La programmation modulaire

au-delà des espaces de noms

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

- Je travaille chez Margo Bank, on créée une banque “from scratch” ;
- J’aime bien la programmation (langages fonctionnels statiquement typés)
- Je gère, avec d’autres, le meetup LilleFP: langages applicatifs, ~tous les mois/2 mois
- On recherche des speakers

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 ;

- L’informatique est un domaine rempli de mots qui ont des sens variés.
Par exemple, en Ruby; la notion de “module” n’a pas grand chose à voir avec les modules de Erlang
(ou d’autres langages)

- l’absence de vérification statique facilite beaucoup de choses.

- Quand on aime bien un langage, on veut pousser tout le monde a en faire, parfois de la mauvaise manière, donc j’aimerais faire la promotion d’OCaml, qui est mon langage favoris sans tomber dans l’habituelle présentation (ou on ne parle pas d’objets ni de modules). Même si OCaml possède un système de module incroyablement riche, l’idée est de présenter des concepts potentiellement transposables dans d’autres outils plus mainstream.

Pourquoi cette présentation

Quand on aime un langage et que comme moi, on peut le défendre maladroitement, il arrive qu'avec d'autre zelotte, on complote pour en faire la promotion.
Objets, modules etc.

La présentation ne va pas aussi loin que ce que j'espérais, parce que sinon ça aurait été trop rapide et dur à digérer pour les gens qui ne sont pas familiers avec ça.
(Notamment sur les équivalence de types et la propagation des types)

Modularisation

Compilation séparée

- La modularisation : décomposer un programme en plus petites unités (des modules) qui peuvent êtres compris “malgré leur isolation”, que l’on composera pour faire un programme. Ça rend les programmes plus compréhensibles par les humains

- La compilation séparée : décomposition d’un programme en plus petites unités (des unités de compilation) qui peuvent être typecheckée et compilées séparément par un compilateur. Ça rend les programmes “plus dociles” pour le compilateur.

De nos jours, les langages cumulent généralement ces deux features de manière indissociées via, souvent, les build-systems. Et c’est souvent les “fichiers” qui font office de modules.
Historiquement, Modula-2 offrait la compilation séparée alors que standard ML offrait un langage de modules riche. L’enjeu du système de module de OCaml était de joindre les deux.

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.

- ou chaque développeur travaille sur un module
- Donc ça le rend plus facile à comprendre à et maintenir
- Plus facile à maintenir, à comprendre et généralement plus fiable grâce à l’abstraction de types, qu’on évoquera plus tard
- Ça permet de rendre la compilation plus rapide … parce qu’on ne compile que ce dont on a besoin

En plus de ces bienfaits, ça permet de programmer avec des idiomes assez naïfs mais récurrents comme l’injection de dépendances.

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
```

On suppose la construction d'un module list très minimaliste;
On exposera les fonctions "publiques" dans le MLI et on implémentera, avec la tuyauterie que l'on veut, les fonctions dans le .ML
Par convention on mettra la documentation dans le mli.
Jusque là, rien de très impressionnant … :P

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]
```

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

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

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

On suppose la construction d'un module list très minimaliste;
On exposera les fonctions "publiques" dans le MLI et on implémentera, avec la tuyauterie que l'on veut, les fonctions dans le .ML
Par convention on mettra la documentation dans le mli.
Jusque là, rien de très impressionnant … :P
Ouverture dans le module, ouverture dans le module non destructive, ouverture dans une fonction
Ou ouverture au niveau de "l'expression"

On a des espaces de noms, un peu pauvres.

Mais bon, le titre de la présentation, c’est d’aller plus loin que les espaces noms…

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
```

C'est rigolo mais le mot clé pour créer un sous-module…est module.
On peut créer autant de sous module que l'on veut dans un module et on peut tout à fait joindre la signature de l'implémentation d'un sous-module dans une autre implémentation.

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
```

On peut inclure des modules dans d’autres modules (au contraire de l’import, on injecte les fonctions d’un modules dans un autre modules), ça donne la possibilité d’étendre des modules.

La différence entre “open” et “include”, est que open ouvre, sans rien “importer” et include injecte dans le module courant un autre module.

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
```

On peut créer des signatures qui ne sont pas reliée à “un seul module”.
Le type ‘a t est “abstrait” car on ne connait pas encore “son corps”
On peut lier “MAPPABLE” a n’importe quel module.
La syntaxe “with type…” n’est là que pour garantir que List.t ne soit pas abstrait.

Le langage de module

Actuellement, on a vu que naïvement, les modules offrent :

- Une séparation entre interface et implémentation (et avec le typage, c'est cool pour garantir d'une certaine manière l'implémentation)
- Un système d'espace nom peu expressif
- Un mécanisme d'inclusion et d'extension assez permissif
- La possibilité de mutualiser des interfaces avec des signatures libres

Allons maintenant plus loin !

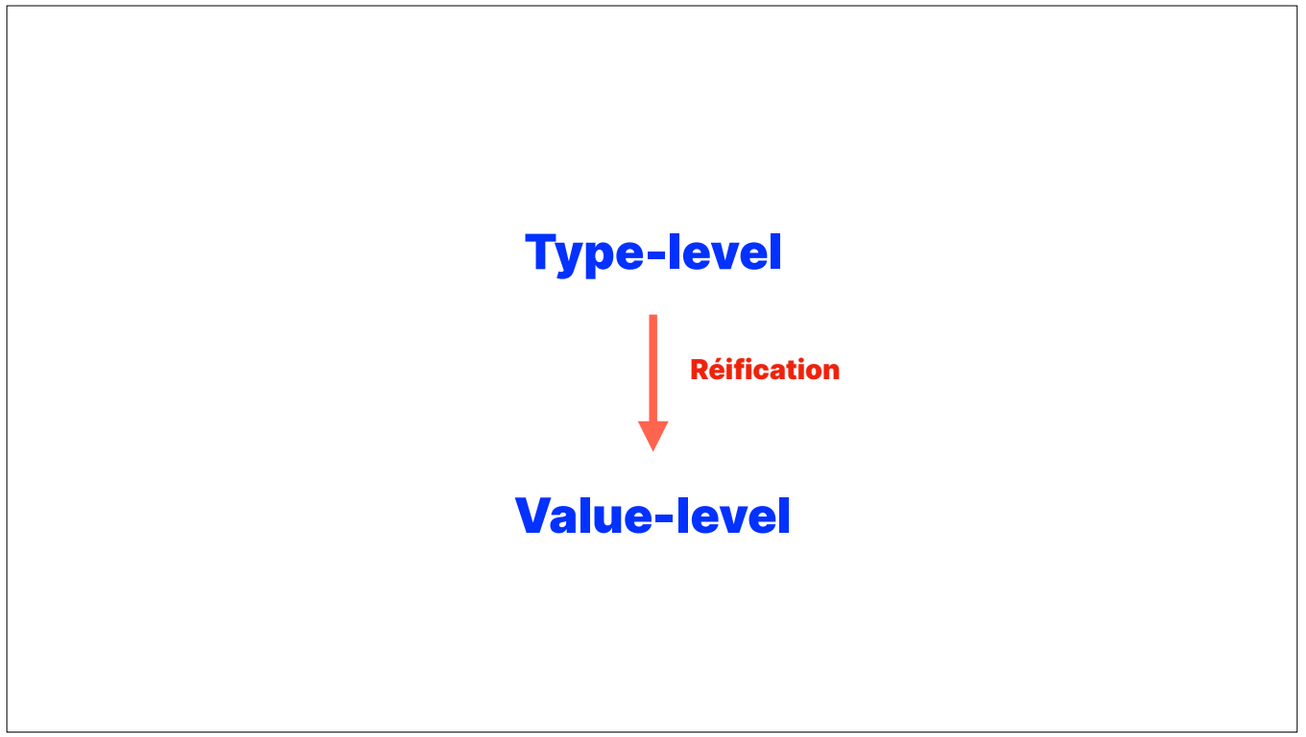
Value-level

Dans n’importe quel type de langages de programmation on manipule des valeurs. On traite des données dans le monde des valeurs.

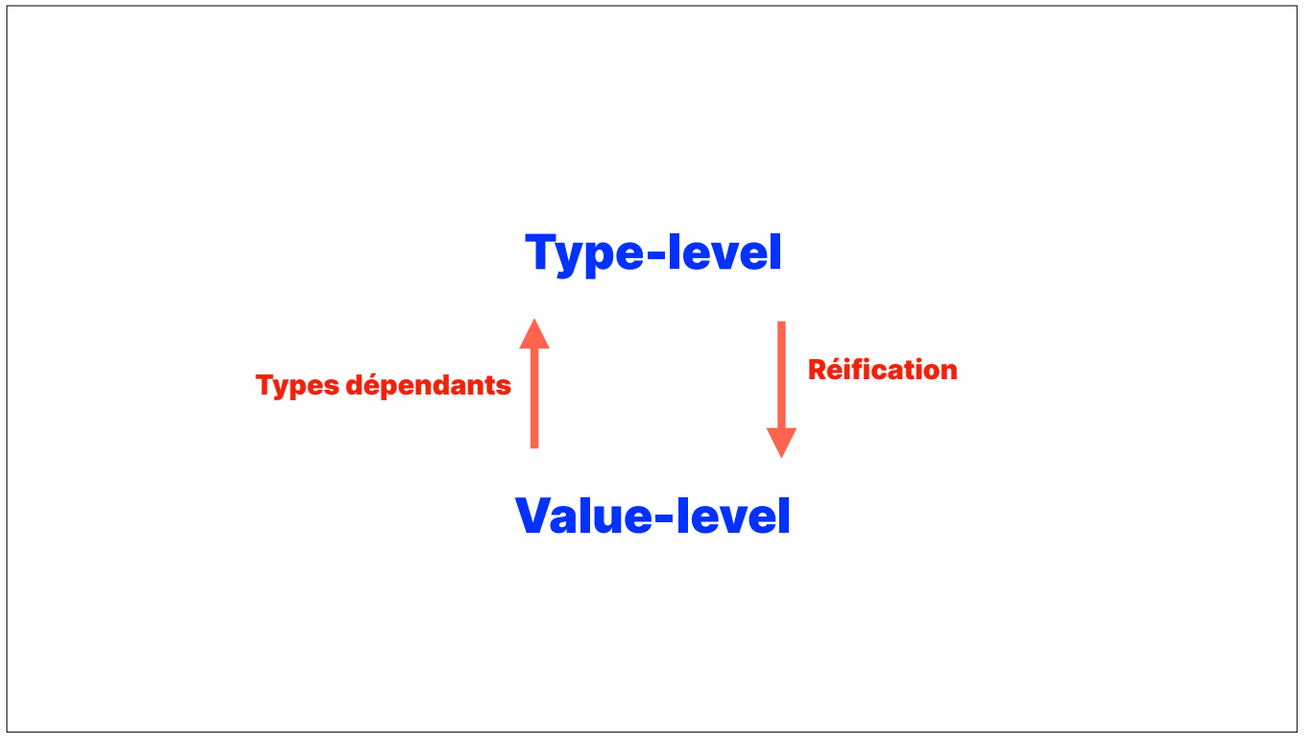
Type-level

Value-level

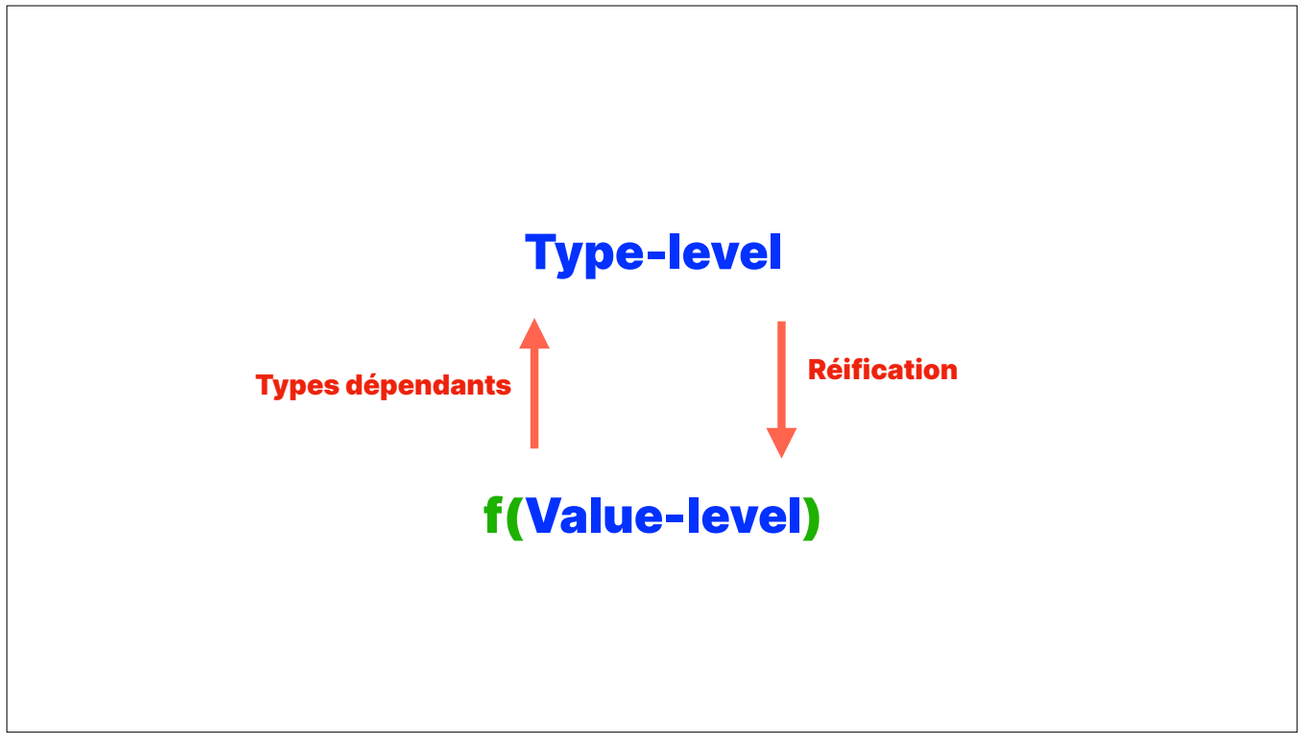
Dans un langage statiquement typé on ajouté un nouveau niveau d’abstraction, le type level.



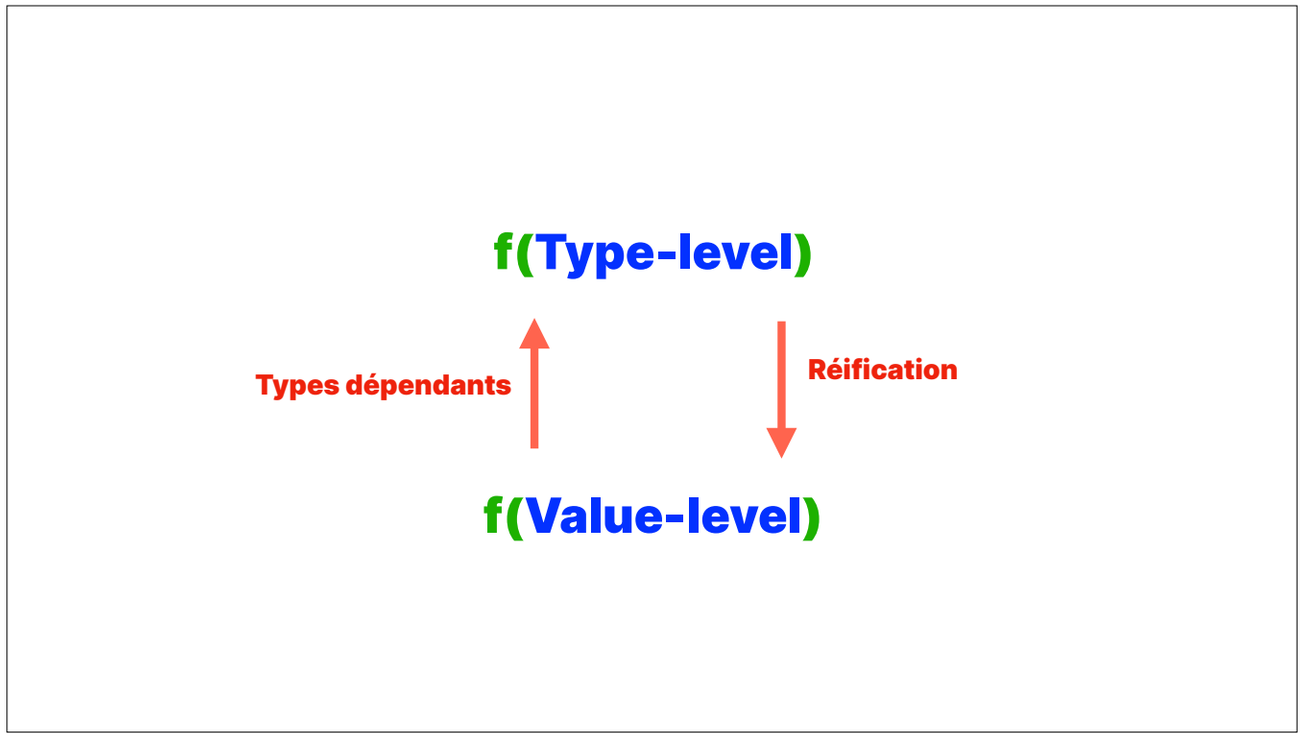
Parfois il arrive que des données du type level entre dans le value level : la réification



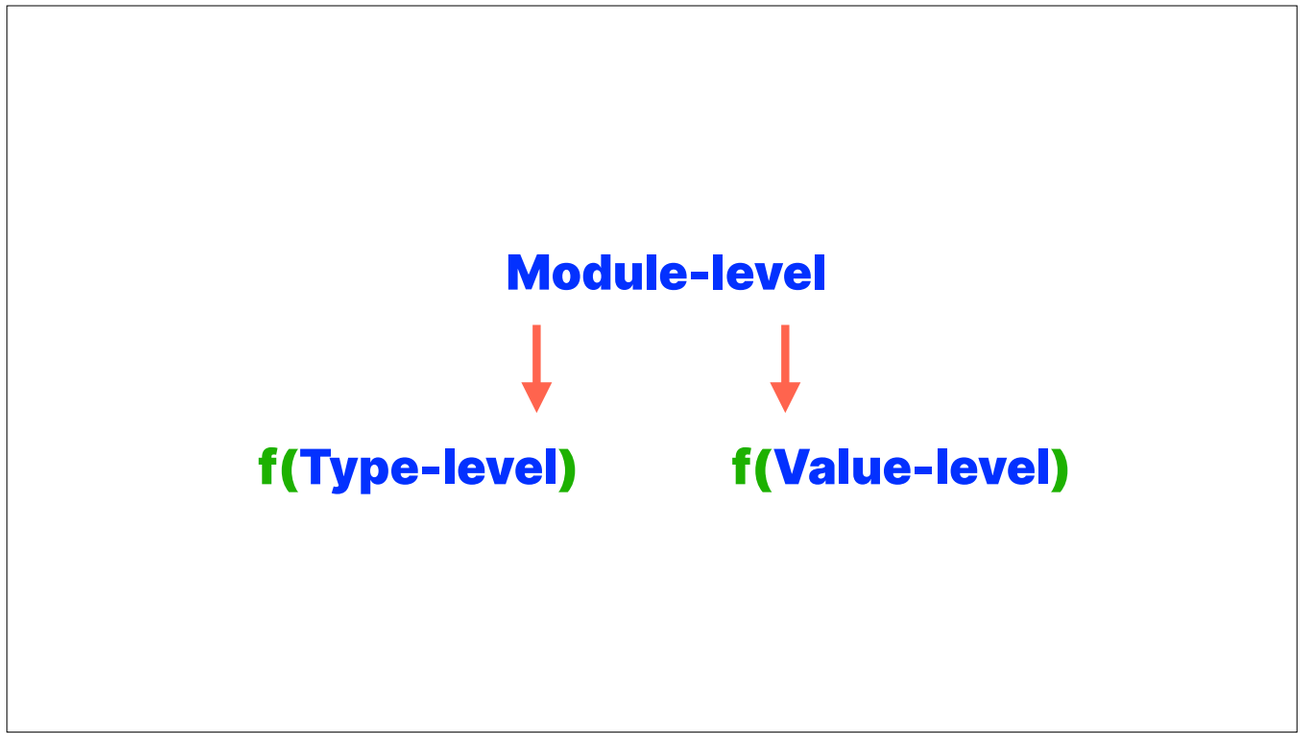
Parfois, il arrive que des valeurs rentrent dans le type level, c’est l’usage des types dépendants (coq, Star, Agda, Idris)



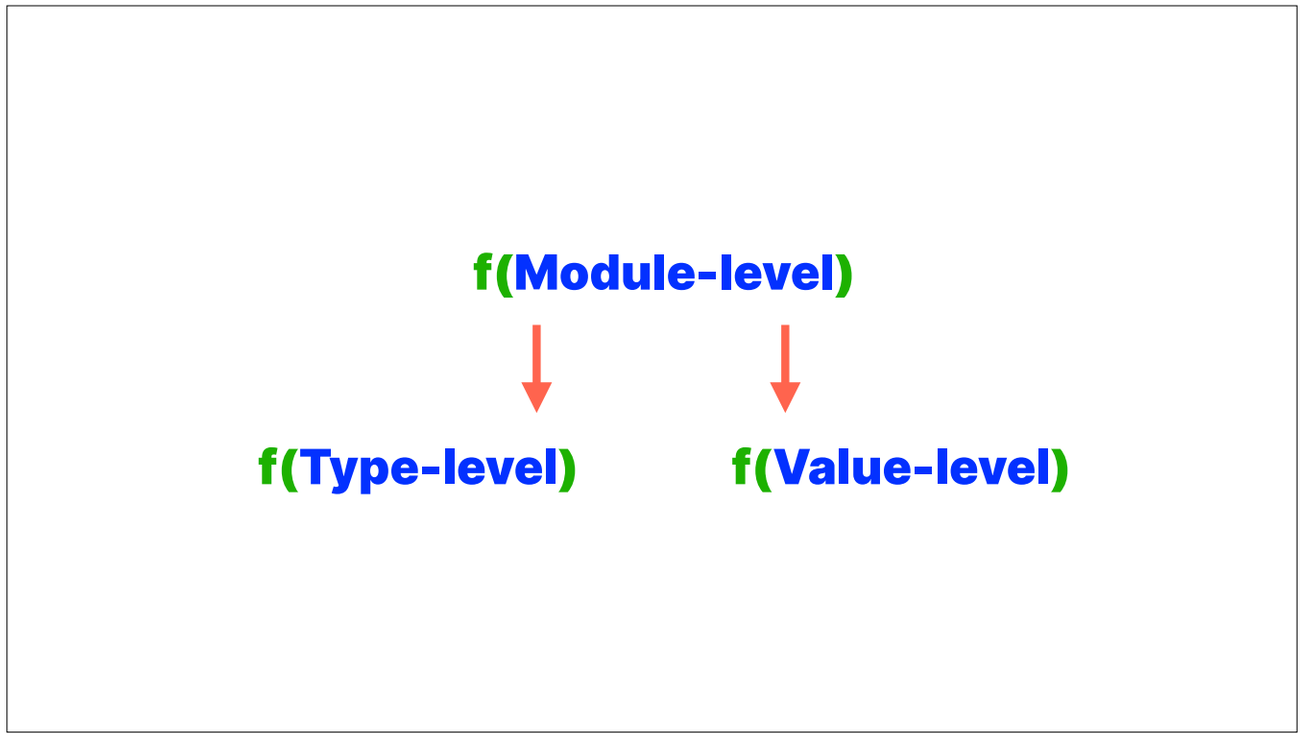
On peut appliquer des fonctions dans le value level



Et dans certains langages avec un système de type plus fin, on peut aussi appliquer des fonctions dans le typelevel (Haskell par exemple, Idris aussi etc)



Qui va pouvoir interagir avec le typelevel et le value level



Et qui offre aussi des fonctions dans le module level… des Foncteurs

Functor, functor, functor...

- Raler sur les mots (foncteur de Haskell, de la théorie catégorie etc)
- Parler de quand on découvre la programmation fonctionnelle
- Essayer de comprendre pourquoi ça s'appelle comme ça (parce qu'en SML ça s'appelait comme ça)
- Expliquer la démonstration post-mortem
-

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.

- ce sont formellement des fonctions dans le module level (exprimable dans un lambda calcul pour les modules)
- Par exemple, on va essayer d’exprimer “iterable” via “mappable”
- En fait, le langage de module est un tout petit langage de programmation fonctionnel à part entière (et dont le système de type est encore plus riche que celui de Caml)

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
```

Rappel sur ce que font map et iter

On voudrait un foncteur de ce type :

MAPPABLE → ITERABLE x MAPPABLE

- ce sont formellement des fonctions dans le module level (exprimable dans un lambda calcul pour les modules)
- Par exemple, on va essayer d’exprimer “iterable” via “mappable”

Par example

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

On spécifie que le type ‘at de ITERABLE est le même que celui de mappable

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]
```

- Expliquer le foncteur et son implem
- Utilisation du module List, qui offre “bien plus de fonctions que just map” mais ça marche parce qu’il faut “au moins respecter l’interface”
- On va rapidement voir un second exemple un peu plus complexe mais sans s’arrêter sur les détails

Par example 2

```
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
```

```
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
```

Imaginons que l’on veuille implémenter une outil que l’on appelerait, au hasard, une monade

Par example 2

```
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
```

```
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
```

Imaginons que l’on veuille implémenter une outil que l’on appelerait, au hasard, une monade

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
```

Imaginons que l’on veuille implémenter une outil que l’on appelerait, au hasard, une monade

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)
```

Pour très peu de boilerplate, je peux promouvoir rapidement l’API de données en, ici, une structure algébrique connue.

Résumé des foncteurs

- Ce sont des fonctions dans le langage de module de OCaml
- On spécifie “un requirement” et on produit un nouveau module
- Ça permet de mutualiser énormément de code
- Ça permet de faire du “backpacking”, qui est une forme d’injection de dépendance
- Il existe une alternative au foncteur applicatifs, qui sont les foncteurs génératifs, qui permettent à chaque instance de générer des “types frais”, donc pour “deux modules avec les mêmes types” on aura une différence structurelle de type.

Modules d’ordre supérieur

- On a dit que les modules étaient des valeurs. Comme les fonctions. On peut donc passer des modules en argument et renvoyer des modules

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
```

- On a dit que les modules étaient des valeurs. Comme les fonctions. On peut donc passer des modules en argument et renvoyer des modules
- C'est tout de même un peu limité car OCaml ne supporte pas de Higher Kinder Types "modulo hack"
- Mais ça mime formellement l'injection de dépendances. On construit un module que l'on passera en argument

Améliorations futures du langage de modules

- Il est déjà très complet, il reste donc “peu de choses” à ajouter.
- Modular “implicit”, implicit de scala “uniquement sur la résolution de modules” pour faire du polymorphisme ad-hoc

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.

- Séparation du travail, de l'exercice de la compilation ce qui rend le code plus facile à maintenir
- Qui possède ses valeurs : des structs, ses types : des sigs et ses fonctions : les foncteurs
- Les fonctions de le module level permettent d'abstraire beaucoup de comportement
- Le fait que les modules soient "first-class citizen" permet de faciliter le backpacking et l'injection de dépendances
- Le langage de module n'est pas arrêté"

Merci !