# Programmation Fonctionnelle Avancée
# Le système de modules

Roberto Di Cosmo

25 septembre 2013



## Plan du cours

- Motivations
- Encapsulation à travers les interfaces, sous-typage simple, visibilité
- Le langage des modules : structures et signatures
- Digression sur la générativité
- Le langage des modules : foncteurs (modules paramétrés)
- Extensions récentes



## La programmation à grande échelle

(*programming in the large*)
Projets de programmation importants :

- $\geq 10^6$ lignes de code
- travail en groupe (la composition du groupe peut changer, tous les membres ne sont pas au même endroit, etc.)
- modifications du programme tout au long de son temps de vie
- conception, programmation, inspection, tests, documentation par des équipes différentes

Il est très difficile de faire cela sans des constructions spécifiques dans le langage du programmation qui nous aident.



## Gérer la compléxité par la décomposition

> *The art of programming is the art of organizing complexity*

(E. Dijkstra)
⇒ Découpage en « morceaux », mais pas n'importe comment :

- Découpage logique correspondant à la logique interne du projet
- Compilation séparée [1]
- Faciliter la maintenance
- Faciliter les extensions du programme
- Réutilisation du code (bibliothèques)
- On veut aussi, autant que possible, *écrire moins de code !* (on y reviendra)

1. La compilation de OpenOffice prend plus de 24h

## Fichiers, packages, ... modules !

C/C++ on utilise les fichiers avec leurs interfaces .h, ...
*impossible de réutiliser le même nom dans deux fichiers*

Java les *packages* organisent les définitions de classes et objets dans un espace de nommage hierarchique

OCaml les *modules* organisent les définitions de types, valeurs et exceptions dans un espace de nommage hierarchique ;
il est aussi possible de les paramétrer par rapport à d'autres unités

Modula-2, Ada, ... avaient les modules depuis longtemps ; le système de modules d'OCaml est l'un des plus puissants et aboutis.



## Définition (informelle)

### Module

Unité de programme qui regroupe un ensemble de *définitions* du langage.

- On peut faire référence à un module par un nom
- Un module *peut être* identifié à un fichier.
- Un module *exporte* certaines de ces définitions, et en *importe* d'autres.
- Certaines définitions d'un module peuvent être cachées, en tout ou en partie : c'est important pour l'*encapsulation* et l'*abstraction*.



## Exemples

petit l'implémentation d'une file ou une pile (module `Stack`)

moyen l'implémentation de l'interface avec le système Unix (module `Unix`)

grand l'interface avec la librairie GTK (collections de modules `LablGTK`)



## Modules, structures et signatures

Un exemple simple, regrouper le code écrit pour un compteur :

```
module Counter =
  struct
    let c = ref 0
    let incr () = c:= !c+1
    let show () = !c
  end
;;
|| module Counter :
||   sig val c : int ref val incr : unit -> unit val show : unit -> int end
||
```

Notions importantes :

nom du module : cela *commence par un majuscule* et il est déclaré

**module** Nomdemodule = ...

structure : code regroupé entre `struct` et `end`
toute construction du langage (y compris des modules)

signature : l'interface, délimitée par `sig` et `end`

## Modules, structures et signatures

Une *signature* peut être inférée automatiquement, comme dans l'exemple précedent.
Mais elle peut aussi être définie explicitement :

```
module type CounterFullItf =
  sig
    val c : int ref
    val incr : unit -> unit
    val show : unit -> int
  end;;
|| module type CounterFullItf =
||   sig val c : int ref val incr : unit -> unit val show : unit -> int end
||
```

et utilisée pour déclarer l'interface d'un module

```
module Counter : CounterFullItf =
  struct
    let c = ref 0
    let incr () = c:= !c+1
    let show () = !c
  end;;
|| module Counter : CounterFullItf
```



## Modules, structures et signatures

On peut aussi écrire :

```
module Counter =
  (struct
    let c = ref 0
    let incr () = c:= !c+1
    let show () = !c
   end : CounterFullItf);;
|| module Counter : CounterFullItf
```

N.B. Les parenthèses sont obligatoires : même notation que les annotations de type habituelles comme

```
let x = (3 : int);;
|| val x : int = 3
```



Pour accéder à un élément exporté par un module, on peut *qualifier son identificateur avec le nom du module* :

```
Counter.incr();;
|| - : unit = ()

Counter.show();;
|| - : int = 1
```

Avantage : le code est explicite, on sait de quoi on parle

Désavantage : peut dévenir très verbeux, donc on a aussi une autre solution...



## Ajouter le module dans la portée : la directive open

```
open Counter;;

show();;
|| - : int = 1
```

N.B. : du *dernier* module ouvert jusqu'à l'environnement standard

```
let x = 3;;
|| val x : int = 3

module A = struct let x = 3.14 end;;
|| module A : sig val x : float end

module B = struct let x = "a" end;;
|| module B : sig val x : string end

x;;
|| - : int = 3

open A;;

x;;
|| - : float = 3.14

open B;;

x;;
|| - : string = "a"
```

## Cacher des parties d'un module

La *signature* inférée automatiquement contient *tous* de détails de la structure qui l'implémente.
Dans le cas de notre `Compteur`, on peut voir la variable `c`, et *l'altérer* ! Cela peut très bien arriver involontairement, si vous avez une autre variable `c` dans le programme.

```
open Counter;;

incr();;
|| - : unit = ()

(* this should not be allowed! *)
c := !c + 32;;
|| - : unit = ()

show();;
|| - : int = 34
```

On a besoin d'empecher cet accès aux détails d'implémentation : il nous faut un méchanisme d'*encapsulation*.



## Le principe d'encapsulation

Exporter aussi peu de définitions que possible : l'interface d'un module peut être *plus restrainte* que son corps ; on cache des fonctions, types, exceptions auxiliaires.
En OCaml :

- le corps peut contenir des types, fonctions, exceptions *privés* (pas exportés)
- Si un type *concret* est exporté par l'interface, alors le corps *doit* contenir la même définition



## Utiliser une signature pour l'encapsulation

Dans notre exemple, on peut définir cette signature, qui est plus restrictive

```
module type CounterItf =
  sig
    val incr : unit -> unit
    val show : unit -> int
  end;;
|| module type CounterItf =
||   sig val incr : unit -> unit val show : unit -> int end
```

pour cacher une partie de l'information d'un module existant :

```
module CounterHide = (Counter : CounterItf);;
|| module CounterHide : CounterItf
```



## Utiliser une signature pour l'encapsulation

On peut aussi l'utiliser pour restreindre la signature inférée par le système :

```
module Counter : CounterItf =
  struct
    let c = ref 0
    let incr () = c:= !c+1
    let show () = !c
  end;;
|| module Counter : CounterItf
```

cela permet d'avoir de *cacher quelques valeurs de la structure*, qui restent donc encapsulés dans le module, et inaccessibles depuis l'extérieur.

## Le cas des valeurs et exceptions

Tout identificateur (ou exception) exporté doit être défini par le corps, *et cela avec un type égal ou plus général* :

```
module A = struct let id x = x end;;
|| module A : sig val id : 'a -> 'a end

A.id;;
|| - : 'a -> 'a = <fun>

module type Aint = sig val id : int -> int end;;
|| module type Aint = sig val id : int -> int end

module Aint = (A:Aint);;
|| module Aint : Aint

Aint.id;;
|| - : int -> int = <fun>
```



## Le cas des valeurs et exceptions

```
(* le contraire ne passe pas *)

module Bint = struct let id x:int = x end;;
|| module Bint : sig val id : int -> int end

module type B = sig val id : 'a -> 'a end;;
|| module type B = sig val id : 'a -> 'a end

module B = (Bint:B);;
|| Characters 12-16:
||   module B = (Bint:B);;
||              ^^^^
|| Error: Signature mismatch:
||        Modules do not match: sig val id : int -> int end is not included
||  in B
||        Values do not match:
||          val id : int -> int
||        is not included in
||          val id : 'a -> 'a
```



## Le principe d'abstraction

On souhaite pouvoir restraindre l'accés à un type de données défini dans un module, même quand ce type ne peut pas être encapsulé dans le module.
Pour cela, l'interface d'un module peut être *plus abstraite* que son corps :

- une interface peut exporter un type *abstrait* : contient la déclaration `type t`, et le corps contient sa définition complète `type t = ...`

Comme la définition n'est accessible qu'à l'intérieur du module, le seul moyen de manipuler des valeurs de ce type est d'appeler des fonctions du module.



## Une interface plus realiste

```
module type MultiCounter =
   sig
    type t
    val create : unit -> t
    val incr : t -> unit
    val show : t -> int
   end;;
|| module type MultiCounter =
||   sig
||     type t
||     val create : unit -> t
||     val incr : t -> unit
||     val show : t -> int
||   end
```

```
module MultiCounter: MultiCounter =
 struct
   type t = int ref
   let create () = ref 0
   let incr c = c:= !c+1
   let show c = !c
 end
;;
||
module MultiCounter : MultiCounter
```

maintenant le compteur est une valeur *abstraite*

```
open MultiCounter;;

let a = create();;
|| val a : MultiCounter.t = <abstr>
```

Ici, on a *caché la définition* du type `t`, pas sa présence.

## En résumé

| Genre | Module | Signature |
|---|---|---|
| public | `type t = ...` | `type t = ...` |
| privé/encapsulé | `type t = ...` | – |
| abstrait | `type t = ...` | `type t` |

Reste à comprendre quelle relation existe entre différentes définition de types.



## Quand considérer équivalents deux types ?

Première approche : *équivalence structurelle*
`t1` et `t2` sont équivalents si leur *structure* est identique.
Si on fait ce choix, le programme suivant est bien typé :

```
type person = {name:string, age:int} in
let p = {name="Nobody", age=1000} in
type employee = {name:string, age:int} in
let e = {name="Somebody", age=2000} in
p = e
```

Cela demande un effort considérable au compilateur, en particulier si on permet des types récursifs (on sait en décider l'équivalence, mais cela sort du cadre de ce cours)



## Quand considérer équivalents deux types ?

Deuxième approche : *types génératifs*

- *Tous* les types sont distincts, même s'ils ont la même structure.
- Chaque nouvelle définition d'un type utilisateur est distinguée de toutes les précédentes (on associe à chaque définition de type une valeur unique, un "time-stamp", qui permettra de le distinguer facilement et rapidement des autres).
- On parle de types *génératifs*, parce-que chaque déclaration de type produit (génére) une nouvelle valeur unique.



## Le choix fait dans divers langages

| Langage | types génératifs | Notes |
|---|---|---|
| Ocaml | oui | les modules sont un cas spécial |
| C/C++ | | oui pour structures, union, tableaux |
| | | non pour le reste |
| Pascal | oui | sauf pour SET |
| Ada | oui | |
| Java | oui | |
| Algol 68 | non | le langage non génératif le plus complexe |
| Modula-3 | | génératif sur les types abstrait, |
| | | structurel sur les types concrets |

En OCaml : deux types abstraits ayant la même implémentation sont incompatibles

On peut vouloir donner des visions différentes d'un même module ...

```
module M =
(struct
  type t = int ref
  let create() = ref 0
  let step x = x:=!x + 1
  let get x = if !x>0 then (decr x; 1) else failwith "Empty"
 end :
 sig
  type t
  val create : unit -> t
  val step : t -> unit
  val get : t -> int
 end
)  ;;

 module type Write = sig
     type t
     val create : unit -> t
     val step : t -> unit
   end ;;

 module type Read = sig type t val get : t -> int end ;;
```



... mais l'abstraction nous empêche de l'utiliser

```
module Mwrite = (M:Write) ;;
|| module Mwrite : Write

module Mread = (M:Read) ;;
|| module Mread : Read


let counter=Mwrite.create();;
|| val counter : Mwrite.t = <abstr>

Mwrite.step counter;;
|| - : unit = ()


Mread.get counter;;
|| Characters 10-17:
||   Mread.get counter;;
||           ^^^^^^^
|| Error: This expression has type Mwrite.t
||        but an expression was expected of type Mread.t
```



On peut s'en sortir de façon élégante avec des *contraintes de partage* :

```
module Mwrite = (M:Write with type t = M.t) ;;
|| module Mwrite :
||   sig type t = M.t val create : unit -> t val step : t -> unit end

module Mread = (M:Read with type t = M.t) ;;
|| module Mread : sig type t = M.t val get : t -> int end

let counter = Mwrite.create();;
|| val counter : Mwrite.t = <abstr>

Mwrite.step counter;;
|| - : unit = ()

Mread.get counter;;
|| - : int = 1
```

`module Nom1 = (Nom2 : SIG with type t1 = t2 and ...)`

- on vérifie que le type `t1` et `t2` sont *compatibles*
- on garde l'information que `t1 = t2`, *même s'ils sont abstraits*



## L'abstraction empêche le pattern matching

```
module type NEL = sig
  type 'a t
  val cons : 'a -> 'a t -> 'a t
  val pop : 'a t -> 'a * 'a t option
  val create : 'a -> 'a t
end
module NonEmptyList : NEL = struct
  type 'a t = Nil | Cons of 'a * 'a t
  let cons x l = Cons(x,l)
  let pop =
     function
       (Cons (x,Cons(y,l))) -> x, Some (Cons(y,l))
     | Cons(x,Nil) -> x, None
     | _ -> assert false
  let create x = Cons(x,Nil)
end;;
```

\+ : On ne peut créer une liste vide

\- : On n'a plus accès au pattern matching : essayez d'écrire `map` !

## Un compromis : les types privés

```
module type NELprivate = sig
  type 'a t = private Nil | Cons of 'a * 'a t
  val cons : 'a -> 'a t -> 'a t
  val pop : 'a t -> 'a * 'a t option
  val create : 'a -> 'a t
end
module NonEmptyListPriv : NELprivate = struct
  type 'a t = Nil | Cons of 'a * 'a t
  let cons x l = Cons(x,l)
  let pop =
       function
         (Cons (x,Cons(y,l))) -> x, Some (Cons(y,l))
       | Cons(x,Nil) -> x, None
       | _ -> assert false
  let create x = Cons(x,Nil)
end;;
|| module type NELprivate =
||   sig
||     type 'a t = private Nil | Cons of 'a * 'a t
||     val cons : 'a -> 'a t -> 'a t
||     val pop : 'a t -> 'a * 'a t option
||     val create : 'a -> 'a t
||   end
|| module NonEmptyListPriv : NELprivate
```



## Un compromis : les types privés

Le type qui implémente `'a t` est exposé *en lecture seule* :

- on peut utiliser le pattern matching
- seulement les fonctions du même module peuvent créer des valeurs de ce type

```
open NonEmptyListPriv;;

(* on peut faire une definition par cas *)
let rec map f =
   function
     (Cons (x,l)) -> cons (f x) (map f l)
   | _ -> assert false;;
|| val map : ('a -> 'b) -> 'a NonEmptyListPriv.t -> 'b NonEmptyListPriv.t =
||
||   <fun>

(* mais il est impossible de violer les invariants en construisant directement des list
cons 3 Nil;;
|| Characters 100-103:
||   cons 3 Nil;;
||          ^^^
|| Error: Cannot create values of the private type int NonEmptyListPriv.t
```



## Pointeurs pour aller plus loin

Le conflit entre abstraction et définition par cas (pattern matching) a été l'objet d'attention en literature depuis 1987 :


P. Wadler.
Views : A way for pattern matching to cohabit with data abstraction.
In POPL, pages 307–313, 1987.



### Pas de cycles de dependences

- On dit que *B depend de A* si le module B utilise une définition exportée par le module A.
- En OCaml, si le module B depend du module A, alors A *doit être défini avant* B, donc : *le graphe des dependence des modules doit être acyclique*.
-

### *Extension :* Modules recursifs

Une extension expérimentale permet des définitions *recursives* de modules, mais son usage est délicat. On conseille de se tenir à la règle du graphe acyclique.

Les fichiers sources OCaml sont traités comme des *modules* :

nom.ml est compilé comme le *module*

```
module Nom = struct
  contenu du fichier nom.ml
end
```

nom.mli est compilé comme la *signature*

```
sig
  contenu du fichier nom.mli
end
```

et utilisé uniquement pour restreindre la signature du module issu du source .ml avec le même nom.

La contrainte d'acyclicité des dependences des modules devient une contrainte sur *l'ordre de compilation* des fichiers source.



## La compilation séparée

Les fichiers sources et les interfaces peuvent être compilés séparement, en respectant l'ordre imposé par les dependences.

### Quelques outils

ocamldep calcule les dependences d'un fichier .ml ou .mli

ocamldsort calcule un ordre d'édition de liens compatible avec les dependences entre fichiers (modules)

make outil standard qui peut servir à compiler un projet, si on connait les dependences

ocamlbuild essaye de compiler un projet OCaml en découvrant les dependences tout seul



## Le bésoin de paramétrer I

Considérons un module qui fournit des piles triées, sa signature peut être

```
module type Stack = sig
 exception Empty
 type elt
 type t
 val push : elt -> t -> t
 val pop : t -> elt * t
 val is_empty : t -> bool
 val empty : unit -> t
end;;
```

Une possible implémentation, pour les piles d'entiers, est



## Le bésoin de paramétrer II

```
module OrderedIntStack : Stack with type elt = int = stru
  exception Empty
  type elt = int
  type t = int list
  let rec push x = function
      [] -> x::[]
    | h::t as l when x < h -> x::l
    | h::t -> h::push x t
  let pop = function
      [] -> raise Empty
    | h::t -> (h,t)
  let is_empty s = s = []
  let empty () = []
end;;
```

## Le bésoin de paramétrer III

Question : comment peut-on rendre le module paramétrique par rapport au type `elt` ?



## Prémière étape : identifier le paramètre I

Une première étape dans la bonne direction est de recueillir dans un module séparé les informations rélatives au type `elt`.

```
module type Comparable = sig
 type elt
 val compare : elt -> elt -> int
end;;
```

Ensuite, on crée un module avec le type qui nous intéresse

```
module T:Comparable with type elt=int = struct
  type elt = int
  let compare x y = x-y
end;;
```

Et on écrit le module pile ordonnée en utilisant `T`.



## Prémière étape : identifier le paramètre II

```
module OrderedTStack : Stack with type elt = T.elt = struct
  exception Empty
  type elt = T.elt
  type t = elt list
  let rec push x = function
      [] -> x::[]
    | h::t as l when T.compare x h < 0 -> x::l
    | h::t -> h::push x t
  let pop = function
      [] -> raise Empty
    | h::t -> (h,t)
  let is_empty s = s = []
  let empty () = []
end;;
```



## Prémière étape : identifier le paramètre III

Ensuite, si on decide de changer le type, on pourra simplement éditer le fichier et changer la définition du module `T`.

### Problème

Mais si on veut avoir de piles ordonnées de types différents, on devrait dupliquer le code, et on ne veut pas faire ça !

## Deuxième étape : expliciter le paramètre I

```
module OrderedStack (T: Comparable) :
       Stack with type elt = T.elt = struct
  exception Empty
  type elt = T.elt
  type t = elt list
  let rec push x = function
      [] -> x::[]
    | h::t as l when T.compare x h < 0 -> x::l
    | h::t -> h::push x t
  let pop = function
      [] -> raise Empty
    | h::t -> (h,t)
  let is_empty s = s = []
  let empty () = []
```



## Deuxième étape : expliciter le paramètre II

```
end;;

(* instantiation et utilisation du module *)

module Int =
   struct type elt = int let compare x y = x-y end;;

module OrderedIntStack = OrderedStack (Int);;


(* mais on peut ecrire aussi *)

module OrderedIntStack =
  OrderedStack (struct type elt = int
                       let compare x y = x-y
```



## Deuxième étape : expliciter le paramètre III

```
                end);;

open OrderedIntStack;;


push 3 (push 9 (push 7 (empty())));;
|| - : OrderedIntStack.t = <abstr>
```

Attention La contrainte de type `with type` est essentielle !
Pourquoi ? Question : pourquoi le type des éléments n'est pas abstrait ?



## Variantes

On peut écrire tout aussi bien

```
module OrderedStack (T: Comparable) : Stack =
  struct
```

que

```
module OrderedStack :
  functor (T: Comparable) -> Stack =
  functor (T: Comparable) -> struct
```

Terminologie On utilise de façon équivalente les termes *foncteur* ou *module paramétrique*.

## Plusieurs arguments

On peut expliciter autant de paramètres qu'on le souhaite

```
module M (T1: Comparable)
         (T2: sig type t val x: t end
              with type t = T1.t) ... =
  struct
```



## Usage typique des foncteurs I

On trouve plusieurs foncteurs dans la librairie standard de OCaml, et on peut observer qu'ils ont une structure commune :

```
(* la signature du parametre: *)
module type Content = sig
  type t
  ...
end

(* Signature de sortie du foncteur: *)
module type Collection = sig
  type content
  type t
  ...
end
```



## Usage typique des foncteurs II

```
(* Le foncteur: *)
module Make :
  (functor (T:Content) ->
      Collection with type content = T.t) =
functor (T:Content) -> struct
  type content = T.t
  type t = ...
  ...
end
```



## Suggestion

Explorez les interfaces des modules `Map`, `Set` et `Hashtbl`.

## Quelques exemples significatifs

Un foncteur construit un module à partir d'un (ou plusieurs) autre(s) module(s).
Si les interfaces sont bien étudiées, cela permet d'assembler des programmes très variés à partir de quelques composants modulaires. Quelques exemples, qu'on discute en cours :

- L'interpète extensible Lua-ML `http://www.cs.tufts.edu/~nr/pubs/maniaws.pdf`
- Les piles de protocoles `http://www.cs.cmu.edu/Groups/fox/papers/lfp-signatures.ps`
- Des polynômes en plusieurs variables à partir de polynomes en une variable `http://cristal.inria.fr/~remy/poly/ocaml/td/TD-2b/index.html`



## Inclusion de structures et signatures

Permet d'ajouter facilement des fonctionnalités à un module

```
module Counter2 =
struct
  include Counter
  let step n = c:=!c+n
  let incr () = Printf.eprintf "Inside␣Counter2\\n%!"; s
end;;
|| module Counter2 :
||   sig
||     val show : unit -> int
||     val step : int -> unit
||     val incr : unit -> unit
||   end
```

Attention : `Counter2` partage le compteur `c` avec `Counter` !



## *module type of* retrouver la signature d'un module

Usage typique : pour un `include` dans une signature

```
module type Counter2 =
sig
  val step : int -> unit
  include module type of Counter
end;;
|| module type Counter2 =
||   sig
||     val step : int -> unit
||     val incr : unit -> unit
||     val show : unit -> int
||   end
```



## *(module M :S)* et *(val e :S)* modules de premiere classe

```
let c1 = (module Counter: CounterItf)
and c2 = (module Counter2: CounterItf);;
|| val c1 : (module CounterItf) = <module>
|| val c2 : (module CounterItf) = <module>

let compteur sel =
     let module C =
        (val (if sel
              then c1
              else c2) : CounterItf)
     in (C.incr,C.show)
;;
|| val compteur : bool -> (unit -> unit) * (unit -> int) =
```

## Pointeurs pour aller plus loin

L'article fondateur du système de modules de OCaml est

Xavier Leroy.
A modular module system.
J. Funct. Program., 10 :269–303, May 2000.

Il ne s'agit pas d'un travail facile : l'implémentation F# de OCaml ne l'inclut pas...