Un peu de théorie des catégories en programmation : les monades (+ de la sémantique ?)

Lê Thành Dũng (Tito) Nguyễn — nltd@nguyentito.eu
Postdoc au LIP (ÉNS Lyon) dans l'équipe Plume (logique, automates...)

Séminaire au ski de la licence informatique de l'ÉNS Lyon, dimanche 15 janvier 2023

- On m'a demandé de parler de *théorie des catégories*…
  mais c'est un sujet qui peut être un peu (voire très) abstrait
- Cet exposé : un usage élémentaire d'un concept venu des catégories
  pour de la « vraie » programmation
- Si le temps permet : 2 ou 3 mots sur ma recherche à la fin

- On m'a demandé de parler de *théorie des catégories*…
  mais c'est un sujet qui peut être un peu (voire très) abstrait
- Cet exposé : un usage élémentaire d'un concept venu des catégories
  pour de la « vraie » programmation
- Si le temps permet : 2 ou 3 mots sur ma recherche à la fin

Commençons par de la pure propagande !

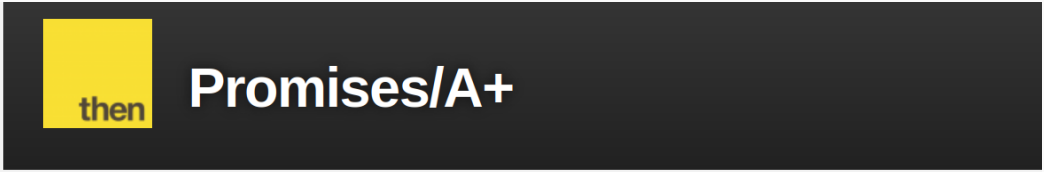
La programmation de la vraie vie : les promesses en JavaScript

Ébauche officielle du W3C (organisme de standards du Web) de 2020

`https://www.w3.org/2001/tag/doc/promises-guide`

> Promises have been battle-tested in many JavaScript libraries, including as part of popular frameworks like Dojo, jQuery, YUI, Ember, Angular, WinJS, Q, and others. This culminated in the Promises/A+ community specification which most libraries conformed to. Now, a standard `Promise` class is included in the JavaScript specification, allowing web platform APIs to return promises for their asynchronous operations. [ECMASCRIPT]

Standard communautaire `https://promisesaplus.com/`



An open standard for sound, interoperable JavaScript promises—by implementers, for implementers.

A *promise* represents the eventual result of an asynchronous operation. The primary way of interacting with a promise is through its `then` method, which registers callbacks to receive either a promise's eventual value or the reason why the promise cannot be fulfilled.

This specification details the behavior of the `then` method, providing an interoperable base which all Promises/A+ conformant promise implementations can be depended on to provide. As such, the specification

Où l'on se dit que les catégories c'est peut-être mieux pour les promesses JS

https://brianmckenna.org/blog/category_theory_promisesaplus

Category Theory for Promises/A+

Brian McKenna — 2013-04-09

Promises are being debated in the JavaScript community. The most popular specification is Promises/A+. It's a fairly small specification, containing only a single function: `then`.

The function is heavily overloaded which makes it quite complicated - way more than it has to be. I'll try to show how category theory can give us a much simpler, more generalised and lawful API!

Ticket GitHub « Incorporate monads and category theory » sur Promises/A+

`https://github.com/promises-aplus/promises-spec/issues/94`

paulmillr commented on Apr 10, 2013

Brian Mckenna criticised current spec. He proposes to use FP approach to achieve much better modularity.

Suggest to read it, really good ideas with just three changes.

http://brianmckenna.org/blog/category_theory_promisesaplus

His proposal is to incorporate into spec three simple apis:

1. `Promise.of(a)` will turn anything into promise.
2. `Promise#then(f)` should take one function, not two.
3. `Promise#onRejected(f)`: move `onRejected` to prototype instead of second arg.

edit: see promises-aplus/constructor-spec#24 for more discussion

👍 111 👎 2 🎉 20 😕 2 ❤️ 42 🚀 12 👀 3

Le ticket est fermé avec réponse cinglante

https://github.com/promises-aplus/promises-spec/issues/94#issuecomment-16176966

domenic commented on Apr 10, 2013 — Member

Yeah this is really not happening. It totally ignores reality in favor of typed-language fantasy land, making a more awkward and less useful API just to satisfy some peoples' aesthetic preferences that aren't even applicable to JavaScript. It misses the point of promises (modeling synchronous control flow from imperative languages), albeit in a novel way from the usual misunderstandings.

It is also hilariously inaccurate, as the thenable described comes nowhere near satisfying the spec. My guess is that it would pass approximately one of the ~500 tests in our test suite.

Someone more diplomatic than me should probably chime in too.

23 473 16 4 37 3 3

Quelques années passent et les gens se plaignent

Billet de blog de 2017 `https://avaq.medium.com/broken-promises-2ae92780f33`

Broken Promises

The unspoken flaws of JavaScript Promises

A few years ago, JavaScript had a callback problem. The community worked hard to replace callbacks with Promises. Now, JavaScript has a Promise problem.

Promises were a big step forward for JavaScript, but unfortunately they have several design flaws. It's important that we acknowledge them, learn from them, and finally reconsider whether we should be embracing Promises the way we do.

Retour sur le ticket « Incorporate monads and category theory »

https://github.com/promises-aplus/promises-spec/issues/94#issuecomment-553767177

kennknowles commented on Nov 14, 2019

I love every additional comment here so many years later. I never muted this thread. The best part is that I have a lifetime of it to look forward to, for two reasons:

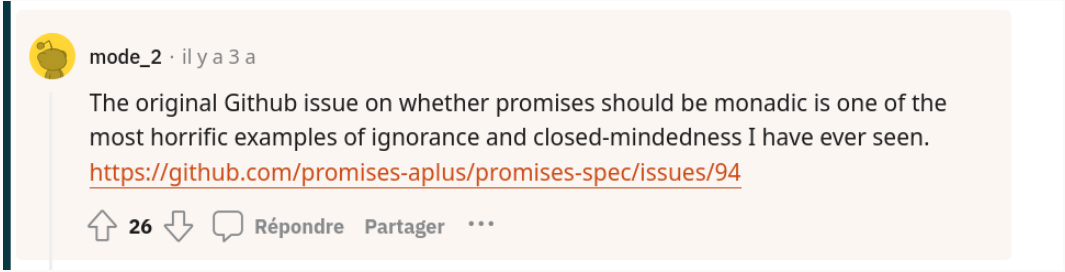
1. The things done wrong here are fundamental to programming/computation itself, so there will never be a time when it is not wrong. And probably never be a time when it doesn't adversely effect programming in Javascript/Typescript.
2. Those who think the conclusions here are OK will never search it up. This thread is kept alive by those who encountered the wrongness and searched the web for what went wrong.

So for those arriving now and forever: Welcome. You are in good company.

40 1 12 1 23 12

Voici ce qu'en pense Reddit

https://www.reddit.com/r/programming/comments/ed7flb/comment/fbgjves/



Revenons sur le billet de blog « Broken Promises »

`https://avaq.medium.com/broken-promises-2ae92780f33`

> The problem is that treating specific values specially breaks a property that type theorists call parametric polymorphism. Not supporting this property means that the data-type in question (Promises) can not be used as one of the generic abstractions that we'll see more and more of as programmers catch up with mathematicians. I'm talking about Functors, Monads and other such alien concepts. If I just lost you there, not to worry! There will be no more mention of them. If I tickled your interest, I urge you to have a look at the Fantasy Land project.

Fantasy Land https://github.com/fantasyland/fantasy-land

Fantasy Land Specification

build passing gitter join chat

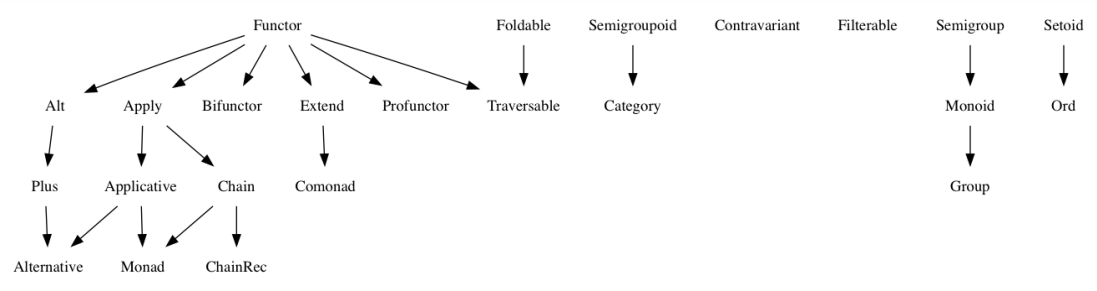
(aka "Algebraic JavaScript Specification")

This project specifies interoperability of common algebraic structures:

- Setoid
- Ord
- Semigroupoid
- Category
- Semigroup
- Monoid

Fantasy Land : structures algébriques et catégoriques en JavaScript (bis)

`https://github.com/fantasyland/fantasy-land`



- groupes : maths de L2
- semigroupes, monoïdes : cf. cours de théorie des automates
- foncteurs, monades : notions issues de la théorie des catégories

Faisons du Haskell

Le premier langage de programmation utilisant les monades est *Haskell*

$\simeq$ un peu comme OCaml (fonctionnel, typage statique) en plus bizarre/rigolo

Les monades répondent à une problématique basique pour Haskell :

comment afficher « Hello, world ! » ? — plus généralement *comment faire des effets ?*

On va voir pourquoi c'est différent des langages de programmation ordinaires

OCaml est un peu paresseux (comme à peu près tous les langages)

En OCaml :

- `true && (print_endline "Hello world!"; false)`
  renvoie `false` et affiche « Hello, world ! »

OCaml est un peu paresseux (comme à peu près tous les langages)

En OCaml :

- `true && (print_endline "Hello world!"; false)`
  renvoie `false` et affiche « Hello, world ! »
- `false && (print_endline "Hello world!"; false)`
  renvoie `false` et *n'affiche rien*

car `false &&` `<bloup>` vaut toujours `false` $\rightsquigarrow$ `<bloup>` n'est pas évalué

OCaml est un peu paresseux (comme à peu près tous les langages)

En OCaml :

- `true && (print_endline "Hello world!"; false)`
  renvoie `false` et affiche « Hello, world ! »
- `false && (print_endline "Hello world!"; false)`
  renvoie `false` et *n'affiche rien*

car `false && <bloup>` vaut toujours `false` ⇝ `<bloup>` n'est pas évalué

Point important

C'est *l'évaluation des sous-expressions* qui provoque des *effets de bord*,
comme par exemple afficher « Hello world ! ».

Haskell est paresseux par défaut

On peut avoir une *liste infinie*, évaluée paresseusement :

```
zipPlus :: [Integer] -> [Integer] -> [Integer]
zipPlus (x:xs) (y:ys) = (x+y) : zipPlus xs ys
-- pas besoin de cas pour les listes vides

fibonacci :: [Integer]
fibonacci = 1 : 1 : zipPlus fibonacci (tail fibonacci)
```

Si on évalue `take 5 fibonacci` on obtient `[1,1,2,3,5]`. Le calcul n'essaie pas d'évaluer l'infinité des éléments de la liste avant de renvoyer un résultat !

Défis de l'évaluation paresseuse

Point important (en OCaml et dans la plupart des autres langages)

C'est *l'évaluation des sous-expressions* qui provoque des *effets de bord*, comme par exemple afficher « Hello world ! ».

En Haskell :

- L'ordre dans lequel les sous-expressions sont évaluées n'est pas clair
- Certaines peuvent ne jamais être évaluées, si leur valeur ne sert à rien
- Donc si l'évaluation produit des effets, ça sera imprévisible

Défis de l'évaluation paresseuse

Point important (en OCaml et dans la plupart des autres langages)

C'est *l'évaluation des sous-expressions* qui provoque des *effets de bord*,
comme par exemple afficher « Hello world ! ».

En Haskell :

- L'ordre dans lequel les sous-expressions sont évaluées n'est pas clair
- Certaines peuvent ne jamais être évaluées, si leur valeur ne sert à rien
- Donc si l'évaluation produit des effets, ça sera imprévisible

Si OCaml était paresseux par défaut, ceci n'afficherait pas « Hello » :

```
print_string "Hello"; print_endline ", world!"
```

Effets en Haskell

Solution : séparer *évaluation d'une expression* et *exécution d'une action*

```
main :: IO () -- les actions sont typées par IO truc
main = do putStrLn "Comment t'appelles-tu ?"
          nom <- getLine -- demande une entrée utilisateur
          putStrLn ("Bonjour, " ++ nom ++ " !")
```

`getLine :: IO String` tandis que `"Bonjour, " ++ nom ++ " !" :: String`

Effets en Haskell

Solution : séparer *évaluation d'une expression* et *exécution d'une action*

```
main :: IO () -- les actions sont typées par IO truc
main = do putStrLn "Comment t'appelles-tu ?"
          nom <- getLine -- demande une entrée utilisateur
          putStrLn ("Bonjour, " ++ nom ++ " !")
```

`getLine :: IO String` tandis que `"Bonjour, " ++ nom ++ " !" :: String`

- Une expression de type `String` s'évalue en une chaîne de caractères, et *l'évaluation ne produit aucun effet* (on dit que Haskell est *purement fonctionnel*)
- Une expression de type `IO String` s'évalue en une *action* dont *l'exécution* produira un effet et renverra une chaîne de caractères

Effets en Haskell

Solution : séparer *évaluation d'une expression* et *exécution d'une action*

```
main :: IO () -- les actions sont typées par IO truc
main = do putStrLn "Comment t'appelles-tu ?"
          nom <- getLine -- demande une entrée utilisateur
          putStrLn ("Bonjour, " ++ nom ++ " !")
```

`getLine :: IO String` tandis que `"Bonjour, " ++ nom ++ " !" :: String`

- Une expression de type `String` s'évalue en une chaîne de caractères, et *l'évaluation ne produit aucun effet* (on dit que Haskell est *purement fonctionnel*)
- Une expression de type `IO String` s'évalue en une *action* dont *l'exécution* produira un effet et renverra une chaîne de caractères

et `putStrLn :: String -> IO ()`

Désucrage de la notation `do`

```
main = do putStrLn "Comment t'appelles-tu ?"
          nom <- getLine
          putStrLn ("Bonjour, " ++ nom ++ " !")
```

est en fait du *sucre syntaxique* (`do` est un mot-clé) qui signifie

```
main = putStrLn "Comment t'appelles-tu ?" >>= (\_ ->
         getLine >>= (\nom ->
           putStrLn ("Bonjour, " ++ nom ++ " !")))
```

(où `\x -> bloup` en Haskell $\simeq$ `fun x -> bloup` en OCaml)

Désucrage de la notation `do`

```
main = do putStrLn "Comment t'appelles-tu ?"
          nom <- getLine
          putStrLn ("Bonjour, " ++ nom ++ " !")
```

est en fait du *sucre syntaxique* (`do` est un mot-clé) qui signifie

```
main = putStrLn "Comment t'appelles-tu ?" >>= (\_ ->
         getLine >>= (\nom ->
           putStrLn ("Bonjour, " ++ nom ++ " !")))
```

(où `\x -> bloup` en Haskell $\simeq$ `fun x -> bloup` en OCaml)

En gros, `a >>= (\x -> b)` correspond à `let x = a in b` en OCaml… sauf que `(>>=) :: IO a -> (a -> IO b) -> IO b` est un « vrai » opérateur sur les actions !

Surchargeons l'opérateur >>=

En Haskell, les actions sont des « valeurs de 1ère classe », avec les opérations :

- `(>>=) :: IO a -> (a -> IO b) -> IO b` (équivalent du `let … = … in …` d'OCaml qui n'est pas un « vrai » opérateur car OCaml n'a pas de notion d'action $\neq$ expression)
- `return :: a -> IO a` (valeurs pures $\rightsquigarrow$ actions sans effet)

Surchargeons l'opérateur >>=

En Haskell, les actions sont des « valeurs de 1ère classe », avec les opérations :

- `(>>=) :: IO a -> (a -> IO b) -> IO b` (équivalent du `let … = … in …` d'OCaml qui n'est pas un « vrai » opérateur car OCaml n'a pas de notion d'action ≠ expression)
- `return :: a -> IO a` (valeurs pures ⇝ actions sans effet)

Si on a un autre type paramétré `m a` avec

```
(>>=) :: m a -> (a -> m b) -> m b     return :: a -> m a
```

alors on peut utiliser la notation `do` ! Comme si on *surchargeait* `let … = … in …`

(Analogie : en OCaml, on écrit `2 + 3` mais `2.71 +. 3.14;`
d'autres langages autorisent l'utilisation du même + pour les entiers et les flottants, alors que ce n'est pas la même opération du processeur)

Exemple de surcharge de >>=

Au lieu de `IO a`, considérons `[a]` ($\simeq$ `'a list` en OCaml)

```
(>>=) :: [a] -> (a -> [b]) -> [b]
xs >>= f = ... -- tel que [1,2,3] >>= f  ==  f 1 ++ f 2 ++ f 3
return :: a -> [a]
return x = [x]
```

Exemple de surcharge de >>=

Au lieu de `IO a`, considérons `[a]` ($\simeq$ `'a list` en OCaml)

```
(>>=) :: [a] -> (a -> [b]) -> [b]
xs >>= f = ... -- tel que [1,2,3] >>= f  ==  f 1 ++ f 2 ++ f 3
return :: a -> [a]
return x = [x]
```

Le code suivant renvoie alors `[-1,1,-2,2,-3,3]` :

```
ex_liste = do x <- [1,2,3]
              y <- [-1,1]
              return (x*y)
```

```
ex_liste = [1,2,3] >>= (\x ->
             [-1,1] >>= (\y ->
               return (x*y)))
```

On dit que « la monade liste représente l'effet du non-déterminisme »

C'est quoi une monade ?

Définition (à la manière d'une structure algébrique)

Une *monade* (en programmation), c'est :

- un constructeur de type `m` (par exemple `IO` ou `[…]` (liste))
- 2 opérations `(>>=) :: m a -> (a -> m b) -> m b`, `return :: a -> m a`
- quelques axiomes à vérifier

C'est quoi une monade ?

Définition (à la manière d'une structure algébrique)

Une *monade* (en programmation), c'est :

- un constructeur de type `m` (par exemple `IO` ou `[…]` (liste))
- 2 opérations `(>>=) :: m a -> (a -> m b) -> m b`, `return :: a -> m a`
- quelques axiomes à vérifier

Par exemple on s'attend à ce que cette équation soit une conséquence des axiomes :

```
(t >> u) >> v = t >> (u >> v)
```

où `t >> u = t >>= (\_ -> u)`

(en OCaml : `(t;u);v = t;(u;v)`).

Mais d'où viennent les axiomes ?

C'est quoi une monade ?

Définition (à la manière d'une structure algébrique)

Une *monade* (en programmation), c'est :

- un constructeur de type `m` (par exemple `IO` ou `[…]` (liste))
- 2 opérations `(>>=) :: m a -> (a -> m b) -> m b`, `return :: a -> m a`
- quelques axiomes à vérifier

Par exemple on s'attend à ce que cette équation soit une conséquence des axiomes :

`(t >> u) >> v = t >> (u >> v)` où `t >> u = t >>= (\_ -> u)`

(en OCaml : `(t;u);v = t;(u;v)`). Mais d'où viennent les axiomes ?

$\rightsquigarrow$ on va voir qu'ils ont justifiés par des *principes mathématiques*

C'est quoi une monade ? (bis)

Définition équivalente, plus pratique pour énoncer les axiomes

Une *monade*, c'est un constructeur de type `m` avec :

- `fmap :: (a -> b) -> m a -> m b`, `join :: m (m a) -> m a`, `return :: …`
- quelques axiomes à vérifier

Par exemple sur les listes : `fmap (*2) [1,2,3] = [2,4,6]`

`join [[1,2],[3],[4,5,6]] = [1,2,3,4,5,6]`

C'est quoi une monade ? (bis)

Définition équivalente, plus pratique pour énoncer les axiomes

Une *monade*, c'est un constructeur de type `m` avec :

- `fmap :: (a -> b) -> m a -> m b`, `join :: m (m a) -> m a`, `return :: …`
- quelques axiomes à vérifier

Par exemple sur les listes : `fmap (*2) [1,2,3] = [2,4,6]`

`join [[1,2],[3],[4,5,6]] = [1,2,3,4,5,6]`

```
fmap f x = do y <- x
              return (f y)
```

```
join x = x >>= (\y -> y)
-- on peut aussi exprimer fmap avec >>=
```

C'est quoi une monade ? (bis)

Définition équivalente, plus pratique pour énoncer les axiomes

Une *monade*, c'est un constructeur de type `m` avec :

- `fmap :: (a -> b) -> m a -> m b`, `join :: m (m a) -> m a`, `return :: …`
- quelques axiomes à vérifier

Par exemple sur les listes : `fmap (*2) [1,2,3] = [2,4,6]`

`join [[1,2],[3],[4,5,6]] = [1,2,3,4,5,6]`

```
fmap f x = do y <- x
              return (f y)
```

```
join x = x >>= (\y -> y)
-- on peut aussi exprimer fmap avec >>=
```

réciproquement `x >>= f = join (fmap f x)`

Les foncteurs (rien à voir avec les modules paramétrés d'OCaml)

On a des axiomes évidents pour `fmap :: (a -> b) -> m a -> m b` :

$$\texttt{fmap}\ (\underbrace{\texttt{f . g}}_{\text{composition de f et g}})\ \texttt{x} = \texttt{fmap f}\ (\texttt{fmap g x}) \qquad \texttt{fmap}\ \underbrace{\texttt{id}}_{\text{fonction identité}}\ \texttt{x} = \texttt{x}$$

Les foncteurs (rien à voir avec les modules paramétrés d'OCaml)

On a des axiomes évidents pour `fmap :: (a -> b) -> m a -> m b` :

$$\texttt{fmap}\ \underbrace{(\texttt{f . g})}_{\text{composition de f et g}}\ \texttt{x} = \texttt{fmap f}\ (\texttt{fmap g x}) \qquad \texttt{fmap}\ \underbrace{\texttt{id}}_{\text{fonction identité}}\ \texttt{x} = \texttt{x}$$

Définition

Un constructeur de type `m` avec une telle opération `fmap` est un *foncteur*.

- Le constructeur des types listes est un foncteur, `IO` est un foncteur...

Les foncteurs (rien à voir avec les modules paramétrés d'OCaml)

On a des axiomes évidents pour `fmap :: (a -> b) -> m a -> m b` :

$$\texttt{fmap}\ \underbrace{(\texttt{f . g})}_{\text{composition de f et g}}\ \texttt{x} = \texttt{fmap f}\ (\texttt{fmap g x}) \qquad \texttt{fmap}\ \underbrace{\texttt{id}}_{\text{fonction identité}}\ \texttt{x} = \texttt{x}$$

Définition

Un constructeur de type `m` avec une telle opération `fmap` est un *foncteur*.

- Le constructeur des types listes est un foncteur, `IO` est un foncteur...
- Le constructeur des dictionnaires à clés `k` et valeurs `v` est un foncteur en `v` (`fmap` applique la fonction à chaque valeur)

  plus généralement, plein de *structures de données* sont des foncteurs !

Les monades rebelote

Reformulation de la définition précédente

Une *monade*, c'est un foncteur (`m`, `fmap`) avec :

- `join :: m (m a) -> m a`, `return :: a -> m a`
- quelques axiomes sur `join` et `return` à vérifier

« Cachons » le paramètre de type `a` en écrivant

$$\texttt{join} :: \texttt{m} \circ \texttt{m} \Rightarrow \texttt{m} \qquad \texttt{return} :: \texttt{Id} \Rightarrow \texttt{m}$$

Reformulation de la définition précédente

Une *monade*, c'est un foncteur (`m`, `fmap`) avec :

- `join :: m (m a) -> m a`, `return :: a -> m a`
- quelques axiomes sur `join` et `return` à vérifier

« Cachons » le paramètre de type `a` en écrivant

$$\texttt{join} :: \texttt{m} \circ \texttt{m} \Rightarrow \texttt{m} \qquad \texttt{return} :: \texttt{Id} \Rightarrow \texttt{m}$$

Ça ressemble à un structure algébrique composée d'un ensemble M muni de :

$$\text{loi de composition : } M \times M \to M \qquad \text{élément choisi : } \underbrace{\{*\} \to M}_{\text{pareil que de se donner un élément fixé dans } M}$$

(remarque : `Id` est neutre pour $\circ$ tout comme $\{*\}$ est neutre pour $\times$)

Creusons cette analogie

$$\texttt{join} :: \texttt{m} \circ \texttt{m} \Rightarrow \texttt{m} \quad \& \quad \texttt{return} :: \texttt{Id} \Rightarrow \texttt{m} \quad \text{vs.} \quad \mu : M \times M \to M \quad \& \quad e \in M$$

On veut une *associativité* `(t >> u) >> v = t >> (u >> v)` comme conséquence des axiomes de monades. Ça fait penser aux *monoïdes* : $\mu(\mu(x, y), z) = \mu(x, \mu(y, z))$

Creusons cette analogie

$$\texttt{join} :: \texttt{m} \circ \texttt{m} \Rightarrow \texttt{m} \quad \& \quad \texttt{return} :: \texttt{Id} \Rightarrow \texttt{m} \quad \text{vs.} \quad \mu : M \times M \to M \quad \& \quad e \in M$$

On veut une *associativité* `(t >> u) >> v = t >> (u >> v)` comme conséquence des axiomes de monades. Ça fait penser aux *monoïdes* : $\mu(\mu(x, y), z) = \mu(x, \mu(y, z))$

$$(M \times M) \times M \xrightarrow{\mu \times \mathrm{id}} M \times M \xrightarrow{\mu} M \quad \cong \quad M \times (M \times M) \xrightarrow{\mathrm{id} \times \mu} M \times M \xrightarrow{\mu} M$$

Creusons cette analogie

$$\texttt{join} :: \texttt{m} \circ \texttt{m} \Rightarrow \texttt{m} \quad \& \quad \texttt{return} :: \texttt{Id} \Rightarrow \texttt{m} \quad \text{vs.} \quad \mu : M \times M \to M \quad \& \quad e \in M$$

On veut une *associativité* `(t >> u) >> v = t >> (u >> v)` comme conséquence des axiomes de monades. Ça fait penser aux *monoïdes* : $\mu(\mu(x,y),z) = \mu(x,\mu(y,z))$

$$(M \times M) \times M \xrightarrow{\mu \times \mathrm{id}} M \times M \xrightarrow{\mu} M \quad \cong \quad M \times (M \times M) \xrightarrow{\mathrm{id} \times \mu} M \times M \xrightarrow{\mu} M$$

Transposons ça pour obtenir les axiomes des monades (enfin !) :

$$(\texttt{m} \circ \texttt{m}) \circ \texttt{m} \xrightarrow{\texttt{join}} \texttt{m} \circ \texttt{m} \xrightarrow{\texttt{join}} \texttt{m} \quad = \quad \texttt{m} \circ (\texttt{m} \circ \texttt{m}) \xrightarrow{\texttt{fmap join}} \texttt{m} \circ \texttt{m} \xrightarrow{\texttt{join}} \texttt{m}$$

+ des équations pour `return` correspondant à $\mu(x,e) = \mu(e,x) = x$

Les catégories, ça sert à généraliser

On a une notion générale de *monoïde interne à une catégorie monoïdale* $\mathcal{C}$.

- Pour $\mathcal{C} = (\text{Ensembles}, \times, \{*\})$ on obtient les monoïdes « habituels »
- Pour $\mathcal{C} = (\text{Foncteurs}, \circ, \texttt{Id})$ on obtient les *monades* –
  une structure algébrique qui s'est révélée utile en programmation !

Les catégories, ça sert à généraliser

On a une notion générale de *monoïde interne à une catégorie monoïdale* $\mathcal{C}$.

- Pour $\mathcal{C} = (\text{Ensembles}, \times, \{*\})$ on obtient les monoïdes « habituels »
- Pour $\mathcal{C} = (\text{Foncteurs}, \circ, \texttt{Id})$ on obtient les *monades* –
  une structure algébrique qui s'est révélée utile en programmation !

James Iry, *A Brief, Incomplete, and Mostly Wrong History of Programming Languages* :
« 1990 – A committee [...] creates Haskell, a pure, non-strict, functional language. Haskell gets some resistance due to the complexity of using monads to control side effects.

Les catégories, ça sert à généraliser

On a une notion générale de *monoïde interne à une catégorie monoïdale* $\mathcal{C}$.

- Pour $\mathcal{C} = (\text{Ensembles}, \times, \{*\})$ on obtient les monoïdes « habituels »
- Pour $\mathcal{C} = (\text{Foncteurs}, \circ, \texttt{Id})$ on obtient les *monades* –
  une structure algébrique qui s'est révélée utile en programmation !

James Iry, *A Brief, Incomplete, and Mostly Wrong History of Programming Languages* :
« 1990 – A committee [...] creates Haskell, a pure, non-strict, functional language. Haskell gets some resistance due to the complexity of using monads to control side effects. Wadler tries to appease critics by explaining that *"a monad is a monoid in the category of endofunctors, what's the problem ?"* »

Voir aussi `https://stackoverflow.com/questions/3870088/a-monad-is-just-a-monoid-in-the-category-of-endofunctors-whats-the-problem`

Mais c'est quoi une catégorie (monoïdale) ?

Définition

Une *catégorie*, c'est une classe d'*objets* avec :

- Pour toute paire d'objets (A, B), une classe de *morphismes* $A \to B$
- Une *composition* $f : A \to B, g : B \to C \rightsquigarrow g \circ f : A \to C$
- Des morphismes *identité* id_A, neutres pour $\circ$

- Les ensembles et applications entre eux
- Les groupes et morphismes de groupes
- Les types Haskell et programmes de type `A -> B`
- ...

Et une *catégorie monoïdale* ça vient avec de la structure supplémentaire

(une loi de composition sur les objets...)

Pourquoi endofoncteur ? (Haskell vous ment)

Définition (selon les mathématicien·nes)

Un *foncteur* F d'une catégorie dans une autre envoie des objets sur des objets, les morphismes $A \to B$ sur des morphismes $F(A) \to F(B)$, et préserve $\circ$ et id.

Par exemple le foncteur des groupes dans les ensembles qui « oublie » la structure

Pourquoi endofoncteur ? (Haskell vous ment)

Définition (selon les mathématicien·nes)

Un *foncteur* F d'une catégorie dans une autre envoie des objets sur des objets, les morphismes $A \to B$ sur des morphismes $F(A) \to F(B)$, et préserve $\circ$ et id.

Par exemple le foncteur des groupes dans les ensembles qui « oublie » la structure

Catégories, foncteurs et monades ont été inventés dans les années 1950 pour faire de la topologie algébrique ; c'est aussi utilisé en géométrie, théorie des nombres…

Pourquoi endofoncteur ? (Haskell vous ment)

Définition (selon les mathématicien·nes)

Un *foncteur* F d'une catégorie dans une autre envoie des objets sur des objets, les morphismes $A \to B$ sur des morphismes $F(A) \to F(B)$, et préserve $\circ$ et id.

Par exemple le foncteur des groupes dans les ensembles qui « oublie » la structure

Catégories, foncteurs et monades ont été inventés dans les années 1950 pour faire de la topologie algébrique ; c'est aussi utilisé en géométrie, théorie des nombres…

Définition

Un *endofoncteur* de $\mathcal{C}$ = un foncteur de $\mathcal{C}$ dans $\mathcal{C}$

(et les monades des mathématicien·nes sont des endofoncteurs)

$\rightsquigarrow$ les foncteurs en programmation

= endofoncteurs de la catégorie des types et programmes

Les apports de la sémantique dénotationnelle

Application initiale des monades en informatique : donner une *sémantique* mathématique à des langages qui ont des *effets de bord* (comme OCaml).

- Naïvement : une fonction `int -> int` représente une application $\mathbb{N} \to \mathbb{N}$

Les apports de la sémantique dénotationnelle

Application initiale des monades en informatique : donner une *sémantique* mathématique à des langages qui ont des *effets de bord* (comme OCaml).

- Naïvement : une fonction `int -> int` représente une application $\mathbb{N} \to \mathbb{N}$
- C'est plus compliqué dès qu'on a de la non-terminaison, de la récursivité...
  $\rightsquigarrow$ nécessite de travailler dans d'autres *catégories* que celle des ensembles
- Et quand on rajoute des effets de bord, il faut en plus considérer des monades sur de telles catégories ! C'est ça qui a inspiré l'usage des monades en Haskell

Les apports de la sémantique dénotationnelle

Application initiale des monades en informatique : donner une *sémantique* mathématique à des langages qui ont des *effets de bord* (comme OCaml).

- Naïvement : une fonction `int -> int` représente une application $\mathbb{N} \to \mathbb{N}$
- C'est plus compliqué dès qu'on a de la non-terminaison, de la récursivité...
  $\rightsquigarrow$ nécessite de travailler dans d'autres *catégories* que celle des ensembles
- Et quand on rajoute des effets de bord, il faut en plus considérer des monades sur de telles catégories ! C'est ça qui a inspiré l'usage des monades en Haskell

Quelques autres innovations syntaxiques inspirées par la sémantique :

- logique linéaire (Jean-Yves Girard) : spécialité de l'équipe Plume au LIP
- théorie homotopique des types (Voïevodski) provenant de sémantiques *topologiques* : le futur de Coq ? (liée aux *catégories de dimension supérieure*)

Ma recherche en deux mots

La sémantique dénotationnelle et catégorique sert aussi à *prouver* des choses sur des langages de programmation (théoriques), par exemple :

Théorème (Hillebrand & Kanellakis 1996)

Les fonctions `string` $\to$ `bool` que peuvent définir des λ-*termes simplement typés* (suivant une convention naturelle) correspondent précisément aux *langages rationnels*.

(Idée : interpréter les λ-termes dans la catégorie des ensembles *finis*.)

Ma recherche en deux mots

La sémantique dénotationnelle et catégorique sert aussi à *prouver* des choses sur des langages de programmation (théoriques), par exemple :

Théorème (Hillebrand & Kanellakis 1996)

Les fonctions `string` $\to$ `bool` que peuvent définir des λ-*termes simplement typés* (suivant une convention naturelle) correspondent précisément aux *langages rationnels*.

(Idée : interpréter les λ-termes dans la catégorie des ensembles *finis*.)

- Mes travaux cherchent à prolonger ces liens entre λ-calcul et automates
- Les catégories et foncteurs sont un langage commun permettant de faire des ponts entre ces domaines différents

Ma recherche en deux mots

La sémantique dénotationnelle et catégorique sert aussi à *prouver* des choses sur des langages de programmation (théoriques), par exemple :

Théorème (Hillebrand & Kanellakis 1996)

Les fonctions `string` $\to$ `bool` que peuvent définir des λ-*termes simplement typés* (suivant une convention naturelle) correspondent précisément aux *langages rationnels*.

(Idée : interpréter les λ-termes dans la catégorie des ensembles *finis*.)

- Mes travaux cherchent à prolonger ces liens entre λ-calcul et automates
- Les catégories et foncteurs sont un langage commun permettant de faire des ponts entre ces domaines différents

Merci pour votre attention !