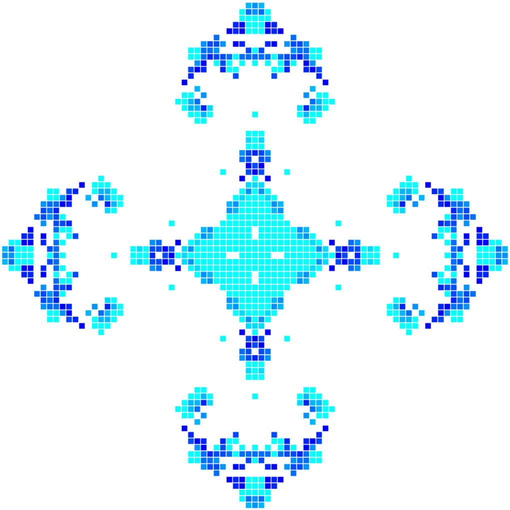
Foncteurs, Monades et Zippers

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 :

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

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

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

```
irb(main):066:0> sum.(2,3)
=> 5
```

```
irb(main):070:0> sum.curry.(1)
=> #<Proc:0x00000000288c3c0 (lambda)>
irb(main):071:0> sum.curry.(1).(2)
=> 3
```

Fonctions à plusieurs paramètres

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

```
irb(main):066:0> sum.(2,3)
=> 5
```

```
irb(main):070:0> sum.curry.(1)
=> #<Proc:0x00000000288c3c0 (lambda)>
irb(main):071:0> sum.curry.(1).(2)
=> 3
```

Fonctions à plusieurs paramètres

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

```
irb(main):066:0> sum.(2,3)
=> 5
```

```
irb(main):070:0> sum.curry.(1)
=> #<Proc:0x00000000288c3c0 (lambda)>
irb(main):071:0> 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

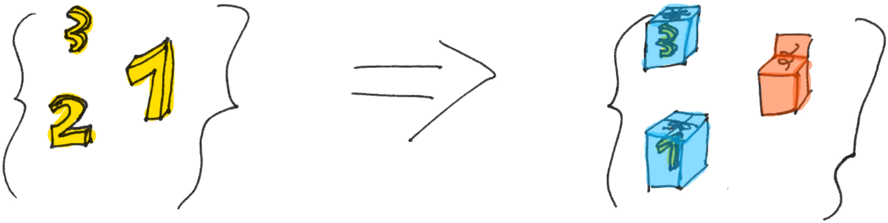
- a => [a]
- a => Tree a
- a => Maybe a

... et sur les fonctions

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

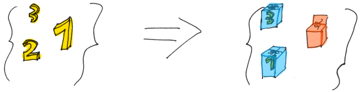
- fmap 2.method(:+2) : : F Int -> F Int
- fmap floor : : F Float -> F Int

Donnée dans un contexte



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

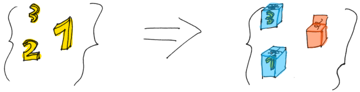
Maybe : Une implémentation




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

```
irb(main):025:0> Just.(3)
=> #<Just:0x00000000359c010 @content=[3]>
```

Maybe : Une implémentation



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

```
irb(main):026:0> Nothing.()
=> #<Nothing:0x000000002d97d38 @content=[]>
```

Maybe : Une implémentation

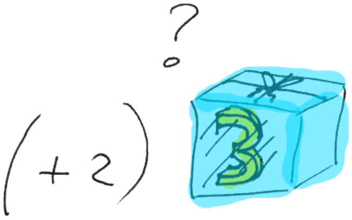
```ruby
class Maybe
  private_class_method :new

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

```
irb(main):053:0> Just.(3).from_maybe
=> 3
irb(main):054:0> Nothing.().from_maybe
=> nil
```

```ruby
case box
when Nothing
  puts "It's␣nothing␣dear."
when Just
  puts "It's␣just␣#{box.from_maybe}."
end
```

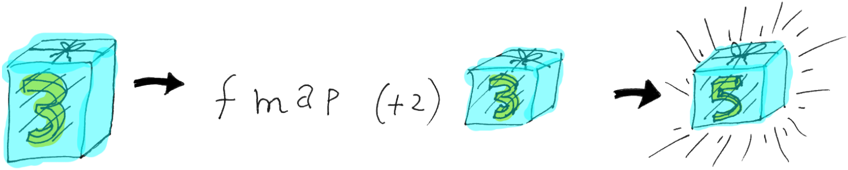
Functorial mapping

On ne peut plus appliquer la fonction telle quelle :



Functorial mapping

Mais le foncteur nous donne une nouvelle flèche.



Dura lex sed lex

Un foncteur doit respecter des lois

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

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)

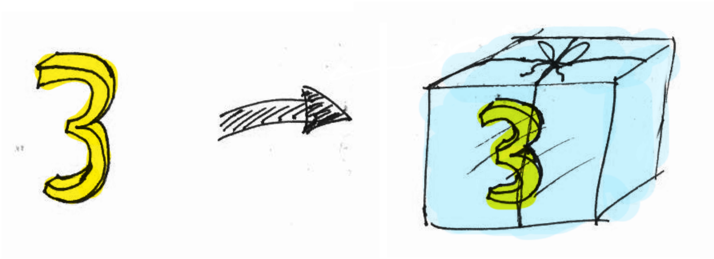
Un foncteur est un endofoncteur de la catégorie des types.

Monades


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

Donnée dans un contexte

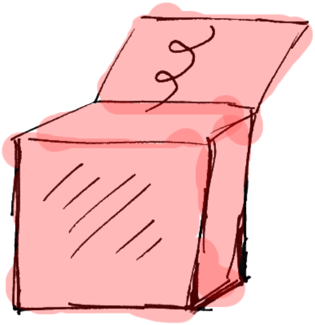
Une monade place une valeur dans un contexte.



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

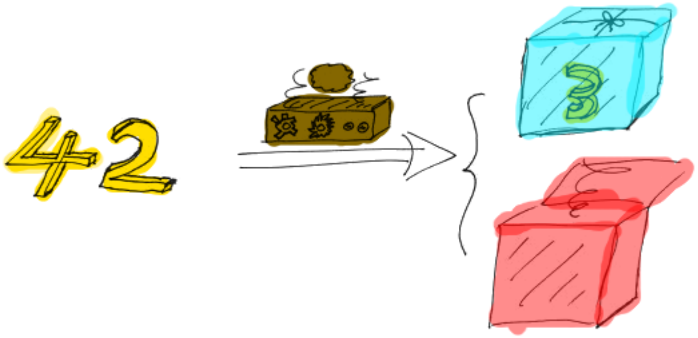
L'opérateur *pure*

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

Quelques cas particuliers

- Just
- (: [])
- Right

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

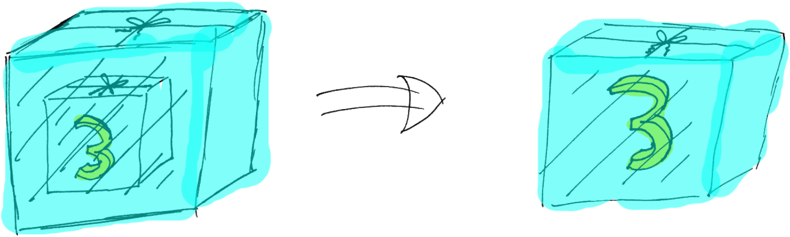
Comment composer `f :: a -> M b` et `g :: b -> M c` ?

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

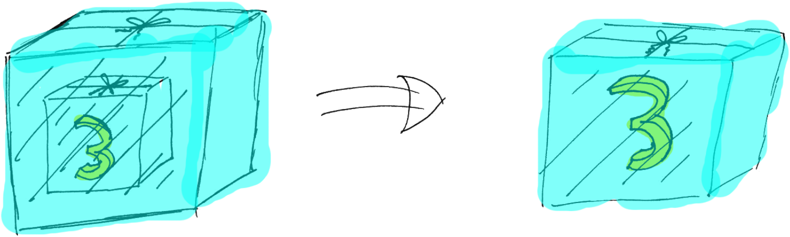
```
join :: M (M a) -> M a
```




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

L'opérateur join

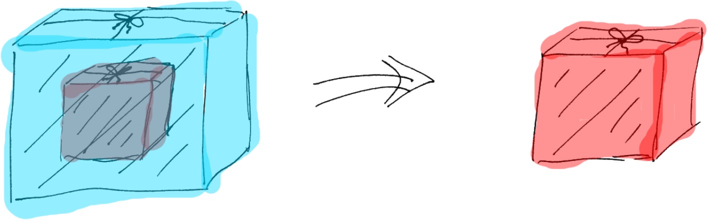
```
join :: M (M a) -> M a
```



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

L'opérateur join

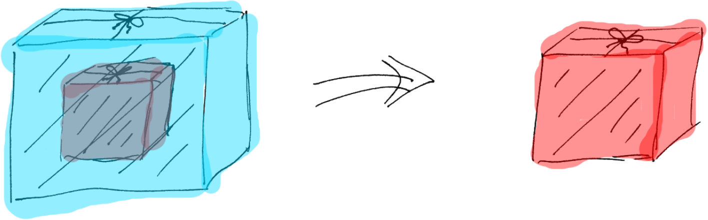
```
join :: M (M a) -> M a
```



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

L'opérateur join

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



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

L'opérateur *bind*

On cherche à définir la composition.

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

Nous avons :

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

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

`f >=> g` ≡ `join . (fmap g) . f`.

L'opérateur *bind*

On cherche à définir la composition.

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

Nous avons :

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

On peut maintenant composer f et g.

`f >=> g` $\equiv$ `join . (fmap g) . f`.

L'opérateur *bind*

On cherche à définir la composition.

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

Nous avons :

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

On peut maintenant composer f et g.

`f >=> g` $\equiv$ `join . (fmap g) . f.`

Récapitulatif

Une monade, c'est

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

Dura lex sed lex

Une monade doit respecter des lois

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

- `join . fmap join ≡ join . join`
- `join . fmap pure ≡ join . 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 :

$$
\begin{tikzcd}
T(T(T(X))) \arrow[r, "T(\mu_X)"] \arrow[d, "\mu_{T(X)}"'] & T(T(X)) \arrow[d, "\mu_X"] \\
T(T(X)) \arrow[r, "\mu_X"'] & T(X)
\end{tikzcd}
\qquad
\begin{tikzcd}
T(X) \arrow[r, "\eta_{T(X)}"] \arrow[d, "T(\eta_X)"'] \arrow[dr, equal] & T(T(X)) \arrow[d, "\mu_X"] \\
T(T(X)) \arrow[r, "\mu_X"'] & T(X)
\end{tikzcd}
$$

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 est 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 :

$$
\begin{tikzcd}
T(T(T(X))) \arrow[r, "T(\mu_X)"] \arrow[d, "\mu_{T(X)}"'] & T(T(X)) \arrow[d, "\mu_X"] \\
T(T(X)) \arrow[r, "\mu_X"'] & T(X)
\end{tikzcd}
\qquad
\begin{tikzcd}
T(X) \arrow[r, "\eta_{T(X)}"] \arrow[d, "T(\eta_X)"'] \arrow[dr, equal] & T(T(X)) \arrow[d, "\mu_X"] \\
T(T(X)) \arrow[r, "\mu_X"'] & T(X)
\end{tikzcd}
$$

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 est la catégorie des types.

A chaque loi son diagramme

pure est une T.N.

```
pure . f ≡ (fmap f) . pure
```

$$\begin{tikzcd} X \arrow[r, "f"] \arrow[d, "\eta_X"'] & Y \arrow[d, "\eta_Y"] \\ T(X) \arrow[r, "T(f)"'] & T(Y) \end{tikzcd}$$

A chaque loi son diagramme

join est une T.N.

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

$$
\begin{tikzcd}
T(T(X)) \arrow[r, "T(T(f))"] \arrow[d, "\mu_X"'] & T(T(Y)) \arrow[d, "\mu_Y"] \\
T(X) \arrow[r, "T(f)"'] & T(Y)
\end{tikzcd}
$$

A chaque loi son diagramme

Associativité

`join . fmap join` $\equiv$ `join . join`

$$\begin{tikzcd} T(T(T(X))) \arrow[r, "T(\mu_X)"] \arrow[d, "\mu_{T(X)}"'] & T(T(X)) \arrow[d, "\mu_X"] \\ T(T(X)) \arrow[r, "\mu_X"'] & T(X) \end{tikzcd}$$

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

A chaque loi son diagramme

Existence d'un neutre

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

$$
\begin{tikzcd}
T(X) \arrow[r, "\eta_{T(X)}"] \arrow[d, "T(\eta_X)"'] \arrow[dr, equal] & T(T(X)) \arrow[d, "\mu_X"] \\
T(T(X)) \arrow[r, "\mu_X"'] & T(X)
\end{tikzcd}
$$

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

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.

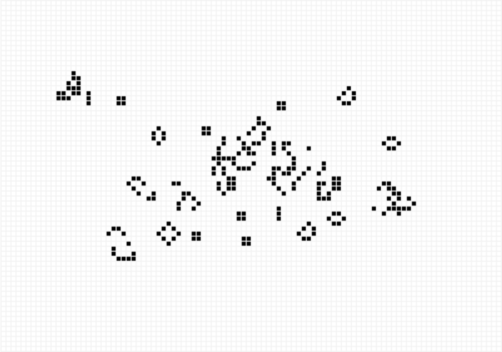
Combien d'automates cellulaires différents ?

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.

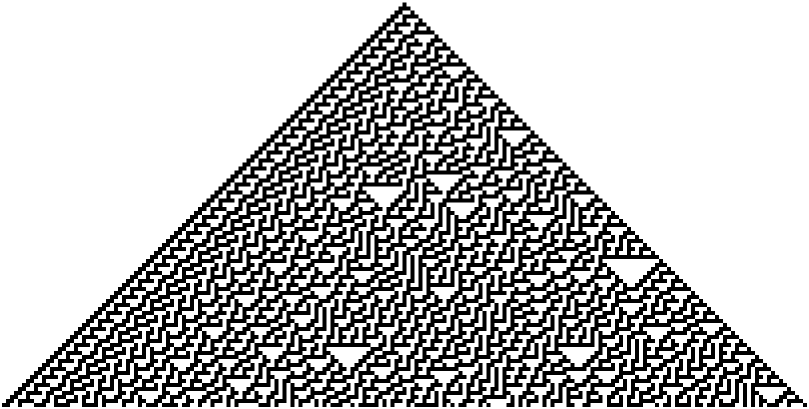
The "Game of Life"

Jeu de la vie (J. H. Conway)



Étude d'un cas : Rule 30

Rule 30



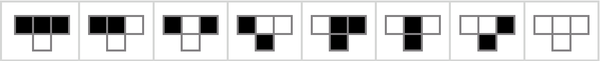
La grille

La grille de l'automate

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

Un voisinage de 3 cellules.

Les règles

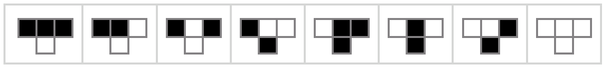


On peut aussi écrire :

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

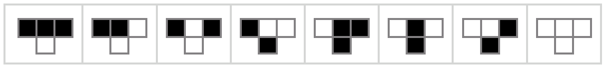


On peut aussi écrire :

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
- extract (copure) (co uinit) : : M a -> a
- duplicate (cojoin) (co product δ) : : M a -> M (M a)

Dura lex sed lex

Une comonade doit respecter des lois

- `(fmap (fmap f)) . duplicate` $\equiv$ `duplicate . fmap f`
- `duplicate . duplicate = fmap duplicate . duplicate`
- `duplicate . duplicate` $\equiv$ `fmap duplicate . duplicate` (commut)
- `fmap extract . duplicate` $\equiv$ `extract . duplicate` $\equiv$ `id` (co unit)

Comonades - Catégories

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

$$
\begin{tikzcd}
T(X) \arrow[r, "\Delta_X"] \arrow[d, "\Delta_X"'] & T(T(X)) \arrow[d, "\Delta_{T(X)}"] \\
T(T(X)) \arrow[r, "T(\Delta_X)"'] & T(T(T(X)))
\end{tikzcd}
\qquad
\begin{tikzcd}
T(X) \arrow[r, "\Delta_X"] \arrow[d, "\Delta_X"'] \arrow[dr, equal] & T(T(X)) \arrow[d, "\epsilon_{T(X)}"] \\
T(T(X)) \arrow[r, "T(\epsilon_X)"'] & T(X)
\end{tikzcd}
$$

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 ≡ extract . (fmap f)`

$$
\begin{tikzcd}
X \arrow[r, "f"] & Y \\
T(X) \arrow[u, "\epsilon_X"'] \arrow[r, "T(f)"'] & T(Y) \arrow[u, "\epsilon_Y"]
\end{tikzcd}
$$

A chaque loi son diagramme

duplicate est une T.N.

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

$$
\begin{tikzcd}
T(X) \arrow[r, "T(f)"'] \arrow[d, "\Delta_X"'] & T(Y) \arrow[d, "\Delta_Y"] \\
T(T(X)) \arrow[r, "T(T(f))"] & T(T(Y))
\end{tikzcd}
$$

A chaque loi son diagramme

Coassociativité

```
duplicate . duplicate = fmap duplicate . duplicate
```

$$\begin{tikzcd} T(X) \arrow[r, "\Delta_X"] \arrow[d, "\Delta_X"'] & T(T(X)) \arrow[d, "\Delta_{T(X)}"] \\ T(T(X)) \arrow[r, "T(\Delta_X)"'] & T(T(T(X))) \end{tikzcd}$$

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

A chaque loi son diagramme

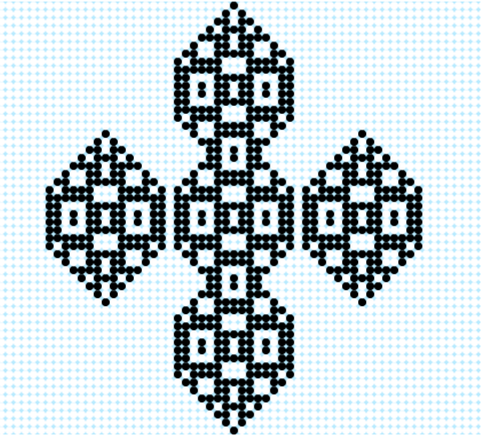
Existence d'une counité

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

$$\begin{tikzcd} T(X) \arrow[r, "\Delta_X"] \arrow[d, "\Delta_X"'] \arrow[dr, equal] & T(T(X)) \arrow[d, "\epsilon_{T(X)}"] \\ T(T(X)) \arrow[r, "T(\epsilon_X)"'] & T(X) \end{tikzcd}$$

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

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 -> 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 -> 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 -> Universe a

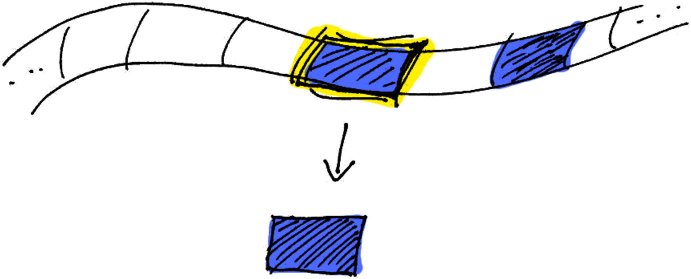
L'univers est fonctoriel

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

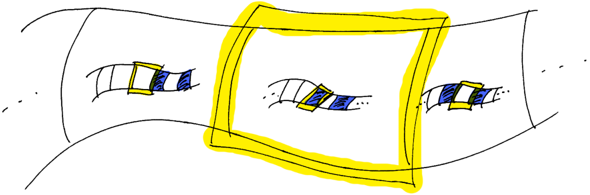


Extraire une information

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

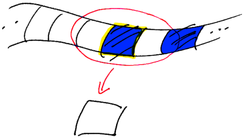


L'opération duplicate

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 -> Universe (Universe a)

Loi de convolution



La loi de notre automate

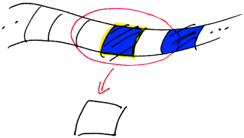
Notre automate est décrit par une fonction qui, à un univers, associe l'état de la cellule observé à la prochaine itération. On a donc accès à tout l'univers.

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 -> a

Loi de convolution



La loi de notre automate

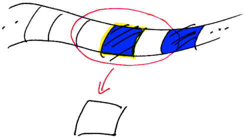
Notre automate est décrit par une fonction qui, à un univers, associe l'état de la cellule observé à la prochaine itération. On a donc accès à tout l'univers.

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 -> 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 -> 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 :

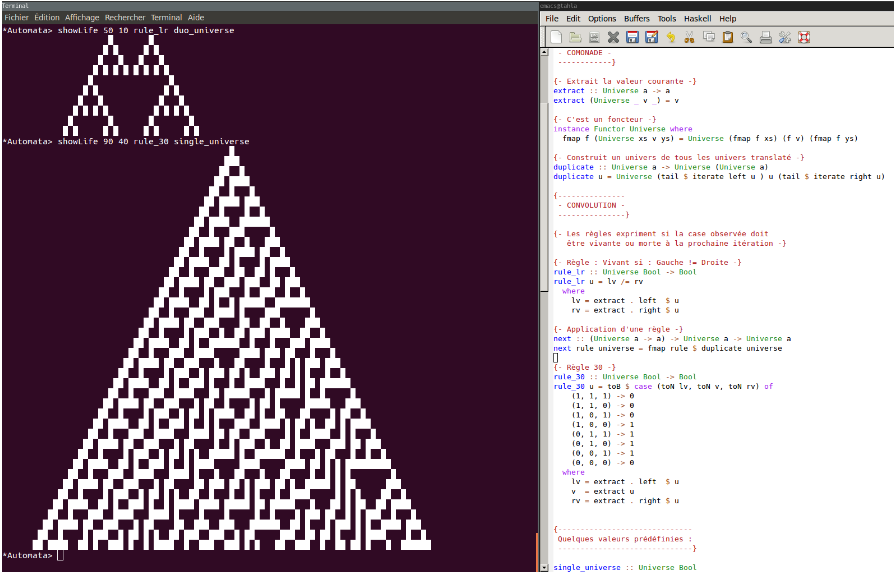
- On duplique notre univers :

  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


Terminal
Fichier Édition Affichage Rechercher Terminal Aide
*Automata> showLife 50 10 rule_lr duo_universe
*Automata> showLife 90 40 rule_30 single_universe
*Automata>
emacs@tahla
File Edit Options Buffers Tools Haskell Help
- 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 !