Typage et Polymorphisme

Cours 3

Emmanuel Chailloux

Spécialité Science et Technologie du Logiciel
Master mention Informatique
Université Pierre et Marie Curie

année 2013-2014

Plan du cours 3

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)

Polymophisme 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)

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

(Let)

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

Polymophisme paramétrique (2)

- inférence en OCaml :

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

- vérification des génériques en Jaca/C#

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

Variables de rangée

système de types simples + enregistrement :

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

où $m_1 ... m_k$ 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.

Enregistrements

en OCaml :

```
# type complexe = {mutable re : float; mutable im : float} ;;
type complexe = { mutable re : float; mutable im : float; }
# let c = {re=3.1; im = 2.2} ;;
val c : complexe = {re = 3.1; im = 2.2}
# c.im ;;
- : float = 2.2
# let getIm c = c.im;;
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 : $t =< m_1 : \tau_1; \ldots; m_n : \tau_n >$. où chaque m_i correspond à une méthode que l'on peut appeler sur un objet de type t.

(Call)

$$\frac{C \vdash o :< m_1 : \tau_1; ...; m_n : \tau_n >}{C \vdash o\#m_i : \tau_i}$$

```
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 :

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

- f : $\forall \alpha\rho.\ <\ m : \alpha \mid \rho\ > \rightarrow\ \alpha$

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

$$\frac{C \vdash o :< m_1 : \tau_1; ...; m_n : \tau_n; \rho >}{C \vdash o\#m_i : \tau_i}$$

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 :
$\mu\alpha. < m1 : int; m2 : int \rightarrow \alpha >$ 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 $< posx : int; deplacex : int \rightarrow 'a >$ (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.

  $< m : A ; m' : B > \leq < m : A >$

- en profondeur :

  $< m : A > \leq < m : A' >$ si $A \leq A'$

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 ! ! !

Classes

Déclaration d'une classe:

class [**virtual**] nom [p_1 p_2 ... p_n] =
  object [(p)]
    inherit autre_classe [p_i p_j]
    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 # :

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

Objets et types

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

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

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

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

```
# let g (x : #point) = x#amess;;
val g :
<amess: 'a; distance: unit -> float; get_x: int; get_y: int;
 moveto: int * int -> unit; to_string: unit -> string;
 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

Sous-typage (1)

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

Soient $t =< m_1 : \tau_1; \ldots; m_n : \tau_n >$ et
$t' =< m_1 : \sigma_1; \ldots; m_n : \sigma_n; \tau' >$ 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 $i \in \{1, \ldots, n\}$.

subsomption: 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 :

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

Invocation: de la méthode `to_string`

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

```
# let l = [p; np];;
val l : point list = [<obj>; <obj>]
# List.map (fun x -> x#to_string()) l;;
- : string list = ["( 4, 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 $t =< m_1 : \tau_1; \ldots; m_n : \tau_n >$ et
$t' =< m_1 : \sigma_1; \ldots; m_n : \sigma_n; \tau' >$ 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 $i \in \{1, \ldots, n\}$.

appel de fonctions: Si $f : \sigma \to \tau$ 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 $t_1 \to t_2$ et $t_3 \to t_4$ deux types fonctionnels, ils sont en relation de sous-typage :
$(t_3 \to t_4) \leq (t_1 \to t_2)$

si et seulement si :

- $t_4 \leq t_2$ $(\text{co} - \text{variance})$
- $t_1 \leq t_3$ $(\text{contra} - \text{variance})$

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 :

$$type\_res(f_b) \leq type\_res(f_a) \Rightarrow t_4 \leq t_2$$

Justification (2)

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

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

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 :

$$eq_{point} : point \to bool \quad eq_{point\_colore} : point\_colore \to bool$$

et on s'aperçoit alors que pour que

$$point\_colore \leq point$$

il faudrait que

$$(point\_colore \to bool) \leq (point \to bool)$$

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

$$point \leq point\_colore$$

Méthodes polymorphes (1)

```
exception Empty
class oqueue () =
  object(self)
    val mutable q = []
    method enq x =  q <-  q @ [x]
    method deq () = match q with [] -> raise Empty
                               |  h::r -> q <- r ; h
    method reset () = q <- []
    method fold f accu = List.fold_left f accu q
  end;;
```

```
File ‘‘oqueue.ml’’, line 3, characters 5-260:
Some type variables are unbound in this type:
  class oqueue :
    unit ->
    object
      val mutable q : 'a list
      method deq : unit -> 'a
      method enq : 'a -> unit
      method fold : ('b -> 'a -> 'b) -> 'b -> 'b
      method reset : unit -> unit
    end
The method deq has type unit -> 'a where 'a is unbound
```

Méthodes polymorphes (2)

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

```
class ['a, 'b] oqueue2 :
  unit ->
  object
    val mutable q : 'a list
    method deq : unit -> 'a
    method enq : 'a -> unit
    method fold : ('b -> 'a -> 'b) -> 'b -> 'b
    method reset : unit -> unit
  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)

```
exception Empty
class ['a] oqueue3 () =
  object(self)
    val mutable q = ([] : 'a list)
    method enq x =  q <-  q @ [x]
    method deq () = match q with
       [] -> raise Empty
    |  h::r -> q <- r ; h
    method reset () = q <- []
    method fold : 'b. ('b -> 'a -> 'b) -> 'b -> 'b =  fun f accu ->
       List.fold_left f accu q
  end;;
```

```
class ['a] oqueue3 :
  unit ->
  object
    val mutable q : 'a list
    method deq : unit -> 'a
    method enq : 'a -> unit
    method fold : ('b -> 'a -> 'b) -> 'b -> 'b
    method reset : unit -> unit
  end
```

Méthodes polymorphes (6)

```
# let oq = new oqueue3 ();;
val oq : '_a oqueue3 = <obj>
# oq#enq ``Salut'';;
- : unit = ()
# oq;;
- : string oqueue3 = <obj>
# oq#enq ``bye'';;
- : unit = ()
# oq#fold (fun x y -> x + (String.length y) ) 0;;
- : int = 8
# oq;;
- : string oqueue3 = <obj>
# oq#fold (fun x y -> x || (y = ``Fin'')) false;;
- : 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 :

```
# let p3 = object(self:'a)
    val mutable x = 0
    val mutable y =0
    method get_x = x
    method get_y = y
    method rmoveto dx dy = x <- x + dx; y <- y + dy
    method eq (z:'a) = x = (z#get_x)
    method id () = self
  end;;
val p3 :
  < eq : 'a -> bool; get_x : int; get_y : int; id : unit -> 'a;
    rmoveto : int -> int -> unit >
  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 Dider Rémy : `http://gallium.inria.fr/~remy/poly/mot/`
- Pierce (Benjamin). “Types and Programming Languages”. MIT Press, 2002.