

Monades, Comonades et Automates cellulaires

Jérémy S. Cochoy

INRIA Paris-Saclay

Octobre 2015

1 Monades

- Types
- Fonctions
- Foncteurs
- Monades

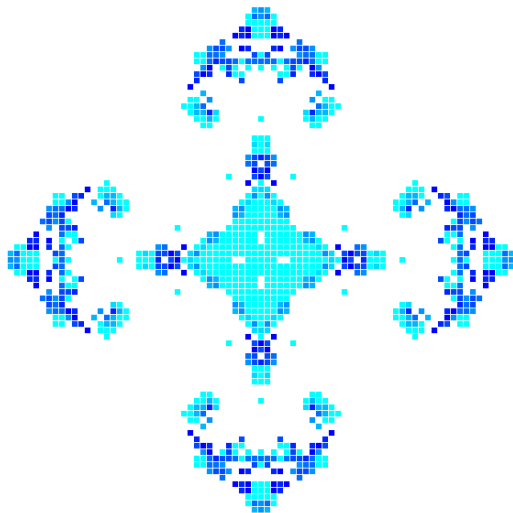
2 Automates Cellulaires

- 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 types

Exemples :

- $Int = \{-2147483648, \dots, 2147483647\}$
- $[Bool] = \{[], [True], [False], [True, False], [False, True], \dots\}$

Construire son type :

- $Trival = Plus \mid Minus \mid Zero$
- $Box\ a = InABox\ a$
- $Maybe\ a = Just\ a \mid Nothing$

$Just$, $Nothing$, $InABox$ etc portent le doux nom de *constructeur de type*.
C'est aussi le cas de $[]$.

Les types

Exemples :

- $Int = \{-2147483648, \dots, 2147483647\}$
- $[Bool] = \{[], [True], [False], [True, False], [False, True], \dots\}$

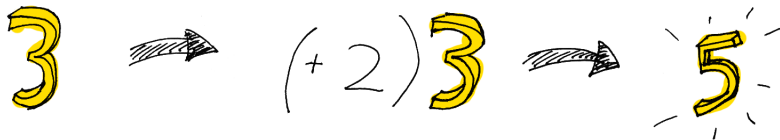
Construire son type :

- $Trival = Plus \mid Minus \mid Zero$
- $Box\ a = InABox\ a$
- $Maybe\ a = Just\ a \mid Nothing$

$Just$, $Nothing$, $InABox$ etc portent le doux nom de *constructeur de type*.
C'est aussi le cas de $[]$.

Les fonctions

Ce sont les traitements que l'on peut implémenter.



Les fonctions

Une fonction a aussi un type : $a \rightarrow b$

- $\text{floor} :: \text{Float} \rightarrow \text{Int}$
- $(+2) :: \text{Int} \rightarrow \text{Int}$
- $\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$

Les fonctions

Ça se compose

- $f1 :: a \rightarrow b$
- $f2 :: b \rightarrow c$
- $f2 . f1 :: a \rightarrow c$
- $. :: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c)$

La collection de tous les types forme une catégorie. Les flèches sont les fonctions implémentables. On l'appelle la catégorie des types.

Les fonctions

Ça se compose

- $f1 :: a \rightarrow b$
- $f2 :: b \rightarrow c$
- $f2 . f1 :: a \rightarrow c$
- $. :: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c)$

La collection de tous les types forme une catégorie. Les flèches sont les fonctions implémentables. On l'appelle la catégorie des types.

Les fonctions

Ça se compose

- $f1 :: a \rightarrow b$
- $f2 :: b \rightarrow c$
- $f2 . f1 :: a \rightarrow c$
- $. :: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c)$

La collection de tous les types forme une catégorie. Les flèches sont les fonctions implémentables. On l'appelle la catégorie des types.

Les foncteurs

Un foncteur F agit sur les types ...

- $a \Rightarrow F\ a$

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

- $a \Rightarrow [a]$

... et sur les fonctions

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

- $\text{fmap } (+2) :: F\ \text{Int} \rightarrow F\ \text{Int}$

- $\text{fmap id} :: F\ a \rightarrow F\ a$

Les foncteurs

Un foncteur F agit sur les types ...

- $a \Rightarrow F\ a$

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

- $a \Rightarrow [a]$

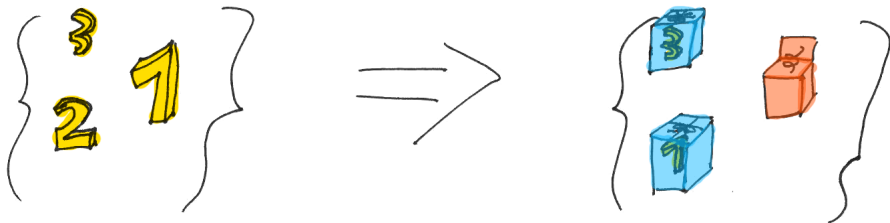
... et sur les fonctions

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

- $\text{fmap } (+2) :: F\ \text{Int} \rightarrow F\ \text{Int}$

- $\text{fmap id} :: F\ a \rightarrow F\ a$

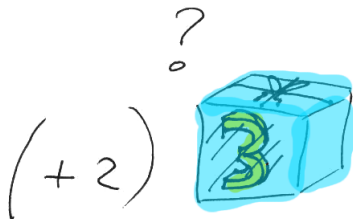
Donnée dans un contexte



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

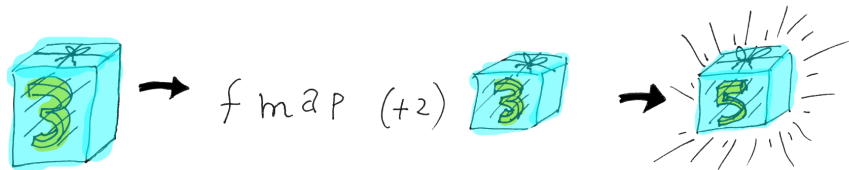
Functorial mapping

On ne peut plus appliquer la fonction telle quelle :



Functorial mapping

Mais le foncteur nous donne une nouvelle flèche.

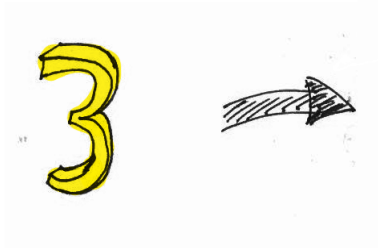


Monades



Donnée dans un contexte

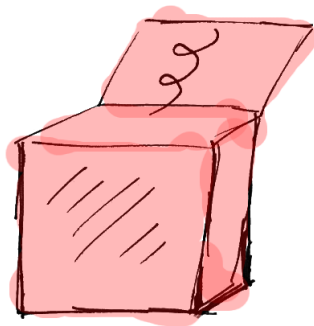
Une monade place une valeur dans un contexte.



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

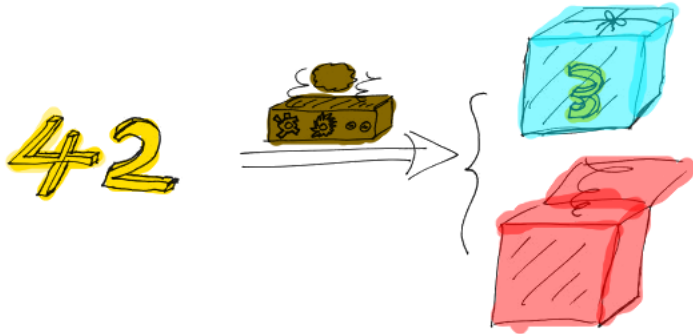
L'opérateur *pure*

`pure :: a -> F a`

Quelques cas particuliers

- Just
- (: [])
- Right

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

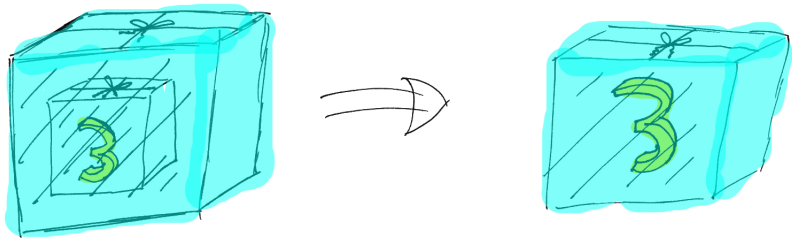
Comment composer $f :: a \rightarrow M\ b$ et $g :: b \rightarrow M\ c$?

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

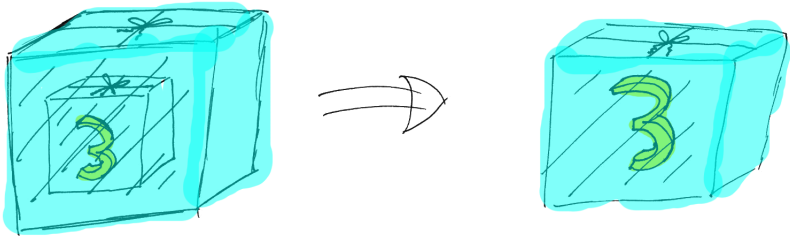
```
join :: M (M a) -> M a
```



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

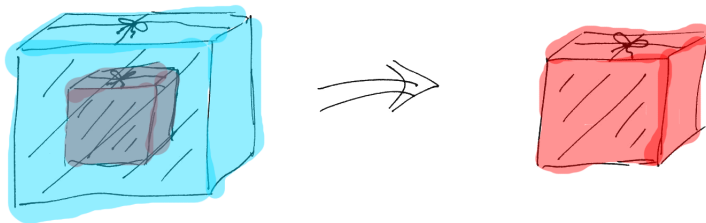

L'opérateur join

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



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

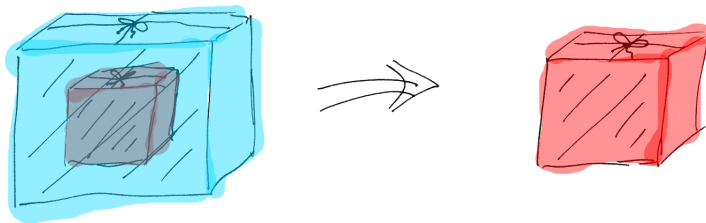
L'opérateur join

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


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

L'opérateur join

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



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

L'opérateur *bind*

On cherche à définir la composition.

$$(=>) :: (a \rightarrow M\ b) \rightarrow (b \rightarrow M\ c) \rightarrow (a \rightarrow M\ c)$$

Nous avons :

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

On peut maintenant composer f et g .

$$f \Rightarrow g \equiv \text{join} . (\text{fmap } g) . f.$$

L'opérateur *bind*

On cherche à définir la composition.

$$(=>) :: (a \rightarrow M\ b) \rightarrow (b \rightarrow M\ c) \rightarrow (a \rightarrow M\ c)$$

Nous avons :

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

On peut maintenant composer f et g .

$$f \Rightarrow g \equiv \text{join} . (\text{fmap } g) . f.$$

L'opérateur *bind*

On cherche à définir la composition.

$$(=>) :: (a \rightarrow M\ b) \rightarrow (b \rightarrow M\ c) \rightarrow (a \rightarrow M\ c)$$

Nous avons :

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

On peut maintenant composer f et g .

$$f \Rightarrow g \equiv \text{join} . (\text{fmap } g) . f.$$

Récapitulatif

Une monade, c'est

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

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.

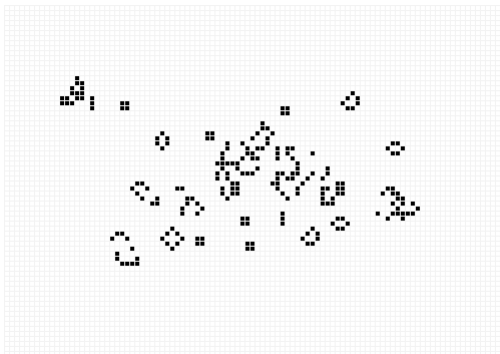
Combien d'automates cellulaires différents ?

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.

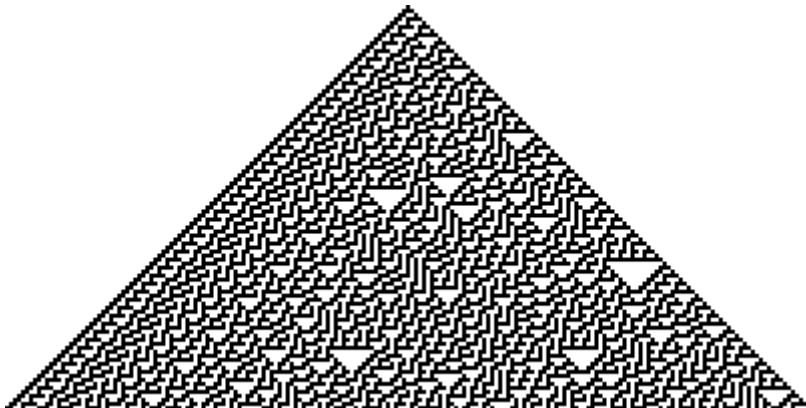
The "Game of Life"

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

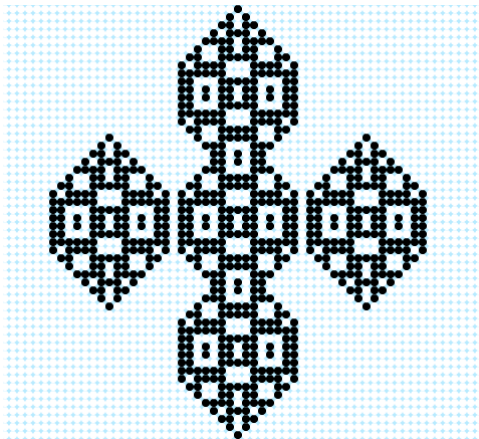
Comonades



C'est le dual d'une monade

- C'est un foncteur M
- `extract` (copure) (`co uinit`) : $M\ a \rightarrow a$
- `duplicate` (cojoin) (`co product` δ) : $M\ a \rightarrow M\ (M\ a)$

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

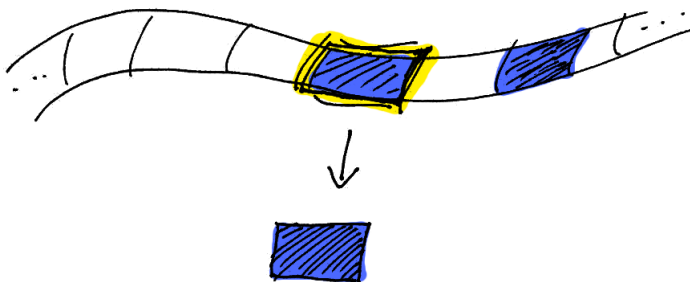
L'univers est fonctoriel

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

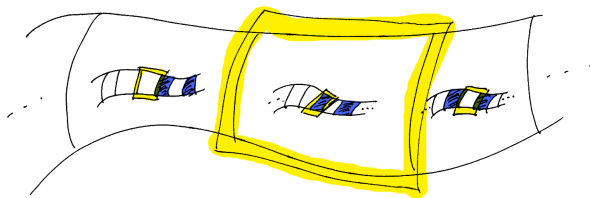


Extraire une information

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

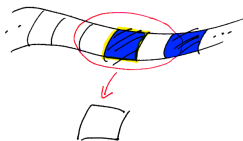


L'opération duplicate

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

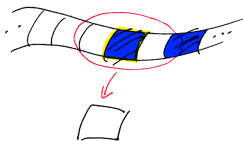
Notre automate est décrit par une fonction qui, à un univers, associe l'état de la cellule observé à la prochaine itération. On a donc accès à tout l'univers.

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

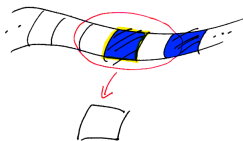
Notre automate est décrit par une fonction qui, à un univers, associe l'état de la cellule observé à la prochaine itération. On a donc accès à tout l'univers.

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

The screenshot displays a Haskell environment with two panes. The left pane shows the execution of two commands in the Automata interpreter:

```
*Automata> showLife 50 10 rule_lr duo_universe
*Automata> showLife 90 40 rule_30 single_universe
```

The first command produces a small, symmetrical pattern of black and white pixels. The second command produces a much larger, more complex pattern that resembles a fractal or a complex cellular structure.

The right pane shows the Haskell source code for the Automata interpreter, which implements Conway's Game of Life. The code is organized into sections: COMONADE, CONVOLUTION, and the main rules for the Game of Life. The code defines the `extract` function, the `fmap` function, and the `rule` function. It also defines the `rule_lr` and `rule_30` functions, which are used to generate the patterns shown in the left pane.

```
-----
- 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 !