À la découverte du langage OCaml

Xavier Van de Woestyne

LilleFP6

Septembre 2017

WHOAMI

- Xavier Van de Woestyne
- **xvw** sur Github et **vdwxv** sur Twitter
- Développeur chez **Fewlines** (Elixir, Elm)
- Erlang/**Elixir**, **OCaml**, F#, Haskell, **Elm**, Ruby, **Nim**, Scheme, Ur, etc.
- J'aurai un blog (https://xvw.github.io)

LilleFP : pour des rencontres techniques

- **Généralisation** des sujets des événements
- Mutation de **l'ambiance** générale
- Ouvert à beaucoup de sujets **techniques** (Pourquoi pas vous ?)
- Rejoignez-nous sur **Slack** : https://slackin-lillefp.herokuapp.com/

Merci à nos partenaires (de tous temps)

Fewlines, Dernier Cri, Mozilla foundation, Synbioz, Epitech Lille et TakeOff talks.

Pourquoi cette présentation ?

Pour réfuter cette affirmation :

> *OCaml est un langage dédié à la recherche.*

Et pour enrichir cette affirmation :

> *OCaml est un bon langage pour faire de l'analyse et des compilateurs.*

Car, OCaml est un langage **très très** polyvalent et c'est donc intéressant d'en survoler rapidement les fonctionnalités.

Malheureusement

Comme je ne suis pas illimité en temps :

- Je survolerai les fonctionnalités du langage
- L'objectif de la présentation est de stimuler un **intérêt**
- Je ferais avec joie un Workshop OCaml/ReasonML

Qu'est ce que, rapidement, OCaml

Un langage :

fonctionnel, impératif, typé fortement et statiquement, compilé/inteprété, portable et performant, issu de la recherche.

Utilisé dans l'industrie :

Airbus, Tezos, Facebook, Mozilla, Docker, JaneStreet, Citrix etc.

Qui a inspiré :

F#, F*, Ur, Rust, Reason, Hack, Swift... etc.

Success-Stories

MirageOS, Coq, MIDonkey, Tezos, aMSN, Unison, HHVM, Infer, Flow, F* et beaucoup d'autres !


Companies utilisant OCaml

Pourquoi apprendre OCaml

- Etendre votre boite à outils
- Ouvrir de nouvelles perspectives (pour d'autres langages)
- Devenir polyglote

OCaml m'a permis de devenir un meilleur programmeur Ruby.

Sommaire

1958 Lisp

- Développé par J. McCarthy
- Impératif et fonctionnel
- Typé dynamiquement
- Premier langage moderne
- Beaucoup d'enfants (Common Lisp, Scheme, Arc, Racket, Clojure, Emacs Lisp)

De ML à Reason

1970 ML (Meta Language)

- Développé par R. Milner et son équipe de l'université de Edimbourg
- Impératif et fonctionnel
- Typé statiquement (pour le système de preuve LCF)
- Inférence de types, correspondance de motifs et modules puissants

De ML à Reason

1983 SML (Standard ML)

- Une standardisation de ML
- Plusieurs implémentations (SML of New Jersey, Moscow ML, MLTon, Poly/ML)

De ML à Reason

1985 Caml (Categorical Abstract Machine Language)

 A History of Caml

- Implémentation Française de ML
- Développé pour compiler vers une machine Lisp (LLM3)

De ML à Reason

1990 Caml Light

- Nouvelle implémentation par Xavier Leroy (entre autre)
- Compilant vers un byte-code interprété en C
- Portable
- Avec un très bon garbage collector

De ML à Reason

1995 Caml Special Light

- Ajout d'une compilation vers du code natif
- Performances compétitives (proche de C++)
- Système de module très évolué inspiré de celui standardisé dans SML
- Utilisé pour enseigner la programmation en Prépa, jusqu'il y a peu...

De ML à Reason

1996 OCaml (Objective-Caml)

- Programmation orientée objets (avec inférence) *type-safe*
- 2000 : Ajout de fonctionnalités expérimentales, variants polymorphiques, arguments optionnels, labellisés, méthodes polymorphiques etc.
- Performances encore acrues

De ML à Reason

2002 F# (comme C# mais avec un F pour fonctionnel)

- Avec un noyau dérivé de OCaml
- Syntaxe allégée
- Compatible .NET
- Pas de modules au sens ML
- Expressions de calcul

De ML à Reason

2015/2016/2017 ReasonML

- Un nouvelle syntaxe pour OCaml
- Ajout de pleins de features modernes “à la Elm”
- Focus sur les outils
- Pour plaire à la communauté JavaScript

OCaml est donc un langage assez ancien, qui a eu l'occasion d'être éprouvé !

Outils

- **ocamlc** : compilateur vers le byte-code
- **ocamlopt** : compilateur vers du code natif
- **opam** : un gestionnaire de paquet (standard)
- **merlin** : IDE-features pour tous les éditeurs modernes (Emacs, VSCode, Atom, Sublim Text et … euh veem).
- **js_of_ocaml** : compilateur JavaScript
- **BuckleScript** : transpileur JavaScript
- **gen_js_api** : un créateur de binding OCaml <-> JavaScript

Et beaucoup d'autres

Spacetime, Camlp4, Lwt, Tests, et pléthore de build-system (**ARGH**).

La partie qui arrive va aller... très vite...

Mais pas de panique, un LiveCoding en fin de présentation devrait fixer tout ça ... :D

Syntaxe (très rapidement)

Variables et fonctions

```
let var = 10
let f x = x + 1
let id x = x

let est_pair x = (x mod 2) = 0

let rec forever f default =
  let new_result = f default in
  forever f new_result
```

Une fonction retourne **toujours** quelque chose.

Syntaxe (très rapidement)

Lambdas

```
let f = fun x y z -> ...
let g = function x -> ...

let _ = List.map (fun x -> x + 1) [1; 2; 3]
let _ = List.map succ [1; 2; 3]
let ( >>= ) x f =
  x
  |> List.map f
  |> List.concat
```

Syntaxe (très rapidement)

Curryfication || Application partielle

```
let f x y = x + y
let plus2 = f 2
```

- Une fonction ne peut prendre qu'un seul argument
- Une fonction peut renvoyer une fonction

Syntaxe (très rapidement)

Correspondance de motif

```
match value with
| (10, 20) -> "On a la valeur 10 et 20"
| (x, y) when y = 2*x -> "y vaut le double de x"
| (_, 3) | (3, _) -> "L'une des deux valeur vaut 3"
| _ -> "cas trivial"
```

Syntaxe (très rapidement)

Contrôle classiques

- if/else
- for/while (mais on peut s'en passer)

Syntaxe (très rapidement)

Listes

```
[1; 2; 3]
= 1 :: [2; 3]
= 1 :: 2 :: [3]
= 1 :: 2 :: 3 :: []
```

Et beaucoup d'autres structures de données :
Array, String, Hashtbl, Set, etc.

Syntaxe (très rapidement)

Exemple de l'implémentation de List.iter

```
let rec iter f list =
  match list with
  | [] -> ()
  | x :: xs ->
    let () = f x in
    iter f xs
```

A propos des types

Types primitifs

- `int`
- `float`
- `bool`
- `string`
- `int -> int -> int`
- `(int * int) -> int`
- etc.

A propos des types

Polymorphisme paramétrique (Génériques)

`[1; 2; 3] :: int list`

- Globalement, une liste est de type `'a list`
- On peut paramétrer un type par n autres types : `('a, 'b, 'c) t`

Globalement, le compilateur peut “déduire les types en fonction de leur utilisation”.
Cependant, il existe plusieurs cas où spécifier les types peut être un plus :

- Fixer une ambiguïté
- Prototyper

Créons nos propre types

- Alias de types
- Types algébriques
  - Types sommes (disjonctions)
  - Types produits (conjonctions)

Alias

```
type firstname = string
type lastname = string
type age = int
type length = float
```

Créons nos propres types

Types sommes

```
type switch =
| On
| Off

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

type ('a, 'b) either =
| Left of 'a
| Right of 'b
```

Créons nos propres types

Utilisation de la correspondance de motifs pour déconstruire des variants

```
let map f opt =
| Some x -> Some (f x)
| None -> None
```

Types sommes pour définir les listes

```
type 'a list =
| []
| (::) of ('a * 'a list)
```

Créons nos propres types

Les produits : n-uplets

```
type human = ( firstname * lastname)
type point = ( int * int )
```

Les produits : records

```
type human = { name : lastname ; firstname : firstname }
type point = { x: int; y: int}
let p = { x = 10; y = 20}
let py = p.y
```

Un record peut avoir des champs mutables

Objets et le sous-typage par inclusion

> *On ne s'étendra pas sur les objets (par soucis de temps et aussi parce que ses usages sont ...* marginaux*)*

```
let f x = x # foo (12) ;;
val f : < foo : int -> 'a; .. > -> 'a
```

Classes, Classes abstraites, Interfaces, héritage simple et multiple, objets pures et mutables.

Une page inutile

Cette diapositive n'est ici que pour vérifier que tu as eu la décence de lire ce que je t'ai envoyé.

A vous tous, si cette diapositive est présente, vous pouvez HUER le conférencier.
Cordialement, **P.**

Programmation modulaire

Un module est un regroupement sémantique de types, structures de données et de fonctions.

- Séparation d'un module et de son interface
- L'interface est un contrat d'implémentation (donc à implémenter... au début)
- L'interface expose l'API d'un module (facilite l'évolution d'un module)
- Possibilité d'avoir des sous-modules
- Il est possible de construire des modules via un ou plusieurs autres modules
- Il est possible d'utiliser des modules comme des valeurs typées.

Implémenter un interpréteur Brainfuck dans un style fonctionnel

Un exercice classique mais qui permet plusieurs choses :

- De la définition de types
- De la récursivité
- Un peu de modularité (toute mignone)

Oui… OCaml est vraiment bon pour implémenter ce genre d'exercice.

```
Je suis un Hello World en Brainfuck !
++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++.>+.+++++++..+++.
>++.<<+++++++++++++++.>.+++.------.--------.>+.>.
```

Aller plus loin avec OCaml

- Les NIF's (via externals)
- La concurrence avec Lwt
- Types extensibles
- Variants polymorphiques
- Faire des applications web
- Types universel/existentiels
- Types fantômes
- Types algébriques généralisés
- Voir Modular Implicits et OCaml-multicore (et les effets)
- Créer son propre unikernel avec Mirage

Les points faibles du langage

- Bibliothèque standard pauvre (mais ...)
- Un éco-système “dur” à prendre en main pour un débutant (build-system)
- Des manuels et de la documentation exhaustive ... mais peu “funky” (ou très “funky”)

Les points forts du langages

- Un langage expressif, mature, sur et performant
- Un éco-système riche (avec une communauté croissante)
- Un système de type ... génial
- Des contributeurs sérieux
- Une modernité qui vient via Facebook et Bloomberg

bref, OCaml est un langage qui a de l'avenir, en tant qu'inspiration, en tout cas, et vous devriez vous y intéresser !

FIN

- Questions, remarques, clarifications?

Merci à vous !