Programmation Fonctionnelle Avancée

Cours 4

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

université
PARIS-SACLAY

Résumé de l'épisode précédent

On a présenté les ABRs, une structure de données :

- persistante : la structure n'est pas modifiée
- efficace : les accès sont en temps logarithmique en la taille
- récursive : basée sur des arbres
- *polymorphe* : le type des éléments est quelconque

Pour ce dernier point, on a triché, car on a utilisé

```
val compare : 'a -> 'a -> int
```

C'est une fonction prédéfinie de la bibliothèque standard OCaml qui fonctionne sur tous les types de valeur, mais qui applique un comportement différent selon le type.

compare

OCaml supporte du *polymorphisme paramétrique*. Cela signifie :

- que l'on peut écrire des fonctions polymorphes, i.e. qui prennent des valeurs de n'importe quel type en argument.
- que l'on peut écrire des types dont des sous-composants sont génériques (ex: `'a list`, `('a, 'info) tree`, …)

Pour que l'argument d'une fonction soit *polymorphe* il faut que la fonction ne l'inspecte pas. Elle doit se contenter

- de le renvoyer : `let id x = x` de type `'a -> 'a`
- de l'ignorer : `let ignore _ = ()` de type `'a -> unit`
- de le passer en paramètre à une fonction : `List.map : ('a -> 'b) -> 'a list -> 'b list`

Dès qu'une fonction fait une hypothèse sur le type possible d'un argument, elle devient monomorphe

Exemples

```
let f x y = x + y
(* f : int -> int -> int, car on a passé x et y comme arguments à l'addition *)

let rec len l = match l with
    [] -> 0
    | _ :: ll -> 1 + len ll
(* len : 'a list -> int, car on a inspecté l. On impose que
ce soit une liste. Par contre on a pas inspecté les éléments donc
le type reste partiellement polymorphe *)


let rec map f l = match l with
[] -> []
| x :: ll -> (f x) :: map f ll
(* len : ('a -> 'b) -> 'a list -> 'b list, car on a inspecté l.
On impose que ce soit une liste. On impose juste que le type
des éléments soit le même que celui attendu par la fonction.
On appelle f, on impose donc que ce soit une fonction. *)
```

`compare` (suite)

La fonction `compare` ne devrait pas pouvoir exister. En effet, c'est une fonction prédéfinie, écrite en C, qui inspecte à l'exécution la représentation mémoire de la valeur et effectue la bonne comparaison:

- Si les valeurs sont des entiers, comparer comme des entiers
- Si ce sont des chaînes de caractères, comparer octet par octet
- Si ce sont des blocs, comparer les étiquettes
- Si ce sont des blocs avec des étiquettes égales, comparer récursivement les contenus

(on étudiera plus tard la représentation mémoire des valeurs OCaml)

Si on veut permettre au programmeur de choisir sa comparaison, il doit l'écrire en fonction du type des données qu'il veut mettre dans l'arbre. Problème: toutes les opérations vont prendre en paramètre la comparaison.

« solution » ? (not)


```
(* dans tree.ml *)
let add compare e t = ...
let remove compare e t = ...
let mem compare e t = ...
```

C'est très lourd à utiliser:

```
let compare_str s1 s2 =
    let u1 = String.uppercase_ascii s1 in
    let u2 = String.uppercase_ascii s2 in
    compare u1 u2

let t =
  add compare_str "b" (add compare_str "b" (add compare_str "c" empty))
```

C'est facile de se tromper, le typage ne nous aide pas:

```
let t1 = add compare "B" t
```

On a créé un arbre incorrect qui contient `"B"` et `"b"`.

Plan

Fichier de module (`.ml`)

En OCaml, un fichier `.ml` contient des définitions de types et de valeurs (dont des fonctions). C'est une *unité de compilation*. Tout fichier `foo.ml` définit un *module* de nom `Foo`. On peut utiliser le nom `Foo` pour accéder aux définitions du module depuis un autre fichier.

```
(* dans le fichier hello.ml *)
let print name = Printf.printf "Hello, %s!\n" name

(* ------------------ *)

(* dans le fichier main.ml *)
let () = Hello.print "Sherlock"
```

Cela permet de grouper des types et des fonctions associées sous un même nom.

La bibliothèque standard d'OCaml est organisée en module : `List`, `String`, ...

Compilation séparée (1)

Un module *dépend* d'un autre module s'il est référencé. Dans l'exmple précédant, le module `Main` dépend du module `Hello`.

On doit compiler les fichiers dans l'ordre de leur dépendences (*).

On doit lier les fichiers dans l'ordre de leur dépendences.

```
$ ocamlc -c hello.ml    #produit hello.cmo

$ ocamlc -c main.cmo    #produit main.cmo

$ ocamlc -o main.exe hello.cmo main.cmo
#lie tous les fichiers objets dans un exécutable

$ ./main.exe #lance le programme
Hello, Sherlock!
$
```

Fichier d'interfaces (`.mli`)

De nombreux langages proposent des modules (Python, JavaScript, Java (depuis Java 9), C++ (bientôt)).
Un point intéressant est de pouvoir laisser des définitions (de type, de valeurs) internes au module et d'en exporter d'autres (publiques).

Un fichier d'interface (extension `.mli`) contient des définitions de types et des *signatures* de valeurs. Une signature de valeur est de la forme :

```
val nom : type
```

Un fichier `foo.mli` est associé à un fichier `foo.ml`. Par exemple:

```
(* dans le fichier hello.ml *)
let print name = Printf.printf "Hello, %s!\n" name

(* ------------------- *)

(* dans le fichier hello.mli *)
val print : string -> unit
```

Fichier d'interfaces (`.mli`) (2)

L'intérêt d'un fichier d'interface est de pouvoir restreindre ou masquer des définitions. On considère l'exemple du TP 1, les fractions :

```
(* dans le fichier frac.ml *)
type t = { num : int; denom : int}
let pgcd a b = ... (* calcul du pgcd *)
let frac a b = ... (* renvoie une fraction :
                      - irréductible *
                      - dont le signe est sur num *)
let num f = f.num
let denom f = f.denom
(* opérations arithmétiques, font l'hypothèse que f1 et f2
   sont au bon format *)
let add f1 f2 = ...
let sub f1 f2 = ...
let neg f = ...
let mul f1 f2 = ...
let div f1 f2 = ...
let inv f1 f2 = ...
```

Fichier d'interfaces (`.mli`) (3)

Si on écrit pas de fichier `frac.mli` OCaml considère que l'interface est celle des types inférés :

```
(* dans le fichier frac.mli *)
type t = { num : int; denom : int}
val pgcd : int -> int -> int
val frac : int -> int -> t
val num : t -> int
val denom : t -> int
val add : t -> t -> t
val sub : t -> t -> t
val neg : t -> t
...
```

C'est problématique :

- On a accès à la fonction auxiliaire `pgcd`
- On peut construire une fraction « à la main » qui ne respecte pas les invariants : `{num = 2; denom = 4 }`

Fichier d'interfaces (`.mli`) (4)

Dans un fichier d'interface, on peut masquer des définitions de type et des valeurs :

```
(* dans le fichier frac.mli *)
type t (* on ne dit pas que c'est un enregistrement *)
       (* la fonction pgcd est cachée *)
val frac : int -> int -> t
val num : t -> int
val denom : t -> int
val add : t -> t -> t
val sub : t -> t -> t
val neg : t -> t
```

Dans un fichier `main.ml` :

```
let p = Frac.pgcd 2 4 (* Error: unbound value Frac.pgcd *)
let f = { num = 2; denom = 4 } (* Error: unbound record field num *)
```

Intérêt ?

L'intérêt est double :

- Le programmeur du module `Frac` peut faire l'hypothèse que les fractions sont toujours au bon format ⇒ code plus simple ou plus efficace (car moins de tests, ...)
- L'utilisateur du module `Frac` n'as pas à se soucier de la représentation interne.

```
(* fichier frac.ml alternatif *)
type t = int * int
let num (n, _) = n
let denom (_, d) = d
...
```

Le fichier `frac.mli` n'a pas besoin d'être modifié, et ses dépendences non plus !

Compilation des fichiers `.mli`

Les fichiers d'interface doivent êtres compilés :

- avant les fichiers qui dépendent d'eux
- avant leur implémentation (le `.ml` correspondant)

Si le fichier `.ml` et `.mli` ne correspondent pas, une erreur est levée:

```
    (* dans le fichier hello.mli, ocamlc -c hello.mli *)
    val print : int -> unit

    (* dans le fichier hello.ml, ocamlc -c hello.ml *)
  let print name = Printf.printf "Hello, %s!\n" name

Error: The implementation hello.ml does not match the interface hello.cmi:
Values do not match:
  val print : string -> unit
is not included in
  val print : int -> unit
The type string -> unit is not compatible with the type int -> unit
Type string is not compatible with type int
```

Plan

Structures

En OCaml, on peut définir des sous-modules dans un fichier :

```
(* dans le fichier num.ml *)
module Frac =
 struct
   type t = { num : int; denom : int }
   let pgcd a b = ...
   ...
   let add f1 f2 = ...
   let to_string f = Printf.sprintf "%s/%s" f.num f.denom
end
type t = Int of int | Float of float | Frac of Frac.t

let to_string n =
  match t with
    Int i -> string_of_int i
    | Float f -> string_of_float f
    | Frac f -> Frac.to_string f

...
```

Explications

Dans le slide précédent, dans le module `num.ml` on a définit un sous-module (structure) `Frac` pour encapsuler le code des fractions. On a ainsi un code plus naturel que si on avait écrit :

```
type frac = { num : int; denom : int }
let add_frac f1 f2 = ...
let frac_to_string f = Printf.sprintf "%s/%s" f.num f.denom
...

type t = Int of int | Float of float | Frac of frac
```

Signatures

De la même façon que l'on peut définir des sous-modules, on peut définir des signatures, i.e. des « types pour les modules ». Les signatures sont pour les sous-modules ce que les `.mli` sont aux `.ml`.

```
module type Frac_S =
 sig
   type t (* on ne rend pas public *)
      (* on exporte pas pgcd *)
   val add : t -> t -> t
   val to_string : t -> string
   ...
end
module Frac : Frac_S =
struct
  type frac = { num : int; denom : int }
  let pgcd a b = ...
  let add f1 f2 = ...
  let to_string f = Printf.sprintf "%s/%s" f.num f.denom
end
  (* Frac.pgcd n'est pas visible, même depuis le fichier *)
```

Signatures (2)

Il n'est pas obligatoire de donner un nom aux signatures, même si c'est souvent utile. Par exemple :

```
module Frac =
sig
  type t (* on ne rend pas public *)
    (* on exporte pas pgcd *)
  val add : t -> t -> t
  val to_string : t -> string
  ...
end =
struct
  type frac = { num : int; denom : int }
  let pgcd a b = ...
  let add f1 f2 = ...
  let to_string f = Printf.sprintf "%s/%s" f.num f.denom
end
```

Et dans les interfaces ?

Dans une interface on peut masquer totalement ou partiellement les définitions de sous-modules et de types de sous-modules. On considère un fichier `num.mli` pour l'implémentation `num.ml` :

```
module type Frac_S : sig
  type t
  val add : t -> t -> t
  val to_string : t -> string
  ...
end
module Frac : Frac_S

type t = Int of int | Fload of float | Frac of Frac.t
```

Ici, on rend public `Frac_S`, `Frac` et la définition du type `Num.t`

Et dans les interfaces ? (2)


```
(* dans num.mli *)

type t = (* Int of int | Fload of float | Frac of Frac.t,
            la définition ci-dessus ne peut plus être publique *)

val add : t -> t -> t
val of_int : int -> t
val of_float : float -> t
val of_frac : int -> int -> t (* utilise Frac.of_frac a b *)
```

Dans l'exemple ci-dessus, on a complètement masqué `Frac`. On ne peut donc pas rendre publique la définition de `t`. Il faut donc fournir des fonctions de constructions (dans le `.ml` aussi)

Plan

« Passer du code en paramètre »

Problème initial :

- On veut « paramétrer » toutes les fonctions d'un module (`tree.ml`) par une fonction (de comparaison).
- On veut qu'un module paramétré d'une certaines façon n'ai pas le même type qu'un module paramétré par une autre fonction, même si les éléments sont du même types

On va procéder en plusieurs étapes :

1. Définir la notion de « type associé à une fonction de comparaison »
2. Définir la notion de « module paramétré par un tel type »
3. Définir plusieurs exemples concrets de types (instancier)

Convention : tous les modules que l'on définit dans la suite auront au moins **un** type principal, exporté par le module qui s'appelera `t`. C'est la convention usuelle quand on programme en OCaml.

Type associé à des opérations

On peut définir une signature de module :

```
(* dans notre fichier tree.ml *)
module type Comparable =
sig
  type t
  val compare : t -> t -> int
end
```

On dit ainsi : il existe des modules contenant un type `t` ainsi qu'une fonction de comparaison prenant 2 `t`s et renvoyant un entier.

```
(* dans notre fichier tree.mli *)
module type Comparable =
sig
  type t
  val compare : t -> t -> int
end
```

On exporte la définition telle qu'elle dans le fichier `tree.mli`

Module paramétré (1)

On peut définir maintenant les opérations que l'on souhaite pour notre module d'arbre:

```
(* suite de notre fichier tree.ml *)
  module type S =
    sig
    type elt (* le type des éléments *)
    type t   (* le type des arbres *)
    val empty : t
    val add : elt -> t -> t
    val mem : elt -> t -> bool
    val union : t -> t -> t
    ...
  end
```

On remet aussi cette signature dans `tree.mli`

Module paramétré (2)

On peut définir notre module paramétré aussi appelé *foncteur*

```
module Make (E : Comparable) : S with type elt = E.t =
 struct
  (* on est dans un foncteur, qui prend un module E en argument *)
 type elt = E.t
            (* les éléments de notre arbre seront du type des
               éléments de E *)
 type t = Leaf  | Node of int * t * elt * t
  (* on suppose des AVL, l'entier est la hauteur *)

 let rec mem e t =
    match t with
      Leaf -> false
      | Node (_, l, v, r) ->
        let c = E.compare e v in
        if c = 0 then true else
        if c < 0 then mem e l else mem e r
  ...
end
```

Explications

On crée un foncteur qui `Make`

- prend en argument un module `E` ayant la signature `Comparable`
- renvoie en résultat un module ayant la signature `S`
- le module renvoyé a en plus la contrainte que le type `elt`(des éléments) est le même que celui des type de `E`
- À tout endroit du code de `Make` où l'on veut utiliser la comparaison, on utilise `E.compare`

Interface et utilisation


```
(* suite du fichier tree.mli *)
...
module Make (E : Comparable) : S with type elt = E.t
(* les définitions de Comparable et S sont au-dessus *)
```

On a juste à indiquer la présence de `Make` tel qu'il est déclaré.

```
(* dans un fichier main.ml *)
module UString = struct
 type t = string
 let compare s1 s2 =
    let u1 = String.uppercase_ascii s1 in
    let u2 = String.uppercase_ascii s2 in
    compare s1 s2
end

module UStringTree = Tree.Make(UString)

let t1 = UStringTree.add "a" UStringTree.empty
```

Interface et utilisation (2)

La notation `Foo.(...)` permet de rendre les fonctions du module `Foo` directement dans les parenthèses.

```
let t2 = UStringTree.(
    add "a" (
      add "b" (
        add "C" empty)))
let t3 = UStringTree.union t1 t2
```

C'est assez agréable à utiliser (même si les définitions sont pénibles).

Les modules de la bibliothèque standard OCaml suivent la même convention (un type `t` avec une fonction `compare`.

```
module IntTree = Tree.Make(Int)
module StringTree = Tree.Make(String)

let t4 = StringTree.(add "a" empty)
let t5 = StringTree.union t4 t5
(* Error: this value has type UStringTree.t but is
          here used with type StringTree.t *)
```

Ensembles d'ensembles

Si on ajoute une fonction `compare : t -> t -> int` dans notre foncteur `Make` (et dans la signature `S`) alors:

```
module IntTree = Tree.Make(Int)
module IntSetTree = Tree.Make(IntTree)
```

On obtient gratuitement des ensembles d'ensembles (d'entiers).

Remarque: la bibliothèque standard d'OCaml utilise des foncteurs pour les collections : ensembles (`Set.Make`), dictionnaires (`Map.Make`), tables de hachages (`Hashtbl.Make`), ...

Plan

Problème avec la compilation séparée

On a joliment séparé notre code en plusieurs fichiers d'interface (.mli) et d'implémentation (.ml). Pour tirer partie de cela il faut :

- Compiler les fichiers dans le bon ordre (sinon erreur)
- Lorsqu'on modifie un fichier, re-compiler tous ceux qui en dépendent (sinon erreur)
- Recompiler le moins de choses possibles (sinon temps de compilation longs)

Constat

- Le faire à la main est fastidieux
- Les outils tels que `make` peuvent aider, mais il sont génériques. Il faut appeler d'autres outils pour cacluler les dépendences.

L'outil dune

L'outil dune est l'outil standard de construction de projet OCaml. Il permet :

- de construire automatiquement des projets OCaml
- de calculer les définitions de types sur tout un projet (pour l'éditeur de code)
- d'indiquer des dépendences vers des bibliothèques externes (graphiques, système, ...)

On ne fera dans ce cours qu'une utilisation minimale de dune.

Structure d'un répertoire/projet sous dune

Un projet (simple) a la structure suivante :

- Un fichier `dune-project` contenant des informations globales
- Un fichier `dune` indiquant la cible à construire :

```
(executable
 (name main)
 (promote (until-clean))
 (libraries gg vg vg.svg))
```

dans le fichier ci-dessus, `dune` va essayer de construire un programme `main.exe`. Il va lire `main.ml` puis récursivement trouver tous les modules référencés (e.g. `Tree.` → `tree.ml/tree.mli`).

- Les fichiers `.ml/.mli` des modules du projet, dont le fichier principal.
- Si un module n'est pas trouvé, et n'est pas dans la bibliothèque standard, `dune` le cherche dans les modules des bibliothèques OCaml listées dans `libraries`.

Commandes

- `dune build` construit un projet (les fichiers objets sont dans un sous-répertoire `_build`
- `dune clean` efface le fichier `.exe` et le répertoire `_build`
- `dune build -w` se met en attente et recompile le projet à chaque fois qu'un fichier est modifié, automatiquement.

L'outil est bien plus riche et permet de définir des bibliothèques, du code de tests etc...

On demande juste une connaissance des deux premières commandes pour ce cours, en particulier, les fichiers `dune` seront toujours fournis.