Modules (OCa)ML

David Baelde

ENS Cachan, L3 2015–2016

Modules et signatures

Modules

Un paquet contenant des définitions de valeurs et de types.

```
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 valeur construite par `struct`.
  (Ici, masquage local du `List` de la librairie standard.)
- Les noms de modules commencent par une majuscule.
- Eléments d'un module dénotés par `Module.champ`.
- Tout fichier définit un module : `module.ml` donne `Module`.

La construction open

Évite d'avoir à nommer explicitement le contenu d'un module.
Enrichit l'environnement courant.

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

Évite d'avoir à nommer explicitement le contenu d'un module.
Enrichit l'environnement courant.

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 un module dans un autre.
Enrichit l'environnement et la structure courants.

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

Organiser les noms

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

```
type t1 = Int of int | Fun of string * term list
        | Var of string
type point2d = { x : float ; y : float }
```

```
type t2 = Int of int | Fun of string * term list
type point3d = { x : float ; y : float ; z : float }
```

```
let e = Fun ("plus", [Int 3; Var "x"])
let r = { x = 1. ; y = 2. }
```

Organiser les noms

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

```
module Foo = struct
  type t1 = Int of int | Fun of string * term list
          | Var of string
  type point2d = { x : float ; y : float }
end
module Bar = struct
  type t2 = Int of int | Fun of string * term list
  type point3d = { x : float ; y : float ; z : float }
end

let e = Foo.Fun ("plus", [Foo.Int 3; Foo.Var "x"])
let r = { Foo.x = 1. ; Foo.y = 2. }
let r = { Foo. x = 1. ; y = 2. }
let rx = r.Foo.x
```

Signatures

Une signature est le type d'un module.

```
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, CRIENT parfois.
- Tout module a un type par défaut, inféré.
- Un fichier `.mli` est une signature.
- On dispose de `open Module` et `include SIGNATURE`.

Pour vivre heureux vivons cachés

Un module donné admet plusieurs signatures,
plus ou moins spécifiques.

Une signature plus permissive permet de cacher des choses :

- des éléments d'un module, types ou valeurs ;
- le type le plus général des valeurs ;
- l'implémentation d'un type.

Démo avec `multiset.ml`.

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 :

- garantir des invariants,
- faciliter l'évolution du code,
- exprimer des librairies génériques, etc.

Exercice

Que cacher dans le code suivant, et comment ?

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

Foncteurs

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 ?

Polymorphisme ad-hoc

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
```

Polymorphisme ad-hoc

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

Tout va bien ? Démo avec `setmakebis.ml`.

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`

Le module `Set` d'OCaml

L'utilisation la plus fréquente d'un foncteur :

```
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)
```

Compilation

Compilation séparée et dépendances

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 `'_a`.

Compilation séparée et dépendances

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 `'_a`.

Exercice : comparer avec la compilation en C.

Aspects avancés

Effets de bord

L'initialisation d'un module peut avoir des effets de bord, et un module peut cacher un état.

Effets de bord

L'initialisation d'un module peut avoir des effets de bord, et un module peut cacher un état.

```
module type S = sig
  type t
  type o = Int of int | Fun of string * t list
  val observe : t -> o
  val make_var : unit -> t
  exception Unification_failure
  val unify : t -> t -> unit
  type state
  val save : unit -> state
  val restore : state -> unit
end
```

Effets de bord

L'initialisation d'un module peut avoir des effets de bord, et un module peut cacher un état.

```
module type S = sig
  type t
  type o = Int of int | Fun of string * t list
  val observe : t -> o
  val make_var : unit -> t
  exception Unification_failure
  val unify : t -> t -> unit
  type state
  val save : unit -> state
  val restore : state -> unit
end

module MakeTerm (U : sig end) : S = struct
  (* ... *)
end
```

Effets de bord

L'initialisation d'un module peut avoir des effets de bord, et un module peut cacher un état.

```
module type S = sig
  type t
  type o = Int of int | Fun of string * t list
  val observe : t -> o
  val make_var : unit -> t
  exception Unification_failure
  val unify : t -> t -> unit
  type state
  val save : unit -> state
  val restore : state -> unit
end

module MakeTerm (U : sig end) : S = struct
  (* ... *)
end
```

Générativité : les types abstraits `t` et `state` seront distincts d'une instantiation à l'autre du foncteur `MakeTerm`.

Modules de première classe

Au delà des foncteurs, un “vrai” calcul qui renvoie un module ?

```
module T = (val if Sys.argv.(1) = "tutu" then begin
                  Printf.printf "Using␣tutu...\n" ;
                  (module Tutu : S)
                end else begin
                  Printf.printf "Using␣toto...\n" ;
                  (module Toto : S)
                end : S)
```

Les ingrédients :

- `(val ... : S)` convertit une valeur en module ;
- `(module T : S)` convertit un module en valeur.

Modules récursifs

Un type d'arbres dont les enfants d'un nœud forment un ensemble ?

Modules récursifs

Un type d'arbres dont les enfants d'un nœud forment un ensemble ?

```
module rec A : sig
  type t = Leaf of string | Node of ASet.t
  val compare: t -> t -> int
end = struct
  type t = Leaf of string | Node of ASet.t
  let compare t1 t2 =
    match (t1, t2) with
      | (Leaf s1, Leaf s2) -> Pervasives.compare s1 s2
      | (Leaf _, Node _) -> 1
      | (Node _, Leaf _) -> -1
      | (Node n1, Node n2) -> ASet.compare n1 n2
end

and ASet : Set.S with type elt = A.t = Set.Make(A)
```

Modules récursifs

Un type d'arbres dont les enfants d'un nœud forment un ensemble ?

```
module rec A : sig
  type t = Leaf of string | Node of ASet.t
  val compare: t -> t -> int
end = struct
  type t = Leaf of string | Node of ASet.t
  let compare t1 t2 =
    match (t1, t2) with
      | (Leaf s1, Leaf s2) -> Pervasives.compare s1 s2
      | (Leaf _, Node _) -> 1
      | (Node _, Leaf _) -> -1
      | (Node n1, Node n2) -> ASet.compare n1 n2
end

and ASet : Set.S with type elt = A.t = Set.Make(A)
```

Condition : les modules doivent être safe,
c'est à dire (en gros) ne définir que des fonctions.

Compilation des modules

Pour se faire une idée de ce qu'il se passe sous le capot...

Démo avec `dlambda.ml`.

Conclusion

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

Résumé

- Module = paquet de déclarations de types et valeurs
- Signature = type d'un module, permet d'en cacher une partie
- Foncteur = fonction entre modules
- Type abstrait = dont l'implémentation est cachée
- Le mot clé `with` pour changer un type abstrait dans une signature, le partager entre plusieurs modules

Références

- Manuel officiel `http://caml.inria.fr/pub/docs/manual-ocaml-4.01/index.html`
- Le livre O'Reilly sur OCaml

`http://www.pps.univ-paris-diderot.fr/Livres/ora/DA-OCAML/`