La programmation modulaire,
au-delà de l'espace de nom

Xavier Van de Woestyne - **Margo Bank** - Lambda Lille

@vdwxv
xvw@merveilles.town
github.com/xvw
https://xvw.github.io

Bruxelles, Lille, Paris

Objectifs
assumés

Exprimer les notions :

- de modularisation
- de compilation séparée

Présenter un langage de modules dans un un contexte statiquement typé

Exprimer les différents niveaux de valeurs dans les langages

Survol de quelques use-cases

Objectifs
cachés

- Faire quelques précisions terminologiques
- Faire la promotion du langage OCaml

- Issu (entre autre) de la recherche Française
- En constante évolution depuis les années 80
- Un langage de programmation impur
- Fonctionnel et impératif
- Typé statiquement (avec un système de types riche)
  - ADTs
  - Variants polymorphes & Objets (riches)
  - GADTs + types ouverts
  - Modules
- Byte Code, Natif et JavaScript (depuis 2003 !)
- Concis, performant, portable
- Eco-système riche

Disclaimer

Modularisation
Compilation séparée

Modularisation
Compilation séparée

- On peut découpler le travail sur un même programme
- Facilite la définition de la structure “haut-niveau” du programme
- Permet de rendre le programme potentiellement plus fiable

Fonctionnalités “simples”

Regroupement et namespacing

```ocaml
module Option =
struct

   type 'a t =
     | Some of 'a
     | None

   let map f = function
     | None → None
     | Some x → Some (f x)

end
```

```ocaml
let x = Some 10
let y = Option.map succ x
```

Regroupement et namespacing

```
module Option =
struct

   type 'a t =
     | Some of 'a
     | None

   let map f = function
     | None → None
     | Some x → Some (f x)

end
```

```
let x = Some 10
let y = Option.map succ x
```

```
open Option
let z = map pred x

open! Option
let z = map pred x
```

Regroupement et namespacing

```ocaml
module Option =
struct

   type 'a t =
     | Some of 'a
     | None

   let map f = function
     | None → None
     | Some x → Some (f x)

end
```

```ocaml
let x = Some 10
let y = Option.map succ x
```

```ocaml
open Option
let z = map pred x

open! Option
let z = map pred x

let f =
  let open Option in
  map id x
```

Regroupement et namespacing

```
module Option =
struct

    type 'a t =
      | Some of 'a
      | None

    let map f = function
      | None → None
      | Some x → Some (f x)

end
```

```
let x = Some 10
let y = Option.map succ x
```

```
open Option
let z = map pred x

open! Option
let z = map pred x

let f =
  let open Option in
  map id x

let g = Option.(map id x)
```

Regroupement et namespacing

```
module Option =
struct

   type 'a t =
     | Some of 'a
     | None

   let map f = function
     | None → None
     | Some x → Some (f x)

end
```

```
let x = Some 10
let y = Option.map succ x
```

```
open Option
let z = map pred x

open! Option
let z = map pred x

let f =
  let open Option in
  map id x

let g = Option.(map id x)
```

```
module New_Name = My_long.Module.Name.L
```

Encapsulation et visibilité

```ocaml
module Option =
struct

   type 'a t =
     | Some of 'a
     | None

   let map f = function
     | None → None
     | Some x → Some (f x)

end
```

```ocaml
module Option :
sig

   type 'a t =
     | Some of 'a
     | None

   (** [Option.map f opt] unwrap [opt] and apply [f] *)
   val map : ('a → 'b) → 'a t → 'b t

end
```

Encapsulation et visibilité

```
module Option =
struct

   type 'a t =
     | Some of 'a
     | None

   type an_internal_type = int

   let map f = function
     | None → None
     | Some x → Some (f x)

   let an_internal_function x = x + 1

end
```

```
module Option :
sig

   type 'a t =
     | Some of 'a
     | None

   (** [Option.map f opt] unwrap [opt] and apply [f] *)
   val map : ('a → 'b) → 'a t → 'b t

end
```

Abstraction de types (encapsulation II)

```
module Option =
struct

   type 'a t =
     | Some of 'a
     | None

   let some x = Some x
   let none = None

   let map f = function
     | None → None
     | Some x → Some (f x)

end
```

```
module Option :
sig

   type 'a t

   val some : 'a → 'a t
   val none : 'a t
   val map : ('a → 'b) → 'a t → 'b t

end
```

Abstraction de types (encapsulation II)

```
module Age = struct

  type t = int

  let make x =
    if x < 0 then None
    else Some x

end
```

```
module Age : sig

  type t
  val make : int → t option

end
```

Extension de modules

```
module My_list = struct
  include List
  let flat_map f x = join (map f x)
end
```

```
module My_list : sig
  include module type of List
  val flat_map : ('a → 'b list) → 'a list → 'b list
end
```

Extension de modules

```
module My_list = struct
  include List
  let flat_map f x = join (map f x)
end

module List = My_list
```

```
module My_list : sig
  include module type of List
  val flat_map : ('a → 'b list) → 'a list → 'b list
end
```

Donc, les modules sont juste des espaces noms…

```ocaml
module List = struct

  type 'a t = 'a list
  let return x = [x]
  let flat_map f x = List.(join (map f x))

end

module Option = struct

  type 'a t = 'a option
  let return x = Some x
  let flat_map f = function
    | Some x → f x
    | None → None
end
```

```ocaml
module List : sig
   type 'a t = 'a list
   val return : 'a → 'a t
   val flat_map : ('a → 'b t) → 'a t → 'b t
end

module Option : sig
   type 'a t = 'a option
   val return : 'a → 'a t
   val flat_map : ('a → 'b t) → 'a t → 'b t
end
```

```ocaml
module List : Monad = struct

  type 'a t = 'a list
  let return x = [x]
  let flat_map f x = List.(join (map f x))

end

module Option : Monad = struct

  type 'a t = 'a option
  let return x = Some x
  let flat_map f = function
    | Some x → f x
    | None → None
end
```

```ocaml
module type Monad = sig
   type 'a t
   val return : 'a → 'a t
   val flat_map : ('a → 'b t) → 'a t → 'b t
end
```

```
module List : Monad
  with 'a t = 'a list = struct

  type 'a t = 'a list
  let return x = [x]
  let flat_map f x = List.(join (map f x))

end

module Option : Monad
  with 'a t = 'a option = struct

  type 'a t = 'a option
  let return x = Some x
  let flat_map f = function
    | Some x → f x
    | None → None
end
```

```
module type Monad = sig
   type 'a t
   val return : 'a → 'a t
   val flat_map : ('a → 'b t) → 'a t → 'b t
end
```

```ocaml
module type Cmp = sig
  type t
  val cmp : t → t → int
end
```

```ocaml
module Age = struct

  type t = int

  let cmp x y =
    if (x > y) then 1
    else if (x < y) then -1
    else 0

  let to_string x =
    String.from_int x

end
```

```ocaml
module type Show = sig
  type t
  val to_string : t → string
end
```

```ocaml
module Gender = struct

  type t =
    | Male
    | Female
    | Other of string

  let cmp _ _ = 0
  let to_string = function
    | Other x → x
    | Male → "Male"
    | Female → "Female"

end
```

Value level

```
val x = 25
```

λ(Value level)

```
val x = f(25)
```

Type level

λ(Value level)

```
val x : Int = f(25)
```

λ(Type level)

λ(Value level)

```haskell
data T1 f a = T1 (f a) -- T1 :: (* -> *) -> * -> *
```

Haskell

$$\begin{tikzcd}\lambda(\text{Type level}) \arrow[d, "\text{Réification}"] \\ \lambda(\text{Value level})\end{tikzcd}$$

```kotlin
inline fun <reified T, A> f(x: A) = x is T
```

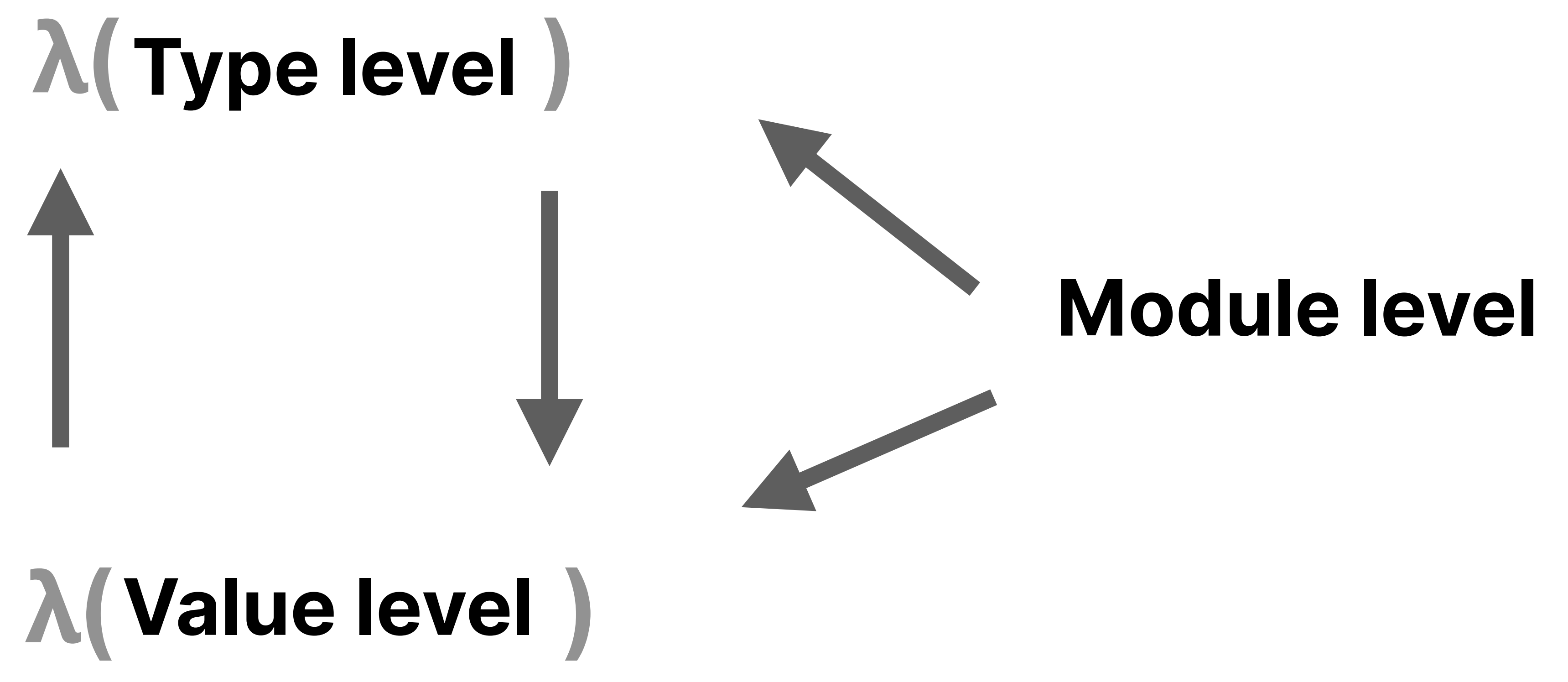
$$\begin{tikzcd}
\lambda(\text{Type level}) \arrow[d, bend left, "\text{Réification}"] \\
\lambda(\text{Value level}) \arrow[u, bend left, "\text{Types dépendants}"]
\end{tikzcd}$$

```
append :
    (xs : Vect lengthA elem) →
    (ys : Vect lengthB elem) →
    Vect (lengthA + lengthB) elem
```

Idris

$$\begin{tikzcd}
\lambda(\textbf{Type level}) \arrow[d, shift left=2] \\
\lambda(\textbf{Value level}) \arrow[u, shift left=2]
\end{tikzcd}$$

Module level



```
let print (type a) (module Showable : Show with type t = a) (x : a) =
  print_endline (Showable.to_string x)

let () = print (module Gender) Female
```

$$\begin{tikzcd}
\lambda(\text{Type level}) \arrow[dd, shift left] & \\
& \text{Module level} \arrow[ul] \arrow[dl, shift left] \\
\lambda(\text{Value level}) \arrow[uu, shift left] \arrow[ur, shift left] &
\end{tikzcd}$$

```ocaml
module type Succ = sig
  val succ : int → int
end

let module_from_value_level () =
  ( module struct
    let succ x = x + 1
  end : Succ )
```

```
let flat_map (type ???) (module M : Monad with type 'a t = ???) f x =
    M.flat_map f x
```

$$\begin{tikzcd}
\lambda(\text{Type level}) \arrow[dd, shift left] & \\
& \lambda(\text{Module level}) \arrow[ul] \arrow[dl, shift left] \\
\lambda(\text{Value level}) \arrow[uu, shift left] \arrow[ur, shift left] &
\end{tikzcd}$$

Le langage de module

- **Valeurs :** Structures
- **Types :** Signatures
- **Fonctions :** Functor*

*** A ne pas confondre avec les foncteurs de Haskell, ni ceux de Prolog, ni ceux de C++, ni ceux de la théorie des catégories, ni ceux de la linguistique.**

```ocaml
module My_functor (M : My_sig) : My_new_module =
struct

    (** Corps du module à produire *)

end

module T = My_functor (An_another_module)
```

Un exemple concret

```
module type Monad = sig

  type 'a t
  val return : 'a -> 'a t
  val flat_map : ('a -> 'b t) -> 'a t -> 'b t
  val map : ('a -> 'b) -> 'a t -> 'b t

  val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t
  val ( <$> ) : 'a t -> ('a -> 'b) -> 'b t
  val ( <*> ) : ('a -> 'b) t -> 'a t -> 'b t
end
```

Un exemple concret

```
module type Monad = sig

  type 'a t
  val return : 'a -> 'a t
  val flat_map : ('a -> 'b t) -> 'a t -> 'b t
  val map : ('a -> 'b) -> 'a t -> 'b t

  val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t
  val ( <$> ) : 'a t -> ('a -> 'b) -> 'b t
  val ( <*> ) : ('a -> 'b) t -> 'a t -> 'b t
end
```

Un exemple concret

```
module type Monad = sig

  type 'a t
  val return : 'a → 'a t
  val flat_map : ('a → 'b t) → 'a t → 'b t
  val map : ('a → 'b) → 'a t → 'b t

  val ( >>= ) : 'a t → ('a → 'b t) → 'b t
  val ( <$> ) : 'a t → ('a → 'b) → 'b t
  val ( <*> ) : ('a → 'b) t → 'a t → 'b t
end
```

```
module type MonadRequirement = sig
  type 'a t
  val return : 'a → 'a t
  val flat_map : ('a → 'b t) → 'a t → 'b t
end
```

Un exemple concret

```
module Monad_make (M : MonadRequirement) : Monad =
struct
  include M
  let ( >>= ) x f = flat_map f x
  let map f x = x >>= fun i -> return (f i)
  let ( <$> ) = map

  let ( <*> ) fs xs =
     fs >>= fun f ->
     xs >>= fun x ->
     return (f x)
end
```

Un exemple concret

```
module List_monad = Monad_make(struct
   type 'a t = 'a list
   let return x = [x]
   let flat_map f x = List.(join (map f x))
end)
```

```
module Option_monad = Monad_make(struct
  type 'a t = 'a option
  let return x = Some x
  let flat_map f = function
    | Some x → f x
    | None → None
end)
```

Side note sur les Applicatives/Génératives

Pour conclure

- Les modules permettent de structurer un programme et de couvrir une grande partie des usages des espaces de noms
- Ils sont des valeurs de premières ordres en OCaml
- Le langage de module est tout petit langage de programmation fonctionnel statiquement typé
- Malgré l'absence de Higher Kinded Types, on peut faire du code générique (un peu plus verbeux)

Dans le futur, Polymorphisme adHoc avec les modules implicites

Merci.