Cours - TD4 : Les modules

Thèmes et objectifs

— Modules, signatures
— Foncteurs

1 Modules et interfaces

1.1 Les modules

▷ **Support étudiant**

Les modules en OCaml ont une expressivité très riche. Un module est similaire à un paquetage ADA ou à un objet JAVA (mais sans notion d'héritage ou de *self*). Voici un exemple de module :

```
module ExempleModule =
 struct
   include Mon_Module_Inclus
   type mon_type = ...
   let ma_variable = ...
   exception Mon_exception ...
   module Mon_Sous_Module = ...
 end
```

La notation pointée, par exemple ExempleModule.ma_variable, permet l'accès aux composants d'un module, ici au composant ma_variable, du module ExempleModule. Pour faciliter cet accès en allégeant les notations, on peut "ouvrir" le module, avec **open** ExempleModule, i.e. rendre accessible directement ses constituants. Il faudra toutefois faire attention aux problèmes d'ambiguïté, dans le cas où deux modules ouverts définissent des constituants de même nom. Il est également possible d'ouvrir localement un module à l'aide de la notation pointée : ExempleModule.(...). L'expression à l'intérieur des parenthèses peut être complexe (et pas seulement un composant du module) et faire référence à des variables du module.

1.2 Les interfaces

Un module peut se conformer à une (ou plusieurs) interfaces, similaires aux signatures de paquetages ADA ou aux interfaces JAVA. Une interface ressemble à :

```
module type ExempleInterface =
 sig
   include Mon_Interface_Incluse
   type mon_type_concret = ...
   type mon_type_abstrait
   val ma_variable : son_type
   exception Mon_exception ...
   module Mon_Sous_Module : Ma_Sous_Interface
 end
```

La déclaration (facultative) d'un module M respectant une interface I s'écrira M: I. Ces déclarations peuvent également être contraintes en y ajoutant des clauses définissant les types et les modules apparaissant dans l'interface, de la forme :

```
M : I with type mon_type = ... and module Mon_Sous_Module = ... and ...
```

Remarque : Les interfaces OCaml contenant des déclaration de types sont proches des interfaces paramétrées (par ces même types) en JAVA par exemple. L'inclusion de plusieurs interfaces est possible si cela ne crée pas de conflit de symbole. Aucune identification ou unification n'est faite entre composants de mêmes noms inclus depuis plusieurs interfaces.

1.3 Lien avec les fichiers

En OCaml, un fichier `fichier.ml` (respectivement `fichier.mli`) correspond implicitement, une fois compilé, à un module Fichier (respectivement à une interface Fichier).

1.4 Liens module / interface

Nous avons vu que la déclaration qu'un module respecte une interface est facultative. Ces déclarations peuvent également être contraintes en y ajoutant des clauses définissant les types. Ces clauses sont facultatives et peuvent ne concerner qu'une partie des types définis dans l'interface. Nous allons étudier, sur un exemple simple les trois variantes.

```
module type I =
sig
  type t
  val f : t
end


module M1 =
struct
  type t = int
  let f = 0
  let g = 1
end
```

```
module M2 : I =
struct
  type t = int
  let f = 0
  let g = 1
end

module M3 : I with type t = int =
struct
  type t = int
  let f = 0
  let g = 1
end
```

- M1
  - M1 n'est pas déclaré comme respectant l'interface I, tout ce qui est déclaré dans M1 est donc visible de l'extérieur. Ainsi un appel à M1.f+1, depuis l'extérieur du module M1, s'évaluera en `- : int = 1` et l'appel à M1.g est autorisé.
  - Le module n'étant pas déclaré comme respectant l'interface, le compilateur ne vérifiera pas la cohérence du module et de l'interface.
  - Le module respectant l'interface, il pourra être utilisé partout où une interface I est attendue (voir foncteur), et ceci même si la déclaration n'a pas été faite explicitement.
- M2
  - M2 est déclaré comme respectant l'interface I, n'est donc visible à l'extérieur de M2 que ce qui est déclaré dans I. Ainsi un appel à M2.f+1, depuis l'extérieur du module M2, provoquera une erreur : "Error : This expression has type M2.t but an expression was expected of type int" et celui à M2.g "Error : Unbound value M2.g".
  - Le module étant déclaré comme respectant l'interface, le compilateur vérifiera la cohérence du module et de l'interface.
  - Le module respectant l'interface, il pourra être utilisé partout où une interface I est attendue (voir foncteur).

— M3
  — M3 est déclaré comme respectant l'interface I avec le type t étant égal à int, n'est donc visible à l'extérieur de M3 que ce qui est déclaré dans I, ainsi que la valeur du type t. Ainsi un appel à M3.f+1, depuis l'extérieur du module M3, s'évaluera en − : int = 1. Par contre, l'appel à M3.g provoquera toujours l'erreur "Error : Unbound value M3.g".
  — Le module étant déclaré comme respectant l'interface, le compilateur vérifiera la cohérence du module et de l'interface.
  — Le module respectant l'interface, il pourra être utilisé partout où une interface I est attendue (voir foncteur).

2 Utilisation des modules

2.1 Motivation

▷ **Support étudiant**

Lors du cours sur les parcours d'arbres, il a été précisé que : *Un parcours des éléments d'un arbre consiste à présenter ses éléments séquentiellement (i.e. "linéariser") en vue d'itérer un traitement particulier sur cette séquence. Pour simplifier le problème, on effectue une décomposition fonctionnelle en remarquant qu'un tel parcours peut se scinder en deux étapes de calcul :*

1. *construire la liste (séquence) des éléments, dans l'ordre où le traitement à itérer les trouverait.*
2. *appliquer itérativement ce traitement sur la liste obtenue (avec un `fold` par exemple).*

Nous nous étions alors intéressés uniquement à la construction de la liste. Nous avions vu que l'utilisation d'une pile permettait un parcours en profondeur de l'arbre et l'utilisation de la file un parcours en largeur.

Nous rappelons que le type des arbres binaires est le suivant :

```
type 'a standard_tree =
  | Empty
  | Node of 'a * 'a standard_tree * 'a standard_tree
```

Nous rappelons également le code de la construction de la liste pour un parcours en profondeur et largeur :

```
let liste_parcours_profondeur arb =
let rec parcours pile =
 match pile with
 | []                  -> []
 | Empty::q            -> parcours q
 | Node (n, g, d)::q -> n::(parcours (g::d::q))
 in parcours [arb]

let liste_parcours_largeur arb =
let rec parcours file =
 match file with
 | []                  -> []
 | Empty::q            -> parcours q
 | Node (n, g, d)::q -> n::(parcours (q@[g; d]))
 in parcours [arb]
```

Si nous souhaitons trouver un élément pair de l'arbre, pour un parcours en profondeur et largeur, nous aurions le code suivant :

```
let trouver_pair_parcours_profondeur arb =
let rec parcours pile =
```

```
 match pile with
 | []                 -> None
 | Empty::q           -> parcours q
 | Node (n, g, d)::q -> if n mod 2 = 0 then Some n else (parcours (g::d::q))
 in parcours [arb]

let trouver_pair_parcours_largeur arb =
let rec parcours file =
 match file with
 | []                 -> None
 | Empty::q           -> parcours q
 | Node (n, g, d)::q -> if n mod 2 = 0 then Some n else (parcours (q@[g; d]))
 in parcours [arb]
```

Ce TD a pour but de proposer une architecture évitant la redondance de code, en abstrayant la structure de donnée responsable du type de parcours, et le traitement à réaliser (similaire à un fold).

2.2 Application

▷ **Exercice 1**

1. *Écrire une interface qui abstrait les structures de données de type "collection".*
2. *Écrire deux modules respectant cette interface, implantant une structure de pile et une structure de file.*

▷ ***Solution***

```
module type CollectionType =
sig
  type 'a t
  exception CollectionVide
  val vide : 'a t
  val est_vide : 'a t -> bool
  val ajouter : 'a -> 'a t -> 'a t
  val enlever : 'a t -> ('a * 'a t)
end

module File : CollectionType =
struct
  type 'a t = 'a list
  exception CollectionVide
  let vide = []
  let est_vide a = (a=[])
  let ajouter a l = l@[a]
  let enlever l =
    match l with
    | []    -> raise CollectionVide
    | t::q -> (t, q)
end

let%test _ = File.(enlever (ajouter 1 vide) = (1, vide))
let%test _ = File.(fst (enlever (ajouter 2 (ajouter 1 vide))) = 1)

module Pile : CollectionType =
 struct
  type 'a t = 'a list
```

```
  exception CollectionVide
  let vide = []
  let est_vide a = (a=[])
  let ajouter a l = a::l
  let enlever l =
    match l with
    | []    -> raise CollectionVide
    | t::q -> (t, q)
end

let%test _ = Pile.(enlever (ajouter 1 vide) = (1, vide))
let%test _ = Pile.(fst (enlever (ajouter 2 (ajouter 1 (vide)))) = 2)
```

▷ **Exercice 2**

1. *Écrire une interface qui abstrait les paramètres d'un itérateur* `fold`.
2. *Écrire deux modules respectant cette interface, implantant la création d'une liste d'entiers et la recherche d'un nombre pair.*

▷ ***Solution***

```
module type Fold =
sig
  type a
  type b
  val cas_de_base : b
  val traite_et_combine : a -> b -> b
end

module CreerListe : Fold with type a = int and type b = int list =
struct
  type a = int
  type b = int list
  let cas_de_base = []
  let traite_et_combine elt lq = elt::lq
end

module TrouverPair : Fold with type a = int and type b = int option =
struct
  type a = int
  type b = int option
  let cas_de_base = None
  let traite_et_combine elt eltDansQueue = if (elt mod 2 = 0) then (Some elt) else eltDansQueue
end
```

Les types a *et* b *sont rendus visibles pour une utilisation extérieure. Ce n'était pas nécessaire pour les files et piles qui restent donc des structures de données complètement abstraites.*

3 Les foncteurs

3.1 Des modules paramétrés par des modules

▷ **Support étudiant**

On peut également définir des foncteurs, i.e. des transformateurs de modules, i.e. des modules paramétrés par

d'autres modules... ou d'autres foncteurs ! Ceci est analogue aux paquetages ADA avec généricité paramétrique ou aux *component diagrams* d'UML.

En OCaml, la paramétrisation des modules n'est pas autant nécessaire qu'elle peut l'être en ADA par exemple. Ceci est dû au fait qu'on dispose déjà des types génériques (comme `'a list`) d'une part et que les éléments de tout type (sauf les fonctions) disposent de comparateurs prédéfinis d'autre part, ce qui permet d'écrire rapidement des égalités, des tris, etc, sur tout type de données.

A faire :

- Leur faire l'exemple d'un foncteur qui réalise un fold_right

```
module FoldList (F:Fold) =
struct
  let rec fold_right l =
    match l with
    | [] -> F.cas_de_base
    | t::q -> F.traite_et_combine t (fold_right q)
end
```

- Dire qu'on peut aussi abstraire la collection qu'on parcourt et leur faire l'exemple d'un foncteur qui réalise un fold sur une collection.

```
module FoldCollection (C:CollectionType) (F:Fold) =
struct
  let rec fold col =
    if C.est_vide col
    then F.cas_de_base
    else let (e,reste) = C.enlever col in
         F.traite_et_combine e (fold reste)
end
```

- Parler d'application partielle : **module** FoldPile = FoldCollection (Pile)
- Dire que si on avait eu un module Liste, le module FoldList aurait le même comportement que FoldCollection (Liste)
- Parler d'utilisation

```
module CreerListePile = FoldPile (CreerListe)

let%test _ = CreerListePile.fold (Pile.(ajouter 1 (ajouter 2 (ajouter 3 vide))))=[1;2;3]
```

3.2 Application

▷ Exercice 3

1. *Écrire un foncteur qui permet de parcourir un arbre binaire tout en réalisant un traitement sur ses éléments. Il sera paramétré par la structure de données (qui permettra de choisir le type de parcours) et par le traitement.*
2. *Écrire, pour chaque item ci-dessous, un module qui :*
   (a) *Crée la liste des éléments d'un arbre avec un parcours en profondeur.*
   (b) *Crée la liste des éléments d'un arbre avec un parcours en largeur.*
   (c) *Recherche le premier élément pair d'un arbre avec un parcours en profondeur.*
   (d) *Recherche le premier élément pair d'un arbre avec un parcours en largeur.*

▷ *Solution*

```
module ParcoursEtTraite (S : CollectionType) (F:Fold) =
struct
  let traite a =
    let rec aux acc =
      if S.est_vide acc
      then F.cas_de_base
      else let (t,q) = S.enlever acc in
           match t with
           |Empty -> aux q
           |Node (r,g,d) -> F.traite_et_combine r (aux S.(ajouter g (ajouter d q)))
    in
    aux S.(ajouter a vide)
end


(* création de la liste avec un parcours en profondeur gauche - droite*)
module CreerListeProfondeur = ParcoursEtTraite (Pile) (CreerListe)

(* création de la liste parcours en largeur droite - gauche *)
module CreerListeLargeur = ParcoursEtTraite (File) (CreerListe)

(* recherche du premier nombre pair avec un parcours en profondeur gauche - droite*)
module TrouverPairProfondeur = ParcoursEtTraite (Pile) (TrouverPair)

(* recherche du premier nombre pair avec parcours en largeur droite - gauche *)
module TrouverPairLargeur = ParcoursEtTraite (File) (TrouverPair)


let b = Node (1,
  Node (3,
    Node (5,Empty,Empty),
    Node(4,Empty,Empty)),
  Node (2,Empty,Empty))

let%test _ = (CreerListeProfondeur.traite b = [1;3;5;4;2])
let%test _ = (CreerListeLargeur.traite b = [1;2;3;4;5])

let%test _ = (TrouverPairProfondeur.traite b = Some 4)
let%test _ = (TrouverPairLargeur.traite b = Some 2)
```