Les modules d'Objective Caml

D.E.A. Programmation: Sémantique, Preuves et Langages

Xavier Leroy
INRIA Rocquencourt
`Xavier.Leroy@inria.fr`
`http://caml.inria.fr/`

La programmation modulaire

- Découpage du programme en morceaux compilables indépendamment

  Rendre les gros programmes compilables

- Donner de la structure au programme et faciliter la réutilisation de certaines parties

  Rendre les gros programmes compréhensibles

Les modules de ML

(MacQueen, Milner, Harper)

- Un petit langage fonctionnel typé.
- Manipule des collections de définitions (de valeurs, de types) du langage de base.
- Leurs types: des collections de déclarations/spécifications (types des valeurs, déclarations des types).
- Modules emboîtés (en guise de structure de données).
- Fonctions (foncteurs) et application de fonctions.

Largement indépendant du langage de base.

Modules de base

Les *structures*: collections de définitions.

`struct` *définition* ... *définition* `end`

Les définitions correspondent aux phrases du langage de base:

définitions de valeurs et de fonctions	`let` $x = \ldots$
définitions de types	`type` $t = \ldots$
définitions d'exceptions	`exception` E [`of` ...]
définitions de classes	`class` $C = \ldots$
définitions de sous-modules	`module` $X = \ldots$
définitions de type de module	`module type` $S = \ldots$

Nommage d'un module

On donne un nom à un module à l'aide de la liaison `module`.

```
module S = struct ... end
```

Exemple:

```
module S =
  struct
    type t = int
    let x = 0
    let f x = x+1
  end
```

Utilisation d'un module

On fait référence aux composantes d'un modules avec la notation pointée *module*.*composante*

```
... (S.f S.x : S.t) ...
```

Autre possibilité: la directive `open` *module* permet d'omettre le préfixe et le point:

```
open S
... (f x : t) ...
```

Modules emboîtés

Un module peut être composante d'un autre module:

```
module T =
  struct
    module R = struct let x = 0 end
    let y = R.x + 1
  end
```

La notation pointée et le `open` s'étendent naturellement:

```
T.R.x
```

```
open T.R
```

Les types des modules de base

Les *signatures*: collections de spécifications (de types).

`sig` *spécification* ... *spécification* `end`

spécification de valeurs	`val` $x : \sigma$
spécification de types, abstraites	`type` t
spécification de types, manifestes	`type` $t = \tau$
spécification d'exceptions	`exception` E
spécification de classes	`class` $C : \ldots$
spécification de sous-modules	`module` $X : M$
spécification de type de module	`module type` $S\ [\ = M]$

Nommage d'un type de module: par la liaison `module type`

```
module type SIG = sig ... end
```

Restriction par une signature

Pour chaque définition de structure, le système infère une signature qui est la plus précise: toutes les composantes de la structure sont visibles, avec leur type le plus général.

```
struct                            sig
    type t = int                      type t = int
    let f x = x                       val f : 'a -> 'a
end                               end
```

Pour cacher des composantes, ou pour restreindre leur type, on utilise une contrainte de signature ($structure : signature$).

La contrainte vérifie que la structure satisfait la signature.

Elle s'évalue en la structure vue avec la signature restreinte.

Exemple de restriction

```
module S =
  (struct
     type t = int
     let x = 1
     let y = x + 1
   end :
   sig
     type t
     val y : t
   end)
```

`S.x;;` *erreur ''S.x is unbound''*

`S.y + 1;;` *erreur ''S.y is not of type int''*

Vues multiples d'un même module

```
module Stamp =
  struct
    type t = int
    let new () = ...
    let equal s1 s2 = (s1 = s2)
  end

module type ABSTRSTAMP =
  sig
    type t
    val equal: t -> t -> bool
  end

module AbstrStamp = (Stamp : ABSTRSTAMP)
```

Les utilisateurs de `AbstrStamp` peuvent seulement tester l'égalité de deux stamps; ceux de `Stamp` peuvent aussi créer de nouveaux stamps et les traiter comme des entiers (affichage).

Pas de duplication de code entre `AbstrStamp` et `Stamp`.

Modules et compilation séparée

Une unité de compilation A se compose de deux fichiers:

- Le fichier d'implémentation A`.ml`:
  une suite de définitions
  semblable à l'intérieur de **`struct...end`**

- Le fichier d'interface A`.mli` (optionnel):
  une suite de spécifications
  semblable à l'intérieur de **`sig...end`**

Une autre unité de compilation B peut faire référence à A comme si c'était une structure, en utilisant la notation pointée A`.x` ou bien en faisant `open` A.

Compilation séparée d'un programme

Fichiers sources: `a.ml`, `a.mli`, `b.ml`

Étapes de compilation:

`ocamlc -c a.mli`	(*compile l'interface de A, crée a.cmi*)
`ocamlc -c a.ml`	(*compile l'implémentation de A, crée a.cmo*)
`ocamlc -c b.ml`	(*compile l'implémentation de B, crée b.cmo*)
`ocamlc -o monprog a.cmo b.cmo`	(*édition de liens finale*)

Le programme se comporte comme le code monolithique suivant:

`module A = (struct` *contenu de a.ml* `end : sig` *contenu de a.mli* `end)`
`module B = struct` *contenu de b.ml* `end`

L'ordre des définitions de modules correspond à l'ordre des fichiers objets `.cmo` sur la ligne de commande de l'éditeur de liens.

Modules paramétrés

Un *foncteur* est une fonction des modules dans les modules:

`functor(`*S*`:` *signature*`) ->` *module*

Le *module* (corps du foncteur) est explicitement paramétré par le paramètre de module *S*. Il fait référence aux composantes de son paramètre avec la notation pointée.

```
module T = functor(S : SIG) ->
  struct
    type u = S.t * S.t
    let y = S.g(S.x)
  end
```

Application de foncteur

On ne peut pas directement accéder dans `T`. Il faut au préalable l'appliquer explicitement à une implémentation de la signature `SIG` (tout comme on applique une fonction normale).

```
module T1 = T(S1)
module T2 = T(S2)
```

`T1`, `T2` s'utilisent alors comme des structures normales:

```
(T1.y : T2.u)
```

`T1` et `T2` partagent entièrement leur code.

Exemple long: les polynômes sur un anneau quelconque

On va implémenter les opérations usuelles sur les polynômes à coefficients dans un anneau quelconque, donné en paramètre.

Le paramètre anneau est une structure ayant la signature suivante:

```
module type ANNEAU =
  sig
    type t              (* type des éléments de l'anneau *)
    val zero: t
    val un: t
    val plus: t->t->t   (* somme *)
    val opp: t->t       (* opposé *)
    val mult: t->t->t   (* produit *)
  end
```

Polynômes pleins

```
module PolyPlein = functor (Anneau : ANNEAU) ->
  struct
    module A = Anneau
    type t = A.t array                  (* le type des polynômes *)
    let zero = [| A.zero |]             (* le polynôme nul *)
    let un = [| A.un |]                 (* le polynôme 1 *)
    let monôme c n =                    (* le monôme c.X^n *)
      let p = Array.create (n+1) A.zero in
      p.(n) <- c; p
    let coeff n p = p.(n)               (* renvoie le coeff de X^n dans p *)
    let degré p =                       (* renvoie le degré de p *)
      let d = ref (Array.length p - 1) in
      while !d >= 0 && p.(!d) = A.zero do decr d done;
      !d
    let plus p1 p2 =                    (* addition *)
      let r = Array.create (max (Array.length p1) (Array.length p2)) A.zero in
      for i = 0 to Array.length p1 - 1 do r.(i) <- p1.(i) done;
      for i = 0 to Array.length p2 - 1 do r.(i) <- A.plus r.(i) p2.(i) done;
      r
    let opp p = ...                     (* opposé *)
    let mult p1 p2 = ...                (* multiplication *)
  end
```

Utilisation du foncteur des polynômes

Polynômes à coefficients entiers:

```
module Entiers =
  struct
    type t = int
    let zero = 0
    let un = 1
    let plus x y = x + y
    let opp x = -x
    let mult x y = x * y
  end
module PolyEntiers = PolyPlein(Entiers)
```

Polynômes à coefficients entiers à deux variables = polynômes de polynômes d'entiers:

```
module Poly2Entiers = PolyPlein(PolyEntiers)
```

C'est correct car la structure résultat de `PolyPlein` satisfait la signature `ANNEAU`.

Polynômes creux

```
module PolyCreux = functor (Anneau : ANNEAU) ->
  struct
    module A = Anneau
    type t = (A.t * int) list             (* le type des polynômes *)
    let zero = []                         (* le polynôme nul *)
    let un = [(A.un, 0)]                  (* le polynôme 1 *)
    let monôme c n = [(c, n)]             (* le monôme c.X^n *)
    let rec degré = function
        [] -> 0
      | [(coeff, d)] -> d
      | (coeff, d) :: reste -> degre reste
    let rec coeff n p =
      match p with
        [] -> A.zero
      | (c, d) :: reste ->
          if d = n then c else
          if d > n then A.zero else coeff n reste
    let plus p1 p2 =  ...                 (* addition *)
    let opp p = ...                       (* opposé *)
    let mult p1 p2 = ...                  (* multiplication *)
  end
```

Indépendance vis-à-vis de la représentation

`PolyPlein` et `PolyCreux` implémentent la même interface et sont essentiellement interchangeables:

```
module PolyEntiers = PolyCreux(Entiers)
module Poly2Entiers = PolyCreux(PolyEntiers)
```

Mais pas totalement interchangeables: le type de représentation des polynômes reste apparent.

```
module P1 = PolyPlein(Entiers);;          (* P1.t = int array *)
P1.degré [| 1;2;3;4 |];;                  (* - : int = 3 *)

module P2 = PolyCreux(Entiers);;          (* P2.t = (int * int) list *)
P2.degré [| 1;2;3;4 |];;                  (* erreur de type *)
```

Cacher la représentation des polynômes

On applique une contrainte de signature au résultat des foncteurs `PolyPlein` et `PolyCreux` pour rendre abstrait le type `t`.

Premier essai:

```
module type POLYNÔME =
  sig
    module A : ANNEAU
    type t
    val zero: t                          val un: t
    val monôme: A.t -> int -> t          val degré: t -> int
    val coeff: int -> t -> A.t           val opp: t -> t
    val plus: t -> t -> t                val mult: t -> t -> t
  end
module PolyPlein = functor (Anneau: ANNEAU) -> (struct ... end : POLYNÔME)
```

Problème: on a aussi rendu abstrait le type `A.t` des coefficients, ce qui rend le résultat de `PolyPlein` inutilisable:

```
module PolyEntiers = PolyPlein(Entiers);;
PolyEntiers.monôme 15 2;;
```

This expression has type int but is used with type PolyEntiers.A.t

La bonne solution: utiliser un type manifeste pour conserver l'identité du type `A.t` du résultat:

```
module PolyPlein = functor (Anneau: ANNEAU) ->
  (struct ... end : POLYNÔME with type A.t = Anneau.t)
```

Dès lors:

```
module PolyEntiers = PolyPlein(Entiers)
```

On a `PolyEntiers.t` abstrait,
mais `PolyEntiers.A.t` = `Entiers.t` = `int` comme désiré.

La notation `with`

Permet d'ajouter des égalités de types dans une signature existante.

L'expression `POLYNÔME with type A.t = Anneau.t` est une abréviation pour la signature

```
sig
  module A : sig type t = Anneau.t
                 val zero: t   val un: t
                 val plus: t->t->t
                 val opp: t->t
                 val mult: t->t->t  end
  type t
  val zero: t
  val un: t
  val monôme: A.t -> int -> t
  val plus: t -> t -> t
  val opp: t -> t
  val mult: t -> t -> t
  val degré: t -> int
  val coeff: int -> t -> A.t
end
```

Modules vs. polymorphisme

Les polynômes s'expriment aussi sans foncteurs, avec juste du polymorphisme à la ML:

```
type 'a polynôme
type 'a anneau =
  { zero: 'a; un: 'a; plus: 'a->'a->'a; opp: 'a->'a; mult: 'a->'a->'a }
val zero: 'a anneau -> 'a polynôme
val un: 'a anneau -> 'a polynôme
val monôme: 'a anneau -> 'a -> int -> 'a polynôme
val plus: 'a anneau -> 'a polynôme -> 'a polynôme -> 'a polynôme
val opp: 'a anneau -> 'a polynôme -> 'a polynôme
val mult: 'a anneau -> 'a polynôme -> 'a polynôme -> 'a polynôme
val degré: 'a anneau -> 'a polynôme -> int
val coeff: 'a anneau -> int -> 'a polynôme -> 'a
```

Premier problème: il faut passer les opérations de l'anneau en argument supplémentaire aux opérations sur les polynômes.

Foncteurs = paramétrisation en masse
Langage de base = paramétrisation individuelle

Deuxième problème: le type des polynômes ne reflète que le type du support de l'anneau, pas l'anneau lui-même.

```
let entiers =
  { zero = 0; un = 1; plus = fun x y -> x + y; ...}
let z_sur_3z =
  { zero = 0; un = 1; plus = fun x y -> (x + y) mod 3; ...}
let p1 = monôme entiers 15 2
let p2 = monôme z_sur_3z 2 3
let p3 = mult entiers p1 p2
```

Il est absurde de mélanger `p1` et `p2`, et pourtant ils ont le même type `int polynôme`.

La formulation à base de foncteurs détecte cette erreur par typage statique:

```
module Z_sur_3Z =
  struct type t = int  let zero = 0  let un = 1  let plus x y = (x+y) mod 3 ... end
module PolyEntiers = PolyPlein(Entiers)
module PolyZ3Z = PolyPlein(Z_sur_3Z)
```

Les types `PolyEntiers.t` et `PolyZ3Z.t` sont tous deux abstraits, donc a priori incompatibles:

```
PolyZ3Z.degré PolyEntiers.un
```

This expression has type PolyEntiers.t
but is here used with type PolyZ3Z.t

Le polymorphisme du langage ML de base:

types paramétrés:	type	$\longrightarrow$	type
fonctions:	valeurs	$\longrightarrow$	valeurs

Le langage de modules:

foncteurs:	types interprétés par des valeurs	$\longrightarrow$	types interprétés par des valeurs

Cas particulier: $\mathbf{Z}/2\mathbf{Z}$

Autre point fort de la solution à base de foncteurs: on peut fournir des implémentations directes (sans passer par le foncteur) pour certains types.

Exemple: pour implémenter efficacement $\mathbf{Z}/2\mathbf{Z}[X]$, on fournit une structure `Z2Z` à base de tableaux de bits avec la même interface que le résultat du foncteur `PolyPlein`:

```
module Poly_Z2Z : (POLYNÔME with type A.t = int) =
  struct
    module A = Z_sur_2Z
    type t = bitvect
    let add = bitvect_xor
    let mult = bitvect_mult
    let opp = id
    ...
  end
```

Le reste du programme ne change pas.

Modules vs. objets

Un certain recouvrement dans le cas des types abstraits modifiables en place et avec opérations unaires:

```
module Hashtbl:
  sig
    type ('a, 'b) t
    val empty: int -> ('a, 'b) t
    val add: ('a, 'b) t -> 'a -> 'b -> unit
    val find: ('a, 'b) t -> 'a -> 'b
  end

class ['a, 'b] hashtbl : int ->
  object
    method add: 'a -> 'b -> unit
    method find: 'a -> 'b
  end
```

Foncteurs d'extension = un cas simple d'héritage.

Modules:

- Opérations n-aires.
- Plusieurs types abstraits reliés.
- Foncteurs généraux.

Objets:

- Sous-typage entre valeurs.
- Liaison tardive (récursion ouverte).