Modules et Foncteurs

INF 441 : Programmation Avancée

Xavier Rival

26 avril 2017

Quelques éléments administratifs

Projets :

- **4 sujets** sur la page du cours, texte complet dans Moodle
- **à choisir maintenant**
  deadline pour les choix : vendredi 28 avril 2017
- proposition d'un **sujet personnel** possible
  en parler avec les enseignants de TDs et moi même
- me communiquer votre choix par mail
  choix effectif lorsque je confirme réception...

Café-LIX : réunion d'information sur la 3A et les PA d'info
prévu chaque mercredi, de 13h00 à 14h00

Regroupement de définitions

Vu dans les cours précédents :

- un programme OCaml consiste en **une suite de déclarations** :
  types (`type t = ...`),
  valeurs, fonctions ou constantes (`let n = ...`)
- on peut effectuer une définition **récursive** en utilisant le mot clé `rec`

```
let rec f x = ...
let rec f x = ... and g y = ... and h x = ...
```

- une déclaration peut dépendre d'une déclaration **précédente**
  mais pas d'une déclaration suivante

Comment évaluer cette organisation des programmes du point de vue des **deux notions d'abstraction** mentionnées au premier cours :

- **masquage d'informations locales**
- **paramétrisation**

Un premier problème : interfaces globales

Abstraction :

- masquage d'informations locales
- paramétrisation

À ce stade, **il n'est pas possible** de :

- **regrouper des déclarations avec un lien logique** :
  on peut les regrouper syntaxiquement, mais ce n'est pas très formel...
- **cacher des déclarations hors d'une section** :
  on peut seulement faire un `let ... in ...` local sur **une** déclaration, mais pas sur une série de déclarations liées logiquement

```
(* Arbres *)
type arbre = ...
let vide = ...
let existe x a = ...
let ajoute x a = ...
(* Code utilisant les arbres *)
let tri ... = ...
```

Un second problème : paramétrage par bloc

Abstraction :

- masquage d'informations locales
- paramétrisation

À ce stade, on peut :

- **paramétrer un type par un autre** : types polymorphes
- **paramétrer une déclaration** : fonctions

Mais il est impossible de faire un **paramétrage global et simultané** d'une série de déclarations

```
(* Arbres binaires de recherche *)
type 'a bst = ...
let vide = ...
let existe comparaison x a = ...
let ajoute comparaison x a = ...
(* Code utilisant les arbres *)
let construit_base ... = ...
```

En Java

Regroupement de déclarations : dans des **classes**

- une classe définit **un type**
- possibilité de masquage via les méthodes **privées**

```
class C {
  t0 c0; t1 c1; // champs definissant un type
  ta ma( ... ){ ... }; // methode
  private tb mb( ... ){ ... }; // methode privee
}
```

Paramétrage de classes : via les **classes génériques**

- classe générique : classée **paramétrée** par une ou plusieurs classes
- s'utilise ensuite comme toute classe

```
class <C0> C { // classe parametree
  // ...
}
```

Système de modules

Dans ce cours, nous allons étudier une solution à ces deux problèmes :

- **Modules** :
  unité logique regroupant plusieurs déclarations
- **Foncteurs** :
  modules **paramétrés** par des modules

Les deux constructions **n'ont aucun rapport avec les objets**

Outline

Module

Comme attendu :

Définition : module

Un **module** est une **unité regroupant une séquence de définitions de types et de valeurs**.

Syntaxe :

```
module M =
  struct
    type t0 = ...
    let x0 = ...
    type t1 = ...
    let f0 u v = ...
    let f1 u = ...
    let f2 u v w = ...
  end
```

Module : exemple

Considérons une implémentation conjointe des **listes et files** :

```
module Liste =
  struct
    type 'a liste = Nil | Cons of 'a * 'a liste
    let l_empty = Nil
    let l_add x l = Cons (x, l)
    let l_rev = (* inversion de liste usuelle *)
    type 'a file = 'a liste * 'a liste
    let f_empty = (Nil, Nil)
    let f_add x (l, m) = (Cons (x, l), m)
    let norm = function
      | a, Nil -> Nil, l_rev a
      | p -> p
    let f_pop p =
      match norm p with
      | _, Nil -> failwith "file vide"
      | l, Cons (a, b) -> a, (l, b)
  end
```

Accès par nom

Supposons que ce module `Liste` est défini :

```
utop # l_empty ;;
Error: Unbound value l_empty
utop # l_rev ;;
Error: Unbound value l_rev
```

On ne peut pas accéder directement aux champs de notre module
C'est précisément le principe d'un module :

- déclarations **regroupées**
- **accès** au contenu du module **contrôlé**

Première méthode d'accès : **notation pointée** `Module.nom`

```
utop # Liste.l_empty ;;
- : 'a Liste.liste = Liste.Nil
utop # Liste.l_rev ;;
- : 'a Liste.liste -> 'a Liste.liste = <fun>
```

À ce stade, les modules **ressemblent à des enregistrements**, dans lesquels un champ **peut aussi définir un type**

Accès par ouverture

La notation pointée n'est pas pratique lorsqu'on utilise souvent les déclarations d'un module...

Seconde méthode d'accès :

1. **ouverture** via `open Module`
2. **accès direct** aux éléments du module (types et valeurs)

Exemple :

```
utop # open Liste ;;
utop # l_empty ;;
- : 'a Liste.liste = Liste.Nil
utop # l_rev ;;
- : 'a Liste.liste -> 'a Liste.liste = <fun>
```

Accès aux champs d'un module

Accès direct :

- on peut toujours accéder aux déclarations **globales** et du **module courant**

Notation pointée :

- `Module.nom` ou `Module.t` ou `Module.Constructeur...`
- un peu lourd à l'usage

Ouverture de module :

- `open Module` **importe** toutes les déclarations de `Module` **dans l'espace de noms global**
- très léger...
- mais potentiellement **dangereux** :

  on peut **masquer** des noms déjà définis...

  e.g., tous les conteneurs ont une valeur `empty`

  donc on ne fait jamais de `open` pour les librairies (`List`...)

Signature de module

Intuitivement, il s'agit du **type d'un module** :

Définition : signature

Une **signature de module** regroupe une séquence de

- **définitions de types** (complètes ou juste en donnant le nom du type)
- **types de valeurs** (de la forme `val nom : t`)

Syntaxe :

```
struct
  type t0 = ...
  val x0: typ_x0
  type t1
  val f0: typ_f0
  val f1: typ_f1
  val f2: typ_f2
end
```

Signature : exemple

Reprenons l'exemple précédent... On peut définir la signature :

```
module type LISTE =
  sig
    type 'a liste
    val l_empty: 'a liste
    val l_add: 'a -> 'a liste -> 'a liste
    val l_rev: 'a liste -> 'a liste
    type 'a file
    val f_empty: 'a file
    val f_add: 'a -> 'a file -> 'a file
    val f_pop: 'a file -> 'a * 'a file
  end
```

Remarques :

- chaque déclaration (valeur, fonction) est caractérisée uniquement **par son type** ; les définitions de types sont **cachées**
- la fonction `f_norm`, **à usage interne** a été **omise** dans cette signature

Signature par défaut

Lorsque l'on définit un module, **la signature par défaut** correspond au **type du module**, qui expose **tous les types et noms** :

```
module Liste :
  sig
    type 'a liste = Nil | Cons of 'a * 'a liste
    val l_empty : 'a liste
    val l_add : 'a -> 'a liste -> 'a liste
    val l_rev : 'a liste -> 'a liste
    type 'a file = 'a liste * 'a liste
    val f_empty : 'a liste * 'b liste
    val f_add : 'a -> 'a liste * 'b -> 'a liste * 'b
    val norm : 'a liste * 'a liste -> 'a liste * 'a liste
    val f_pop : 'a liste * 'a liste
          -> 'a * ('a liste * 'a liste)
  end
```

En particulier, la **fonction interne** `norm`, que l'on souhaitait masquer demeure **visible**.

Contrainte de signature

On peut **contraindre** un module **à l'aide d'une signature** :

```
module M = (struct (* ... *) end: SIG)
```

- la signature `SIG` doit être **plus précise** que celle du corps du module (on peut avoir moins de fonctions, avec des types moins généraux mais pas le contraire)
- de l'extérieur, on ne voit **que les éléments** de `SIG`

```
module Liste =
  (struct
    (* meme fonctions que precedemment... *)
  end: LISTE)
```

Les **signatures** sont donc un élément important pour **masquer** des informations devant demeurer locales...

Unité de compilation

Nous allons maintenant nous intéresser aux **liens entre modules et compilation** :

- comment **découper** un programme **en plusieurs fichiers** ?
- comment **compiler** l'ensemble de ces fichiers ?

Définition : unité de compilation

Une **unité de compilation** est un bloc de code atomique du point de vue du compilateur.

En **OCaml**, une **unité de compilation** peut être vue exactement comme un **un module**.

Conventions

L'utilisation de modules définis par des fichiers obéit aux conventions suivantes :

- **fichier** `.ml` : implémente le **corps d'un module**
- **fichier** `.mli` : implémente une **signature de module**
- **syntaxe** : pas de `struct` / `end` ou `sig` / `end`
  (autrement, cela revient à définir un module dans un module)
- `f.mli` est la signature de `f.ml`
- en l'absence de `.mli`, OCaml utilise la signature par défaut
- le module correspondant à `f.ml` est `F` (si on souhaite faire un `open` ou accéder à l'un de ses champs depuis un autre fichier)

Voir exemple à suivre...

Exemple en plusieurs fichiers : définition

Reprenons notre exemple listes / files, mais en utilisant des fichiers séparés :

```
(* fichier fileliste.ml *)
type 'a liste = Nil | Cons of 'a * 'a liste
let l_empty = Nil
let l_add x l = Cons (x, l)
let l_rev = (* inversion de liste usuelle *)
type 'a file = 'a liste * 'a liste
let f_empty = (Nil, Nil)
let f_add x (l, m) = (Cons (x, l), m)
let norm = function
  | a, Nil -> Nil, l_rev a
  | p -> p
let f_pop p =
  match norm p with
  | _, Nil -> failwith "file vide"
  | l, Cons (a, b) -> a, (l, b)
```

Exemple en plusieurs fichiers : définition

Reprenons notre exemple listes / files, mais en utilisant des fichiers séparés :

```
(* fichier fileliste.mli *)
type 'a liste
val l_empty : 'a liste
val l_add : 'a -> 'a liste -> 'a liste
val l_rev : 'a -> 'b
type 'a file
val f_empty : 'a liste * 'b liste
val f_add : 'a -> 'a liste * 'b -> 'a liste * 'b
val f_pop : 'a liste * 'b liste
      -> 'b * ('a liste * 'b liste)
```

Note : le module ainsi défini est `Fileliste`

Compilation de module

Compilation d'une signature :

- signature compilée : fichier avec extension `.cmi`
- `ocamlc -c src.mli` produit un fichier `src.cmi`

Compilation d'une implémentation de module, en **bytecode** :

- code compilé bytecode avec extension `.cmo`
  portable d'une machine à l'autre
- `ocamlc -c src.ml` produit un fichier `src.cmo`
- si pas de `.cmi`, c'est la signature par défaut qui compte
  et un `.cmi` correspondant est généré

Compilation d'une implémentation de module, en **code natif** :

- code compilé avec extension `.cmx`
  non portable d'une machine à l'autre, mais **plus efficace**
- `ocamlopt -c src.ml` produit un fichier `src.cmx`

Édition de liens

La compilation d'un module **ne produit pas un fichier exécutable**...
Il faut aussi **lier** entre elles les modules, pour obtenir soit un **programme** soit une **librairie**.

Création d'un exécutable :

- `ocamlc -o exec a.cmo b.cmo c.cmo ...`
  ou `ocamlopt -o exec a.cmx b.cmx c.cmx ...`
- le fichier obtenu **peut être lancé en ligne de commande**
- placer les modules **dans l'ordre de leurs dépendances**

Création d'une librairie :

- `ocamlc -o lib.cma a.cmo b.cmo c.cmo ...`
  ou `ocamlopt -o lib.cmxa a.cmx b.cmx c.cmx ...`

Compilation d'un projet en plusieurs fichiers

Étapes :

1. compiler les signatures
2. compiler les fichiers d'implémentations
3. lier l'ensemble avec `ocamlc -o exec fichiers.cmo`

Note : il faut **préserver l'ordre des dépendances**...

Exemple :

- sources : `a.ml`, `a.mli`, `b.ml`, `b.mli`, `main.ml`
- dépendances : `A` utilise `B` ; `Main` utilise `A` et `B`

```
ocamlc -c b.mli
ocamlc -c a.mli
ocamlc -c b.ml
ocamlc -c a.ml
ocamlc -c main.ml
ocamlc -o exec b.cmo a.cmo main.cmo
```

Compilation d'un projet en plusieurs fichiers

Une **variation** sur l'exemple précédent :

- sources : `a.ml`, `a.mli`, `b.ml`, `main.ml`
- **pas de** `b.mli`
- mêmes dépendances : `A` utilise `B` ; `Main` utilise `A` et `B`
- il faut alors compiler `b.ml` avant `a.mli` puisque `A` utilise `B`

```
ocamlc -c b.ml
ocamlc -c a.mli
ocamlc -c a.ml
ocamlc -c main.ml
ocamlc -o exec b.cmo a.cmo main.cmo
```

En pratique, on utilise souvent un **outil automatique** (`make`, `obuild`)

Remarques générales

Découpage en unités de compilation séparées :

- critique pour découper le travail de programmation
- un `.mli` (signature) est très utile pour spécifier et convenir de définitions communes
- très certainement indispensable pour les projets en binôme...

Librairies : sont définies dans des modules

- librairies standard :
  `List`, `Printf`, `Unix`, `Arg`...
- librairies supplémentaires, disponibles dans `opam` :
  `tls` (Transport Layer Security)
  `camomile` (support pour UTF-8)...
- souvent plusieurs fichiers linkés en un `.cma` ou `.cmxa`
  un tel module peut contenir plusieurs modules...

Outline

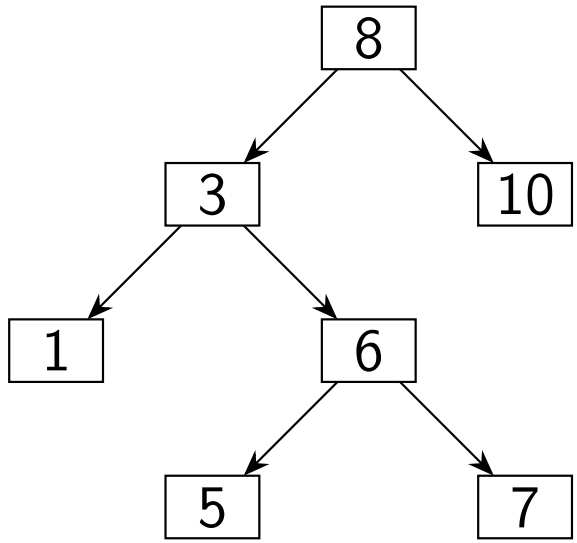
Arbres binaires de recherche

On considère une implémentation des **arbres binaires de recherche** avec disposition des éléments déterminée par une relation d'ordre :

Types de base :

```
(* arbres *)
type 'a tree =
  | Leaf
  | Node of 'a tree
            * 'a * 'a tree
(* ordres possibles *)
type ordre = Inf | Eq | Sup
(* fonction de comparaison *)
val comparaison:
    'a -> 'a -> ordre
```

Exemple avec des valeurs entières :



Arbres binaires de recherche

On considère une implémentation des **arbres binaires de recherche** avec disposition des éléments déterminée par une relation d'ordre :

Un module définissant **quelques fonctions** :

```
module type ABR =
  sig
    type 'a tree
    val vide:      'a tree
    val chercher:  ('a -> 'a -> ordre)
          -> 'a -> 'a tree -> bool
    val inserer:   ('a -> 'a -> ordre)
          -> 'a -> 'a tree -> 'a tree
    val supprimer: ('a -> 'a -> ordre)
          -> 'a -> 'a tree -> 'a tree
  end
let ord x y = (* ... *)
let t0 = ABR.vide
let t1 = inserer ord x1 t0
let t2 = inserer ord x2 t1
```

Foncteurs : principe

Cette construction n'est pas satisfaisante :

- la répétition de l'argument nuit à la **concision**
- on peut également utiliser **plusieurs relations** sans aucun message d'erreur, même si le code suivant **n'a pas de sens** :

```
let orda x y = (* ... *)
let ordb x y = (* ... *)
let t0 = ABR.vide
let t1 = inserer orda x1 t0
let t2 = inserer ordb x2 t1
let t3 = inserer orda x3 t2
```

Principe alternatif : **procéder à une paramétrisation globale**

Définition : foncteur

Un **foncteur** est un **module paramétré par un autre module**.

Déclaration

Syntaxe :

```
module F = functor (M: S) ->
  struct
    (* corps d'un module classique *)
  end
```

ou

```
module F (M: S) =
  struct
    (* corps d'un module classique *)
  end
```

- la signature `S` doit être définie avant le foncteur
- le corps de `F` peut faire référence aux définitions de `M`, soit via la notation pointée, soit via un `open`
- on peut également contraindre la signature "de sortie" de `F`

Application

Syntaxe :

```
module type S = sig (* ... *) end
module F (M: S) = struct (* ... *) end

module Param = struct (* ... *) end

module Instance = F( Param )
```

- la signature du paramètre `Param` doit être **au moins aussi générale** que `S`
- le module `Instance` peut être utilisé directement
  il **définit tous les éléments** qui apparaissent dans le corps de `F`
- note : `F` ne peut pas être utilisé directement ;
  il faut d'abord en créer une instance

Exemple : arbres sur les types ordonnés

Reprenons l'exemple complet **des arbres binaires de recherche**, à l'aide d'un **foncteur**...

Signature des éléments ordonnés :

```
(* ordres possibles *)
type ordre = Inf | Eq | Sup

(* signature parametre du foncteur *)
module type OrdType =
  sig
    type t
    val comparaison: t -> t -> ordre
  end
```

- l'interface décrit n'importe quel type sur lequel on peut définir une fonction de comparaison
- mais à ce stade, on n'a pas d'implémentation pour `OrdType`...

Exemple : arbres sur les types ordonnés

La définition des arbres binaires de recherche prend la forme d'un **foncteur** qui est paramétré par `OrdType` :

```
module Arbre = functor (Elt: OrdType) ->
  struct
    type tree =
      | Leaf
      | Node of tree * Elt.t * tree
    let vide = Leaf
    (* fonction utilisant Elt.comparaison *)
    let chercher (x: Elt.t) (t: tree): bool = (* .. *)
    let inserer (x: Elt.t) (t: tree): tree = (* ... *)
    let supprimer (x: Elt.t) (t: tree): tree = (* . *)
  end
```

- **exercice** : compléter la définition des fonctions
- **note** : toutes les fonctions de `Arbre` utilisent la même fonction de comparaison, à savoir `Elt.comparaison`

Exemple : instantiation

Avant de définir une **instance**, il faut **implémenter une version** de `OrdType`.

Par exemple, si on utilise **des clés entières** :

```
module OrdInt =
  struct
    type t = int
    val comparaison n m =
      if n < m then Inf
      else if n > m then Sup
      else Eq
  end
```

Instance du foncteur `Arbre` :

```
module ArbreInt = Arbre( OrdInt )
let t = AbreInt.inserer 3 ArbreInt.vide
```

Exemple : instantiations multiples

On peut créer une autre instance tout aussi simplement :

```
module OrdFloatLexico =
  struct
    type t = float * float
    val comparaison (f0,f1) (g0,g1) =
      if f0 < g0 then Inf
      else if f0 > g0 then Sup
      else if f1 < g1 then Inf
      else if f1 > g1 then Sup
      else Eq
  end
module ArbrePoints = Arbre( OrdFloatLexico )
```

Note :
Deux instances différentes d'un même foncteur sont deux modules **distincts** ; en particulier, les références de l'une n'ont rien à voir avec les références de l'autre.

Foncteurs présents dans la librairie

La **librairie standard** de OCaml contient plusieurs définitions de foncteurs :

- `Set` :
  définit une **représentation pour ensembles finis** sur un type quelconque, muni d'une fonction de comparaison ; la structure du code est très similaire à notre exemple à base d'arbre binaires de recherche (il s'agit d'arbres AVL...).

- `Map` :
  définit une **représentation pour fonctions à support fini**
  et utilise le même principe

Exercice : instancier et utiliser ces foncteurs

Outline

Modules de première classe

Foncteurs et **généricité** :

- à ce stade, les instantiations sont **complètement définies** dans le code source...
- donc, il est **impossible** de choisir lors de l'exécution quel module on souhaite passer en paramètre à un foncteur...
  "hack" : prévoir plusieurs instances d'un foncteur dans le source et choisir à runtime les fonctions que l'on appelle...

Cette solution nuit à la clarté et à l'efficacité du code.

Modules de première classe

Les **modules de première classe** sont des **expressions** OCaml qui **décrivent des modules** et peuvent être manipulées **comme toute autre expression**.

"De première classe" = qui fait partie du langage

Manipulation des modules de première classe

Trois opérations fondamentales :

- **construction d'une expression décrivant un module**, **à partir d'un module** `M` de signature `SIG` :

```
(module M : SIG)
```

- **construction d'un module**, **à partir d'une expression** `expr` **décrivant un module** de signature `SIG` :

```
(val expr : SIG)
```

- **définition locale d'un module**, dans une expression :

```
let module M = (val expr : SIG) in
expr (* on peut utiliser M ici *)
```

Exemple

Application : **création** et **utilisation dynamique** d'une instance des arbres binaires de recherche

```
module Ord0 = ...
module Ord1 = ...
module Ord2 = ...

let main ( ) =
  (* ... *)
  let e_mod =
    match kind with
    | K0 -> (module Ord0: OrdType)
    | K1 -> (module Ord1: OrdType)
    | K2 -> (module Ord2: OrdType) in
  let module O = (val e_mod : OrdType) in
  let module A = Arbre( O ) in
  A.f ( )
```

Modules, foncteurs, modules de première classe et abstraction

Modules :

- les définitions "**internes**" peuvent être **masquées** à l'aide d'une signature restrictive
- fournit un **découpage naturel** du code

Foncteurs :

- **généricité** via paramétrage des **types** de l'argument
- **généricité** via paramétrage des **champs** de l'argument
- de plus, permettent de garantir des **invariants plus forts** que les fonctions d'ordre supérieur classiques

Modules de première classe :

- **création dynamique** de module (code très générique)
- mais code pouvant être **plus complexe**

Outline

Principaux éléments à retenir

Module :

- groupe d'éléments (types, valeurs) définis dans une même unité
- **syntaxe** : `struct type t = (*... *)let x = (*... *)end`

Signature :

- définit les éléments que doit implémenter un module et qui sont visibles à l'extérieur
- **syntaxe** : `sig type t = (*... *)val x = (*... *)end`

Foncteur :

- définition d'un module paramétré par un autre module
- **syntaxe (définition)** :
  `module F = functor (M: SIG)-> struct (*... *)end`
- **syntaxe (application)** : `F(P)`

Paramétrage et langages de programmation

La plupart des langages généralistes offrent un système de **structuration** et de **généricité** :

- **Java** :

  interfaces : jouent le rôle des signatures (mais pas seulement)

  classes : jouent le rôle des modules (mais pas seulement)

  classes génériques : idem, mais pour les foncteurs
  $\Rightarrow$ nous étudierons les objets lors du prochain cours

- **C++** :

  utilise **classes**, **objets** et **templates**

- **Haskell**, **SML** :

  système de modules assez proche de celui d'OCaml