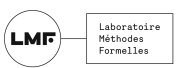
Les modules voyagent en première classe

Jean-Christophe Filliâtre

Algorithmique et programmation
Luminy, 27 avril–1er mai 2026



1. les modules d'OCaml
2. leur compilation
3. les modules de première classe
4. deux applications

une série d'articles au milieu des années 1990 pour définir un système de modules pour Standard ML et Caml

introduit en 1995 avec Caml Special Light

voir notamment

Xavier Leroy, A Modular Module System (JFP, 2000)

qui explique comment ajouter ce système de modules sur un langage donné
(article auto-référentiel !)

le code est découpé en unités de compilation appelées modules

- un fichier = un module
- un module agrège des types, des valeurs et des modules

les modules structurent l'espace de noms

```
Array.make
```

```
Euler.Gauss.add
```

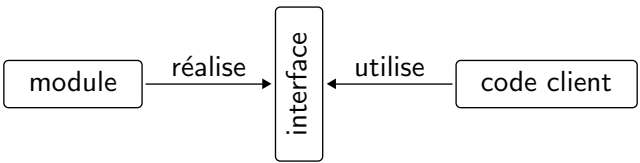
c'est pourquoi

- beaucoup de types s'appellent `t`
- beaucoup de valeurs s'appellent `empty`
- beaucoup de fonctions s'appellent `add`
- etc.

car ils ou elles vivent dans des modules différents

```
module Gauss = struct
  type t = ...
  let add x y = ...
  ...
end
```

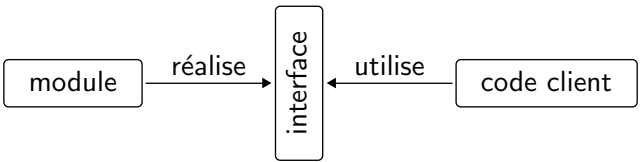
les modules offrent le concept de barrière d'abstraction




```
module M : sig ...  interface  ... end (* aussi appelée signature *)
      = struct ... réalisation ... end
```

```
module C = struct ... M.f ... end
```

les modules offrent le concept de barrière d'abstraction

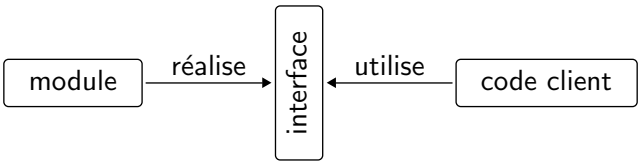



```
module Avl : sig
  type t
  val empty: t
  val add: string -> t -> t
  val mem: string -> t -> bool
  val union: t -> t -> t
end
```

ne révèle pas la définition du type `t`

n'exporte pas non plus les fonctions auxiliaires `height`, `join_left`, etc.

les modules offrent le concept de barrière d'abstraction



le compilateur vérifie, statiquement,

- d'une part l'adéquation du module à son interface
  - ce qui est promis est bien là, avec le type attendu
- d'autre part la bonne utilisation de l'interface
  - ce qui est utilisé existe, avec un type convenable

il le fait indépendamment (compilation séparée)

on peut définir des signatures

```
module type SET = sig
  type elt
  type t
  val empty: t
  val add: elt -> t -> t
  val mem: elt -> t -> bool
  ...
end
```

et les instancier

```
module Avl: SET with type elt = string
```

évite notamment de dupliquer des signatures

```
module Avl : SET with type elt = string
module Rbt : SET with type elt = string
module Avlc: SET with type elt = char
...
```

mais le compilateur va les expanser (le système est structurel, pas nominatif)

on peut définir un module paramétré par un ou plusieurs modules

on appelle cela un foncteur

cela permet notamment

- la factorisation de code (un code générique instancié plusieurs fois)
- le code ouvert (un code générique qui sera instancié plus tard)

```
module Avl(E: sig type t
                  val compare: t -> t -> int end)

= struct


  type t = Empty | Node of int * t * E.t * t
  let add x t = ... E.compare ...
  ...

end
```

```
module Avl(E: sig type t
                  val compare: t -> t -> int end)
: SET with type elt = E.t (* <- abstraction *)
= struct

  type elt = E.t
  type t = Empty | Node of int * t * elt * t
  let add x t = ... E.compare ...
  ...

end
```

```
module type ORD = sig
  type t
  val compare: t -> t -> int
end

module Avl(E: ORD) : SET with type elt = E.t
= struct

  type elt = E.t
  type t = Empty | Node of int * t * elt * t
  let add x t = ... E.compare ...
  ...

end
```

pour avoir des ensembles d'entiers :

```
module S = Avl(struct type t = int let compare = Int.compare end)
```

et on peut même écrire directement

```
Avl(Int)
```

parce que le module `Int` fournit au moins le type `t` et la fonction `compare` attendus
(c'est du sous-typage structurel)

dans la bibliothèque standard d'OCaml, on trouve notamment

- des ensembles applicatifs

```
Set.Make(X: sig type t
                val compare: t -> t -> int end) : ...
```

- des dictionnaires applicatifs

```
Map.Make(X: sig type t
                val compare: t -> t -> int end) : ...
```

- des tables de hachage (mutables)

```
Hashtbl.Make(X: sig type t
                    val equal: t -> t -> bool
                    val hash: t -> int         end) : ...
```

une bibliothèque de graphes pour OCaml, dans laquelle on trouve

- des structures de données de graphes

```
module G = Persistent.Digraph.ConcreteLabeled(Int)(Int)
```

- des algorithmes sur les graphes

```
module Comp = Components.Make(G)
```

dans des cas simples, on peut se contenter de polymorphisme et d'ordre supérieur

une fonction

```
val sort: ('a -> 'a -> int) -> 'a list -> 'a list
```

est sans doute plus simple à utiliser qu'un foncteur

```
module Sort(E: ORD) : sig
  val sort: E.t list -> E.t list
end
```

modules et types sont décorrélés

un module peut définir zéro, un ou plusieurs types

pour calculer modulo `m`

```
module Modular(M: sig val m: int end) : sig
  type t
  val of_int: int -> t
  val add: t -> t -> t
  val mul: t -> t -> t
  ...
end
```

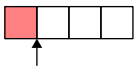
```
include Modular(struct let m = 1_000_000_007 end)
```

	OCaml	Haskell	Rust	Java
espace de noms	modules	modules	modules	paquets, classes
encapsulation	modules	modules	modules	paquets, classes
interface	`sig`	`class`	`trait`	`interface`
implémentation	`module`	`instance`	`impl`	`class`
prog. générique	foncteur	polymorphisme ad-hoc	polymorphisme ad-hoc	polymorphisme borné

2. compilation des modules

une valeur OCaml est un mot

- soit un entier
  - une valeur de type `int`, `bool`, `char`
  - un constructeur constant comme `[]`
  - etc.
- soit un pointeur vers un bloc de plusieurs mots, alloué sur le tas
  - un enregistrement
  - un tableau
  - un constructeur comme `::`
  - etc.



cette uniformité de représentation (toute valeur est un mot) permet le polymorphisme paramétrique

une fonction polymorphe comme

```
let twice x =
  (x, x)
```

est compilée une seule fois et fonctionne à l'identique sur n'importe quel type

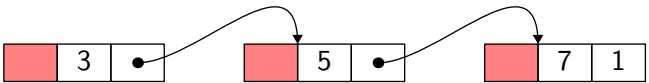
il n'y a aucune représentation des types à l'exécution

un entier n est en réalité représenté par $2n + 1$

de cette façon, le GC distingue

- les pointeurs qui sont pairs car alignés
- les entiers qui sont donc impairs

ainsi, la liste `1::2::3::[]` se retrouve représentée comme



sur du code premier ordre, une fonction OCaml

```
let f x y z =
  x + y + 8*z
```

est compilée comme on s'y attend

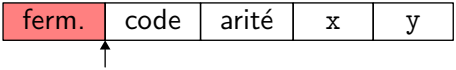
```
f: addq %rbx, %rax
   leaq -9(%rax,%rdi,8), %rax
   ret
```

(ou presque)

une fonction utilisée comme une valeur de première classe, en revanche

```
let g x y =
  fun z -> x + y + 8*z
```

est compilée comme une fermeture

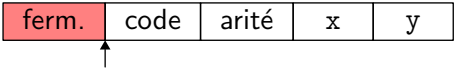



```
g: ... allouer 40 octets ...
   movq $4343, -8(%rdi)
   movq fun_288(%rip), %rsi
   movq %rsi, (%rdi)
   movabsq $72057594037927941, %rsi
   movq %rsi, 8(%rdi)
   movq %rax, 16(%rdi)  # sauver x
   movq %rbx, 24(%rdi)  # sauver y
   movq %rdi, %rax
   ret
fun_288:
  movq  24(%rbx), %rdi  # récupérer y
  movq  16(%rbx), %rbx  # récupérer x
  addq  %rdi, %rbx
  leaq  -9(%rbx,%rax,8), %rax
  ret
```

une fonction utilisée comme une valeur de première classe, en revanche

```
let g x y =
  fun z -> x + y + 8*z
```

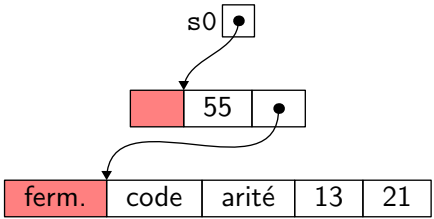
est compilée comme une fermeture




```
g: ... allouer 40 octets ...
   movq $4343, -8(%rdi)
   movq fun_288(%rip), %rsi
   movq %rsi, (%rdi)
   movabsq $72057594037927941, %rsi
   movq %rsi, 8(%rdi)
   movq %rax, 16(%rdi)  # sauver x
   movq %rbx, 24(%rdi)  # sauver y
   movq %rdi, %rax
   ret
fun_288:
  movq  24(%rbx), %rdi  # récupérer y
  movq  16(%rbx), %rbx  # récupérer x
  addq  %rdi, %rbx
  leaq  -9(%rbx,%rax,8), %rax
  ret
```

```
type s = { v: int; f: int -> int }

let s0 = { v = 27; f = g 6 10 }
```



si on n'utilise pas de foncteur, mais seulement des modules pour structurer

```
module A = struct
  let f x = ...
  module B = struct
    let f x y = ...
    ...
```

alors le code compilé n'est pas différent de celui d'un code à plat

```
let f1 x = ...
let f2 x y = ...
```

les choses se compliquent avec les foncteurs, car

- deux applications d'un même foncteur ont des comportements différents
- on ne connaît pas forcément les applications (bibliothèque)

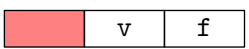
une solution serait la défonctorisation, c'est-à-dire une élimination statique des foncteurs

le compilateur OCaml adopte une autre approche où un module est un enregistrement

exemple :

```
module M : sig
  type t
  val v: t
  val f: t -> t
end
```

devient un enregistrement

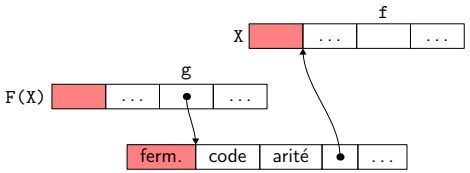


et le type `t` n'est pas représenté

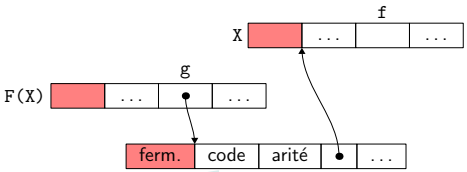
un foncteur est une fonction recevant un enregistrement en paramètre et renvoyant un enregistrement

une application de foncteur n'est pas différente d'une application de fonction

```
module F(X: sig ... val f: ... end) =
struct
  ...
  let g x = ... X.f ...
  ...
end
```

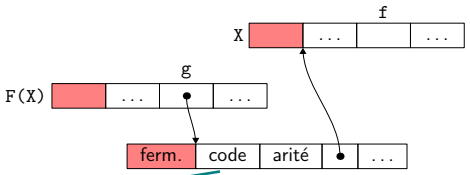


```
module F(X: sig ... val f: ... end) =
struct
  ...
  let g x = X.f (x + 1)
  ...
end
```




```
g: addq $2, %rax
   movq 16(%rbx), %rbx  # récupérer X dans la fermeture
   movq  8(%rbx), %rbx  # récupérer f dans X
   movq    (%rbx), %rdi  # puis son code
   jmp  *%rdi            # et enfin l'appeler
```

```
module F(X: sig ... val f: ... end) =
struct
  ...
  let g x = X.f (x + 1)
  ...
end
```




```
g: addq $2, %rax
   movq 16(%rbx), %rbx   # récupérer X dans la fermeture
   movq  8(%rbx), %rbx   # récupérer f dans X
   movq   (%rbx), %rdi   # puis son code
   jmp  *%rdi            # et enfin l'appeler
```

les deux codes

```
type s = {

  v: int;
  f: int -> int
}

let f (s: s) : s = {

  v = s.v;
  f = fun x -> s.f (s.f x)
}
```

```
module type S = sig
  type t
  val v: t
  val f: t -> t
end

module F(X: S) : S = struct
  type t = X.t
  let v = X.v
  let f = fun x -> X.f (X.f x)
end
```

sont compilés exactement de la même façon

que ce soit OCaml/Haskell/Rust/Java, le typage statique assure la sûreté :
la fonction/méthode appelée existera toujours

pour autant,

- OCaml/Haskell : la liaison est dynamique mais pourrait être statique
- Rust : la liaison est statique
- Java : la liaison est forcément dynamique

3. modules de première classe

puisque les modules sont compilés comme des enregistrements,
rien n'empêche de les utiliser comme des citoyens de première classe

ainsi, un module peut être

- paramètre d'une fonction
- renvoyé par une fonction
- construit localement
- stocké dans une structure de données
- etc.

(depuis 2011 et OCaml 3.12)

un module en paramètre

```
let f (module M: S) ... = ...
```

et l'application correspondante

```
f (module ...)
```

un module défini localement

```
let module M = ... in ...
```

renvoyé par une fonction

```
let f ... = (module ... : S)
```

ou encore stocké dans une structure

```
Hashtbl.add table "foo" (module ...)
```

```
let f p =
  let module X = struct
    type t = ...
    let compare = ...
  end in
  let module S = Set.Make(X) in
  ...
```

une comparaison qui utilise p

les expressions font leur entrée dans le langage de modules :

```
module M = (val ...)
```

ou encore

```
F(val ...)
```

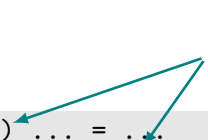
enfin, une signature de module devient un type :

```
val f: int -> (module S)
```

```
val table: (string, (module S)) Hashtbl.t
```

l'inférence de types atteint ses limites avec les modules

signature attendue
(nommée, possiblement contrainte
avec `with type`)



```
let f (module M: S) ... = ...
let f ... = (module ... : S)
```

avec la signature

```
module type S = sig  type t  val x: t  ... end
```

la fonction

```
let f (module M: S) = M.x
```

est rejetée au typage

```
The type constructor M.t would escape its scope
```

le langage des expressions et celui des modules sont devenus mutuellement dépendants, de même que ceux des types et des signatures

c'est au-delà de *Modular Module System*

4. deux applications

1. anagrammes
2. calculatrice

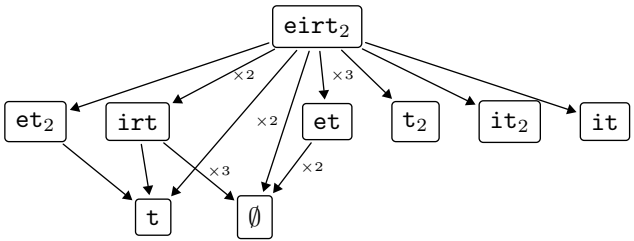
avec toutes les lettres de « `algorithmique et programmation` », c'est-à-dire le multiensemble

$$\mathtt{a}_3\mathtt{e}_2\mathtt{g}_2\mathtt{hi}_3\mathtt{lm}_3\mathtt{no}_3\mathtt{pqr}_3\mathtt{t}_3\mathtt{u}$$

et tous les mots d'un dictionnaire donné,

combien peut-on faire de phrases anagrammes,
c'est-à-dire de concaténations de mots utilisant exactement toutes ces lettres ?

avec les lettres de « titre »,



il y a $2 \times 3 + 2 + 3 \times 2 = 14$ chemins

comment représenter des multiensembles raisonnablement petits ?

en particulier, on veut

- tester l'inclusion
- calculer la différence
- parcourir tous les sous-multiensembles

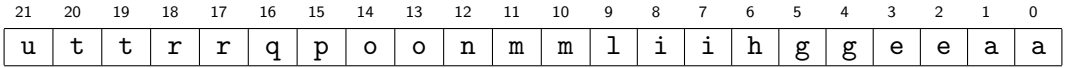
```
module type MULTISET = sig
  type elt
  type t
  val empty: t
  val occ: elt -> t -> int
  val add1: elt -> t -> t
  val remove1: elt -> t -> t
  val inclusion: t -> t -> bool
  val diff: t -> t -> t
  val iter_sub: (t -> t -> unit) -> t -> unit
  ...
end
```

si le multiensemble n'est pas trop grand, il peut être représenté facilement par un entier

ainsi, tous les multiensembles contenus dans $\mathtt{a}_3\mathtt{e}_2\mathtt{g}_2\mathtt{hi}_3\mathtt{lm}_3\mathtt{no}_3\mathtt{pqr}_3\mathtt{t}_3\mathtt{u}$ peuvent être représentés sur 22 bits :

- 2 bits pour compter les a
- 2 bits pour compter les e
- 2 bits pour compter les g
- 1 bit pour compter les h
- etc.

par exemple comme ceci



pour des capacités maximales annoncées,

$$\texttt{a}_3\texttt{e}_2\texttt{g}_2\texttt{hi}_3\texttt{lm}_3\texttt{no}_3\texttt{pqr}_3\texttt{t}_3\texttt{u}$$

on pré-calcule une table comme ceci

	a	e	g	h	i	l	m	n	o	p	q	r	t	u
position	0	2	4	6	7	9	10	12	13	15	16	17	19	21
masque	3	3	3	1	3	1	3	1	3	1	1	3	3	1

- égalité, comparaison et hachage sont immédiats
- si $x \subseteq y$ alors $y - x$ est la différence multiensembliste
- si l'union $x \cup y$ est représentable, alors $x + y$ la calcule
- si $x \subseteq y$ alors $x \leq y$

pour un type d'éléments

```
module type UNIVERSE = sig
  type t
  val hash: t -> int
  val equal: t -> t -> bool
end
```

on fournit des multiensembles sur un univers borné

```
module Make(X: UNIVERSE) : sig
  val create: (X.t * int) list ->
    (module MULTISET with type elt = X.t)
end
```

calcul de la table ici

(avec possiblement `Invalid_argument "capacity exceeded"`)

bien entendu, on pourrait se contenter d'un foncteur (en passant la liste dans un module)

mais c'est bien plus pratique avec une fonction, par exemple pour construire la liste à partir d'une chaîne de caractères

pour « `algorithmique et programmation` », c'est-à-dire le multiensemble

$$\mathtt{a}_3\mathtt{e}_2\mathtt{g}_2\mathtt{hi}_3\mathtt{lm}_3\mathtt{no}_3\mathtt{pqr}_3\mathtt{t}_3\mathtt{u}$$

et le dictionnaire français de `ispell`, on obtient

392 894 754 754 917 420 anagrammes

en explorant un graphe de 1 673 451 sommets et 493 858 465 arcs en 20 minutes

objectif : réaliser une calculatrice offrant

- plusieurs formats de nombres
  `int`, `i32`, `u32`, `f64`, etc.
- plusieurs formats d'affichage pour chacun
  `dec`, `hex`, `bin`, etc.
- le produit cartésien de plusieurs formats
  `int` $\times$ `i63`, etc.

un format de nombres est un module

```
module type Number = sig
  val name: string
  type t
  ...
end
```

les formats sont des modules de première classe

stockés dans une table

```
let formats : (string, (module Number)) H.t = H.create 16

let add_format (module N: Number) =
  if H.mem formats N.name then Lib.panic "already a format '%s'" N.name;
  H.add formats N.name (module N)

...
```

on peut trier les formats par ordre alphabétique avant de les afficher

```
let cmp (module X: Number) (module Y: Number) =
  String.compare X.name Y.name in
H.fold (fun _ -> List.cons) formats []
|> List.sort cmp
|> List.iter show_format
```

on peut sélectionner un format sur la ligne de commande

```
> tuc -f i32
```

va chercher le module correspondant à `"i32"` dans la table
et le passe à un foncteur qui réalise la calculatrice

- Xavier Leroy. A Modular Module System.
  Journal of Functional Programming, 2000.
- P. Wadler, S. Blott. How to make ad-hoc polymorphism less ad hoc.
  POPL 89.
- OCamlGraph.
- Compiler Explorer.
- JCF. Les anagrammes de ce titre.
  JFLA 2025.
- Leo White, Frédéric Bour, Jeremy Yallop. Modular Implicits.
  ML Family/OCaml Workshops, 2014.