Programmation fonctionnelle, introduction

Xavier Van de Woestyne

5 Juillet 2016

Dernier Cri

Bonjour !

- @vdwxv sur Twitter, @xvw sur Github ;
- développeur à Dernier Cri ;
- travail de recherche qui n'avance pas ;
- de 2000 à 2008 : **PHP** (et d'autres langages impératifs) ;
- depuis 2008 : **Ruby**, Erlang, Haskell, OCaml/Fsharp ;
- Meetup LilleFP

Sommaire

Préambule

- Objectifs
- Mise en contexte historique
  - Logique formelle et décision
  - Le λ-Calcul

Programmation fonctionnelle

- Première définition
- Les fonctions
  - Curryfication
  - Expressivité
  - Compositions

Les types comme outil de design

- Types algébriques
  - Apport des types

Propriété par le type : le monoïde

Conclusion

- Influence et application
- Plus de super-pouvoirs
- Fin

Préambule

Objectifs

- Présentation sommaire ;
- démystification (je suis nul en math...) ;
- présentation de motifs ;

Présenter des concepts appropriables dans 80% des langages modernes.

Pourquoi ?

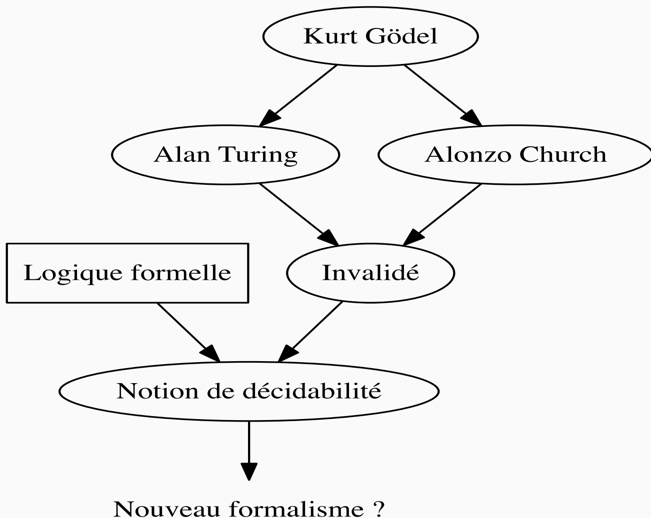
- Passé et futur de la programmation ;
- facile à appréhender ;
- OCaml, Haskell, FSharp, Scala, Clojure, Ruby, Pytho, JavaScript, C++, CSharp, Smalltalk, Java ;
- Elixir, Elm ;
- FRP, continuations/promesses, parallélisme ;
- découverte.

Tout commence avec ...

Le problème de la décision

Comment définir l'ensemble des expressions calculables ?

> *En logique formelle, une propriété mathématique est dite décidable s'il existe un procédé mécanique qui détermine, au bout d'un temps fini, si elle est vraie ou fausse dans n'importe quel contexte possible.*



Kurt Gödel
Alan Turing
Alonzo Church
Logique formelle
Invalidé
Notion de décidabilité
Nouveau formalisme ?

Formalismes pour représenter l'intégralité des fonctions calculables

Trois réponses pour le prix d'une ...

- **1935** : le lambda-calcul de Church ;
- **1935** : les fonctions récursives de Gödel et Kleen ;
- **1936** : les machines de Turing.

C'est une découverte (et non une invention).

Types et preuves

Les systèmes formels servant la logique sont étroitement liés aux preuves, donc, par extension, aux systèmes de types :

- Lutter contre le paradoxe de la logique formelle (ex : Paradoxe du Barbier) ;
- Correspondance preuve-programme (l'isomorphisme de Curry-Howard) : logique mathématique <-> Informatique théorique ;
- Hindley-Milner, Martin Löf.

Le λ-Calcul, la base

- Un langage théorique ;
- la base de tout langage fonctionnel ;
- initialement non typé, mais peut l'être (Système F).

Seulement trois éléments gramaticaux

- **Les variables** : x, y, ..., des lambda-termes ;
- **les applications** : f x, est un lambda-terme si f et x sont des lambda-termes ;
- **les abstractions** : $\lambda x.v$ est un lambda-terme si x est une variable et v un lambda-terme.

Quelques constructions

Les entiers de Church :

$$0 = \lambda fx.x$$
$$1 = \lambda fx.f\ x$$
$$2 = \lambda fx.f\ (f\ x)$$
$$3 = \lambda fx.f\ (f\ (f\ x))$$

Les Boolééns :

$$true = \lambda ab.a$$
$$false = \lambda ab.b$$
$$or = \lambda pq.p\ p\ q$$
$$and = \lambda pq.p\ q\ p$$

Approcher le « pur » orienté Objets

```ruby
module Number

  class Number
    def succ
      NonZero.new(self)
    end
  end

  class Zero < Number
  end

  class NonZero < Number
    def initialize(pred)
      @pred = pred
    end
  end

end
```

```ruby
module Bool

  class True
    def or(other)
      self
    end
    def and(other)
      other
    end
  end

  class False
    def or(other)
      other
    end
    def and(other)
      self
    end
  end

end
```

Seul l'IO ne semble pas représentable en pur objet.

Programmation fonctionnelle

Un langage fonctionnel

Un langage fonctionnel est simplement un langage qui peut manipuler des fonctions comme des valeurs (lambdas).

1935 λ-calcul ;

1958 Lisp (premier langage moderne !)

Un langage fonctionnel peut être

- Pur ou impur ;
- strict ou non strict*.

* Les paramètres doivent être évalué complètement avant d'être appelés.

Concept principal : les fonctions

```
let value = 12

let hello () = print_endline "Hello World"
let increment x = x + 1
let add x y = x + y

let mult = fun x y -> x * y
```

A propos des fonctions

- Les fonctions sont des valeurs ;
- on peut en utiliser partout !
- Les fonctions ne **prennent qu'un seul argument** (héritage du λ-calcul).

```
(* Facile, un seul argument : un n-uplet*)
let add (x, y) = x + y

(* wtf... ça marche *)
let add' x y = x + y
```

Curryfication

En λ-calcul, les n-uplet n'existent pas encore. On utilise la logique combinatoire de Haskell Curry et Moses Schönfinkel pour utiliser plusieurs arguments :

- $let\, add\, x\, y\ =\ x + y$
- $let\, add\ =\ fun\, x\, y\ \rightarrow\ x + y$
- $let\, add\ =\ fun\, x\ \rightarrow\ (fun\, y\ \rightarrow\ x + y)$

Les fonctions peuvent être des valeurs d'entrée, mais aussi des valeurs de retour.

Conséquence (cool) : l'application partielle

```
let add x y = x + y
let increment = add 1
```

add 1 renvoie une fonction qui attend le paramètre y

Expressivité : moins de dépendance à la grammaire

```php
<?php
foreach ([1, 2, 3] as $elt) {
    echo $elt;
}
```

Expressivité : moins de dépendance à la grammaire

```
(* Listes : [1; 2; 3] == 1 :: 2 :: 3 :: [] *)

let rec foreach list f =
  match list with
  | [] -> ()
  | x :: xs -> f x ; foreach xs f

let () = foreach [1; 2; 3] print_int
```

Les fonctions sont des légos...

Au sens figuré :)
L'enjeu du programmeur fonctionnel est de combiner des fonctions entre elles pour construire des fonctions plus complexes.

- une action atomique est une fonction ;
- une collection d'action atomique est un service ;
- une collection de services est une application.

Ce qui permet de reproduire une grande partie des motifs de conceptions de l'orienté objet...

Les types comme outil de design

Un système de type

- Une extension à la théorie des ensembles ;
- donne une notion de domaines aux fonctions ;
- garantit une certaines sûrté ;
- peut inférer un type pour une valeur ;
- peut être un outil de design précis.

Types algébriques

Types primitifs :

- int ;
- float ;
- char ;
- bool ;
- string etc.

Composition de types :

```
-- Produits (Conjonction)
type couple =
  (int * string)

-- Sommes (Disjonction)
type couleur =
  | Carreau
  | Pique
  | Trefle
  | Coeur
```

Plus de types

Types paramétrés :

```
type 'a option =
  | Some of 'a
  | None
```

- α list ;
- (α, β) Hashtbl.t

Types récursifs :

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

```
[1;2]=Cons(1,Cons (2,Nil))
```

Alias de types :

type prenom = string

Types abstraits

Type des fonctions

- x = 'h' : $char$;
- id x : $\alpha \to \alpha$;
- add x y : $int \to int \to int$;
- couple x y = (x, y) : $\alpha \to \beta \to (\alpha * \beta)$;
- map f list = $(\alpha \to \beta) \to \alpha\ list \to \beta\ list$.

Déconstruction de types

On peut déconstruire des types composés au moyen du Pattern-Matching :

```
let defaultValue default opt =
  match opt with
  | Some x -> x
  | None -> default
```

Apport des types

- Sécurité du programme (vérification statique !) ;
- documentation : $map = (\alpha \rightarrow \beta) \rightarrow \alpha\ list \rightarrow \beta\ list$;
- Domain-Driven-Design : Types nommés/alias de types, options etc.
- Possibilité de faire découler des règles simples au moyen d'assertions simples.

Propriété par le type : le monoïde

Le monoïde

Tout type α qui possède un opérateur de composition ($\oplus$) et un élément neutre (e) et qui est régit par ces trois règles :

- $\oplus : \alpha \to \alpha \to \alpha$ (la cloture)
- $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ (l'associativité)
- $e \oplus x = x$ et $x \oplus e = x$ (l'élément neutre)

est un monoïde.

Par exemple, les entiers

- $\oplus = (+)$
- $1 + (2 + 3) = (1 + 2) + 3$
- $e = 0$

Par exemple, les strings

- $\oplus = String.concat$
- $e = ''$

Par exemple, les listes

- $\oplus = List.concat$
- $e = []$

Par exemple, les endofonctions $(\alpha \to \alpha)$

- $\oplus = (.)\ (f.gx = f(g(x)))$
- $e = id$

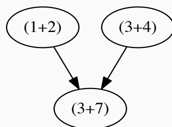
Apport des règles : l'opérateur OPLUS

Transforme une opération binaire en opération qui fonctionne sur des listes (reduce/fold)

$fold \oplus (liste\ de\ \alpha) : \alpha$

Apport des règles : l'associativité

Accumulation incrémentale, parallélise facilement. Par exemple :
$1 + 2 + 3 + 4$



Chaque noeud peut être calculé sur une unité de calcul différente (et parallèle, par exemple). Ou chaque résultat peut être accumulé.

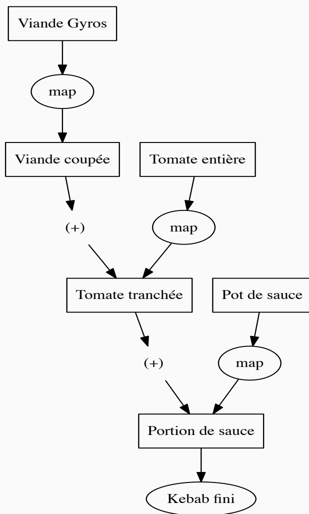
Apport des règles : l'élément neutre

Permet facilement de traiter les cas où des données sont invalides/manquantes.

Dériver un type α en monoïde

Il suffit d'implémenter l'opérateur $\oplus$, en respectant les règles évoquées.

Il est possible de décorer un type β en un monoïde au moyen d'une fonction de map.


Viande Gyros
map
Viande coupée
Tomate entière
(+)
map
Tomate tranchée
Pot de sauce
(+)
map
Portion de sauce
Kebab fini

Et voici ... Map/reduce de Google

Si on occulte la notion de Speed Up. (but no time ...)

Conclusion

Influence et application

- TypeScript, Flow, HaXe
- Applicable à des langages non typés (les promesses)
- Evolution des langages (Java, CSharp)
- révolutionner la navigation dans le web (avec des continuations !)

Plus de super-pouvoirs

Jutsus fonctionnels

- Monades
- Comonades
- Flèches
- Structure de données purement fonctionnelles

Typages

- Types existentiels
- Types fantômes
- Types algèbriques généralisés

Fin

Merci, questions, remarques ?