MONADS IN RUBY

BY VICTOR ZAGORODNY

Email: post.vittorius@gmail.com
Twitter: https://twitter.com/vittoriuz
Github: https://github.com/vittorius

Vyacheslav Egorov
@mraleph

Follow

year 2013: Haskell people are still writing monad tutorials, JavaScript people are still trying to explain inheritance.

1:48 AM - 13 Apr 2013

MONADS: WHAT IS IT?

The Universe of Discourse : Monads are like burritos

https://blog.plover.com/prog/burritos.html ▾

Dec 15, 2009 - A few months ago Brent Yorgey complained about a certain class of tutorials which present **monads** by explaining how **monads** are like **burritos** ...

Monads are not Burritos

https://neoeinstein.github.io/monads-are-not-burritos/ ▾

Monads are not **Burritos** Explaining **monads** to the uninitiated Marcus Griep @neoeinstein So, we've all had it happen. We've been going along having a lot of ...

Code To Joy: Monads are Burritos

codetojoy.blogspot.com/2009/03/monads-are-burritos.html ▾

Mar 30, 2009 - Brent Yorgey has a wonderful post on the tarpit of writing monad tutorials. ... Dude, a **monad is a burrito**, if a burrito is a functor-like object with a ...

> *A monad is a monoid in the category of endofunctors, what's the problem?*
>
> — *James Iry*

~~MONADS: WHAT IS IT?~~

MONADS: WHY?

LAZY EVALUATION

EXAMPLE #1 (PSEUDOCODE):

```
let numbers = [1, ...]
fun is_prime x = all (map (y -> x mod y <> 0) [2..x-1])
let primes = filter is_prime numbers
let tenth_prime = first (take primes 10)
print tenth_prime
```

EXAMPLE #2:

```
if user.signed_in? && user.current_order&.pending?
# ...
```

WELCOME TO THE FUTURE

```
f = Future.new do
# executed asynchronously
end

f.complete? # non-blocking

f.on_complete do |value|
# called upon completion; preferred approach
end

f.value # blocks until complete; less preferred approach
```

AN UGLY EXAMPLE

```
users = UsersService.all

archived_users = ArchivedUsersService.for_last_month

(users + archived_users).select(&:active?).size
```

FUNCTIONALLY BEAUTIFUL EXAMPLE

```
count(
  only_active(
    concat(
      Future.new { UsersService.all },
      Future.new { ArchivedUsersService.for_last_month }
    )
  )
)
```

FUNCTIONALLY BEAUTIFUL EXAMPLE

```
count_f(
  only_active_f(
    concat_f(
      Future.new { UsersService.all },
      Future.new { ArchivedUsersService.for_last_month }
    )
  )
)
```

INGREDIENTS: UNIT

(aka "bind", "return")

```
f = Future.new(x)
# or
f = Future.new
f.complete_with(x)

f.value # doesn't block
f.complete? # == true
```

```
class Future
  # unit : Value -> Future<Value>
  def self.unit(value)
    Future.new(value)
  end
end
```

INGREDIENTS: FMAP

```
# count_f : Future<Enumerable> -> Future<Integer>
def count_f(future)
  future.fmap(
    # f: Enumerable -> Integer
    ->(enumerable) { enumerable.count }
  )
end
```

```
class Future
  # ...
  # fmap: Func<Value, Value> -> Future<Value>
  def fmap(func)
    f_new = Future.new
    on_complete do |value|
      f_new.complete_with(func.call(value))
    end
    f_new
  end
end
```

INGREDIENTS: FMAP

```
# only_active_f : Future<Enumerable> -> Future<Enumerable>
def only_active_f(future)
  future.fmap(
    # f: Enumerable -> Enumerable
    ->(enumerable) { enumerable.select(&:active?) }
  )
end
```


WE NEED TO GO
DEEPER

CHAINING FUTURES

```
f = Future.new { UsersService.all }.fmap(
  # f: Enumerable -> Future<Profile>
  ->(users) {
    Future.new { ProfileService.profile_for(users.first) }
  }
)
f.value # let's get the profile of a first user...
```

oops...

```
class Future
  def fmap(func)
    f_new = Future.new
    on_complete do |value|
      f_new.complete_with(func.call(value)) # it's a Future!
    end
    f_new
  end
end
```

INGREDIENTS: FLATTEN

(aka "join")

```
class Future
  # ...
  protected

  # flatten: Future<Future<Value>> -> Future<Value>
  def flatten
    f_flat = Future.new
    on_complete do |f_internal|
      f_internal.on_complete do |value|
        f_flat.complete_with(value)
      end
    end
    f_flat
  end
end
```

INGREDIENTS: BIND

(aka "flatMap")

```
class Future
  # ...

  # bind: Func<Value, Future<Value>> -> Future<Value>
  def bind(func)
    fmap(func).flatten
  end
end
```

CHAINING FUTURES CORRECTLY

```
f = Future.new { UsersService.all }.bind(
  # f: Enumerable -> Future<Profile>
  ->(users) {
    Future.new { ProfileService.profile_for(users.first) }
  }
)
# fmap(func).flatten was called under the hood
f.value # now it's the Profile
```

TWO IMPORTANT THINGS

1. Future is a monad
   Just believe me
2. I won't tell you how we built the `concat_f` function
   But I can give some clues

LIFTING

```
count -> lift -> count_f

def lift(func)
  ->(future) { future.fmap(func) }
end

concat -> lift2 -> concat_f

# Func<Value, Value, Value>
#   -> Func<Future<Value>, <Future<Value>, <Future<Value>>
def lift2(func) # aka "liftM2"
  # ...
end
```

MONAD: DEFINITION (AT LAST)

- It's a data type
- It's a container for some (immutable) value
- It has a `unit` operation defined which wraps a value in it
- It has a `bind` operation defined which allows building a chain of transformations over a monadic value

All of these make monad a *"programmable semicolon"*

EXAMPLE: MAYBE MONAD

```
class Maybe
  attr_reader :value

  class Some < Maybe
    def initialize(value)
      @value = value
    end
  end

  class None < Maybe; end
end
```

EXAMPLE: MAYBE MONAD

```
class Maybe
  # "unit" function
  def self.[](value)
    value.nil? ? None.new : Some.new(value)
  end

  def bind(func)
    is_a?(None) ? self : func.call(self.value)
  end

  alias_method :>=, :bind
end
```

EXAMPLE: MAYBE MONAD (USAGE)

```
# add_m2:
#  Func<Maybe<Integer>, Maybe<Integer>> -> Maybe<Integer>
def add_m2(m1, m2)
  m1.>=( # calling bind
    ->(x) { # in binding function; value of m1 unpacked to x
      # we need to return Maybe from here; m2 is Maybe
      # m2.>= will also return Maybe
      m2.>=( # calling bind
        ->(y) { # value of m2 unpacked to y
          Maybe[x + y] # x and y are Integers
          # add them and wrap the result into Maybe
        }
```

EXAMPLE: MAYBE MONAD (USAGE)

```
add_m2(Maybe[1], Maybe[2]) # => Maybe::Some#<value = 3>
add_m2(Maybe[4], Maybe[5]).value # => 9
add_m2(Maybe[nil], Maybe[5])
# => Maybe::None; "bind" of Maybe[5] was not executed
```

MAYBE MONAD

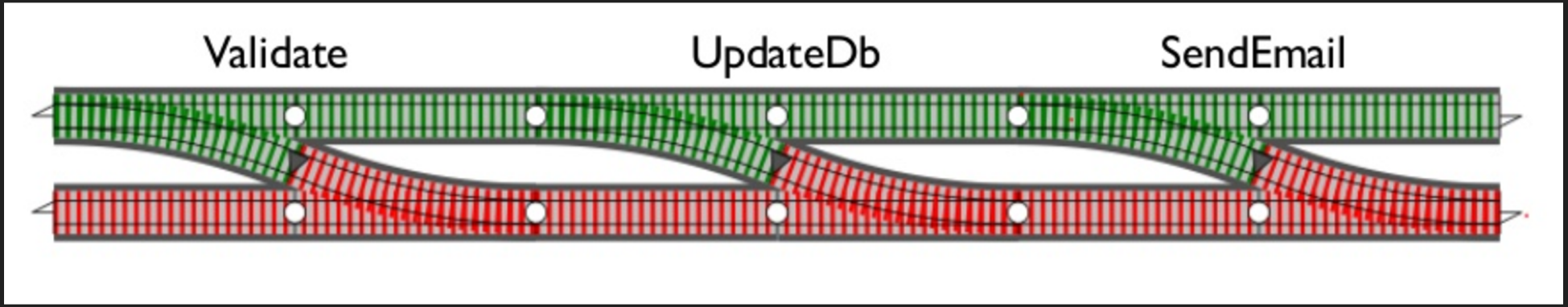
```
maybe_user = Maybe(user).bind do |u|
  Maybe(u.address).bind do |a|
    Maybe(a.street)
  end
end

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

EITHER MONAD

```
def calculate(input)
  Right(input).bind do |value|
    value > 1
      ? Right(value + 3)
      : Left("value was less than 1")
  end.bind do |value|
    value % 2 == 0
      ? Right(value * 2)
      : Left("value was not even")
  end
end
```

LOOKS FAMILIAR?



TRY MONAD

```
res = Try() { 10 / 2 }
res.value if res.success?
# => 5

res = Try() { 10 / 0 }
res.exception if res.failure?
# => #<ZeroDivisionError: divided by 0>

Try(NoMethodError, NotImplementedError) { 10 / 0 }
# => raised ZeroDivisionError: divided by 0 exception
```

AND OTHERS

- List
- Continuation
- ?

MONADS IN RUBY: EDITOR'S PICK

- monadic
  Maybe, Either monads. Looks like it's not supported anymore.
- deterministic
  Option (actually, it's Maybe), Either, Result (similar to Either). Almost no contributions are made for a long time.
- monads
  Option (aka Maybe), Many (aka List), Eventually (aka Continuation). Is a demo for the conference talk, but very clean and usable yet without any observable activity.
- kleisli
  Maybe, Either, Try, Future, functional composition operators. Good quality code but is abandoned in favor of its successor, dry-monads.
- dry-monads
  Maybe, Either, Try, List. Used by dry-rb ecosystem, most notable example is dry-transactions. Fairly good and regular support. Has its community and used in real projects.

MONADS IN RUBY: HONORABLE MENTIONS

- solid_use_case
  One of the first gems in the field, was mentioned in a famous talk Railway-Oriented programming
- Remember, you can always write your own

THANK YOU!

Questions?