TD4 : Les modules

1 Modules et interfaces

1.1 Les modules

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 Mon_Module.ma_variable, permet l'accès aux composants d'un module, ici au composant ma_variable, du module Mon_Module. Pour faciliter cet accès en allégeant les notations, on peut "ouvrir" le module, avec **open** Mon_Module, 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 : Mon_Module.(...). 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 de 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 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.*

▷ **Exercice 2**

1. *Écrire une interface qui abstrait un itérateur* `List.fold_right`.
2. *Écrire deux modules respectant cette interface, implantant la création d'une liste d'entier et la recherche d'un nombre pair.*

2 Les foncteurs

2.1 Des modules paramétrés par des modules

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.

2.2 Application

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* `List.fold_left` *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.

Ici, nous allons utiliser les foncteurs pour combiner le parcours de l'arbre et le traitement à réaliser.

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;;
```

▷ **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.*

3 Modules de première classe

3.1 Empaquetage/dépaquetage

Nous avons vu que les modules peuvent être manipulés statiquement, par création locale (à une fonction par exemple) ou globale (à un fichier) et instantiation à travers des foncteurs. L'empaquetage consiste à faire d'un module une valeur comme les autres, c'est-à-dire qu'il pourra être manipulé dynamiquement, passé en paramètre aux fonctions, etc. On se rapproche ainsi beaucoup des objets. Pour convertir un module en valeur ordinaire et réciproquement, on pourra utiliser par exemple les constructions suivantes :

```
module type Info =
sig
  val description : string
  val author : string
end

module Test =
struct
  let description = "this is a fake module"
  let author = "X. Thirioux"
end

let rtfm (module M : Info) = Format.printf "@[<v 0>Description: %s@.Author: %s@]@." M.description M.author

rtfm (module Test)
```

L'écriture **module** Test est un "empaquetage", qui correspond à la construction d'une valeur ordinaire à partir du module Test, argument de la fonction rtfm. Au contraire, l'écriture rtfm (**module** M : Info) est un "dépaquetage", qui correspond à un filtrage d'une valeur représentant un module et récupération de celui-ci sous le nom M. Le mot-clé **module** fait ici office de "constructeur", qui peut se retrouver à la fois dans une valeur ou dans un filtre. Les valeurs représentant un module seront affichées par l'interprète OCaml sous la forme `<module>`.

Dans ce second exemple, on construit une liste contenant des modules (de même interface, car les listes sont homogènes), puis on réalise une sélection dynamique d'un des modules de cette liste :

```
module type Show =
sig
  type t
  val show : t -> string
end

module I =
struct
  type t = int
```

```
  let show i = string_of_int i
end

module B =
struct
  type t = bool
  let show b = if b then "true" else "false"
end

let l =
  let i = (module I : Show) in
  let b = (module B : Show) in
  [ i; b ]

let _ =
  let (module M) = List.nth l (read_int ())
  in (* on peut utiliser le module M *) ()
```

Le module M ainsi créé respecte simplement l'interface Show mais son origine exacte est perdue, son type t est par exemple devenu abstrait. Le type de la valeur renvoyée (ici ()) ne doit pas faire apparaître un des types définis par le module M créé localement, puisque celui-ci disparaît après l'évaluation.

3.2 Application

▷ **Exercice 4** *Les structures de tests sont toujours les mêmes et sont caractérisées par :*
- *— la fonction à traiter*
- *— le paramètre de la fonction*
- *— la valeur de retour attendue*

1. *Écrire une interface* Test *qui abstrait la structure d'un test.*
2. *Écrire deux modules conformes à cette interface et qui permettent de tester*
   (a) *la création d'une liste avec un parcours en profondeur ;*
   (b) *la recherche d'un élément pair avec un parcours en largeur.*
3. *Écrire une fonction de type* (**module** Test) -> bool *qui permet de jouer un test.*

PF
TD 4.1

TD4 : Les modules

I/ Modules et interfaces.

Exercice 1 :

1/
```
module type CollectionInterface =
  sig
    type 'a collection
    val creervide : unit -> 'a collection
    val ajouter : 'a -> 'a collection -> 'a collection
    val enlever : 'a collection -> ('a * 'a collection)
    val est_vide : 'a collection -> bool
    exception CollectionVide
  end
```

2/
```
module Pile : CollectionInterface =
  struct
    type 'a collection = 'a list
    let creervide = []
    let ajouter e p = e::p
    let estvide p = (p=[])
    let enlever p =
      match p with
        | [] -> raise CollectionVide
        | t::q -> (t,q)
    exception CollectionVide
  end
```

```
module File : CollectionInterface with type 'a collection = 'a list =
  struct
    type 'a collection = 'a list
    let creervide = []
    let ajouter e l = l @ [e]
    let estvide p = (p=[])
    let enlever p =
      match p with
        | [] -> raise CollectionVide
        | t::q -> (t,q)
    exception CollectionVide
  end
```

cette déclaration rend visible à l'utilisateur ou au compilateur le type qui est encapsulé.

Exercice 2 :

1/
```
module type FoldRightInterface =
  sig
    type a
    type b
    val traiteEtCombine : a -> b -> b
    val casdeBase : b
  end
```

2/
```
module CrearListe : FoldRightInterface with a = int and b = int list =
  struct
    type a = int
    type b = int list
    let casdeBase = []
    let traiteEtCombine e l = e :: l.
  end

module searchPair : FoldRightInterface with a = int and b = int option.
  struct
    type a = int
    type b = int option
    let cas de base F = None
    let traite et combine t qt = if t mod 2 = 0 then some t else q b.
  end
```

Exercice 3 :

1/
```
module FoldCollection ((CollectionInterface)(FFoldRightInterface) =
  struct
    let rec fold_right col =
      if C.est vide
      then F.casdeBase
      else let (e, reste) = C.enlever col in F.traiteEtCombine e (fold right reste)
  end.
```