

## Cours 6-2 : Modules

# 1 Modules

Le système de modules d’OCaml permet de manipuler des collections de définitions du langage de base. Un *module* est constitué d’un ensemble de définitions (fonctions, types, exceptions, autres modules, ...) délimitées par les mots-clés `struct ... end`. Un module peut être nommé avec la construction `module <nom> = ...`, par exemple :

```
module M = struct
  type t = int
  let f x = x * 5
end
```

Pour accéder à une valeur `v` définie dans un module `M`, on utilise la notation `M.v` :

```
# M.f 12;;
- : int = 60
```



Même si les deux mécanismes sont à un certain point similaire, il ne faut pas confondre modules et fichiers. En particulier, un fichier peut contenir plusieurs modules, et les modules ne permettent pas de compilation séparée.

# 2 Signatures

## 2.1 Définition et utilisation

Comme toutes les autres entités du langage OCaml, les modules sont typés et le type d’un module est appelé une *signature*. Une signature est constituée d’un ensemble de déclarations (types, signatures de fonctions, ...) délimitées par les mots-clés `sig ... end`. Une signature peut être définie avec la construction `module type <nom> = ...`, par exemple la signature correspondant au module `M` ci-dessus :

```
module type MS = sig
  type t = int
  val f : int -> int
end
```

Lors de la création d’un module, on peut forcer la signature de celui-ci en utilisant la syntaxe `module <nom> : <signature> = ...`, ainsi on pourrait re-définir le module `M` en spécifiant sa signature :

```
module M' : MS = M
```

## 2.2 Types abstraits, contraintes de types

Spécifier explicitement la signature d'un module permet de ne faire apparaître que certaines définitions, et de cacher les autres. Ainsi, dans l'exemple suivant, on cache la fonction `debug` du module `A` :

```
module A : sig val f : int -> int end = struct
  let f x = x + 1
  let debug () = print "debug"
end
```

De la même manière que l'on peut rendre un type abstrait dans un `.mli`, on peut rendre un type abstrait dans un module en le déclarant sans définition dans la signature :

```
module type NS = sig type t val f : t -> t end
module N : NS = M
# N.f 3;;
This expression has type int but is here used with type M'.t
```

Le mot-clé `with` permet enfin d'ajouter des contraintes de types à une signature lors de la création d'un module. Cela est typiquement utile pour exprimer des contraintes sur des types abstraits :

```
module N' : NS with type t = int = M
# N'.f 3;;
- : int = 15
```

## 3 Extensions de modules/signatures

À la manière de l'héritage dans les langages orientés objets, on peut définir des modules par extension de modules existant. Le mot clé `include` permet d'inclure les définitions d'un module dans un autre.

```
module Mplus = struct
  include M
  type other = t * string
  let h (x, s) = (f x, s)
end
```

On peut similairement définir des signatures par extension, en utilisant le même mot-clé `include` au sein d'une signature.



`include M` permet d'ajouter les définitions du module `M`, et non le module `M` lui-même. Autrement dit, le module `Mplus` ci-dessus n'est pas équivalent au module suivant :

```
module Mplus' = struct
  module M = M
  type other = M.t * string
  let h (x, s) = (M.f x, s)
end
```