

LANGAGE DE MODULES

Modules

Les notions d'**implémentations** et **interfaces** peuvent être rendues plus fines par des constructions du langage.

Définition d'une interface (**signature**) I dans un programme :

```
module type I = sig
    val a : int
    val f : int -> int
end
```

Définition d'une implémentation (**module**) M ayant cette interface :

```
module M : I = struct
    let a = 42
    let b = 3
    let f x = a * x + b
end
```

Le compilateur fait alors les même opérations que si I était un fichier M.mli et M un fichier M.ml

Modules paramétrés

- ▶ Comme les fonctions, les modules peuvent attendre des paramètres. Ces modules paramétrés sont appelés **foncteurs**.
- ▶ Le langage impose que ces paramètres soient des **modules**.

Définition d'un foncteur **M** qui attend un module **S** de signature **T** en paramètre :

```
module M ( S : T ) = struct  
    ...  
end
```

Pour créer une instance de M, il suffit de l'appliquer à un module ayant la signature T. Par exemple, si B a pour signature T, alors on crée un M en faisant :

```
module A = M(B)
```

EXAMPLE

Exemple : des éléments comparables

Les éléments à trier sont rendus abstraits à l'aide d'une **signature** qui indique just le fait qu'ils ont une fonction de comparaison :

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

Le type `t` est **abstrait**.

Exemple : interface d'un trieur d'éléments comparables

Le module de tri à développer aura la signature S suivante :

```
module type S =  
  sig  
    type t  
    val insertion : t -> t list -> t list  
    val tri : t list -> t list  
  end
```

Exemple : foncteur d'un trieur d'éléments comparables

Voici le foncteur correspondant :

```
module T(E : ELT) : S with type t = E.t
=
struct

  type t = E.t
  type tl = t list

  let rec insertion x l =
    match l with
    | [] -> [x]
    | y :: s ->
      if E.compare x y <= 0 then
        x :: l
      else
        y :: (insertion x s)

  let rec tri l =
    match l with
    | [] -> []
    | x :: s -> insertion x (tri s)
end
```

Exemple : modules trieurs d'éléments comparables

Instanciation du foncteur T :

```
module E1 = struct
  type t = int
  let compare x y = y - x
end

module T1 = T(E1)
let l1 = [4;1;5;3;-7]
let l2 = T1.tri l1
let l3 = T1.insertion 6 l2

module E2 = struct
  type t = int
  let compare x y = y*y - x*x
end

module T2 = T(E2)
let l4 = T2.tri l1
```


Exemple : préconditions...

`T1.insertion` prend une liste triée selon `T1` comme argument, et renvoie une liste triée selon `T1`.

Or `T1.insertion` accepte toute liste de type entier. On peut toujours insérer un élément dans une liste non triée, ou mélanger les fonctions de ces deux instances :

```
let l5 = T1.insertion 4 [3;1;2]      (* Bug *)  
let l6 = T1.insertion 6 l4          (* Bug *)
```

Il faudrait un type liste spécifique à chaque module (à chaque instance du foncteur `T`).

Exemple : préconditions... garanties par les types

Nouvelle signature pour le foncteur T avec type abstrait `tl`.

```
module type SL =  
  sig  
    type t  
    type tl  
    val export : tl -> t list  
    val insertion : t -> tl -> tl  
    val tri : t list -> tl  
  end
```

Dans notre implémentation, `tl` sera bien une `t list`, mais au niveau de la déclaration de type du foncteur, rien ne le garantira : ce type `tl` sera donc interne à chaque instance du foncteur T.

Exemple : préconditions... garanties par les types

De cette manière, impossible de mélanger les listes triées par les différentes instances de `T`. Impossible également d'insérer un élément dans une liste non triée.

```
let l5 = T1.insertion 4 [3;1;2]
```

Error: This expression has type 'a list
but an expression was expected of type `T1.tl = Sort.T(E1).tl`

```
let l6 = T1.insertion 6 l4
```

Error: This expression has type `T2.tl = Sort.T(E2).tl`
but an expression was expected of type `T1.tl = Sort.T(E1).tl`

Exemple : exportation type interne vers type externe

```
module T(E : ELT) : SL with type t = E.t
=
  struct

    type t = E.t
    type tl = t list

    let export l = l

    ...
  end
```

Pour pouvoir faire :

```
List.iter (Printf.printf "%d ") (T1.export l2);
```