Programmation Orientée Objet

INF 441 : Programmation Avancée

Xavier Rival

3 mai 2017

Quelques éléments administratifs

Rencontre délégués–enseignants :

le 9 mai 2017

- mardi prochain, Safran, entre 12h15 et 13h30
- merci de regrouper les commentaires éventuels !

Projets :

Deadline le 31 mai 2017

- contacter les enseignants responsables pour toutes les questions éventuelles

Documents sur OCaml

Deux **nouveaux liens** sur la page du cours :

- un **document court qui rappelle les principales constructions** :

  `http://gallium.inria.fr/~fpottier/X/INF441/fiche-ocaml.pdf`

- un **document à l'attention des programmeurs Java** :

  `http://gallium.inria.fr/~fpottier/X/INF441/memo-java-ocaml.pdf`

Dans les deux cas, des documents de références, à utiliser si vous êtes bloqués par un point particulier, lié à la syntaxe ou autre...

Programmation Orientée Objet (POO)

Jusqu'à présent, nous n'avons pas parlé d'**objets**, et pourtant :

- certains **langages très utilisés** reposent fortement sur ce concept en particulier : Java et C++
- plusieurs **cours** précédents utilisent Java et donc les objets INF 311, INF 321, INF 421...

Pourquoi **n'avons nous pas parlé d'objets** jusqu'à présent ?

Aujourd'hui :

- concepts fondamentaux de **programmation orientée objet**
- **comparaison** avec d'autres traits de langage : fonctions, modules...

Lien entre structures et code

Rappel : en **Java**, vous avez vu qu'une classe relie

1. une **structure de données** : objets et champs
2. un **processus de création d'éléments** de cette structure (le ou les constructeur(s))
3. des **algorithmes** qui manipulent cette structure (méthodes)

Lors du précédent cours, nous avons vu les **modules** (et **foncteurs**) qui regroupent :

- des déclarations de **types**
- des déclarations de **noms** décrivant fonctions et autres valeurs

Note :

- ces types et valeurs ne sont pas forcément liés entre eux
- ex : cas de `Fileliste`, qui contient plusieurs types et fonctions sur les listes et files...

Définition graduelle

Rappel : en **Java**, il est possible de définir des classes de manière graduelle, en rajoutant à une classe existante de nouveaux éléments :

```
// Definition d'une premiere classe
class C0 { /* ... */ }
// Definition d'une classe C1 par heritage de C0
class C1 extends C0 { /* ... */ }
```

- le corps de `C1` définit une extension de `C0`
- quelle différence avec la construction suivante ?

```
module M0 = struct (* ... *) end
module M1 =
  struct
    include M0
    (* ... *)
  end
```

Objets et classes

Aujourd'hui :

- **objets** :
  éléments fondamentaux
- **classes** :
  cadre engendrant des objets,
  peuvent être définies par **héritage**
- **lien** entre ces notions et celles définies dans les séances précédentes :
  types, **fonctions**, **modules**
- **programmation orientée objet** en OCaml
  et **comparaison** avec d'autres langages

Outline

Notion d'objet et création

Définition : objet

Un **objet** est une structure de données qui contient :

- un ensemble de **champs** (généralement mutables), qui décrivent les valeurs stockées dans l'objet ;
- un ensemble de **méthodes**, qui décrivent les opérations possibles sur cet objet.

Syntaxe en OCaml :

```
object
  val ch_0 = v_0              (* champ non mutable *)
  val mutable ch_1 = v_1      (* champ mutable *)
  val mutable ch_2 = v_2      (* champ mutable *)
  method meth_0 = (* ... *)
  method meth_1 = (* ... *)
end
```

Champs d'un objet

On considère une définition d'objet :

```
let o =
  object
    val ch_0 = v_0            (* champ non mutable *)
    val mutable ch_1 = v_1    (* champ mutable *)
    method meth_1 x = body
  end
```

Champs :

- peuvent être **mutables** ou **non mutables**
  non mutables : ne changent jamais après création de l'objet
- sont **invisibles depuis l'extérieur**
  note : en Java, on peut accéder à un champ d'un objet depuis une autre classe

Méthodes d'un objet

On considère une définition d'objet :

```
let o =
  object
    val ch_0 = v_0            (* champ non mutable *)
    val mutable ch_1 = v_1 (* champ mutable *)
    method meth_1 x = body
  end
```

Méthodes :

- un **appel** se note comme suit :

  `o#meth_1 argx`

- l'objet o est toujours argument d'une méthode
  note : en Java ce n'est pas le cas pour les méthodes *statiques*
- donc, essentiellement toutes les méthodes sont **dynamiques**
  (aussi appelées **méthodes d'objet**)
  i.e., il n'existe pas de méthodes **statiques** en OCaml

Création à l'aide d'une fonction

Initialisation d'un objet **lors de sa création** :

- les champs reçoivent tous une valeur donnée par une expression
- on peut rendre ces valeurs **paramétriques**, en définissant l'objet comme étant **le résultat d'une fonction**
- en Java, on procède à l'aide de **constructeurs**

Exemples :

```
(* objet sans parametre *)
let o0 = object (* ... *) end
(* fonction engendrant un objet *)
let make x y z = object (* ... *) end
(* instance *)
let o1 = make arg0 arg1 arg2
```

Nous reviendrons sur **la création des objets** ultérieurement...

Exemple : objet décrivant un point dans l'espace

```
let mkpoint px py =
  object
    val mutable x = px
    val mutable y = py
    method translate (vx, vy) =
      x <- x + vx;
      y <- y + vy
    method sym_center =
      x <- -x;
      y <- -y
  end
let point_origin = mkpoint 0 0
let point_a = mkpoint 3 2
point_a#sym_center
```

Opérations :

- **initialisation** à l'aide des paramètres de la fonction
- **méthodes** décrivant l'effet d'une translation et d'une symétrie
- **création** de deux points et **application** d'une symétrie

Objet et référence à lui-même

Référence à l'objet sur lequel porte un appel de méthode :

- via un nom donné à l'objet, généralement `self` et déclaré juste après `object`
- utile **pour les appels de méthodes**
- pas pour l'accès aux champs, qui est toujours direct
- penser à `this` en Java...

```
let mkpoint px py =
  object (self)
    val mutable x = px
    val mutable y = py
    method sym_x = y <- -y
    method sym_y = x <- -x
    method sym_center =
      self#sym_x;
      self#sym_y
  end
```

Type d'un objet

Considérons le type de `mkpoint` :

```
# let mkpoint px py =
  object
    (* ... *)
  end
val mkpoint :
  int -> int -> < sym_center : unit;
                  translate : int * int -> unit >
  = <fun>
```

Un **type d'objet** décrit le type de chacune de ses méthodes.

Notes :

- les champs **n'apparaissent pas** dans ce type
  cohérent puisque les champs sont invisibles depuis l'extérieur...
- comme dans les cours précédents, **un type décrit les interactions possibles avec un élément**

Accès à l'état interne

Les champs sont **invisibles depuis l'extérieur**, ce qui signifie qu'on ne peut pas les lire ni les modifier directement.

Astuce : utiliser des méthodes

```
let mkpoint px py =
  object
    val x = px
    val mutable y = py
    method get_x = x
    method get_y = y
    method set_y ny = y <- ny
  end
```

- **getter** : méthode retournant la valeur d'un champ
- **setter** : méthode affectant la valeur d'un champ mutable
- en général pas souhaitable pour tous les champs :
  pas pour les champs qu'on veut abstraire / masquer

Sous-typage

Considérons la fonction suivante, qui prend un objet et lui applique "sa" méthode `translate` :

```
# let tr o = o#translate (2, 3) ;;
val tr : < translate : int * int -> 'a; .. >
  -> 'a = <fun>
```

Noter le type de cette fonction :

- elle peut s'appliquer à un objet créé par `mkpoint`
- mais **plus généralement** à n'importe quel objet ayant une méthode `translate` prenant deux entiers en paramètres

Il reste donc **plusieurs choses à explorer** au sujet des types des objets :

- comment **comparer** le type de deux objets
- comment **modifier** a posteriori le type d'un objet pour masquer des méthodes

Sous-typage et informations locales

Définition : sous-typage

On dit que `t0` est un **sous-type** de `t1` si et seulement si `t1` est plus général que `t0` ou autrement dit, si tout objet ayant le type `t0` admet aussi le type `t1`. L'opération qui transforme une expression de type `t0` en une expression de type `t1` s'appelle une **coercion**.

Sous-typage : `t0` est sous type de `t1` si et seulement si

- toute méthode de `t1` apparaît dans `t0`
- pour toute méthode apparaissant dans les deux types d'objets, le type dans `t1` est plus général que celui dans `t0`

Coercion :

- si `o0` est un objet de type `t0`, sous type de `t1`, on note sa **coercion** `(o0 :> t1)` (ou `(o0 : t0 :> t1)`)
- l'objet résultant est de type `t1`

Outline

Notion de classe

En général, en programmation orientée objet, **on ne parle pas d'objets sans aussi introduire des classes** :

Définition : classe

Une **classe** est un **schéma servant à engendrer des objets**.

- Nous avons vu qu'en **OCaml**, on **peut déjà définir des objets** : un objet peut être défini **en tout point**
- Le **concept de classe** est crucial en POO en particulier, pour parler d'**héritage**
- Une **classe** peut être déclarée à l'aide de la syntaxe :

```
class nom =
  object
    (* corps similaire a un objet *)
  end
```

Classe : exemple et génération d'un objet

Reprenons **l'exemple précédent** :

```
class mkpoint px py =
  object (self)
    val mutable x = px
    val mutable y = py
    method sym_x = y <- -y
    method sym_y = x <- -x
    method sym_center =
      self#sym_x;
      self#sym_y
  end ;;
let o = new mkpoint 8 (-1) in
o#sym_center ;;
```

- **création** de l'objet à l'aide du mot clé `new` (penser à `new` en Java)
- **initialisation** via une fonction, comme pour les objets standards
- chaque objet définit une **nouvelle instance de la classe**
  aucun partage des champs mutables entre instances distinctes

Classe et génération d'un objet

Deux instances d'une classe sont distinctes, à la fois physiquement et logiquement :

```
# class make i =
    object
      val x: int = i
    end ;;
  let c0 = new make 0 in
  let c1 = new make 0 in
  (c0 == c1, c0 = c1) ;;
- : bool * bool = (false, false)
```

- == teste l'égalité **physique** (même adresse)
- = teste l'égalité **logique** (même valeur)

Type d'une classe

Lors de la déclaration de `mkpoint`, nous obtenons le **type** de classe suivant :

```
class mkpoint :
  int ->
  int ->
  object
    val mutable x : int
    val mutable y : int
    method sym_center : unit
    method sym_x : unit
    method sym_y : unit
  end
```

Ce type décrit **le type des objets engendrés** par la classe.
Note : le type des champs y figure également, on verra bientôt pourquoi.

Méthodes statiques et méthodes dynamiques

Un bref retour sur les méthodes...

Java a **deux types importants de méthodes** :

- **méthodes dynamiques** (ou **méthodes d'objet**) :
  appliquées à un objet (impossible de les appeler sans avoir créé un objet, `NullPointerException` si appelées sur `null`...)
- **méthodes statiques** :
  appelées directement (pas sur un objet)

Une classe OCaml **n'a que des méthodes dynamiques**

- en Java toute fonction doit être méthode d'une classe d'où l'importance des méthodes statiques
- en OCaml, on peut définir une fonction hors de toute classe

Outline

Notion d'héritage

Un principe **d'évolutivité du code**, par **addition** et **redéfinition** :

- on peut définir plusieurs classes **partageant** certaines caractéristiques (champs et méthodes)
- **utilité** :
  définir d'abord une **classe générale** puis la **spécialiser**
- le terme "**surcharger**" signifie :
  remplacer par une entité **compatible** au sens des types

D'où la définition :

Définition : Héritage

Soit `C0` une classe. Une seconde classe `C1` **hérite** de `C0` si elle est définie à partir de celle-ci, en ajoutant ou surchargeant des champs et / ou méthodes. On appelle la classe `C1` la classe **fille** et `C0` la classe **mère**.

Héritage : syntaxe

Déclaration en OCaml d'une classe héritant d'une autre :

```
class mere =
  object
    (* ... *)
  end
class fille =
  object
    inherit mere
    (* ... *)
  end
```

- dans le corps de la classe `fille`, on peut **accéder directement** aux champs et méthodes de la classe `mere`
- si `fille` **redéfinit** une méthode de `mere`, le **type** doit être **préservé** (surcharge)
- `fille` peut aussi ne pas redéfinir tous les champs de `mere`
- en Java, on utilise le mot clé `extends`, avant le début de la classe, mais le principe est le même...

Héritage : un exemple simple

Une application de **gestion de fichiers** :

- plusieurs formats
- **tous** les formats ont une méthode de **suppression**
  méthode `supprimer`
- seuls les **exécutables** peuvent être **lancés**
  méthode `executer`
- seules les **images** peuvent être **affichées**
  méthode `afficher`

L'héritage permet ici une **spécialisation**

```
class fichier =
  object
    val nom
    val chemin
    method supprimer =
      (* ... *)
  end
class fichier_exe =
  object
    inherit fichier
    method executer =
      (* ... *)
  end
class fichier_jpg =
  object
    inherit fichier
    method afficher =
      (* ... *)
  end
```

Liaison dynamique

```
class c0 = object (* ... *) method m = (* ... *)
class c1 =
  object
    inherit c0
    method m = (* ... redefinition de m ... *)
  end
class c2 =
  object
    inherit c0
    (* ... pas de redefinition de c0 ... *)
  end
List.iter (fun o -> o#m) [ new c0 ; new c1 ; new c2 ]
```

Quelle occurrence de la méthode `m` exécuter ?

- occurrence de `C0` ou `C2` : méthode `m` de `C0`
- occurrence de `C1` : méthode `m` de `C1`

Liaison dynamique

```
class c0 = object (* ... *) method m = (* ... *)
class c1 =
  object
    inherit c0
    method m = (* ... redefinition de m ... *)
  end
class c2 =
  object
    inherit c0
    (* ... pas de redefinition de c0 ... *)
  end
List.iter (fun o -> o#m) [ new c0 ; new c1 ; new c2 ]
```

Quelle occurrence de la méthode `m` exécuter ?

- cette information est **dynamique**, c'est à dire qu'elle ne peut être résolue **que pendant que le programme tourne**
- code **très expressif** mais **plus complexe et cher à l'exécution**

Liaison dynamique

La **liaison dynamique** est **encore plus compliquée** qu'il n'y parait :

```
class c0 =
  object (self)
    val mutable x = 0
    method m = x <- 1
    method n = self#m
  end
class c1 =
  object
    inherit c0
    method m = x <- 2
  end
```

Même dans la méthode `n` de classe `c0`, on ne sait pas quelle `m` est appelée :

- lors d'un appel à `n`, il se peut que `n` soit surchargée ou non
- si `n` n'a pas été surchargée, on exécute `m`
- si `m` est surchargée, alors on exécute la surcharge, comme dans `c1` !
- sinon, on exécute la méthode `m` de `c0`...

Visibilité : méthodes publiques et privées

On peut modifier la **visibilité** d'une méthode :

- par défaut, une méthode est **publique** : visible partout
  depuis une instance de la classe ou depuis une méthode d'une des classes qui en héritent
- une méthode **privée** est déclarée avec le mot clé `private`
  dans ce cas, elle est visible depuis :
  - les méthodes de la classe où elle est définie,
  - les méthodes des classes qui en héritent

Utilité : **masquer une méthode**, par volonté d'abstraction

- application 1 : méthode destinée à être utilisée dans les autres méthodes de la classe
  e.g., une sous-opération pouvant être appelée par `lire`, `supprimer`...
- application 2 :
  méthode destinée à être surchargée, liée dynamiquement, dans les classes héritères...

Méthodes virtuelles et notion de classe virtuelle

On peut également **utiliser** une méthode qui ne sera **définie que dans une classe fille** :

- on appelle **méthode virtuelle** une méthode dont on donne **seulement un type**, mais **pas la définition**
- **syntaxe** :

```
class virtual c0 =
  object
    val mutable x = 0
    method virtual m: unit
  end
```

```
class c1 =
  object
    inherit c0
    method m = x <- x + 1
  end
```

- on ne peut pas créer un objet à partir d'une classe virtuelle :

```
# let o = new c0 ;;
Error: Cannot instantiate the virtual class c0
```

- à l'inverse, on peut créer un objet à partir de c1

Exemple : transformations géométriques

Données :

- description de **transformations géométriques usuelles** (symétries, translations, rotations, etc)
- description de **leur effet** sur **un point** ou une forme (segment, rectangle, etc)

Observation

Pour chaque transformation considérée, l'image d'un segment est un segment, dont les extrémités sont obtenues par la transformation donc seul le calcul de l'image d'un point dépend vraiment de la transformation.

Structure choisie :

- une **classe virtuelle**, avec une **méthode virtuelle** `tr_point`
- des **méthodes générales** pour les segments, etc.
- une **classe dérivée** pour chaque type de transformation

Exemple : transformations géométriques

Voici le code obtenu :

```
class virtual transformation =
  object (self)
    method virtual tr_point: (float * float) -> (float
        * float)
    method tr_segment (pt0, pt1) =
      (self#tr_point pt0, self#tr_point pt1)
  end
class sym_x =
  object
    inherit transformation
    method tr_point (x, y) = (x, -. y)
  end
class translation a b =
  object
    inherit transformation
    method tr_point (x, y) = (x +. a, y +. b)
  end
```

Sous-typage et héritage

Une remarque importante :

sous-typage et **héritage** sont deux notions **différentes**

- **sous-typage** : notion **sémantique**
  on peut construire deux classes définissant des objets de mêmes types, sans aucune relation d'héritage

- **héritage** : notion **syntaxique**
  i.e., une classe définie syntaxiquement à partir d'une autre
  voir exemple (un peu plus subtil) :
  `https://caml.inria.fr/pub/docs/oreilly-book/html/book-ora144.html`

Outline

Applications

Nous avons vu **plusieurs méthodes de mise en oeuvre des classes** :

1. **comme définition générale** pouvant être **utilisées** comme telles ou **spécialisées** avant utilisation
   exemple : le système de fichiers
2. **comme canevas** à **spécialiser avant utilisation**
   exemple : les transformations géométriques

Dans chaque cas, il est possible d'**utiliser d'autres traits de langage**, comme des modules, des foncteurs, des types sommes.

- quand utiliser ces autres traits ?
- quand utiliser quand même les objets ?

Classes, objets et types sommes

Un exemple déjà vu il y quelques semaines :

un **langage d'expressions arithmétiques**

Définition du type :

```
type bin = Add | Sub | Mul
type expr =
  | Ecst of int
  | Eneg of expr
  | Ebin of bin * expr * expr
```

Quelques exemples de **fonctions associées** :

```
val to_string: expr -> string
val eval: expr -> int
```

Classes, objets et types sommes

La **définition** complète de l'une de ces fonctions :

```
let rec eval e =
  match e with
  | Ecst i -> i
  | Eneg e0 -> - (eval e0)
  | Ebin (Add, e0, e1) -> (eval e0) + (eval e1)
  | Ebin (Mul, e0, e1) -> (eval e0) * (eval e1)
  | Ebin (Sub, e0, e1) -> (eval e0) - (eval e1)
```

- définition mathématique par **induction sur la syntaxe** ;
- le code suit très exactement ce principe.

Comment aurait-on fait en Java ? **Avec des classes / objets !**
Pourquoi, et quelles sont les **avantages** et **inconvénients** ?

Exemple : expressions arithmétiques

Classe virtuelle décrivant le type des expressions :

```
class virtual e_expr =
  object
    method virtual to_string: string
    method virtual eval: int
  end
```

Classe dérivée pour les **constantes** :

```
class e_const (i: int) =
  object
    inherit e_expr
    method to_string = string_of_int i
    method eval = i
  end
```

Exemple : expressions arithmétiques

Classe virtuelle décrivant le type des expressions :

```
class virtual e_expr =
  object
    method virtual to_string: string
    method virtual eval: int
  end
```

Classe dérivée pour **opérations unaires** :

```
class e_uni (e: e_expr) =
  object
    inherit e_expr
    method to_string = Printf.sprintf "- (%s)" e#
        to_string
    method eval = - e#eval
  end
```

Exemple : expressions arithmétiques

Classe virtuelle décrivant le type des expressions :

```
class virtual e_expr =
  object
    method virtual to_string: string
    method virtual eval: int
  end
```

Classe dérivée pour **opérations binaires** :

```
class e_bin (b: bin) (e0: e_expr) (e1: e_expr) =
  object
    inherit e_expr
    method to_string =
      Printf.sprintf "(%s) %s (%s)"
        (op_to_string b) e0#to_string e1#to_string
    method eval = op_eval e0#eval e1#eval b
  end
```

où `op_to_string` et `op_eval` convertissent les opérateurs en chaînes et

Exemple : expressions arithmétiques et ajout d'un cas

Ajoutons un cas, par exemple, une fonction **qui calcule la moyenne de trois expressions** :

```
class e_moy (e0: e_expr) (e1: e_expr) (e2: e_expr) =
  object
    inherit e_expr
    method to_string =
      Printf.sprintf "MOY( %s, %s, %s)"
        e0#to_string e1#to_string e2#to_string
    method eval = (e0#eval + e1#eval + e2#eval) / 3
  end
```

- il nous a suffit d'ajouter **une nouvelle classe dérivée**
- pas de modification aux autres classes existantes
- en fait, on n'a même pas besoin de savoir si il en existe

Classes, héritage et types sommes

Nous venons de **simuler un type somme** à l'aide de **classes** et **héritage**, comme vous l'avez sans doute souvent fait en Java... (mais il n'y avait pas le choix, car il n'y a pas de types sommes en Java)

Classes + héritage :

- regroupe le code **par cas** (constantes, opérations binaires...)
- **facilite l'ajout d'un cas** a posteriori...
- mais l'ajout d'une fonction nécessite l'**édition de toutes les classes**

Type sommes :

- regroupe le code **par fonction** (affichage, évaluation...)
- **facilite l'ajout d'une fonction** supplémentaire
- mais l'ajout d'un cas nécessite **d'éditer toutes les fonctions**

Classes et modules

En Java, les classes sont souvent utilisées là où il serait possible d'utiliser des **modules** :

- notion d'**espace de noms**
- **paramétrage** via les **classes génériques**

Les deux constructions **ne sont pas équivalentes** :

- les classes ne permettent pas de définir directement des types locaux
- pas de méthodes applicables sans avoir un objet (méthodes statiques en Java)
- les modules peuvent être étendus (via `include`), mais l'extension ne permet pas la liaison tardive obtenue par héritage

Simulation de traits de langage

On peut **simuler les types sommes à l'aide d'objets / classes**

- une **classe virtuelle**
- une classe **dérivée par héritage** pour chaque constructeurs

On peut **simuler les modules à l'aide d'objets / classes**

- une classe peut jouer le rôle d'un **espace de noms**

Que gagnons nous ? Que perdons nous ?

- un **langage plus simple**, avec moins de constructions. . .
- mais **un code moins clair** (en raison des encodages)
- mais **pas le même support du langage** :
  pas de `Warning 8: this pattern-matching is not exhaustive` si on utilise l'approche objet !

Exemple : JavaScript simule objets, modules, foncteurs à l'aide d'enregistrements extensibles dynamiquement

Outline

Principaux éléments à retenir

Classes :

- servent de **cadre** à la création d'objets
- combinent **code** et **méthodes** (fonctions)

Héritage :

- **mécanisme d'extension syntaxique** pour les classes
  par surcharge (redéfinition de méthodes existantes)
  et par définition de méthodes supplémentaires
- nécessite la **résolution dynamique** des appels de méthodes
  c'est-à-dire pendant l'exécution du programme

Applications et langages

Applications multiples :

- types sommes extensibles indéfiniment
- structuration du code dynamique
- bien d'autres motifs (dont certains vus dans les prochains cours)

Selon **le langage de programmation**, les objets sont plus ou moins importants :

- **Java** : tout est objet ou presque...
  autres langages où les objets sont prépondérants : Eiffel, Smalltalk...
- **OCaml** :
  objets en pratique moins souvent utilisés que les fonctions, types sommes, etc
- **C++** :
  conçu comme une extension de C, avec classes et objets