

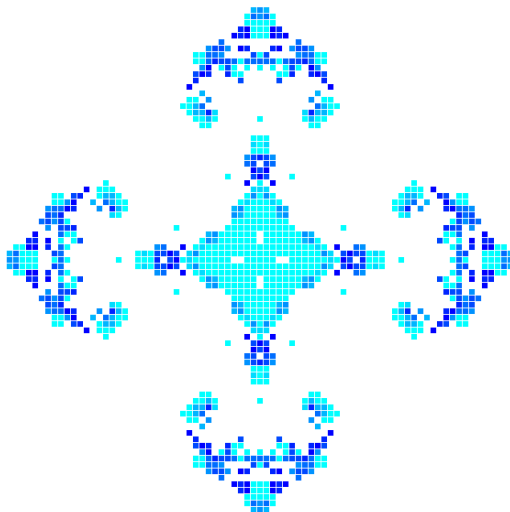
Monades, Comonades et Automates cellulaires

Jérémy S. Cochoy

INRIA Paris-Saclay

Octobre 2015

- 1 Monades
 - Types
 - Fonctions
 - Foncteurs
 - Monades
- 2 Automates Cellulaires
- 3 Comonades
- 4 Évaluer un automate est comonadique
 - Un univers
 - Un foncteur
 - Une comonade
 - Evaluation



Les types

Qu'est-ce qu'un type ?

C'est un *ensemble* de valeurs.

Exemples :

- $Int = \{-2147483648, \dots, 2147483647\}$
- $Bool = \{True, False\}$
- $Char = \{'a', 'b', 'c', \dots\}$
- $[Bool] = \{[], [True], [False], [True, False], [False, True], \dots\}$
- $[a]$

Les types

Qu'est-ce qu'un type ?

C'est un *ensemble* de valeurs.

Exemples :

- $Int = \{-2147483648, \dots, 2147483647\}$
- $Bool = \{True, False\}$
- $Char = \{'a', 'b', 'c', \dots\}$
- $[Bool] = \{[], [True], [False], [True, False], [False, True], \dots\}$
- $[a]$

Les types

Qu'est-ce qu'un type ?

C'est un *ensemble* de valeurs.

Exemples :

- $Int = \{-2147483648, \dots, 2147483647\}$
- $Bool = \{True, False\}$
- $Char = \{'a', 'b', 'c', \dots\}$
- $[Bool] = \{[], [True], [False], [True, False], [False, True], \dots\}$
- $[a]$

Les types

Qu'est-ce qu'un type ?

C'est un *ensemble* de valeurs.

Exemples :

- $Int = \{-2147483648, \dots, 2147483647\}$
- $Bool = \{True, False\}$
- $Char = \{'a', 'b', 'c', \dots\}$
- $[Bool] = \{[], [True], [False], [True, False], [False, True], \dots\}$
- $[a]$

Les types

Construire son type :

- $\text{Trival} = \text{Plus} \mid \text{Minus} \mid \text{Zero}$
- $\text{Box } a = \text{InABox } a$
- $\text{Maybe } a = \text{Just } a \mid \text{Nothing}$
- $\text{Either } a \ b = \text{Left } a \mid \text{Right } b$

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

Les types

Construire son type :

- $\text{Trival} = \text{Plus} \mid \text{Minus} \mid \text{Zero}$
- $\text{Box } a = \text{InABox } a$
- $\text{Maybe } a = \text{Just } a \mid \text{Nothing}$
- $\text{Either } a \ b = \text{Left } a \mid \text{Right } b$

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

Les types

Construire son type :

- $\text{Trival} = \text{Plus} \mid \text{Minus} \mid \text{Zero}$
- $\text{Box } a = \text{InABox } a$
- $\text{Maybe } a = \text{Just } a \mid \text{Nothing}$
- $\text{Either } a \ b = \text{Left } a \mid \text{Right } b$

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

Les types

Construire son type :

- $\text{Trival} = \text{Plus} \mid \text{Minus} \mid \text{Zero}$
- $\text{Box } a = \text{InABox } a$
- $\text{Maybe } a = \text{Just } a \mid \text{Nothing}$
- $\text{Either } a \ b = \text{Left } a \mid \text{Right } b$

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

Les types

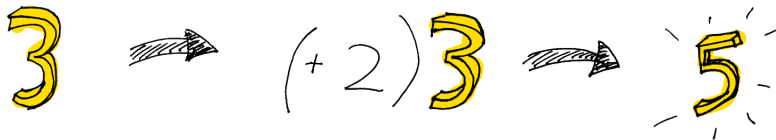
Construire son type :

- $\text{Trival} = \text{Plus} \mid \text{Minus} \mid \text{Zero}$
- $\text{Box } a = \text{InABox } a$
- $\text{Maybe } a = \text{Just } a \mid \text{Nothing}$
- $\text{Either } a \ b = \text{Left } a \mid \text{Right } b$

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

Les fonctions

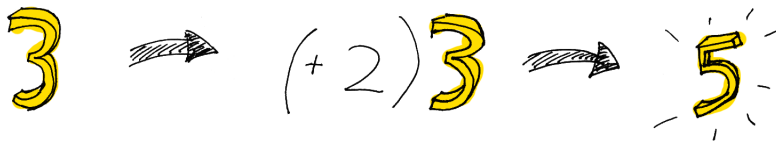
Ce sont les traitements que l'on peut implémenter.



Une fonction ne lance pas de missile.

Les fonctions

Ce sont les traitements que l'on peut implémenter.



Une fonction ne lance pas de missile.

Les fonctions

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

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

Les fonctions

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

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

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 ...

- $a \Rightarrow F\ a$

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

- $a \Rightarrow [a]$

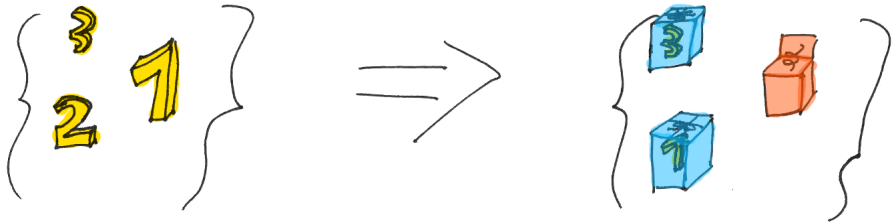
... et sur les fonctions

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

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

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

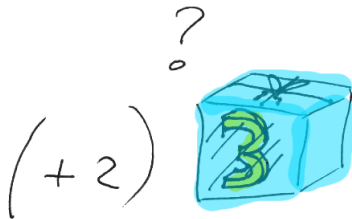
Donnée dans un contexte



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

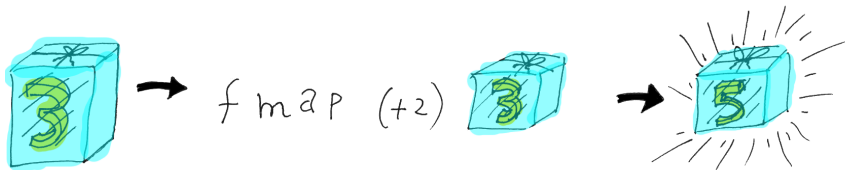
Functorial mapping

On ne peut plus appliquer la fonction telle quelle les diagrammes suivant commutent :



Functorial mapping

Mais le foncteur nous donne une nouvelle flèche.



Dura lex sed lex

Un foncteur doit respecter des lois

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

Un foncteur est un endofoncteur de la catégorie des types.

Dura lex sed lex

Un foncteur doit respecter des lois

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

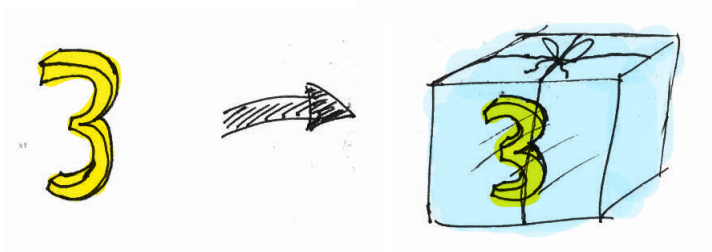
Un foncteur est un endofoncteur de la catégorie des types.

Monades



Donnée dans un contexte

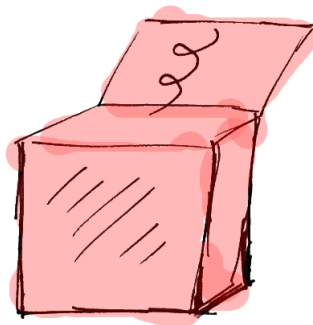
Une monade place une valeur dans un contexte.



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

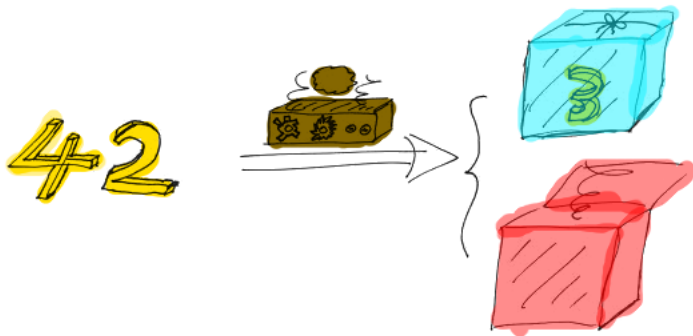
L'opérateur *pure*

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

Quelques cas particuliers

- Just
- (: [])
- 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

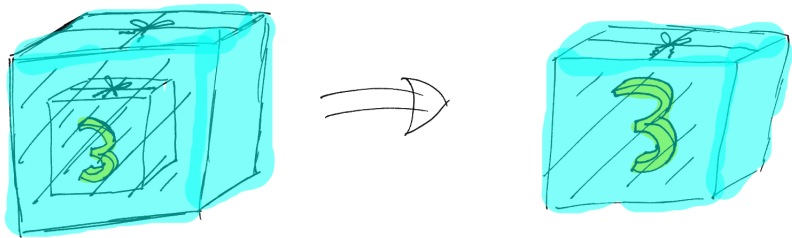
Comment composer $f :: a \rightarrow M\ b$ et $g :: b \rightarrow M\ c$?

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

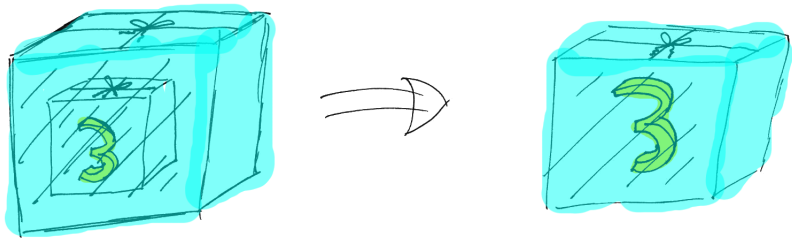
`join :: M (M a) -> M a`



`join $ Just (Just 3).`

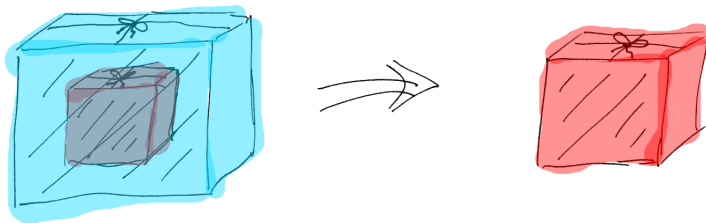
L'opérateur join

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



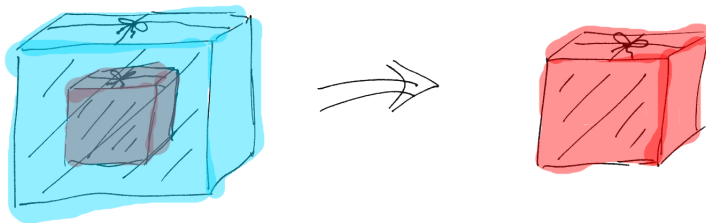
`join $ Just (Just 3).`

L'opérateur join

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


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

L'opérateur join

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

$$\text{join } \$ \text{ Just (Nothing)}.$$

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

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

Dura lex sed lex

Une monade doit respecter des lois

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

Monades - Catégories

Une monade (T, μ, η) 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 est la catégorie des types.

Monades - Catégories

Une monade (T, μ, η) 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 est la catégorie des types.

A chaque loi son diagramme

pure est une T.N.

$\text{pure} \cdot f \equiv (\text{fmap } f) \cdot \text{pure}$

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

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

TODO : COMMIT TEXTILE CONE

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 ,
- D'une fonction de transition qui à une cellule associe son nouvel état.

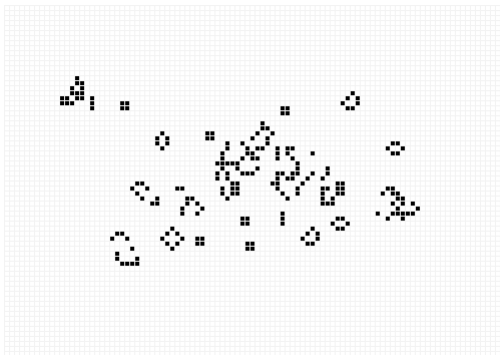
Combien d'automates cellulaires différents ?

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.

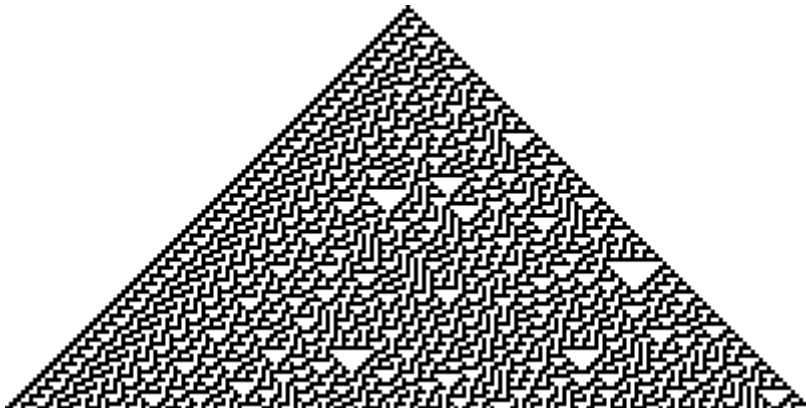
The "Game of Life"

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 unit) : : $M\ a \rightarrow a$
- duplicate (cojoin) (co product δ) : : $M\ a \rightarrow M\ (M\ a)$

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, δ, ϵ) est la donné 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

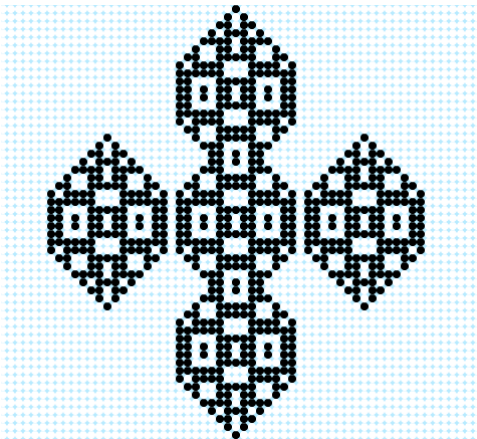
Existence d'une counité

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

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

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

Évaluer 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

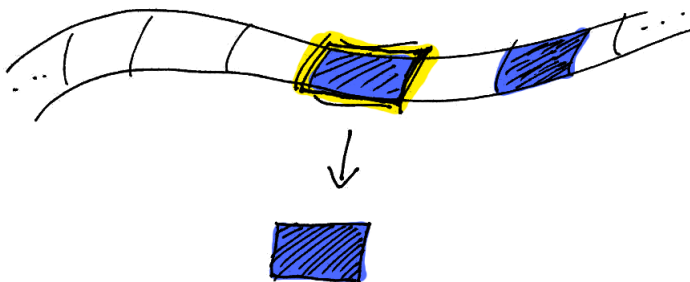
Univers est fonctoriel

Un foncteur

Notre ruban est naturellement un foncteur : Il suffit d'appliquer à notre *universe* a une fonction $a \rightarrow b$ sur chacune des cellules pour obtenir un *Universe* b .

$\text{fmap} :: (a \rightarrow b) \rightarrow \text{Universe } a \rightarrow \text{Universe } b$

Comonades, nous voilà : extract

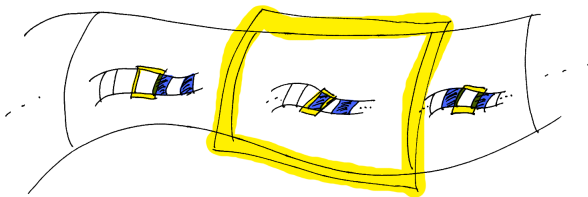


Extraire une information

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

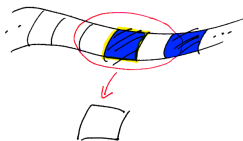


L'opération duplicate

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

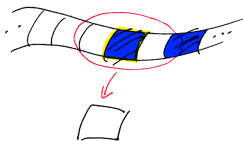
Notre automate est décrit par une fonction qui, à un univers, associe l'état de la cellule observé à la prochaine itération. On a donc accès à tout l'univers.

Rule 30

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

rule : : Universe $a \rightarrow a$

Loi de convolution



La loi de notre automate

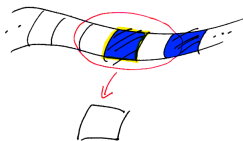
Notre automate est décrit par une fonction qui, à un univers, associe l'état de la cellule observé à la prochaine itération. On a donc accès à tout l'univers.

Rule 30

Pour Rule 30, on a besoin de la cellule couramment observé, 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 observé, 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 \rightarrow Universe (Universe a)
- On map notre règle sur chaque case :
fmap rule : : Universe (Universe a) \rightarrow Universe a

fmap rule . duplicate : : Universe a \rightarrow Universe a

Live démo

The image shows a Haskell IDE with two panes. The left pane is a terminal window showing the execution of Conway's Game of Life simulations. The right pane is a code editor showing the Haskell code for these simulations.

Terminal Window (Left Pane):

```
*Automata> showLife 50 10 rule_lr duo_universe
```

The output shows a small, symmetrical pattern of black and white pixels, representing the 'duo_universe'.

```
*Automata> showLife 90 40 rule_30 single_universe
```

The output shows a large, complex, and highly detailed pattern of black and white pixels, representing the 'single_universe'.

Code Editor (Right Pane):

The code defines several functions for the Game of Life simulation:

- `extract`: Extracts the value at a given position in the universe.
- `fnmap`: Applies a function to each element of the universe.
- `duplicate`: Duplicates the universe to create a new one.
- `rule_lr`: The rule for the 'duo_universe' simulation, which is a simple rule where a cell is alive if it is alive in the previous state.
- `rule_30`: The rule for the 'single_universe' simulation, which is a more complex rule with a list of 20 predefined values.



Merci pour votre attention !