# second Failure
c.integer = 3
calculator.calc # => Failure(:value_should_be_odd)
```

Dry::Monads::Try

The Try monads rescues a block from an exception. Useful to wrap code that might raise an exception like HTTP requests or queries to DB.

Methods:

- bind
- fmap
- value! / value?
- exception / error?
- to_result / to_maybe

Dry::Monads::Try


```ruby
require 'dry/monads/try'

module CatchExceptions
  extend Dry::Monads::Try::Mixin

  res = Try() { 6 / 3 }
  res.value! if res.value?
  # => 2

  res = Try() { 6 / 0 }
  res.exception if res.error?
  # => #<ZeroDivisionError: divided by 0>

  # By default Try catches all exceptions inherited from StandardError.
  # However you can catch only certain exceptions like this
  Try(NoMethodError, NotImplementedError) { 10 / 0 }
  # => raised ZeroDivisionError: divided by 0 exception
end
```

Dry::Monads::Task

The Task monad is used for asynchronous computations. It can be used to wrap side-effectful actions (IO). Internally uses Promise from concurrent-ruby gem.

Methods:

- bind
- fmap
- or / or_fmap
- value!
- wait (optional_timeout)
- to_result / to_maybe

Dry::Monads::Task


```ruby
require 'dry/monads/task'

class GetCompaniesWithEmployees
  include Dry::Monads::Task::Mixin

  def call
    # Start two concurent tasks
    companies = Task { fetch_companies }
    employees = Task { fetch_employees }

    # Combine two tasks
    companies.bind { |comp| employees.fmap { |empl| [comp, empl] } }
  end

  def fetch_companies
    sleep 3
    [{ id: 1, name: 'Solid Inc.' }, { id: 2, name: 'Rigid Inc.' }]
  end

  def fetch_employees
    sleep 2
    [
      { id: 2, employee_id: 2, name: 'Norville' },
      { id: 1, employee_id: 1, name: 'Jake' },
    ]
  end
end
```

Dry::Monads::Task


```ruby
# GetCompaniesWithEmployees instance
get = GetCompaniesWithEmployees.new

# Spin up two tasks
task = get.call

task.fmap do |companies, employees|
  puts "Companies: #{ companies.inspect }"
  puts "Employees: #{ employees.inspect }"
end

# => Task(?)

# => Companies: [{:id=>1, :name=>"Solid Inc."}, {:id=>2, :name=>"Rigid Inc."}]
# => Employees: [{:id=>2, :employee_id=>2, :name=>"Norville"}, {:id=>1, :employee_id=>1, :name=>"Jake"}]
```

Dry::Monads::List

The List monad allows us to wrap multiple values/objects in a list.

Methods:

- bind
- fmap
- value
- concatenation (+)
- head/tail
- traverse

Dry::Monads::List


```ruby
require 'dry/monads/list'

M = Dry::Monads

M::List[3, 4].bind { |x| [x + 1] } # => List[4, 5]
M::List[3, 4].bind(-> x { [x, x + 1] }) # => List[3, 4, 4, 5]

M::List[3, nil].bind { |x| [x + 1] } # => error

M::List[3, 4].fmap { |x| x + 1 } # => List[4, 5]

M::List[3, 4].value # => [3, 4]

M::List[1, 2] + M::List[3, 4] # => List[1, 2, 3, 4]

M::List[1, 2, 3, 4].head # => Some(1)
M::List[1, 2, 3, 4].tail # => List[2, 3, 4]
```

Summary

netguru

“A change of perspective is worth 80 IQ points”

- Alan Kay

Summary

“In addition to it begin useful, it is also cursed and the curse of the monad is that once you get the epiphany, once you understand - ‘oh that's what it is’ - you lose the ability to explain it to anybody.”

Douglas Crockford

Summary

Monad:

- A thing that contains one or more other things
- Used to define a pipeline, series of computational steps
- Allows to reuse code and write composable parts
- Programmable semicolons
- Controls code complexion

Introduction to Monads in Ruby

Do not fear, Monads are here

netguru

Kamil Kula

Ruby on Rails Developer

kamil.kula@netguru.com