10.4 Variables de classes

Nous avons vu les *variables d'instances* des classes : chaque objet qui est instance d'une telle classe possède une instance de cette variable.

Il y a normalement dans les langages orientés objet aussi un mécanisme pour définir des *variables de classe*. Ce sont des variables qui existent une fois par *classe* et qui sont donc partagées entre tous les objets d'une telle classe.

En OCaml : il n'existe pas de construction syntaxique pour les variables de classe mais on peut les simuler par des variables qui sont globales à l'objet.

Voir la démo `globalvars`)

Cette solution a le désavantage que la variable est complètement globale et donc visible pour tout le monde.

Pour avoir une *vraie* variable de classe qui n'est visible que pour les objets de cette classe il faut utiliser une généralisation de la construction des classes :

```
class <nom> = exp end
```

où `exp` est une expression dont le type est une fonction qui renvoie un objet. En particulier,

```
class c =
  fun x1 x2 ... xn ->
  object
    ...
```

```
  end
```

peut être abrégé comme

```
class c x1 x2 ... xn = object
  ...
end
```

Mais la construction ci-dessus est plus générale

Voir la démo `classvars`.

10.5 Classes et méthodes virtuelles

Les classes *virtuelles* (aussi appelées des classes *abstraites*) ne sont pas prévues pour être instanciées, mais ne servent que pour définir une étape dans une hiérarchie de héritage.

En OCaml on peut définir des classes virtuelles à l'aide de la construction

```
class virtual <nom> = ....
```

On ne peut pas effectuer un `new` pour une classe virtuelle.

Une méthode *virtuelle* n'est que déclarée avec son type. Une classe contenant une méthode virtuelle doit être définie comme une classe virtuelle.

```
method virtual <nom> : <type>
```

Voir la démo `virtual`. L'intérêt des classes virtuelles :

- on peut déclarer une classe avec une méthode dont la définition est spécifique à des classes dérivées
- on peut « coercer » des objets des classes dérivées vers la classe ancêtre (voir plus tard).
- on peut spécifier le type d'une méthode (voir l'exercice suivant)

Exercice: On désire définir une classe virtuelle *intlist* des listes d'entier comme

```
class virtual intliste = object
  method virtual hd : unit -> int
  method virtual tl : unit -> intliste
  method virtual sum : unit -> int
  method virtual append : intliste -> intliste
  method virtual to_string : unit -> string
end
```

et puis la spécialiser par deux classes concrètes *cons* et *nil*. Après avoir défini ces deux classes, on pourra par exemple faire

```
let x = new cons 1 (new cons 2 (new cons 3 (new nil)));;
x#sum ();;              (* 6 *)
x#to_string ();;        (* "1, 2, 3, NIL" *)
```

```
let y = x#append x;;
y#to_string ();;      (* "1, 2, 3, 1, 2, 3, NIL" *)
```

Donner les définitions des deux classes *cons* et *nil*. Faites attention au fait que (jusqu‘ici) les types des méthodes ne peuvent pas être polymorphes.

10.6 Initialisateurs

On peut demander l'évaluation d'une expression lors de la création d'un objet : `initialiser` $expr$.

Voir la démo `initializer`

10.7 Restrictions de visibilité

Il y a dans les langages orienté objets un nombre de constructions qui permettent de restreindre la visibilité des méthodes.

Par exemple en C++, une méthode peut être

publique Visible partout (défaut)

privée Seulement visible par les méthodes de la classe même, et pas par les classe dérivées.

protégée Visible pour les méthodes de la classe et ses classes dérivées.

De plus, on peut contourner ces mécanismes de protection par des déclarations des classes *amies* : Si une classe $C1$ déclare une classe $C2$ son amie alors toutes les méthodes de $C1$ sont visibles par les méthodes de la classe $C2$.

En OCaml : Il y a des méthodes *privées* qui correspondent aux méthodes *protégées* de C++. Voir la démo `private.ml` :

```
class point (x_init,y_init) =
object
  val mutable x = x_init
  val mutable y = y_init
  method moveto (a,b) = x <- a ; y <- b
  method rmoveto (dx,dy) = x <- x + dx ; y <- y + dy
  method to_string () =
    "(" ^ (string_of_int x) ^ ", " ^ (string_of_int y) ^")"
end

class point_mem (x0,y0) =
object(self)
  inherit point (x0,y0) as super
  val mutable old_x = x0
  val mutable old_y = y0
```

```
  method private mem_pos () = old_x <- x ; old_y <- y
  method moveto (x1, y1) = self#mem_pos () ; super#moveto (x
  method rmoveto (x1, y1) = self#mem_pos () ; super#rmoveto
  method undo () = x <- old_x; y <- old_y
end

let p = new point_mem (0,0);;
p#moveto(42,42);;
p#rmoveto(17,17);;
p#to_string ();;
p#undo ();;
p#to_string ();;


(* la méthode mem_pos est privée, elle ne peut pas être invo
   l'extérieur
*)
```

```
p#mem_pos ();; (* erreur *)

p = new point (42,42);; (* erreur de typage *)

(* les methodes privées sont heritées par les classes dérivé

class colored_point_mem (x0,y0) color =
object(self)
  inherit point_mem (x0,y0) as super
  val c = color
  method to_string () = super#to_string () ^ " de couleur " ^
end

let p = new colored_point_mem (0,0) "pink";;
p#moveto(42,42);;
p#rmoveto(17,17);;
p#to_string ();;
p#undo ();;
```

```
p#to_string ();;
```

Il n’y a en OCaml pas de construction syntaxique pour les classes amies.

11 Typage des classes en OCaml

En OCaml, les définitions de types ne peuvent pas conténir des variables de types libres :

```
type t = 'a list
```

déclenche une erreur « Unbound type parameter 'a » .

Puisque la définition d'une classe entraîne une définition de type, les types des méthodes et variables ne doivent pas contenir des variables de types libres. Il y a deux solutions possibles quand le type inféré d'une classe n'est pas clos :

1. Ajouter des contraintes de types
2. Définir une classe polymorphe

11.1 Contraintes de types

Avec des contraintes de type de la forme `(expr : type)` on peut contraindre les types des arguments des classes et des méthodes, et aussi les types des résultats des méthodes.

11.2 Types ouverts

En Caml « classique » : tout nom de champ est spécifique à un type d'enregistrement, on peut donc dériver d'un terme comme `x.nom` l'information que le type de x est le type *unique* qui contient le champ `nom` (ou déclencher une erreur de typage quand il n'y a pas de tel type).

Avec les objets : Plusieurs classes peuvent conténir la même méthode, on ne peut donc pas dériver d'un appel de méthode `o#meth` la classe de l'objet o.

$\Rightarrow$ On a maintenat besoin d'un système de types plus souple qui peut exprimer qu'un objet a *au moins* telle et telle méthode, avec tel type. En OCaml, ces types sont appelés des *types ouverts*.

Un type

```
< m1: t1 -> s1; m2: t2 -> s2; .. >
```

dénote tous les objets qui ont au moins une méthode `m1` du type $t_1 \rightarrow s_1$, et une méthode `m2` du type $t_2 \rightarrow s_2$. Notez que les `..` sont une partie *syntaxique* de cette expression de type. (voir la démo `open1`).

```
(* Démonstration des types ouverts *)

(* la seule information sur l'argument de la fonction f qu'o
   inférer est que x possède une méthode get_x sans argument
let f o = o#get_x ;;

(* les deux classes des points et des point colorés, comme a

class point (x_init,y_init) =
object
  val mutable x = x_init
  val mutable y = y_init
```

```
  method get_x = x
  method get_y = y
  method moveto (a,b) = x <- a ; y <- b
  method rmoveto (dx,dy) = x <- x + dx ; y <- y + dy
  method to_string () =
    "(" ^ (string_of_int x) ^ ", " ^ (string_of_int y) ^")"
end

class colored_point (x,y) c =
object
  inherit point (x,y) as super
  val mutable c = c
  method get_color = c
  method set_color nc = c <- nc
  method to_string () = super#to_string () ^ " [" ^ c ^ "] "
end

let x = new point (0,1);;
```

```
let y = new colored_point (2,3) "vert";;
f x;;
f y;;
```

Les arguments d'une classe peuvent être d'un type conténant des variables libres : La fonction de création d'instances de cette classe est simplemenmt une fonction polymorphe (voir la démo `open2`).

```
(* Exemple d'une classe où les paramêtres sont d'un type lib

class couple (a,b) =
object
  val p0 = a
  val p1 = b
  method to_string() = p0#to_string() ^ ", " ^ p1#to_string(
  method copy () = new couple (p0,p1)
end
```

```
class newint  i = object
  val c = i
  method to_string () = string_of_int i
end

class newstring s = object
  val c = s
  method to_string () = "[" ^ s ^ "]"
end

let x = new couple (new newint 1, new newstring "bonjour");;
let y = x#copy ();;
y#to_string ();;
```

Par contre, les types des méthodes doivent être fermés. Ajouter des contraintes de types si nécessaire (voir la démo open3).

```
(* une classe virtuelle pour les objets avec méthode « print
```

```
class virtual printable = object(self)
  method print () = print_string (self#to_string ()); print_
  method virtual to_string : unit -> string
end

(* la classe « point » comme d'habitude *)
class point (x_init,y_init) =
object
  inherit printable
  val mutable x = x_init
  val mutable y = y_init
  method get_x = x
  method get_y = y
  method moveto (a,b) = x <- a ; y <- b
  method rmoveto (dx,dy) = x <- x + dx ; y <- y + dy
  method to_string () =
    "(" ^ (string_of_int x) ^ ", " ^ (string_of_int y) ^")"
  method distance () = sqrt (float(x*x + y*y))
```

```
end

(* cela ne marche pas parce que le type de la méthode « m » s
class t p = object
  val x = p
  method m () = p#to_string ()
end

(* première solution : dire que p possède au moins les métho
   printable *)
class t (p : #printable)  = object
  val x = p
  method m () = x#to_string ()
end

(* deuxième solution : contraindre le type du résultat de la
class t p = object
  val x = p
```

```
  method m = (x#to_string () : string)
end

let x = new point (0,1);;
let y = new t x;;
y#m;;
```

Attention : un type ouvert est considéré comme un type qui contient une variable de type libre (qui dénote le « reste » du type). Voir la démo `open4` :

```
class virtual printable = object(self)
  method print () = print_string (self#to_string ()); print_
  method virtual to_string : unit -> string
end

(* cela déclenche une erreur puisque le type de p n'est pas
class t p = object
```

```
  val x = p
  method get () = x
end

(* contraindre p par  un type ouvert n'est pas suffisant  *)
class t (p: #printable) = object
  val x = p
  method get () = x
end

(* avec une contrainte de type fermé ca marche ... *)
class t (p: printable) = object
  val x = p
  method get () = x
end
;;
```

Il y a une seule exception à la règle que le type d'une méthode ne doit pas contenir une variable de type libre : Le type d'une méthode peut contenir une variable de type qui dénote le type de la classe elle même. (voir la démo `selftype`)

```
(* toujours la classe point *)
class point (x_init,y_init) =
object
  val mutable x = x_init
  val mutable y = y_init
  method get_x = x
  method get_y = y
  method moveto (a,b) = x <- a ; y <- b
  method rmoveto (dx,dy) = x <- x + dx ; y <- y + dy
  method to_string () =
    "(" ^ (string_of_int x) ^ ", " ^ (string_of_int y) ^")"
end;;
```

```
(* classe dérivée avec une opération d'égalité *)
class point_eq (x,y) =
object (self : 'a)
  inherit point (x,y)
  method eq (p:'a) = (self#get_x = p#get_x) && (self#get_y =
end ;;

(* class dérivée avec coleur *)
class colored_point_eq (xc,yc) c =
object (self : 'a)
  inherit point_eq (xc,yc) as super
  val c = (c:string)
  method get_c = c
  method eq (pc : 'a) =  (self#get_x = pc#get_x) && (self#ge
                         && (self#get_c = pc#get_c)
end
```

12 Classes paramétriques

On peut définir des classes paramétriques (voir la démo `parameter`)

```
(* erreur : variable de type libre *)
class pair x0 y0 =
object
  val x = x0
  val y = y0
  method fst = x
  method snd = y
end

(* classe paramétrique *)
class ['a,'b] pair (x0:'a) (y0:'b) =
object
  val x = x0
```

```
  val y = y0
  method fst = x
  method snd = y
end

let p = new pair 2 'X';;
p#fst;;
let q = new pair 3.12 true;;
q#snd;;

(* il faut indiquer les paramètres d'une classe quand on hér
   paramétrée  *)

class ['a,'b] acc_pair (x0 : 'a) (y0 : 'b) =
object
  inherit ['a,'b] pair x0 y0
  method get1 z = if x = z then y else raise Not_found
end
```

```
let p = new acc_pair 3 true;;
p#get1 3;;

(* Ici, la classe par_point n'est pas paramétrique *)
class pair_point (p1,p2) =
object
inherit [point,point] pair p1 p2
end

(* démonstration des contraintes de types *)
class virtual printable = object(self)
  method print () = print_string (self#to_string ()); print_
  method virtual to_string : unit -> string
end

class printable_pair (x0 ) (y0 ) =
object
  inherit [printable, printable] acc_pair x0 y0
```

```
  method print () = x#print(); y#print ()
end

(* on ne peut pas utiliser cette classe pour des objets d'un
   que printable *)

class point (x_init,y_init) =
object
  inherit printable
  val mutable x = x_init
  val mutable y = y_init
  method get_x = x
  method get_y = y
  method moveto (a,b) = x <- a ; y <- b
  method rmoveto (dx,dy) = x <- x + dx ; y <- y + dy
  method to_string () =
    "(" ^ (string_of_int x) ^ ", " ^ (string_of_int y) ^")"
end
```

```
let p = new point (0,1);;
let q = new point (2,3);;

let r = new printable_pair p q;;

(* premier essai : « ouvrir » le type de printable *)

class printable_pair (x0 ) (y0 ) =
object
  inherit [ #printable, #printable ] acc_pair x0 y0
  method print () = x#print(); y#print ()
end

(* la bonne solution consiste en l'utilisation d'une contrai
class ['a,'b] printable_pair (x0 ) (y0 ) =
object
  constraint 'a = #printable
  constraint 'b = #printable
```

```
  inherit [’a,’b] acc_pair x0 y0
  method print () = x#print(); y#print ()
end

let r = new printable_pair p q;;
r#print();;
```

13 Objets et sous-typage

13.1 Types ouverts, sous-typage et contraintes

Dans le λ-calcul « classique » : le type principal d'une expression peut être obtenu par unification :

- pour chaque identificateur il y a une variable qui dénote son type
- on obtient d'une expression un système d'équations qui représente toutes les *contraintes* de typage. Par exemple, de l'expression

  $$x + f(x)$$

  on obtient le système

  $$\begin{array}{lcllcl} x & = & Int & & fr & = & Int \\ f & = & fa \to fr & & fa & = & x \end{array}$$

dont la restriction du mgu aux variables de départ $\{x, f\}$ est

$$\begin{aligned} x &= Int \\ f &= Int \to Int \end{aligned}$$

Comment intégrer les types d'objets ?

$\Rightarrow$ Étendre le langage de type par des types d'enregistrements :

$$A ::= \mathcal{T} \mid A \times A \mid A \to A \mid < \mathcal{C} : A, \ldots, \mathcal{C} : A >$$

où $\mathcal{C}$ est une ensemble donné de noms de champs

Comment intégrer les types d'objets ouverts ?

Les types ouverts correspondent à la notion de *sous-typage* !

$\Rightarrow$ Étendre le langage de *contraintes* (qui pour l'instant ne contient que la relation d'égalité) par des *inégalités* : $x \leq y$.

Définition d'une structure (dans le sens de la logique du premier ordre) :

- langage : constantes de $\mathcal{T}$, opérateurs binaires $\times$ et $\rightarrow$, et pour tout ensemble $\{c_1, \ldots, c_n\}$ de noms de champs une opération d'arité n : $< c_1, \ldots, c_n >$. Prédicats binaires : $=$ et $\leq$.
- domaine : dans une première approche les types selon la grammaire au-dessus.
- interprétation des symboles de fonctions : comme dans les structures de *Herbrand*
- interprétation du symbole $=$: égalité
- interprétation du symbole $\leq$: $t \leq s$ est vrai ssi
  - soit t et s sont des enregistrements, et

$$\begin{aligned} t &= < c_1 : t_1, \ldots, c_n : t_n, d_1 : u_1, \ldots, s_m : u_m > \\ s &= < c_1 : s_1, \ldots, c_n : s_n > \end{aligned}$$

avec $t_i \leq s_i$

– soit t et s sont des types produits, et

$$\begin{aligned} t &= t_1 \times t_2 \\ s &= s_1 \times s_2 \end{aligned}$$

avec $t_1 \leq s_1$ et $t_2 \leq s_2$

– soit t et s sont des types fonctionnels, et

$$\begin{aligned} t &= t_1 \to t_2 \\ s &= s_1 \to s_2 \end{aligned}$$

avec $t_1 \geq s_1$ (*contra-variance* !) et $t_2 \leq s_2$

– soit t et s sont des types de base, et $t = s$. (Ici il y a des variantes, on pourrait aussi imaginer des relation de sous-typage non triviales entre types de base, comme *Int* $\leq$ *Real*.)

Le problème d'*unification* devient maintenant un problème de *satisfaisabilité de contraintes* : Étant donnée une conjonction de formules positives atomiques, est-elle satisfaisable dans la structure fixée ?

Exemple 1 : $1 + x\#m(2)$ donne lieu au système de contraintes

$$\begin{aligned} x &\leq < m : t > \\ t &= t_1 \rightarrow t_2 \\ t_1 &= Int \\ t_2 &= Int \end{aligned}$$

Toutes les types d'objets qui contiennent au moins $< m : Int \rightarrow Int >$ en sont des solutions.

Exemple 2 : $1 + x\#m(x)$ donne lieu au système de contraintes

$$\begin{aligned} x &\leq < m : t > \\ t &= t_1 \rightarrow t_2 \\ t_1 &= x \\ t_2 &= Int \end{aligned}$$

Toutes les types t qui satisfont la relation $t \leq < m : t \rightarrow Int >$ en sont des solutions. Mais avec notre définition de types un tel type n'existe pas !

Pourtant, OCaml accepte une telle expression (voir la démo `open5`) :

```
let f = function x ->  1 + x#m(x);;

class moo = object(self:'a)
  method m (x:'a) = 42
```

```
end;;

f (new moo);;
```

$\Rightarrow$ Il faut changer la définition des types (et de la structure qui interprète les contraintes de typage) et permettre aussi des types *infinis* !

13.2 Coercion de type

Si un objet o est d'un type t_1 et t_1 est un sous-type de t_2, alors on peut *coercer* o vers le type t_2. Notion en OCaml :

```
(nom :> type)
```

Voir la démo `coercion` :

```
(* Démonstration de la coercion *)

class virtual printable = object(self)
  method print () = print_string (self#to_string ()); print_
  method virtual to_string : unit -> string
end

class point (x_init,y_init) =
object
  inherit printable
  val mutable x = x_init
  val mutable y = y_init
  method get_x = x
  method get_y = y
  method moveto (a,b) = x <- a ; y <- b
  method rmoveto (dx,dy) = x <- x + dx ; y <- y + dy
```

```
  method to_string  () =
    "(" ^ (string_of_int x) ^ ", " ^ (string_of_int y) ^")"
end

class colored_point (x,y) c =
object
  inherit point (x,y) as super
  val mutable c = c
  method get_color = c
  method set_color nc = c <- nc
  method to_string () = super#to_string () ^ " [" ^ c ^ "] "
end

let x = new point (1,2);;
let y = new colored_point (3,4) "pink";;
let z = [x; y];; (* erreur de typage *)
let z = [ (x :> printable); (y :> printable) ];; (* ca marche
List.iter (function o -> o#print () ) z ;; (* imprimer les ob
```

```
(* La même chose avec coercion vers point *)
let z = [ x; (y :> point) ];; (* ca marche *)
List.iter (function o -> o#print () ) z ;; (* imprimer les o

(* c'est bien la méthode print de la classe colored_point qu
   pour y ! *)
```

13.3 Sous-typage $\neq$ Héritage

On trouve parfois dans la littérature la fausse idée que l'héritage est une forme de sous-typage. On vérité, les deux concepts sont indépendants !

Exemple d’un sous-typage sans héritage

Voir la démo `soustypes1` :

```
class c1 = object
  method m1 = "coocoo"
  method m2 = "toto"
end

class c2 = object
  method m1 = "hello"
end

(* il n’y a aucune relation d’héritage entre les classes c1 c

let x1 = new c1;;
let x2 = new c2;;
```

```
(* pourtant, le type de c1 est un sous-type de du type de c2
let x3 = (x1 :> c2);;
x3#m1;;
```

Exemple d’un héritage sans sous-typage

Voir la démo `soustypes2` :

```
class point ( (x0:int) ,(y0:int) ) =
object(self: ’a)
  val x = x0
  val y = y0
  method getx = x
  method gety = y
  method equal (p:’a) = (self#getx = p#getx) && (self#gety =
```

```
end

class colored_point ( (x0:int) ,(y0:int) ) (c0:string) =
object(self: 'a)
  inherit point(x0,y0) as super
  val c = c0
  method getc = c
  method equal (p:'a) =
    (self#getx = p#getx) && (self#gety = p#gety) && (self#ge
end

let x = new point (0,1);;
x#equal x;;
let y = new colored_point (0,1) "red";;
let z = new colored_point (0,1) "black";;
y#equal z;;
```

```
(* on ne peut pas pas comparer un point avec un point coloré
x#equal y;;
y#equal x;;

(* de plus, on ne peut pas coercer un colored_point vers un p
(y :> point);;
```

14 Design patterns

Quand on programme en orienté objet, dans des langages sans typage fort, il est très important de respecter une certaine discipline de programmation, telle qu'elle peut se trouver par exemple dans

> Design Patterns (1994)
> Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides