La programmation modulaire

au-delà des espaces de noms

Xavier Van de Woestyne - https://xvw.github.io

Margo Bank

Objectifs de la présentation

- Se mettre d'accord sur certains points terminologiques ;
- Présenter la programmation modulaire dans un langage statiquement typé ;
- Présenter un langage de module expressif ;

Pourquoi cette présentation

Modularisation

Compilation séparée

Les bienfaits de la programmation modulaire

- On peut découpler le travaille sur un même programme ;
- Ça permet de définir la structure “haut niveau” du programme ;
- Ça permet de rendre le programme potentiellement plus fiable.

Séparation de l’implémentation et de l’interface

list.ml

```
type 'a t = 'a list

let map f list = ...

let iter f list = …

let internal = ...
```

list.mli

```
type 'a t = 'a list


val map : ('a -> 'b) -> 'a t -> 'b t

val iter : ('a -> unit) -> 'a t -> unit
```

A l’usage

- Une fois compilé, on bénéfice, dans notre scope de compilation d’un module **List**

```
let () =
  List.map (fun x -> x + 1) [1;2;3]
```

Et on bénéficie de mécanismes d’ouverture

```
open List
let () =
  map (fun x -> x + 1) [1;2;3]
```

```
open! List
let () =
  map (fun x -> x + 1) [1;2;3]
```

```
let () =
  let open List in
  map (fun x -> x + 1) [1;2;3]
```

```
let () =
  List.( map (fun x -> x + 1) [1;2;3]
         |> iter print_int
   )
```

On a des espaces de noms, un peu pauvres.

Sous modules

test.ml

```
module List =
struct

  type 'a t = 'a list
  let map f list = ...
  let iter f list = ...
end
```

test.mli

```
module List :
sig

  type 'a t = 'a list
  val map : ('a -> 'b) -> 'a t -> 'b t
  val iter : ('a -> unit) -> 'a t -> unit
end
```

A l’usage

```
Test.List.map
```

Inclusion et extension

list.ml

```
include List
let my_extension list = ...
```

list.mli

```
include (module type of List)
val my_extension : 'a list -> 'b list
```

Signature libres

```
module type MAPPABLE =
sig
  type 'a t
  val map : ('a -> 'b) -> 'a t -> 'b t
end

module List : MAPPABLE with type 'a t = 'a list = struct
  type 'a t = 'a list
  let map = ...
end
```

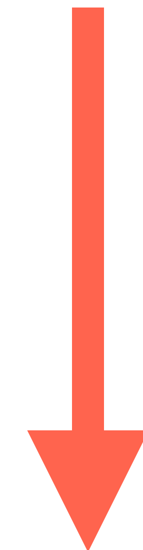
Le langage de module

Value-level

Type-level

Value-level

Type-level



Value-level

Type-level

Types dépendants

Réification

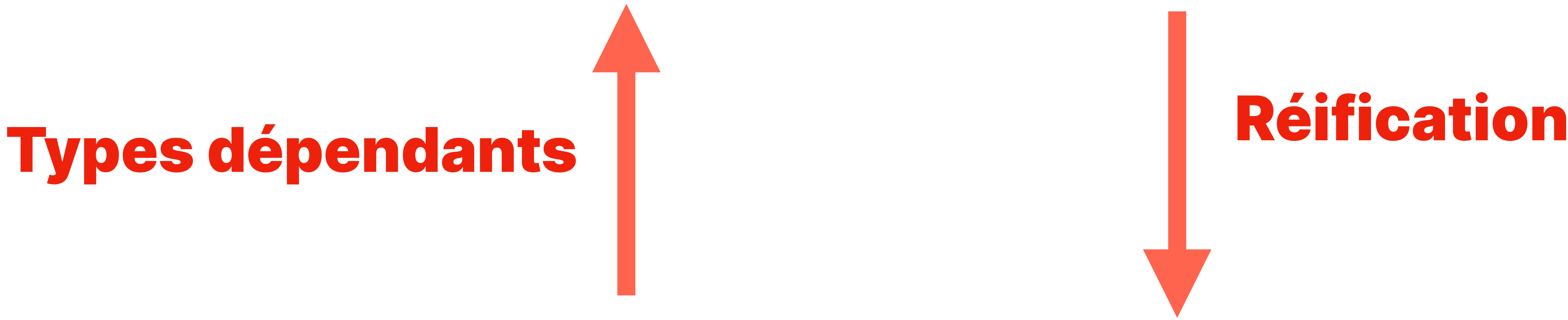
Value-level

Type-level

Types dépendants

Réification

f(Value-level)


f(Type-level)
Types dépendants
Réification
f(Value-level)

Module-level

f(Type-level) f(Value-level)

f(Module-level)

f(Type-level)

f(Value-level)

Functor, functor, functor...

Foncteurs applicatifs

Une module “spécial”, qui prend un, ou plusieurs modules en argument pour produire un nouveau module. Donc, une fonction de le module level.

Par example

```
module type MAPPABLE =
sig
  type 'a t
  val map : ('a -> 'b) -> 'a t -> 'b t
end

module type ITERABLE =
sig
  type 'a t
  val iter : ('a -> unit) -> 'a t -> unit
end
```

On voudrait un foncteur de ce type :

MAPPABLE → ITERABLE x MAPPABLE

Par example

```
module type MAPPABLE_AND_ITERABLE = sig
  include MAPPABLE
  include ITERABLE with type 'a t := 'a t
end
```

Par example

```
module Iterable_By_Mappable (M : MAPPABLE) :
  MAPPABLE_AND_ITERABLE with type 'a t = 'a M.t = struct
  include M

  let iter f x =
    let _ = map f x in
    ()
end

(* Example d’instantiation de module *)
module L = Iterable_By_Mappable (List)
let () = L.iter print_int [1; 2; 3; 4]
```

Par example 2

```ocaml
module type REQUIREMENT_BIND = sig
  type 'a t
  val return : 'a -> 'a t
  val bind : ('a -> 'b t) -> 'a t -> 'b t
end

module type REQUIREMENT_JOIN = sig
  type 'a t
  val return : 'a -> 'a t
  val map : ('a -> 'b) -> 'a t -> 'b t
  val join : 'a t t -> 'a t
end
```

```ocaml
module type API = sig
  type 'a t

  module Api : sig
    include REQUIREMENT_JOIN with type 'a t := 'a t
    include REQUIREMENT_BIND with type 'a t := 'a t

    val void : 'a t -> unit t
  end

  include module type of Api

  module Infix : sig
    val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t
    val ( >|= ) : 'a t -> ('a -> 'b) -> 'b t
    val ( <=< ) : ('b -> 'c t) -> ('a -> 'b t) -> 'a -> 'c t
    val ( >=> ) : ('a -> 'b t) -> ('b -> 'c t) -> 'a -> 'c t
    val ( =<< ) : ('a -> 'b t) -> 'a t -> 'b t
    val ( >> ) : 'a t -> 'b t -> 'b t
  end

  include module type of Infix
end
```

Par example 2

```ocaml
module type REQ = sig
  include Sigs.Monad.REQUIREMENT_BIND
  include Sigs.Monad.REQUIREMENT_JOIN with type 'a t := 'a t
end

module Join (M : Sigs.Monad.REQUIREMENT_JOIN) :
  REQ with type 'a t = 'a M.t = struct
  include M
  let bind f m = join (map f m)
end

module Bind (M : Sigs.Monad.REQUIREMENT_BIND) :
  REQ with type 'a t = 'a M.t = struct
  include M
  let join m = bind id m
  let map f m = bind (return % f) m
end
```

```ocaml
module WithReq (M : REQ) : Sigs.Monad.API
with type 'a t = 'a M.t =
struct
  module Api = struct
    include M
    let ( >>= ) x f = bind f x
    let void _ = return ()
  end

  include Api

  module Infix = struct
    let ( >>= ) x f = M.bind f x
    let ( >|= ) x f = M.map f x
    let ( >> ) m n = m >>= fun _ -> n
    let ( <=< ) f g x = g x >>= f
    let ( >=> ) f g = flip ( <=< ) f g
    let ( =<< ) = M.bind
  end

  include Infix
end
```

Par example 2

```
module Make_with_join (M : Sigs.Monad.REQUIREMENT_JOIN) :
  Sigs.Monad.API with type 'a t = 'a M.t = struct
  include WithReq (Join (M))
end

module Make_with_bind (M : Sigs.Monad.REQUIREMENT_BIND) :
  Sigs.Monad.API with type 'a t = 'a M.t = struct
  include WithReq (Bind (M))
end
```

Par example 2

list.ml

```
module Monad = struct
  include Monad.Make_with_join (struct
    type 'a t = 'a list

    let return x = [x]
    let map = Stdlib.List.map
    let join = Stdlib.List.concat
end)
```

option.ml

```
module Monad = struct
  module M = Monad.Make_with_bind (struct
    type 'a t = 'a option

    let return x = Some x
    let bind f x = match x with
      | None -> None
      | Some x -> f x
end)
```

Résumé des foncteurs

Modules d'ordre supérieur

Modules d’ordre supérieur

Injection de dépendances

```
module type Findable = sig
  type id
  val find_all : unit -> entity list
  val find_by_id : id -> entity option
end

let find_by_id (type a) (module F : Findable with type id = a) (id : a) =
  F.find_by_id id
```

Améliorations futures du langage de modules

Conclusion

- La programmation modulaire à des attraits en ingénierie ;
- OCaml dispose d’un langage de module à part entière ;
- Les foncteurs abstraient facilement beaucoup de comportement ;
- Ils facilitent l’injection de dépendance.

Aller plus loin !

Jouer avec les égalité et les abstractions de types.

Merci !