Les objets en Ocaml: constructions du langage

María-Virginia Aponte

April 1, 2003

1 Introduction

Ocaml est un langage de la famille ML avec des traits fonctionnels et impératifs et étendu par des constructions objets. Il est développé à l'INRIA depuis 1995. Plusieurs raisons justifient le choix d'Ocaml pour l'apprentissage de la programmation objet:

- Ocaml possède les traits habituels des langages objets: classes, instances, héritage multiple et liaison retardée. Il propose aussi des constructions plus évoluées: parmi celles-ci les classes abstraites et les classes paramétrées.
- Ocaml est fortement typé avec des types inféres et polymorphisme. Cela en fait un langage sûr et facile à employer.
- Les objets sont des valeurs comme les autres et se mélangent sans contraintes aux autres constructions du langage. En particulier, les objets se mélangent au système de types, ce qui résulte en un langage objet avec polymorphisme et inférence de types typé statiquement.
- Le système de types d'Ocaml propose une solution originale à une classe importante de problèmes de typage des langages objets.
- Ocaml possède un glanneur de cellules (*garbage collector*), et des nombresuses bibliothèques. Il peut exécuter des processus légers (*threads*) et communique sur Internet (ouverture de canaux de communication, applications client-serveur, etc.). Il offre également la possibilité d'intéragir avec le langage C.
- Ocaml est un langage compilé qui possède une boucle intéractive: les tests y sont plus faciles. On peut également produire du code exécutable portable (*bytecode*) ou natif, dont l'éfficacité est proche du code C dans certains cas.

2 Objets et classes

La notion d'objet en programmation est empruntée au monde de la simulation. "L'approche de programmation orientée objet est basée dans l'analogie entre un programme qui simule un système physique et le système physique lui-même. Par analogie avec les composants du système physique, les composants du programme sont appelés *objets*[1]".

La plupart des langages orientés objets sont basés dans la notion de *classe*. Une classe est un description ou spécification de tous les objets générés à partir d'elle. Elle regroupe les données des objets futurs (*variables d'instance*) avec les opérations de traitement de ces données (*méthodes*).

```
class one =
 object
    val x = 1
    method two = x+1
 end;;
```

[1] Cardelli: *A theory of types*.

La classe `one` contient une variable d'instance `x` (constante) et une méthode `two` (constante aussi) qui opère sur cette variable. Lorsqu'une variable est déclaré `mutable`, il est possible de modifier sa valeur. C'est le cas de la variable `x` dans la classe `cell` plus bas. Sans cette déclaration, la valeur d'une variable est celle calculée lors de la création de l'objet et reste constante. La classe `cell` décrit les données et les opérations pour manipuler des cellules modifiables contenant un entier. La variable modifiable `x` de type `int` est initialisée à zéro. Deux méthodes `get` et `set` traitent le contenu d'une cellule.

```
class cell =
 object
     val mutable cont = 0
     method get  = cont
     method set n = cont <- n
 end;;
```

Une classe a une *interface* ou *type* inféré par le compilateur. Il décrit les types des paramètres éventuels de la classe, et des variables et des méthodes qui la composent. L'interface inférée pour `cell` est:

```
class cell :
  object
    val mutable cont : int
    method get : int
    method set : int -> unit
end
```

Une classe peut avoir des paramètres. La classe `point` prend en argument les coordonnées initiales du point. Les variables d'instance `x` et `y` correspondent aux coordonnées du point et sont initialisées aux valeurs des paramètres. Cette classe possède aussi deux methodes d'accèss aux coordonnées, une méthode de déplacement du point et une méthode d'affichage.

```
class point (xinit, yinit) =
 object
   val mutable x = xinit
   val mutable y = yinit

   method getx = x
   method gety = y
   method moveto (dx, dy) = begin x <- x + dx; y <- y + dy end
   method print () =
     begin
       print_string"( ";
       print_int x;
       print_string", ";
       print_int y;
       print_string" )"
     end
  end;;
```

L'interface inférée pour cette classe est fonctionnelle: elle est paramétrée par un couple d'entiers:

```
class point :
  int * int ->
  object
    val mutable x : int
    val mutable y : int
    method getx : int
    method gety : int
    method moveto : int * int -> unit
    method print : unit -> unit
  end
```

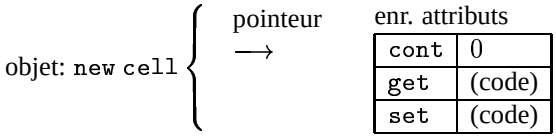
2.1 Création d'objets

Un objet est créé à partir d'une classe *c* avec la syntaxe `new` *c* $a_1\ a_2 \ldots a_n$, où *c* est la classe génératrice et $a_1 a_2 \ldots a_n$ sont les (éventuels) paramètres d'initialisation. L'objet créé est dit *instance de la classe c* ou simplement *de classe c*.

```
let c1 = new cell;;
val c1 : cell = <obj>

let p1 = new point(0,1);;
val p1 : point = <obj>
```

L'objet `c1` est un instance de `cell` et `p1` est une instance de `point` avec `(0,1)` en coordonnées initiales. Chaque nouvel objet contient toutes les variables et les méthodes de sa classe génératrice. Cet ensemble de variables et méthodes est couramment nommé *attributs* de la classe (ou de l'objet).



Le comportément lors de la création d'un objet peut être expliqué de la manière suivante[2] : `new` *c* alloue un *enregistrement d'attributs* qui contient les valeurs initiales des variables, et le code des méthodes de *c*, et retourne en résultat un pointeur vers celui-ci. Des exécutions différentes de `new` produisent des objets différents, c.a.d., des pointeurs vers des enregistrements d'attributs différents. Ainsi, bien que `p2` plus bas soit une instance de `point` initialisée aux mêmes valeurs que `p1`, il s'agit de deux objets différents.

```
let p2 =  new point(0,1);;
val p2 : point = <obj>

p1 = p2;;
- : bool = false
```

De plus, les variables de tous les objets d'une classe évoluent séparemment: chaque objet possède des variables séparées (ce qu'on nomme parfois, *l'état de l'objet*).

En Ocaml, bien que l'enregistrement d'attributs d'un objet contienne éffectivement tous les attributs de sa classe, *seules les méthodes sont visibles en dehors de l'objet*. Ni les variables, et encore moins les valeurs des paramètres ne peuvent être extraites de l'objet (pour plus des détails, voir partie suivante).

Ce masquage des variables (qui revient à *protéger l'accèss aux données* de l'objet), se reflète dans le type inféré pour un objet: il spécifie seulement les méthodes de sa classe génératrice. Il faut donc distinguer entre le type ou interface d'une classe, et le type des objets crées à partir de celle-ci. Dans la première sont spécifiés tous les attributs de la classe; dans le deuxième, seulement les méthodes. Prenons l'exemple de la classe `cell`. Son interface est

```
class cell :
  object
    val mutable x : int
    method get : int
    method set : int -> unit
end
```

[2] Le modèle de stockage décrit ici est naif. Par exemple, dans certains langages, le code des méthodes de tous les objets d'une même classe est partagé.

Un objet de cette classe, a pour type une version de l'interface restreinte aux spécifications des méthodes. Si l'on note *Type_Instance_de(c)* le type des objets de classe `c`, alors le type de `c1` est donné par

`c1 :` *Type_Instance_de*`(cell)`

En Ocaml, ce type correspond à la spécification

Type_Instance_de`(cell) = < get : int; set : int -> unit >`

mais, dans la syntaxe Ocaml, *Type_Instance_de*`(cell)` est noté `cell`. Ceci explique le message

```
let c1 = new cell;;
val c1 : cell = <obj>
```

qu'il faut comprendre

```
val c1 : < get : int; set : int -> unit > = <obj>
```

2.2 Envoi de messages

L'utilisation d'une méthode dans un objet se fait par l'*envoi des messages*. Le message envoyé est la méthode demandée, et le destinataire l'objet qui la possède. On utilise le symbole # pour l'envoi des messages (le symbole `.` étant déjà utilisé pour l'accès aux champs d'enregistrements et aux composants d'un module). Dans l'exemple qui suit, le message `get` est envoyé à l'objet `c1`

```
# c1#get;;
- : int = 0
```

L'exemple suivant montre l'envoi du message `set` qui attend un paramètre de type `int` au même objet `c1` (voir interface de la classe `cell` plus haut).

```
# c1#set 3;;   (   * autre syntaxe: c1#set(3)  *)
- : unit = ()

# c1#get;;
- : int = 3

# c1#cont;;
This expression has type cell
It has no method cont
```

La dernière erreur montre qu'on ne peut pas extraire la variable `x` contenue dans l'objet `c1` elle n'est pas visible à l'extérieur de l'objet.

2.3 Visibilité dans les objets et classes

Jusqu'ici les règles de visibilité sont données par:

- dans un objet, les variables d'instance sont systématiquement masquées alors que les méthodes sont toujours visibles (mais on peut les masquer par la spécification `private`).
- dans le corps d'une classe, ses paramètres sont visibles par les variables d'instance et par les méthodes; les variables d'instance sont visibles par les méthodes.

Dans l'exemple suivant, il est nécessaire de définir une méthode `getx` pour récuperer à l'extérieur de l'objet la valeur de la variable x.

```
#class point init =
  object
   val mutable x = init
   method getx = x
   method move d = x <- x+d
  end;;

# let p = new point 7;;

# p#getx;;
- : int = 7
# p#move 3;;
- : unit = ()
# p#getx;;
- : int = 10
```

2.4 Manipulation d'objets

En Ocaml, les objets sont des valeurs comme les autres, et à ce titre ils peuvent être envoyés en paramètre ou en résultat des fonctions, ou faire partie des structures des données plus complexes. Par exemple, on peut reutiliser le code contenu dans un objet en utilisant l'abstraction fontionnelle classique:

```
# let modify (c:cell) (v:int) = c#set v;;
val modify : cell -> int -> unit = <fun>

# modify c1 2;;
- : unit = ()

# c1#get;;
- : int = 2
```

De même, les objets sont manipulables par les constructions polymorphes d'Ocaml:

```
let l = [p1;p2];;
val l : point list = [<obj>; <obj>]

List.tl l;;
- : point list = [<obj>]
```

3 Héritage

En programmation objet, on appele *héritage* la re-utilisation d'attributs d'une classe existante pour en dériver une nouvelle de *manière incrémentale*. La classe existante est appelée *classe parente* ou *super-classe*, alors que la nouvelle est dite *dérivée* ou *sous-classe*. La classe dérivée, comme toute classe, décrit le comportément des objets futurs, mais cette description est incrémentale: on ne donne que les modifications ou extensions par rapport aux comportéments et données de la super-classe. Les variables et méthodes de la super-classe sont implicitement présentes dans la sous-classe. Pendant l'héritage, il est possible d'ajouter des nouvelles définitions de variables ou de méthodes dans la sous-classe, ou au contraire, de fournir du code différent pour une méthode en provenance de la super-classe. Tous les attributs repris dans la sous-classe sans modification sont dit *hérités* de la super-classe; toutes les méthodes changées dans la sous-classe sont dites *re-définies*.

L'héritage entre classes se traduit pas le partage d'attributs. La sous-classe partage avec sa super-classe *le code*, pour les méthodes héritées, et les *valeurs initiales*, pour les variables.

3.1 Héritage simple

La classe `compte` contient une variable avec le solde d'un compte, et des méthodes pour consulter le solde, pour virer et pour retirer de l'argent.

```
class compte s_init =
 object
   val mutable solde = s_init
   method depot n = solde <- solde + n
   method retrait n = solde <- solde - n
   method donne_solde = solde
   method affiche_solde =
       begin print_string"Votre solde est: ";
             print_int solde; print_newline ()
       end
  end;;
class compte :
  int ->
  object
    val mutable solde : int
    method affiche_solde : unit -> unit
    method depot : int -> unit
    method donne_solde : int
    method retrait : int -> unit
  end
```

La syntaxe de l'héritage est: `inherit` nom_1 $a_1 \ldots a_n$ `[as` nom_2 `]`, où nom_1 est le nom de la classe dont on veut hériter et $a_1 \ldots a_n$ sont ses arguments d'initialisation. L'annotation optionnelle `as` nom_2 est détaillée dans les parties suivantes. La classe `compteremunere` est une sous-classe de `compte`. Elle hérite de tous les attributs de `compte`, et les étend par la définition de deux nouvelles variables `taux` et `interets`, et d'une nouvelle méthode `ajoute_interets`.

```
class compteremunere s_init =
  object
    inherit compte s_init
    val taux = 5   (* Donne en % *)
    val mutable interets = 0
    method ajout_interets =
       begin interets <- (solde*taux)/100;
             solde <- solde + interets
       end
  end;;
```

`compteremunere` possède toutes les variables et méthodes de sa classe parente. Ceci se reflète dans l'interface inférée par le typeur:

```
class compteremunere :
  int ->
  object
    val mutable interets : int
    val mutable solde : int
    val taux : int
    method affiche_solde : unit
    method ajout_interets : unit
    method depot : int -> unit
    method donne_solde : int
    method retrait : int -> unit
  end
```

On peut utiliser toutes les méthodes, qu'elles soient nouvelles ou héritées:

```
# let dupont = new compteremunere 200;;
val dupont : compteremunere = <obj>

# dupont#depot(1000);;
- : unit = ()

# dupont#ajout_interets;;
- : unit = ()

# dupont#affiche_solde;;
Votre solde est: 1260
- : unit = ()
```

3.2 Re-définition

Le code pre-existant n'est pas toujours adaptée à tous les contextes de re-utilisation. Il arrive alors que l'on mélange héritage et re-définition des méthodes pour décrire, en particulier, du code plus spécialisé dans une sous-classe. Une nouvelle version de la sous-classe `compteremunere` re-définit la méthode d'affichage pour signaler la part des interêts dans le solde du compte.

```
class compteremunere s_init =
  object
    inherit compte s_init
    val taux = 5   (* Donne en % *)
    val mutable interets = 0
    method ajout_interets =
       begin interets <- (solde*taux)/100;
             solde <- solde + interets
       end
    method affiche_solde =
       begin print_string"Votre solde est: ";
             print_int solde; print_newline ();
             print_string" dont la part d'interets est: ";
             print_int interets
       end
  end;;
```

L'exemple suivant montre le nouveau comportément de l'affichage:

```
# let compterem1 = new compteremunere 200;;
val compterem1 : compteremunere = <obj>
# compterem1#ajout_interets;;
- : unit = ()
# compterem1#affiche_solde;;
Votre solde est: 210
 dont la part d'interets est: 10
- : unit = ()
```

En Ocaml, il existe une contrainte (que nous justifierons plus tard) à la re-définition d'une méthode *pendant l'héritage*: le type de la méthode dérivée doit être égal ou plus spécialisé (du point de vue du polymorphisme[3]) que celui de la méthode parente. Intuitivement, cette contrainte n'est pas surprenante: la spécialisation (éventuelle) du type et du comportément d'une méthode est cohérente avec la notion même de spécialisation d'une classe par héritage. Dans notre exemple, le type de `affiche_solde` reste inchangé, et de ce fait, la re-définition de la méthode dans la nouvelle classe est validée par le typeur:

[3] Ici, il s'agit de *polymorphisme paramétrique*. Nous préciserons cette notion dans la partie du cours dédiée au typage.

```
class compteremunere :
  int ->
  object
    val mutable interets : int
    val mutable solde : int
    val taux : int
    method affiche_solde : unit
    method ajout_interets : unit
    method donne_solde : int
    method depot : int -> unit
    method retrait : int -> unit
  end
```

3.3 Auto-référencement: *self*

L'auto-référencement est la possibilité pour une méthode d'utiliser les autres méthodes (ou méthodes *soeurs*) qui se trouvent dans un objet. Considérons une autre version de la classe `cell` où l'on incorpore la nouvelle méthode `double`. Dans cette dernière, on utilise la méthode `set` de la même classe[4].

```
class cell =
 object(self)
     val mutable cont = 0
     method get  = cont
     method set n = cont <- n
     method double = self#set (cont*2)
     method print = print_int self#get
 end;;
```

Nous ne pouvons pas employer `set` qu'au moyen d'un envoi de message. Or, au moment de définir la classe, il n'y a pas encore d'objet qui en soit une instance! En revanche, une invocation future de `double` se fait par `o#double`, via un certain objet `o`, dit *objet courant*. Quoi de plus normal que d'aller chercher, *pendant l'exécution de* `double`, le code de sa méthode soeur dans cet objet? On utilise souvent le mot *self* pour parler de *l'objet courant*. En Ocaml, lors de la définition d'un classe, on peut donner un nom quelconque à cet objet, au moyen de la construction `object`(*ident*). Nous preférons utiliser la terminologie objet et l'appelons toujours `self`. Dans notre exemple, la méthode d'affichage utilise également une méthode soeur (`get`) via `self`.

```
# let c = new cell;;
val c : cell = <obj>

# c#set 1;;
- : unit = ()

# c#print;;
1- : unit = ()

# c#double;;
- : unit = ()

# c#get;;
- : int = 2

# c#print;;
2- : unit = ()
```

[4]Plus précisement, on peut utiliser toute méthode soeur `s` qui se trouve dans la hiérarchie de cette classe: soit dans une super-classe, si `s` est hérité, soit dans une sous-classe, si `s` est redéfinie plus tard dans la classe éffective de l'objet auquel on envoit le message. Ce point est exploré en détail dans la partie dédié à la liaison tardive.

De manière générale, il est pratique d'avoir accès aux méthodes *soeurs* d'une classe, mais là où réside la veritable puissance de l'auto-referencement c'est dans son caractère dynamique: l'objet referencé par `self` n'est pas d'une classe figée. Dans notre exemple, il peut être de classe `cell` ou d'une de ses sous-classe. Ceci implique que le comportément des méthodes invoquées avec `self` peut varier dynamiquement (selon la classe effective de `self`). Il s'agit là d'un problème de liaison des variables (ici les méthodes), que nous étudierons un peu plus loin dans la partie dediée à la *liaison tardive*.

3.4 Référencement aux classes parentes: *super*

C'est la possibilité pour une classe d'invoquer une méthode de la classe parente. Cela est utile dans un contexte de redéfinition de la méthode invoquée (autrement, `self` est suffisant). En Ocaml, pour employer une méthode d'une super-classe, on doit nécessairement nommer cette dernière. Le but est d'identifier la classe d'origine de la méthode demandée, ce qui est justifié par l'ambigüité que peut entrainer l'héritage multiple. On donne un nom à la super-classe par la construction `inherit C a1 ... an as` *nom_superclasse*. Par la suite, toute utilisation de la méthode de la super-classe se fait par envoi de message via ce nom-là. La classe `backupCell` définit des cellules dont on garde une copie après modification. Elle re-définit la méthode `set` héritée de `cell`, et utilise via le nom `super` le code de cette méthode avant re-définition.

```
class backupCell =
 object
    inherit cell as super
    val mutable backup = 0
    method  set n = backup <- cont; super#set n;
    method print = print_string "Actual value: ";
                   super#print; print_newline();
                   print_string "Backup value: ";
                   print_int backup; print_newline()
    method restore = cont <- backup
end;;

let bc = new backupCell;;
val bc : backupCell = <obj>

 bc#set 1;;
- : unit = ()

 bc#print;;
Actual value: 1
Backup value: 0
- : unit = ()

 bc#restore;;
- : unit = ()

 bc#print;;
Actual value: 0
Backup value: 0
- : unit = ()
```

3.5 Héritage multiple

Ocaml permet l'héritage multiple, à savoir, l'héritage des méthodes et valeurs de plusieurs classes pas forcément reliées entre elles. En cas de noms identiques de variables ou de méthodes, seulement la dernière déclaration dans l'ordre de l'héritage est conservée, les autres étant masquées. Les méthodes restent cpendant accessibles à travers le referencement aux classes parentes via des noms différents donnés à chacune des super-classes. En revanche, les

variables ne sont plus accesibles directement. Seules les méthodes permettent d'y accèder. Nous définissons les classe des points `point`, et la classe des couleurs `color`. La classe `colorpoint` est définie par héritage multiple à partir des classes `point` et `color`.

```
 class point init =
  object
   val mutable x = init
   method getx = x
   method move d = x <- x + d
   method print = print_int x
  end;;

class color (c_init:string) =
 object
  val mutable c = c_init
  method getcolor = c
  method setcolor c' = c <- c'
  method print = print_string c
 end;;
```

Ces deux classes ont une méthode `print` spécifique:

```
let p = new point 7;;
val p : point = <obj>

p#print;;
7- : unit = ()

let c = new color "rouge";;
val c : color = <obj>

c#print;;
rouge- : unit = ()
```

Pour acceder aux méthodes `print` des classes parentes sans créer d'ambiguité, il est indispensable de distinguer la classe d'où vient chacune d'entre elles, d'où le besoin de nommage:

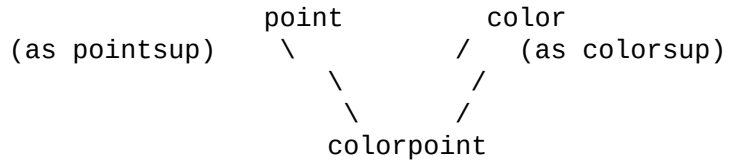
```
class colorpoint x (c:string) =
 object
  inherit point x as pointsup
  inherit color c as colorsup
  method print =
     print_string("(");
     pointsup#print;
     print_string(" ,");
     colorsup#print;
     print_string(")")
  end;;

let cp = new colorpoint 7 "bleu";;
val cp : colorpoint = <obj>

 cp#print;;
(7 ,bleu)- : unit = ()
```

3.6 Hierarchie des classes

Dans l'exemple précédent, la hierarchie donnée par l'héritage multiple s'exprime par un *treillis* et non plus par un arbre comme c'était le cas dans l'héritage simple.



4 Re-utilisation et polymorphisme

Un morceau de programme est re-utilisable si on peut l'employer facilement dans plusieurs contextes. Par exemple, un module est re-utilisé par tous les programmes qui l'importent. Dans les langages fortément typés, la vérification de contraintes de type présentes dans le contexte d'utilisation assurent la cohérence finale du programme. Par exemple, un programme qui importe un module, pose comment contrainte que celui-ci ait une certaine interface. L'utilisation du module n'est possible qu'après vérification de cette contrainte.

Dans la plupart des langages procéduraux classiques, la re-utilisation impose une concordance exacte entre le type éffectif du composant re-utilisé et le type attendu dans le contexte où il doit être employé. Dans les langages objet cette concordance peut être *approximative*. Rappelons, dans l'exemple suivant, que la classe `compteremunere` est une sous-classe de `compte`.

```
let c = new compte 100;;

let cr =  new compteremunere 200;;

let est_crediteur (x:#compte) = x#donne_solde >= 0;;
val est_crediteur : #compte -> bool = <fun>
```

La notation `(x:#compte)` est lue

`x:`*Type_Instance_de(*`compte`*)*

Considérons le code suivant:

```
est_crediteur(c);;
est_crediteur(cr);;
```

La fonction `est_crediteur` attend un objet instance de `compte` (par exemple, `c`). On peut donc voir l'appel `est_crediteur(cr)` comment une tentative de re-utiliser la fonction sûr un objet `cr` dont le type éffectif n'est pas instance de `compte`. Dans un langage classique, cet appel provoque une erreur de typage car les types *Type_Instance_de(*`compte`*)* et *Type_Instance_de(*`compteremunere`*)* sont considérés incompatibles. Dans les langages objet, cet appel est valide grâce à la règle suivante, dite de *polymorphime de l'approche objet*[5]. Nous considérons deux classes quelconques c et c_s.

Polymorphisme dans l'approche objet:

Soit c_s une sous-classe de c,

- o_s est une instance de $c_s \Rightarrow o_s$ est une instance de c (vue des objets)
- o_s : `Type_Instance_de(`c_s`)` $\Rightarrow o_s$: `Type_Instance_de(`c`)` (vue des types).

La première version exprime la règle en termes des objets vus comme des entités générés à partir de leurs classes. La deuxième, exprime la règle du point de vue des types des objets. Informellement, un objet d'une certaine classe

[5] Plus précisement, cette règle décrit informellement le *polymorphisme d'inclusion*. En Ocaml, le polymorphisme mis en oeuvre pour les objets *n'est pas* celui d'inclusion, mais une extension du *polymorphisme paramétrique* qui permet d'obtenir un comportément équivalent pour la plupart des programmes, tout en apportant des avantages importants, qui seront détaillés plus tard.

peut être employé là où une autre classe d'objet est attendue, si le premier possède au moins tous les attributs du deuxième. C'est le cas de tout objet appartenant à une sous-classe de la classe attendue. Les attributs additionnels dans le premier, restent alors *invisibles* par le code initialement prevu pour la classe moins étendue: ils sont préservés mais ne sont pas accessibles. Ainsi, selon cette régle, le type d'un objet est polymorphe: il s'agit du type donné par sa classe, mais aussi par tous les types de ses super-classes.

```
est_crediteur c;;
- : bool = true

est_crediteur cr;;
- : bool = true
```

En résumé, l'héritage, et avec la lui, la notion de sous-classe entraîne une utilisation polymorphe des constructions objet. Le but recherché est celui d'augmenter les chances de re-utiliser du code, tout en donnant la possibilité d'enrichir incrémentalement celui-ci.

Mais dans les langages objets, toute re-utilisation n'est pas l'unique fruit du polymorphisme, ou ce qui est équivalent, de l'héritage. La combinaison de deux autres mécanimes favorise également la re-utilisation. Ce sont la re-définition et la liaison tardive. Ainsi, au chapitre des fonctionnalités fondamentales des langages objet, et qui font sans doute leur succès, on compte:

- un objet d'une certaine classe peut employer le code prevu pour des objets d'une super-classe (*polymorphisme*);
- une méthode peut être partageée par les objets issus de ses sous-classes (*héritage*)
- le code d'une méthode peut être spécialisé, ou simplement modifié, dans une sous-classe en respectant certaines contraintes (*re-définition*).
- l'utilisation de `self` diminue la quantité de code hérité qu'il est nécessaire de spécialiser dans les sous-classes (*liaison tardive*).

5 Liaison tardive

Avant d'exécuter un programme, on doit établir le lien entre chaque identificateur rencontré et sa valeur (ou son code): il s'agit du problème de la *liaison d'identificateurs*. Si le lien est établit au moment de la compilation on parle de *liaison statique* (ou précoce, *early-binding*), et lorsqu'il est retardé jusqu'à l'exécution du programme on parle de *liaison tardive* (ou dynamique, *late-binding*). Dans les dialectes ML (et dans une grande majorité de langages) la liaison des identificateurs est statique. Dans les dialectes Lisp, celle-ci est dynamique. Considérons l'exemple suivant en ML:

```
# let x = 3;;
val x : int = 3

# let f (y) = x+y;;
val f : int -> int = <fun>

f (2);;
- : int = 5

let x = 7;;
val x : int = 7

f (2)
- : int = 5
```

L'identificateur `x` dans le corps de `f` est lié *statiquement* à la valeur de `x` au moment de la définition de `f`, c.a.d., à la valeur 3. La redéfinition ultérieure de `x` ne change pas le comportément de `f`.

Dans les langages à objets, la liaison tardive concerne le lien entre une méthode et son code: un même identificateur de méthode peut être lié à un code différent au sein de chaque objet, et ainsi, c'est seulement à travers l'objet receveur qu'il est possible d'activer le code appropié.

Dans les langages objet, la liaison retardée est un moyen de plus au service de la re-utilisation. Comme nous l'avon vu, re-utilisation et polymorphisme des objets sont intimement liés. Nous retrouverons ces notions dans les deux cas étudiés par la suite. Ils correspondent aux deux schémas typiques de re-utilisation via la liaison tardive.

5.1 Polymorphisme et liaison tardive

Considérons le code suivant:

```
let trois (x : #cell) = x#set(3);;
val trois : #cell -> unit = <fun>

let bc = new backupCell;;

trois(bc);;
```

Le polymorphisme d'inclusion autorise un objet à être vu comme une instance de sa classe éffective, ou comme une instance d'un de ses super-classes. Ainsi, `bc` de classe `backupCell` peut être employé comme un objet de classe `cell` sans danger, et l'appel `trois(bc)` est correctement typé.

Mais quel est le comportément de cet appel? En d'autres termes, quel est le comportément de `x#set(3)` lorsque `x = bc`? Deux informations sont disponibles. À la compilation, un typage approximatif de `x`: il est instance de `cell`; lors de l'appel, `x = bc` où `bc` est instance de `backupCell`.

à la compilation	⇒	`x : Type_Instance(cell)`
à l'exécution	⇒	`x = bc : Type_Instance(backupCell)`

Dans `backupCell`, le code de `set` est re-défini. Par conséquent, nous avons le choix entre deux comportéments:

Liason statique	⇒	`x#set(3)` exécute le code de `set` dans `cell`
Liason tardive	⇒	`x#set(3)` exécute le code de `set` dans `backupCell`

Dans l'approche orienté objet, on dit que `backupCell` est le *vrai* type de `bc`, et que c'est le vrai type d'un objet qui détermine le choix de la méthode à employer. Ainsi, lors de la compilation de la fonction `trois`, le choix du code à exécuter pour `set`, est retardé jusqu'au moment de l'exécution, où un objet éfféctif, par exemple `bc`, est lié au paramètre `x`. C'est le code de `set` dans cet objet qui est alors invoqué.

```
# bc#set 1;;
- : unit = ()

# bc#print;;
Actual value: 1
Backup value: 0
- : unit = ()

# trois(bc);;
- : unit = ()

# bc#print;;
Actual value: 3
Backup value: 1
- : unit = ()
```

On trouve la liason tardive dans tous les langages objet. Il s'agit d'un mécanisme important de l'abstraction dans l'approche objet: un objet doit savoir se comporter de manière autonome, et donc, le contexte n'a pas besoin d'être examiné afin de décider quelle méthode appliquer.

5.2 Utilisation de `Self`

Rappelons le code des classes `cell` et `backupCell`

```
class cell =
 object(self)
     val mutable cont = 0
     method get  = cont
     method set n = cont <- n
     method double = self#set (cont*2)
     method print  = print_int self#get
 end;;

class backupCell =
 object
    inherit cell as super
    val mutable backup = 0
    method  set n = backup <- cont; super#set n;
    method print = print_string "Actual value: ";
                   super#print; print_newline();
                   print_string "Backup value: ";
                   print_int backup; print_newline()
    method restore = cont <- backup
end;;

let c = new cell;;

let bc = new backupCell;;

 bc#set(1);;
- : unit = ()

 bc#print;;
Actual value: 1
Backup value: 0
- : unit = ()

bc#double;;
```

Quel est le comportément de l'invocation `bc#double`?

```
bc#double => invocation de la methode double de cell
          => self#set(cont*2)
          => invocation de quelle methode set?
```

L'objet référencé par `self` est l'objet courant. Mais qui est cet objet là? Clairement, au moment où le code `self#set(cont*2)` est écrit, il n'y a pas d'objet encore lié à `self`. Cette liaison ne peut être établie que de manière dynamique. Dans notre cas, l'objet courant (le dernier activé par un envoi de messages) est `bc`. Nous obtenons

```
bc#double => self#set(cont*2)  avec self = bc
          => bc#set(cont*2)
          => backup <- cont; super#set(cont*2)
          => backup <- cont; cont <- cont*2
          => backup <- 1; cont <- 2
```

Ainsi, lors de la compilation de `self#set(cont*2)`, le choix du code à invoquer pour la méthode `set` est retardé jusqu'au moment de l'exécution, quand il est possible d'établir *dynamiquement* une liaison pour `self`. C'est à ce moment-là que le code pour `set` est selectionné: il s'agit de celui dans l'objet lié à `self`. Dans notre cas, il s'agit du code de `set` dans `bc`.

Dans notre exemple, il est important de comprendre que le code exécuté, *n'est pas celui* qui correspond à une liaison statique pour la méthode `set`. La méthode `double` invoquée est celle héritée de `cell`. Or, `double` invoque à son tour sa méthode soeur `set`. Dans la mesure où cette demande *apparaît textuellement* dans `cell`, on peut s'interroger: s'agit-il de `set` dans `cell` ou dans `backupCell`? Le schéma suivant montre la pertinence de la question:

```
class cell =
   ...
     method set n  = ...
     method double = self#set (cont*2)   (* quel methode set ici ? *)
   ...
end

class backupCell =
  ...
    inherit cell
    method  set n = (* methode redefinie *)
  ...
end
```

Comme dans le cas précédent, nous avons deux choix possibles

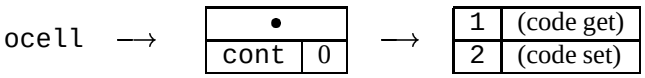
Liason statique ⇒ `self#set(cont*2)` exécute le code dans `cell`
Liason tardive ⇒ `self#set(cont*2)` exécute le code dans `backupCell`

Et comme dans le cas précédent, le choix de la méthode est retardé jusqu'à l'exécution. Cette fois, c'est grâce à l'invocation de `set` via `self`, et au mécanisme de liaison dynamique pour celui-ci, que le choix de code pour `set` devient retardé.

5.3 Liaison tardive et représentation mémoire

L'*enregistrement d'attributs* qui sert à représenter en mémoire les données d'un objet est, pour la plupart des langages, un peu plus compliqué que celui étudié dans la partie 2.1. En général un enregistrement d'attributs est composé de deux parties: une *partie variable et propre* à chaque objet contient les variables d'instance; une *partie fixe et partagée* par tous les objets instances d'une même classe, correspond à une table de méthodes.

La table de méthodes est organisée comme un tableau qui contient le code de toutes méthodes, chacune indexée par le numéro de la méthode. Le numéro associé à une méthode est calculé au moment de la compilation et ne varie pas au sein d'un même programme.



Nous supposons qu'il existe une instruction machine `SELECT(o,k)` capable d'extraire le code d'une méthode de numéro k dans la table de méthodes de `o`. Dans notre exemple, l'envoi de message `ocell#get` es compilé par l'instruction `SELECT(ocell,1)`. La liaison tardive est obtenu par l'appel à cet instruction lors de l'exécution de l'envoi de message, plutôt que lors de sa compilation.

L'envoi de messages à `self` est aussi compilé en une sélection de méthode, à partir de son numéro, dans la table des méthodes dynamiquement liée à `self` lors de l'exécution.

Pendant la compilation de l'héritage, les méthodes, re-définies ou non, conservent le même numéro que dans leurs classes parentes. Ceci garantît, en particulier, la re-utilisation polymorphe et sans danger pendant l'exécution, des programmes conçus pour les super-classes, par les objets instances des sous-classes. En effet, le code compilé pour ces programmes, selectionne un numéro de méthode qui correspond uniformément au même nom de méthode dans tous les enregistrements d'attributs.

6 Autres constructions

6.1 Initialisation

Une classe peut incorporer une méthode anonyme qui est declenchée immédiatement après construction d'un objet de cette classe. Cette méthode est définie à l'aide du mot-clé `initializer`. Elle peut réaliser n'importe quel calcul, et a accès à `self` et aux variables d'instance:

```
class point init =
 object(self)
  val mutable x = init
  method getx = x
  method print = print_int x

  initializer print_string"Nouveau point aux coordonnees: ";
              self#print; print_newline()
end;;

class point :
  int ->
 object
  val mutable x : int
  method getx : int
  method print : unit
 end

class colorpoint xinit cinit =
 object(self)
  inherit point xinit as pointsup
  inherit color cinit as colorsup
  .....
  initializer print_string"Nouveau point colore de couleur: ";
              print_string self#getcolor; print_newline()
end;;

class colorpoint :
  int ->
  string ->
  object
    val mutable color : string
    val mutable x : int
    method getcolor : string
    method setcolor : string -> unit
    method getx : int
    method print : unit
  end
```

Les méthodes d'initialisation ne peuvent pas être redéfinies. Elles s'exécutent séquentiellement, comme le montre l'exemple suivant où les initialisations pour un point et pour un point coloré se succèdent lors de la création d'un point coloré:

```
# let p = new point 3;;
```

```
Nouveau point aux coordonnees: 3
val p : point = <obj>

# let pc = new point_colore 4 "blue";;
Nouveau point aux coordonnees: (4 ,blue)
Nouveau point colore de couleur: blue
val pc : point_colore = <obj>
```

Notez, lors de la création de `pc`, que la méthode `print` employée par l'initialisation d'un `point` n'est pas celle des points, mais celle de l'objet courant (ici, `self`) = `pc`).

6.2 Méthodes abstraites

Grâce au mot-clé `virtual`, il est possible de déclarer des méthodes de manière *abstraite* ou *virtuelle*, c'est-à-dire, sans fournir leur implantation, qu'on pourra donner plus tard dans une sous-classe. La classe qui contient une méthode abstraite doit être déclarée elle aussi abstraite, et ne peut être instanciée (aucun objet de cette classe ne peut être crée). Les classes abstraite sont utiles pour abstraire sous une même classe parente (et donc, sous un même type), plusieurs classes dont l'implantation des méthodes change dans chacune d'entre-elles.

```
class virtual abstract_point xinit =
 object(self)
  val mutable x = xinit
  method virtual getx : int
  method get_offset = self#getx - xinit
  method virtual move : int -> unit
end;;

class virtual abstract_point :
  int ->
  object
    val mutable x : int
    method get_offset : int
    method virtual getx : int
    method virtual move : int -> unit
  end

# let ap = new abstract_point 2;;
One cannot create instances of the virtual class abstract_point
```

Dans la classe `point` on hérite de `abstrac_point`, et l'on définit enfin le code des deux méthodes abstraites `get` et `move`. La nouvelle classe `point` n'est plus abstraite (si elle n'avait pas défini toutes les méthodes indéfinies dans `abstrait_point`, le résultaut aurait été une classe `point` abstraite).

```
class point xinit =
 object
  inherit abstract_point xinit
  method getx = x
  method move d = x <- x + d
end;;

class point :
  int ->
  object
    val mutable x : int
    method get_offset : int
    method getx : int
    method move : int -> unit
  end
```

```
# let p = new point 2;;
val p : point = <obj>
# p#getx;;
- : int = 2
# p#get_offset;;
- : int = 0
```

6.3 Méthodes privées

Une méthode déclarée privée dans une classe est spécifiée comme privée par son interface et ne peut être utilisée qu'à partir d'autres méthodes de la classe:

```
# class point_restraint xinit =
 object(self)
  val mutable x = xinit
  method getx = x
  method private move d = x <- x + d
  method bump = self#move 1
  method print = print_int x
end;;
```

L'interface inférée pour `point_restraint` est:

```
class point_restraint :
  int ->
  object
    val mutable x : int
    method bump : unit
    method getx : int
    method private move : int -> unit
    method print : unit
  end

# let p = new point_restraint 1;;
val p : point_restraint = <obj>

# p#move;;
This expression has type point_restraint
It has no method move
```

Les méthodes privées apparaissent dans l'interface des sous-classes tout en restant privées dans celles-ci. Elles apparaissent ainsi dans la hiérarchie des classes mais ne sont pas directement utilisables.

```
class point_colore xinit c =
 object
  inherit point_restraint xinit as super
  val mutable color = c
  method getcolor = color
  method print =
     print_string("(");
     super#print;
     print_string(" ,");
     print_string color;
     print_string(")")
end;;

class point_colore :
```

```
int ->
string ->
object
  val mutable color : string
  val mutable x : int
  method bump : unit
  method getcolor : string
  method getx : int
  method private move : int -> unit
  method print : unit
end
```

7 Autres lectures

- Emmanuel Chailloux, Pascal Manoury et Bruno Pagano. “Devéloppement d’applications en Objective Caml”. Ed. O’Reilly. `http://www.editions-oreilly.fr/`
- La documentation d’Ocaml sur les objets: `http://caml.inria.fr/ocaml/htmlman/`
- Le cours d’introduction aux objets de Didier Rémy au magistère MMFAI: `http://pauillac.inria.fr/ remy/classes/magistere/`
- Le cours de Roberto Di Cosmo au magistère MMFAI.