Programmer avec des produits, des sommes, des exponentiels ~~et des constructeurs non-surjectifs~~

Xavier Van de Woestyne
xaviervdw@gmail.com, @vdwxv
marigold.dev

Nuit des Meetups (Nantes), Avril 2022

Développeur **OCaml** chez Marigold
"*Constantly aiming to make impactful contributions to **Tezos** blockchain.*"

Je travaille spécifiquement sur le protocol et certaines solutions de *Layer-2*.

Tenter de proposer un événement récurrent dédié à la **programmation fonctionnelle** et à l'utilisation des **langages applicatifs**

Restons en contact si ça vous intéresse (En tant que présentateur ou pour participer au projet)

Programme et motivations

FIG. : https ://www.npmtrends.com/typescript

La popularité de **TypeScript** est une belle preuve qu'il faut s'intéresser au typage statique.

Beaucoup de *nouveaux* langages offrent des *liftings* à des langages un peu tristes (**Kotlin, TypeScript, Swift**). Ces ajouts convergent généralement (de près ou de loin) vers les langages **ML**.

- Présenter les **types algébriques** (`ADTs`) avec le langage **OCaml**
  (Pourquoi "*algébrique*")
- Comprendre leur **utilité**
  (`DDD`, structure de données, uniformisation)
- Une considération les **abstractions échappées**
  (Introduction des *exponentiels*)
- ~~**Types algébriques généralisés** (`GADTs`)~~
  ~~(Pour des constructeurs non-surjectifs)~~

Types algébriques

Comme souvent dans le monde des langages de programmation fonctionnelle, **la terminologie, par conviction idéologique, peut être intimidante.**

Le terme **Type Algébrique** rejoint *curryfication*, *monoïde*, *monade* et beaucoup d'autres termes intimidants **derrière lesquels se cache, en fait, une description relativement simple** (et pertinente).

Types algébriques

Comme souvent dans le monde des langages de programmation fonctionnelle, **la terminologie, par conviction idéologique, peut être intimidante.**

Le terme **Type Algébrique** rejoint *curryfication*, *monoïde*, *monade* et beaucoup d'autres termes intimidants **derrière lesquels se cache, en fait, une description relativement simple** (et pertinente).

Un petit rappel du collège, l'enseignement des mathématiques est découpé en deux branches spécifiques (pouvant inter-agir entre elles)

Géométrie

Algèbre

Types algébriques

Comme souvent dans le monde des langages de programmation fonctionnelle, **la terminologie, par conviction idéologique, peut être intimidante.**

Le terme **Type Algébrique** rejoint *curryfication*, *monoïde*, *monade* et beaucoup d'autres termes intimidants **derrière lesquels se cache, en fait, une description relativement simple** (et pertinente).

Un petit rappel du collège, l'enseignement des mathématiques est découpé en deux branches spécifiques (pouvant inter-agir entre elles)

Géométrie

Algèbre

Heureusement que les types algébriques ne s'appellent pas des types géométriques. (Et pourtant...)

Types algébriques

Comme souvent dans le monde des langages de programmation fonctionnelle, **la terminologie, par conviction idéologique, peut être intimidante.**

Le terme **Type Algébrique** rejoint *curryfication*, *monoïde*, *monade* et beaucoup d'autres termes intimidants **derrière lesquels se cache, en fait, une description relativement simple** (et pertinente).

Un petit rappel du collège, l'enseignement des mathématiques est découpé en deux branches spécifiques (pouvant inter-agir entre elles)

Géométrie

Heureusement que les types algébriques ne s'appellent pas des types géométriques. (Et pourtant...)

Algèbre

Quel peut bien être le rapport avec $(a+b)^2 = a^2 + 2ab + b^2$?

Mais **que veut dire *"Algèbre"***

L'algèbre, très informellement

Comme notre système numérique est *basé* sur le système de numération Arabe, il est normal que certains termes tirent leur éthymologie de la langue Arabe (comme "Algorithme", qui est dérivé du nom *al-Khuwārizmī*, un mathématicien Perse et aussi l'auteur du livre à l'origine du terme "algèbre").

L'algèbre, très informellement

Comme notre système numérique est *basé* sur le système de numération Arabe, il est normal que certains termes tirent leur éthymologie de la langue Arabe (comme "Algorithme", qui est dérivé du nom *al-Khuwārizmī*, un mathématicien Perse et aussi l'auteur du livre à l'origine du terme "algèbre").

Algèbre provient de الجبر (*al-djabr*) voulant dire **La réunion des parties cassées**.

L'algèbre, très informellement

Comme notre système numérique est *basé* sur le système de numération Arabe, il est normal que certains termes tirent leur éthymologie de la langue Arabe (comme "Algorithme", qui est dérivé du nom *al-Khuwārizmī*, un mathématicien Perse et aussi l'auteur du livre à l'origine du terme "algèbre").

Algèbre provient de الجبر (*al-djabr*) voulant dire **La réunion des parties cassées**.

$$(12 + 3 * 10) - 3 = 39$$

Les **nombres sont *des parties cassées*** (des fragments) et **les opérateurs permettent leur réunion**. En d'autres mots, en arithmétique, on peut construire des nombres en utilisant des opérateurs.

L'algèbre ~~abstraite~~, toujours informellement, mais un peu moins

On retrouve "***trois*** *ingrédients*".
Par exemple, pour le *semi-anneau* des nombres naturels

Un ensemble de "*valeurs*"

$0, 1, 2, 3, ...n$

Des opérateurs

$+$, $*$

des lois équationnelles

sur le comportement des opérateurs (et par rapport à certaines valeurs)

Et donc, pourquoi "*types algébriques*"

On retrouve nos "***trois*** *ingrédients*".

Un ensemble de "*valeurs*"

$int, float, etc.$

Des opérateurs

$+, *$

des lois équationnelles

Dont nous ne parlerons pas ... (le temps, le temps)

mais retenons que les ADTs forment un semi-anneau, eux aussi

Pour résumer, les types algébriques nous permettent de construire des **nouveaux types** par **composition conjonctive**, avec $*$ (on appelle ça des **produit**) ou par **composition disjonctive**, avec $+$ (appelé des **sommes**).

Petit rappel

$$\Gamma \vdash e_1 : \tau_1$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma_n \vdash e_n : \tau_n}{\Gamma \vdash e : \tau}$$

Les types produits

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash fst\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash snd\ e : \tau_2}$$

```
let make a b = (a, b)

let fst product =
  match product with (x, _) -> x

let snd product =
  match product with (_, y) -> y
```

Les types produits

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash fst\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash snd\ e : \tau_2}$$

```
let make a b = (a, b)

- let fst product =
-   match product with (x, _) -> x
+ let fst (x, _) = x

- let snd product =
-   match product with (_, y) -> y
+ let snd (_, y) = y
```

Les types produits

Même si le type $(a * b)$ suffit largement a représenter l'ensemble des produits :

- $(a * b * c) = (a * (b * c))$
- $(a * b * c * d) = (a * (b * (c * d)))$

Beaucoup de langages qui supportend les ADTs proposent la syntaxe $a * b * c \quad \cdots \quad * n$ (et d'autres manière de décrire des produits) pour un confort d'utilisation.

Plusieurs manière de faire le produit de types en OCaml

En utilisant un n-uplet

```
type point_3d = int * int * int
```

En utilisant un record

```
type point_3d = {
  x : int
; y : int
; z : int
}
```

En utilisant un objet

```
type point_3d = <
  x: int
; y: int
; z: int >
```

Plusieurs manière de faire le produit de types en OCaml

En utilisant un n-uplet

```
type point_3d = int * int * int
```

En utilisant un record

```
type point_3d = {
  x : int
; y : int
; z : int
}
```

En utilisant un objet

```
type point_3d = <
  x: int
; y: int
; z: int >
```

Garder en tête que $a * b$ est la représentation canonique d'un produit est utile

Elle permet de traiter toute sorte de produits de manière **générique**, notamment avec une fonction de ce type :

```
val bimap : ('a -> 'c) -> ('b -> 'd) -> ('a * 'b) -> ('c * 'd)
```

Calculer la cardinalité d'un produit

On peut facilement calculer le nombre **d'habitants possibles potentiels** d'un produit, **il suffit de faire le produit cartésien des cardinalité de chaque membre**

```
type user = {
  username: string
; password: string
; age : int
}
```

$$|user| = |string| * |string| * |int|$$

Calculer la cardinalité d'un produit

On peut facilement calculer le nombre **d'habitants possibles potentiels** d'un produit, **il suffit de faire le produit cartésien des cardinalité de chaque membre**

```
type user = {
  username: string
; password: string
; age : int
}
```

$$|user| = |string| * |string| * |int|$$

Attention

Une méthode qui ne tient pas compte des validations ad-hoc

Résumé des produits

- Construire des types *plus* complexes avec les produits est possible dans énormément de langages (via des `struct` ou des classes *à la OOP*)
- Ils permettent de faire la **conjonction** de plusieurs types
- Leur **cardinalité** est facile à calculer (*modulo* les validations *ad-hoc*)
- La représentation la plus générique d'un produit est **le couple**.

Les types sommes

Après les produits, passons aux types sommes, aux compositions disjonctive

De la même manière que l'on a considéré $a * b$ comme constructeur minimal (pour les produits, mais étant lui aussi un produit), on suppose l'existance d'une somme minimale : `(a, b) either = Left of a | Right of b`

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathit{Left}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathit{Right}\ e : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad x : \tau_1, \Gamma \vdash e_1 : \tau \qquad y : \tau_2, \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\textbf{match}\ e\ \textbf{with}\ \mathit{Left}\ x \to e_1; \mathit{Right}\ y \to e_2) : \tau}$$

```
let left x = Left x
let right y = Right y

let fold sum is_left is_right =
  match sum with
  | Left x -> is_left x
  | Right y -> is_right y
```

Les types sommes

Même si le type $(a, b)either$ suffit largement a représenter l'ensemble des sommes :

- $(a|b|c) = (Left\ a|Right\ (Left\ b|Right\ c))$
- $(a|b|c|d) = (Left\ a|Right\ (Left\ b|Right\ (Left\ c|Right\ d)))$

C'est tout de même très handicapant et dur à lire, donc OCaml (comme Haskell et beaucoupo d'autres) propose une syntaxe dédiée à décrire des sommes.

Quelques exemples de types sommes

Recréation du type booléen

```
type my_boolean = My_true | My_false
```

Uniformisation de plusieurs types hétérogènes

```
type int_or_float_or_string =
  Int of int
| Float of float
| String of string
```

Uniformisation de plusieurs types hétérogènes (avec des produits)

```
type point =
  Point_2d of {x: int; y: int}
| Point_3d of {x: int; y: int; z: int}
```

Quelques exemples de types sommes

Recréation du type booléen

```
type my_boolean = My_true | My_false
```

Uniformisation de plusieurs types hétérogènes

```
type int_or_float_or_string =
  Int of int
| Float of float
| String of string
```

Uniformisation de plusieurs types hétérogènes (avec des produits)

```
type point =
  Point_2d of {x: int; y: int}
| Point_3d of {x: int; y: int; z: int}
```

Attention

My_true ou Int ou encore $Point\_2d$ **ne sont pas** des types ou des sous-types, ce sont des **constructeurs** qui permettent de **construire des valeur du type dans lequel ils sont circonscrit**. Donc `My_true` aura le type `my_boolean` et `String ``foo bar''` aura le type `int_or_float_or_string`.

Une intéressante symétrie

Qui donne beaucoup d'indices sur la différence entre les sommes et les produits

Produits

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash fst\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash snd\ e : \tau_2}$$

Sommes

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash Left\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash Right\ e : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad x : \tau_1, \Gamma \vdash e_1 : \tau \qquad y : \tau_2, \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\textbf{match}\ e\ \textbf{with}\ Left\ x \to e_1; Right\ y \to e_2) : \tau}$$

Une intéressante symétrie

Qui donne beaucoup d'indices sur la différence entre les sommes et les produits

Produits

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash fst\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash snd\ e : \tau_2}$$

Sommes

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash Left\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash Right\ e : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad x : \tau_1, \Gamma \vdash e_1 : \tau \qquad y : \tau_2, \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\textbf{match}\ e\ \textbf{with}\ Left\ x \to e_1; Right\ y \to e_2) : \tau}$$

Comme un produit **fait une conjonction de plusieurs valeurs**, on ne possède qu'une seule fonction de création qui prend tous les arguments nécéssaires et il faut plusieurs fonctions de consommation, alors qu'une somme **fait une disjonction entre plusieurs valeurs** donc il faut plusieurs fonctions pour créer une valeur (du type de la somme), par contre il ne faut qu'une seule fonction de consommation.

Consommer une somme

Écrivons une fonction qui transforme notre type int or float or string en string

```
let to_string value =
  match value with
  | Int i -> int_to_string i
  | Float f -> float_to_string f
  | String s -> s
```

Consommer une somme

Écrivons une fonction qui transforme notre type int or float or string en string

```
let to_string value =
  match value with
  | Int i -> int_to_string i
  | Float f -> float_to_string f
  | String s -> s
```

Ce qui correspond exactement à :

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad x : \tau_1, \Gamma \vdash e_1 : \tau \qquad y : \tau_2, \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\textbf{match}\ e\ \textbf{with}\ Left\ x \to e_1; Right\ y \to e_2) : \tau}$$

Typage statique et exhaustivité

Comme une somme est (généralement) **fermée**, soit que l'on connait l'ensemble de ses constructeurs, le compilateur peut verifier statiquement (**donc avant l'exécution du programme**) que tout les cas ont été traité dans les branches de la correspondance de motif (`match`).

(Cela permet, entre autre, de résoudre des problèmes bien ennuyant comme la présence, ou non d'une valeur, en la trackant dans le système de type avec un type de ce genre : `type 'a option = Some of 'a | None`, fini les `null` ou `nil` ou `undefined` pouvant survenir sans crier garde.)

Les sommes dans les langages mainstreams

Même si parfois, on peut croire que des `ENUM` comme en Java ou en C sont des sommes, elles sont différentes elles ne permettent pas le même niveau de flexibilité. Cependant, il est possible de les simuler avec du sous-typage au moyen d'interface ou de familles scellées, au moyen d'un **encodage de Church** ou de **smart-cast** (*à la Kotlin*).

Résumé sur les sommes

- les sommes permettent la **disjonction**, soit d'unifier plusieurs valeurs au sein d'un même type.
- Ils sont très pratique pour décrire un traitement homogène. (ie : `une liste de 'a` dont le traitement sera uniforme)
- La cardinalité des produits se calculait en faisant le produit cartésien de tous les membres, pour le sommes ... il s'agit, *logiquement*, de la somme de toutes les branches. (un constructeur sans valeur à la cardinalité 1)
- La représentation la plus générique d'une somme est une valeur de type **either**.

En résumé, les types algébriques sont :

- Des types construits par sommes et produits (soit une **somme de produits**)
- On parle de **types algébriques** parce que les ADTs forment un **semi-anneau** (avec des habitants, des opérateurs ($+$ et $*$) et des lois équationnelles (dont nous n'avons pas parlé))
- Leur cardinalité est toujours facile à calculer
- Couplé au **polymorphisme paramétrique**, à la capacité des types à être **récursifs** et au **pattern matching** ils permettent de décrire **toute sorte de structure de données**.

Par exemple, décrire une liste avec un ADT

Une liste est un type **récursif** qui possède deux cas :

- soit elle est vide
- soit elle contient une tête et une queue (qui est elle même une liste)

Par exemple, décrire une liste avec un ADT

Une liste est un type **récursif** qui possède deux cas :

- soit elle est vide
- soit elle contient une tête et une queue (qui est elle même une liste)

Cette description se traduit assez bien dans une somme de produit :

```
type 'a my_list =
  | Nil
  | Cons of ('a * 'a my_list)
```

Par exemple, décrire une liste avec un ADT

Une liste est un type **récursif** qui possède deux cas :

- soit elle est vide
- soit elle contient une tête et une queue (qui est elle même une liste)

Cette description se traduit assez bien dans une somme de produit :

```
type 'a my_list =
  | Nil
  | Cons of ('a * 'a my_list)
```

Ce qui nous permet de décrire la liste `[1, 2, 3]` de cette manière :
`Cons (1, (Cons (2, Cons (3, Nil))))`

C'est aussi un outil de modelisation puissant

```
type status =
  | Active
  | Inactive

type gender =
  | Male
  | Female
  | Other of string

type user = {
  name   : string
; email  : string
; status : status
; gender : gender
; age    : int
}
```

La concision de descriptiond de types font des ADTs un outil de modèlisation très puissant :

- Le rôle du type est explicite
- sa structure aussi
- solutionne la **crise du booléen**
- peut servir de base à la construction d'un langage **universel** (*ubiquitous language*) parce que **description de la données n'est pas mélangée avec les stratégies de consommation**

C'est aussi un outil de modelisation puissant

```
type status =
  | Active
  | Inactive

type gender =
  | Male
  | Female
  | Other of string

type user = {
  name   : string
; email  : string
; status : status
; gender : gender
; age    : int
}
```

La concision de descriptiond de types font des ADTs un outil de modèlisation très puissant :

- Le rôle du type est explicite
- sa structure aussi
- solutionne la **crise du booléen**
- peut servir de base à la construction d'un langage **universel** (*ubiquitous language*) parce que **description de la données n'est pas mélangée avec les stratégies de consommation**

Complément

Le fait que la cardinalité d'un type soit facile à calculer permet de **restreindre le domaine au maximum** pour tâcher de rendre, dans la mesure du possibles, **les états impossibles, impossible à représenter**.

Mais parfois, la nécéssité d'utiliser du pattern-matching pour déconstruire des sommes (donc potentiellement des structures de données) est un **frein à l'abstraction**. Devoir déconstruire une structure implique de connaitre cette structure et donc de **leaker l'abstraction**

Interdire la construction en dehors du module : `private type`

```
module Age : sig
  type t = private int
  val make : int -> (t, error) result
end = struct
  type t = int
  let make x =
    if x < 0 || x > 125
    then Error (Invalid_age x)
    else Ok x
end
```

On ne peut construire plus que des ages valides mais qui **peuvent toujours être interprétés comme des entiers** (mais qui induit une réduction de cardinalité).

Cacher la représentation du type en dehors du module : `abstract type`

```
module Stack : sig
  type 'a t
  val empty : 'a t
  val push : 'a -> 'a t -> 'a t
  val pop : 'a t -> ('a option, 'a t)
end = struct
  type 'a t = 'a list
  let empty = []
  let push x stack = x :: stack
  let pop stack = match stack with
    | [] -> (None, [])
    | x :: xs -> (Some x, xs)
end
```

On cache complètement le fait qu'une `Pile` est une `Liste`. En dehors du module, une pile ne peut donc plus être interprété comme tel. **ce qui nous permet de changer l'implémentation à tout moment et de ne pas laisser s'échapper la représentation abstraite de notre strucutre.**

Comment appliquer des transformations génériques sur un type abstrait ?

Dans notre exemple notre `Stack.t` est impossible à étendre, il faut donc **que l'implémenteur de la bibliothèque connaisse tous les cas d'usages possibles**. Il existe deux manière de palier ce problème.

Comment appliquer des transformations génériques sur un type abstrait ?

Dans notre exemple notre `Stack.t` est impossible à étendre, il faut donc **que l'implémenteur de la bibliothèque connaisse tous les cas d'usages possibles**. Il existe deux manière de palier ce problème.

- Fournir des fonctions d'injection et de projection (`to_list` et `from_list`) permettant de se reposer sur une autre API plus riche.
- Fournir des combinateurs génériques pour capturer des combinateurs récurrents

```
module Stack : sig
  type 'a t
  val empty : 'a t
  val push : 'a -> 'a t -> 'a t
  val pop : 'a t -> ('a option, 'a t)

  val map : ('a -> 'b) -> 'a t -> 'b t
end = struct
  type 'a t = 'a list
  let empty = []
  let push x stack = x :: stack
  let pop stack = match stack with
    | [] -> (None, [])
    | x :: xs -> (Some x, xs)

  let map f stack =
    let rec aux new_stack stack = match stack with
    | [] -> List.rev new_stack
    | x :: xs -> aux (f x :: new_stack) xs
end
```

('a -> 'b) ? Des types functionnels (ou parfois exponentiels)

$\rightarrow$ est un nouvel opérateur (qui ne fait pas partie de la définition originale des types algébriques) et qui décrit une fonction de `a` dans `b`.

('a -> 'b) ? Des types functionnels (ou parfois exponentiels)

$\rightarrow$ est un nouvel opérateur (qui ne fait pas partie de la définition originale des types algébriques) et qui décrit une fonction de `a` dans `b`.

- Comme pour les sommes et les produits, on peut écrire $a \rightarrow (b \rightarrow c)$ de cette manière $a \rightarrow b \rightarrow c$.
- Sa représentation la plus générique est $a \rightarrow b$
- Sa cardinalité est aussi facile à calculer : $|a \rightarrow b| = |b|^{|a|}$
- Ils sont le **coeur** de la programmation **fonctionnelle**

Pour conclure

- Le terme *algébrique*, dans **types algébriques** à du sens
- On peut composer des types entre eux au moyen d'opérations algébriques (des produits et des sommes)
- Couplé à du polymorphisme paramétrique et à des types potentiellement récursif, c'est une forme concise pour décrire explicitement de la données (métier ou non)
- La privatisation et l'abstraction permettent respectivement de restreindre un domaine et de ne pas faire s'échapper la représentation d'une donnée.
- On peut tout de même appliquer des transformations structurées au moyen de **type fonctionnels**.

Merci pour votre attention

Je serai ravi d'en discuter avec vous pendant la pause ou dans un futur plus lointain.