

Programmation Fonctionnelle Avancée

Le système de modules

Pierre Letouzey

Université de Paris
UFR Informatique
Institut de Recherche en Informatique Fondamentale
`letouzey@irif.fr`

26 janvier 2022

Gérer la complexité par la décomposition

Une bonne pratique du Génie Logiciel :

- ▶ Découpage logique correspondant à la logique interne du projet
- ▶ Compilation séparée
- ▶ Faciliter la maintenance
- ▶ Faciliter les extensions du programme
- ▶ Réutilisation du code (bibliothèques)
- ▶ On veut aussi, autant que possible, *écrire moins de code* : factorisation !
- ▶ Il est très difficile de faire cela sans des constructions spécifiques dans le langage du programmation qui nous aident.

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 hiérarchique

OCaml les *modules* organisent les définitions de types, valeurs et exceptions dans un espace de nommage hiérarchique ;
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 son nom
- ▶ Un module *peut être* identifié à un fichier (vu en L3)
- ▶ 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*.

Modules, structures et signatures en OCaml

Notions importantes :

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

module Nomdemodule = ...

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

Exemples (counter1.ml)

```
(* Regrouper le code écrit pour un compteur *)
module Counter =
  struct
    let c = ref 0
    let incr () = c := !c+1
    let show () = !c
  end
;;
```

Modules, structures et signatures

- ▶ Une *signature* peut être inférée automatiquement, comme dans l'exemple précédent.
- ▶ Mais elle peut aussi être définie explicitement et utilisée pour déclarer l'interface d'un module.
- ▶ **signature** : l'interface, délimitée par les mots clefs **sig** et **end**
- ▶ On voit souvent des noms de signature écrits tout en majuscules, mais c'est seulement une convention (rien ne vous oblige de faire pareil).

Exemples (counter2.ml)

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

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


Exemples (counter3.ml)

(syntaxe alternative : signature apres struct ...*

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

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

```
Counter.incr();;  
Counter.show();;
```

- Avantage : le code est explicite, on sait de quoi on parle
- Inconvénient : peut devenir très verbeux, donc on a aussi une autre solution :

```
open Counter
```

- Avantage : plus concis
- Inconvénient : difficile à contrôler quand plusieurs modules exportent le même nom.

Exemples (open1.ml)

```
(* open global *)
```

```
let x = 3;;
```

```
module A = struct let x = 3.14 end;;
```

```
module B = struct let x = "a" end;;
```

```
x;;
```

```
open A;;
```

```
x;;
```

```
open B;;
```

```
x;;
```

```
(* le dernier module ouvert a la priorite *)
```

Exemples (open2.ml)

```
(* open local , OCaml >= 3.12 *)
```

```
module A=struct let a=17 let b=42 end
```

```
let a = 1;;
```

```
let b = 2;;
```

```
a+b;;
```

```
let open A in a+b;;
```

```
(* open restricted to expression after in *)
```

```
a+b;;
```

Exemples (open3.ml)

```
(* open local , OCaml >= 3.12 *)
```

```
module A=struct let a=17 let b=42 end
```

```
let a = 1
```

```
let b = 2;;
```

```
a+b;;
```

```
A.(a+b);;
```

```
(* A open locally in expression between ( ) *)
```

```
a+b;;
```

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 la *modifier*! Cela peut très bien arriver involontairement, si vous avez une autre variable *c* dans le programme.

```
open Counter;;
incr();;
(* this should not be allowed! *)
c := !c + 32;;
show();;
```

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

Le principe d'encapsulation

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

En OCaml :

- ▶ Le corps peut contenir des types, fonctions, exceptions *locaux* (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
- ▶ l'utiliser pour cacher une partie de l'information d'un module existant.

Exemples (encapsulation1.ml)

```

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

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

module CounterHide = (Counter : CounterItf);;

```

Exemples (encapsulation2.ml)

```
(* equivalent *)
module type CounterItf =
  sig
    val incr : unit -> unit
    val show : unit -> int
  end;;
module CounterHide : CounterItf =
  struct
    let c = ref 0
    let incr () = c := !c+1
    let show () = !c
  end;;
CounterHide.c;;
CounterHide.show() ;;
```

Le cas des valeurs et exceptions : encapsulation

- ▶ Tout identificateur (ou exception) exporté doit être défini par le corps du module, *et cela avec un type égal ou plus général* que le type donné dans l'interface.
- ▶ En imposant une signature on peut seulement (cas des valeurs et exceptions) :
 - ▶ cacher une valeur ou exception,
 - ▶ faire un type plus spécifique.
- ▶ Analogue aux contraintes de type explicites vues la semaine dernière.

Exemples (implementation.ml)

```
(* corps plus general que l'interface *)
module A = struct let id x = x end;;
A.id;;
module type AintSig = sig val id : int -> int end;;
module Aint = (A:AintSig);;
Aint.id;;
```

```
(* interface plus general que le corps *)
module Bint = struct let id x:int = x end;;
module type BintSig = sig val id : 'a -> 'a end;;
module B = (Bint:BintSig);;
```

Le cas des types : encapsulation et abstraction

- ▶ On souhaite pouvoir restreindre l'accès 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.
- ▶ On cache l'implémentation du type, pas son existence !

Exemples (counter4.ml)

```
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
```

Exemples (counter5.ml)

```
let a = MultiCounter.create();;
MultiCounter.incr a;;
MultiCounter.show a;;
```

```
(* not authorized due to typing *)
a := !a+1;;
```

En résumé

Genre	Module	Signature
public	<code>type t = ...</code>	<code>type t = ...</code>
local/encapsulé	<code>type t = ...</code>	—
abstrait	<code>type t = ...</code>	<code>type t</code>

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

Quand considérer équivalents deux types ?

Première approche (hypothétique) : *équivalence structurelle*

t1 et t2 sont équivalents si leur *structure* est identique.

Si on faisait ce choix, le programme suivant serait bien typé :

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

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

Exemples (types1.ml)

```
type person = {name:string; age:int}  
let p = {name="Nobody"; age=1000};;
```

```
type employee = {name:string; age:int}  
let e = {name="Somebody"; age=2000};;
```

```
p = e
```

Quand considérer équivalents deux types ?

Deuxième approche (adoptée par OCaml) : *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 (OCaml 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*, car chaque déclaration de type produit (génère) une nouvelle valeur unique.

Le choix fait dans divers langages

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

Exemples (types2.ml)

```
type pair = P of int*int
let p = P (1,2);;
```

```
type pair = P of int*int
let c = P (3,4);;
```

```
p = c;; (* types différents *)
```

```
type duo = pair;;
let d = (P (5,6):duo);; (* contrainte de type *)
```

```
c = d;; (* meme type *)
```

Equivalence de types et abstraction

- ▶ En OCaml : deux types abstraits ayant la même implémentation sont incompatibles.
- ▶ Les interfaces sont opaques (si on choisit d'exporter seulement un type abstrait).
- ▶ Choix différent en SML (une autre implémentation de ML avec aussi un système de modules).
- ▶ On verra plus tard comment OCaml résoud ce problème.

```

module type T =
  sig
    type t (* abstrait *)
    val c : t
  end;;

```

```

module M1 : T =
  struct
    type t = int
    let c = 42
  end;;

```

```

module M2 : T =
  struct
    type t = int
    let c = 17
  end;;

```

```

M1.c = M2.c;; (* types différents *)

```

Plusieurs interfaces du même module

- ▶ Un autre exemple où on veut avoir une égalité entre types abstraits :
- ▶ On peut vouloir donner des visions différentes d'un même module, par exemple une interface qui permet de construire/modifier des valeurs, et une autre qui permet seulement de les lire.
- ▶ Mais on veut aussi utiliser un type abstrait pour protéger l'accès direct à des valeurs.
- ▶ Problème : l'abstraction nous empêche de l'utiliser !

Exemples (sharing1.ml)

```

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

```

Exemples (sharing2.ml)

```
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 ;;
```

Exemples (sharing3.ml)

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

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

```
let counter = Mwrite.create();;
```

```
Mwrite.step counter;;
```

```
Mread.get counter;;
```

```
(* Mwrite.t est un type différent de Mread.t *)
```

Solution : Les contraintes de partage

- ▶ Une solution élégante : des *contraintes de partage*.
- ▶ `module Nom1 = (Nom2 : SIG with type t1 = t2 and ...)`
- ▶ OCaml vérifie que les types `t1` et `t2` sont *compatibles*.
- ▶ Une fois la compatibilité vérifiée, OCaml garde l'information que `t1 = t2`, *même s'ils sont abstraits*.

Exemples (sharing4.ml)

```
(* avec contraintes de partage *)
module Mwrite = (M:Write with type t = M.t) ;;
module Mread = (M:Read with type t = M.t) ;;

let counter = Mwrite.create();;
Mwrite.step counter;;
Mread.get counter;;

(* OK car Mwrite.t = M.t = Mread.t = int ref *)
(* NB : Pas encore de type abstrait ici *)
```

Solution : Les contraintes de partage

- ▶ Attention ! Dans l'exemple précédent `M.t` n'est pas abstrait
- ▶ Mieux : on définit *d'abord* une version `Mabs` de `M` où `Mabs.t` est abstrait
- ▶ Puis seulement ensuite on la découpe en `Mwrite` et `Mread`
- ▶ Note : l'interface complète `ReadWrite` pourra s'obtenir ensuite de `Read` et `Write` via des `include`

Exemples (sharing5.ml)

```
(* Interface abstraite pour Read et Write *)
module type ReadWrite = sig
  type t
  val create : unit -> t
  val step : t -> unit
  val get : t -> int
end ;;
```

Exemples (sharing6.ml)

```
(* avec contraintes de partage ET types abstraits *)
module Mabs = (M: ReadWrite) ;;
module Mwrite = (Mabs: Write with type t = Mabs.t) ;;
module Mread = (Mabs: Read with type t = Mabs.t) ;;

let counter = Mwrite.create();;
Mwrite.step counter;;
Mread.get counter;;

(* OK car Mwrite.t = Mabs.t = Mread.t *)
(* Et aucun n'est compatible avec int ref *)
```


Exemples (private1.ml)

```

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 = One of 'a | Cons of 'a * 'a t
  let cons x l = Cons(x, l)
  let pop = function
    | Cons(x, l) -> x, Some l
    | One(x) -> x, None
  let create x = One(x)
end;;

```

Exemples (private2.ml)

```
open NonEmptyList
```

```
let is_singleton = function
  | One(_) -> true
  | Cons(_,_) -> false
;;
```

```
(* le type est abstrait *)
```

L'abstraction empêche le pattern matching

- ▶ Abstraction d'un type algébrique : on ne peut plus, à l'extérieur du corps du module, faire du pattern matching.
- ▶ Première solution : écrire des fonctions pour tester pour les constructeurs, et pour accéder aux composants des valeurs (très lourd !)
- ▶ Un compromis : les types *privés* !
- ▶ Le type qui implémente 'a t est exposé *en lecture seule* :
 - ▶ on peut utiliser le pattern matching pour décomposer une valeur ;
 - ▶ mais seulement les fonctions du même module peuvent créer des valeurs de ce type.
- ▶ Permet toujours des *invariants* dans la construction de valeurs.

Exemples (private3.ml)

```
module type NELprivate = sig  
  type 'a t = private One of 'a | 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 = One of 'a | Cons of 'a * 'a t  
  let cons x l = Cons(x, l)  
  let pop = function  
    | Cons(x, l) -> x, Some l  
    | One(x) -> x, None  
  let create x = One(x)  
end;;
```

Exemples (private4.ml)

```
open NonEmptyListPriv;;
```

```
(* impossible de construire directement des listes *)  
cons 3 (One(4));;
```

```
(* on peut faire une definition par cas *)
```

```
let rec map f = function
```

```
  | One(x) -> create (f x)
```

```
  | Cons(x,l) -> cons (f x) (map f l)
```

```
;;
```

```
map (function x -> x+1) (cons 1 (cons 2 (create 3)))
```

Pointeurs pour aller plus loin

Le conflit entre abstraction et définition par cas (pattern matching) a été l'objet d'attention en littérature 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 dépendances

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

Extension : Modules récursifs

Une extension expérimentale permet des définitions *récurives* 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*

```
module type Nom = 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 dépendances 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 séparément, en respectant l'ordre imposé par les dépendances.

Quelques outils

- `ocamldep` calcule les dépendances d'un fichier `.ml` ou `.mli`
- `ocamldsort` calcule un ordre d'édition de liens compatible avec les dépendances entre fichiers (modules)
- `make` outil standard qui peut servir à compiler un projet, si on connaît les dépendances
- `ocamlbuild` essaye de compiler un projet OCaml en découvrant les dépendances tout seul
- `dune` alternative à `ocamlbuild`

Exemples (stack1.ml)

```
(* signature d'un module pour les piles trieées *)
module type Stack = sig
  type elt
  type t
  val push : elt -> t -> t
  val pop : t -> elt * t
  val is_empty : t -> bool
  val empty : unit -> t
end;;
```

```

(* Implementation *)
module OrdIntStack : Stack =
struct
  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 Not_found
    | h::t -> (h,t)
  let is_empty s = s = []
  let empty () = []
end;;

open OrdIntStack
let x = push 42 (empty());;
(* ou est l'erreur ? *)
    
```

Rendre le module paramétrique par rapport à `elt` ?

- ▶ On souhaite avoir des piles triées d'entiers, de flottants, de liste de chaînes de caractères,
- ▶ Première étape : identifier le paramètre : On peut recueillir dans un module séparé les informations relatives au type `elt`.
- ▶ Le type `elt` doit être équipé d'une fonction de comparaison.

```
(* structure d'entiers *)
```

```
module T = struct
```

```
  type elt = int
```

```
  let compare x y = x-y
```

```
end;;
```

```
(* utiliser "with" pour faire le lien *)
```

```
module OrdTStack : Stack with type elt = T.elt =  
struct
```

```
  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 Not_found
```

```
    | h::t -> (h,t)
```

```
  let is_empty s = s = []
```

```
  let empty () = []
```

```
end;;
```

Première étape : identifier le paramètre

- ▶ Ensuite, si on décide de changer le type, on pourra simplement éditer le fichier et changer la définition du module T , non ??
- ▶ Non ! Si on veut avoir de piles ordonnées de types différents, on devrait dupliquer le code, et on ne veut pas faire ça !
- ▶ Solution : On utilise des *modules paramétrés*, ou *foncteurs* qui peuvent prendre un ou plusieurs autres modules en paramètre.

```

(* signature pour des types ordonnées *)
module type Comparable = sig
  type elt
  val compare : elt -> elt -> int
end;;

(* module paramétrique *)
module OrdStack (T:Comparable) :
  Stack with type elt = T.elt =
struct
  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 Not_found
    | h::t -> (h,t)
  let is_empty s = s = []
  let empty () = []
end;;
    
```

Exemples (stack5.ml)

```

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

module OrdIntStack = OrdStack (Int);;
(* or: *)
module OrdIntStack =
  OrdStack
    (struct type elt = int
      let compare x y = x-y
    end)

open OrdIntStack;;
let p = push 3 (push 9 (push 7 (empty())));;
pop p;;

```


Deuxième étape : expliciter le paramètre

Attention La contrainte `with type` dans la définition du module `OrdStack` est essentielle !

Correction de l'application d'un foncteur

- ▶ Étant donné et un module
 - ▶ un foncteur $M(T : S) : R$
 - ▶ un module M'
- ▶ quand peut-on appliquer le foncteur M au module M' ?
- ▶ C'est le cas quand on peut restreindre M' à la signature S , c'est-dire quand $(M' : S)$ est valide, i.e. :
- ▶ M' fournit tous les éléments déclarés dans la signature S
- ▶ avec un type égal ou plus général
- ▶ éventuellement des définitions pour les types abstraits de S
- ▶ Si on connaît la signature S' de M' on peut vérifier la compatibilité en comparant directement S et S' , qui doivent être dans une relation de *sous-typage*.

Variantes

On peut écrire tout aussi bien

```
module OrdStack (T: Comparable) : Stack =  
  struct ... end
```

que

```
module OrdStack =  
  functor (T: Comparable) -> (struct ... end: Stack)
```

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
```

à chaque étape on peut définir des contraintes de type mentionnant les signatures définies précédemment.

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 (T:Content) :
    Collection with type content = T.t =
struct
    type content = T.t
    type t = ...
    ...
end
```

Extensions

- ▶ Inclusion d'une structure : permet d'ajouter facilement des fonctionnalités à un module.
- ▶ Inclusion d'une signature : permet des extensions faciles de signatures.
- ▶ Récupérer la signature d'une structure
- ▶ Inclusion d'une signature avec redefinition de type
- ▶ First-class modules

Exemples (include1.ml)

```

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

```

module Counter2 =
  struct
    include Counter
    let step n = c := !c+n
    let incr () =
      Printf.eprintf "Inside Counter2\n%!"; step 1
  end;;
    
```


Exemples (include2.ml)

(compteur c partagé entre les deux modules ! *)*

```
let _ = Counter.incr ();;
let _ = Counter.show ();;
let _ = Counter2.incr ();;
let _ = Counter.show ();;
```

Exemples (include3.ml)

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

```
module type Counter2 =
sig
  include CounterFullItf
  val step : int -> unit
end;;
```

Exemples (include4.ml)

```
(* module type of : usage typique *)
```

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

Exemples (include5.ml)

```

module type Printable = sig
  type t
  val print : Format.formatter -> t -> unit
end;;

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

module type PrintableComparable = sig
  include Printable
  include Comparable with type t := t
end;;

```

First-Class Modules

- ▶ Extension expérimentale du langage depuis OCaml 3.12, ne fait toujours pas partie du manuel officiel.
- ▶ **module** convertit un module en une valeur.
- ▶ **val** convertit une valeur en un module.
- ▶ Dans les deux cas il est parfois nécessaire d'indiquer la signature du module.

Exemples (firstclass.ml)

```
(* suite de include1.ml *)
```

```
let c1 = (module Counter: CounterFullItf);;
```

```
let c2 = (module Counter2: CounterFullItf);;
```

```
let compteur sel =
```

```
  let module C = (val (if sel then c1 else c2))
```

```
  in (C.incr(),C.show())
```

```
;;
```

```
compteur true;;
```

```
compteur false;;
```

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...