MODULES ET FONCTEURS EN OCAML

JEAN-VINCENT LODDO
OCTOBRE 2009

1. Introduction

Les modules en ocaml représentent le mécanisme d'abstraction ou d'encapsulation typique des langages de programmation modernes. L'idée est qu'une partie du programme, qu'on appelle *module*, "offre" un ensemble d'éléments (types, valeurs, exceptions) pour une éventuelle utilisation ailleurs dans le reste du programme. La partie "offerte" est <u>résumée</u> par ce que l'on appelle *signature*. L'implémentation du module respecte sa signature, tout en cachant éventuellement plusieurs autres éléments (d'autres définitions de types, de valeurs, d'exceptions ou sous-modules).

2. Syntaxe

2.1. Signatures. Une *expression de signature* est représentée par la lettre σ.

$$\sigma ::= \texttt{sig}\ \theta_1\ ..\ \theta_n\ \texttt{end} \mid \texttt{X}$$

La première production de la syntaxe indique une signature composée des éléments θ_1 .. θ_n. La seconde permet d'exprimer la signature par son *nom* (**X** est un identificateur commençant par une majuscule). Dans le premier cas, chaque symbole θ représente la signature d'un élément spécifique, c'est-à-dire un type, une exception ou une valeur (par exemple une fonction) :

$$\begin{aligned}\theta ::= &\ \texttt{type}\ \ [\texttt{a}_1 \mid (\texttt{a}_1,..,\texttt{a}_n)]\ \ \texttt{x}\ \ [\texttt{= t}]\\ \mid &\ \texttt{exception X [of t]}\\ \mid &\ \texttt{val x : t}\end{aligned}$$

Remarque : dans la définition de la signature d'un type, seul le nom est obligatoire, alors que sa définition est facultative (elle sera certainement présente dans l'implémentation mais peut être anticipée ici). On attribue un nom à une signature au top-level :

$$\texttt{l} ::= \texttt{module type X = } \sigma\ \ \texttt{;;}$$

Example. Voici une signature pour un module de manipulation de matrices. Au top-level on écrit :

```
module type Module_Matrices =
 sig
   type 'a matrice
   val transpose : 'a matrice -> 'a matrice
   val sel : 'a matrice -> int -> int -> 'a
   val row : 'a matrice -> int -> 'a matrice
   val col : 'a matrice -> int -> 'a matrice
 end;;
```


2.2. Implémentations. Par rapport à une signature, la syntaxe de l'implémentation (ou module) est similaire, sauf que dans l'implémentation on spécifie toute l'information concernant les éléments présents (la signature, en effet, n'est qu'une sorte de résumé). Une *expression de module* est représentée par la lettre ρ.

$$\rho ::= \texttt{struct}\ \texttt{l}_1\ ..\ \texttt{l}_n\ \texttt{end} \mid \texttt{X}$$

La première production de la syntaxe indique un module composé des éléments définis par les phrases de top-level $\mathbf{l}_1$ .. $\mathbf{l}_n$. La seconde permet d'évoquer le module identifié par (dont le *nom* est) **X**.

2.2.1. *Définition de modules ordinaires.* On attribue un nom à un module au top-level :

$$\texttt{l} ::= \texttt{module X [: } \sigma\texttt{] = } \rho\ \ \texttt{;;}$$

Si la signature σ est spécifié, le module est forcé présenter tous les éléments contenus dans la signature. Sinon, la signature implicite du module est constituée de <u>tous</u> les éléments présent dans le module (autrement dit, par défaut aucun élément est caché à l'extérieur).

Example. Dans l'exemple précédent, nous ajoutons la définition d'un module "Matrices" de manipulation des matrices (forcé avoir la signature "Module_Matrices") :

```
module Matrices : Module_Matrices = struct
   type 'a matrice = int * int -> 'a
   exception OutOfRange
   let transpose = function m -> function (i,j) -> m (j,i)
   let sel = fun m i j -> m (i,j)
   let row = fun m x -> fun (i,j) -> if i=1 then m (x,j)
                                                else raise OutOfRange
   let col = fun m y -> fun (i,j) -> if j=1 then m (j,y)
                                                else raise OutOfRange
end;;
```

2.3. Expressions et types. L'introduction des modules ajoute une production à la syntaxe des expressions et à celle des types. En effet, les modules regroupent des types et des valeurs accessibles par un nom à l'intérieur du module. On pourra évoquer ces types et ces valeurs en préfixant le nom de l'élément (type ou valeur) par le nom du module, les deux séparés par un point :

$$\begin{aligned}\texttt{t} &::= \texttt{X.x}\\ \texttt{e} &::= \texttt{X.x}\end{aligned}$$

Example. Dans l'exemple précédent, nous pouvons interroger l'interpréteur sur la valeur suivante :

```
Matrices.row;;
```

Il donnera la réponse :

```
# - : 'a Matrices.matrice -> int -> 'a Matrices.matrice = <fun>
```

En revanche, si on essaye de soulever l'exception `Matrices.OutOfRange` :

```
raise Matrices.OutOfRange;;
```

il répondra :

```
Unbound constructor Matrices.OutOfRange
```

l'exception n'étant pas incluse dans la signature, elle doit être cachée à l'utilisateur du module.

2.4. Ouvrir un module. Pour éviter de devoir spécifier à chaque fois le nom du module en préfixe, on peut "ouvrir" un module. Au top-level on écrit :

```
l ::= open X ;;
```

Cela permet d'ajouter les identificateurs (noms) du modules à l'environnement d'exécution *comme s'ils* avait étés définis un par un au top-level. Tous les noms du modules sont ainsi directement accessibles au top-level, sans plus besoin de spécifier le préfixe.

3. Foncteurs

Il est possible de définir des modules paramétrés par rapport à... d'autres modules.

3.1. Abstraction. Les expressions de module ρ sont enrichies par une production qui permet l'abstraction du module qu'on écrit par rapport à un autre qui sera donné par la suite (par application) :

$$\rho ::= \texttt{functor (X :}\sigma\texttt{) ->}\ \rho$$

Comme pour les fonctions, un raccourci syntaxique permet d'abréger l'expression d'un foncteur avec plusieurs paramètres :

$$\texttt{l} ::= \texttt{module X [: }\sigma\texttt{] (X}_1\texttt{ :}\sigma_1\texttt{)..(X}_n\texttt{ :}\sigma_n\texttt{) = }\rho\texttt{ ;;}$$

est un raccourci pour :

$$\texttt{module X [: }\sigma\texttt{] = functor (X}_1\texttt{ :}\sigma_1\texttt{) -> .. -> functor (X}_n\texttt{ :}\sigma_n\texttt{) -> }\rho\texttt{ ;;}$$

On remarque que si la signature du foncteur défini est optionnelle, celle des arguments doit, en revanche, être spécifiée obligatoirement (pour des raisons liées au typage).

3.2. Application. Pour utiliser une "instance" particulière du module paramétré (ou "foncteur") il suffira de lui donner un nom au top level et de la définir en tant qu'application du foncteur :

```
module PileInt64 = Pile (Int64);;
```

Ceci signifie qu'une implémentation ρ peut être aussi spécifiée comme étant une application[1] d'un foncteur à des argument de type modules :

$$\rho ::= \texttt{X (X}_1\texttt{)..(X}_n\texttt{)}$$

3.3. Exemple. La librairie standard contient un module, appelé `Set`, pour la gestion des ensembles de type arbitraire. Tout en voulant une librairie polymorphe, pour que l'implémentation de cette structure de données soit efficace, on demande cependant que le type `t` des éléments soit comparables. Il faut donc qu'il existe une fonction de comparaison :

```
compare : t -> t -> int
```

telle que (`compare x y`) est zéro si les éléments `x` et `y` sont égaux, strictement négative si `x` est plus petit que `y`, et strictement positive si `x` est plus grand que `y`. La signature du module standard `Set` est la suivante :

[1] qui est effectuée statiquement


```
module Set : sig
  module type OrderedType = sig type t val compare : t -> t -> int end
  module type S = sig
    type elt
    type t
    val empty : t
    val is_empty : t -> bool
    val mem : elt -> t -> bool
    val add : elt -> t -> t
    val singleton : elt -> t
    val remove : elt -> t -> t
    val union : t -> t -> t
    val inter : t -> t -> t
    val diff : t -> t -> t
    val compare : t -> t -> int
    val equal : t -> t -> bool
    val subset : t -> t -> bool
    val iter : (elt -> unit) -> t -> unit
    val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a
    val for_all : (elt -> bool) -> t -> bool
    val exists : (elt -> bool) -> t -> bool
    val filter : (elt -> bool) -> t -> t
    val partition : (elt -> bool) -> t -> t * t
    val cardinal : t -> int
    val elements : t -> elt list
    val min_elt : t -> elt
    val max_elt : t -> elt
    val choose : t -> elt
    val split : elt -> t -> t * bool * t
end
module Make : functor (Ord : OrderedType) -> S with type elt = Ord.t
end
```

Pour créer la structure qui nous convient on devra définir un module de type `OrderedType` et le donner en argument au foncteur `Set.Make`, par exemple :

```
module String_set =
  Set.Make (struct  type t = string  let compare = Pervasives.compare  end)
;;
```

4. Modules et fichiers de compilation séparés

On peut “placer” la définition d’une signature et celle d’un module à l’intérieur de deux fichiers séparés du fichier contenant le programme. On appelle “unité de compilation” un couple de fichiers (`x.mli`, `x.ml`) où `x` est un nom quelconque (mais pas anodin) :

`x.mli` : contient des phrases du type θ_1 .. θ_n (exactement comme dans la définition d’une signature σ)

`x.ml` : contient des phrases du type $\mathbf{l}_1$ **;;** .. $\mathbf{l}_n$ **;;** (tout comme dans la définition d’une implémentation ρ, sauf que les doubles point-virgules remplacent les blancs)

Cela définit implicitement un module dont le nom `X` est obtenu en mettant en majuscule la première lettre de `x`. Par exemple, le couple (`formules.mli`, `formules.ml`) définit un module dont le nom est `Formules`.

Si le fichier contenant la signature n’est pas fourni, la signature du module est (comme d’habitude) implicitement définie par l’ensemble des éléments implémentés dans le module. Les deux fichiers doivent être compilé séparément :

– `ocamlc x.mli`
  produit le fichier `x.cmi` (Compiled Module Interface)
– `ocamlc x.ml`
  produit le fichier `x.cmo` (Compiled Module Object)

Après avoir compilé de cette façon plusieurs modules $\mathtt{x}_1$`.cmo`,.., $\mathtt{x}_n$`.cmo`, nous pouvons compiler un programme les utilisant par la ligne :

– `ocamlc -o program` $\mathtt{x}_1$`.cmo` $\mathtt{x}_n$`.cmo program.ml`

Un module compilé peut être aussi utilisé dans l’interpréteur. Il suffit d’utiliser la directive :

```
#load “x.cmo”;;
```

puis, si on veut l’ouvrir :

```
open X;;
```