Programmation Fonctionnelle Avancée

Cours 5

kn@lri.fr
http://www.lri.fr/~kn

université PARIS-SACLAY

Résumé de l'épisode précédent

Compilation séparée

- `fichier.ml` = module = implémentation
- `fichier.mli` = signature (= type de module) = interface

Foncteur : module paramétré par un autre module

```
module Point =
struct
    type t = float * float
    let compare (x1, y1) (x2, y2) =
      let c = Float.compare x1 x2 in
      if c = 0 then Float.compare y1 y2 else c
    let to_string (x, y) = Printf.sprintf "(%f, %f)" x y
end
module PointSet = Tree.Make(Point)
```


Plan

Corrigé commenté du TP

On fait un corrigé commenté du TP (~30 à 40 minutes)

Plan

Le module `Set`

OCaml propose en standard un module `Set` (implémenté par des AVLs), associé à un foncteur `Set.Make`

L'argument du foncteur est un module de signature :

```
module type OrderedType = sig
    type t
    val compare : t -> t -> int
end
```

Que l'on connaît bien. La plupart des types de données d'OCaml sont dans un module correspondant, qui implémente une fonction `compare` pour ce type

- `Int, Int64, Int32,` : les entiers machine (64 bits, 32 bits)
- `Char, UChar` : les caractères ASCII, les valeurs scalaires unicode
- `String` : les chaînes de caractères
- ...

```
module UCharSet = Set.Make(Uchar)
```


Le module `Map`

Le module `Map` représente des dictionnaires, implémentés comme des AVLs

Le foncteur `Map.Make` prend en argument un `OrderedType` représentant les clés.

```
module type S = sig
  type key (* le type des clés = OrderedType.t *)
  type 'a t  (* le type des dictionnaires contenant des 'a *)

  val empty : 'a t
  val add : key -> 'a -> 'a t -> 'a t
  val remove : key -> 'a t -> 'a t
  ...
end
module SIMap = Map.Make(String)
let days =
  List.fold_left (fun acc (k, v) -> SIMap.add k v acc) SIMap.empty
   [ "monday", 0; "tuesday", 1; "wednesday", 2 ;
     "thursday", 3; "friday", 4; "saturday", 5; "sunday", 6]

let n = SIMap.find days "saturday" (* 5 *)
```


`Set` ou `Map` ?

Peut-on exprimer `Set` en fonction de `Map` ?

Oui !

```
module Make (E : OrderedType) =
  struct
    module MyMap = Map.Make(E)

    type t = unit MyMap.t

    let add e s = MyMap.add e () s
    let remove e s = MyMap.remove e s
    let singleton e s = add e MyMap.empty
    ...
  end
```

On crée des dictionnaires dont les clés sont les éléments de l'ensemble et les valeurs la constante `()`

(en pratique OCaml ne fait pas ça mais duplique et spécialise le code pour des questions de performances)

Plan

Bibliothèque `ocamlgraph`

La bibliothèque `ocamlgraph` propose des algorithmes sur les graphes représentés par des foncteurs

- `Bfs(G : Graph)`, `Dfs(G : Graph)` : parcours en largeur ou en profondeur d'un graphe
- `Dijkstra (G : Graph)`, `BellmanFord (G : Graph)` : plus courts chemins
- ...

Il suffit de donner une représentation pour un graphe avec des opérations élémentaires (liste des sommets, voisins d'un sommet etc...).

Plan

`include`

Le mot clé `include M` permet d'inclure le code du module `M` à l'endroit de l'instruction. Cela permet de créer des copies étendues de modules.

```
module StringExt =
struct
    include String

    let explode s =
      let rec loop i acc =
        if i >= 0 then loop (i-1) (s.[i] :: acc)
        else acc
      in
    loop (String.length s - 1) []
end

let s = StringExt.lowercase_ascii "TOTO"
let ss = StringExt.explode s (* [ 't'; 'o'; 't'; 'o' ]*)
```

Ça *ressemble* à de l'héritage des langages objets

include (2)

Attention cependant, ce n'est qu'une facilité pour éviter la duplication de code.

```
module StringExt =
struct
    include String

    let capitalize _ = failwith "interdit!"
end

let s = StringExt.capitalize "TOTO" (* erreur *)
let ss = String.capitalize "TOTO" (* ok *)
```

On peut masquer des définitions de valeurs.

include (3)

Include fonctionne aussi sur les signatures, on peut simuler des interfaces multiples, mais attention

```
module type Comparable = sig
    type t
    val compare : t -> t -> int
end
module type Printable = sig
    type t
    val to_string : t -> string
end
module type CompPrint = sig
  include Comparable
  include Printable
end
(* Error: Illegal shadowing of included type  *)
```

On peut masquer des définitions de valeurs mais pas des définitions de types

include (4)

```
module type Comparable = sig
    type t
    val compare : t -> t -> int
end
module type Printable = sig
    type t
    val to_string : t -> string
end
module type CompPrint = sig
  include Comparable
  include Printable with type t := t
  (* on dit que les 2 types t sont les mêmes *)
end
```


Directive open

La directive `open M` permet de rendre visible les définitions du module `M` dans l'espace de noms courant.

```
open List
let l = length [ "a"; "b"; "c"] (* 3 *)
let l2 = map String.uppercase_ascii l
```

Attention, c'est dangereux car les fonctions d'un module ont souvent un nom court et générique, deux opens peuvent être en conflit

```
open List
open String
let l = length [ "a"; "b"; "c"] (* erreur, appelle String.length sur une liste *
let l2 = map uppercase_ascii l
```

Sauf indication contraire, on n'utilisera pas `open`

Différence entre `open` et `include`

`open` ne fait que rendre visible des symboles, `include` les ajoutes comme s'ils avaient été écrits dans le fichier :

```
module A = struct
  include String
  let explode s = ...
end
let n = A.length "abcde" (* String.length est copiée dans A *)
module B = struct
  open String
  let explode s =
    let rec loop i acc =
    if i >= 0 then loop (i-1) (s.[i] :: acc)
    else acc
in
loop (length s - 1) []

end
let b = B.length "abcde" (* error, unbound value B.length *)
```


Conclusion sur les modules

Les modules sont un outil précieux qui permet:

- La compilation séparée (fichiers .ml)
- La définition d'interfaces propres (fichiers .mli, restrictions, ...)
- De paramétrer du code par des types et des opérations associées (grâce aux foncteurs)

De façon générale, lorsque l'on crée un type de donnée en OCaml

- Il est conseillé de le mettre dans son propre module
- Il est conseillé d'appeler le type principal du module `t`
- Il est conseillé de le munir d'opérations de base :
  - `compare`
  - `to_string`
  - `hash` et `equal` que l'on verra la semaine prochaine