

# Foncteurs, Monades et Zippers

Jérémy Cochoy

Paris RB

Février 2018



## 1 Foncteurs applicatifs

- Fonctions
- Types
- Foncteurs
- Monades

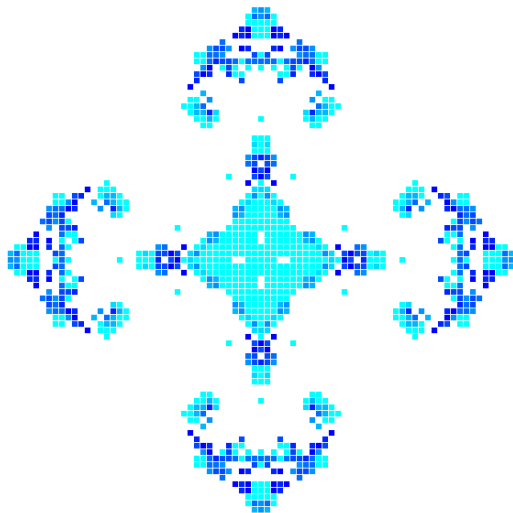
## 2 Automates Cellulaires

- Qu'est-ce que c'est ?
- Le jeu de la vie
- Algorithme 1D

## 3 Comonades

## 4 Évaluer un automate est comonadique

- Un univers
- Un foncteur
- Une comonade
- Evaluation



# 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 :

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

# Les types

Qu'appelons nous un type ?

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

Exemples :

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

# Les types

Qu'appelons nous un type ?

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

Exemples :

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



# Les types

Les fonctions sont de type :  $a \rightarrow b$

- `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:0x000000000288c3c0 (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:0x000000000288c3c0 (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:0x000000000288c3c0 (lambda)>  
irb> sum.curry.(1).(2)  
=> 3
```

# Fonctions à plusieurs paramètres

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

## Curryfication

- $\text{sum} :: (a, b) \rightarrow c$
- $\text{sum.curry} :: a \rightarrow (b \rightarrow c)$

# Fonctions à plusieurs paramètres

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

## Curryfication

- $\text{sum} :: (a, b) \rightarrow c$
- $\text{sum.curry} :: a \rightarrow b \rightarrow c$

## Exercice

$\text{compose} :: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c)$

# Fonctions à plusieurs paramètres

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

## Curryfication

- $\text{sum} :: (a, b) \rightarrow c$
- $\text{sum.curry} :: a \rightarrow b \rightarrow c$

## Exercice

$\text{compose} :: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c)$

# Les foncteurs applicatifs

Un foncteur  $F$  agit sur les types ...

- $a \Rightarrow F\ a$

- $a \Rightarrow [a]$

- $a \Rightarrow \text{Tree } a$

- $a \Rightarrow \text{Maybe } a$

... et sur les fonctions

- $a \rightarrow b \Rightarrow F\ a \rightarrow F\ b$

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

- `fmap floor :: F Float -> F Int`



# Les foncteurs applicatifs

Un foncteur  $F$  agit sur les types ...

- $a \Rightarrow F\ a$

- $a \Rightarrow [a]$

- $a \Rightarrow \text{Tree } a$

- $a \Rightarrow \text{Maybe } a$

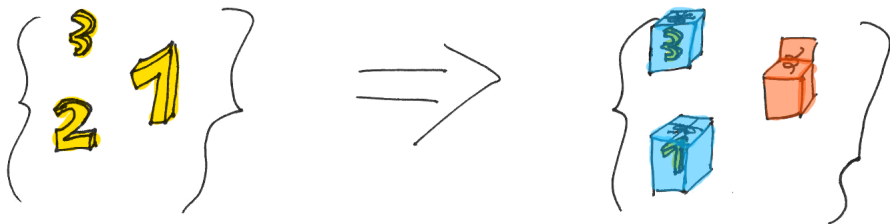
... et sur les fonctions

- $a \rightarrow b \Rightarrow F\ a \rightarrow F\ b$

- `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  $F a$ ).

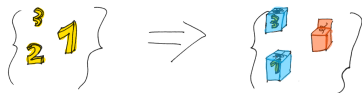
# Maybe : Une implémentation



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

```
irb> Just.(3)
=> #<Just:0x000000000359c010 @content=[3]>
```

# Maybe : Une implémentation



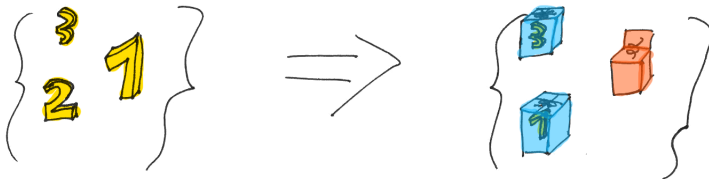
```
class Nothing < Maybe
  def self.call()
    new []
  end
end
```

```
irb> Nothing.()
=> #<Nothing:0x0000000002d97d38 @content=[] >
```

# Maybe : Une implémentation

```
class Maybe
  private_class_method :new

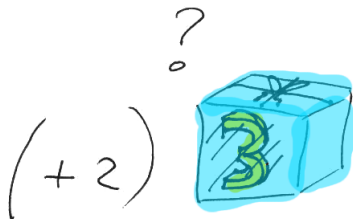
  def initialize(content)
    @content = content
  end
  def from_maybe(default_value)
    return default_value if @content.empty?
    @content.first
  end
end
```



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

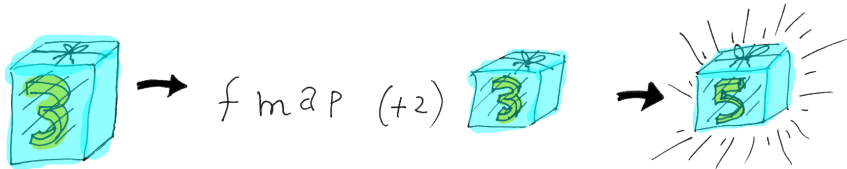
# Functorial mapping

On ne peut plus appliquer la fonction telle quelle :



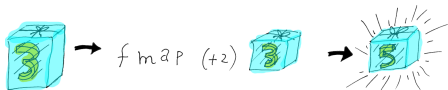
# Functorial mapping

Mais le foncteur nous donne une nouvelle flèche.





# 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

- $\text{fmap id} = \text{id}$
- $\text{fmap } (p \circ q) = (\text{fmap } p) \circ (\text{fmap } q)$

```
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

- $\text{fmap id} = \text{id}$
- $\text{fmap } (p \circ q) = (\text{fmap } p) \circ (\text{fmap } q)$

```
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

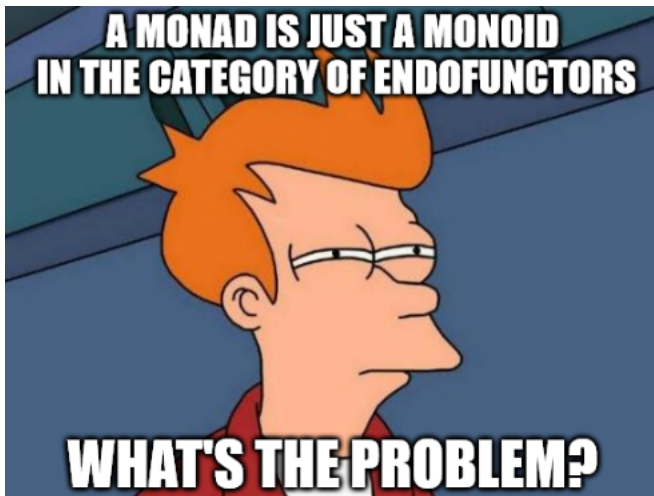
- $\text{fmap id} = \text{id}$
- $\text{fmap } (p \circ q) = (\text{fmap } p) \circ (\text{fmap } q)$

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

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

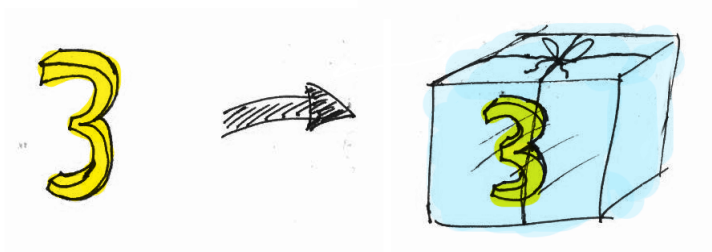
Un foncteur est un endofoncteur de la catégorie des types.

# Monades



# Donnée dans un contexte

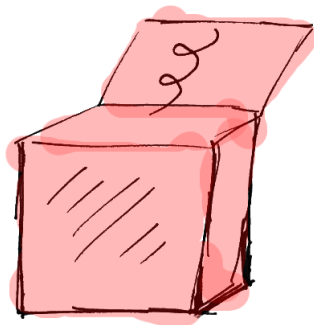
Une monade place une valeur dans un contexte.



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*

$$\text{pure} :: a \rightarrow F a$$

## Quelques cas particuliers

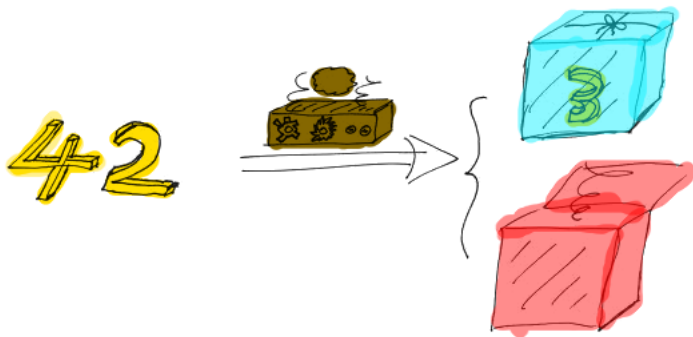
- Just
- [] <<

## 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  $Int \rightarrow Maybe\ Int$ .

# Composer des traitements avec échec

Comment composer  $f :: a \rightarrow M\ b$  et  $g :: b \rightarrow M\ c$  ?

Si  $M$  est un foncteur, on peut composer  $f :: a \rightarrow M\ b$  avec  $\text{fmap } g :: M\ b \rightarrow M\ (M\ c)$ .

Que faire d'un  $M\ (M\ c)$  ?

# Composer des traitements avec échec

Comment composer  $f :: a \rightarrow M\ b$  et  $g :: b \rightarrow M\ c$  ?

Si  $M$  est un foncteur, on peut composer  $f :: a \rightarrow M\ b$  avec  $\text{fmap } g :: M\ b \rightarrow M\ (M\ c)$ .

Que faire d'un  $M\ (M\ c)$  ?

# Composer des traitements avec échec

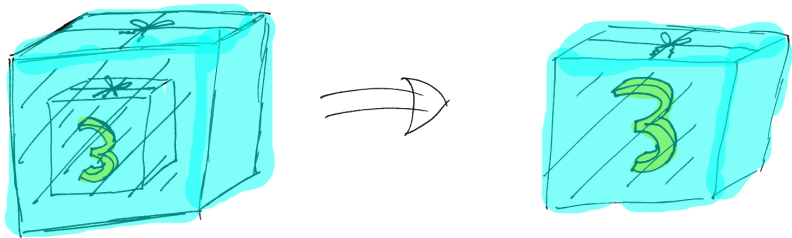
Comment composer  $f :: a \rightarrow M\ b$  et  $g :: b \rightarrow M\ c$  ?

Si  $M$  est un foncteur, on peut composer  $f :: a \rightarrow M\ b$  avec  $\text{fmap } g :: M\ b \rightarrow M\ (M\ c)$ .

Que faire d'un  $M\ (M\ c)$  ?

# L'opérateur join

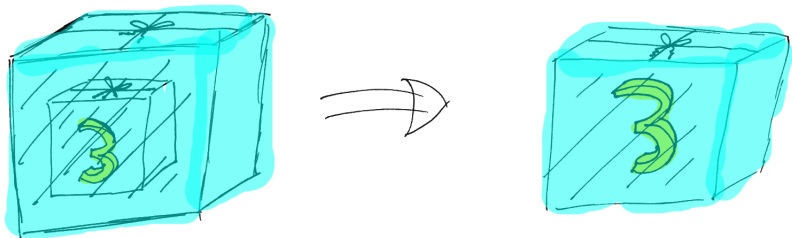
```
join :: M (M a) -> M a
```



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

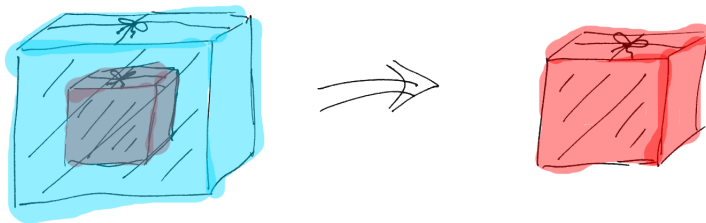
# L'opérateur join

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



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

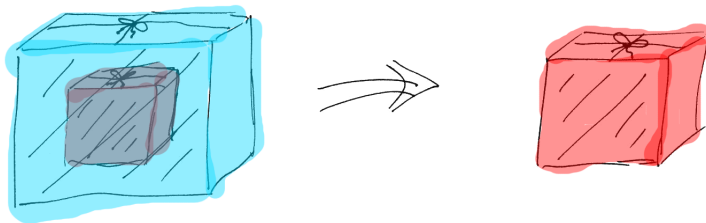
# L'opérateur join

$$\text{join} :: M (M a) \rightarrow M a$$


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

# L'opérateur join

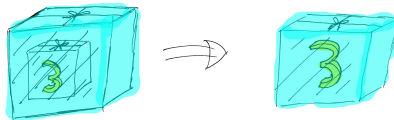
```
join :: M (M a) -> M a
```



```
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 *bind*

On cherche à définir la composition.

$$\text{bind} :: (a \rightarrow M\ b) \rightarrow (b \rightarrow M\ c) \rightarrow (a \rightarrow M\ c)$$

Nous avons :

- $(\text{fmap}\ g) \circ f :: a \rightarrow M\ (M\ c)$
- $\text{join} :: M\ (M\ a) \rightarrow M\ a$

On peut maintenant composer  $f$  et  $g$ .

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

# L'opérateur *bind*

On cherche à définir la composition.

$$\text{bind} :: (a \rightarrow M\ b) \rightarrow (b \rightarrow M\ c) \rightarrow (a \rightarrow M\ c)$$

Nous avons :

- $(\text{fmap}\ g) \circ f :: a \rightarrow M\ (M\ c)$
- $\text{join} :: M\ (M\ a) \rightarrow M\ a$

On peut maintenant composer  $f$  et  $g$ .

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

# L'opérateur *bind*

On cherche à définir la composition.

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

Nous avons :

- $(\text{fmap } g) \circ f :: a \rightarrow M (M c)$
- $\text{join} :: M (M a) \rightarrow M a$

On peut maintenant composer  $f$  et  $g$ .

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

# Récapitulatif

## Une monade, c'est

- $\text{pure} :: a \rightarrow M\ a$
- $\text{fmap} :: (a \rightarrow b) \rightarrow (M\ a \rightarrow M\ b)$
- $\text{join} :: M\ (M\ a) \rightarrow M\ a$

# Dura lex sed lex

## Une monade doit respecter des lois

- $\text{pure} \circ f \equiv (\text{fmap } f) \circ \text{pure}$
- $\text{join} \circ \text{fmap } (\text{fmap } f) \equiv (\text{fmap } f) \circ \text{join}$
- $\text{join} \circ \text{fmap } \text{join} \equiv \text{join} \circ \text{join}$
- $\text{join} \circ \text{fmap } \text{pure} \equiv \text{join} \circ \text{pure} = \text{id}$

# Monades - Catégories

Une monade  $(T, \mu, \eta)$  est la donnée d'un endofoncteur  $T : \mathcal{C} \rightarrow \mathcal{C}$  et de deux transformations naturelles  $\mu : T \circ T \rightarrow T$  et  $\eta : 1_{\mathcal{C}} \rightarrow T$  telles que :

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

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  $\mathcal{C}$  la catégorie des types.

# Monades - Catégories

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

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

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  $\equiv$  (fmap f) . pure`

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ \eta_X \downarrow & & \downarrow \eta_Y \\ T(X) & \xrightarrow{T(f)} & T(Y) \end{array}$$

# A chaque loi son diagramme

join est une T.N.

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

$$\begin{array}{ccc} T(T(X)) & \xrightarrow{T(T(f))} & T(T(Y)) \\ \mu_X \downarrow & & \downarrow \mu_Y \\ T(X) & \xrightarrow{T(f)} & T(Y) \end{array}$$

# A chaque loi son diagramme

## Associativité

`join . fmap join ≡ join . join`

$$\begin{array}{ccc} T(T(T(X))) & \xrightarrow{T(\mu_X)} & T(T(X)) \\ \mu_{T(X)} \downarrow & & \downarrow \mu_X \\ T(T(X)) & \xrightarrow{\mu_X} & T(X) \end{array}$$

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

# A chaque loi son diagramme

## Existence d'un neutre

`join . fmap pure  $\equiv$  join . pure = id`

$$\begin{array}{ccc} T(X) & \xrightarrow{\eta_{T(X)}} & T(T(X)) \\ T(\eta_X) \downarrow & \searrow & \downarrow \mu_X \\ T(T(X)) & \xrightarrow{\mu_X} & T(X) \end{array}$$

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

## Automates cellulaires



Toison d'or

# Qu'est-ce qu'un automate cellulaire ?

Un automate cellulaire, c'est :

- Un nombre fini d'états  $S$ ,
- Une grille de cellules,
- La notion de voisinage d'une cellule  $V_C$ ,
- Une fonction de transition qui à une cellule associe son nouvel état.

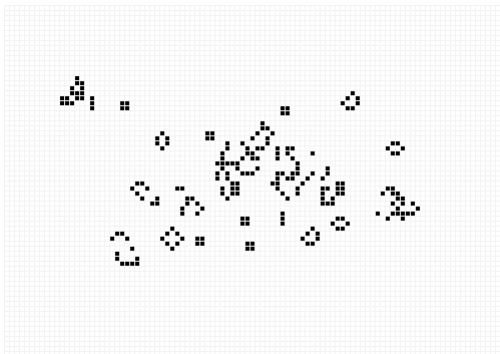
# Combien d'automates cellulaires différents ?

On a le choix :

- De la dimension de la grille,
- Des lois,
- Du nombres d'états (couleurs),
- De la forme du voisinages (boules de rayon  $r$ , etc.),
- De ne pas être déterministe.

# The "Game of Life"

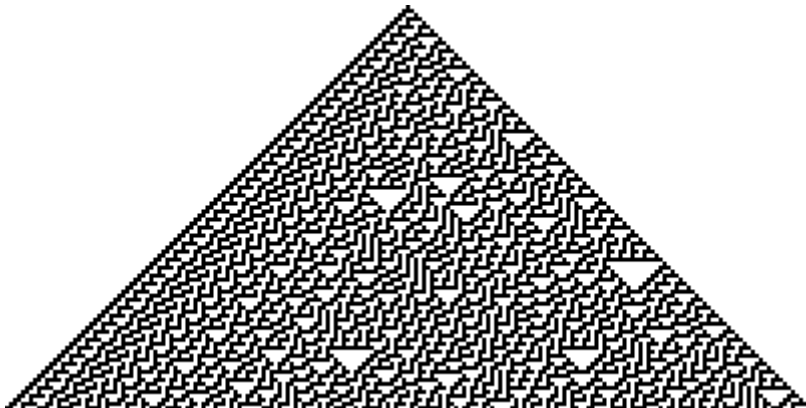
## Jeu de la vie (J. H. Conway)





# Étude d'un cas : Rule 30

Rule 30



# La grille

## La grille de l'automate



- Une grille 1D
- Deux états (Blanc / Noir)

Un voisinage de 3 cellules.

Les règles



On peut aussi écrire :

Ancien état	111	110	101	100	011	010	001	000
Nouvel état	0	0	0	1	1	1	1	0

Un voisinage de 3 cellules.

Les règles



On peut aussi écrire :

Ancien état	111	110	101	100	011	010	001	000
Nouvel état	0	0	0	1	1	1	1	0

Un voisinage de 3 cellules.

Les règles



On peut aussi écrire :

Ancien état	111	110	101	100	011	010	001	000
Nouvel état	0	0	0	1	1	1	1	0

## Comonades



# C'est le dual d'une monade

- C'est un foncteur  $M$
- $\text{extract (copure) (co uinit)} : : M\ a \rightarrow a$
- $\text{duplicate (cojoin) (co product } \delta) : : M\ a \rightarrow M\ (M\ a)$

# Dura lex sed lex

## Une comonade doit respecter des lois

- $(\text{fmap } (\text{fmap } f)) \cdot \text{duplicate} \equiv \text{duplicate} \cdot \text{fmap } f$
- $\text{duplicate} \cdot \text{duplicate} = \text{fmap duplicate} \cdot \text{duplicate}$
- $\text{duplicate} \cdot \text{duplicate} \equiv \text{fmap duplicate} \cdot \text{duplicate}$   
(commut)
- $\text{fmap extract} \cdot \text{duplicate} \equiv \text{extract} \cdot \text{duplicate} \equiv \text{id } (\text{co unit})$



## Comonades - Catégories

Une comonade  $(T, \delta, \epsilon)$  est la donnée d'un endofoncteur  $T : C \rightarrow C$  et de deux transformations naturelles  $\Delta : T \rightarrow T \circ T$  et  $\epsilon : T \rightarrow 1_C$  telles que :

$$\begin{array}{ccc}
 T(X) & \xrightarrow{\Delta_X} & T(T(X)) \\
 \Delta_X \downarrow & & \downarrow \Delta_{T(X)} \\
 T(T(X)) & \xrightarrow{T(\Delta_X)} & T(T(T(X)))
 \end{array}
 \qquad
 \begin{array}{ccc}
 T(X) & \xrightarrow{\Delta_X} & T(T(X)) \\
 \Delta_X \downarrow & \searrow & \downarrow \epsilon_{T(X)} \\
 T(T(X)) & \xrightarrow{T(\epsilon_X)} & T(X)
 \end{array}$$

c'est à dire  $\Delta_T \circ \Delta = T\Delta \circ \Delta$  et  $T\epsilon \circ \Delta = \epsilon_T \circ \Delta = id$ .

# A chaque loi son diagramme

`extract` est une T.N.

`f . extract`  $\equiv$  `extract . (fmap f)`

$$\begin{array}{ccc}
 X & \xrightarrow{f} & Y \\
 \uparrow \epsilon_X & & \uparrow \epsilon_Y \\
 T(X) & \xrightarrow{T(f)} & T(Y)
 \end{array}$$

# A chaque loi son diagramme

`duplicate` est une T.N.

`(fmap (fmap f)) . duplicate ≡ duplicate . fmap f`

$$\begin{array}{ccc}
 T(X) & \xrightarrow{T(f)} & T(Y) \\
 \Delta_X \downarrow & & \downarrow \Delta_Y \\
 T(T(X)) & \xrightarrow{T(T(f))} & T(T(Y))
 \end{array}$$

# A chaque loi son diagramme

## Coassociativité

`duplicate . duplicate = fmap duplicate . duplicate`

$$\begin{array}{ccc}
 T(X) & \xrightarrow{\Delta_X} & T(T(X)) \\
 \Delta_X \downarrow & & \downarrow \Delta_{T(X)} \\
 T(T(X)) & \xrightarrow{T(\Delta_X)} & T(T(T(X)))
 \end{array}$$

$$\Delta_T \circ \Delta = T\Delta \circ \Delta$$

# A chaque loi son diagramme

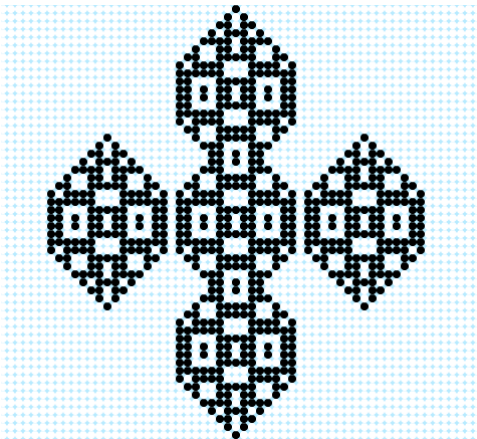
## Existence d'une counité

```
extract . duplicate = fmap extract . duplicate = id
```

$$\begin{array}{ccc}
 T(X) & \xrightarrow{\Delta_X} & T(T(X)) \\
 \Delta_X \downarrow & \searrow & \downarrow \epsilon_{T(X)} \\
 T(T(X)) & \xrightarrow{T(\epsilon_X)} & T(X)
 \end{array}$$

$$\epsilon_T \circ \Delta = T\epsilon \circ \Delta = id_T$$

## Evaluer un automate est comonadique



# L'univers



## Un ruban

On représente l'univers dans lequel vit notre automate par un ruban, que l'on voit comme trois parties :

- La partie infinie à gauche
- La case observée
- La partie infinie à droite

```
data Universe a = Universe [a] a [a]
```

# Quelques opérations sur notre univers



## Voyageons

On s'autorise à effectuer quelques opérations raisonnables sur notre univers :

- Regarder à gauche (left shift)
- Regarder à droite (right shift)

Moralement, on *translate* notre ruban.

left, right : : Universe  $a \rightarrow$  Universe  $a$



# Quelques opérations sur notre univers



## Voyageons

On s'autorise à effectuer quelques opérations raisonnables sur notre univers :

- Regarder à gauche (left shift)
- Regarder à droite (right shift)

Moralement, on *translate* notre ruban.

left, right : : Universe  $a \rightarrow$  Universe  $a$

# Quelques opérations sur notre univers



## Voyageons

On s'autorise à effectuer quelques opérations raisonnables sur notre univers :

- Regarder à gauche (left shift)
- Regarder à droite (right shift)

Moralement, on *translate* notre ruban.

left, right : : Universe  $a \rightarrow$  Universe  $a$

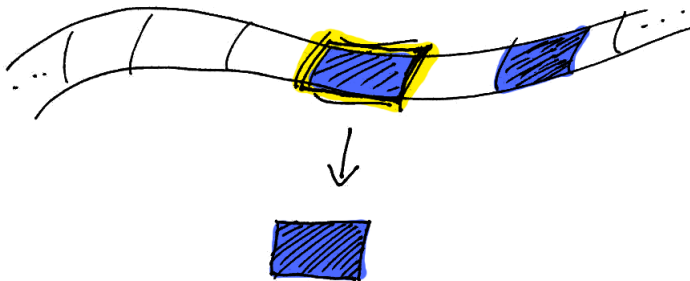
# L'univers est fonctoriel

## Un foncteur

Notre ruban est naturellement un foncteur : il suffit d'appliquer à notre `Universe a` une fonction `a -> b` sur chacune des cellules pour obtenir un `Universe b`.

```
fmap :: (a -> b) -> Universe a -> Universe b
```

# Comonades, nous voilà : extract

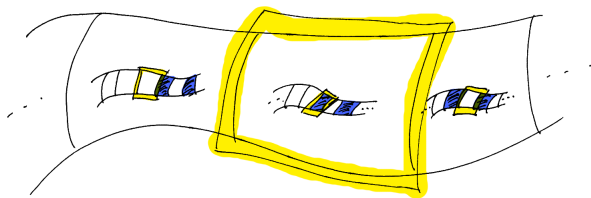


## Extraire une information

Depuis notre univers, on peut extraire une valeur : celle de la case que l'on est en train d'observer !

`extract : Universe a -> a`

# Comonades, nous voilà : duplicate

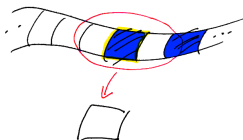


## L'opération duplicate

On veut construire un univers où chaque case du ruban contient elle-même... un univers. Il s'agit de contenir tous les shift possible de notre univers de départ.

duplicate : : Universe  $a \rightarrow$  Universe (Universe  $a$ )

# Loi de convolution



## La loi de notre automate

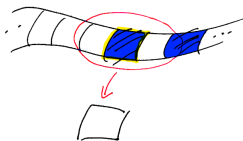
Notre automate est décrit par une fonction qui, à un univers, associe l'état de la cellule observé à la prochaine itération. On a donc accès à tout l'univers.

## Rule 30

Pour Rule 30, on a besoin de la cellule couramment observée, et de ses voisines de droite et de gauche.

rule : : Universe  $a \rightarrow a$

# Loi de convolution



## La loi de notre automate

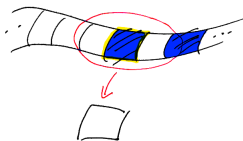
Notre automate est décrit par une fonction qui, à un univers, associe l'état de la cellule observé à la prochaine itération. On a donc accès à tout l'univers.

## Rule 30

Pour Rule 30, on a besoin de la cellule couramment observée, et de ses voisines de droite et de gauche.

rule : : Universe  $a \rightarrow a$

# Loi de convolution



## La loi de notre automate

Notre automate est décrit par une fonction qui, à un univers, associe l'état de la cellule observé à la prochaine itération. On a donc accès à tout l'univers.

## Rule 30

Pour Rule 30, on a besoin de la cellule couramment observée, et de ses voisines de droite et de gauche.

rule : : Universe  $a \rightarrow a$



# L'évaluation est comonadique

## Comment obtenir l'itération $n+1$ depuis l'itération $n$ ?

Nous disposons maintenant de tous les outils pour, en une ligne, décrire l'itération au rang  $n + 1$  depuis l'univers au rang  $n$ .

La pipeline :

- On duplique notre univers :  
duplicate : : Universe a -> Universe (Universe a)
- On map notre règle sur chaque case :  
fmap rule : : Universe (Universe a) -> Universe a

fmap rule . duplicate : : Universe a -> Universe a

# L'évaluation est comonadique

## Comment obtenir l'itération $n+1$ depuis l'itération $n$ ?

Nous disposons maintenant de tous les outils pour, en une ligne, décrire l'itération au rang  $n + 1$  depuis l'univers au rang  $n$ .

### La pipeline :

- On duplique notre univers :  
duplicate : : Universe a -> Universe (Universe a)
- On map notre règle sur chaque case :  
fmap rule : : Universe (Universe a) -> Universe a

fmap rule . duplicate : : Universe a -> Universe a

# L'évaluation est comonadique

## Comment obtenir l'itération $n+1$ depuis l'itération $n$ ?

Nous disposons maintenant de tous les outils pour, en une ligne, décrire l'itération au rang  $n + 1$  depuis l'univers au rang  $n$ .

### La pipeline :

- On duplique notre univers :  
duplicate : : Universe a -> Universe (Universe a)
- On map notre règle sur chaque case :  
fmap rule : : Universe (Universe a) -> Universe a

fmap rule . duplicate : : Universe a -> Universe a

## Live démo

```

*Automata> showLife 50 10 rule_lr duo_universe

*Automata> showLife 90 40 rule_30 single_universe

*Automata>

```

```

-----
- COMONADE -
-----

{- Extrait la valeur courante -}
extract :: Universe a -> a
extract (Universe _ v _) = v

{- C'est un foncteur -}
instance Functor Universe where
  fmap f (Universe xs v ys) = Universe (fmap f xs) (f v) (fmap f ys)

{- Construit un univers de tous les univers translaté -}
duplicate :: Universe a -> Universe (Universe a)
duplicate u = Universe (tail $ iterate left u) u (tail $ iterate right u)

-----
- CONVOLUTION -
-----

{- Les règles expriment si la case observée doit
être vivante ou morte à la prochaine itération -}

{- Règle : Vivant si : Gauche != Droite -}
rule_lr :: Universe Bool -> Bool
rule_lr u = lv /= rv
  where
    lv = extract . left $ u
    rv = extract . right $ u

{- Application d'une règle -}
next :: (Universe a -> a) -> Universe a -> Universe a
next rule universe = fmap rule $ duplicate universe

[ Règle 30 -]
rule_30 :: Universe Bool -> Bool
rule_30 u = toB $ case (toN lv, toN v, toN rv) of
  (1, 1, 1) -> 0
  (1, 1, 0) -> 0
  (1, 0, 1) -> 0
  (1, 0, 0) -> 1
  (0, 1, 1) -> 1
  (0, 1, 0) -> 1
  (0, 0, 1) -> 1
  (0, 0, 0) -> 0
  where
    lv = extract . left $ u
    v = extract u
    rv = extract . right $ u

-----
Quelques valeurs prédéfinies :
-----

single_universe :: Universe Bool

```



Merci pour votre attention !