Les objets et les classes d'Objective Caml

D.E.A. Programmation: Sémantique, Preuves et Langages

Xavier Leroy
INRIA Rocquencourt
Xavier.Leroy@inria.fr
http://caml.inria.fr/

Généralités sur les objets

Un objet regroupe ensemble

- Un état interne:
  - une ou plusieurs variables d'instance;
  - accessibles seulement aux méthodes de l'objet;
  - mutables ou non.
- Une collection de méthodes:
  - des fonctions qui opèrent sur l'état interne;
  - accessibles de l'extérieur;
  - assurent l'interface entre l'extérieur et l'état interne.

La seule opération sur un objet est l'envoi de méthode, noté

objet#*méthode* [*arguments*]

Exemple: un objet “compteur”

État interne:

- Valeur courante du compteur
- Valeur initiale
- Valeur maximale

Méthodes:

- Lire la valeur courante
- L’incrémenter
- La remettre à zéro

Classes et objets

Une classe est un moule à objets:

- Elle définit les méthodes.
- Elle déclare les variables d'instances et définit leur valeur initiale.
- Elle prend en argument un ou plusieurs paramètres d'initialisation qui servent à déterminer les valeurs initiales des variables d'instance.

La construction `new` renvoie un nouvel objet – une nouvelle instance de la classe:

`new` *nom-de-classe paramètres initiaux*

La classe “compteur”

```
class compteur mini maxi =
  object
    val mutable v = mini

    method valeur = v

    method incr =
      if v < maxi then v <- v + 1 else failwith "incr"

    method raz =
      v <- mini
  end

let c = new compteur 1 10
     val c : compteur = <obj>

m#incr; m#incr; m#valeur
     - : int = 3
```

Exemple long: les entrées-sorties unifiées

Une classe qui encapsule les impressions sur le terminal:

```
class standard_output =
  object
    method put_char c = print_char c
    method put_string s = print_string s
    method put_int n = print_int n
  end

let stdout = new standard_output
     val stdout : standard_output = <obj>

stdout#put_string "Hello!\n"
     Hello!
     - : unit = ()
```

Polymorphisme sur les objets

```
let euros obj francs =
  obj#put_int francs;
  obj#put_string " francs = ";
  obj#put_int (francs * 100 / 655);
  obj#put_string " euros\n"

     val euros :
     < put_int : int -> 'a; put_string : string -> 'b; .. > -> int -> 'b
```

`euros` s'applique à n'importe quel objet possédant au moins les méthodes `put_int` et `put_string` avec les types indiqués.

```
euros stdout 322;;
     322 francs = 49 euros
     - : unit = ()
```

Quel est le type de `stdout`?

La définition de la classe `standard_output` définit également une abréviation de type pour le type des objets de cette classe:

```
type standard_output =
    < put_char : char -> unit;
      put_string : string -> unit;
      put_int : int -> unit >
```

Sorties sur fichiers

```
class fileout filename =
  object
    val chan = open_out filename
    method put_char c = output_char chan c
    method put_string s = output_string chan s
    method put_int n = output_string chan (string_of_int n)
    method seek pos = seek_out chan pos
    method close = close_out chan
  end
```

En plus des opérations `put_` vues précédemment, on ajoute la possibilité de positionnement à l'intérieur du fichier (`seek`) et la fermeture explicite (`close`).

Utilisation de la classe `fileout`:

```
let f = new fileout "/tmp/myfile";;
f#put_string "Hello, world!\n";;
euros f 1234;;
f#seek 7;;
f#put_string "monde";;
f#close;;
```

L'appel `euros f 1234` est bien typé car `f` fournit les méthodes `put_string` et `put_int` requises (et d'autres).

Implémentation de méthodes par défaut

Lorsqu'on sait faire `put_char`, on sait également implémenter `put_string` (imprimer chaque caractère) et `put_int` (convertir l'entier en chaîne).

Cependant, sur certains types de sorties, on dispose d'implémentations plus efficaces (p.ex. `put_string` sur un fichier, où l'on peut faire des écritures par blocs).

- Utilisation d'une classe virtuelle pour définir les implémentations par défaut une fois pour toute.
- Utilisation de l'héritage entre classes pour récupérer les implémentations par défaut dans une classe de sortie particulière.
- Redéfinition dans la classe de sortie des méthodes pour lesquelles on a une meilleure implémentation.

La classe virtuelle des “écrivains”

```
class virtual writer =
  object (self)
    method virtual put_char : char -> unit
    method put_string s =
      for i = 0 to String.length s - 1 do self#put_char s.[i] done
    method put_int n =
      self#put_string (string_of_int n)
  end
```

Cette classe est marquée “virtuelle” car certaines de ses méthodes ne sont pas définies (`put_char`); on ne peut donc pas créer d’instances de cette classe.

L’identificateur `self` est lié (au moment de l’appel de méthode) à l’objet lui-même. Il permet donc à une méthode de rappeler une autre méthode du même objet.

De `writer` vers `fileout` par héritage

Une implémentation simple des sorties sur fichiers hérite `put_string` et `put_int` de `writer`, et définit uniquement les autres méthodes:

```
class fileout filename =
  object
    inherit writer
    val chan = open_out filename
    method put_char c = output_char chan c
    method seek pos = seek_out chan pos
    method close = close_out chan
  end
```

Cette classe se comporte exactement comme la classe `fileout` vue précédemment (implémentée entièrement à la main):

```
let f = new fileout "/tmp/myfile";;
f#put_string "Hello, world!\n";;
f#close;;
```

De `writer` vers `fileout`, optimisé

Dans la classe `fileout`, nous pouvons fournir une implémentation plus efficace de `put_string`, qui va masquer l'implémentation par défaut héritée de `writer`:

```
class fileout filename =
  object
    inherit writer
    val chan = open_out filename
    method put_char c = output_char chan c
    method put_string s = output_string chan s
    method seek pos = seek_out chan pos
    method close = close_out chan
  end
```

Nous conservons l'implémentation par défaut de `put_int`, car elle est suffisamment efficace.

La liaison tardive (récursion ouverte)

Les liaisons entre méthodes sont déterminées au moment où l'on fait `new`, et non pas à la définition des classes. C'est ce qu'on appelle la liaison tardive, par contraste avec la liaison statique en usage dans le reste du langage.

```
class c =
  object (self)
    method pair n =                     let rec pair n =
      n=0 || self#impair(n-1)             n=0 || impair(n-1)
    method impair n =                   and impair n =
      n=1 || self#pair(n-1)               n=1 || pair(n-1)
  end

class d =
  object
    inherit c
    method pair n =                     let pair n =
      (n mod 2) = 0                       (n mod 2) = 0
  end

let o = new d() in o#impair 1000001     impair 1000001
(* termine immédiatement *)             (* itère 1000001 fois*)
```

Sous-typage sur les objets

Si $A : \langle m_1 : \tau_1;\ \ldots;\ m_k : \tau_k \rangle$
et $B : \langle m_1 : \tau_1;\ \ldots;\ m_k : \tau_k;\ m_{k+1} : \tau_{k+1};\ \ldots \rangle$
alors B peut être utilisé partout à la place de A.

On dit que le type de B est *sous-type* du type de A
(cf. la notion de sous-ensemble).

Application: on peut mettre des objets de différent types dans une structure de donnée homogène (p.ex. liste, tableau), à condition de voir tous ces objets comme appartenant à leur super-type commun.

Sous-typage en Objective Caml

```
let w1 = new fileout "/tmp/myfile" in
let w2 = new stringout in

let writers = [(w1 :> writer); (w2 :> writer)] in

List.iter (fun w -> w#put_string "Hello") writers;
w1#close;
w2#contents
```

En Objective Caml, le sous-typage doit être marqué explicitement comme une coercion de types:

(*objet* :> *type*)

(*objet* : *type-source* :> *type-résultat*)

Polymorphisme vs. sous-typage

Le polymorphisme permet d'avoir des fonctions s'appliquant à des arguments de différents types, mais pas des structures de données hétérogènes:

`List.rev [1;2];;`	*- : int list = [2;1]*
`List.rev [true;false];;`	*- : bool list = [false;true]*
`[ [1]; [true] ];;`	*Erreur de type*

`int list` et `bool list` n'ont pas de supertype commun.

Seul le sous-typage permet les structures de données hétérogènes.

Faire référence à la superclasse

Après les sorties génériques, faisons maintenant les entrées sur le même moule:

```
class virtual reader =
  object (self)
    method virtual get_char : char
    method get_string n =
      let s = String.create n in
      for i = 0 to n - 1 do s.[i] <- self#get_char done;
      s
    method get_int =
      let rec read accu =
        match self#get_char with
          '0'..'9' as c -> read (10 * accu + Char.code c - Char.code '0')
        | _ -> accu
      in read 0
  end
```

Problème: `get_int` lit un caractère de trop!

Faire référence à la superclasse, suite

C'est facile à corriger si l'on sait faire `seek` sur l'entrée: on revient en arrière d'un caractère après avoir lu l'entier.

```
class virtual seekable_reader =
  object (self)
    inherit reader as super
    method virtual seek : int -> unit
    method virtual pos : int  (* position courante de lecture *)
    method get_int =
      let res = super#get_int in
      self#seek (self#pos - 1);
      res
  end
```

Le `as super` donne le nom `super` aux méthodes de la superclasse. `super#get_int` fait alors référence à l'implémentation de `get_int` dans la superclasse. On peut ainsi redéfinir une méthode héritée tout en réutilisant l'implémentation héritée.

Lecture sur fichier

```
class filein filename =
  object
    inherit seekable_reader
    val chan = open_in filename
    method get_char = input_char chan
    method seek pos = seek_in chan pos
    method pos = pos_in chan
    method close = close_in chan
  end
```

Héritage multiple

Une classe peut hériter de plusieurs autres classes.

```
class pipe =
  object
    val q = Queue.create ()
    inherit writer
    inherit reader
    method put_char c = Queue.add c q
    method get_char =
      try Queue.take q
      with Queue.Empty -> raise End_of_file
  end
```

Une instance de `pipe` est utilisable comme écrivain aussi bien que comme lecteur:

```
let p = new pipe;;
euros p 100;;
p#get_string 13;;             - : string = "100 francs = "
p#get_int;;                   - : int = 15
```

Sous-typage et héritage

Le sous-typage:

- permet de considérer des implémentations différentes avec la même interface;
- autorise les fonctions génériques;
- autorise le mélange de différentes classes dans une même structure de données.

L'héritage:

- permet la programmation incrémentale et la réutilisation de code;
- apporte la liaison tardive.

Contrairement à la plupart des langages à objets, OCaml dissocie complètement sous-typage et héritage:

- Une classe A peut être sous-type d'une classe B sans hériter de B.
  (Implémentations indépendantes d'une même interface, ou d'interfaces en relation de sous-typage.)

- Une classe A peut hériter de B sans être sous-type de B.
  (Cas rare, se produit avec selftype $+$ méthodes binaires.)

Selftype

En général, il est incorrect de redéfinir une méthode dans une sous-classe avec un type différent de celle qu'elle a dans la super-classe:

```
class c =
  object (self)
    method a = 1 + self#b
    method b = 2
  end
```

```
class d =
  object (self)
    inherit c
    method b = "toto"
  end
```

En effet, le corps de la méthode `a` est typé sous l'hypothèse que `self#b` renvoie un `int`. La redéfinition de `b` invalide cette hypothèse.

Cependant, on peut sans risque permettre des méthodes dont le type fait intervenir le type de l'objet courant:

```
class point =
  object (self: 'selftype)
    val x = 0
    method coord = x
    method translate d = ( {< x = x + d >} : 'selftype)
    method m1 = ... (self#translate 5)#coord ...
  end

class colorpoint =
  object (self: 'selftype)
    inherit point as super
    val c = "black"
    method color = c
    method m2 = ... (self#translate 5)#color ...
  end
```

`self#translate` renvoie un objet de type `point` si `self` est de type `point`, et de type `colorpoint` si `self` est de type `colorpoint`. La méthode `m1` est bien typée dans `point` et le reste dans les classes qui héritent de `point`.

Inheritance is not subtyping (Cook et al, 1989)

```
class point =
  object (self: 'selftype)
    val x = 0
    method coord = x
    method equal (p : 'selftype) = (p#coord = x)
  end

class colorpoint =
  object (self: 'selftype)
    inherit point as super
    val c = "black"
    method colour = c
    method equal (p : 'selftype) = (p#coord = x) && (p#colour = c)
  end
```

`colorpoint` ne doit pas être un sous-type de `point`; sinon, le code erroné suivant passerait le typage:

```
let p = new point and cp = (new colorpoint :> point) in cp#equal p
```

Classes paramétrées

Tout comme les types concrets, les classes peuvent être paramétrées par un ou plusieurs types, ce qui leur ajoute du polymorphisme.

Exemple typique: les classes “conteneurs”.

```
class ['a, 'b] hashtable size =
  object
    val buckets = (Array.make size [] : ('a * 'b) list array)
    method find key =
      let h = (Hashtbl.hash key) mod (Array.length buckets) in
      List.assoc key buckets.(h)
    method add key data =
      let h = (Hashtbl.hash key) mod (Array.length buckets) in
      buckets.(h) <- (key, data) :: buckets.(h)
end
```

Les types concrets à la sauce objet

```
class virtual ['a] list =
  object (self)
    method virtual isnil : bool
    method virtual head : 'a
    method virtual tail : 'a list
    method length =
      if self#isnil then 0 else 1 + self#tail#length
  end
```

```
class ['a] nil =
  object
    inherit ['a] list
    method isnil = true
    method head = failwith "nil"
    method tail = failwith "nil"
  end
```

```
class ['a] cons h0 t0 =
  object
    val h = h0
    val t = t0
    inherit ['a] list
    method isnil = false
    method head = h
    method tail = t
  end
```

Codage populaire en Java, mais peu utile en Caml car les types concrets du langage sont généralement plus commodes.