Programmation – TD2 : Modules et foncteurs

{ Jeremie.Detrey, Emmanuel.Jeandel } @ens-lyon.fr

6/7 octobre 2004

0 Le système de modules de OCaml

0.1 Modules ? Qu'es aco ?

Un module est un ensemble de définitions de types, de variables, de fonctions, etc... On peut le voir comme un programme, auquel on donne un nom. Par exemple, un module `MyList` permettant de manipuler des listes :

```
module MyList =
struct
  exception EmptyList              (* <- notez l'absence de ;; *)

  let hd l = match l with
    | [] -> raise EmptyList
    | x::_ -> x                    (* ici aussi *)

  let tl l = match l with
    | [] -> raise EmptyList
    | _::q -> q

  let rec length l =
    try let q = tl l in
      1 + (length q)
    with
      | EmptyList -> 0
end;;
```

Si l'on donne ce bout de code à l'interpréteur Caml, celui-ci répond :

```
module MyList :
  sig
    exception EmptyList
    val hd : 'a list -> 'a
    val tl : 'a list -> 'a list
    val length : 'a list -> int
  end
```

Cela signifie que `MyList` a bien été défini comme module, et sa *signature* (aussi appelée *interface*) nous renseigne sur les divers éléments définis par le module. Ainsi, dans notre cas, nous avons :

- `EmptyList` : une exception, qui sera levée en cas d'erreur due à une liste vide ;
- `hd : 'a list -> 'a` : une fonction renvoyant le premier élément d'une liste, ou levant une exception `EmptyList` s'il s'agit de la liste vide ;
- `tl : 'a list -> 'a list` : une fonction renvoyant la queue d'une liste, ou levant une exception `EmptyList` s'il s'agit de la liste vide ;
- `length : 'a list -> int` : une fonction renvoyant la longueur d'une liste (d'une manière certes cochonne, mais c'est pour l'exemple ; pour ceux qui ne sont pas familiers avec les `try ... with ...`, essayez de comprendre comment cette fonction marche, ou demandez au TDman).

On peut alors utiliser les fonctions du module ainsi défini, en faisant par exemple :

```
MyList.length (MyList.tl [1; 6; 6; 4]);;
```

qui renvoie bien la valeur `3`.

Supposons maintenant que nous souhaitons "cacher" les fonctions `hd` et `tl`, c'est-à-dire qu'elles soient accessibles depuis le module `MyList`, mais pas depuis l'extérieur. Pour cela, on peut spécifier la signature d'un module, en écrivant par exemple :

```
module MyList =
  (struct
     (* tout comme avant *)
   end : sig
     exception EmptyList
     val length : 'a list -> int
   end);;
```

On peut vérifier que l'interpréteur Caml attribue bien la signature spécifiée au module `MyList`. Tenter d'évaluer `MyList.tl` renvoie désormais une erreur `Unbound value MyList.tl` : mission accomplie, cette fonction n'est pas accessible depuis l'extérieur, alors que `length` y l'appelle sans problème.

Dans le cas où l'on veut construire plusieurs modules de signatures identiques, on peut définir cette signature et lui donner un nom. Ainsi, pour l'exemple précédent, on pouvait aussi écrire :

```
module type LIST =
sig
  exception EmptyList
  val length : 'a list -> int
end;;

module MyList : LIST =
struct
  (* tout comme avant *)
end;;
```

Attention : notez bien les majuscules / minuscules des conventions de nommage (`valeur`, `Module`, `SIGNATURE`).

0.2 Modules et fichiers

Il existe un moyen très simple de faire des modules en utilisant plusieurs fichiers. Ainsi, en revenant à l'exemple précédent, on peut écrire les deux fichiers `myList.ml` et `myList.mli` suivants :

```
exception EmptyList

let hd l = match l with
  | [] -> raise EmptyList
  | x::_ -> x

let tl l = match l with
  | [] -> raise EmptyList
  | _::q -> q

let rec length l =
  try let q = tl l in
    1 + (length q)
  with
    | EmptyList -> 0
```

`myList.ml`

```
exception EmptyList
val length : 'a list -> int
```

`myList.mli`

Ainsi, le fichier `toto.ml` fournit le code du module `Toto`, et le fichier optionnel `toto.mli` fournit sa signature. Cela revient à écrire :

```
module Toto =
  (struct
     (* contenu de toto.ml *)
   end : sig
     (* contenu de toto.mli *)
   end);;
```

Remarque : pour les férus de C, on peut voir les fichiers `.ml` comme les `.c`, et les `.mli` comme les `.h` correspondants.

0.3 Compilation séparée

La compilation de programmes OCaml fait intervenir un bon nombre de types de fichiers, dont voici une courte description :

- `toto.ml` : code du module `Toto` ;
- `toto.mli` : interface du module `Toto` ;
- `toto.cmo` : fichier résultant de la compilation de `toto.ml` ;
- `toto.cmi` : fichier d'interface résultant de la compilation de `toto.mli` ;
- `biblio.cma` : collection de fichiers `.cmo` (utile pour créér une bibliothèque) ;
- `prog` : l'exécutable final (que nous appelons `prog` ici, mais peut avoir n'importe quel nom).

Pour passer d'un type de fichier à un autre en utilisant le compilateur `ocamlc`, voici les diverses commandes :

- de `toto.mli` vers `toto.cmi` : `ocamlc -c toto.mli` ;
- de `toto.ml` vers `toto.cmo` : `ocamlc -c toto.ml` (attention, si le fichier `toto.mli` existe aussi, il faut le compiler d'abord) ;
- de `titi.cmo`, `tata.cmo` et `toto.cmo` vers `biblio.cma` :
  `ocamlc -a titi.cmo tata.cmo toto.cmo -o biblio.cma` ;
- de `titi.cmo`, `tata.cmo`, `toto.cmo` et `biblio1.cma`, `biblio2.cma` vers `prog` :
  `ocamlc titi.cmo tata.cmo toto.cmo biblio1.cma biblio2.cma -o prog`.

Pour essayer tout ça sur un vrai exemple, téléchargez les fichiers `myList.ml`, `myList.mli` et `test.ml` sur `http://perso.ens-lyon.fr/jeremie.detrey/04_prog/`.

Pour les compiler, faites dans l'ordre :

```
ocamlc -c myList.mli                # génération de myList.cmi,
                                    #   nécessaire à myList.cmo

ocamlc -c myList.ml                 # génération de myList.cmo

ocamlc -c test.ml                   # génération de test.cmo,
                                    #   ainsi que de test.cmi
                                    #   (car il n'y a pas de test.mli)

ocamlc myList.cmo test.cmo -o test  # génération du programme final
```

Et voilà ! Vous venez de produire un fichier `test` que vous pouvez exécuter. Il est censé afficher `3`. Amusez-vous à le modifier un peu pour vous familiariser avec tout ça.

Remarque : en utilisant `ocamlc`, le programme produit n'est pas en assembleur (langage machine), mais en *byte-code* OCaml, une sorte d'assembleur qui doit être interprété par `ocamlrun` pour être exécuté. Pour produire du code natif (plus lents à compiler mais plus rapides à l'exécution), il faut utiliser `ocamlopt` dans la chaîne de compilation, à la place de `ocamlc`. `ocamlopt` s'emploie de manière identique à `ocamlc`, mais produit des fichiers `.cmx` et `.cmxa` au lieu des `.cmo` et `.cma` de `ocamlc`.

0.4 Le tout avec un `Makefile`

Pour vous éviter de vous embrouiller avec toutes ces commandes, et pour gérer correctement les dépendances entre les fichiers, on peut utiliser un `Makefile`. Il s'agit d'un fichier contenant toutes les dépendances entre les fichiers, ainsi que les commandes pour compiler chacun d'entre eux. Allez lire `http://www.gnu.org/software/make/`si vous voulez en savoir plus. Pour l'instant, allez sur `http://perso.ens-lyon.fr/jeremie.detrey/04_prog/` et téléchargez le `Makefile` dans le même répertoire que les fichiers précédents. Effacez tous les `.cmo`, `.cmi` et autres, puis tapez `make` pour exécuter le `Makefile`. Lisez ce fichier, et essayez de voir comment il marche dans les grandes lignes.

0.5 Utilisation d'un module compilé dans le *top-level* OCaml

Vous pouvez faire `#load "toto.cmo";;` dans le *top-level* OCaml pour charger un module compilé. Vous aurez alors accès au module `Toto` correspondant.

1 Modules z'et foncteurs

1.1 Compteur

On veut construire un compteur entier tout simple, doté de 3 fonctions pour le manipuler.

1. Déclarez ce compteur par `let c = ref 0` puis écrivez les fonctions :
`clear : unit -> unit` qui remet le compteur à zéro ;
`incr : unit -> unit` qui incrémente le compteur ;
`value : unit -> int` qui renvoie la valeur actuelle du compteur.

Ce n'est pas encore au point, car de cette façon n'importe qui peut modifier la valeur du compteur en faisant `c := 1664`, par exemple.

2. En n'utilisant *que* des constructions de type `let ... in ...`, trouvez un moyen de protéger `c`.

La construction précédente, vous l'avez compris, est pénible. Heureusement, les modules sont là pour vous sauver la vie !

3. Faites un module `Counter` proposant les trois fonctions `clear`, `incr` et `value` mais protégeant `c`. C'est-y pas plus joli comme ça quand même ?

4. Utilisons un peu ce formidable compteur : comptez les appels récursifs à la fonction de Fibonacci.

1.2 Structures algébriques : les anneaux unitaires

Le but de cet exercice est de définir des modules permettant de manipuler des anneaux unitaires divers et variés. Nous voulons donc que nos modules aient la signature suivante :

```
module type RING =
sig
  type t
  val zero  : t               (* 0             *)
  val one   : t               (* 1             *)
  val add   : t -> t -> t     (* x y ->  x + y *)
  val opp   : t -> t          (* x   -> -x     *)
  val mult  : t -> t -> t     (* x y ->  x * y *)
  val print : t -> unit
end;;
```

5. Écrivez un module `ZRing` de signature `RING` correspondant à l'anneau des entiers relatifs $\mathbb{Z}$:

```
module ZRing : (RING with type t = int) =
struct
  type t = int
  (* . . . *)
end;;
```

Remarque : la syntaxe `with type t = int` permet de préciser dans la signature du module `ZRing` que le type `t` (déclaré abstrait dans `RING`) sera en fait `int`. Ce n'est pas essentiel, mais vous permettra d'écrire `ZRing.mult 32 (ZRing.add ZRing.one 51)`, par exemple.

6. Écrivez un module `BRing : (RING with type t = bool)` correspondant à l'anneau des booléens $\mathbb{B} = \mathbb{Z}/2\mathbb{Z}$.
7. Écrivez enfin un module `CRing : (RING with type t = float * float)` correspondant à l'anneau des complexes $\mathbb{C}$.

Tout va bien jusqu'ici ? Parfait, passons donc aux foncteurs.

8. On veut écrire un foncteur `SeqRing` qui prend un module `A` de signature `RING` et renvoie un autre module de signature `RING` correspondant à l'anneau des suites d'éléments de `A`. Pour cela, on représentera une suite $(u_n) \in \mathtt{A}^{\mathbb{N}}$ par une fonction `u : int -> A.t`.
   Complétez donc le modèle de foncteur suivant :

```
module SeqRing (A : RING) : (RING with type t = int -> A.t) =
struct
  type t = int -> A.t
  let zero = fun n -> A.zero
  let one  = fun n -> A.one
  let add  u1 u2 = fun n -> A.add  (u1 n) (u2 n)
  (* . . . *)
end;;
```

   Remarque : la fonction `SeqRing.print` peut se contenter d'afficher les premiers éléments de la suite.

9. Écrivez un foncteur `ProdRing` qui prend deux modules `A` et `B` de signature `RING` et renvoie un autre module de signature `RING` correspondant à l'anneau produit cartésien $\mathtt{A} \times \mathtt{B}$:

```
module ProdRing (A : RING) (B : RING) : (RING with type t = A.t * B.t) =
struct
  type t = A.t * B.t
  (* . . . *)
end;;
```

   Remarque : pour utiliser ce double foncteur, par exemple pour des paires $\mathbb{Z} \times \mathbb{C}$, vous pouvez écrire `module R = ProdRing(ZRing)(CRing)`.

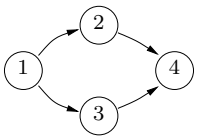
10. Écrivez un foncteur `PolyRing` qui prend un module `A` de signature `RING` et renvoie un autre module de signature `RING` correspondant à l'anneau des polynômes à coefficients dans `A`.

2 Graphes orientés

Dans cet exercice nous voulons écrire des algorithmes (parcours, ...) pour manipuler des graphes orientés. Tout d'abord un petit rappel :

Définition 1. *Un graphe fini orienté G est un couple (V, E), où V est un ensemble fini de sommets (vertices) et $E \subseteq V \times V$ est un ensemble d'arêtes orientées (edges).*

Exemple : le graphe $(\{1,2,3,4\}, \{(1,2),(1,3),(2,4),(3,4)\})$:



Pour cet exercice, il vous est demandé d'utiliser plusieurs fichiers et un `Makefile` pour compiler le tout. Vous reprendrez donc l'exemple de `Makefile` présenté au début du TD et vous l'adapterez.

2.1 Représentation des graphes

Avant de passer aux choses sérieuses et aux algorithmes proprement dits, nous devrons définir un module proposant les types et fonctions de bases (constructeurs et accesseurs). Un tel module devra avoir pour signature :

```
sig
  (* Types *)
  type vertex = int    (* Les sommets sont représentés par des entiers *)
                       (*   numérotés de 1 à n                         *)
  type graph           (* Type abstrait *)

  (* Constructeurs *)
  val empty : graph                  (* Graphe vide *)
  val add_vertex : graph -> graph * vertex
                                     (* Rajoute un sommet au graphe et *)
                                     (*   renvoie ce nouveau sommet    *)
  val add_edge : graph -> vertex -> vertex -> graph
                                     (* Rajoute une arête au graphe *)

  (* Accesseurs *)
  val size : graph -> int    (* Renvoie le nombre de sommets du graphe *)
  val has_edge : graph -> vertex -> vertex -> bool
                             (* Teste l'existence d'une arête *)
  val successors : graph -> vertex -> vertex list
                             (* Renvoie la liste des voisins d'un sommet *)
end
```

Il existe plusieurs méthodes pour représenter un graphe orienté. Nous allons ici étudier les deux plus courantes.

On peut ainsi représenter un graphe par une liste de sommets et, pour chaque sommet $i \in V$, la liste de ses voisins $j \in V$ tels que $(i,j) \in E$. Cela peut s'écrire en Caml avec le type `(vertex * (vertex list)) list`.

On peut alors représenter l'exemple de graphe donné par :

```
[ (1, [ 2; 3 ]);
  (2, [ 4 ]   );
  (3, [ 4 ]   );
  (4, []      )  ]
```

1. Écrivez dans un fichier `lGraph.ml` un module représentant les graphes comme précédemment. Ce module doit avoir la signature décrite plus haut.

Une autre méthode pour représenter un graphe est d'utiliser une matrice d'adjacence : il s'agit, pour un graphe $G = (V, E)$ avec $V = 1, \ldots, n$, d'une matrice $n \times n$ de booléens telle que $A_{i,j}$ soit vrai si et seulement si $(i, j) \in E$. Cela nous donne le type `bool array array`.

L'exemple précédent devient alors :

```
[| [| false;  true;  true; false |];
   [| false; false; false;  true |];
   [| false; false; false;  true |];
   [| false; false; false; false |]  |]
```

2. Écrivez dans un fichier `aGraph.ml` un module représentant les graphes comme des matrices d'adjacence. Ce module aussi doit avoir la signature décrite plus haut.

2.2 Algorithmes

Maintenant qu'on sait construire des graphes, on veut écrire des algorithmes pour les manipuler. Pour cela, nous allons utiliser un foncteur, qui nous permettra de nous abstraire du système de représentation des graphes :

```
module Algos (G : sig
                type vertex = int
                type graph
                (* . . . *)
              end) =
struct
  (* Vos algos ici *)
end
```

Mettez cette ébauche de foncteur dans un fichier `graph.ml`. Vous pourrez alors utiliser le foncteur en écrivant par exemple `module A = Graph.Algos(LGraph)`.

3. Écrivez un algorithme permettant d'effectuer un parcours en profondeur du graphe : `depth_first : G.graph -> G.vertex list`. Attention à ne pas renvoyer plusieurs fois un même sommet.
   Indication : n'hésitez pas à utiliser des structures supplémentaires.
4. Écrivez enfin un algorithme permettant d'effectuer un parcours en largeur du graphe : `breadth_first : G.graph -> G.vertex list`.

Remarque : il existe une bibliothèque qui fait tout ce que vous venez de faire sur les graphes et plus encore, le tout à grands coups de foncteurs. Elle s'appelle Averell et est disponible à l'adresse `http://cristal.inria.fr/~simonet/soft/index.fr.html`.