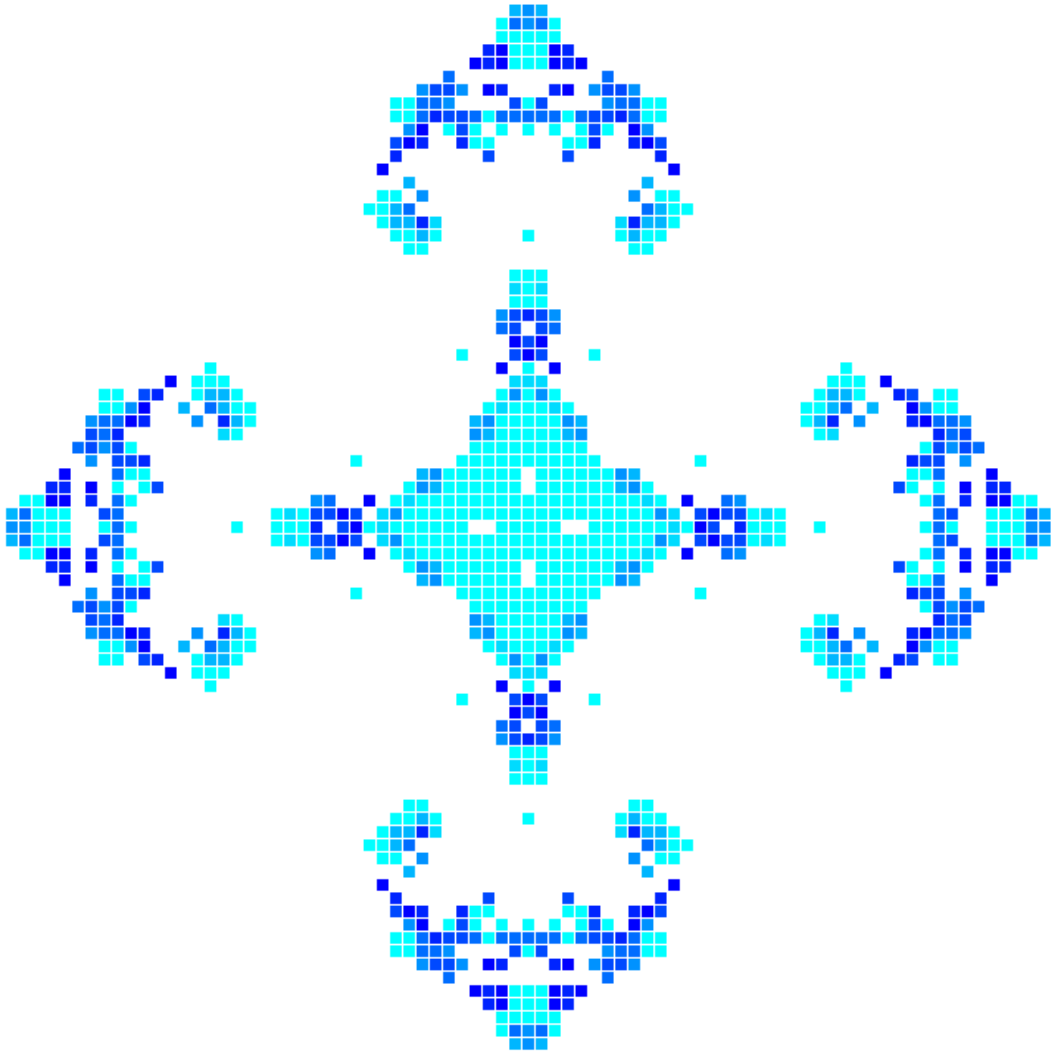
Foncteurs, Monades et Zippers

Jérémy Cochoy

Paris RB

Février 2018



Les fonctions

On considère des fonctions *pures* :

- déterministe
- sans effet de bord


```
def add2(n)
  n + 2
end
```

Les fonctions

On considère des fonctions *pures* :

- déterministe
- sans effet de bord


```
def add2(n)
  n + 2
end
```

Les types

Qu'appelons nous un type ?

Pour nous, c'est un *ensemble* de valeurs.

Exemples :

- $Integer = \{-2147483648, \ldots, 2147483647\}$
- $NilClass = \{nil\}$
- $Boolean = \{True, False\}$
- $[Nilclass] = \{[], [True], [False], [True, False], [False, True], \ldots\}$

Les types

Qu'appelons nous un type ?

Pour nous, c'est un *ensemble* de valeurs.

Exemples :

- $Integer = \{-2147483648, \ldots, 2147483647\}$
- $NilClass = \{nil\}$
- $Boolean = \{True, False\}$
- $[Nilclass] = \{[], [True], [False], [True, False], [False, True], \ldots\}$

Les types

Qu'appelons nous un type ?

Pour nous, c'est un *ensemble* de valeurs.

Exemples :

- $Integer = \{-2147483648, \ldots, 2147483647\}$
- $NilClass = \{nil\}$
- $Boolean = \{True, False\}$
- $[Nilclass] = \{[], [True], [False], [True, False], [False, True], \ldots\}$

Les types

Les fonctions sont de type : *a -> b*

- floor : : Float -> Integer
- 2.method(:+) : : Integer -> Integer

Les fonctions se composent

- f1 : : a -> b
- f2 : : b -> c
- { |x| f2(f1(x)) } : : a -> c

Fonctions à plusieurs paramètres

```
sum = ->(a, b) do
  a + b
end
```

```
irb> sum.(2,3)
=> 5
```

```
irb> sum.curry.(1)
=> #<Proc:0x0000000000288c3c0 (lambda)>
irb> sum.curry.(1).(2)
=> 3
```

Fonctions à plusieurs paramètres

```
sum = ->(a, b) do
  a + b
end
```

```
irb> sum.(2,3)
=> 5
```

```
irb> sum.curry.(1)
=> #<Proc:0x0000000000288c3c0 (lambda)>
irb> sum.curry.(1).(2)
=> 3
```

Fonctions à plusieurs paramètres

```
sum = ->(a, b) do
  a + b
end
```

```
irb> sum.(2,3)
=> 5
```

```
irb> sum.curry.(1)
=> #<Proc:0x0000000000288c3c0 (lambda)>
irb> sum.curry.(1).(2)
=> 3
```

Fonctions à plusieurs paramètres

```
sum = ->(a, b) do
  a + b
end
```

Curryfication

- sum : : (a, b) -> c
- sum.curry : : a -> (b -> c)

Fonctions à plusieurs paramètres

```
sum = ->(a, b) do
  a + b
end
```

Curryfication

- sum : : (a, b) -> c
- sum.curry : : a -> b -> c

Exercice

compose : : (a -> b) -> (b -> c) -> (a -> c)

Fonctions à plusieurs paramètres

```
sum = ->(a, b) do
  a + b
end
```

Curryfication

- sum : : (a, b) -> c
- sum.curry : : a -> b -> c

Exercice

compose : : (a -> b) -> (b -> c) -> (a -> c)

Les foncteurs applicatifs

Un foncteur F agit sur les types ...

- a => F a

- a => [a]
- a => Tree a
- a => Maybe a

... et sur les fonctions

- a -> b => F a -> F b

- fmap 2.method(:+2) : : F Int -> F Int
- fmap floor : : F Float -> F Int

Les foncteurs applicatifs

Un foncteur F agit sur les types ...

- a => F a

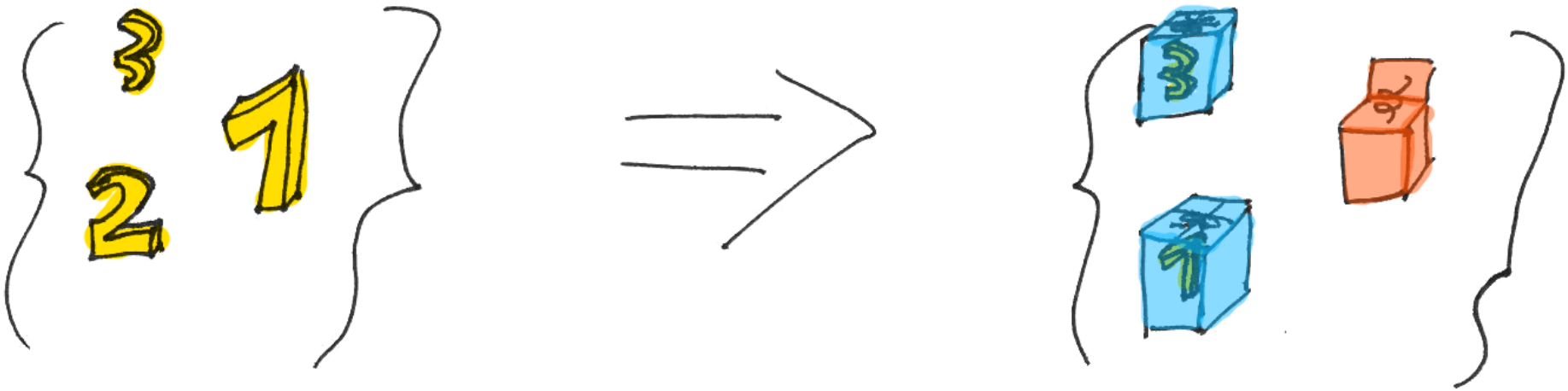
- a => [a]
- a => Tree a
- a => Maybe a

... et sur les fonctions

- a -> b => F a -> F b

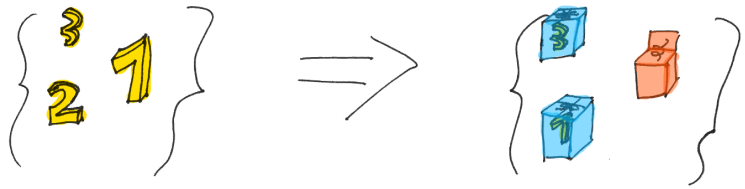
- fmap 2.method(:+2) : : F Int -> F Int
- fmap floor : : F Float -> F Int

Donnée dans un contexte



Un foncteur permet de passer d'un monde (les types a) vers un autre (les types F a).

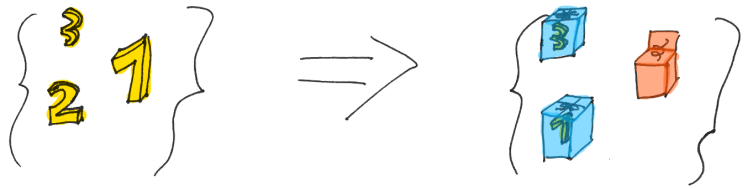
Maybe : Une implémentation



```ruby
class Just < Maybe
  def self.call(value)
    new [value]
  end
end
```

```ruby
irb> Just.(3)
=> #<Just:0x0000000000359c010 @content=[3]>
```

Maybe : Une implémentation

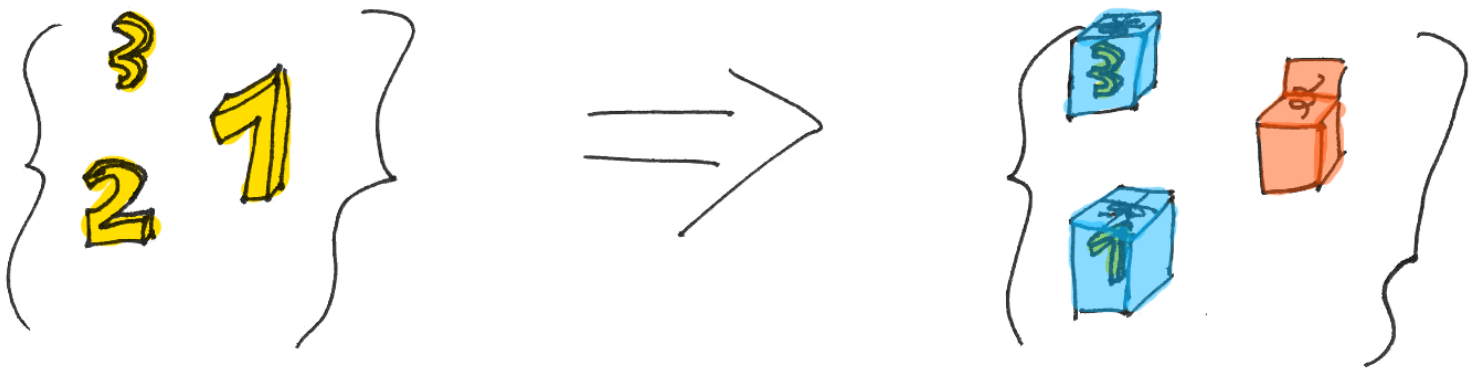


```
class Nothing < Maybe
  def self.call()
    new []
  end
end
```

```
irb> Nothing.()
=> #<Nothing:0x00000002d97d38 @content=[]>
```

Maybe : Une implémentation

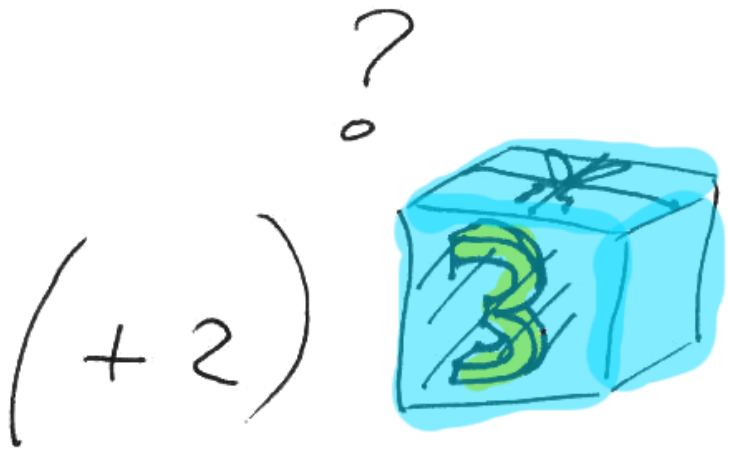
```ruby
class Maybe
  private_class_method :new

  def initialize(content)
    @content = content
  end
  def from_maybe(default_value)
    return default_value if @content.empty?
    @content.first
  end
end
```



```
irb> Just.(3).from_maybe
=> 3
irb> Nothing.().from_maybe
=> nil
```

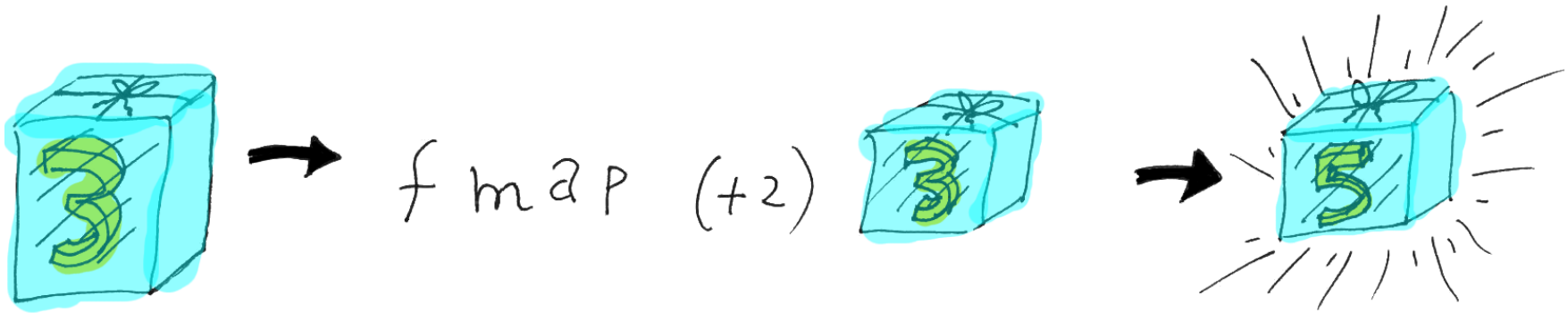
Functorial mapping

On ne peut plus appliquer la fonction telle quelle :



Functorial mapping

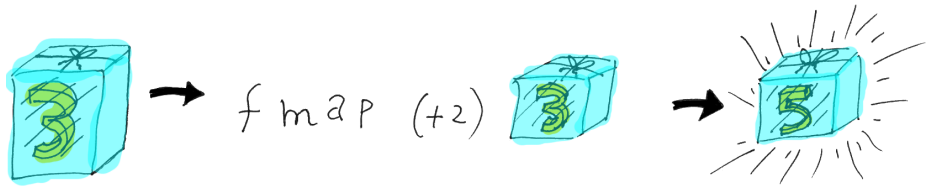
Mais le foncteur nous donne une nouvelle flèche.



Implémentation de fmap

```
def fmap(f)
  ->(box) do
    case box
    when Nothing
      return Nothing.()
    when Just
      r = f.(box.from_maybe)
      Just.(r)
    end
  end
end
```

Functorial mapping




```
add2 = ->(x) {x + 2}
```

```
add2.call Just.(3)
# NoMethodError (undefined method `+' for
#   #<Just:0x0000000003485280 @content=[3]>)

(fmap add2).call Just.(3)
# => #<Just:0x00000000036f8ff8 @content=[5]>
```

Dura lex sed lex

Un foncteur doit respecter des lois

- fmap id = id
- fmap (p ∘ q) = (fmap p) ∘ (fmap q)

```
id = ->(x) {x}

(p o q) = ->(x) { p.(q.(x)) }
```

Un foncteur est un endofoncteur de la catégorie des types.

Dura lex sed lex

Un foncteur doit respecter des lois

- fmap id = id
- fmap (p ∘ q) = (fmap p) ∘ (fmap q)

```
id = ->(x) {x}

(p o q) = ->(x) { p.(q.(x)) }
```

Un foncteur est un endofoncteur de la catégorie des types.

Dura lex sed lex

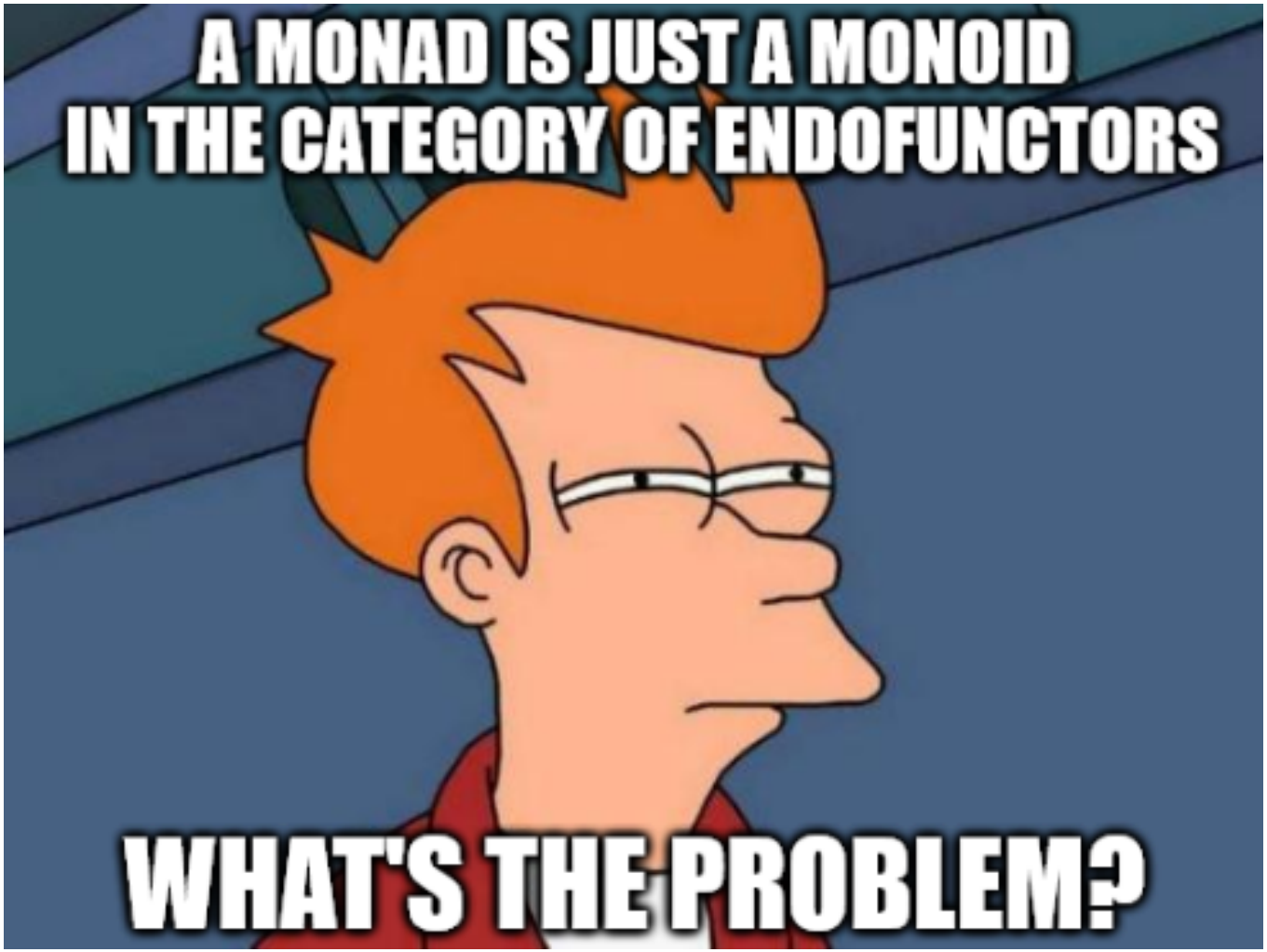
Un foncteur doit respecter des lois

- fmap id = id
- fmap (p ∘ q) = (fmap p) ∘ (fmap q)

```
id = ->(x) {x}

(p o q) = ->(x) { p.(q.(x)) }
```

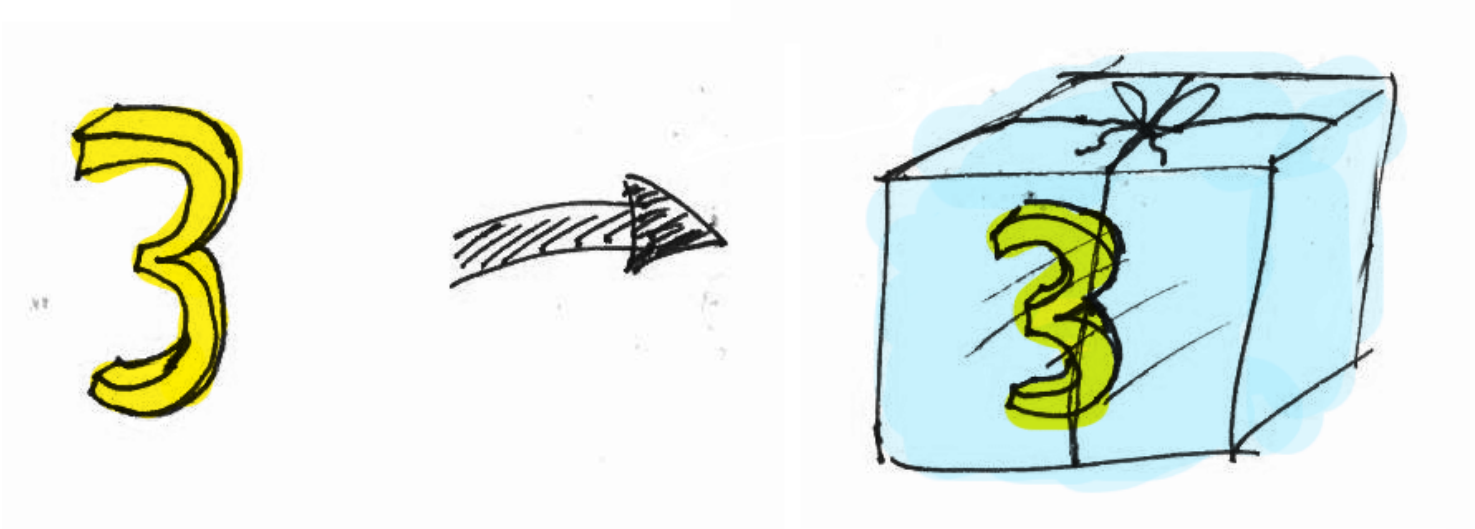
Un foncteur est un endofoncteur de la catégorie des types.

Monades


A MONAD IS JUST A MONOID
IN THE CATEGORY OF ENDOFUNCTORS
WHAT'S THE PROBLEM?

Donnée dans un contexte

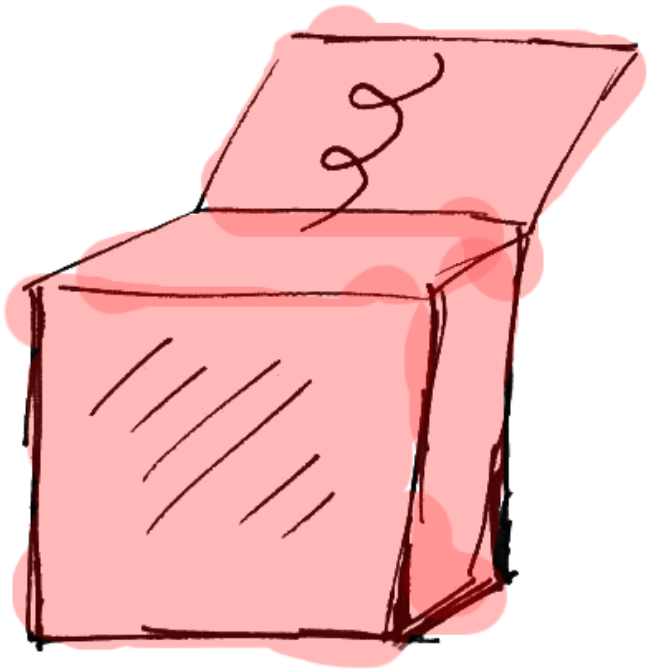
Une monade place une valeur dans un contexte.



L'exemple de Maybe : `Just 3`

Donnée dans un contexte

Un contexte peut aussi ne pas contenir de valeur.



L'exemple de Maybe : `Nothing`

Placer une donnée dans un contexte

L'opérateur *pure*

```
pure :: a -> F a
```

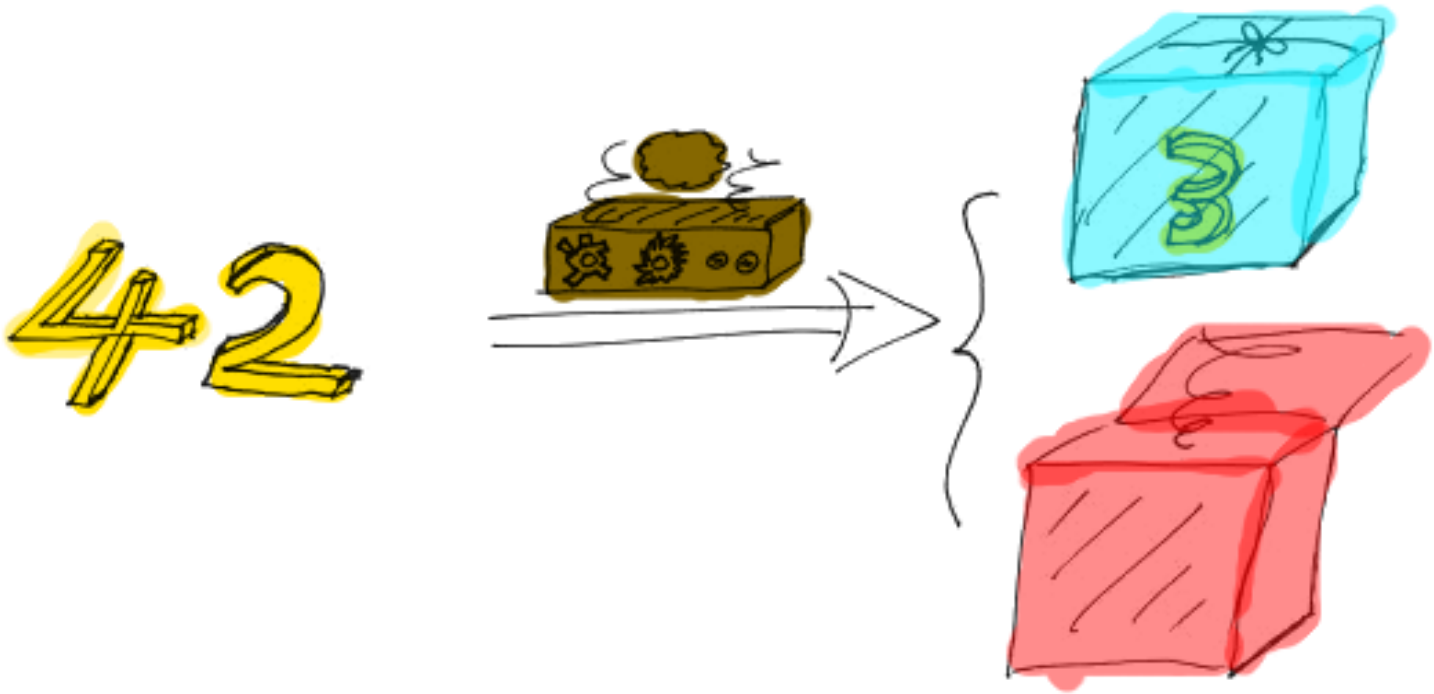
Quelques cas particuliers

- `Just`
- `[] <<`

D'autres types

- Maybe = Nothing | Just a
- Tree = Leaf | Node a (Tree a) (Tree a)
- Either = Left a | Right b

Un traitement qui peut échouer



Une fonction de type *Int -> Maybe Int*.

Composer des traitements avec échec

Comment composer `f :: a -> M b` et `g :: b -> M c` ?

Si M est un foncteur, on peut composer `f :: a -> M b` avec `fmap g :: M b -> M (M c)`.

Que faire d'un `M (M c)` ?

Composer des traitements avec échec

Comment composer `f :: a -> M b` et `g :: b -> M c` ?

Si M est un foncteur, on peut composer `f :: a -> M b` avec `fmap g :: M b -> M (M c)`.

Que faire d'un `M (M c)` ?

Composer des traitements avec échec

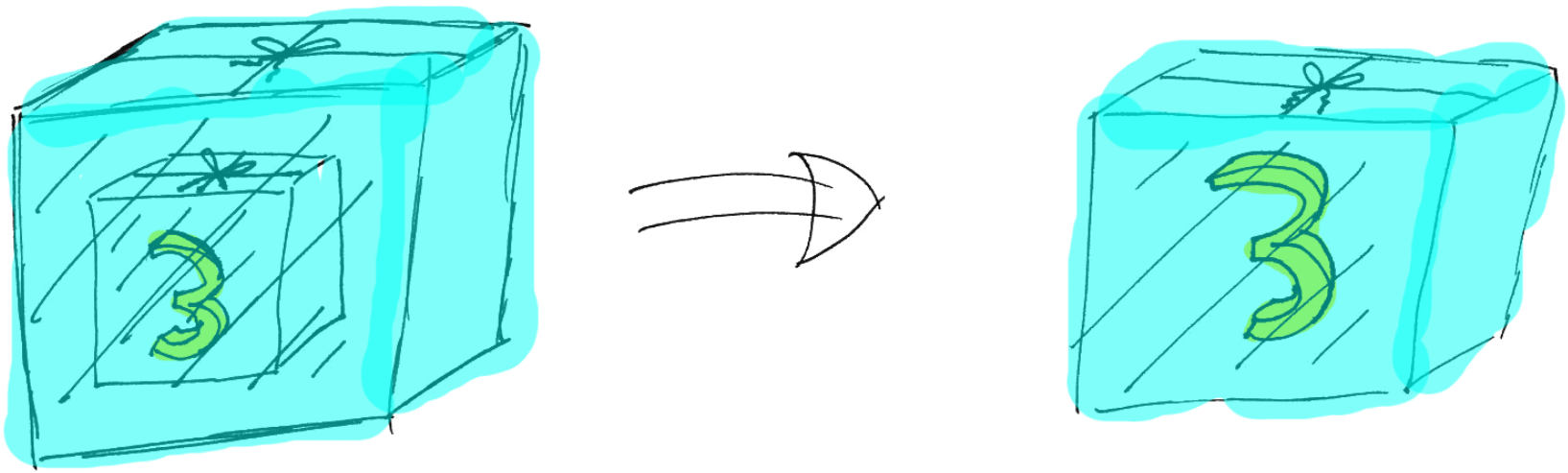
Comment composer `f :: a -> M b` et `g :: b -> M c` ?

Si M est un foncteur, on peut composer `f :: a -> M b` avec `fmap g :: M b -> M (M c)`.

Que faire d'un `M (M c)` ?

L'opérateur join

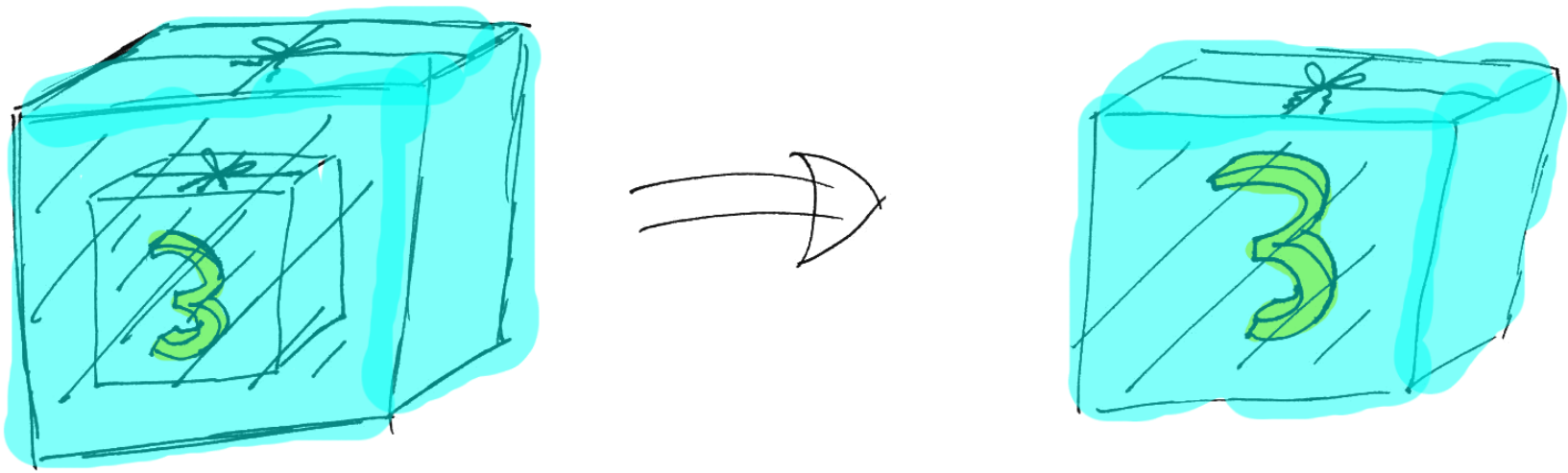
```
join :: M (M a) -> M a
```



```
join Just.(Just.(3)).
```

L'opérateur join

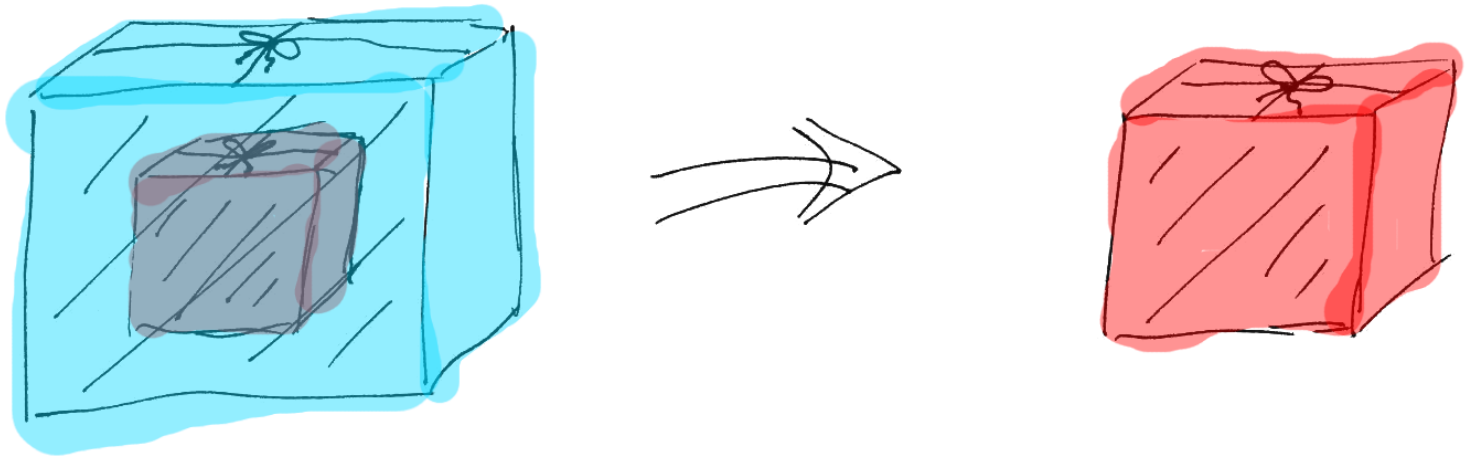
```
join :: M (M a) -> M a
```



```
join Just.(Just.(3)).
```

L'opérateur join

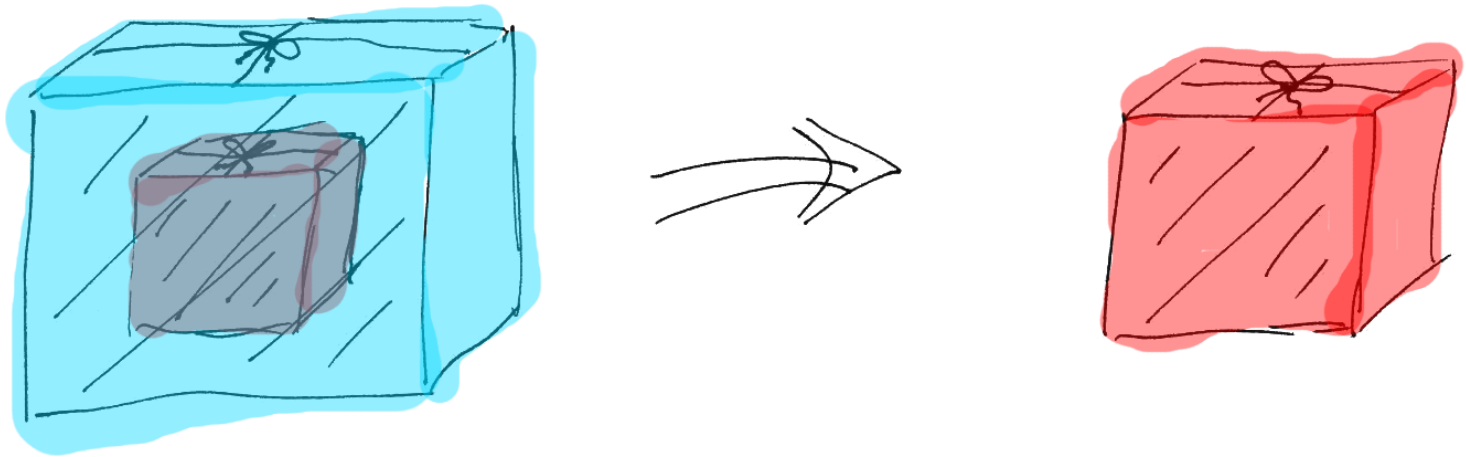
```
join :: M (M a) -> M a
```



```
join Just.(Nothing.()).
```

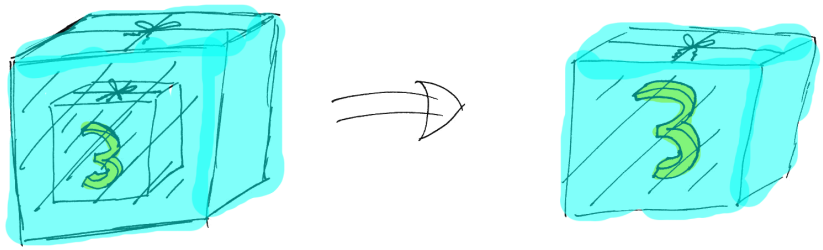
L'opérateur join

```
join :: M (M a) -> M a
```



```
join Just.(Nothing.()).
```

L'opérateur *join*



Une implémentation de join :

```
def join(bbox)
  case bbox
  when Nothing
    Nothing.()
  when Just
    bbox.from_maybe
  end
end
```

L'opérateur *rfish*

On cherche à définir la composition.

```
rfish :: (a -> M b) -> (b -> M c) -> (a -> M c)
```

Nous avons :

- `(fmap g) o f :: a -> M (M c)`
- `join :: M (M a) -> M a`

On peut maintenant composer f et g.

```
def rfish(f, g)
  ->(x) do
    join (fmap g).(f.(x))
  end
end
```

L'opérateur *rfish*

On cherche à définir la composition.

```
rfish :: (a -> M b) -> (b -> M c) -> (a -> M c)
```

Nous avons :

- `(fmap g) o f :: a -> M (M c)`
- `join :: M (M a) -> M a`

On peut maintenant composer f et g.

```
def rfish(f, g)
  ->(x) do
    join (fmap g).(f.(x))
  end
end
```

L'opérateur *rfish*

On cherche à définir la composition.

```
rfish :: (a -> M b) -> (b -> M c) -> (a -> M c)
```

Nous avons :

- `(fmap g) o f :: a -> M (M c)`
- `join :: M (M a) -> M a`

On peut maintenant composer f et g.

```
def rfish(f, g)
  ->(x) do
    join (fmap g).(f.(x))
  end
end
```

Récapitulatif

Une monade, c'est

- pure : : a -> M a
- fmap : : (a -> b) -> (M a -> M b)
- join : : M (M a) -> M a


A must read
Haskell
The understandable parts
O RLY?
Bob Dylan

Dura lex sed lex

Une monade doit respecter des lois

- `pure o f ≡ (fmap f) o pure`
- `join o fmap (fmap f) ≡ (fmap f) o join`

- `join o fmap join ≡ join o join`
- `join o fmap pure ≡ join o pure = id`

Monades - Catégories

Une monade (T, μ, η) est la donnée d'un endofoncteur $T : C \to C$ et de deux transformations naturelles $\mu : T \circ T \to T$ et $\eta : 1_C \to T$ telles que :

$$\begin{array}{ccc} T(T(T(X))) & \xrightarrow{T(\mu_X)} & T(T(X)) \\ \downarrow{\scriptstyle \mu_{T(X)}} & & \downarrow{\scriptstyle \mu_X} \\ T(T(X)) & \xrightarrow[\mu_X]{} & T(X) \end{array} \qquad \begin{array}{ccc} T(X) & \xrightarrow{\eta_{T(X)}} & T(T(X)) \\ \downarrow{\scriptstyle T(\eta_X)} & \searrow & \downarrow{\scriptstyle \mu_X} \\ T(T(X)) & \xrightarrow[\mu_X]{} & T(X) \end{array}$$

c'est à dire $\mu \circ T\mu = \mu \circ \mu_T$ et $\mu \circ T\eta = \mu \circ \eta_T = id_T$.

Dans notre cas C la catégorie des types.

Monades - Catégories

Une monade (T, μ, η) est la donnée d'un endofoncteur $T : C \to C$ et de deux transformations naturelles $\mu : T \circ T \to T$ et $\eta : 1_C \to T$ telles que :

$$\begin{array}{ccc} T(T(T(X))) & \xrightarrow{T(\mu_X)} & T(T(X)) \\ \downarrow{\scriptstyle \mu_{T(X)}} & & \downarrow{\scriptstyle \mu_X} \\ T(T(X)) & \xrightarrow[\mu_X]{} & T(X) \end{array} \qquad \begin{array}{ccc} T(X) & \xrightarrow{\eta_{T(X)}} & T(T(X)) \\ \downarrow{\scriptstyle T(\eta_X)} & \searrow & \downarrow{\scriptstyle \mu_X} \\ T(T(X)) & \xrightarrow[\mu_X]{} & T(X) \end{array}$$

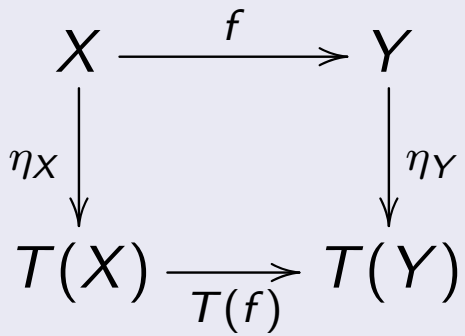
c'est à dire $\mu \circ T\mu = \mu \circ \mu_T$ et $\mu \circ T\eta = \mu \circ \eta_T = id_T$.

Dans notre cas C la catégorie des types.

A chaque loi son diagramme

pure est une T.N.

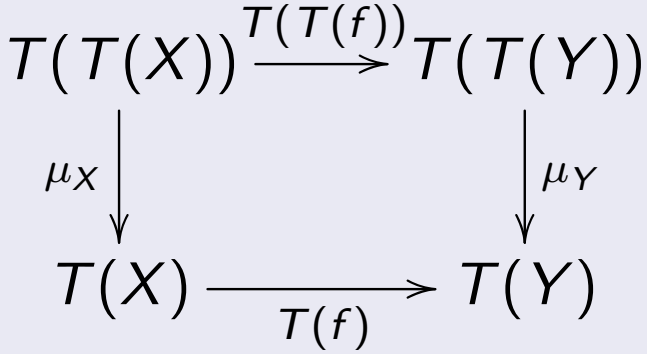
```
pure . f ≡ (fmap f) . pure
```



A chaque loi son diagramme

join est une T.N.

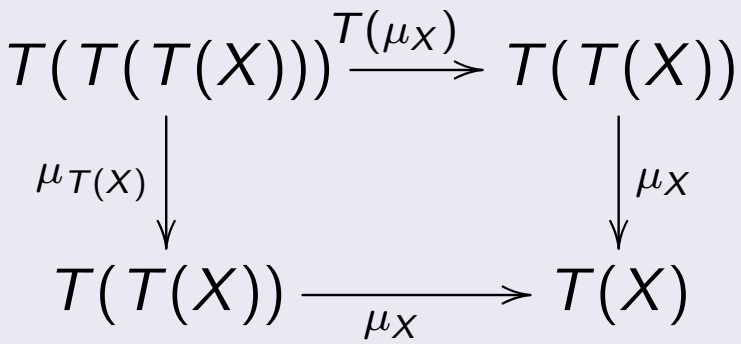
```
join . fmap (fmap f) ≡ (fmap f) . join
```



A chaque loi son diagramme

Associativité

```
join . fmap join ≡ join . join
```

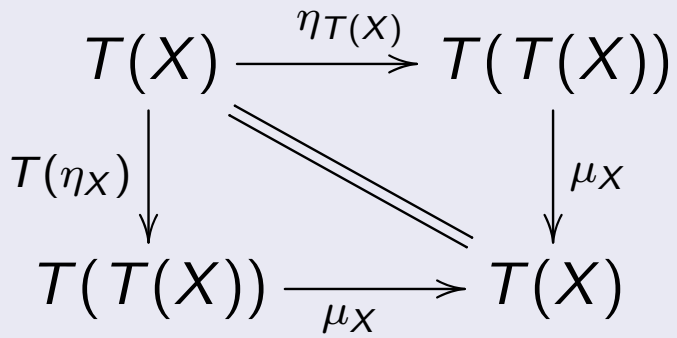


$$\mu \circ T\mu = \mu \circ \mu_T$$

A chaque loi son diagramme

Existence d'un neutre

```
join . fmap pure ≡ join . pure = id
```

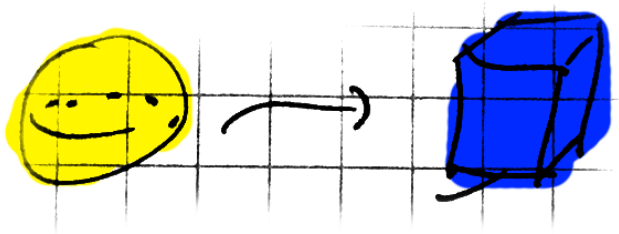


$$\mu \circ T\eta = \mu \circ \eta_T = id_T$$

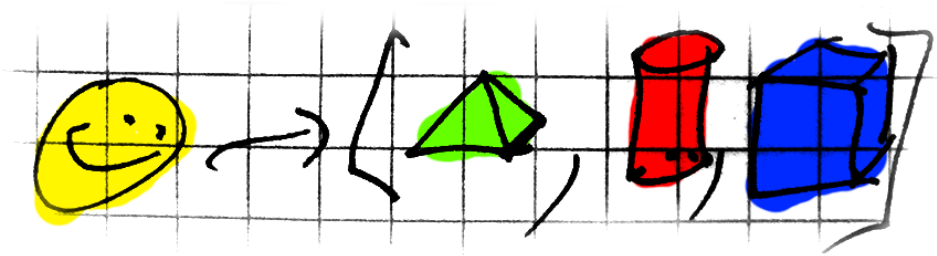
Et dans la vraie vie ?

Tout ça, à quoi ça sert ?

Calculs non déterministes



- Déterministe : Calcul produisant une valeur.



- Non déterministe : Calcul produisant plusieurs valeurs.

Quelques exemples

```
powers = ->(n) do
  [n, n*n, n*n*n]
end
```

```
neighbors = ->(n) do
  [n-1, n+1]
end
```

Implémentation de fmap

C'est tout simplement `Array#map`

$$[1, 2, 3] \xrightarrow{\text{fmap f}} [f(1), f(2), f(3)]$$

```
def fmap(f)
  ->(list) { list.map(f) }
end
```

Join et RFish

join

```
def join(llist)
  llist.flatten(1)
end
```

rfish

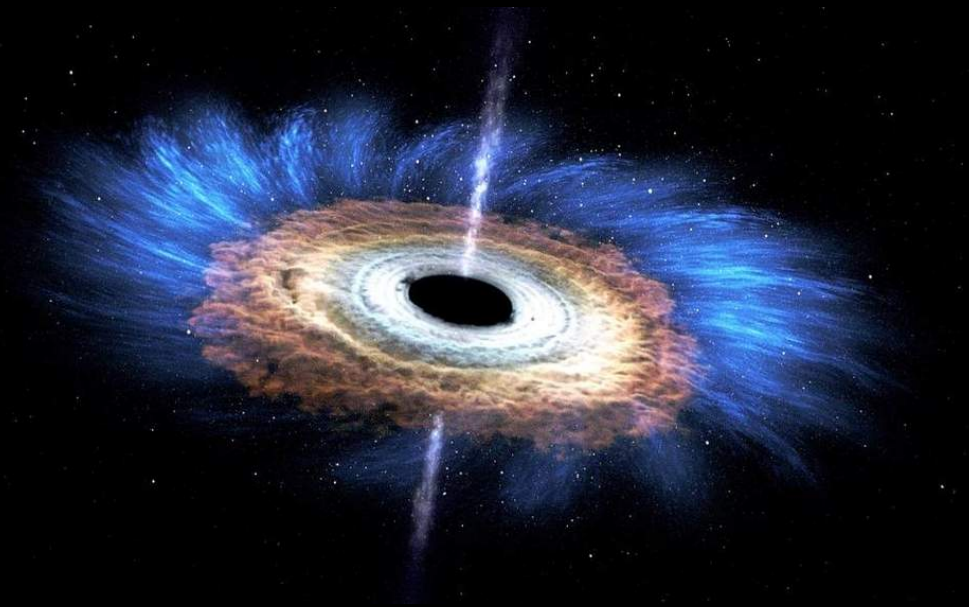
```
def rfish(f, g)
  ->(x) { f.(x).map(&g).flatten }
end
```

Monade de listes

Example d'utilisation

```
irb> rfish(neighbors, neighbors).(42)
=> [40, 42, 42, 44]
irb> rfish(powers, powers).(2)
=> [2, 4, 8, 4, 16, 64, 8, 64, 512]
irb> rfish(neighbors, powers).(42)
=> [41, 1681, 68921, 43, 1849, 79507]
irb> rfish(->(x) { [x, -x] }, neighbors).(42)
=> [41, 43, -43, -41]
```

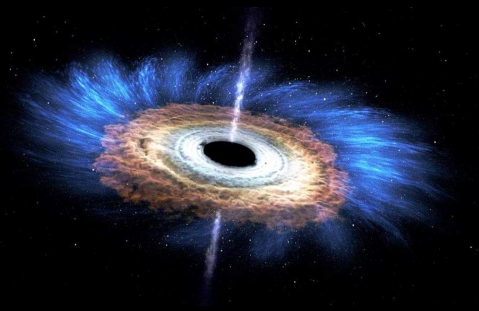
La monade `nil`



En ruby Maybe s'appelle `nil`

Union disjointe $a \bigsqcup \{nil\}$.

La monade nil



Opérateur bind &.

```
some_computation(42)&.get_value("bidirectional")
```

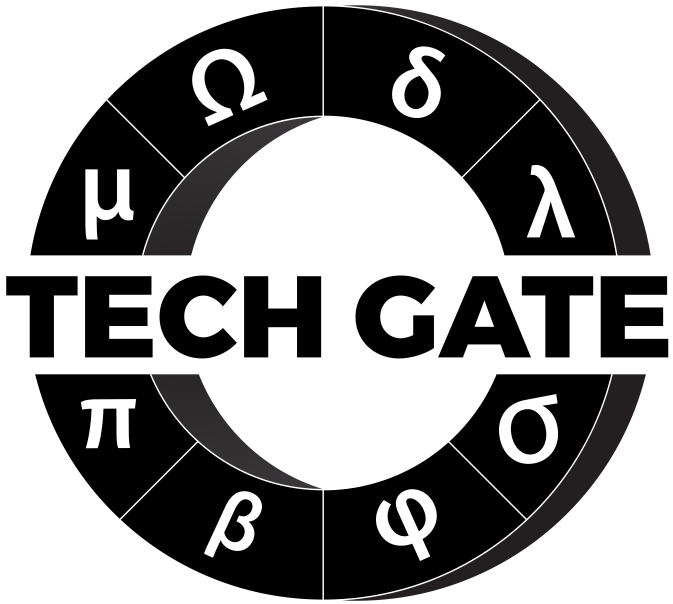
Opérateur bind &.

- &. : : $a \bigsqcup \{nil\}$ -> (a -> $b \bigsqcup \{nil\}$) -> $b \bigsqcup \{nil\}$
- bind : : M a -> (a -> M b) -> M b

On aurait aussi pu parler de...

- Left/Right Fish and Bind
- List comprehension
- Monades transformeurs
- Comonades et zippers
- Application aux automates cellulaires

Qui suis-je ?



Dr. Jérémy Cochoy

jeremy.cochoy@gmail.com

http://techgate.fr

Backend Developper

klaxit

Merci pour votre attention !