Programmation 1

Modules OCaml

David Baelde

ENS Rennes, L3 SIF, 2024–2025

Introduction

Programmer à large échelle (taille, complexité, re-utilisation...)
nécessite des mécanismes spécifiques de structuration des programmes.

Deux grands principes :

- **modularité** : organisation du code en unités distinctes, sous-unités, etc.
- **abstraction** : cacher des détails, e.g. comment une fonctionalité est implémentée.

Aujourd'hui nous allons découvrir le système de modules d'OCaml,
principal mécanisme de structuration du code à large échelle dans notre langage :

- structures, signatures, foncteurs,
- compilation séparée,
- types abstraits, privés, programmation générique...

D'autres mécanismes existent, e.g. objets (C++, Java, OCaml) ou classes de types (Haskell).

Modularité

Structures

Structure = paquet de déclarations, e.g. valeurs et types.
On confond souvent “structure” et “module”.

```
module List = struct
  type 'a list = Nil | Cons of 'a * 'a list
  let rec mem x = function
    | Nil -> false
    | Cons (a,l) -> a = x || mem x l
end
let l : int List.list = List.Cons (1, List.Nil)
```

Remarques

- `module` lie un nom à une structure construite par `struct` (similaire à un `let`).
- Les noms de structures commencent par une majuscule, convention `CommeCeci`.
- Eléments d'une structure `Struct` dénotés par `Struct.élément`.

La construction open

Ajoute à l'environnement courant le contenu de la structure.
Évite ensuite d'avoir à faire référence explicitement à cette structure.

Version 1 :

```
module SetList = struct
  type 'a set = 'a List.list
  let empty = List.Nil
  let add x l =
    if List.mem x l then l else List.Cons (x,l)
end
```

La construction open

Ajoute à l'environnement courant le contenu de la structure.
Évite ensuite d'avoir à faire référence explicitement à cette structure.

Version 2, rigoureusement équivalente :

```
module SetList = struct
  open List
  type 'a set = 'a list
  let empty = Nil
  let add x l =
    if mem x l then l else Cons (x,l)
end
```

La construction open

Ajoute à l'environnement courant le contenu de la structure.
Évite ensuite d'avoir à faire référence explicitement à cette structure.

Version 2, rigoureusement équivalente :

```
module SetList = struct
  open List
  type 'a set = 'a list
  let empty = Nil
  let add x l =
    if mem x l then l else Cons (x,l)
end
```

Après la définition de `SetList`, que dénote `SetList.Nil` ?

La construction include

Inclut une structure dans un autre. Enrichit l'environnement et la structure.

```
module List_ext = struct
  include List
  let rec length = function
    | Nil -> 0
    | Cons (_,l) -> 1 + length l
end

let l = List_ext.Nil
```

Organiser les noms

Un constructeur (resp. champ) appartient à un unique type variant (resp. enregistrement)

```
type t1 = Nil | Cons of int*t1 | App of t1*t1
type point2d = { x : int ; y : int }

type t2 = Nil | Cons of int*t2
type point3d = { x : float ; y : float ; z : float }

(* Sans information de typage supplémentaire,
   Nil et Cons appartiennent à t2; x et y à point3d. *)

let a = Nil                (* a : t2                *)
let f r = r.x              (* f : point3d -> float  *)
let g (r:point2d) = r.x    (* g : point2d -> int    *)
let b = App (Nil,Nil)      (* b : t1                *)
```

Organiser les noms

Un constructeur (resp. champ) appartient à un unique type variant (resp. enregistrement) ... dans un module donné.

```
module Foo = struct
  type t1 = Nil | Cons of int*t1 | App of t1*t1
  type point2d = { x : int ; y : int }
end
module Bar = struct
  type t2 = Nil | Cons of int*t2
  type point3d = { x : float ; y : float ; z : float }
end

let a = Foo.Nil
let f r = r.Foo.x
```

Abstraction

Signatures

Une signature est le type d'une structure — son interface.
Elle indique quels éléments de la structure sont visibles du dehors.

```
module type LIST = sig
  type 'a list = Nil | Cons of 'a * 'a list
  val mem : 'a -> 'a list -> bool
end

module List : LIST = struct (* comme avant *) end
```

Remarques

- Distinguer `module type` et `sig ... end`.
- Noms avec ou sans majuscule, convention `CommeCeci` en OCaml, `COMMECELA` en SML.
- Tout module a un type par défaut, qui exporte tout.

Cachotteries

Pour une même structure, on peut décider de cacher plus ou moins de choses via des interfaces plus ou moins riches :

- la déclaration d'un type ou d'une valeur ;
- le type le plus général des valeurs ;
- l'implémentation d'un type.

Nota : l'interface est satisfaite par moins de structures quand on la rend plus riche.

Démo avec `comparing_sigs.ml`.

Exercice

Que cacher dans le code suivant, et comment ?

```
let c = ref 0
let fresh () = incr c ; "var_" ^ string_of_int !c
let reset () = c := 0
```

Exercice

Que cacher dans le code suivant, et comment ?

```
let c = ref 0
let fresh () = incr c ; "var_" ^ string_of_int !c
let reset () = c := 0
```

La solution la plus agréable utilise les modules.
Nota : l'initialisation du module comporte un effet de bord.

Types abstraits

Quand l'implémentation d'un type est cachée par une interface, on parle de type abstrait.

Un des traits les plus importants du système de modules :

- **Faciliter l'évolution du code** :
  la définition du type pourra changer sans impacter les utilisateurs du module.
- **Garantir des invariants** :
  un type abstrait est toujours manipulé explicitement via le module.
- **Re-utiliser du code sans le dupliquer** :
  on verra plus loin comment faire de la programmation générique sur des types abstraits.

Les types abstraits sont nécessaires pour exprimer les *types de données abstraits* (ADT).
Exemple d'implémentation prouvée d'un ADT : `setlist.ml`.

Variantes sur l'abstraction

Type privé

Exposer l'implémentation d'un type sans autoriser la construction de valeurs de ce type.

- Mêmes avantages pour assurer des invariants.
- Facilité d'utilisation, e.g. affichage, accès à des champs, pattern matching.

Type fantôme

Paramètre de type inutilisé dans l'implémentation, néanmoins contraignant dans la signature.

- Exprimer dans les types des propriétés simples des valeurs.
- Contraindre l'utilisation de certaines fonctionalités sur cette base.

Exemples : `abstract_variantes.ml`.

Compilation séparée

Compilation séparée et dépendances

Tout fichier `foo.ml` définit implicitement une structure `Foo`.
Si `foo.mli` est présent il définit la signature associée.

Démo dans `compil/` :

- retour sur la compilation séparée en OCaml ;
- génération automatique de dépendances ;
- différences entre bytecode et code natif ;
- l'erreur commune `inconsistent assumptions` ;
- le problème avec les `'_weak`.

Compilation séparée et dépendances

Tout fichier `foo.ml` définit implicitement une structure `Foo`.
Si `foo.mli` est présent il définit la signature associée.

Démo dans `compil/` :

- retour sur la compilation séparée en OCaml ;
- génération automatique de dépendances ;
- différences entre bytecode et code natif ;
- l'erreur commune `inconsistent assumptions` ;
- le problème avec les `'_weak`.

Remarque : il n'y a pas de module "`Main`" privilégié ;
tous les modules peuvent provoquer des calculs lors de leur initialisation.

Créer des signatures en pratique

Démo avec `multiset.ml` :

1. Création : `ocamlc -c -i multiset.ml > multiset.mli`
2. Minimisation de `multiset.mli`

Créer des signatures en pratique

Démo avec `multiset.ml` :

1. Création : `ocamlc -c -i multiset.ml > multiset.mli`
2. Minimisation de `multiset.mli`

En pratique il faudrait aussi mettre en place la documentation utilisateur dans la signature.

- Les commentaires destinés au développeur n'ont leur place que dans le fichier `.ml`.
- Éviter la duplication de la documentation utilisateur entre `.ml` et `.mli`.
  Inconvénient : doc utilisateur invisible pour le développeur, attention aux erreurs.

Généricité

Ensembles

Une signature abstraite pour les ensembles :

```
module type SET = sig
  type 'a t
  val empty : 'a t
  val add : 'a -> 'a t -> 'a t
  val remove : 'a -> 'a t -> 'a t
  val member : 'a -> 'a t -> bool
  val fold : ('a -> 'b -> 'b) -> 'a t -> 'b -> 'b
end
```

Implémentations possibles :

- listes non triées ;
- listes triées, arbres de recherche... quel ordre ?

Abstraire une structure dans une signature

On veut former des ensembles sur **tout type équipé d'un ordre**.
On peut le dire avec un type de modules :

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

module type MAKESET = functor (E:ORDERED) -> sig
  type t
  type elt = E.t
  val empty : t
  val add : elt -> t -> t
  val remove : elt -> t -> t
  val member : elt -> t -> bool
  val fold : (elt -> 'b -> 'b) -> t -> 'b -> 'b
end
```

Abstraitre une structure dans une structure

On réalise cette signature par un foncteur, i.e., une fonction d'une structure vers une autre.

```
module MakeSetList : MAKESET =
  functor (E:ORDERED) -> struct
    type t = E.t list
    type elt = E.t
    let empty = []
    let eq e e' = E.compare e e' = 0
    let neq e e' = E.compare e e' <> 0
    let remove e s = List.filter (neq e) s
    let member e s = List.exists (eq e) s
    let add e s = if member e s then s else e :: s
    let fold f s x =
      List.fold_left (fun x e -> f e x) x s
  end
```

Une autre écriture

Il serait plus utile de nommer le résultat du foncteur...

```
module type SET = sig
  type t
  type elt
  val empty : t
  val add : elt -> t -> t
  val remove : elt -> t -> t
  val member : elt -> t -> bool
  val fold : (elt -> 'b -> 'b) -> t -> 'b -> 'b
end

module Make (E:ORDERED) : SET = struct
  type t = E.t list
  type elt = E.t
  let empty = []
  (* ... *)
end
```

Démo avec `setmakebis.ml` : définissons le singleton $\{42\}$.

Le mot-clé `with`

Il faut faire le lien entre le type `E.t` et `SET.elt` :

```
module Make (E:ORDERED) : (SET with type elt = E.t) =
  struct
    type t = E.t list
    type elt = E.t
    let empty = []
    (* ... *)
  end
```

Exercice

```
module type PRINTABLE = sig
  type t
  val to_string : t -> string
end

module Test (P:PRINTABLE) (O:ORDERED) = struct
  let f x y =
    let cmp =
      match O.compare x y with
        | 0 -> '='
        | 1 -> '>'
        | _ -> '<'
    in
      Printf.printf "%s␣%c␣%s"
        (P.to_string x) cmp (P.to_string y)
end
```

Exercice

```
module type PRINTABLE = sig
  type t
  val to_string : t -> string
end

module Test (P:PRINTABLE) (O:ORDERED) = struct
  let f x y =
    let cmp =
      match O.compare x y with
        | 0 -> '='
        | 1 -> '>'
        | _ -> '<'
    in
      Printf.printf "%s␣%c␣%s"
        (P.to_string x) cmp (P.to_string y)
end
```

On doit exiger `O : ORDERED with type t = P.t`

Encore un exercice

Retour sur `comparing_sigs.ml`.

Le module Set d'OCaml

Une utilisation fréquente (et bien moins douloureuse) des foncteurs :

```
module S = Set.Make(String)

let s1 = S.add "bonjour" (S.add "au␣revoir" S.empty)
let s2 = S.add "au␣revoir" (S.add "bonjour" S.empty)
let () = assert (S.equal s1 s2)
let () = assert (s1 <> s2)

module SSet = Set.Make(S)

let s = SSet.remove s2 (SSet.add s1 SSet.empty)
let () = assert (SSet.is_empty s)
```

Conclusion

Le système de modules est essentiel pour la pratique d'OCaml.
Garanties du typage pour la programmation à grande échelle.

À retenir

- **Structure** : paquet de types et valeurs.
- **Signature** : type d'un module, permet d'en cacher une partie.
- **Type abstrait** : dont l'implémentation est cachée.
- **Foncteur** : fonction entre structures.
- Les constructions `open`, `include`, `with`...

Conclusion

Le système de modules est essentiel pour la pratique d'OCaml.
Garanties du typage pour la programmation à grande échelle.

À retenir

- **Structure** : paquet de types et valeurs.
- **Signature** : type d'un module, permet d'en cacher une partie.
- **Type abstrait** : dont l'implémentation est cachée.
- **Foncteur** : fonction entre structures.
- Les constructions `open`, `include`, `with`...


ML Module Mania: a Type-Safe, Separately Compiled, Extensible Interpreter, Norman Ramsey, 2005.


Manuel de référence OCaml, Xavier Leroy et al.