Introduction aux Objets

(version provisoire encore en cours de rédaction)

Didier Rémy

5 mars 1997

Résumé

Ces notes sont avant tout une introduction aux notions fondamentales à la base de la programmation avec objets (auto-référence, classes, héritage, sous-typage) traitée de façon rigoureuse en se plaçant toujours dans un contexte fortement typé. C'est aussi un guide dans la découverte de la programmation avec objets.

Au travers de ces notes, vous retrouver quelques messages :

1. Généralité

   La programmation avec objets est avant tout un style de programmation.

   Au travers du langage Objetcive ML qui est présenté, nous illustrons les concepts des langages à objets. Ceux-ci peuvent ensuite être déclinés de multiples façons et founir des variantes du langage proposé.

2. Simplicité

   Les objets peuvent s'expliquer simplement dans le langage ML

3. Objets et modules :

   Ce sont des solutions différentes à un même problème.

   Ces extensions se cotoient sans conflit ; toutefois elles devraient être fusionnées.

Pré-requis

Le langage ML est supposé connu.

Table des matières

Chapitre 1

Introduction

Dans ce chapitre nous présentons le concept de la programmation à objets, et illustrons la plupart de ses notions au travers de nombreux exemples écrits sans utiliser de constructions spécifiques aux objets. Nous espérons ainsi que cette approche progressive facilite la compréhension et aide à démystifier la notion d'objets.

La dernière partie présente les points délicats de la programmation à objets. Il n'est pas nécessaire de les comprendre dans le détail dans un premier temps, car nous reviendrons sur les difficultés dans les chapitres suivants.

1.1 Pourquoi la programmation à objets ?

L'idée d'objets est apparue à une époque où les langages de programmation ne fournissaient aucune aide à la programmation à grande échelle. Celle-ci est très différente de la programmation à petite échelle. Le concept essentiel de la programmation à grande échelle est la modularité, que l'on peut décliner sous différents aspects : sûreté, flexibilité, évolutivité, réutilisabilité, etc.

Une notion qui n'est pas entièrement dérivée de la modularité, mais qui lui est très liée est celle d'encapsulation, qui peut apparaître sous des formes différentes : fermeture, types abstraits, objets, etc.

1.1.1 Modularité

La modularité est la possibilité d'organiser un programme en plusieurs entités (modules) relativement indépendantes. Cela permet une plus grande flexibilité dans la mise au point des programmes, chaque module pouvant être implémenté séparément. L'écriture d'un programme de façon modulaire augmente aussi la sécurité, car la spécification du comportement d'un module, est plus simple que celle de l'ensemble, et sa réalisation peut être testée séparément. Le comportement du système tout entier peut être compris en se limitant à une abstraction du comportement de ses parties.

La modularité favorise également l'évolutivité des programmes. Puisqu'il suffit de comprendre une sous-partie d'un programme (et sa spécification) pour la modifier. Enfin, la programmation modulaire doit aussi faciliter la réutilisabilité des programmes. En effet, en écrivant un programme par assemblage de parties plus élémentaires, on s'attend à ce que certaines de ses parties soient en fait des bibliothèques pouvant être réutilisées dans d'autres programmes.

1.1.2 Encapsulation

L'encapsulation désigne la possibilité de construire des valeurs en cachant une partie de leur implémentation. Cela garantit qu'une autre partie du programme ne pourra pas directement accéder aux parties cachées de l'implémentation ; au contraire, il sera nécessaire de communiquer en utilisant les fonctions fournies dans l'interface, ce qui assure le respect de certains invariants, et permet également de faire des vérifications dynamiques de sûreté aux points de communication.

En ML, par exemple, les fonctions peuvent encapsuler des données dans leur fermeture. L'encapsulation peut aussi être réalisée par abstraction de type.

1.1.3 Un test de modularité

Considérons un programme typique en ML : nous allons construire des données structurées, puis les manipuler. Bien souvent, les données sont représentées par des arbres. Nous pouvons définir un type concret (récursif) :

```
type arbre = A of arbre * arbre | B of bool | C of int;;
```

Puis nous ajoutons des opérations d'impression, d'évaluation, etc...

```
let (imprime : arbre → unit) = ...
let (évalue : arbre → int) = ...
```

Plus tard, nous pouvons être amenés à modifier notre programme de deux façons différentes :

- soit en enrichissant la structure de données : par exemple, en ajoutant un nouveau type de nœuds aux arbres manipulés ; on parlera d'*extension horizontale*.
- soit en ajoutant des opérations sur la structure existante : par exemple, en ajoutant une fonction de simplification :

  ```
  let (simplifie : arbre → arbre) = ...
  ```

  On parlera d'*extension verticale*.

L'extension verticale est une opération modulaire en ML, alors que l'extension horizontale ne l'est pas. En effet, pour ajouter une nouvelle fonction, il suffit d'écrire celle-ci à la suite de toutes les autres. Il n'est pas nécessaire de modifier les anciennes définitions. En particulier, on peut écrire les anciennes définitions dans un fichier, et la nouvelle définition dans un nouveau fichier.

Par contre, cela n'est pas possible pour l'extension horizontale. D'une part, il n'est pas prévu de pouvoir ajouter un constructeur à un type concret déjà défini[1]. Pire, après avoir déclaré un nouveau constructeur dans le type arbre, il faut corriger les définitions des fonctions `imprime` et `évalue` pour y préciser le calcul à effectuer sur les arbres construits avec le nouveau nœud.

À l'inverse, les objets offrent une bonne modularité horizontale, mais ils rendent plus difficile la modularité verticale. Nous reviendrons sur cette bivalence de la modularité dans la conclusion de ce cours.

1.2 La programmation à objets : un style de programmation

Tout comme la programmation impérative, ou la programmation fonctionnelle, la programmation à objets est avant tout un *style* de programmation. Bien sûr, nous retrouvons dans tous les exemples écrits dans un style objets des constructions similaires, mais aussi des problèmes similaires et des façons analogues de les résoudre (même s'il existe parfois des variations).

Le style à objets n'est donc pas réservé à des programmes écrits dans un langage à objets. Un langage à objets est simplement un langage offrant des constructions de base qui facilitent la programmation dans un style à objets.

Dans le restant de ce chapitre, nous allons découvrir la notion d'objets au travers d'exemples écrits dans le langage ML. Nous montrerons aussi les limites de cette approche, et dans le prochain chapitre nous présenterons un vrai langage à objets.

1.3 Qu'est qu'un objet ?

Un objet est une valeur regroupant un ensemble de données, et à laquelle sont associés un certain nombre d'informations. Un point sur la droite, par exemple, peut être représenté par un type concret et les définitions suivantes :

```
type point = Point of int ref;;
let p = Point (ref 1);;
let getx  (Point x) = !x;;
let setx  (Point x) = fun y → x := y;;
let print (Point x as self) = print_int (getx self);;
```

Dans la terminologie des objets, `ref 1` dans `p` est l'état interne de l'objet, `getx`, `setx` et `print` sont les méthodes pouvant s'appliquer à l'objet `p`. En général, beaucoup d'objets

[1]Nous verrons ci-dessous que cela n'est ni tout à fait vrai, ni vraiment fondamental.

partagent le même comportement, ici tous les objets ayant le constructeur `Point`. On peut évidemment définir un fonction

```
let new_point v = Point (ref v);;
```

pour créer de nouveaux points sans avoir à se soucier de l'implémentation des points. On pourra dire que ces objets sont de la même *classe* (nous préciserons cette notions ci-dessous).

Dans la programmation avec objets, le calcul est effectué par envoi de messages aux objets. Ici, c'est simplement une application de la méthode de même nom que le message à l'objet adressé :

```
setx p (getx p + 1); print_int (getx p);;
```

La programmation par envoi de messages signifie aussi que l'on puisse adresser le même message à des objets de différentes classes, le comportement de l'objet à la réception d'un message étant déterminé en fonction de sa classe.

Par exemple, on peut aussi définir des points avec une information de couleur. En suivant le modèle ci-dessus, nous obtenons :

```
type point_coloré = Point_coloré of int ref * string;;
let new_point_coloré v c = Point_coloré (ref v, c);;
let getx  (Point_coloré (x,c)) = !x;;
let setx  (Point_coloré (x,c)) y = x := y;;
let print (Point_coloré (x,c) as self) =
               print_int (getx self); print_string c;;
```

On peut alors écrire :

```
let p = new_point_coloré 1 "gris" in setx p (getx p); print p;;
```

Par contre, il n'est plus possible d'écrire

```
let p = new_point 1 in setx p (getx p);;
```

En effet la méthode `getx` qui a été redéfinie requiert maintenant des points colorés. On aurait pu choisir un nouveau nom distinct du nom de l'ancienne méthode ; mais ceci limite fortement la programmation par envoi de message.

Une solution à ce problème serait que `Point` et `Point_coloré` soient les constructeurs d'un même type. Le type des objets peut être un type extensible, étendu à la définition de chaque nouvelle classe. Le type `exn` des exceptions est le seul type extensible en ML, mais il serait possible de fournir à l'utilisateur d'autres types extensibles. Reprenons les deux exemples ci-dessus :

```
exception Point of int ref;;
let new_point v = Point (ref v);;
let getx  (Point x) = !x;;
let setx  (Point x) y = x := y;;
let print (Point x as self) = print_int (getx self);;
```

```
exception Point_coloré of int ref * string;;
let new_point v c = Point_coloré (ref v, c);;
let getx =  function (Point_coloré (x,c)) → !x | p → getx p;;
let setx =  function (Point_coloré (x,c)) → fun y → x := y | p → setx p;;
let print = function (Point_coloré (x,c) as self) →
                  print_int (getx self); print_string c
                | p → print p;;
```

Cet exemple présente clairement quelques aspects du style objet. Toutefois, cet exemple n'est pas satisfaisant car le typage ne garantit pas que l'envoi d'un message réussit toujours. En effet, si la méthode n'est pas définie pour l'objet auquel le message est envoyé, une exception de filtrage sera levée.

Il existe une autre approche, duale, que l'on peut également ébaucher dans le noyau ML. L'idée est d'attacher les méthodes à l'objet lui-même, ce qui évite le problème de leur surcharge :

```
let new_point x =
  let r = ref x in
  let getx = function () → !r in
  let setx = function y → r := y in
  (getx, setx);;
```

Les données sont appelées les *variables d'instance* (l'état interne étant l'ensemble de toutes les variables d'instances). Les méthodes sont les opérations pouvant s'appliquer à ces données. Dans cet exemple, seules les méthodes ont été retournées. L'état interne a été encapsulé à l'intérieur de l'objet.

En pratique, il est commode de nommer les méthodes. Un objet devient un enregistrement de méthodes. Nous pouvons réécrire l'exemple précédent :

```
type point = {getx : unit → int; setx : int → unit};;
let new_point v =
  let x = ref v in
  let getx () = !x in
  let setx y = x := y in
  {getx = getx; setx = setx};;
```

Pour créer un point, il suffit d'appliquer la fonction `new_point` à une valeur initiale.

```
let p = new_point 1 in
  p.setx (p.getx() +1); print_int (p.getx());;
```

Peut-on encore comme dans l'exemple précédent étendre les points avec une composante de couleur ? Oui, mais comme dans le cas précédent, l'envoi de message polymorphe (le même message peut être envoyé à des objets de différentes classes) ne sera pas possible.

Ci-dessus nous avions résolu le problème en utilisant des sommes extensibles. De façon similaire, on peut remplacer les enregistrements par des enregistrements extensibles, et retrouver l'envoi de messages polymorphes. Cette approche sera reprise plus loin dans ce chapitre de façon plus systématique dans un langage ML avec des enregistrements extensibles.

1.3.1 Les deux approches

Les deux exemples (très simples) ci-dessus mettent en avant deux approches des objets. Dans la première les objets transportent leurs variables d'instance et un constructeur permettant d'identifier leur classe. Les méthodes sont définies à l'extérieur des objets, et pour chaque classe auxquelles elles doivent s'appliquer. La sélection du code à utiliser se fait dynamiquement à partir du constructeur de l'objet dans notre exemple. Afin de garantir un typage statique de l'envoi de messages, la sélection sur le constructeur peut être remplacée par une sélection par filtrage sur le type de l'objet à l'exécution. Ce modèle est appelé *le modèle à surcharge*.

Dans la seconde approche les objets portent directement les méthodes qui peuvent leur être appliquées. Elles sont rangées dans un enregistrement, de telle façon que l'envoi de message consiste à appliquer la méthode de l'objet enregistrée sous ce nom. Ce modèle est appelé le modèle à enregistrements.

Dans le modèle à enregistrements, l'état interne de l'objet peut facilement être caché (dans la fermeture des méthodes, ou bien par un mécanisme de typage). Ce modèle permet donc l'encapsulation. Ce n'est pas le cas du modèle à surcharge ; la sélection étant faite sur son type, un objet doit révéler toute information qui détermine son comportement.

Bien que le modèle à surcharge semble plus naturel, il est en fait plus difficile à mettre en œuvre dans un contexte fortement typé. Nous développons plus en détail le modèle à enregistrements dans le reste de ce chapitre afin d'introduire les autres concepts de la programmation avec objets. Nous reviendrons sur le modèle à surcharge dans le chapitre ??

1.4 Objets et Classes avec des enregistrements extensibles

Dans cette section nous reprenons le modèle à enregistrements. Pour aller plus loin dans la découverte des objets, nous supposons que le langage est équipé d'enregistrements extensibles.

Les enregistrements extensibles sont une extension des enregistrements de ML, ayant les particularités suivantes :

- Il n'est pas besoin de les déclarer ; tout identificateur peut être utilisé comme une étiquette. Par exemple,

  ```
  let ab = {a = 1; b = 2};;
  ```

- Pour toute étiquette fixée, il est possible d'écrire une fonction polymorphe qui retourne le contenu du champ associé et que l'on puisse appliquer à tout enregistrement possédant ce champ. Par exemple :

```
fun x → x.a;;
```

- Pour tout étiquette fixé, il est possible d'écrire une fonction polymorphe qui, étant donné un enregistrement et une valeur quelconque, retourne un enregistrement identique au premier excepté sur ce champ où il est défini avec la valeur donnée en argument. Par exemple :

```
fun r x → r @ {a = x}
```

Le but n'est pas ici de formaliser les enregistrements extensibles, mais de les utiliser pour comprendre les mécanismes des objets. La section suivante permet de se faire une bonne idée de leur fonctionnement, mais sa lecture n'est pas indispensable pour comprendre la suite de ce chapitre.

L'annexe 1.7 montre comment simuler les enregistrements extensibles en ML (dans le cas où toutes les étiquettes utilisés dans tout le programme sont connues à l'avance).

1.4.1 Objets et Classes

Reprenons rapidement l'exemple des points. Nous représentons également l'état interne par un enregistrement (dont les champs sont les variables d'instance). Un objet, par exemple un `point`, est définit par son état interne `point_R` et son enregistrement de méthodes `point_M`. Ainsi :

```
let point_R x = {x = x};;

let point_M R =
  let getx () = R.x in
  let print () = print_int (getx R) in
  {getx = getx; print = print};;

let new_point x =
  let R = point_R x in point_M R;;
```

Comme précédemment, ce codage assure l'encapsulation. Cependant, chaque objet sera créer avec un nouvel enregistrement de méthodes, puisque les méthodes contiennent l'état interne dans leur fermeture. Il est possible aussi d'écrire :

```
let point_M =
  let getx R () = R.x in
  let print R () = print_int (getx R) in
  {getx = getx; print = print};;

let new_point x = let R = point_R x in (R, point_M R);;
```

Mais l'encapsulation est perdue, puisque l'état interne est laissé visible. En fait, l'encapsulation peut aussi être effectuée par un mécanisme de typage. Aussi, nous utilisons ce second codage, plus souple. Pour plus de clarté, nous définissons :

```
type 'a objet = Objet of 'a;;
let new_point x = let R = point_R x in Objet (R, point_M R);;
```

Nous pouvons également exiger que l'utilisateur définisse les messages :

```
type 'a message = Message of 'a;;
let getx = Message (fun x → x.getx);;
let print = Message (fun x → x.print);;

let getc = Message (fun x → x.getx);;
let move = Message (fun x → x.print);;
```

L'envoi du message `m` à un objet `p` n'est plus `p.m` est noté `p⇒m`, et définit par :

```
let (⇒) (Objet (R,M)) (Message m) = m M R;;
```

les points colorés peuvent s'écrire de façon analogue.

```
let point_coloré_R (x, c) = {x = x; c = c};;

let point_coloré_M =
  let getx R () = R.x in
  let getc R () = R.c in
  let print R () = print_int (getx R); print_string (getc R) in
  {getx = getx; getc = getc; print = print};;

let new_point_coloré x c =
  let R = point_coloré_R (x, c) in point_coloré_M R;;
```

Nous pouvons envoyer le message `print` à la fois aux points et aux points colorés :

```
let p = new_point 1 in p ⇒ print;;
let q = new_point_coloré 1 "blanc" in p ⇒ print;;
```

Nous avons déjà le minimum pour pouvoir parler d'objets ; l'envoi polymorphe de messages.

1.4.2 Héritage

L'*héritage* est la possibilité de réutiliser les définitions d'anciens objets pour en construire de nouveaux avec un comportement différent, par modification de méthodes existantes ou ajout de nouvelles. Par exemple, les points colorés peuvent être définis à partir des points :

```
let point_coloré_R (x, c) = point_R x @ {c = c};;
let point_coloré_M =
   let getc R () = R.c in
```

```
    let print R () = print_int (getx R); print_string (getc R) in
    point_coloré_M @ {getc = getc; print = print};;
```

Nous obtenons évidemment le même objet. Nous dirons que les points colorés *héritent* des points. Remarquons que les variables d'instances sont hérités elles aussi. Nous pouvons aller plus loin dans la définition de `print` et la remplacer par :

```
    let print R () = point_M.print (); print_string (getc R) in
```

Cette définition est préférable car une modification ultérieur du comportement des points entraînera automatiquement la même modification sur les points colorés.

1.4.3 Classes

Dans la pratique, les objets ne sont pas créés directement, mais à partir de classes, ou bien par duplication d'autres objets existants. Les classes sont donc avant tout des générateurs d'objets. Elle permettent également de créer d'autres classes, par héritage. En fait, ce sont simplement une façon de grouper ensemble toutes les définitions des variables d'instances et les définitions de méthodes :

```
let point_C = (point_M, point_R);;
```

On peut écrire une fonction `new` qui prend une classe en argument et des paramètres d'initialisation et retourne un nouvel objet ce cette classe.

```
let new (R, M) x = let state = R x in (r, M);;
let new_point = new point_C;;
```

On notera que tous les objets d'une même classe partagent effectivement le même enregistrement de méthodes.

De façon systématique, on pourra écrire :

```
type 'a class = Class of 'a;;
let point_C =
  let point_R = ... in
  let point_M = ... in
  Class (point_R, point_M);;
```

et avec héritage :

```
let point_coloré_C =
  let (Class (superR, superM)) = pointC in
  let point_coloré_R (x,c) = super_R x @ {c = c} in
  let point_coloré_M =
    let getc R () = R.c in
    let print R () = super_M.print (); print_string (getc R) in
    super_M @ {getc = getc; print = print} in
  Class (point_coloré_R, point_coloré_M);;
```

Nous disons que la classe `point_coloré` est une sous-classe de la classe `point`.

Dans les exemples ci-dessus, nous avons volontairement ignoré les points difficiles pour faire ressortir les notions les plus importantes. Toutefois le codage est relativement robuste, et quelques variations permettront de l'adapter pour rendre compte des notions plus délicates.

Exercice 1 *Testez (et corrigez) les exemples de cette section en utilisant la simulation des enregistrements extensibles de la section précédente.*

Ne soyez pas surpris par la longueur des expressions de types. On pourra séparer les étiquettes utilisées pour les variables d'instances de celles utilisées pour les méthodes, en utilisant deux types différents d'enregistrements. □

1.5 Les notions délicates

La programmation à objets serait extrêmement facile (et ce cours sans objet) s'il n'y avait pas la notion d'envoi de messages à soi-même. Cela augmente considérablement l'expressivité des langages à objets ; simultanément les programmes avec objets deviennent plus difficiles à écrire et à comprendre.

Pour pouvoir typer tous les programmes de cette partie, il faut autoriser les types récursifs implicites.

1.5.1 Self et liaison tardive

Dans la classe points, nous avons définies les méthodes par :

```
let point_M =
  let getx R  () = R.x in
  let print R () = print_int (getx R) in
  {getx = getx; print = print}
```

Par expansion de `getx`, cela est équivalent à :

```
let point_M =
  let getx R  () = R.x in
  let print R () = print_int R.x in
  {getx = getx; print = print}
```

Imaginons maintenant que l'on souhaite définir une sous-classe des points pour décaler l'origine. Nous voudrions que la redéfinition de la méthode `getx` dans les points suffise pour réaliser ce changement.

```
let point_décalé_C =
  let (Class (super_R, super_M)) = point_C in
  let point_décalé_R = super_R
  let point_décalé_M =
    let getx R () = R.x + 1 in
```

```
    super_M @ {getx = getx}
  Class (super_R, point_décalé_M)
```

Mais cela ne produira pas l'effet attendu, car la méthode `print` continuera à utiliser l'ancienne méthode `getx`. Une solution consiste à redéfinir la méthode `print` également. Cependant, l'implémentation de la méthode `print` peut ne pas être accessible au moment de la définition de la classe `point_décalé_C`.

La solution consiste pour la méthode `print` à envoyer le message `getx` à l'objet appelant, habituellement lié à la variable `self`. Il faut donc abstraire les méthodes par rapport à la variable `self` qui à l'évaluation sera lié à l'objet appelant au moment de l'envoi de message. On modifie ainsi la définition de `point_M` dans la définition de la classe des points :

```
let point_C =
  let point_R x = {x = x} in
  let point_M =
    let getx  R self () = R.x in
    let print R self () = print_int (self ⇒ getx) in
    {getx = getx; print = print} in
  Class (point_R, point_M);;
```

Nous devons simultanément redéfinir l'envoi de message :

```
let (⇒) (Objet (état, méthodes) as self) (Message m) =
  m méthodes état self;;
```

Cependant l'envoi d'un message nécessite des types récursifs. En effet, une partie de l'objet la méthode sélectionnée est appliquée à l'objet tout entier. En ML, une récursion de type doit se faire au travers de types déclarés (c'est le cas du type des listes par exemple). Cependant si l'on effectue un déclaration de type pour cacher la récursion ici, nous rendrons l'envoi de message monomorphe. La solution consiste à autoriser les types récursifs, ce qui ne pose pas de problème en soi.

La nouvelle définition de la classe des points décalés :

```
let point_décalé_C =
  let (Class (super_R, super_M)) = point_C in
  let point_décalé_R = super_R
  let point_M =
    let getx R self () = R.x + 1 in
    super_M @ {getx = getx} in
  Class (super_R, point_décalé_M)
```

produit enfin l'effet attendu.

L'envoi de message à `self` est d'un usage systématique. Il produit un effet de *liaison tardive*. En effet dans la définition de la méthode `print`, la méthode `getx` n'est plus égale à la valeur de celle définie conjointement, mais à la valeur de cette méthode dans l'objet lié à `self` à l'exécution. On parle aussi parfois, mais à tort[2], de liaison dynamique.

La liaison tardive pose des difficultés liées à son interaction avec l'héritage : le typage d'une classe parente doit faire des hypothèses sur le type de `self`. Mais dans une sous-classe `self` désigne des objets ayant des types différents dans la classe parente et dans la sous-classe.

1.5.2 Méthodes binaires

Une autre difficulté du style à objets sont les méthodes binaires. Typiquement elles apparaissent, dés que l'on veut comparer des objets entre eux. Par exemple, pour ajouter une méthode d'égalité aux points, on définira une nouvelle classe :

```
let point_comparable_C
  let (Class (super_R, super_M)) = point_C in
  let point_comparable_M =
    let equal R (self:'a) (y:'a) = (self⇒getx) = (y⇒getx) && R.c (y⇒getc)) in
    super_M || {equal = equal} in
  Class (super_R, point_comparable_C);;
```

Les méthodes binaires posent parfois des difficultés de typage, en particulier en présence de sous-typage. Nous reviendrons sur ce point délicat dans les deux chapitres suivants.

Une *méthode binaire* est une méthode qui reçoit parmi un ou plusieurs arguments contenant un objet du même type que l'objet lié à la variable `self`.

1.5.3 Sélection multiple

Il ne faut pas confondre les méthodes binaires avec la sélection multiple, bien que les deux ne soient pas sans rapport. La *sélection multiple* (*multiple dispatch* en anglais) signifie que le choix de la méthode à appliquer à la réception du message dépend de plusieurs arguments à la fois.

La sélection multiple est une opération relativement naturelle dans le modèle à surcharge, mais plus difficile à réaliser dans le modèle à enregistrements.

1.6 Typage et sous-typage : une nécessité, des difficultés

Pour conclure ce chapitre, nous voulons insister sur le typage. Le concept d'objet est une notion relativement simple, mais elle se trouve compliquée par la combinaison de la liaison

[2]La liaison dynamique a un autre sens, précis, signifiant que la valeur d'un identificateur à l'exécution est la dernière valeur liée à cet identificateur au moment de l'exécution ; c'est par opposition à la liaison statique qui signifie que la valeur d'un identificateur à l'exécution est celle liée dans le contexte d'utilisation de l'identificateur

tardive et de l'héritage, qui restent deux aspects essentiels de la programmation à objets, et qu'il n'est pas question d'interdire.

De façon générale, le typage apporte énormément à la sécurité, et la rapidité de la programmation. Cet aspect est encore plus essentiel pour les langages à objets, étant donné la difficulté des constructions. Plusieurs langages à objets typés, utilisés dans l'industrie, ont été erronés pendant plusieurs années...

Un concept que nous n'avons pas encore évoqué est la possibilité pour un objet d'une sous-classe d'être utilisé comme s'il appartenait à une classe parente. En effet, il semble raisonnable de pouvoir utiliser un point coloré là où un point simple est attendu, puisqu'un point coloré implémente l'interface des points et même plus ! Mais un raisonnement trop hâtif pourrait conduire à des erreurs. Par exemple, les points colorés avec égalité ne peuvent pas être considérés comme des points simples. En effet, la méthode `equal` des points colorés exige un argument qui soit un point coloré. C'est-à-dire que son résultat risque d'être erroné si son argument n'est pas au moins un point coloré. Considérer un point coloré comme un point permettrait da"pliquer à la méthode `equal` un point coloré vu comme un point, puis de lui passer point comme argument. L'exécution de ce message va tenter d'accédeer au champ couleur de l'argument qui n'existe pas.

1.6.1 Héritage Multiple

L'héritage simple permet d'étendre une définition de classe avec de nouvelles variables d'instance et de nouvelles méthodes. L'héritage multiple permet de construire une nouvelle classe à partir de plusieurs classes.

L'héritage multiple peut être réalisé de deux façons différentes. La plus naturelle repose sur la concaténation des enregistrements, mais cette opération est très difficile à typer.

Une autre approche consiste à abstraire les classes par rapport à une hypothétique classe parente. On dit quelque fois que les classes sont traitées comme des *wrappers* (fonctions d'emballage). Plus précisément, cela revient à définir la classe des points comme suit :

```
let point_C =
  let point_R super_R v = super_R @ {x = ref v} in
  let point_M super_M =
    let getx R self = ...
    ...
    in super_M @ {getx = getx; setx = setx; print = print}
  in Class (point_R, point_M);;
```

On peut alors définir un *wrapper* qui ajoute la couleur à une classe arbitraire.

```
let coloré_C =
  let coloré_R super_R v = super_R @ {c = v} in
  let point_M super_M =
    let getc R self = ...
    let print R self = super_M. print; ...
```

```
    in super_M @ {getc = getc; print = print}
  in Class (point_R, point_M)
```

L'héritage de deux classes se fait de façon systématique

```
let inherit (Class (R1, M1)) (Class (R2, M2)) =
  let R super_R (x1,x2) = R2 (R1 super_R x1) x2 in
  let M super_M = M2 (M1 super_M) in
  Class (R, M);;
```

Ainsi, nous obtenons la classe des points colorés :

```
let point_coloré_C = inherit point_C coloré_C;;
```

Conclusion

Les exemples que nous avons présentés ci-dessus souffrent de l'absence de syntaxe concrète pour les objets. C'est le problème des codages en général, compensé par l'avantage, pédagogique, d'expliquer de nouvelles constructions en termes de constructions plus simples.

La possibilité de coder les objets en ML confirme la robustesse de ML. Elle infirme aussi l'idée répandue que les objets sont des constructions trop compliquées pour être utilisables. Malgré tout, la programmation avec des objets reste un pradigme délicat d'emploi ; Le "tout objet" est à proscrire. Mais avec un peu de rigueur, et de modération, les objets sont facilement abordables et utilisables.

1.7 Simulation des enregistrements extensibles en ML

Cette annexe décrit comment simuler les enregistrements extensibles en ML. Cela suppose que l'on connaisse à l'avance toutes les étiquettes utilisées dans tout le programme. Nous l'illustrons sur un exemple en considérant trois champs `a`, `b` et `c`. Les enregistrements extensibles seront simplement représentés par un enregistrement à trois champs, certains étant marqués "absents".

```
type ('a, 'b, 'c) abc = {a:'a; b:'b; c:'c};;
```

Nous voulons garantir par le typage que l'accès à un champ autorisé réussit toujours. Nous utilisons deux types incompatibles pour représenter les champs présents et les champs absents :

```
type 'a pre = Pre of 'a;;
type abs = Abs;;
```

L'enregistrement vide est simplement

```
let nul = {a = Abs; b = Abs; c = Abs};;
val nul : (abs, abs, abs) abc = {a=Abs; b=Abs; c=Abs}
```

Tous les autres enregistrements peuvent être construits par modification de champs à partir de l'enregistrement vide ; il suffit de définir trois fonctions :

```
let extend_a r x = {a = Pre x; b = r.b;   c = r.c  };;
val extend_a : ('a, 'b, 'c) abc → 'd → ('d pre, 'b, 'c) abc = ⟨fun⟩

let extend_b r x = {a = r.a;   b = Pre x; c = r.c  };;
let extend_c r x = {a = r.a;   b = r.b;   c = Pre x};;
```

Par exemple, l'enregistrement `{a = 1; b = 2}` s'écrit :

```
let deux = extend_b (extend_a nul 1) 2;;
val deux : (int pre, int pre, abs) abc = {a=Pre 1; b=Pre 2; c=Abs}
let trois = extend_b (extend_c deux 3) 4;;
val trois : (int pre, int pre, int pre) abc = {a=Pre 1; b=Pre 4; c=Pre 3}
```

De façon similaire, nous définissons trois fonctions d'accès à un enregistrement :

```
let extract_a r = let (Pre v) = r.a in v;;
val extract_a : ('a pre, 'b, 'c) abc → 'a = ⟨fun⟩

let extract_b r = let (Pre v) = r.b in v;;
let extract_c r = let (Pre v) = r.c in v;;
```

On vérifie que `extact_a ab` et `extract_b abc` sont typables, mais que `extract_c ab` ne l'est pas.

Le codage ci-dessus est tout à fait utilisable en pratique. En particulier, les exemples que nous avons écrits dans ce chapitre en utilisant des enregistrements extensibles peuvent être réécrit dans le noyau ML. Pour cela, il sera plus convivial de définir :

```
type 'a label = Label of 'a;;
let a = Label (extract_a, extend_a);;
let b = Label (extract_b, extend_b);;
let c = Label (extract_c, extend_c);;

let ( // ) r f = f r;;
let ( = ) (Label (_, extend)) x r = extend r x;;
let ( $ ) r (Label (extract, _)) = extract r;;
```

Les exemples ci-dessus s'écrivent alors très naturellement :

```
let deux = nul // (a = 1) // (b = 2) ;;
let trois = deux // (c = 3) // (b = 4);;
trois$b;;
```

Pour plus de sûreté, on pourra aussi contraindre les types des abréviations ci-dessus.

Chapitre 2

Objective ML

Introduction

Dans ce chapitre, nous présentons le langage *Objective Caml* d'abord par des exemples, puis nous formalisons les aspects objets, sans toutefois entrer dans les détails.

2.1 Introduction à Objective Caml

Objective Caml est un *langage à classes*[1]. Les objets ne sont pas créés directement, mais à partir de classes ou par modification d'objets existants. En particulier, les objets créés par modification d'autres objets partagent les mêmes méthodes. Un objet a donc une classe associée, celle à partir de laquelle il a été créé ou celle de l'objet à partir duquel il a été dupliqué. Cette propriété est importante. Comme tout invariant, elle permet d'analyser plus facilement les programmes, mais aussi de mieux les compiler.

Toutefois, les objets existent en temps que valeurs. Il est naturel et plus simple d'autoriser leur création directe. Nous choisirons cette approche pour l'exposition, et nous rendrons la construction correspondante inaccessible à l'utilisateur.

2.1.1 Classes et objets

Un objet est composé d'un ensemble de variables d'instance et d'un ensemble de méthodes. Par exemple, un point sur la droite peut être défini par une variable d'instance `x` et deux méthodes `getx` et `print` :

```
let p = object val x = 1 method getx = x method print = print_int x end;;
```

Son type est :

```
p : ⟨ getx : int; print : unit ⟩
```

L'objet `p` est identique a celui que l'on aurait obtenu à partir d'une définition de classe :

[1] En anglais on dit aussi *class-based language* par opposition à *object-based language*

```
class point x =
  val x = x
  method getx = x
  method print = print_int x
end;;
class point (int) = val x : int method getx : int method print : unit end
```

La définition de la classe `point` a trois effects sur l'environnement :

- ajouter la définition à la variable de classe `point` dans l'environnement des classes.
- ajouter une valeur `new point` qui permet de créer des objets de la classe correspondante.
- ajouter des abbréviations de type `point` et `#point`. La première est égale au type des points ⟨`getx : int; print : unit`⟩. La seconde est plus générale et s'unifie avec les types de toutes les sous-classes de `point`.

Vérifions que l'on peut créer de nouveaux points et leur envoyer des messages :

```
let p = new point 1;;
p#print;;
```

L'envoi de message est polymorphe, comme le montre l'exemple suivant :

```
let send_print  p = p#print;;
val send_print : ⟨ print : 'a; .. ⟩ → 'a = ⟨fun⟩
```

Le type de cette fonction indique que l'argument doit être un objet qui possède une méthode `print` de type `'a` où `'a` est la valeur retournée par le message. En particulier `send_print` peu être appliqué à `p`.

2.1.2 Héritage

Les points colorés peuvent être définis à partir des points par héritage :

```
class point_coloré x c =
  inherit point x as parent
  val color = c
  method getc = color
  method print = parent#print; print_string color
end;;
```

L'héritage a le même effet que de recopier dans la classe définie toutes les définitions de la classe héritée. La classe parente peut être liée à une variable comme `parent` dans l'exemple ci-dessus. Une variable liant une classe parente n'est pas une variable ordinaire ; elle ne peut être utilisée que pour envoyer des messages (comme dans `parent#print`). L'envoi d'un message m_1 à une classe parente P par une méthode m_2 définie dans une classe C revient à appliquer à l'objet exécutant la méthode m_2 la méthode m_1 définie dans la classe P, au lieu de sa valeur prise dans l'objet exécutant m_2.

L'héritage peut aussi être multiple. Par exemple, définissons une classe `couleur` :

```
class couleur c =
  val color = c
  method getc = color
  method print = print_string color
end;;
```

Les points colorés peuvent alors être défini par héritage multiple des classes `point` et `couleur`.

```
class point_coloré x c =
  inherit point x as point
  inherit couleur c as couleur
  method print = point#print; couleur#print
end;;
```

Les deux classes sont bien sûr équivalentes.

```
let p = new point_coloré 2 "blanc_cassé" in p#print;;
```

2.1.3 Auto-référence

L'auto-référence est la possibilité pour une méthode de référencer l'objet qui a provoqué l'exécution de cette méthode. Cet objet est en général lié à la variable `self` ; en Objective Caml la liaison de cet objet à une variable doit être explicite et utilise la construction *as*. L'auto-référence est le plus souvent utilisée pour envoyer d'autres messages à `self` ; en quelque sorte, la méthode exécutée délégue une partie du calcul à d'autres méthodes. Par exemple, on aurait pu définir la classe `point` ainsi :

```
class point u as moi_même =
  val x = u
  method getx = x
  method print = print_int moi_même#getx
end;;
```

Les points créés par cette classe auront le même comportement que ceux créés par l'ancienne classe. Toutefois, la nouvelle classe est différente. En particulier, si une autre classe en hérite et puis modifie la définition de la méthode `getx`, alors la méthode `print` subit l'effet de cette modification, alors qu'elle resterait inchangée si la classe héritait de l'ancienne définition. Cet effet lié à l'envoi de message à l'objet auto-référencé est appelé *liaison tardive*.

Un autre usage de l'auto-référence est la possibilité de retourner l'objet lui-même en résultat. On peut en effet écrire une classe :

```
class moi_même() as moi_même =
```

```
  method moi_même = moi_même
end;;
class moi_même (unit) : 'a = method moi_même : 'a end
```

Exercice 2 *Que retourne le programme suivant*

```
let p = new moi_même () in p # moi_même # moi_même;;
```

□

Objective Caml fournit une construction pour retourner une copie de l'objet dans laquelle certaines variables d'instance ont été modifiées. Par exemple, un entier peut être représenté par :

```
class entier () =
  val n = 0
  method nul () =  (n = 0)
  method add p = {⟨ n = n + p ⟩}
end;;
```

A chaque appel, la méthode `add` crée un nouvel entier dont la variable d'instance n est incrémenté de un. Il y a donc deux façons de créer des objets. La façon habituelle consiste à demander un nouvel objet de la classe

```
let p = new point 1;;
```

Cette solution est générale et s'applique toujours possible. On peut également créer un objet de la classe C par reproduction d'un objet déjà construit de la classe C à condition que la classe C ait elle-même définit une méthode de reproduction. Par exemple, on peut étendre la classe point avec une méthode de reproduction :

```
class point_neuf x =
  inherit point x
  method neuf y = {⟨ x = y ⟩}
end;;
let un_point = new point_neuf 0;;
```

Il est alors équivalent d'écrire `new point` ou `un_point.neuf`. La création d'un objet par reproduction n'est intéressante que lorsque, comme dans l'exemple des entiers, la reproduction initialise les variables d'instances à des valeurs dépendant de leur ancienne valeur.

2.1.4 Style impératif

Jusqu'à maintenant, nous avons défini des objets en style purement applicatif. Il est bien sûr possible de programmer avec des objets dans un style impératif. Évidemment, il est possible d'utiliser des références à l'intérieur des objets.

```
class point_mutable x as moi =
```

```
  val x = ref x
  method getx = !x
  method setx y = x := y
  method print = print_int moi#getx
end;;
```

Le champ `x` n'est pas mutable mais il contient un objet mutable (une référence). En particulier, il est de type `int ref`. Une façon plus directe consiste à rendre ce champ mutable sans passer par une référence intermédiaire.

```
class point_mutable u as moi =
  val mutable x = u
  method getx = x
  method setx y = x ← y
  method print = print_int moi#getx
end;;
```

Le champ `x` est alors de type `int`. De façon plus concise, on aurait aussi pu étendre et modifier la classe `point` :

```
class point_mutable u =
  inherit point u
  val mutable x
  method setx y = x ← y
end;;
```

Il existe un module `Oo` qui fournit une fonction `copy` qui permet de duppliquer un objet : `Oo.copy p` retourne une copie superficielle de l'objet `p`. Les variables mutables ont les mêmes valeurs que celle de l'originale, mais elles seront mutées indépendemment.

```
Oo.copy;;
- : (⟨ .. ⟩ as 'a) → 'a = ⟨fun⟩
```

Cette fonction peut-être utilisée pour duppliquer un objet :

Exercice 3 *Écrire une classe* `copie` *qui fournit une méthode* `copy` *qui dupplique l'objet à qui elle est envoyée. Quand la méthode* `copy` *est-elle intéressante ?* □

Exercice 4 *La fonction* `Oo.copy` *n'effectue qu'une copie superficielle de l'objet. Comment peut-on effectuer une copie profonde ? Donner un exemple.* □

Exercice 5 *Il existe deux façons de représenter les points en style applicatif et en style impératif. Les comparer.* □

Exercice 6 *Pour les points écrits en style impératif, il existe deux façons de duppliquer un objet. Les comparer.* □

2.1.5 Classes paramétriques

Les références peuvent aussi être implémentées par une classe. Toutefois, le programme suivant échoue :

```
class ref x =
  val mutable x = x
  method get = x
  method set y = x ← y
end;;
```

En effet, les méthodes sont polymorphes, et la classe `ref` est naturellemnt paramétrique. Il est bien sûr possible d'écrire une version monomorphe :

```
class ref x =
  val mutable x = (x:int)
  method get = x
  method set y = x ← y
end;;
class ref (int) =
  val mutable x : int
  method get : int
  method set : int → unit
end
```

Pour écrire une classe paramétrique, il est nécessaire d'indiquer explicitement les variables de types dont elle dépend. Il faut également que ces variables soient liées dans le corps de la définition :

```
class 'a ref x =
  val mutable x = (x:'a)
  method get = x
  method set y = x ← y
end;;
class 'a ref ('a) =
  val mutable x : 'a
  method get : 'a
  method set : 'a → unit
end
```

Dans un définition de classe un paramètre de type peut devoir être instancié par un type plus précis. Dans le type de la class, cela est alors indiqué par une clause `constraint`.

```
class 'a ref (x:'a) =
  val mutable x = x + 1
  method get = x method set y = x ← y
end;;
```

```
class 'a ref ('a) =
  constraint 'a = int
  val mutable x : int
  method get : int
  method set : int → unit
end
```

Une constrainte peut également être mise explicitement :

```
class 'a ref (x:'a) =
  constraint 'a = 'b list
  val mutable x = x
  method get = x method set y = x ← y
end;;
class 'a ref ('a) =
  constraint 'a = 'b list
  val mutable x : 'a
  method get : 'a
  method set : 'a → unit
end
```

2.1.6 Coercion

Il est parfois possible de coercer un objet vers un autre objet. Par exemple un point coloré peut aussi être considéré comme un point. On écrit cela

```
let p = new point 1 and q = new point_coloré 2 "blanc_cassé";;
val p : point = ⟨obj⟩
val q : point_coloré = ⟨obj⟩
let pq = [p ; (q :point_coloré :⟩ point)];;
val pq : point list = [⟨obj⟩; ⟨obj⟩]
```

La coercion d'une expression a de type τ_1 vers le type τ_2 est possible si l'expression a peut être considéré de type τ_2 en toute sécurité. Nous verrons dans le chapitre suivant qu'il en est ainsi lorsque les types τ_1 et τ_2 sont dans une relation de sous-typage que nous définirons formellement.

Dans bien des cas, il est commode de ne pas indiquer seulement le type du domaine de la coercion. Pour cela un ensemble de primitives $(\_ :> \tau)$ sont founies, par exemple :

```
fun x → (x :⟩ point);;
- : ⟨ getx : int; print : unit; .. ⟩ → point = ⟨fun⟩
```

Cependant, le type du domaine est calculé à partir du type du co-domaine, et il n'est pas vrai que $(q :> t)$ soit toujours bien typé si $(q : \tau' :> \tau)$ pour un certain τ' la seconde forme étant plus générale.

Cela ne doit pas être considéré comme une coercions, mais comme une primitive avec un type fixé et la construction $(q :> \tau)$ doit être considéré comme l'application de cette fonction à q. Dans le cas présent, il se trouve que cette application est bien typé si q est un point coloré.

La coercion ne modifie pas la valeur coercée. En particulier, un point coloré peut être considéré comme un point, mais l'envoi d'un message à un point coloré vu comme un point aura le même effet que l'envoi du même message au même objet vu comme un point coloré.

La coercion de type est une perte d'information. Une fois cette information oubliée, il n'est plus possible de revenir en arrière. Par exemple un point coloré vu comme un point ne plus être considéré comme un point coloré à nouveau, sauf à avoir gardé un pointeur sur le même objet sans perte d'information, comme le montre le programme suivant.

```
let q' = (q :⟩ point) in (q, q');;
− : point_coloré * point = ⟨obj⟩, ⟨obj⟩
```

Évidemment ! puisque la liaison de q est statique.

Lorsqu'il est nécessaire de réifier ultérieurement un objet vers sa véritable classe, alors on devra utiliser des types concrets :

```
type any_point = Point of point | Point_coloré of point_coloré;;
type any_point = Point of point | Point_coloré of point_coloré

let pq = [Point p; Point_coloré  q];;
val pq : any_point list = [Point ⟨obj⟩; Point_coloré ⟨obj⟩]

let send_any f = function Point p → f p | Point_coloré q → f (q:⟩point);;
val send_any : (point → 'a) → any_point → 'a = ⟨fun⟩

List.iter (send_any (fun x → x#print)) pq;;
12blanc_cassé− : unit = ()
```

Ceci peut paraître un peu lourd, se produit raisonable peu souvent.

2.1.7 Méthodes virtuelles

Ils est possible de déclarer une méthode d'une classe sans la définir, ni l'hériter. Elle est alors dite virtuelle. Une méthode utlisé mais non déclarée ni héritée est implicitement virtuelle. Une classe qui contient des méthodes virtuelle est elle même virtuelle ; elle doit être annotée par le mot clef `virtual`, et il n'est pas possible d'en prendre d'instance. Les méthodes virtuelles devront être définies dans une sous-classe.

2.1.8 Variables d'instance privées

Il est possible d'annoter les variables d'instance par la propriété `private`. Cela a pour effet de les rendre locale à la définition de la classe et donc invisibles dans le type de la classe, et a fortiori dans les sous-classes.

2.2 Génération automatique de types

Les définitions de classe génèrent de abbréviations de types automatiquement. Ces abbréviations utilisent le partage créé dans les types par unification pendant le typage de la classe pour déterminer l'abbrévaition correspondante à la classe, en particulier pour séparer les occurences de `self` et des paramètres d'autres types qui pourraient leur être accidentellement égaux.

Correction des exercices

Chapitre 3

Les Objets : encapsulation, récursion et sous-typage

Introduction

Dans le premier chapitre, nous avons expliqué comment simuler les objets dans le langage Caml. Nous avons justifié cette approche par son aspect pédagogique : elle explique les objets en termes de constructions plus élémentaires. Nous avons expliqué que l'encapsulation pouvait être assurée par fermeture, ou par abstraction de type. Nous avons ensuite abandonner le codage de l'abstraction par fermeture, qui rend certaines opérations plus difficiles, notamment la reproduction des objets, et qui n'induit pas un partage de l'enregistrement des méthodes d'une même classe. Nous avons donc choisi l'autre alternative qui s'appuyant sur une encapsulation par abstraction de type, sans toutefois réaliser cette abstraction. Il nous reste donc à expliquer en détail comment réaliser l'abstraction par un mécanisme de typage.

Dans le second chapitre, nous avons présenté un langage avec des constructions primitives pour les opérations sur les objets. Nous avons modélisé l'encapsulation par abstraction. S'agit-il d'une encapsulation par fermeture ou d'une encapsulation par abstraction de type ?

Nous avons également observé que le recours aux types récursifs était inévitable : il permettent qu'une méthode retourne `self` ou le prenne en argument. Il faudra donc également revenir sur les types récursifs et expliquer leur traitement.

Enfin nous avons mentionné la possibilité de considérer parfois les objets d'une sous-classe comme des objets de leur classe parente par un mécanisme de sous-typage. Cela est-il toujours possible ? Nous verrons dans la section suivante que la réponse est négative. Il faudra pour expliquer cela définir le sous-typage plus formellement.

Dans ce chapitre consacré aux objets le but est de donner un sens plus précis à ces trois aspects nouveaux : encapsulation, récursion sous-typage. En particulier, il sera nécessaire de comprendre leur interaction pour découvrir ce que sont réellement les objets.

3.1 Un contre-exemple

Nous décrivons brièvement un contre-exemple à la coercion des objets d'une sous-classe vers une classe parente. Ceci devrait justifier une étude approfondie des objets qui sera réalisée dans le reste de ce chapitre. Considérons les points colorés mutables :

```
class point x =
  val mutable x = (x:int)
  method getx = x
end;;

let point = new point 1;;
class color c =
  val c = (c : string)
  method getc = c
end;;

class point_coloré x c =
  inherit point x
  inherit color c
end;;

let point_coloré = new point_coloré 2 "blanc-cassé";;
```

Les points colorés peuvent être considérés comme des points. Ajoutons une méthode qui teste l'égalité de deux points.

```
class point_physique x as self : 'a =
  inherit point x
  method equal (y:'a) = self#getx = y#getx
end;;

let point_physique = new point_physique 3;;
```

Sur les points colorés l'égalité doit aussi tenir compte de la couleur :

```
class point_coloré_physique x c as self =
  inherit point_physique x as pp
  inherit color c
  method equal y = pp#equal y & self#getc = y#getc
end;;

let point_coloré_physique = new point_coloré_physique 4;;
```

Les points colorés avec égalité peuvent-ils être considérés à la fois comme des points colorés ou comme des points avec égalité ? La première réponse est affirmative, mais la seconde est négative.

En effet, considérons un point coloré p avec égalité. Si nous considérons p comme un point simple avec égalité, il posséderait la méthode `equal` qui reçoit un point avec égalité en argument et retourne un booléen. Il serait donc possible d'envoyer à p le message `equal` en passant comme argument un point avec égalité non coloré q. Cependant l'exécution de la méthode `equal` de p va envoyer le message `getc` à q qui ne pourra pas le recevoir. Par contre, p peut être considéré comme un point coloré sans égalité en toute sécurité : on peut s'en convaincre en essayant d'envoyer tous les messages des point colorés à p.

Pour savoir si un objet d'une classe peut être considéré comme un objet d'une autre classe, il ne suffit donc pas de regarder la liste des méthodes auxquels il répond mais également au type de ses arguments, en particulier s'il reçoit un autre objet du même type que lui-même parmi ses arguments, la coercion n'est pas sure. Nous pouvons résumer les coercions possibles entre les différentes sortes de points dans la table suivante :

	`point`	`_coloré`	`_physique`	`_coloré_physique`
`point`	√			
`_coloré`	√	√		
`_physique`	√	√	√	
`_coloré_physique`	√	√	?	√

Exercice 7 *Dans l'exemple ci-dessus, la méthode d'égalité est différente pour les points colorés et pour les point simples. Il n'est donc pas trop surprenant que la coercion soit impossible.*

1. *Donner un exemple où il n'y a pas forcément d'erreur à l'exécution, mais qui soit indistingable par le typage d'un exemple qui produit une*

   erreur à l'exécution.

2. *Donner un exemple où bien que la méthode soit inchangée, la coercion produirait encore une erreur à l'exécution.*

□

Exercice 8 *A partir des points, des couleurs et points colorés, définir des cercles puis des cercles colorés. Vérifiez que certaines coercions sont possibles.*

Ajouter aux cercles une méthode qui en reçoit un autre cercle q en argument et construit un nouveau cercle co-centrique à `self` *et circonscrit à q. Le nouveau cercle sera coloré seulement si les deux cercles* `self` *et q le sont. Quelles sont les coercions possibles ?* □

3.2 Les objets en Objective Caml

Dans cette section présentons le langage Objective Caml plus formellement

Notation

Une liaison est une paire (k, t) composée d'une cl ;é k et d'un élément t. On lécrit aussi $k = t$ lorsque t est une expression ou $k : t$ lorsque t est un type. Dans la notation, les liaisons peuvent aussi étre prédixées par un mot clé, par exemple `val` $x = a$ ou `method` $m = a$. Le mot clé est redondant et rapelle la classe syntaxique de la clé.

Une séquence peut être vu comme une fonction qui à la clé k associe l'élément t avec laquelle elle est apairée et la plus à droite. On note $dom\,(S)$ le domaine (i.e. l'ensemble des clés) d'une séquence S. On écrit $+$ la concaténation des séquence avec priorité à droite. On écrit $\oplus$ la contaténation de deux séquences compatibles, c'est à dire égales sur l'intersection de leur domaines.

On écrit aussi $\overrightarrow{t}$ pour un tuple d'éléments de la classes syntaxique représentée par t.

3.2.1 Le noyau Caml

Les expression sont définie par la grammaire suivante :

$$
\begin{aligned}
a &::= z \mid \texttt{fun}\ (x)\ a \mid a\ a \mid \texttt{let}\ z = a\ \texttt{in}\ a \\
z &::= x \\
\\
\tau &::= \alpha \mid \tau \rightarrow \tau \mid \bar{\tau} \\
\sigma &::= \forall\, \bar{\alpha}.\, \tau
\end{aligned}
$$

On a volontairememnt séparer les variables qui peuvent être liées z de celles qui peuvent également être abstraites x, en vue des extensions ci-dessous.

Les jugements de typage sont de la forme $A \vdash a : \tau$, et donnés par la figure 3.1

$$
\text{(VAR)}\ \frac{x : \forall\, \alpha.\, \tau_o \in A}{A \vdash x : \tau_0[\bar{\tau}/\bar{\alpha}]}
\qquad
\text{(FUN)}\ \frac{A \oplus x : \tau_1 \vdash a : \tau_2}{A \vdash \texttt{fun}\ (x)\ a : \tau_2}
\qquad
\text{(APP)}\ \frac{A \vdash a_1 : \tau_2 \rightarrow \tau_1 \qquad A \vdash a_2 : \tau_2}{A \vdash a_1\ a_2 : \tau_2}
$$

$$
\text{(LET)}\ \frac{A \vdash a_1 : \tau_1 \qquad A \oplus (x : \texttt{Gen}(\tau_1, A)) \vdash a_2 : \tau_2}{A \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}
$$

Figure 3.1 : règles de typage pour Caml

L'évaluation est simplement modélisée par la β-réduction. 3.2. La réduction peut se produire dans les contextes déterminés pas la grammaire E. Il faut compléter les règles de réduction par :

$$E[a_1] \longrightarrow E[a_2] \qquad \text{si} \qquad a_1 \longrightarrow a_2$$

qui exprime qu'un terme est réduit en le réduisant à un rédex.

Contextes d'évaluation

$$E ::= [] \mid E\ a \mid v\ E$$

Règles de réduction

$$(\texttt{fun}\ (x)\ a)\ v \longrightarrow a[v/x]$$
$$\texttt{let}\ x = v\ \texttt{in}\ a \longrightarrow a[v/x]$$

Figure 3.2 : Sémantique du noyau Caml

3.2.2 Extension de la syntaxe pour les objets

Bien que l'implémentation Objective Caml ne permette pas à l'utilisateur de construire directement des objets, ces constructions existent dans le langage.

Nous supposons deux nouvelles classes syntaxiques u et m pour les champs et les méthodes. On étend étend la syntaxe de Caml avec les constructions suivantes :

$$a \ldots a\#m \mid \langle\ \texttt{val}\ u_i = a_i{}^{i \in 1..p},\ \texttt{method}\ m_j = a_j{}^{j \in q} \rangle \mid \texttt{self}$$

Dans cette présentation formelle de Objective Caml, il est plus simple de traiter `self` comme un mot clé.

Il faut aussi enrichir les types avec des types objets qui sont similaires aux types enregistrements. Nous suposons donné une nouvelle collection de variables $\rho \in \mathcal{P}$ appelées variables de rangée

$$\tau \ldots \mid \langle \tau' \rangle \mid \tau\ \texttt{as}\ \alpha \qquad\qquad \tau'(m : \tau; \tau') \mid \rho \mid \emptyset$$

Les types d'objets qui se terminent avec une variable de rangée sont dits *ouverts*, alors que les autres sont dits *fermés*.

Les types sont également considérer modulo des équations permettant de ré-ordonner les champs :

$$(m_1 : \tau_1; m_2 : \tau_2; \tau') = (m_2 : \tau_2; m_1 : \tau_1; \tau')$$

Une étiquette ne peut apparaître qu'une seule fois dans un type d'objet. Nous imposons également que deux types d'objets qui se terminent avec la même variable de rangée soient égaux. Cette règle permet d'éliminer des types incorrects tels que : $\langle \rho \rangle \to \langle m : \tau; \rho \rangle$.

Cette condition est assurée en remplaçant dans la syntaxe concrète les variables de rangée par des ellipses, et en utilisant les alias pour exprimer le partage. Par exemple, $\langle m : \tau; \rho \rangle \to \langle m : \tau; \rho \rangle$ is written $(\langle m : \tau; .. \rangle\ \texttt{as}\ \alpha) \to \tau'$.

Les alias τ `as` α permettent d'exprimer des types récursifs, indispensables pour les objets. Par exemple $(\mu\alpha.\tau_1) \to \tau_2$ peut s'écrire $(\tau_1\ \texttt{as}\ \alpha) \to \tau_2$. Il sont aussi plus généraux car α est liée à τ_1 dans τ_1 et τ_2. Ils permettent d'exprimer le partage des variables de rangées, et plus généralement le partage des expressions de types qu'il est important de conservé pour le bon fonctionnement des déclarations automatiques d'abbréviations (section 2.2)

3.2.3 Typage des objets

La règle de typage la plus simple pour les objets est la suivante :

$$\frac{A \oplus (\texttt{self} : \langle m_j : {\tau_j}^{j \in J} \rangle) \vdash a_j : {\tau_j}^{j \in J}}{A \vdash \langle\ \texttt{method}\ m_j = {a_j}^{j \in J} \rangle : \langle m_j : {\tau_j}^{j \in J} \rangle}$$

Un objet peut aussi avoir des variables d'instances, qui doivent être cachées dans son type final.

Le typage des variables d'instances $\texttt{val}\ u_i = {a_i}^{i \in 1..p}$ produit un petit environnement de typage $\texttt{val}\ u_i = \tau_i$ dans lequel les méthodes sont ensuite typées :

$$\frac{A^\star \vdash a_i : {\tau_i}^{i \in 1..p} \qquad A^\star \oplus (\ \texttt{val}\ u_i : {\tau_i}^{i \in 1..p}) \oplus (\texttt{self} : \langle m_j : {\tau_j}^{j \in 1..q} \rangle) \vdash a_j : {\tau_j}^{j \in 1..q}}{A \vdash \langle\ \texttt{val}\ u_i = {a_i}^{i \in 1..p}\ \texttt{method}\ m_j = {a_j}^{j \in 1..q} \rangle : \langle m_j : {\tau_j}^{j \in 1..q} \rangle}$$

L'environnement $A^\star$ est égal à A privé de `self` et de toutes les liaisons $\texttt{val}\ u : \tau$. La raison est que les variables d'instances et `self` liées dans A doivent toujours être les composantes du même objet, car elles partagent implicitement le type de self.

Les règles pour les variables d'instances et l'envoi de message sont :

$$\text{(FIELD)}\ \frac{\texttt{val}\ u : \tau \in A}{A \vdash u : \tau} \qquad \text{(SEND)}\ \frac{A \vdash a : \langle m : \tau_k ; \tau \rangle}{A \vdash a\#m : \tau_k}$$

3.2.4 Sémantique des objets

Pour évaluer les objets, on autorise l'évaluation des variables d'instance. La seule nouvelle règle de réduction est l'envoi de message (voir figure 4.2).

$$E ::= \ \dots \mid \langle\ \texttt{val}\ u_i = {v_i}^{i \in 1..p};\ \texttt{val}\ u = E;\ \texttt{val}\ u_j = {a_j}^{j \in 1..q};\ \texttt{method}\ m_k = {a_k}^{k \in 1..k} \rangle$$

Réduction de l'envoi de message

$$\langle w \rangle \# m \longrightarrow w(m)[\langle w \rangle / \texttt{self}][w(u)/u \in \mathit{dom}\,(w)]$$

3.2.5 Redéfinition des champs

La construction $\{\langle u_1 = a_1; \dots u_n = a_n \rangle\}$ permet de dupliquer l'objet lié à `self` en modifant les variables d'instances mentionnées. Formellement :

$$a ::= \ \dots \mid \{\langle u = a; \dots u = a \rangle\} \qquad \text{(OVERRIDE)}\ \frac{A \vdash a_i : {\tau_i}^{i \in 1..p} \qquad A \vdash \texttt{self} : \tau_y}{A \vdash \{\langle\ \texttt{val}\ u_i = {a_i}^{i \in 1..p} \rangle\} : \tau_y}$$

$$\langle w \rangle \# m \longrightarrow w(m)[\langle w \rangle / \texttt{self}][w(u)/u \in \mathit{dom}\,(w)][w@(\ \texttt{val}\ u = {a_u}^{u \in V}) / \{\langle u = {a_u}^{u \in V} \rangle\}]^{V \subset U}$$

Le reste de ce chapitre n'est pas à jour

3.3 Types récursifs

3.3.1 Pourquoi ?

Que les objets soient simulés dans un langage avec enregistrements polymorphes ou primitifs comme dans Objective Caml, il est nécessaire que les types soient récursifs. Sans cela, il ne serait pas possible pour une méthode de retourner ou de prendre en argument une structure dont le type contient `self`. Cela interdirait en particulier la duplication des objets et les méthodes binaires qui sont deux aspects importants de la programmation à objets.

3.3.2 Comment ?

Les types non récursifs sont représentés par des termes. Le langage ML peut manipuler des structures récursives grâce à des types récursifs définis au travers de types concrets : Par exemple les listes sont représenté par un type non récursif α `list` ... qui cache en fait le type :

$$\texttt{rec}\ \alpha_0.[\texttt{Cons of}\ \alpha \times \alpha_0 \mid \texttt{Nil}]$$

Cette façon de contourner les types récursifs par des symboles associés à des définition récursives a l'inconvénient majeur de rendre tous les types représentant des types récursifs différents incompatibles entre eux.

Cette méthode ne peut donc pas s'appliquer aux objets, l'envoi de message polymorphe exige que les objets possédant une même méthode aient une compatibilité minimale permettant l'envoie du message correspondant. Nous devons choisir un autre mécanisme que les types concrets.

Il n'y a pas de difficulté à ajouter à ML des types récursifs implicites. La synthèse de type qui repose sur l'unification de premier ordre ne pose pas de problème. Les algorithmes d'unification sont souvent implémentés sur des types récursifs puis ensuite réglés pour les rejetés.

L'ajout de types récursifs en Objective Caml est un ajout mineur. Toutefois l'interaction des types récursifs avec le sous-typage que nous verrons ci-dessous est plus délicate. Il est donc important de bien les formaliser.

Les types récursifs peuvent être représentés de façon équivalente par

- Des μ-termes.
- Un terme et un ensemble d'équations de la forme $\alpha = \tau$.
- Des arbres infinis réguliers (i.e. ayant un ensemble fini de sous-arbres).

Les alias sont une formes intermédiaire entre les μ-termes et les équations. Un terme τ avec alias se transforme en la paire du terme obtenu en remplaçant tous les alias dans τ par la variable liée, et l'ensemble d'équation égalant les alias (en faisant attention au renommage des variables).

Les représentations par μ-termes par des équations entre termes non récussifs ou par des alias sont en fait très voisines. Elles sont toutes trois syntaxiques (proche de la syntaxe). La dernière représentation est plus sémantique, puisqu'un arbre infini est une fonction des occurrences vers l'union de l'ensemble des symboles et de l'ensemble des variables.

Intuitivement un arbre infini régulier associé à un μ-termes est obtenu par déroulement des récursions. Formellement, il peut aussi être défini par un automate. Plusieurs μ-termes définissent le même arbre infini régulier.

Dans ce qui suit nous exclurons la notation avec alias en raison de son mécanisme global de liaison. Nous les remplaçons par des μ-termes.

On peut définir l'égalité des types récursifs sur les μ-termes par un système de règles d'inférence :

$$\text{(Ref)}\quad \tau = \tau \qquad \text{(Arrow)}\ \frac{\tau_1 = \tau_1' \qquad \tau_2 = \tau_2'}{\tau_1 \to \tau_2 = \tau_1' \to \tau_2'} \qquad \text{(Object)}\ \frac{\tau_1 = \tau_1' \qquad t_n = \tau_n'}{\begin{array}{c}\langle x_1 : \tau_1, \dots x_n : \tau_n\rangle = \langle x_1 : \tau_1', \dots x_n : \tau_n'\rangle \\ \langle x_1 : \tau_1, \dots x_n : \tau_n ; \rho\rangle = \langle x_1 : \tau_1', \dots x_n : \tau_n' ; \rho\rangle\end{array}}$$

$$\text{(Trans)}\ \frac{\tau_1 = \tau_2 \qquad \tau_2 = \tau_3}{\tau_1 = \tau_3} \qquad (\mu)\ \frac{\tau_1 = \tau_2}{\texttt{rec}\ \alpha_1.\tau_1 = \texttt{rec}\ \alpha_2.\tau_2}$$

$$\text{(Unfold)}\ \frac{\texttt{rec}\ \alpha.\tau = \tau[\tau/\alpha] \qquad \tau_1 = \tau[\tau_1/\alpha] \qquad \tau_2 = \tau[\tau_2/\alpha]}{\tau_1 = \tau_2}$$

La dernière règles permet d'identifier des types récursifs définis sur différentes périodes.

Exercice 9 *Montrer que* $f(\texttt{rec}\ \alpha.f(f(\alpha)))$ *et* $\texttt{rec}\ \alpha.f(\texttt{rec}\ \alpha)$ *sont égaux.* □

On peut facilement calculer cette égalité sur la forme équationnelle, en vérifiant que les deux termes possèdent un unificateur commun qui ne modifient pas leurs variables libres.

3.4 Sous-typage

3.4.1 Pourquoi ?

L'exemple donné dans l'introduction montre que la possibilité de coercer un objet vers un autre n'est pas si intuitive qu'elle ne le parait. Il est nécessaire de définir formellement quand une coercion est-elle possible.

L'opération de coercion de base est entièrement explicite ; elle exige que le domaine τ_1 et le co-domaine τ_2 de la coecions soient à la fois précisés. Elle repose sur une opération de sous-typage, notée $\tau_1 \leq \tau_2$ ne dépendant que des types τ_1 et τ_2 concernés par la coercion. C'est donc la relation de sous-typage $\leq$ qu'il nous faut préciser.

3.4.2 Comment ?

Intuitivement nous souhaitons que la relation $\tau_1 \leq \tau_2$ implique que tout objet de type τ_1 peut être considéré comme un objet de type τ_2. Par exemple, un point coloré de type $\tau_1 \equiv \langle print : unit; getx : int; ..\rangle$ peut être utilisé là où un point peut l'être $\langle print : unit; getx : int; ..\rangle$,

La relation de sous-typage est engendrée par l'inclusion sur les types d'objets :

$$\langle a_1 : \tau_1, \ldots a_n : \tau_n, \ldots a_{n+p} : \tau_{n+p}\rangle \leq \langle a_1 : \tau_1', \ldots a_n : \tau_n'\rangle$$

et par propagation au travers des constructeurs de type. Elle doit aussi être fermée par réflexivité et transitivité. Plus précisément, si l'on exclue momentanément les types récursifs, la relation de sous-typage est la plus petite relation telle que :

$$\text{(Arrow-Cong)}\quad \frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

$$\text{(Object-Cong)}\quad \frac{\tau_1 \leq \tau_1' \qquad \tau_n \leq \tau_n'}{\begin{array}{c}\langle a_1 : \tau_1, \ldots a_n : \tau_n, \ldots a_{n+p} : \tau_{n+p}\rangle \leq \langle a_1 : \tau_1', \ldots a_n : \tau_n'\rangle \\ \langle a_1 : \tau_1, \ldots a_n : \tau_n, \ldots a_{n+p} : \tau_{n+p}; \tau\rangle \leq \langle a_1 : \tau_1', \ldots a_n : \tau_n'\rangle\end{array}}$$

$$\text{(Reflex)}\quad \frac{\tau_1 = \tau_2}{\tau_1 \leq \tau_2}$$

Deux types d'objets ouverts ne peuvent être dans une relation de sous-typage que s'ils sont égaux. Ceci résulte du fait de la restriction imposée à la formation des types d'objets ouverts qui ne peuvent avoir la même variable de rangée que s'il sont égaux.

La transitivité de la relation de sous-typage

$$\text{(Trans)}\quad \frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$$

est une règle dérivable.

Pour étendre la relation de sous-typage aux types récursifs, il est plus simple de les représenter par des arbres rationnels. En fait, on va inverser la relation ci-dessus, et définir les conditions nécessaire pour que deux arbres soient dans une relation de sous-typage. Précisément, la relation de sous-typage est définie sur les arbres rationnels comme la plus grande relation satisfaisant hypothèses suivantes :

$$\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' \Longrightarrow \tau_1' \leq \tau_1 \wedge \tau_2 \leq \tau_2'$$

$$\langle ..x : \tau, ..\rangle \leq \langle ..x : \tau, ..\rangle \Longrightarrow \tau \leq \tau \qquad \text{(types ouverts ou fermés)}$$

$$\alpha \leq \tau \Longrightarrow \alpha = \tau$$

$$\tau \leq \alpha \Longrightarrow \alpha = \tau$$

$$\tau \leq \tau_1 \to \tau_2 \Longrightarrow \tau \equiv \tau_1' \to \tau_2'$$

$$\tau \leq \langle x_1 : \tau_1, ..x_n : \tau_n\rangle \Longrightarrow \tau \equiv \langle x_1 : \tau_1', ..x_n : \tau_n', ..\rangle \qquad \text{(type ouvert ou fermé)}$$

$$\tau \leq \langle x_i : {\tau_i}^{i \in 1..p}, \rho\rangle \Longrightarrow \tau = \langle x_i : {\tau_i}^{i \in 1..p}, \rho\rangle$$

Exercice 10 *Vérifiez que les points colorés et les points sont dans une relation de sous-typage. Vérifiez que ce n'est pas le cas lorsque tous deux ont une méthode avec égalité.* □

Exemple des points colorés et des points.

3.4.3 Coercions

La construction de coercion est explicite. Elle exige que le domaine et le co-domaine de la coercion soit précisés.

$$\frac{A \vdash e : \theta(\tau) \qquad \tau \leq \tau'}{A \vdash (a : \tau :> \tau') : \theta(\tau')} \qquad (\theta \text{ subsitution quelconque})$$

Pour des raisons de commodité, Objective Caml fournit également une collection de primitives de coercions

$$(\_ : \tau) : S(\tau) \rightarrow \tau$$

où S est un opérateur choisi tel que

$$S(\tau) \leq \tau \qquad\qquad \exists\theta(\theta(S(\tau)) = \tau \wedge \theta(\tau) = \tau)$$

Il se trouve que si τ n'est pas récursif, il existe une seule solution satisfaisant ces deux conditions. Elle est obtenue en remplaçant dans τ tous les objets fermés apparaissant uniquement en position covariante par leur types ouverts correspondants.

Ce n'est en général plus le cas pour des types récusifs. Par exemple, pour $\tau \equiv \langle x : \alpha \rangle$ `as` α que on peut choisir pour $S(\tau)$ égal à $\langle x : \tau; \rho \rangle$ ou bien $\langle x : \alpha; \rho \rangle$ `as` α qui mais il n'existe pas de solution plus générale que ces deux types à la fois.

Si α apparaît négativement dans α, alors $S(\texttt{rec}\ \alpha.\tau)$ est égal à $\texttt{rec}\ \alpha.\tau$. Sinon, $S(\texttt{rec}\ \alpha.\tau)$ est égal à $S(\tau[\texttt{rec}\ \alpha.S(\tau)/\alpha^k])$. Par exemple, pour l'exemple précédent, on prendra pour $\tau \equiv \langle x : \texttt{rec}\ \alpha_1.\langle x : \alpha_1, \rho_1 \rangle; \rho \rangle$.

Exercice 11 *Décrire tous les types que l'on peut choisir pour $S(\tau)$ pour le dernier exemple ci-dessus.* □

La relation de sous-typage est standard. On pourra lire [AC93]. L'algorithme simple donné ci-dessus est décrit dans [KP93]

Chapitre 4

Les Classes

Introduction

Dans le chapitre précédent nous avons expliqué ce que sont les objets. Ceux-ci étant maintenant bien compris, il est facile d'expliquer ce que sont les classes.

Dans une première partie, nous présentons d'abord la définition formelle des classes en Objective Caml.

Dans une seconde partie, nous insistons d'abord sur trois points essentiels : avant tout, il faut distinguer la notion de sous-classe de celle de sous-type ; nous montrons qu'une classe se comporte comme une collection de pré-méthodes, i.e. de fonctions dans lesquelles `self` n'est pas lié, et l'état interne n'est pas caché ; enfin, l'héritage simple ou multiple, peut être vu comme la concaténation de ces pré-méthodes.

4.1 Les classes en Objective Caml

Dans cette partie nous définissons formellement les classes du langage Objective Caml, leur syntaxe, leur typage, puis nous donnons leur sémantique. Comme son titre l'indique, il ne suppose pas de connaissance des langages à objets.

4.1.1 Syntaxe des classes

Le corps des classes est une séquence b de petites définitions e. La syntaxe des classes est donnée ci-dessous

$a ::= \ldots \mid \texttt{class}\ (x)\ b \mid \texttt{new}\ a \mid s\#m$	Expressions
$z ::= x \mid c$	
$b ::= \emptyset \mid e\ b$	Class body
$e ::= \texttt{inherit}\ a\ a\ \texttt{as}\ x \mid \texttt{let}\ x = a \mid \texttt{val}\ x = a \mid \texttt{method}\ m = a$	

Chaque définition à pour effet d'ajouter une ou plusieurs liaisons de variables d'instance et de méthodes. Les définitions les plus à droite peuvent cacher des définitions antérieures.

Les liaisons `let` permettent de partager des valeurs ou des calculs, en particulier pour l'initialisation des variables d'instance en présence d'effets de bord. Techniquement, elles facilitent la définition de la sémantique des classes.

4.1.2 Typage

Les schémas de types sont étendus avec des types de classe. Le type d'une classe, est composé du type τ du paramètre d'initialisation (pour simplifier on suppose que les classes ne sont pas currifiées), du type généralement noté τ_y de `self`, des types $\tau^{\mathcal{X}}$ de ses variables d'instance et des types $\tau^{\mathcal{M}}$ de ses méthodes.

$$\boxed{\gamma ::= \texttt{class}\ (\tau) : \tau\ \texttt{val}\ \tau^{\mathcal{X}}\ \texttt{method}\ \tau^{\mathcal{M}}}$$

Pour simplifier, nous omettons la construction de dupplication ainsi que l'envoi de message à une classe parente. Nous considérons également `self` comme un mot clé. Pendant le typage d'une classe, les jugements de typage sont de la forme $A, B \vdash b : \varphi$ où φ est une séquence de liaisons de la forme `val` $x : \tau$ ou `method` $m : \tau$. L'environnement A est utilisé pour typer les variables d'instance et l'héritage, alors que l'environnement B est utilisé pour typer les méthodes. L'environnement A représente l'environnement pendant l'initialisation de l'objet, et B l'environnement pendant l'invocation d'une méthode.

Les règles de typage sont données dans la figure 4.1. Rappelons que $\oplus$ étend une séquence avec priorité à dorite, alors que $+$ impose que les deux séquences soient compatibles (même valeur sur leur domaines communs). Les classes sont typées par ajout successif de petites définitions (classes hérités, variables d'instance ou méthodes) avec priorité à droite.

Quand une méthode est redéfinie, son type doit être le même que celui de l'ancienne méthode, ce qui est exprimé par l'opérateur $+$ entre les types des méthodes. Bien sûr, une sous-classe peut utiliser des versions spécialisées de ses classes parentes en instantiant certaines variables.

La règle Inherits montre que le type de `self` dans une classe est le même que celui de la classe héritée. Ce qui garantit que les méthodes qui retournent `self` sont typées correctement, mais ne restreint pas la puissance, car le type de la classe héritée est une instance du type de la classe parente.

```
let copy = Oo.copy;;

class copy () as s = method copy = copy s end;;
```

Dans cette classse, `'a` est lié au type de `self`, c'est-à-dire que les objets de cette classe ont le type $\langle$`copy : 'a`$\rangle$ `as 'a` (avec la même variable `'a`). La classe `copy` peut maintenant être héritée. La méthode `copy` a bien le type attendu.

$$\textsc{(Basic)}\quad \frac{}{A, B \vdash \emptyset : \texttt{val}\ \emptyset\ \texttt{method}\ \emptyset}$$

$$\textsc{(Inherit)}\quad \frac{B \vdash \texttt{self} : \tau_y \qquad A \vdash c : \texttt{class}\ (\tau) : \tau_y\ \varphi_1 \qquad A \vdash a : \tau \qquad A, B \oplus \varphi\restriction \texttt{val} \vdash b : \ \varphi}{A, B \vdash \texttt{inherit}\ c\ a\ \texttt{as}\ s\ b : \varphi + \varphi_1}$$

$$\textsc{(Method)}\quad \frac{B \vdash \texttt{self} : \tau_y \qquad B \vdash a : \tau \qquad A, B \vdash b : \varphi}{A, B \vdash \texttt{method}\ m = a\ b : \varphi + \texttt{method}\ m : \tau}$$

$$\textsc{(Val)}\quad \frac{A \vdash a : \tau \qquad A, B \oplus (x : \tau) \vdash b : \varphi}{A, B \vdash \texttt{val}\ x = a\ b : \varphi + \texttt{val}\ x : \tau}$$

$$\textsc{(Let)}\quad \frac{A \vdash a : \tau \qquad A \oplus (x : \tau), B \vdash b : \varphi}{A, B \vdash \texttt{let}\ x = a\ b : \varphi}$$

$$\textsc{(Class)}\quad \frac{A \oplus (x : \tau), A \oplus (\texttt{self} : \tau_y) \vdash b : \varphi \qquad \tau_y = \langle \varphi\restriction \texttt{method}; \tau'' \rangle \qquad A \oplus (c : \texttt{Gen}(A, \texttt{class}\ (\tau) : \tau_y\ \varphi)) \vdash d : \tau'}{A \vdash \texttt{let}\ c = \texttt{class}\ (x)\ b\ \texttt{in}\ d : \tau'}$$

$$\textsc{(Rec)}\quad \frac{A \oplus (\texttt{new}\ c : \tau_0) \vdash \texttt{let}\ c = \texttt{class}\ (x)\ b\ \texttt{in}\ ((\texttt{new}\ c : \tau_0); d) : \tau'}{A \vdash \texttt{let}\ c = \texttt{class}\ (x)\ b\ \texttt{in}\ d : \tau'}$$

$$\textsc{(New)}\quad \frac{A \vdash c : \texttt{class}\ (\tau_0) : \langle \varphi\restriction \texttt{method} \rangle \varphi}{A \vdash \texttt{new}\ c : \tau_0 \to \langle \tau^{\mathcal{M}} \rangle}$$

Figure 4.1 : Règles de typage des classes

```
class point x as self =
  val x = x
  method getx = x
  method print = print_int self#getx
end;;

class copy_point x = inherit copy () inherit point x end;;
```

Pendant l'héritage, les parents sont ordonnés, ce qui évite toute ambiguïté : la définition finale de la méthode est celle qui est héritée par le parent qui apparaît le plus tard dans la liste.

La règle Rec permet à l'expression new c d'apparaître dans la définition de la classe c. Autrement la règle New est utilisée. La généralisation des types de classe est similaire à celle des autres types.

La prise en compte de la dupplication et de la liaison des méthodes définies dans des classes parentes ne posent pas de problème réel, mais elle complique l'écriture des règles de typage. Dans le typage des corps des classes, il faut notament séparé le contexte B en deux parties B et C. Les variables d'instances sont ajoutée dans C. Les méthodes définies dans les classes parentes sont ajoutées dans B. Les méthodes sont alors typées dans le contexte $B \oplus C \oplus (\{\langle C \rangle\}^* : \tau_y)$.

$$\text{(Inherit)}\quad \frac{B \vdash \mathtt{self} : \tau_y \qquad A \vdash c : \mathtt{class}\ (\tau) : \tau_y\ \varphi_1 \qquad A \vdash a : \tau \qquad A, B \oplus s\#\tau_1^{\mathcal{M}}, C \oplus \tau_1^{\mathcal{X}} \vdash b : \varphi}{A, B, C \vdash \mathtt{inherit}\ c\ a\ \mathtt{as}\ s\ b : \varphi + \varphi_1}$$

$$\text{(Val)}\quad \frac{A \vdash a : \tau \qquad A, B, C \oplus (x : \tau) \vdash b : \varphi}{A, B, C \vdash \mathtt{val}\ x = a\ b : \varphi + \mathtt{val}\ x : \tau}$$

$$\text{(Method)}\quad \frac{B \vdash \mathtt{self} : \tau_y \qquad B \oplus C \oplus (\{\langle C \rangle\}^* : \tau_y) \vdash a : \tau \qquad A, B, C \vdash b : \varphi}{A, B, C \vdash \mathtt{method}\ m = a\ b : \varphi + \mathtt{method}\ m : \tau}$$

Les autres règles sont modifiées de façon cohérente.

4.1.3 Sémantique

L'évaluation des classes consiste simplement à résoudre l'héritage. Pour cela, il faut évaluer la concaténation des différentes composantes (avec priorité à droite) dans les définitions de classes. Il y a trois difficultés à surmonter : impérative :

- Il faut réordonner les champs (et en éliminer certains) car les objets exigent que les variables d'instance apparaissent en premier. Il faut éviter d'inverser l'ordre d'évaluation.
- Il faut éviter de dupliquer le calcul d'initialisation des variables d'instance.
- Il faut lier "super" méthodes définie dans les classes parentes.

Ces difficultés sont facilement résolues grâce à la liaison dans les définitions de classe.

La création d'un objet à partir d'une classe, consiste simplement à évaluer les liaisons, ce qui initialise les variables d'instance, puis à fabriquer un objet avec ces variables d'instance et les méthodes de la classe.

La définition formelle de l'évaluation est donnée dans la figure 4.2.

4.1.4 Oublis

Les variables d'instance mutables ne posent pas de difficulté particulière. Elle sont typées comme les variables non mutables. La modification en place des objets est typée par ajout de primitives, de façon similaire à la modification en style applicatif. La sémantique peut être décrite dans le formalisme décrivant les références.

La sécurité des constructions impératives est garantie par la restriction que seule les valeurs peuvent être polymorphes. En particulier, même si les classes sont paramétriques, les

Contextes d'évaluation

$$\begin{aligned} E ::= & \ \ldots \mid \texttt{let}\ z = E\ \texttt{in}\ d \\ & \mid \texttt{class}\ (x)\ E \mid w; E \end{aligned}$$

Réduction des définitions de classe

$$\begin{array}{rcll}
\texttt{val}\ x = a\ w & \longrightarrow & \texttt{let}\ x = a\ \ \texttt{val}\ x = x\ w & (a \notin \mathcal{X}) \\
\texttt{val}\ x' = x''\ \ \texttt{let}\ x = v\ w & \longrightarrow & \texttt{let}\ x = v\ \ \texttt{val}\ x' = x''\ w & (x' \neq x) \\
\texttt{method}\ m = a\ \ \texttt{let}\ x = v\ w & \longrightarrow & \texttt{let}\ x = v\ \ \texttt{method}\ m = a\ w & \\
\multicolumn{4}{l}{\texttt{inherit}\ (\texttt{class}\ (x)\ \texttt{let}\ a_1^{\mathcal{X}}\ \texttt{val}\ a_2^{\mathcal{X}}\ \texttt{method}\ a^{\mathcal{M}})\ a\ \texttt{as}\ s\ w} \\
& \longrightarrow & \texttt{let}\ x = a\ \texttt{let}\ a_1^{\mathcal{X}}\ \texttt{val}\ a_2^{\mathcal{X}} & \\
& & \texttt{method}\ a^{\mathcal{M}}\ \ w[a^{\mathcal{M}}(m)/s\#m]^{m \in dom\,(a^{\mathcal{M}})} & \\
\multicolumn{4}{l}{\texttt{new}\ (\texttt{class}\ (x)\ \texttt{let}\ a_1^{\mathcal{X}}\ \texttt{val}\ x_2^{\mathcal{X}}\ \texttt{method}\ a^{\mathcal{M}}} \\
& \longrightarrow & \texttt{fun}\ (x)\ \texttt{let}\ a^{\mathcal{X}}\ \texttt{in}\ \langle\ \texttt{val}\ x_2^{\mathcal{X}}\ \texttt{method}\ a^{\mathcal{M}} \rangle &
\end{array}$$

Figure 4.2 : Sémantique des classes

objets ne sont jamais polymorphes. En effet, la création d'une classe n'est pas une valeur ; elle est donc monomorphe. D'autre part, les objets produits par duplication ont le même type que leur original.

4.2 L'héritage n'est pas du sous-typage

Avant d'étudier plus en détail le mécanisme d'héritage, il est important de se convaincre que les deux concepts n'ont rien à voir.

L'héritage est la possibilité de construire des classes à partir d'autres classes en réutilisant globalement toutes leurs méthodes. C'est donc avant tout une propriété syntaxique. Le sous-typage est la possibilité pour un objet d'un certain type d'être considéré et donc utilisé comme un objet d'un autre type. C'est une propriété sémantique.

Nous avons montré dans la première partie du chapitre précédent que les points colorés construits à partir de points par héritage ne peuvent pas être considérés comme des points, si les points comportent une méthode testant l'égalité de deux points. Autrement dit, le type des points colorés n'est pas un sous-type des points. Cet exemple montre que l'héritage n'entraîne pas le sous-typage des types des objets des classes considérés.

Inversement, si les points colorés (sans égalité) sont aussi des points, il s'agit d'une propriété des objets qui est indépendante de la façon dont les points colorés ont été créés. On peut, bien entendu redéfinir sans hérités des points :

```
class point_coloré x c =
  val mutable x = (x:int)
```

```
    method getx = x
    val c = (c : string)
    method getc = c
  end;;

  let point_coloré = new point_coloré 1 "blanc-cassé";;
  (point_coloré :> point);;
```

On peut aussi coercer le type d'un objet vers un type qui ne correspond pas à une définition de classe.

```
  type pauvre_point = < >;;
  (point_coloré :> pauvre_point);;
```

On pourrait aussi considérer deux implémentations indépendante des points dans le plan, fonctionnant en coordoonées cartésiennes pour l'une et en coordonnées polaires pour l'autre. Les deux classes devront fournir la même interface, c'est à dire que les types des objets de ces classes sont identiques donc a fortiori dans une relation de sous-typage. Pourtant les classes ne sont pas dans une relation de sous-classage. Identifier sous-classage et sous-typage interdirait cette abstraction de la représentation d'une classe. Ou forcerait à définir artificiellement une classe abstraite ayant la même interface, mais aucune implémentation dont hériteraient à la fois les deux implémentations des points.

Retenons que :

Le sous-typage n'a rien à voir avec l'héritage

La notion d'héritage est plus proche d'une notion de *filtrage* sur les types associés. La déclaration de la classe `point_coloré` ci-dessus produit la déclaration automatique de type équivalente à :

```
    type 'r #point_coloré = <getx : int; getc : string; 'r>
```

En fait la variable de rangée est un paramètre implicite et elle est remplacée par une ellipse dans le type : on écrit simplement `#point_coloré` pour signifier `<getx : int; getc : string; ..>`. La #-abréviation d'une classe filtre tous le type des objets de tous les sous-classes. Cette propriété est immédiate : elle résulte du fait qu'une sous-classe ne peut pas caché une méthode définie dans une classe parente, ni changer son type, mais seulement le préciser.

Il n'y a pas de règles sans exceptions. Nous devons donc parler des nombreux langages qui pour simplifier identifient la notions d'héritage et de sous-typage. Ils sont de deux sortes.

- Certains comme Eiffel, O2 font cette identification au dépend de la sécurité. Ils permettent de considérer des points colorés avec égalité comme des points avec égalité. Ce qui bien entendu conduit à des erreurs à l'exécution. En général, les versions plus récentes de ces langages insèrent maintenant des tests dynamiques lorsque ces coercions dangereuses sont effectuées afin de rattraper les erreurs plus proprement.

- D'autres langages comme Java contourne le problème en affaiblissant les règles de typage. Il suffit d'interdire le type de `self` d'apparaître dans le type d'une méthode. Par exemple la méthode d'égalité aura le type `point` $\rightarrow$ `point` dans la classe `point` et conservera ce type lorsqu'elle sera héritée dans la classe des points colorés.

  Cet affaiblissement du système de typage force l'utilisateur à effectuer des coercions explicites qui peuvent échouées. On n'est donc pas très loin de l'approche ci-dessus, sinon que l'échec provient d'une coercion mise par l'utilisateur. Elle est donc prévisible.

  Le traitement de C++ est très similaire. La différence est qu'en C++ l'utilisateur est encouragé à effectuer une coercion sans test, c'est-à-dire forcer le type à la main avec un `cast`, ce qui peut avoir des conséquences plus graves.

4.3 Pré-méthodes

Le but de cette section est de montrer que les classes ne sont au fond que de l'assemblage des méthodes abstraites par rapport à l'objet `self` dans lequel elle seront placées.

Que ce soit dans la simulation des classes par des enregistrements extensibles dans le langage Caml, ou bien le typage des classes dans Objective Caml, les méthodes sont définies dans les classes comme des fonctions, abstraites par rapport aux variables d'instance et par rapport à `self`. Nous les appelons *pré-méthodes*.

Les pré-méthodes doivent pouvoir être utilisées comme méthodes des objets de la classe où elles sont définies (si cette classe n'est pas virtuelle), mais aussi comme méthodes des objets de ses sous-classes.

Rappelons la règle de typage de la méthode m dans une classe :

$$\textsc{(Method)}\quad \frac{B \vdash \texttt{self} : \tau_y \qquad B \oplus C \vdash a : \tau \qquad A, B, C \vdash b : \texttt{val}\ \tau^{\mathcal{X}}\ \texttt{method}\ \tau^{\mathcal{M}}}{A, B, C \vdash \texttt{method}\ m = a\ b : \texttt{val}\ \tau^{\mathcal{X}}\ \texttt{method}\ \tau^{\mathcal{M}} + m : \tau}$$

On pourrait dire que l'on type la pré-méthode $\texttt{fun}\ (\texttt{self} : \tau_y)\ \texttt{fun}\ (C)\ a$

$$\textsc{(Method)}\quad \frac{B \vdash \texttt{self} : \tau_y \qquad B \vdash \texttt{fun}\ (\texttt{self} : \tau_y)\ \texttt{fun}\ (C)\ a : \texttt{self} \rightarrow C \rightarrow \tau \qquad A, B, C \vdash b : \texttt{val}\ \tau^{\mathcal{X}}\ \texttt{method}\ \tau^{\mathcal{M}}}{A, B, C \vdash \texttt{method}\ m = a\ b : \texttt{val}\ \tau^{\mathcal{X}}\ \texttt{method}\ \tau^{\mathcal{M}} + m : \tau}$$

4.4 Liaison tardive et typage

La liaison tardive implique que dans une classe la valeur liée à `self` peut-être tout objet d'une sous-classe. En particulier, les méthodes qui dépendent de `self` doivent fonctionner aussi bien dans les classes que dans les sous-classes.

Cela est obtenu en donnant à `self` un type ouvert dans la définition d'une classe. Les pré-méthodes sont donc paramétrique par rapport à la valeur réelle de `self`. Le type dans une pré-méthode contient toutes les contraintes qui lui sont imposées dans cette pré-méthode.

Le type de `self` dans est unifié dans toutes les pré-méthodes d'une classe. Il contient alors toutes les contraintes imposées à `self` dans cette classe.

Une sous-classe ne relâchera jamais des contraintes imposés à `self` dans une classe parente, car cela invaliderait le typage. En revanche, elle peuvent le contraindre d'avantage.

Une pré-méthode définie dans une classe parente et hérité est spécialisée par instantiation de type pour satisfaire les nouvelles contraintes imposées à `self`.

C'est donc le mécanisme général du polymorphisme de ML, et le polymorphisme des variables de rangées qui assure la sécurité du typage de la liaison tardive.

Le raisonnement ci-dessus reste valide pour les pré-méthodes binaires.

> La liaison tardive est permise et sécurisée par le polymorphsime de ML sur les variables de rangée des types objets

> Dans une classe, le type de `self` est normalement ouvert

Il peut arrivé que le type de `self` soit fermé.

- Volontairement : Cela interdit d'ajouter ultérieurement des méthodes par héritage. Mais n'a guère d'intérêt.
- Involontairement : parce que `self` est mélangé avec un objet déjà créé.

4.5 Création des objets

Une classe est une collection de fonctions pour d'initialisation des variables d'instance, et une collection de pré-méthodes.

La création d'un objet à partir d'une classe revient à grouper

- créer l'état interne,
- créer l'enregistrement des méthodes à partir des pré-méthodes, en donnant à `self` sa vraie valeur.

À la création des objets à partir des classes, le type objet est fermé. Cela revient à dire que les seules méthodes de l'objet sont celles couramment définies. S'il reste des méthodes virtuelles (des méthodes requises mais non définies, la classe est virtuelle).

4.6 Héritage

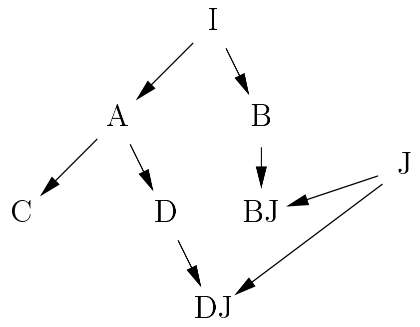
4.6.1 Héritage simple

L'héritage est simplement un assemblage des composantes de classes. variables d'instance et pré-méthodes, tout en gardant un contrôle précis sur la portée des variables. L'héritage simple est un ajout de composantes une par une. Il repose sur la concaténation des enregistrements. L'héritage multiple repose sur la concaténation des enregistrements. On peut ajouter les composantes en blocs.

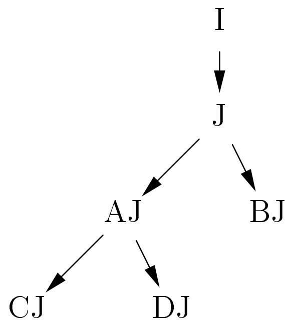
4.6.2 Héritage multiple

Rappelons que l'héritage multiple consiste à construire une classe à partir de plusieurs classes parentes. La construction d'héritage en Objective Caml est naturellement multiple. L'héritage multiple permet surtout d'ajouter à une classe existante les fonctionnalité d'une autre classe a posteriori. Sans l'héritage multiple, il faudrait ou bien dupliquer certaines opérations ou bien les insérer à la racine commune aux branches qui doivent être étendues. Par exemple, on peut construire la hiérarchie suivante (figure de gauche) :

Héritage multiple



Héritage simple



L'intérêt est que seule les classes B et D peuvent hériter de la classe J. Pour conserver le partage (écrire une seule fois les composantes de J) avec seulement de l'héritage simple, il est nécessaire d'introduire J très haut dans la hiérarchie (figure de droite). Il en résulte que la classe J est hérité dans toute la hiérarchie, ce qui peut être dangereux, ou créer des conflits.

De façon plus générale, l'héritage multiple permet d'organiser les classes de façon plus proche de leur dépendances réelles. Ceci accroît la sécurité bien entendu, mais renforce également tous les aspects de la modularité. Dans l'exemple ci-dessus, l'héritage multiple permet d'exprimer que BJ et DJ sont les seules classes à dépendre de J, alors que la version avec héritage simple force une dépendance artificielle de les classes en dessous de la racine commune à B et D.

4.7 Exemple

La difficulté dans la programmation avec des objets est souvent l'organisation des classes. Quelle sorte de modularité recherche-t-on ?

Pour montrer que ce problème n'est pas toujours simple, considérons l'exemple suivant. Imaginons que l'on manipule des figures géométriques. En commençant par des points, des segments, etc. et que l'on veuille introduire les cercles.

Qu'est-ce qu'un cercle ? Il semble naturelle de représenter un cercle par un point — son centre, et un rayon :

```
class circle p r =
  val c = (p : point)
  val r = (r : int)
  method center = c
  method radius = r
  method move dv = p#move
end;;
```

On s'aperçoit qu'un objet cercle comporte un certain nombre de méthodes telles que `move` sont identiques à celles des points ; de plus l'objet fait simplement suivre ces méthodes vers son centre. La définitions alternative suivante semble donc aussi raisonable :

```
class circle x r =
  inherits point x
  method center = (x :> point)
  val r = r
  method radius = r
end;;
```

Quelles sont les différences entre les deux définitions ? L'une est-elle préférable ? Il n'est pas évident de répondre à ces questions. D'abord, l'interface d'un objet cercle sera dans les deux cas la même ; il va de soit que l'on attend aussi un comportement identique.

La seconde définition évite de redéfinir les méthodes des points dans les cercles. D'autre l'ajout d'une méthode aux points est immédiatement répercutée vers les cercles. Enfin elle permet de considérer un cercle comme un point, et donc par exemple de définir un segment à partir de deux cercles comme le segment reliant leur centre. En bref, cette deuxième solution semble préférable.

Pourtant, elle ne permet pas de créer un point à partir d'un point déjà créé. Imaginons qu'un point soit partager par plusieurs objets géométriques et que l'on veuille maintenant créer un cercle autour de ce point. Avec la seconde définition, il faudra créer un nouveau point-cercle, c'est-à-dire que le cercle ne pourra pas partager le point existant. Ceci est embêtant si ce point est bougé physiquement, le cercle restera là où il était.

Il n'y a donc pas de meilleure solution. Il s'agit simplement de bine comprendre ce que l'on veut.

4.8 Conclusion

Il faut retenir que les classes sont essentiellement un ensemble de déclaration toplevel permettant de construire des objets ou bien d'être ultérieurement étendues avec d'autres déclarations.

Dans cette approche les classes sont essentiellement une collection de pré-méthodes, c'est-à-dire des méthodes abstraite par rapport à l'objet `self` dans lequel elle apparaîtront ainsi que par rapport aux variables d'instance de cet objet.

Les règles de typage des classes dans Objective Caml, font également apparaître une méticuleuse gestion de la visibilité des différentes liaisons. C'est aussi un des aspects important de la notion de classe.

Il faut retenir que la vision de `self` dans une classe est une vision ouverte, puisque nous ne connaissons alors qu'un sous-ensemble des méthodes de `self`. Dans Objective Caml cette ouverture est exprimé par un type objet ouvert, donc comportant une variable de rangée polymorphe.

Enfin, il est essentiel de distinguer la notion de sous-type et celle de sous-classe.

Annexe A

Correction des exercices

Exercice 2, page 24

L'objet `p` lui-même.

Exercice 3, page 25

```
class copy () as moi_m^^eame =
  method copy = Oo.copy moi_m^^eame
end;;
class copy (unit) : 'a = method copy : 'a end
```

Cet exemple n'est guère intéressant car la fonction copy peut toujours être utilisée à la place d'un objet. Par contre si l'objet a un état interne, une copie profonde devra répercuter la copie sur les variables d'instances.

Exercice 4, page 25

On utilise la construction $\{\langle\_\rangle\}$. Par exemple, on peut réécrire la classe précédente :

```
class copy () =
  method copy = {⟨  ⟩}
end;;
class copy (unit) : 'a = method copy : 'a end
```

Si dans une sous-classe, une variable d'instance est un objet qui doit également être dupliqué, on pourra re-définir la méthode `copy` :

```
class deep_copy x as self =
  inherits copy () as super
  val x = x
  method local_copy = {⟨ x = x#copy ⟩}
  method copy = (super#copy)#local_copy
```

The use `local_copy` permits inheritance of the `super#copy` method, that will do a possibly deep copy in the superclass, and the later application of the local copy. The is not quite efficient since the objet will be dupplicated as many times as the number of parent classes. In this case the imperative style may be preferable. See the one after next exercise.

Another practical simple situtation is illustrated by vectors in the state.

```
class vector f n =
  val m =
     let v = Vect.create (f k) p in
     for i = 0 to p−1 do v.(i) ← f i done; v
  method deep_copy = {⟨ m = Vect.copy m ⟩}
end;;
```

Pour en savoir plus

Il y a de nombreux livres sur la programmation à objets, mais hélas encore peu de bons livres. La liste ci-dessous ne se veut ni exhaustive, ni sélectionnant les meilleures ouvrages, mais quelques pointeurs illustrant des idées ou des langages précis.

Des documents directement liéés au cours

Le chapitre **??** est assez bien couvert par l'article [Rém94], toutefois plus technique. La présentation d'Objective Caml est également disponible en ligne à l'adresse `http://pauillac.inria.fr:caml`. La formalisation d'Objective Caml, en partie présentée dans les chapitres **??** et **??**, est décrite en détail dans [RV97]. La référence pour le sous-typage des types récursifs est [AC93], mais l'algorithme simple et efficace est décrit dans [KP93].

Des pointeurs pour un approfondissement plus théorique

La différence en héritage et sous-typage est le mieux décrit dans [CHC93].

Pour comprendres les objets dans un calcul plus simple (mais explicitement typés) on pourra regarder

- Un codage dans le système $F^{\Omega}_{<:}$. [PT94]
- Une théorie des objects primitifs [AC94, AC95].

Pour une très bonne comparaison entre les différentes façons de représenter les objets [BCP96].

La meilleure référence sur la théorie et le typage des objets est certainement un livre à paraître en été 1996 par M. Abadi et L. Cardelli.

Documents plus pratiques

Les autres documents décrivent des cours sur les objets ou des langages à objets.

Références

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *Transactions on Programming Languages and Systems. ACM*, 15(4) :575–631, 1993.

[AC94] Martín Abadi and Luca Cardelli. A theory of primitive objects : Untyped and first-order systems. In *Theoretical Aspects of Computer Software*, pages 296–320. Springer-Verlag, April 1994.

[AC95] Martín Abadi and Luca Cardelli. A theory of primitive objects : Second-order systems. *Science of Computer Programming*, 25(2-3) :81–116, December 1995. Preliminary version appeared in D. Sanella, editor, Proceedings of European Symposium on Programming, pages 1-24. Springer-Verlag, April 1994.

[BCP96] Kim Bruce, Luca Cardelli, and Benjamin Pierce. Comparing objects encodings. Presented at the FOOL'3 workshop. Available electronically in informal workshop proceedings, July 1996.

[CHC93] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1993.

[KP93] Dexter Kozen and Jens Palsberg. Efficient recursive subtyping. In *Proc. 20th symp. Principles of Programming Languages*, pages 419–428. ACM press, 1993.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2) :207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[Rém93] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1993.

[Rém94] Didier Rémy. Programming objects with ML-ART : An extension to ml with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 321–346. Springer-Verlag, April 1994.

[RV97] Didier Rémy and Jérôme Vouillon. Objective ML : A simple object-oriented extension to ML. In *Proceedings of the 24th ACM Conference on Principles of Programming Languages*, January 1997. To appear. Also presented at the FOOL'3 workshop, available electronically in informal workshop proceedings.