Programmation Avancée (INF441)

Services, objets, clôtures

François Pottier

19 avril 2016

Rappel : les projets

Quatre sujets de projet, disponibles en ligne dès maintenant.

Réfléchissez-y et travaillez-y **dès maintenant !**

Faites connaître votre choix DEMAIN au plus tard.

- par email à `francois.pottier@inria.fr`
- (depuis un email `@polytechnique.edu`)

Types et abstraction

Les types offrent un mécanisme de protection.

Ils permettent donc de construire et faire respecter des abstractions.

Je distingue deux manières d'obtenir ce résultat :

- grâce aux types abstraits ;
  - la semaine dernière
- grâce aux types de service.
  - aujourd'hui

Types abstraits (rappel)

Le typage statique permet de définir et faire respecter un type abstrait.

Le mot-clef `private` interdit l'accès aux champs et méthodes privés :

```
public class Count implements ICount {
  // Fields. (Data.)
  // Invariant: step divides (next - init).
  private int next;
  private final int init, step;
  ...
}
```

Ainsi, le respect de l'invariant est garanti.

Types abstraits (rappel)

Les types abstraits d'OCaml jouent un rôle analogue.

Le fichier `Count.ml` contient une définition de type :

```
type count =
  { mutable next: int; init: int; step: int }
  (* Invariant: step divides (next - init). *)
...
```

`Count.mli` contient une déclaration, ce qui rend le type `count` abstrait.

```
type count
...
```

Certains types et fonctions peuvent ne pas apparaître dans la signature.

1 Services

2 Clôtures d'OCaml et objets de Java

3 Clôtures de Java et classes internes de Java

4 Des clôtures aux données

Service

Une autre façon d’imposer une abstraction est de proposer un service :

- un objet dont on ne sait rien, excepté qu’il offre certaines méthodes ;
- une fonction dont on ne sait rien, excepté qu’on peut l’appeler.

Service

C'est possible en Java et en OCaml, qui sont typés statiquement.

- un service est décrit non pas par un type abstrait,
- mais par un « type de service ».
    - interface de Java
    - type de fonction d'OCaml

C'est possible aussi en JavaScript, qui est typé dynamiquement.

Interface = description d'un service

Une interface de Java est un « type de service ».

Elle décrit les méthodes proposées, donc le service rendu :

```
public interface ICount {
  int next ();
  void reset ();
}
```

Elle ne donne pas accès aux champs, dont on ignore tout.

Sous-typage

Un objet de type `Count` est aussi un objet de type `ICount`.

```
ICount c = new Count (100, 10);
```

On oublie la nature exacte de l'objet, pour ne retenir que le service rendu.

Autre exemple, issu de la bibliothèque de Java :

```
Map<String, Integer> m1 = new TreeMap<String, Integer> ();
Map<String, Integer> m2 = new HashMap<String, Integer> ();
```

`m1` et `m2` rendent le même service. Ils sont interchangeables.

Une interface impose une abstraction... (*)

Supposons que la variable `c` a le type `ICount`.

`c.next()` et `c.reset()` sont les seuls moyens d'exploiter l'objet `c`. (*)

L'interface restreint les actions permises, donc protège l'objet.

Elle impose une abstraction. (*)

(*) ... enfin, presque

Malheureusement, ce n'est pas tout-à-fait vrai, car l'interface n'interdit pas de tester dynamiquement à quelle classe appartient l'objet :

```
if (c instanceof Count) {
  Count cc = (Count) c;
  ...
}
```

`instanceof` et le « downcast » sont dangereux au sens où ils violent l'abstraction qui (idéalement) devrait être offerte par l'interface `ICount`.

Heureusement, ces tests sont interdits (en dehors d'une certaine zone du code) si la classe `Count` n'est pas marquée `public`.

Type de fonction = description d'un service

En OCaml, un type de fonction est un « type de service ».

Par exemple, un compteur doté d'une seule opération `next` :

```
type counter = unit -> int
```

Un compteur doté d'opérations `next` et `reset` est une paire de fonctions :

```
type counter = (unit -> int) * (unit -> unit)
```

Ou bien un enregistrement à deux champs :

```
type counter = { next: unit -> int; reset: unit -> unit }
```

Un itérateur (dont l'état interne est modifiable) :

```
type 'a iterator = unit -> 'a option
```

Interlude

Implémentation du compteur comme paire de fonctions, en OCaml.
(Code en ligne)
Poly §1.2.2.

Une clôture est une abstraction

Deux points méritent d'être soulignés.

1. Une fonction a accès aux variables qui existent déjà à l'extérieur.
   - fonction = clôture
   - comment ça marche ? on va le voir dans la suite
2. Une fonction est une abstraction : elle est opaque.
   - on ne peut rien en faire, sauf l'appeler ; le typage statique le garantit

Une clôture est une abstraction

En JavaScript aussi, une clôture est une abstraction.

On ne peut rien en faire, sauf l'appeler : le typage dynamique le garantit.

La clôture porte une étiquette qui interdit l'inspection de son contenu.

Une clôture est une abstraction

Notre compteur OCaml peut être transcrit en JavaScript :

```
function Counter(init, step) {
  var value = init
  this.next = function() {
    var oldValue = value; value += step; return oldValue;
  }
  this.reset = function() { value = init; }
}
```

Une clôture est une abstraction

Ce compteur fonctionne comme espéré :

```
> c = new Counter (100, 10)
{ next: [Function], reset: [Function] }
> c.next()
100
> c.next()
110
> c.reset()
undefined
> c.next()
100
> c.next()
110
```

Une clôture est une abstraction

L'objet `c` n'a que deux champs, `next` et `reset`.

```
> c.value
undefined
```

Ces champs contiennent des clôtures, qui sont opaques.

```
> c.next
[Function]
> c.next.value
undefined
```

Impossible d'accéder à l'état du compteur, excepté via `next` et `reset`.

1 Services

2 Clôtures d’OCaml et objets de Java

3 Clôtures de Java et classes internes de Java

4 Des clôtures aux données

Vous avez dit « clôture » ?

Qu'est-ce au juste qu'une clôture ?

Deux points de vue, tous deux utiles :

1. Les mots « clôture » et « fonction » sont synonymes.
   - une fonction a accès aux variables (immuables) définies à l'extérieur
   - (on les appelle variables libres)
   - c'est « magique », ou « naturel »
2. Une clôture est un bloc de mémoire alloué dans le tas.
   - avec une étiquette et des champs
   - donc un comportement et un état

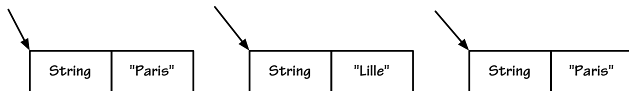
Exemple

Illustrons ces deux points de vue à l'aide d'un nouvel exemple :

le partage maximal de chaînes de caractères.

Égalité des adresses ou des contenus ?

Deux blocs distincts dans le tas peuvent avoir le même contenu.



Si ce sont, par exemple, des chaînes de caractères, ils peuvent être :

- distincts au sens de l'égalité des adresses
  - `x == y` en Java et en OCaml
- égaux au sens de l'égalité des contenus
  - `x.equals(y)` en Java
  - `x = y` en OCaml

De nombreux blocs peuvent contenir la même chaîne, p.ex. `"Paris"`.

Partage maximal

Dotons-nous d'un service de partage maximal de chaînes de caractères.

Cela s'appelle string interning.

Dans le cas général, le partage maximal s'appelle aussi hash-consing.

Il joue un rôle essentiel dans les BDDs.

Filliâtre et Conchon (2006) expliquent le hash-consing en OCaml.

Description du service

Appelons « un interne » une fonction `i` qui attend une chaîne `s` et renvoie une chaîne de même contenu (donc, `i s = s`).

```
type intern = string -> string
```

Imposons de plus que `i` effectue un partage maximal des chaînes qu'elle renvoie (donc, si `s1 = s2`, alors `i s1 == i s2`).

Voyez-vous comment implémenter un tel service ?

Interlude

Implémentation du partage maximal
d'abord comme une clôture en OCaml,
puis comme un objet en Java.
(Code en ligne)
Poly §1.2.2.

À retenir (1)

Dans les deux versions (OCaml et Java) :

- la table de hachage est inaccessible, sauf via le service ;
- il en découle que son contenu évolue de façon monotone,
- ce qui est nécessaire pour que `s1 = s2` implique `i s1 == i s2`.

L'abstraction est donc nécessaire pour garantir le partage maximal.

À retenir (2)

La comparaison entre OCaml et Java éclaire la nature des clôtures :

- en OCaml, le service est une clôture ;
  - elle a « naturellement » accès à la table de hachage
- en Java, nous l'avons traduite en un objet.
  - alloué dans le tas
  - son étiquette est le nom de la classe, `Intern`
  - son champ contient un pointeur vers la table de hachage

Le compilateur OCaml effectue pour nous cette traduction. Donc,

- une clôture OCaml est allouée dans le tas
- ses champs contiennent les informations dont elle a besoin

1 Services

2 Clôtures d’OCaml et objets de Java

3 Clôtures de Java et classes internes de Java

4 Des clôtures aux données

Fonctions/clôtures en Java

Java 8 offre des fonctions anonymes, ou clôtures.

On peut les comprendre :

- soit par analogie avec les fonctions d'OCaml ;
- soit par traduction vers des objets de Java.

Clôtures versus objets

Une fonction OCaml offre un seul service : on peut l'appeler, et c'est tout.

```
type intern = string -> string
```

Elle est donc analogue à un objet Java qui offre une seule méthode.

```
public interface IIntern {
  String intern (String s);
}
```

Un type de fonction d'OCaml correspond à
une interface à une seule méthode de Java.

Java 8 appelle cela une interface fonctionnelle.

Clôtures versus objets

Voici un autre exemple. Le type OCaml d'une fonction de comparaison :

```
type 'a comparator = 'a -> 'a -> int
```

correspond à l'interface `Comparator<T>` de la bibliothèque de Java :

```
public interface Comparator<T> {
  int compare (T o1, T o2);
}
```

Ce type et cette interface sont paramétrés par un type (noté `'a` ou `T`).

Une fonction OCaml de type `string comparator` est
analogue à un objet Java de type `Comparator<String>`.

Clôtures versus objets

L'interface `Function<T, R>` est la plus abstraite possible :

```
public interface Function<T, R> {
  R apply (T argument);
}
```

Elle est définie dans la bibliothèque de Java 8.

Une fonction OCaml de type `string -> int` est analogue
à un objet Java de type `Function<String, Integer>`.

Scala considère une fonction comme un objet de type Function1.

Définition de fonctions (OCaml)

OCaml offre une syntaxe concise pour définir des fonctions.

L'expression `fun x -> x + 1` construit une fonction de type `int -> int`.

On peut lui donner un nom :

```
let id = fun x -> x + 1
```

ou si l'on préfère :

```
let id x = x + 1
```

Comment, en Java, construit-on un objet
de type `Function<Integer, Integer>` ?

Définition de fonctions (Java)

Comment construit-on un objet de type `Function<Integer, Integer>` ?

- soit « à la main », via une classe qui implémente cette interface,
- soit en écrivant une fonction anonyme de Java 8.

Le code sera le même : une fonction anonyme est un sucre (une notation).

Deux exemples suivent...

Définition de fonctions, à la Java 1.0

En principe, il faut définir une classe qui implémente l'interface `Function` :

```
public class Successor implements Function<Integer, Integer> {
  public Integer apply (Integer x) { return x + 1; }
}
```

Alors on peut instancier cette classe, c'est-à-dire en créer un objet :

```
Function<Integer, Integer> id = new Successor ();
```

C'est simple (en principe). Et assez lourd.

Définition de fonctions, à la Java 1.1

Heureusement (?), on peut définir et instancier aussitôt une classe :

```
Function<Integer, Integer> id =
  new Function<Integer, Integer> () {
    public Integer apply (Integer x) { return x + 1; }
  };
```

Le mot `new` est suivi ici d'une interface,
complétée aussitôt par le code de la méthode `apply`.

La classe est alors anonyme : le nom `Successor` a disparu.

C'est toujours simple (en principe). Et toujours lourd.

Définition de fonctions, à la Java 8

Heureusement, il existe à présent une syntaxe beaucoup plus concise :

```
Function<Integer, Integer> id =
  x -> { return x + 1; };
```

ou encore mieux, dans ce cas :

```
Function<Integer, Integer> id =
  x -> x + 1;
```

Ouf ! Voilà un sucre agréable.

Cela fonctionne non seulement pour l'interface `Function`,
mais aussi pour toute interface fonctionnelle.

Clôtures avec variables libres

La fonction `fun x -> x + 1` est un exemple simpliste, car elle n'utilise aucune variable définie à l'extérieur.

Or, en général, cela est permis, comme ici en OCaml :

```
let tax threshold rate : float -> float =
  fun price ->
    if price < threshold then
      price
    else
      threshold +. (price -. threshold) *. rate
```

La fonction anonyme `fun price -> ...` a deux variables libres, à savoir `threshold` et `rate`.

Clôtures avec variables libres

Un appel à `tax` produit une clôture `f` de type `float -> float`.

Ici, dans une session interactive `ocaml` :

```
# let f = tax 100. 1.05;;
val f : float -> float = <fun>
# f 90.;;
- : float = 90.
# f 110.;;
- : float = 110.5
```

D'une façon ou d'une autre, `f` a mémorisé que pour elle, `threshold` vaut `100.` et `rate` vaut `1.05`.

Clôtures avec variables libres

On peut écrire en Java 8 une fonction `tax` analogue.

Un appel à `tax` renvoie une clôture :

```
Function<Float, Float> tax (float threshold, float rate) {
  return (price) -> {
    if (price < threshold) return price;
    return threshold + (price - threshold) * rate;
  };
}
```

Comment ça marche ? Éliminons le sucre pour comprendre...

(Mêmes étapes que précédemment, expliquées cette fois en sens inverse.)

Clôtures avec variables libres

D'abord, on revient à une classe interne anonyme de Java 1.1 :

```
Function<Float, Float> tax (float threshold, float rate) {
  return new Function<Float, Float> () {
    public Float apply (Float price) {
      if (price < threshold) return price;
      return threshold + (price - threshold) * rate;
    }
  };
}
```

Comment ça marche ? On élimine encore un sucre pour comprendre...

Clôtures avec variables libres

On revient à une classe ordinaire de Java 1.0 :

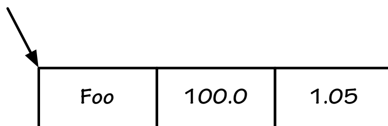
```
Function<Float, Float> tax (float threshold, float rate) {
  return new Foo (threshold, rate);
}
```

On définit séparément cette classe `Foo` :

```
class Foo implements Function<Float, Float> {
  final float threshold, rate;          // Fields.
  Foo (float threshold, float rate) { // Constructor.
    this.threshold = threshold; this.rate = rate; }
  public Float apply (Float price) {  // Code.
    if (price < threshold) return price;
    return threshold + (price - threshold) * rate;
  }
}
```

Elle a besoin de deux champs, sans quoi on ne peut écrire `apply`.

Clôtures avec variables libres



On comprend alors que notre clôture est représentée ainsi en mémoire :

- un pointeur de code ($\sim$ une étiquette), ici `Foo` ;
  - elle détermine une méthode `apply`, ici celle de la classe `Foo`
- des champs, ici `threshold` et `rate` ;
  - ils contiennent des données dont la méthode `apply` a besoin

Clôtures avec variables libres

On comprend aussi que :

- construire une clôture `(price) -> { ... }`,
  c'est en fait allouer un bloc `new Foo (...)`,
- et ce bloc a autant de champs
  que la clôture a de variables libres.

1 Services

2 Clôtures d'OCaml et objets de Java

3 Clôtures de Java et classes internes de Java

4 Des clôtures aux données

Défonctionalisation

Une clôture est un bloc de mémoire.

On peut donc être tenté de la considérer

- non pas comme un « objet » mariant données et comportement,
- mais comme une donnée pure.

Pour clarifier cette idée, effectuons (encore) une traduction, appelée défonctionalisation (Reynolds, 1972).

Elle va nous permettre de mieux comprendre comment les clôtures forment des structures de données en mémoire.

Exemple

Illustrons la défonctionalisation à l'aide d'un exemple :

des ensembles implémentés comme des fonctions.

Ensembles comme fonctions

On peut représenter un ensemble par sa fonction caractéristique.

```
type set =
  int -> bool
let empty : set =
  fun y -> false
let singleton (x : int) : set =
  fun y -> y = x
let union (s1 : set) (s2 : set) : set =
  fun y -> s1 y || s2 y
let mem (x : int) (s : set) : bool =
  s x
```

Ça marche. Mais est-ce une idée intelligente... ou pas ?

Ensembles comme fonctions

On peut représenter un ensemble par sa fonction caractéristique.

```
type set =
  int -> bool
let empty : set =
  fun y -> false
let singleton (x : int) : set =
  fun y -> y = x
let union (s1 : set) (s2 : set) : set =
  fun y -> s1 y || s2 y
let mem (x : int) (s : set) : bool =
  s x
```

Ça marche. Mais est-ce une idée intelligente... ou pas ?

`empty`, `singleton`, `union` travaillent en temps $O(1)$. Quid de `mem` ?

Ensembles comme fonctions

On peut représenter un ensemble par sa fonction caractéristique.

```
type set =
  int -> bool
let empty : set =
  fun y -> false
let singleton (x : int) : set =
  fun y -> y = x
let union (s1 : set) (s2 : set) : set =
  fun y -> s1 y || s2 y
let mem (x : int) (s : set) : bool =
  s x
```

Ça marche. Mais est-ce une idée intelligente... ou pas ?

`empty`, `singleton`, `union` travaillent en temps $O(1)$. Quid de `mem` ?

Pour répondre à cette question, il faut comprendre la structure des clôtures que nous construisons.

Ensembles comme fonctions

On construit une clôture de type `set` en trois points du code :

```
let empty : set =
  fun y -> false
let singleton (x : int) : set =
  fun y -> y = x
let union (s1 : set) (s2 : set) : set =
  fun y -> s1 y || s2 y
```

Quels champs ont les clôtures construites en ces trois points ?

- dans `empty`, zéro champ ;
- dans `singleton`, un champ de type `int` correspondant à `x` ;
- dans `union`, deux champs de type `set` correspondant à `s1` et `s2`.

Ces trois sortes de clôtures ont trois étiquettes distinctes, disons `E`, `S`, `U`.

Ensembles comme fonctions

Un « ensemble » est en fait une clôture de l'une de ces trois sortes.

On pourrait aussi bien définir `set` comme un type algébrique :

```
type set =
  | E
  | S of int
  | U of set * set
```

Et définir ainsi les trois opérations de construction :

```
let empty : set =                               E
let singleton (x : int) : set =                 S (x)
let union (s1 : set) (s2 : set) : set = U (s1, s2)
```

La représentation en mémoire des ensembles serait la même !

Or, que représente ce type ?

Ensembles comme fonctions

Un « ensemble » est en fait une clôture de l'une de ces trois sortes.

On pourrait aussi bien définir `set` comme un type algébrique :

```
type set =
  | E
  | S of int
  | U of set * set
```

Et définir ainsi les trois opérations de construction :

```
let empty : set =                              E
let singleton (x : int) : set =                S (x)
let union (s1 : set) (s2 : set) : set = U (s1, s2)
```

La représentation en mémoire des ensembles serait la même !

Or, que représente ce type ?

C'est un type d'arbres avec des feuilles `E` et `S` et des nœuds binaires `U`.

Ensembles comme fonctions

Chaque sorte de clôture a sa propre « méthode » `apply`.

Puisque nous avons trois sortes de clôtures, leurs « méthodes » `apply` se combinent en une fonction `apply` où on distingue trois cas :

```
let rec apply (s : set) (y : int) : bool =
  match s with
  | E          -> false
  | S (x)      -> y = x
  | U (s1, s2) -> apply s1 y || apply s2 y
```

Le test d'appartenance à un ensemble devient alors :

```
let mem (x : int) (s : set) : bool =
  apply s x
```

Intelligent ou pas ?...

Un ensemble est en fait un arbre non équilibré !

`mem s x` parcourt l'arbre `s` à la recherche de `x`.

La complexité de `mem` est donc $O(n)$. Inefficace !

Il faut comprendre les clôtures pour pouvoir analyser ce code.

Notes historiques

Le concept de clôture, combinant code et données, est dû à Landin (1964).

En Scheme (1975), les fonctions sont compilées sous forme de clôtures.

Il en va de même aujourd'hui en OCaml, JavaScript (1995), Scala (2003), C# 2.0 (2005), Rust (2010), C++ (2011), Java 8 (2014), etc.

La notion d'objet remonte à Simula 67.

À retenir

On peut avoir une double compréhension des objets et des clôtures.

Vision de haut niveau :

- Une clôture ou un objet est une abstraction ; elle offre un service ;
- Une clôture a « naturellement » accès à ses variables libres.

Vision de plus bas niveau :

- Une clôture est un bloc de mémoire, avec étiquette et champs ;
- Il faut un champ pour chaque variable libre.