

Modèles de Programmation



Emmanuel Chailloux

Programmation par objets:

- ▶ polymorphisme de rangée
 - ▶ enregistrements
 - ▶ variables de rangées
- ▶ sous-typage objet structurel
- ▶ objets en OCaml
 - ▶ classes, objets, héritage
 - ▶ sous-typage et méthodes binaires

Les langages à objets (1) : historique

historique des langages : poster O'Reilly

http://oreilly.com/news/graphics/prog_lang_poster.pdf

- ▶ années 80 : la recherche
 - ▶ communauté scientifique : langages de programmation + IA
 - ▶ langages Simula, SmallTalk (80)
- ▶ années 90 : l'industrie
 - ▶ langages ou extensions objets utilisés dans l'industrie : SmallTalk, C++ (ATT), Objective C (NextStep, puis MacOSX), CLOS, Delphi (Borland), Java (Sun 95), C# (Microsoft), Python, Javascript (NetScape 95), Ruby, ...
 - ▶ émergence du génie logiciel : langage de modélisation (UML)
- ▶ années 2000 : méthodes et outils
 - ▶ Programmation générique typée, tests unitaires,
 - ▶ Intégration d'autres paradigmes : fonctionnel, concurrent, ...
 - ▶ génie logiciel orienté modèles (voir cours Ingénierie Logicielle - M1)

Les langages à objets (2) : caractéristiques

- ▶ avec structuration en classes
 - ▶ typés dynamiquement (SmallTalk)
 - ▶ typés statiquement
 - ▶ sous-typage nominal (C++, Java, C#, Scala)
 - ▶ sous-typage structurel (OCaml)
- ▶ sans classes
 - ▶ à base de multi-méthodes, fonctions génériques (CLOS)
 - ▶ à base de prototypes (JavaScript)

Polymorphisme paramétrique (1)

système de types simples :

- ▶ type : $\tau ::= \alpha \mid \tau * \tau \mid \tau \rightarrow \tau$
- ▶ schéma de types : $\sigma ::= \forall \vec{\alpha}. \tau$

où τ est un type, α une variable de type et σ un schéma de type.

le polymorphisme est introduit par le **let** :

(Var)

$$(x_1 : \sigma_1), \dots, (x_n : \sigma_n) \vdash x_i : \tau[\tau_i / \alpha_i] \quad \sigma_i = \forall \alpha_1, \dots, \alpha_n. \tau$$

(Let)

$$\frac{C \vdash N : \tau \quad \alpha_1, \dots, \alpha_n = V(\tau) - V(C) \quad (x : \forall \alpha_1, \dots, \alpha_n. \tau) \quad C \vdash M : \tau'}{C \vdash \text{let } x = N \text{ in } M : \tau'}$$

Polymorphisme paramétrique (2)

- ▶ inférence en OCaml :

```
1 # let id = function x -> x;;
2 val id : 'a -> 'a = <fun>
3
4 # let rec map f l = if l = [] then [] else (f(hd l))::(map f (tl l));;
5 val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
6
7 # let compose = function f -> function g -> function x -> f (g x);;
8 val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

- ▶ vérification des génériques en Java/C#

```
1 ArrayList<Integer> al = new ArrayList<Integer>();
2 al.add(new Integer(3));
3 Integer i = al.get(0);
```

typage objet structurel (variables de rangée)

système de types simples + enregistrement :

- ▶ type : $\tau ::= \alpha \mid \tau * \tau \mid \tau \rightarrow \tau \mid \langle m_1 : \tau_1; \dots; m_k : \tau_k ; \rho \rangle$
- ▶ schéma de types : $\sigma ::= \forall \vec{\alpha} \vec{\rho}. \tau$

où $m_1 \dots m_k$ sont des labels et ρ une variable de rangée
et τ est un type, α une variable de type et σ un schéma de type.

Les variables de rangée vont permettre le typage structurel des objets.

- ▶ type objet fermé : sans variable de rangée
- ▶ type objet ouvert : avec variable de rangée

Enregistrements

en OCaml :

```
1 # type complexe = {mutable re : float; mutable im : float} ;;
2 type complexe = { mutable re : float; mutable im : float; }
3 # let c = {re=3.1; im = 2.2} ;;
4 val c : complexe = {re = 3.1; im = 2.2}
5 # c.im ;;
6 - : float = 2.2
7 # let getIm c = c.im;;
8 val getIm : complexe -> float = <fun>
```

le champ `im` est lié au type `complexe`.

on aurait aimé une fonction `getIm` qui prend n'importe quel enregistrement ayant un champ `im`.

pour les objets on peut utiliser les relations de sous-typage mais aussi le polymorphisme de rangées.

Objets

On peut coder les objets comme des enregistrements, où chaque label correspond à une méthode à laquelle on associe son type : $t = \langle m_1 : \tau_1; \dots; m_n : \tau_n \rangle$. où chaque m_i correspond à une méthode que l'on peut appeler sur un objet de type t .

(Call)

$$\frac{C \vdash o : \langle m_1 : \tau_1; \dots; m_n : \tau_n \rangle}{C \vdash o \# m_i : \tau_i}$$

```
1 # let f (x:<getIm : unit -> float> ) = x # getIm() +. 1.2;;  
2 val f : < getIm : unit -> float > -> float = <fun>
```

Polymorphisme de rangée

un type objet peut aussi avoir des variables de rangées :

$$\text{let } f = \lambda x. (x\#m)$$

► $f : \forall \alpha \rho. \langle m : \alpha \mid \rho \rangle \rightarrow \alpha$

```
1 # let f = function x -> x # getIm() +. 1.2;;
2 val f : < getIm : unit -> float; .. > -> float = <fun>
3
4 # let g = function x -> x#m ;;
5 val g : < m : 'a; .. > -> 'a = <fun>
```

f attend un argument de type objet ayant au moins une méthode `getIm` de type $unit \rightarrow float$. Elle pourra donc s'appliquer à tout objet ayant une méthode de ce type.

(Call)

$$\frac{C \vdash o : \langle m_1 : \tau_1; \dots; m_n : \tau_n; \rho \rangle}{C \vdash o\#m_i : \tau_i}$$

Type cyclique

Le lieur μ est un opérateur de point fixe d'un type. Il introduit un type cyclique, c'est-à-dire le remplacement d'une variable de type par un type la contenant.

La méthode m_2 du type objet suivant :

$\mu\alpha. \langle m1 : int; m2 : int \rightarrow \alpha \rangle$ retourne un objet du même type que le receveur de la méthode.

En Ocaml μ se traduit pas as :

```
1 # type point = <posx : int; deplacex : int -> 'a> as 'a;;  
2 type point = < deplacex : int -> 'a; posx : int > as 'a
```

où $'a$ correspond au type $\langle posx : int; deplacex : int \rightarrow 'a \rangle$ (contenant $'a$).

Sous-typage structurel (1)

- ▶ principe de subsomption
 - ▶ utilisation d'un objet d'une certaine classe/spécification à la place et lieu d'un objet d'une autre classe/spécification
- ▶ vérification
 - ▶ par la relation de sous-typage (notée \leq)
on peut utiliser une valeur d'un sous-type lorsqu'une valeur d'un super-type est attendue. Dans ce cas la valeur n'est pas changée, elle est juste vue sous un type différent, c'est toujours la même référence.

Si τ est sous-type de τ' alors un terme de type τ peut être utilisé avec le type τ'

$$\frac{e : \tau \quad \tau \leq \tau'}{e : \tau'}$$

Sous-typage structurel (2)

- ▶ en largeur : La relation de sous-typage correspond à l'inclusion des champs.

$$\langle m : A ; m' : B \rangle \leq \langle m : A \rangle$$

- ▶ en profondeur :

$$\langle m : A \rangle \leq \langle m : A' \rangle \text{ si } A \leq A'$$

- ▶ avec type cyclique : attention à la variance :
 $\mu\alpha. \langle m1 : int ; m2 : int \rightarrow \alpha \rangle$

Variances et sous-typage

sous-typage, type fonctionnel:

On suppose un type A qui dépend d'un type X (on note $A(X)$), et les types S et T en relation de sous-typage ($S \leq T$). On dit que la dépendance de A à X est

- ▶ **Covariante** : si $A(S) \leq A(T)$
- ▶ **Contravariante** : si $A(S) \geq A(T)$
- ▶ **Invariante** : dans les autres cas

notation : on peut trouver des indications de variance sur des paramètres de type : + pour covariant et - pour contravariant (OCaml, Scala, ...).

Extension objet en OCaml

- ▶ extension objet \neq langage objet
- ▶ langage à classes
- ▶ sans surcharge
- ▶ avec héritage multiple
- ▶ et classes paramétrées
- ▶ sous-typage \neq sous-classes

Seul langage avec extension objet, statiquement typé avec inférence de types!!!

Déclaration d'une classe:

```
class [virtual] nom [ p1 p2 ... pn ] =  
  object [ ( p ) ]  
    inherit autre_classe [ pi pj ]  
    constraint typeexpr = typeexpr  
    val [mutable] ident = expr  
    initializer expr  
    method [private] [virtual] nom_methode = expr  
end
```


Classe Point

```
1 class point (x_init,y_init) =
2 object
3   val mutable x = x_init
4   val mutable y = y_init
5   method get_x = x
6   method get_y = y
7   method moveto (a,b) = begin x <- a; y <- b end
8   method rmoveto (dx,dy) =
9     begin x <- x + dx; y <- y + dy end
10  method to_string () = "( ^(string_of_int x)^
11                        ", ^(string_of_int y)^)"
12  method distance () = sqrt(float(x*x + y*y))
13 end;;
```

Qu'infère OCaml ?

2 choses:

- ▶ 1 abréviation d'un type object
- ▶ 1 fonction de construction à utiliser avec new

```
1 class point : int * int ->  
2   object  
3     val mutable x : int  
4     val mutable y : int  
5     method distance : unit -> float  
6     method get_x : int  
7     method get_y : int  
8     method moveto : int * int -> unit ...  
9     method to_string : unit -> string   end
```

Appel de méthode (envoi de messages)

Un objet sait répondre à un envoi de message du nom d'une méthode de sa classe suivi des paramètres du bon type.

On utilise la notation # :

```
1 # p1#get_x;;           - : int = 0
2 # p2#get_y;;           - : int = 4
3 # p1#to_string();;     - : string = "( 0, 0)"
4 # p2#to_string();;     - : string = "( 3, 4)"
5 # if (p1#distance()) = (p2#distance())
6     then print_string ("c'est le hasard\n")
7     else print_string ("on pouvait parier\n");;
8 on pouvait parier
```

Type des instances

Le type inféré pour les instances p1 et p2 est le type objet (<obj> point). C'est une abréviation du type objet long suivant :

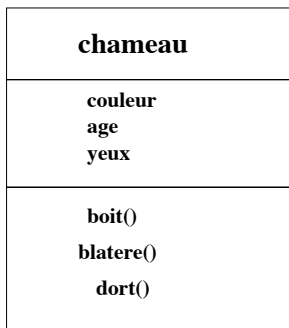
```
1 point =  
2   < distance : unit -> float; get_x : int; get_y : int;  
3     moveto : int * int -> unit; rmoveto : int * int -> unit;  
4     to_string : unit -> unit; >
```

correspondant aux types de ses méthodes.

Typage statique: garantie que les requêtes (appel de méthode) pourront être traitées.

Notation graphique des classes

Les classes se notent par un rectangle constitué de trois parties. Une partie portant le nom de la classe. Une autre où figure les attributs d'une instance de la classe. Enfin une dernière où sont inscrites les méthodes d'une instance de la classe.



Relations entre objets

- ▶ relation d'agrégation : **Has-a**

ex1: C_1 a un champs de C_2

ex2: C_1 a de 0 à n champs de C_2 ,

- ▶ relation d'héritage : **Is-a**

ex3: SC est sous-classe de C

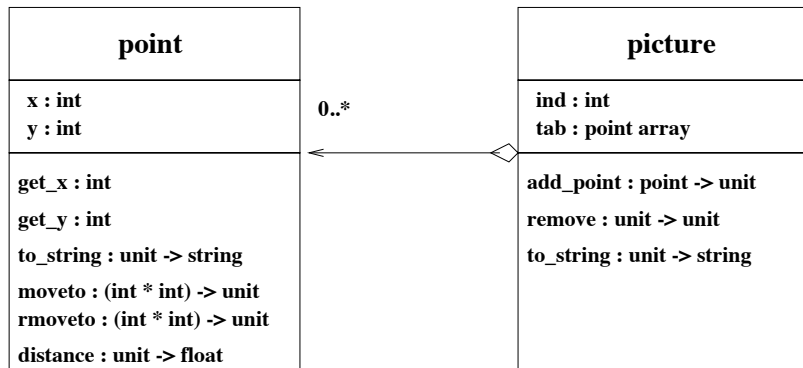
C'est l'avantage majeur de la programmation objet que de pouvoir étendre le comportement d'une classe existante tout en continuant à utiliser le code écrit par la classe originale.

Quand on étend une classe, la nouvelle classe hérite de tous les champs, de données et de méthodes, de la classe qu'elle étend.

Exemple d'agrégat

classe picture:

- ▶ possède entre 0 à n instances de point



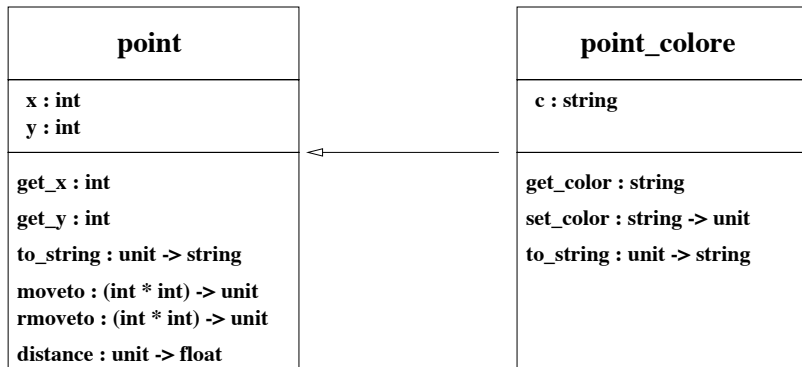
Code de la classe picture

```
1 class picture n =
2 object
3   val mutable ind = 0
4   val tab = Array.create n (new point(0,0))
5   method add p = if (ind < n -1) then begin
6     tab.(ind)<-p; ind <- ind + 1 end
7   else failwith ("picture.add:ind ="^(string_of_int ind))
8   method remove () = if (ind > 0) then ind <-ind-1
9   method to_string () = let s = ref "" in
10     for i=0 to ind do s:= !s^" "^tab.(i)#to_string() done;
11     !s
12 end;;
```


Exemple d'héritage

classe point_colore:

- ▶ hérite de la classe point



Code de la classe point_couleur

```
1 class point_couleur ((x_init,y_init) as paire) c =  
2 object  
3   inherit point paire  
4   val mutable c = c  
5   method get_color = c  
6   method set_color nc = c <- nc  
7   method to_string () =  
8     "( " ^ (string_of_int x) ^  
9     ", " ^ (string_of_int y) ^ ")" ^  
10    " de couleur " ^ c  
11 end;;
```

Référencement : self et super

```
1 class point_colore p c =  
2 object(self)  
3   inherit point p as super  
4   val c = c  
5   method get_color = c  
6   method to_string () =  
7     begin  
8       super#to_string() ^" de couleur " ^ self#get_color  
9     end  
10  end;;
```

Les noms sont libres, mais on utilisera `this` ou `self` pour soi-même et `super` pour la classe ancêtre.

Liaison retardée

- ▶ liaison “tardive” ou “retardée” (ou liaison “dynamique”) :
*détermination à l'exécution de la méthode à utiliser
lors de l'envoi d'un message*
- ▶ liaison “précoce” (ou liaison “statique”) :
effectue cette résolution à la compilation

OCaml implante la liaison retardée!!!

Recherche d'une méthode

Le typage statique garantit que l'envoi d'un message correspondra bien à l'appel d'une méthode de même nom et de même signature.

Exemple de liaison retardée

On modifie la méthode distance de la classe point :

```
1 method distance () =  
2   sqrt(float(self#get_x*self#get_x) +.  
3     float(self#get_y*self#get_y) +. )
```

et on redéfinit la méthode get_x de point_coloire :

```
1 method get_x = x * 2
```

Alors

```
1 (new point_coloire (2,3) "bleu")#distance()
```

retourne la valeur $5.0 \neq \sqrt{4+9}$

⇒ permettant de modifier le comportement de méthodes héritées.

Initialisation

initialiseur: méthode particulière déclenchée immédiatement après la construction de l'objet.

```
1  class point (x_init,y_init) =  
2  object ...  
3  initializer print_string "Creation d'un point";  
4             print_newline(); flush stdout  
5  end;;  
6  class point_colore p c =  
7  object  
8     inherit point p ...  
9     initializer print_string "Creation d'un point colore";  
10             print_newline(); flush stdout  
11 end;;
```

Trace d'initialiseurs

L'exécution suivante permet de suivre l'ordre de déclenchement de la construction des objets et de leur initialisateur :

```
1
2 # let p = new point;;
3 val p : int * int -> point = <fun>
4 # let p = new point (3,4);;
5 Creation d'un point
6 val p : point = <obj>
7 # let pc = new point_colore (3,4) "blanc";;
8 Creation d'un point
9 Creation d'un point colore
10 val pc : point_colore = <obj>
```


Classes et méthodes abstraites

classe abstraite: classe dont certaines méthodes ne possèdent pas de corps.

- ▶ ces méthodes sont dites *abstraites*;
- ▶ utilisation du mot clé `virtual`.

Si une sous-classe, d'une classe abstraite, redéfinit toutes les méthodes abstraites de l'ancêtre, alors elle devient concrète, sinon elle reste abstraite.

Exemple d'une classe abstraite

```
1 class virtual graphical_object () =  
2 object(self)  
3   method virtual to_string : unit -> string  
4   method display () = print_string (self#to_string())  
5 end;;
```

L'interface calculée est la suivante : _____

```
1 class virtual graphical_object :  
2   unit ->  
3   object  
4     method display : unit -> unit  
5     method virtual to_string : unit -> string  
6   end
```

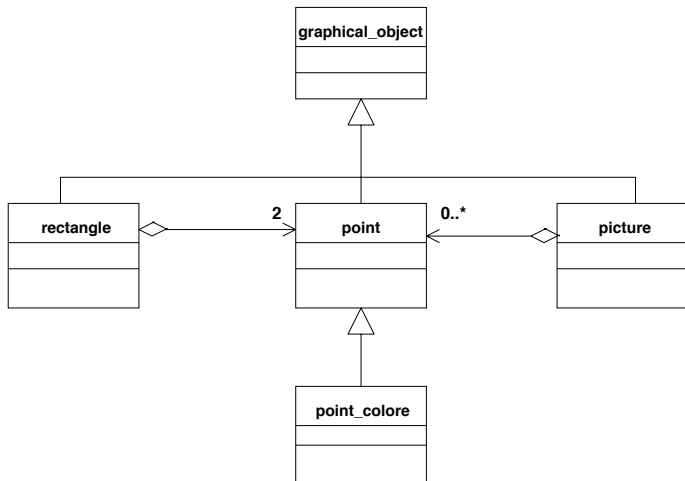
Classe rectangle

```
1 class rectangle (p1,p2) =
2 object
3   inherit graphical_object ()
4   val mutable llc = (p1 : point)
5   val mutable ruc = (p2 : point)
6   method to_string()="(^p1#to_string()^", "^
7                       p2#to_string()^")"
8 end;;
```

d'interface

```
1 class rectangle :
2   point * point -> object
3   val mutable llc : point
4   val mutable ruc : point
5   method to_string : unit -> string end
```

Diagramme des relations de classes



Objets et types

Le type d'un objet est le type de ses méthodes. Par exemple le type point est une abréviation du type :

```
1 point =<distance: unit -> float; get_x: int; get_y: int;  
2     moveto: int * int -> unit; rmoveto: int*int-> unit;  
3     to_string: unit -> string >
```

Lors d'un envoi de message l'inférence de types peut construire un type objet ouvert :

```
1 # let f x = x#get_x;;  
2 val f : < get_x : 'a; .. > -> 'a = <fun>  
3 # let p = new point(2,3);;  
4 val p : point = <obj>  
5 # f p;;           - : int = 2
```

Types ouverts

type ouvert: est représenté par la notation `< ..>`, pour passer d'un type objet fermé à un type objet ouvert, on utilisera alors la notation `#type_obj` comme dans l'exemple suivant :

```
1 # let g (x : #point) = x#amess;;
2 val g :
3 <amess: 'a; distance: unit -> float; get_x: int; get_y: int;
4   moveto: int * int -> unit; to_string: unit -> string;
5   rmoveto : int * int -> unit; .. > -> 'a = <fun>
```

où la coercion de type avec `#point` force `x` à avoir au moins toutes les méthodes de `point`, et l'envoi du message `amess` ajoute une méthode au type du paramètre `x`.

Héritage multiple

L'héritage multiple permet d'hériter des champs de données et des méthodes de plusieurs classes.

En cas de noms de champs ou de méthodes identiques, seulement la dernière déclaration, dans l'ordre de la déclaration de l'héritage, sera conservée.

Les différentes classes héritées n'ont pas forcément de liens d'héritage entre elles.

Intérêt: augmenter la réutilisabilité des classes.

Classes paramétrées

Utilisation: du polymorphisme paramétrique dans les classes

Intérêt: augmente la généricité du code

```
1 class ['a,'b] pair (a:'a) (b:'b) =
2   object
3     val x = a
4     val y = b
5     method fst = x
6     method snd = y
7   end;;
8 # let v = new pair 3 true;;
9 val v : (int, bool) pair = <obj>
```


Classe paramétrée Pile

```
1 class ['a] pile ((x:'a),n) =
2 object(self)
3   val mutable ind = 0
4   val tab = Array.create n x
5   method is_empty () = if ind = 0 then true else false
6   method private is_full () =
7     if ind = n+1 then true else false
8   method pop() =
9     if self#is_empty() then failwith "pile vide"
10    else ind <- ind -1 ; tab.(ind)
11   method push y =
12     if self#is_full() then failwith "pile pleine"
13     else tab.(ind) <- y; ind <- ind + 1
14 end;;
```

Variables libres dans une classe

Une définition de classe engendre un constructeur et une abréviation de type.

Comme toute déclaration de type, toute variable de type doit être liée :

- ▶ comme variable d'une classe paramétrée
- ▶ variable de rangée
- ▶ liée au nom de l'instance
- ▶ liée à une méthode polymorphe

Néanmoins le type de `self` peut apparaître dans un type cyclique :

```
1 class t = object (self : 'a)
2   val x = 0
3   method getX = x
4   method getSelf () = self
5   end ;;
6 class t :
7   object ('a) val x : int method getSelf : unit -> 'a method getX :
8   int end
```

Sous-typage (1)

sous-typage: est une relation entre deux types objets.

Soient $t = \langle m_1 : \tau_1; \dots; m_n : \tau_n \rangle$ et $t' = \langle m_1 : \sigma_1; \dots; m_n : \sigma_n; \tau' \rangle$ où τ' est une suite de méthodes, on dit que t' est un sous-type de t dans C (contexte de typage), noté $t' \leq t$ si $\sigma_i \leq \tau_i$ pour $i \in \{1, \dots, n\}$.

subsumption: est la possibilité pour un objet d'un certain sous-type d'être considéré et utilisé comme un objet d'un sur-type au sens de la relation de sous-typage.

sous-typage (2)

Notation: La relation "est un sous-type de" se note $:>$. On note que `point_colore` est un sous-type de `point` de la manière suivante :

```
point_colore :> point
```

Si le membre gauche de la relation est omis, alors c'est le type de la valeur qui sera considéré comme membre gauche.

La relation de sous-typage, combinée avec la liaison tardive, introduit une nouvelle forme de polymorphisme : le polymorphisme d'**inclusion**.

Sous-typage et polymorphisme d'inclusion

Soient les déclarations suivantes :

```
1 # let p = new point (4,5);;
2 val p : point = <obj>
3 # let pc = new point_colore (4,5) "blanc";;
4 val pc : point_colore = <obj>
5 # let np = (pc :> point);;
6 val np : point = <obj>
7 # let np2 = (pc : point_colore :> point);;
8 val np2 : point = <obj>
```

Invocation: de la méthode `to_string`

```
1 # p#to_string();;
2 - : string = "( 4, 5)"
3 # pc#to_string();;
4 - : string = "( 4, 5) de couleur blanc"
5 # np#to_string();;
6 - : string = "( 4, 5) de couleur blanc"
```

où l'envoi d'un message `to_string` sur `np`, valeur considérée de type `point` déclenche la méthode `to_string` de la classe `point_colore`.

Exemple

Construction: d'une liste de points

```
1 # let l = [p; np];;
2 val l : point list = [<obj>; <obj>]
3 # List.map (fun x -> x#to_string()) l;;
4 - : string list = ["( 4, 5)";
5     "( 4, 5) de couleur blanc"]
```

Cela vient de la liaison tardive (choix de la méthode à utiliser à l'exécution).

Sous-typage \neq héritage

2 arguments:

- ▶ on peut être sous-type sans héritage

il est possible de forcer un type classe dans un autre type classe sans que le premier corresponde à un descendant du deuxième

- ▶ on peut hériter sans être sous-type

cela arrive quand une des méthodes de la classe ancêtre a un paramètre du type de la classe

Sous-typage entre objets

Soient $t = \langle m_1 : \tau_1; \dots; m_n : \tau_n \rangle$ et
 $t' = \langle m_1 : \sigma_1; \dots; m_n : \sigma_n; \tau' \rangle$ où τ' est une suite de méthodes,
on dit que t' est un sous-type de t dans C (contexte de typage),
noté $t' \leq t$
si $\sigma_i \leq \tau_i$ pour $i \in \{1, \dots, n\}$.

appel de fonctions: Si $f : \sigma \rightarrow \tau$ dans C , $a : \sigma'$ dans C et
 $\sigma' \leq \sigma$ dans C

alors (fa) est bien typé dans C et a le type τ .

Une fonction f qui attend un argument de type σ peut recevoir
sans danger un argument d'un sous-type de σ .

sous-typage de types fonctionnels (1)

Si on définit les classes suivantes :

```
1  class a =  
2    ...  
3    method f : t1 -> t2  
4    ...  
5  end;;  
6  class b =  
7    ...  
8    method f : t3 -> t4  
9    ...  
10 end;;
```

Si on veut montrer que $b \leq a$ alors il faut vérifier
 $(t_3 \rightarrow t_4) \leq (t_1 \rightarrow t_2)$.

sous-typage de types fonctionnels (2)

Pour distinguer les deux méthodes f on les nomme : f_a et f_b .
Soient $t_1 \rightarrow t_2$ et $t_3 \rightarrow t_4$ deux types fonctionnels, ils sont en relation de sous-typage :

$$(t_3 \rightarrow t_4) \leq (t_1 \rightarrow t_2)$$

si et seulement si :

- ▶ $t_4 \leq t_2$ (co - variance)
- ▶ $t_1 \leq t_3$ (contra - variance)

Justification (1)

Soient les 2 fonctions suivantes bien typées :

```
1 let g (p : t2) = ...
2 let h ((o:a),(x:t1)) = g(o#f(x));;
```

avec

```
1 g : t2 -> nt
2 h : ( a * t1) -> nt
```

- ▶ **[co-variance]** : la fonction g attend un argument de type t_2 ou d'un de ses sous-types. Comme cet argument est dans le corps de h résultat de l'envoi du message $f(x)$, il peut être résultat de l'appel de f_b , donc :

$$\text{type_res}(f_b) \leq \text{type_res}(f_a) \Rightarrow t_4 \leq t_2$$

Justification (2)

- ▶ **[contra-variance]** : En appliquant f à une instance de b (notée o_b on obtient :

$$h(o_b, x) \Rightarrow g(o_b \# f_b(x))$$

Le type de x est t_1 (type des arguments de f_a , mais il doit pouvoir être passé comme argument de f_b (de type t_3 , donc

$$(type_arg(f_a) = type(x) = t_1) \leq type_arg(f_b) \Rightarrow t_1 \leq t_3$$

La relation $t_3 \leq t_1$ est impossible car alors f_b ne pourrait recevoir un argument de type t_1 et l'appel $h(o_b, r)$ avec r de type t_1 serait alors incorrect.

Exemples

En reprenant l'exemple sur les `point` et `point_colore` précédent, on obtient :

$$eq_{point} : point \rightarrow bool \quad eq_{point_colore} : point_colore \rightarrow bool$$

et on s'aperçoit alors que pour que

$$point_colore \leq point$$

il faudrait que

$$(point_colore \rightarrow bool) \leq (point \rightarrow bool)$$

c'est-à-dire, avec la relation de contra-variance des types fonctionnels

$$point \leq point_colore$$

Méthodes polymorphes (1)

```
1  exception Empty
2  class oqueue () =
3    object(self)
4      val mutable q = []
5      method enq x = q <- q @ [x]
6      method deq () = match q with [] -> raise Empty
7                          | h::r -> q <- r ; h
8      method reset () = q <- []
9      method fold f accu = List.fold_left f accu q
10  end;;
```

```
1  File "oqueue.ml", line 3, characters 5-260:
2  Some type variables are unbound in this type:
3    class oqueue :
4      unit ->
5      object
6        val mutable q : 'a list
7        method deq : unit -> 'a
8        method enq : 'a -> unit
9        method fold : ('b -> 'a -> 'b) -> 'b -> 'b
10       method reset : unit -> unit
11     end
12  The method deq has type unit -> 'a where 'a is unbound
```

Méthodes polymorphes (2)

```
1 exception Empty
2 class ['a, 'b] oqueue2 () =
3   object(self)
4     val mutable q = ([] : 'a list)
5     method enq x = q <- q @ [x]
6     method deq () = match q with
7       [] -> raise Empty
8       | h::r -> q <- r ; h
9     method reset () = q <- []
10    method fold f (accu : 'b) = List.fold_left f accu q
11  end;;
```

```
1 class ['a, 'b] oqueue2 :
2   unit ->
3   object
4     val mutable q : 'a list
5     method deq : unit -> 'a
6     method enq : 'a -> unit
7     method fold : ('b -> 'a -> 'b) -> 'b -> 'b
8     method reset : unit -> unit
9   end
```

Méthodes polymorphes (3)

```
1 # let oq = new oqueue2();;
2 val oq : ('_a, '_b) oqueue2 = <obj>
3 # oq#enq "Salut";;
4 - : unit = ()
5 # oq;;
6 - : (string, '_a) oqueue2 = <obj>
7 # oq#enq "bye";;
8 - : unit = ()
9 # oq;;
10 - : (string, '_a) oqueue2 = <obj>
11 # oq#fold (fun x y -> x + (String.length y) ) 0;;
12 - : int = 8
13 # oq;;
14 - : (string, int) oqueue2 = <obj>
15 # oq#fold (fun x y -> x || (y = "Fin")) false;;
16 This expression has type int but is here used with type bool
```


Méthodes polymorphes (4)

- ▶ Si le polymorphisme d'une méthode est indépendant de variables de type de la classe de définition, alors il n'est pas dangereux de le lier localement.
- ▶ liaison explicite en indiquant le type et les variables quantifiés :

```
1 method nom : 'a 'b. ('a -> 'b -> 'a) = expr
```

Méthodes polymorphes (5)

```
1 exception Empty
2 class ['a] oqueue3 () =
3   object(self)
4     val mutable q = ([] : 'a list)
5     method enq x = q <- q @ [x]
6     method deq () = match q with
7       [] -> raise Empty
8       | h::r -> q <- r ; h
9     method reset () = q <- []
10    method fold : 'b. ('b -> 'a -> 'b) -> 'b -> 'b = fun f accu ->
11      List.fold_left f accu q
12  end;;
```

```
1 class ['a] oqueue3 :
2   unit ->
3   object
4     val mutable q : 'a list
5     method deq : unit -> 'a
6     method enq : 'a -> unit
7     method fold : ('b -> 'a -> 'b) -> 'b -> 'b
8     method reset : unit -> unit
9   end
```

Méthodes polymorphes (6)

```
1 # let oq = new oqueue3 ();;
2 val oq : '_a oqueue3 = <obj>
3 # oq#enq "Salut";;
4 - : unit = ()
5 # oq;;
6 - : string oqueue3 = <obj>
7 # oq#enq "bye";;
8 - : unit = ()
9 # oq#fold (fun x y -> x + (String.length y) ) 0;;
10 - : int = 8
11 # oq;;
12 - : string oqueue3 = <obj>
13 # oq#fold (fun x y -> x || (y = "Fin")) false;;
14 - : bool = false
```

Objets immédiats (1)

- ▶ création d'objet sans être instance de classe :

```
1 # let p =
2   object
3     val mutable x = 0
4     val mutable y = 0
5     method get_x = x
6     method get_y = y
7     method rmoveto dx dy = x <- x + dx; y <- y + dy
8   end;;
9 val p : < get_x : int; get_y : int; rmoveto : int -> int -> unit > = <↔
   obj>
10 # p#rmoveto 1 2;;
11 - : unit = ()
12 # p#get_y;;
13 - : int = 2
```

Objets immédiats (2)

- ▶ manipulation du type de self :

```
1  # let p3 = object(self:'a)
2      val mutable x = 0
3      val mutable y = 0
4      method get_x = x
5      method get_y = y
6      method rmoveto dx dy = x <- x + dx; y <- y + dy
7      method eq (z:'a) = x = (z#get_x)
8      method id () = self
9  end;;
10 val p3 :
11   < eq : 'a -> bool; get_x : int; get_y : int; id : unit -> 'a;
12     rmoveto : int -> int -> unit >
13 as 'a = <obj>
```

Objets immédiats (3)

- ▶ variables de type classiques :

```
1  # let r2 =
2    object(self)
3      val mutable q = ([] : 'a list)
4      method enq x = q <- q @ [x]
5    end ;;
6  val r2 : < enq : '_a -> unit > = <obj>
7  # let r4 z =
8    object(self)
9      val mutable q = z
10     method enq x = q <- q @ [x]
11   end ;;
12  val r4 : 'a list -> < enq : 'a -> unit > = <fun>
```

Objets immédiats (4)

- ▶ avantages
 - ▶ utilisable dans une fonction, ou un foncteur
 - ▶ moins de contraintes de types
- ▶ désavantages
 - ▶ pas d'héritage
 - ▶ type anonyme

Bibliographie

- ▶ Cours de Didier Rémy :
<http://gallium.inria.fr/~remy/poly/mot/>
- ▶ Cours de Maria-Virginia Aponte :
deptinfo.cnam.fr/~barthe/00/types-slid-2006.pdf
- ▶ Pierce (Benjamin). "Types and Programming Languages". MIT Press, 2002.