Introduction au typage des objets en Ocaml

2005 – 2006

María-Virginia Aponte

18 mai 2007

Le typage

- Analyse des programmes visant à détecter les incohérences de types :
  - un opérateur qui est appliqué à des objets pour lesquels il n'est pas défini,
  - l'utilisation d'un entier comme un pointeur, etc.
- Ces incohérences, peuvent occasionner, soit une erreur fatale (arrêt du programme), soit un résultat incorrect.

Les langages *fortement typés*, détectent et proposent un diagnostic des erreurs de type trouvés dans les programmes.

Typage statique vs. dynamique

- *Typage dynamique* : réalisé *pendant* l'exécution.
  - Les tests sur les types sont insérés dans le code compilé et réalisés pendant l'exécution du programme $\Rightarrow$ diminution considérable de l'efficacité.
  - Si une erreur est détectée, elle est signalée et le programme est arrêté.
- *Typage statique* : réalisé à la compilation.

  Le code compilé ne contient aucun test sur les types, leur cohérence ayant ete vérifiée avant la traduction en code machine.

Sûreté des programmes typés statiquement

On peut **séparer typage et exécution** grâce à une propriété très importante du typage statique :

les programmes acceptés par le typeur sont garantis libres d'erreurs de types : ils ne produiront pas d'erreurs dûs aux types pendant leur exécution.

Bien entendu ils pourront toujours produire d'autres erreurs (des entrées/sorties, de non terminaison, etc).

Il s'agit donc d'un résultat de *correction partielle*.

L'enjeu du typage est donc celui de détecter les erreurs des types le plus tôt possible avant son exécution : idéalement, au moment de la compilation.

Le type des objets

Le type d'un objet est un petit enregistrements de types :

- Un objet est formé d'une suite de composantes (variables d'instance, méthodes).
- Chaque composante possède un nom (comment dans un enregistrement).
- dans le type d'un objet, seuls les types des méthodes sont visibles.

Le type d'un objet est un "enregistrement" où chaque composante est le nom d'une méthode avec son type

Interface de point

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

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

Un point est construit en donnant un entier. L'objet résultat contient :

```
object
    val mutable x : int
    method getx : int
    method move : int -> unit
    method print : unit
 end
```

Le type des objets : enregistrements de types

$\Rightarrow$ Le type d'un objet est un "enregistrement" où chaque composante est le nom d'une méthode avec son type.

En Ocaml, seules les méthodes de l'objet sont accessibles. Les variables d'instance sont systématiquement cachées.

```
object
    val mutable x : int
    method getx : int
    method move : int -> unit
    method print : unit
 end
```

Le type d'un objet instance de `point` est :

$$< \texttt{getx} : \texttt{int}; \quad \texttt{move} : \texttt{int} \rightarrow \texttt{unit}; \quad \texttt{print} : \texttt{unit} >$$

Remarque : Le nom et type de la variable d'instance `x` sont cachés.

Typage structurel

Lorsque le typeur Ocaml compare deux types d'objets, il *compare leurs structures et pas leurs noms.*

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

class toto  =
 object
   method getx = 3
   method move x = print_string (string_of_int x)
   method print = print_string "toto"
end;;
class toto :
  object method getx : int
         method move : int -> unit
         method print : unit end
  end
```

Typage structurel (suite)

Quels sont les types des deux objets suivants ?

```
# let p = new point 2 ; ;
val p : point = <obj>

# let t = new toto ; ;
val t : toto = <obj>
```

$$\texttt{p} : \texttt{point} =< \texttt{getx} : \texttt{int}; \texttt{move} : \texttt{int} \rightarrow \texttt{unit}; \texttt{print} : \texttt{unit} >$$

$$\texttt{t} : \texttt{toto} =< \texttt{getx} : \texttt{int}; \texttt{move} : \texttt{int} \rightarrow \texttt{unit}; \texttt{print} : \texttt{unit} >$$

Les types de `p` et `t`, ont la même structure : ils ont les mêmes noms et types de méthodes.

Exemple

$\texttt{p} : < \texttt{getx} : \texttt{int}; \texttt{move} : \texttt{int} \rightarrow \texttt{unit}; \texttt{print} : \texttt{unit} >$

$\texttt{t} : < \texttt{getx} : \texttt{int}; \texttt{move} : \texttt{int} \rightarrow \texttt{unit}; \texttt{print} : \texttt{unit} >$

En Ocaml, les types de `p` et `t` sont considérés *égaux*.

Les confondre est tout à fait **sûr** du point de vue du typage : ils possèdent exactement les mêmes méthodes avec les mêmes types, et donc on peut effectuer avec eux les mêmes opérations.

Alors que l'on ne peut pas mélanger un int et un string :

```
# [1 ; "ab"] ; ;
        ^^
This expression has type string but is here
used with type int
```

On pourra mélanger p et t : leurs types sont identiques.

Mélange de types et polymorphisme paramétrique

Pourquoi ce code est mal typé ?

```
# [1 ; "ab"] ; ;
       ^^
This expression has type string but is here
used with type int
```

Le types des listes est `'a list`. Dans ce type, il faut instancier `'a` de manière à coller avec la liste `[1 ; "ab"]`. On aura ainsi trouvé son type.

Les contraintes :

- D'après `1` : $'\texttt{a} \mapsto \texttt{int}$
- D'après `"ab"` : $'\texttt{a} \mapsto \texttt{string}$
- et $\texttt{int} \neq \texttt{string}$

$\Rightarrow$ Contradiction !

Exemple de typage structurel

```
# [p ; t] ; ;
- : point list = [<obj> ; <obj>]

# let h(a :point) = a#getx + 1 ; ;
val h : point -> int = <fun>

# h p ; ;
- : int = 3

# h t ; ;
- : int = 4
```

Dans `[p ; t]` nous avons les contraintes :

- d'après `p` : $'$`a` $\mapsto$ `point`
- d'après `t` : $'$`a` $\mapsto$ `toto`
- et `point` = `toto` car ils sont deux abréviations pour la même structure.

$\Rightarrow$ `[p ;t] : point list`

Ocaml : typage structurel

Le type donné à `p` ou à `t` aurait pu être tout aussi bien `point` que `toto`.

`point` et `toto` sont deux *abréviations* pour une même structure de type.

⇒ En Ocaml, seule la structure des types est prise en compte pendant les typage.

Les abréviations sont systématiquement expansés et ce sont les structures qu'elles désignent qui sont comparées.

⇒ En Ocaml, le typage est structurel

Java : typage nominal

En Java deux types (classes) sont potentiellement comparables si leurs *noms* ont été déclarés dans une hiérarchie de sous-classes.

L'utilisation d'interfaces introduit du typage structurel.

Sans utilisation d'interfaces, deux objets de structure identique mais dans différentes hiérarchies de classes sont incompatibles $\Rightarrow$ seuls les noms comptent.

En Java le typage sur les classes est *nominal*

Typage structurel vs. nominal

Le typage structurel va dans le sens de la re-utilisation du code : une fonction qui demande à son argument d'avoir une certaine *structure*, peut accepter n'importe quel objet avec cette structure *indépendamment du nom de la classe* qui a servi à le générer.

Exemple : `f` peut être appliquée sur `p` et `t` :

```
#let f (x : point) = getx + 1 ; ;
val f : point -> int = <fun>

# let z1 = f(p) ; ;

# let z2 = f(t) ; ;
```

Le typage nominal est plus simple à implanter et est bien adapté aux test sur les types dans le code compilé, à la portabilité du code, etc.

Les types d'objets *fermés*

Quel est le type de la fonction ?

```
#let f (x : point) = x + 1 ; ;
val f : point -> int = <fun>
```

$$f : < \texttt{getx} : \texttt{int}; \texttt{move} : \texttt{int} \rightarrow \texttt{unit}; \texttt{print} : \texttt{unit} > \rightarrow \texttt{int}$$

Ce type est-il polymorphe ?

Types d'objets fermés (suite)

```
class pointColore (i,c) =
object
 inherit point i as super
 val c = c
 method get_couleur = c
 method print =  super#print ;
   print_string (", "^c)
end
class pointColore :
  int * string ->
  object
    val c : string
    val mutable x : int
    method get_couleur : string
    method getx : int
    method move : int -> unit
    method print : unit
  end

let pc = new pointColore (2,"rouge") ; ;

f(pc) ; ;
```

L'appel `f(pc)` est-il autorisé ?
Si oui ⇒ le type de `f` est polymorphe.

Types d'objets fermés = types monomorphes

```
# f(pc) ; ;
This expression has type pointColore but is here used with type point
Only the first object type has a method get_couleur
```

Cet appel est mal typé. Le type de `f` est monomorphe ! Il n'admet en argument que des objets ayant exactement la même structure que :

$$< \texttt{getx} : \texttt{int}; \texttt{move} : \texttt{int} \rightarrow \texttt{unit}; \texttt{print} : \texttt{unit} >$$

Quel est le problème ici ?

- Il est clair que `pointColore` est un **sous-type** de `point`.
- Par ailleurs, Ocaml admet le **polymorphisme paramétrique** pour les fonctions sur des valeurs **non objet**.

$\Rightarrow$ En Ocaml, les fonctions et les méthodes sur les objets n'admettent-elles pas le polymorphisme ? ? ?

Polymorphime objet d'Ocaml $\neq$ sous-typage

En Ocaml, les fonctions sur les objets et les méthodes n'admettent-elles pas le polymorphisme ? ? ?

Si, bien sûr, *mais pas au moyen du sous-typage.*

Le polymorphisme objet d'Ocaml *c'est aussi du polymorphisme paramétrique*

Les types d'objets ouverts

```
# let g(p) = p#getx + 1 ; ;
```

Quel est le type de `g` ? Le compilateur signale :

```
# let g(p) = p#getx + 1 ; ;
val g : < getx : int ; .. > -> int = <fun>
```

Le type `< getx : int ; .. >` est un **type ouvert**.

Il indique que l'argument de `g` peut être n'importe quel objet *possédant au moins une méthode* `getx` *de type* `int`, avec éventuellement des méthodes en plus.

La possibilité d'avoir plus de méthodes est exprimée par les points de suspension "...".

Fonctions polymorphes sur les objets

```
# let g(p) = p#getx + 1 ; ;
val g : < getx : int ; .. > -> int = <fun>
```

Exemple : on peut appliquer g à

```
# let p = new point 2 ; ;
val p : point = <obj>

let pc = new pointColore (4,"rouge") ; ;
val p : point = <obj>

# g(p) ; ;    (* p est un point *)
- : int = 2

# g(pc) ; ;   (* pc est un pointColore *)
- : int = 4
```

$\Rightarrow$ g est une fonction polymorphe sur les objets : elle admet tout objet possédant au moins une méthode `getx` de type `int`.

Fonctions polymorphes sur les objets (suite)

On ne peut pas appliquer g aux objets instance de la classe un :

```
class un =
 object
  method un = 1
end ; ;
class un  = method un : int end

let a = new un in g(a) ; ;
                        ^
This expression has type un = < un : int >
 but is here used with type  < getx : int ; .. >
```

Mais on peut appliquer g sur un objet qui n'est pas une instance de point, par exemple, sur un objet instance d'une sous-classe de toto.

En Ocaml il n'y a pas de hiérarchie de classes

Types ouverts : des types polymorphes paramétriques

```
# let g(p) = p#getx + 1 ; ;
val g : < getx : int ; .. > -> int = <fun>
```

La fonction `g` est clairement polymorphe :

$\Rightarrow$ elle peut recevoir des objets d'une infinité de types : tous ceux qui possèdent au moins une composante `getx` de type `int`.

Pour le typeur Ocaml, le type de `g` est :

$$\texttt{g} : < \texttt{getx} : \texttt{int}; \rho > \rightarrow \texttt{int}$$

où ρ est une *variable de type de rangée.*

Variables de type “de rangée”

$$\texttt{g} : < \texttt{getx} : \texttt{int}; \rho > \rightarrow \texttt{int}$$

où ρ est une *variable de type de rangée*.

C’est une *variable de type* d’une nouvelle nature :

1. Les variables de types habituelles `'a, 'b` peuvent être remplacées (instanciées) par n’importe quel type.

   Exemple : si $f : {'\texttt{a}} \rightarrow {'\texttt{a}}$, alors on peut remplacer $'\texttt{a}$ par `int`. f devient de type $\texttt{int} \rightarrow \texttt{int}$, et on peut l’appliquer par $f(1)$.

2. *une variable de rangée* ρ peut être remplacée par n’importe quelle **suite de méthodes avec leurs types (y compris la suite vide)**.

la variable ρ peut être instanciée en “plus de méthodes”.

Instancier une variable de rangée

Si on remplace

$$\rho \mapsto (\texttt{move} : \texttt{int} \rightarrow \texttt{unit}; \texttt{print} : \texttt{unit})$$

dans le type de g :

$$(< \texttt{getx} : \texttt{int}; \rho > \rightarrow \texttt{int})$$

nous obtenons

$$g : < \texttt{getx} : \texttt{int}; \texttt{move} : \texttt{int} \rightarrow \texttt{unit}; \texttt{print} : \texttt{unit} > \rightarrow \texttt{int}$$

c'est le type des fonctions admettant `point` en argument !

Le type de g est polymorphe paramétrique :
sa variable de type ρ, peut être instanciée en plus de méthodes.

Types fermés = types monomorphes

```
let origin(p :point) = p#getx = 0 ; ;
val origin : point -> bool = <fun>
```

$$\texttt{origin} : \ < getx : int; move : int \rightarrow unit > \rightarrow bool$$

Ce type est *fermé* $\Rightarrow$ *sans variable de type de rangée*!

$\Rightarrow$ il ne peut pas être instancié avec plus de composantes.
$\Rightarrow$ un type sans variables de types est monomorphe!

```
# let cp = new pointColore(3, "rouge") in origin(cp) ; ;
This expression has type
  pointColore =
    < getx : int ; move : int -> unit ; getcolor : string ;
      setcolor : string -> unit >
but is here used with type point = < getx : int ; move : int -
```

Là où un objet de type fermé est attendu, on ne peut donner qu'un objet qui possède exactement la même interface.

Abréviations de types fermés et ouverts

Toute déclaration de classe :

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

produit **deux nouvelles abréviations** de types dans l'environnement de typage :

$$\texttt{point} =< \texttt{getx} : \texttt{int}; \texttt{move} : \texttt{int} \to \texttt{unit} >$$
$$\#\texttt{point} =< \texttt{getx} : \texttt{int}; \texttt{move} : \texttt{int} \to \texttt{unit}; \rho >$$

`point` (fermé) = type des objets avec *exactement* les méthodes et types de la classe `point`

`#point` (ouvert) = type des objets possédant *au moins les méthodes et types* de la classe `point`

Utilisation d'une abréviation ouverte

Réécrivons `origin` à l'aide de `#point` :

```
# let origin (p : #point) = p#getx = 0 ; ;
val origin : #point -> bool = <fun>
```

Nous pouvons maintenant l'appliquer aux points colorés :

```
# let p = new point 2 ; ;
val p : point = <obj>

# origin (p) ; ;
- : bool = false

# origin (cp) ; ;
- : bool = false
```

Sous-typage sur les objets en Ocaml

Polymorphisme objet d'Ocaml $\neq$ sous-typage.

$\Rightarrow$ les types `point` et `pointColore`
sont incompatibles

1. `pointColore` ne peut pas être employé là où l'on attend exactement un `point`.

```
# let origin(p :point) = p#getx = 0 in origin(pc) ; ;
This expression has type pointColore but is here used
with type point
Only the first object type has a method get_couleur
```

2. On ne peut pas mélanger `pointColore` et `point`

```
# let l = [p ;pc] ; ;
This expression has type pointColore but is here
used with type point
Only the first object type has a method get_couleur
```

Résoudre l'incompatibilité (1)

1. `pointColore` ne peut pas être employé là où l'on attend exactement un `point`.

```
# let origin(p :point) = p#getx = 0 in origin(pc) ; ;
This expression has type pointColore but is here used
with type point
Only the first object type has a method get_couleur
```

Solution (déjà vue) : introduire du polymorphisme paramétrique en utilisant les types ouverts .

```
# let origin(p : #point) = p#getx = 0 in origin(pc) ; ;
- : false = bool
```

Résoudre l'incompatibilité (2)

(2) on ne peut pas mélanger `pointColore` et `point` :

```
# let l = [p ;pc] ; ;
This expression has type pointColore but is here
used with type point
Only the first object type has a method get_couleur
```

Comme avant, ce mélange est interdit. Pour instancier le type des listes 'a list, il faudrait :

- d'après p : 'a $\mapsto$ point
- d'après t : 'a $\mapsto$ pointColore
- point $\neq$ pointColore, car ce sont deux structures différentes.

Ici, nous aimerions voir `pc : pointColore` *comme s'il était de type* `point`.

$\Rightarrow$ C'est du sous-typage !

Conversion (vers un sûr-type)

En Ocaml, il est possible de faire du sous-typage, mais seulement de manière *explicite*, via une *conversion de types*.

On peut explicitement convertir un `pointColore` vers un `point`, en *utilisant le sous-typage*.

On parle aussi de **coercion**.

Exemple : un `point` et un `pointColore` dans une liste.

```
# let cp' = (cp :pointColore :> point) ; ;

# let l = [p ;cp'] ; ;
```

Conversion (suite)

La notation (`cp : pointColore :> point`) dit au compilateur de vérifier :

1. que le type de `cp` est `pointColore`,

2. que `pointColore` *est un sous-type* de `point` (qu'il peut être vu comme)

3. puis de *ne voir* `cp` *qu'à travers l'interface de* `point`. Autrement dit, seules les méthodes de `point` sont visibles dans `cp'`.

`cp : (pointColore :>point)` est une *conversion*

Conversion par sûr-typage = perte d'information

```
let cp' = (cp : pointColore :>point)
```

Le type *cible* de cette conversion est `point`. Il contient moins d'information que `pointColore`.

Cette conversion entraîne une **perte d'information** : on ne verra de `cp'` que les méthodes de `point`.

```
# cp'#get_couleur ; ;
This expression has no method get_couleur
```

Conversion = changement du type

Attention : une coercion ne change que le type d'un objet, pas sa valeur dynamique !

```
class a =
object
 method m = 1
end

class b =
object
  inherit a
  method m = 2
  method n = 3
end;;

let x = new a;;
let y = new b;;
let z = (y : b :> a);;
```

Coercion = seul le type change !

```
let x = new a ; ;
let y = new b ; ;
let z = (y : b :> a) ; ;
```

Dans x, m renvoie 1, alors que dans y, m est redéfini et renvoie 2.

```
# x#m ; ;
- : int = 1

# y#m ; ;
- : int = 2

# y#n ; ;
- : int = 3
```

z *contient exactement la même valeur* que y, et donc, *contient toujours* la méthode n. Mais le changement de type de z rend inaccessible n.

```
#  z#m ; ;
- : int = 2

# z#n ; ;
This expression has type a
It has no method n
```

Pas de *down-cast* en Ocaml

Contrairement à Java, il n'est pas possible de spécifier la conversion d'un objet d'un type t vers un type t' plus petit (par exemple de `Object` vers `point`).

Ce genre de conversion est :

- *non sûre* : elle peut produire des *erreur de typage* à l'exécution.
- *coûteuse* : on doit ajouter des test de types dans le code compilé.

Le compilateur Ocaml réjète ces conversions :

```
# (pc :> point) ; ;
- : point = <obj>

# (p :> pointColore) ; ;
This expression cannot be coerced to type
  pointColore =
  < get_couleur : string ; getx : int ; move : int -> unit ; print : unit >
it has type point but is here used with type #pointColore
Only the second object type has a method get_couleur
```

Coercion dans une fonction

Exemple 2 : dans la fonction `origin`, on veut accepter en argument n'importe quel objet sous-type des points :

```
# let origin p = (p :pointColore :> point)#getx = 0 ; ;
val origin : pointColore -> bool = <fun>

# let cp = new pointColore (3, "rouge") ; ;

# origin (cp) ; ;
- : false = bool
```

- Le typeur vérifie que `pointColore` est bien un sous-type de `point`.
- `origin` prend un `pointColore` qui est ensuite convertit vers un `point`.

$\Rightarrow$ `origin` : `pointColore` $\rightarrow$ `bool`

$\Rightarrow$ Le type inféré est fermé.

Notation abrégée pour la coercion

Une notation plus compacte, permet de ne donner que le type cible de la conversion. Le type inféré est plus général.

```
# let origin p = (p :> point)#getx = 0 ; ;
val origin : < getx : int ; move : int -> unit ; .. > -> bool =
```

On peut donner en argument n'importe quel sous-type de `point`, et pas seulement un `pointColore`.

```
[ (new point 3) ;
  (new pointColore (2,''bleu'') :> point) ;
  (new pointXY 4 7 :> point)
] ; ;

- : point list
```

On suppose que la classe `pointXY` est un sous-type de `point`.

Classes paramétrées par des types

Ce sont des classes paramétrées par des *variable de type*, à la manière des types polymorphes de ML : les listes polymorphes, les tableaux polymorphes, etc.

La généricité de Java s'inspire des classes paramétrées Ocaml.

Le *paramètre de type* est une variable de type `'a`, qui précède le nom de la classe comme dans `('a list)`.

```
class ['a] cellule =
object
   ....
end
```

Classes paramétrées par des types

Exemple : Cellules paramétrées par le type `'a` de leur contenu.

```
#class ['a] cellule (init : 'a) =
 object
   val mutable cont = init
   method get = cont
   method set d = cont <-d
 end ; ;
```

L'entête de la classe pose `'a` comme paramètre de type. La valeur d'initialisation `init` doit aussi être de type `'a`.

Le type inféré pour cette classe est :

```
class ['a] cellulle :
  'a ->
  object
    val mutable cont : 'a
    method get : 'a
    method set : 'a -> unit
end
```

Créer un objet à partir d'une classe paramétrée

Pour définir une cellule, nous donnons une valeur d'initialisation.

Elle servira à instancier le type du contenu des cellules :

```
# let i = new cellule 1 ; ;
val i : int cellule = <obj>

# let b = new cellule true ; ;
val b : bool cellule = <obj>
```

Instancier un type polymorphe

En Ocaml, une liste est paramétrée par le type de ces éléments. Son type est `'a list`. Lorsque nous écrivons :

```
# [1 ;2 ;3] ; ;
- : int list = [1 ; 2 ; 3]
```

Le compilateur *instancie* ce type en remplaçcant la variable de type `'a` $\mapsto$ `int` dans le type `'a list`.

$\Rightarrow$ Même mécanisme pour les classes paramétrées :

```
class ['a] cellule = ...

# let i = new cellule 1 ; ;
val i : int cellule = <obj>

# let b = new cellule true ; ;
val b : bool cellule = <obj>
```

- pour `i`, `'a` $\mapsto$ `int` $\Rightarrow$ `int cellule`,
- pour `b`,`'a` $\mapsto$ `bool` $\Rightarrow$ `bool cellule`.

Mélange de cellules ?

Le même phénomène est en jeu si on mélange de cellules avec des contenus incompatibles.

```
# [(new cellule 1); (new cellule true)];;
                      ^^^^^^^^^^^^^^^^
This expression has type bool cellule but is here
used with type int cellule
Types for method get are incompatible
```

Le type d'une liste est : `'a list`

- D'après `(new cellule 1)` : $'a \mapsto$ `int cellule`
- D'après `(new cellule true)` : $'a \mapsto$ `bool cellule`
- `int cellule` $\neq$ `bool cellule`

Le contenu d'une cellule conditionne son type. Ces contenus incompatibles empêchent le mélange.

Héritage d'une classe paramétrée

Pour hériter d'une classe paramétrée, nous devons fournir la manière d'instancier les paramètres de type :

```
inherit [int] cellule
```

Indique que la variable de type `'a` est instanciée en `int`.

Héritier pour instancier le paramètre de type

Exemple 1 : Les cellules d'entiers sont tout simplement une spécialisation des `['a] cellules` avec `'a` $\mapsto$ `int`.

```
# class intCellule init =
object
   inherit [int] cellule init
end ; ;

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

let cint = new intCellule 3 ; ;
val cint : intCellule = <obj>

# cint#get ; ;
- : int = 3
```

Hériter pour instancier le type (suite)

Peut-on mélanger intCellule et int cellule ?

```
# [(new intCellule 3) ; (new cellule 4)] ; ;
- : intCellule list = [<obj> ; <obj>]
```

Oui, car les types des deux objets sont identiques :

```
class ['a] cellule :  'a ->
object
  val mutable cont : 'a
  method get : 'a
  method set : 'a -> unit
end
```

dans `(new intCellule 3)`, 'a $\mapsto$ int, ce qui donne à cet objet le type :

$$(\texttt{new intCellule3}) : < \texttt{get} : \texttt{int}; \texttt{set} : \texttt{unit} \rightarrow \texttt{unit} >$$

qui est exactement le type d'une instance de intCellule :

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

Héritage pour spécialiser la classe

Exemple 2 : la classe des cellules avec sauvegarde. On souhaite ajouter des fonctionnalités de sauvegarde, mais maintenir le caractère polymorphe du contenu.

```
# class ['a] backupCellule (init : 'a) =
object
   inherit ['a] cellule init as super
   val mutable backup = init
   method  set n = backup <- cont ; super#set n ;
   method restore = cont <- backup
end ; ;

class ['a] backupCellule :
  'a ->
  object
    val mutable backup : 'a
    val mutable cont : 'a
    method get : 'a
    method restore : unit
    method set : 'a -> unit
  end

let bi = new backupCellule 3 in
         bi#set 4 ; bi#restore ; bi#get ; ;
- : int = 3
```

Héritage pour spécialiser la classe (suite)

Peut-on mélanger intCellule et int backupCellule ?

```
# [(new intCellule 3); (new backupCellule 4)];;
                        ^^^^^^^^^^^^^^^^^^^^
This expression has type int backupCellule but is here
used with type   intCellule
Only the first object type has a method restore
```

Non, car ces deux types ont des structures différentes : `backupCellule` possède une méthode supplémentaire restore.

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

class ['a] backupCellule :  'a ->
  object
    val mutable backup : 'a
    val mutable cont : 'a
    method get : 'a
    method restore : unit
    method set : 'a -> unit
  end
```

Fonction polymorphe sur les cellules

```
# let rec cell_mem x (l :'a cellule list) =
      match l with
      [] -> false
   | (a : :r) -> if a#get = x then true else cell_mem x r ; ;
val cell_mem : 'a -> 'a cellule list -> bool = <fun>
```

Nous l'appliquons sur différents types de cellules :

```
let l1 = [new cellule 1 ; new cellule 7] ; ;
val l1 : int cellule list = [<obj> ; <obj>]

# let l2 = [new cellule "a" ; new cellule "bc"] ; ;
val l2 : string cellule list = [<obj> ; <obj>]

#  cell_mem 7 l1 ; ;
- : bool = true

# cell_mem "c" l2 ; ;
- : bool = false

# cell_mem 2 l2 ; ;
This expression has type string cellule list but is here used with type
  (int #cellule as 'a) list
Type string cellule = < get : string ; set : string -> unit >
is not compatible with type 'a = < get : int ; set : int -> unit ; .. >
Types for method get are incompatible
```

Fonction doublement polymorphe

```
# let rec cell_mem2 x (l :'a #cellule list) =
      match l with
      [] -> false
   | (a : :r) -> if a#get = x then true else cell_mem2 x r ; ;
val cell_mem2 : 'a -> 'a #celulle list -> bool = <fun>
```

La contrainte de type

`(l :'a #cellule list)`

sur la liste des cellules l, nous dit qu'elle est doublement polymorphe :

- dans le contenu `'a` des cellules,
- dans la forme ouverte `#cellule` des cellules (avec possiblement des méthodes).

Nous pouvons appliquer `cell_mem2` sur des listes de `intCellule`, ou sur des listes de `bool backupCellule`, etc.

```
# cell_mem2 1 [new intCellule 1] ; ;
- : bool = true

# cell_mem2 "ab" [new backupCellule "a"] ; ;
- : bool = false
```

2 niveaux de polymorphisme

```
# let rec cell_mem1 x (l :'a cellule list) =
      match l with
      [] -> false
   | (a : :r) -> ...
val cell_mem1 : 'a -> 'a celulle list -> bool = <fun>

# let rec cell_mem2 x (l :'a #cellule list) =
      match l with
      [] -> false
   | (a : :r) -> ...
val cell_mem2 : 'a -> 'a #celulle list -> bool = <fun>
```

- `cell_mem1` : $'a \rightarrow$ `'a celulle list` $\rightarrow$ `bool`
  polymorphe **uniquement** sur le type du contenu des cellules.

- `cell_mem2` : $'a \rightarrow$ `'a #celulle list` $\rightarrow$ `bool`
  polymorphe sur le type du contenu
  et sur la possibilité d'avoir plus de composantes.

Contraintes sur classes paramétrées

```
#class ['a] compteur (x_init :'a) =
   object
     val mutable x = x_init
     method get = x
     method set y = x <- y
     method compter =  x <- x + 1
   end ; ;
```

Malgré la variable de type `['a']` dans l'entête, il est clair à `'a` ne peut être instancié qu'en `int`.

La contrainte `constraint 'a = int` est inférée par le typeur et ajoutée dans le type de la classe :

```
class ['a] compteur :
  'a ->
  object
    constraint 'a = int
    val mutable x : 'a
    method compter : unit
    method get : 'a
    method set : 'a -> unit
  end
```

Contraintes sur classes paramétrées

```
class ['a] compteur :
  'a ->
  object
    constraint 'a = int
    val mutable x : 'a
    method compter : unit
    method get : 'a
    method set : 'a -> unit
  end

# let c = new compteur 0 ; ;
val c : int compteur = <obj>
```

Toute tentative de définir un compteur sur autre chose que des entiers, provoque une erreur de typage.

```
# let d = new compteur "ab" ; ;
This expression has type string but is here
used with type int
```

Classes paramétrées par des d'autres classes, contraintes explicites

L'exemple suivant montre :

- qu'on peut paramétrer une classe par le type d'une autre classe.

- qu'on peut poser des contraintes **explicites**.

Exemple : cercles dont le centre est au moins un point.

```
#class ['a] cercle (c : 'a) =
   object
     constraint 'a = #point
     val mutable center = c
     method center = center
     method set_center c = center <- c
     method move = center#move
   end ; ;

# let circ1 = new cercle (new point 2) ; ;
val circ1 : point cercle = <obj>
```

Classes paramétrées par des d'autres classes(suite)

Le typeur infère la même contrainte que celle explicitement posée :

```
#class ['a] cercle (c : 'a) =
   object
     constraint 'a = #point
     val mutable center = c
     method center = center
     method set_center c = center <- c
     method move = center#move
   end ; ;

class ['a] cercle :
  'a ->
  object
    constraint 'a = #point
    val mutable center : 'a
    method center : 'a
    method move : int -> unit
    method set_center : 'a -> unit
  end
```

Contraintes implicites ⇒ types ouverts

On aurait pu écrire `cercle` sans poser de contrainte explicite.

```
#class ['a] cercle (c : 'a) =
   object
     val mutable center = c
     method center = center
     method set_center c = center <- c
     method move = (center#move : int -> unit)
   end ; ;

class ['a] cercle :  'a ->
  object
    constraint 'a = < move : int -> unit ; .. >
    ...
  end
```

Le typeur déduit la contrainte :

```
constraint 'a = < move : int -> unit ; .. >
```

qui dit : "`'a` doit être un objet possédant au moins une méthode `move : int -> unit`".

Cercles colorés

Les cercles colorés, ayant pour centre un point coloré. On donne une contrainte explicite :

```
#class ['a] cercleColore c =
   object
     constraint 'a = #pointColore
     inherit ['a] cercle c
     method color = center#get_couleur
   end ; ;


class ['a] cercleColore :
  'a ->
  object
    constraint 'a = #pointColore
    val mutable center : 'a
    method center : 'a
    method color : string
    method move : int -> unit
    method set_center : 'a -> unit
  end
```

Self

– Pendant l'exécution d'une méthode, Self désigne l'objet qui a appelé cette méthode.

– En Java on le désigne par `this`. En Ocaml, on doit déclarer en tête de la classe un identificateur pour le désigner

– "self" **peut représenter un objet de la classe mais aussi d'une sous classe**.

– Envoyer un message à self c'est envoyer un message à l'objet en train de s'exécuter, ce qui réalise la récursion (appel à lui-même).

– Parce que cette méthode est prise dans l'objet qui s'exécute, et non dans la classe, *un message à self effectue une liaison tardive.*

Classes avec le type de Self

Ce sont des classes dont les méthodes prennent en argument ou renvoient en résultat, un objet de même type que Self.

```
class c =
object (self)
   method m = self
end
```

```
class c :
object ('a)
  method m : 'a
end
```

Le type inféré pour c utilise une variable `'a`, mais **ce n'est pas une variable de type**.

`'a` est une abréviation qui désigne *"le type de Self pour la classe courante"*.

Le type de self en Ocaml

Nous notons *"le type de Self pour la classe c"* par

$$Type_{self}(c) = \texttt{'a}$$

```
class c =
object (self)
   method m = self
end

class c :
object ('a)
  method m : 'a
end
```

On peut réécrire ce type par

$$\begin{array}{lll} Type_{self}(c) & = & < \texttt{m} : Type_{self}(c); .. > \\ Type_{self}(c) & = \texttt{'a} = & < \texttt{m} : \texttt{'a}; .. > \end{array}$$

Il s'agit d'un **type récursif** :

le type (`'a`) apparaît dans sa propre définition (via le type de la méthode `m`).

Comment typer Self ?

Dans une classe, le type de self doit contenir :

- toutes les méthodes de la classe,
- mais il doit aussi, pouvoir s'enrichir, **dans une sous-classe**, de toutes les méthodes ajoutées par cette sous-classe.

Si lors de l'héritage, le type de Self ne s'enrichit des nouvelles méthodes, il aura, dans la sous-classe, le type de la classe parente et non celui de la classe que courante !

Ce ne sera donc pas le type de Self !

Exemple d'héritage de Self

classe c : possède une méthode `m` qui renvoie self.

```
class c =
object (self)
   method m = self
end
```

classe `cd` : { hérite de la méthode `m` (avec le type self de c)
ajoute une nouvelle méthode `n : int`

```
class cd =
object (self)
   inherit c
   method n = 2
end
```

Le code :

```
# let ocd = new cd ; ;
# let est_ce_une_erreur = (ocd#m)#n ????
```

1. Doit-il s'exécuter correctement ?
2. Est-il correctement typé ?

Exemple d'héritage de Self (suite)

```
class c =
object (self)
   method m = self
end

class cd =
object (self)
   inherit c
   method n = 2
end
```

Le code :

```
# let ocd = new cd ; ;
# let est_ce_une_erreur = (ocd#m)#n ????
```

1. Doit-il s'exécuter correctement ?

   Oui, car l'utilisation de self **doit** se traduire par une liaison (tardive) sur l'objet courant `ocd` qui est une instance de cd, et qui **contient bien** une méthode `n`.

   Donc, l'appel doit s'exécuter correctement et renvoyer 2 en résultat.

Exemple d'héritage de Self en Ocaml

Ce code :

```
# let ocd = new cd ; ;
# let est_ce_une_erreur = (ocd#m)#n ????
```

Est-il correctement typé ? ⇒ Ca dépend du langage . . .

En Ocaml, il est bien typé et s'exécute correctement.

```
# let ocd = new cd ; ;
# let est_ce_une_erreur = (ocd#m)#n ; ;
- : int = 2
```

Exemple d'héritage de Self en Java

En java, le type de self (this) est celui de la classe courante, et il est *transmis tel quel lors de l'héritage*.

⇒ Une méthode qui retourne `this`, garde le type de la classe parente dans une sous-classe.

```
class C {
    C () { }
    C m() { return this ; }
}
class CD extends C {
    CD () { }
    int n() { return 2 ; }
}
class Error {
    static CD ocd = new CD() ;
    static int x = (ocd.m()).n() ;
}
%javac Error.java
Error.java :3 : cannot resolve symbol
symbol  : method n ()
location : class C
    static int x = (ocd.m()).n() ;
                    ^
1 error
```

Java ne réussit pas à traiter la **liaison tardive** de self. En cause : un typage trop restrictif !

Héritage de Self en Java (suite)

```
class Error {
    static CD ocd = new CD() ;
    static int x = (ocd.m()).n() ;
}
%javac Error.java
Error.java :3 : cannot resolve symbol
symbol  : method n ()
location : class C
    static int x = (ocd.m()).n() ;
                    ^
1 error
```

Lors de l'héritage, la méthode m de CD garde le type de self de la classe parente :

$$\mathtt{m}_{CD} : \mathtt{C}$$

`(ocd.m())` renvoie this, autrement dit, renvoie `ocd`.

Or, `(ocd.m())` est une instance de `OCD`, et donc contient bien la méthode n. `(ocd.m()).n` devrait réussir.

Mais, le typage, ne voit que le type C hérité pour m, et donc, ne voit pas la méthode n !

Héritage de Self en Java (fin)

Si l'on veut recuperer un typage correcte pour self, on devra faire une conversion explicite.

Le programme où l'on fait

```
int x = ((CD) (ocd.m())).n() ;
```

au lieu de

```
int x = (ocd.m()).n() ;
```

non seulement réussit à la compilation, mais s'exécute correctement.

```
public class TestC{
    public static void main (String arg[]) {
        CD ocd = new CD() ;
        int x = ((CD) (ocd.m())).n() ;
        System.out.println(x) ;

    }
}
javac TestC.java
java TestC
2
```

Ceci montre que ocd contient bien la méthode n, mais qu'elle était cachée par un typage trop restrictif de self.

Le type de self en Ocaml(suite)

En Ocaml, le type de self est polymorphe ; il contient une variable de rangée ρ.

Lorsque le type de self est hérité, cette variable est instanciée par toutes les méthodes nouvelles de la sous-classe.

$$Type_{self}(c) \ = \ < \texttt{m} : Type_{self}(c); \rho_1 >$$

Contraintes pour typer `cd` en tant qu'extension de `c` :

1. Le type de `cd` contient les méthodes ajoutées (`n`), et possiblement d'autres (variable de rangée ρ_2) :

$$Type_{self}(cd) \ = < \texttt{n} : \texttt{int}; \rho_2 >$$

2. Il contient également les méthodes de `c`. De plus, le type de self de `c` doit, au sein de `cd`, pouvoir devenir **identique** au type de self de `cd` :

$$Type_{self}(c) = Type_{self}(cd)$$

Le type de self en Ocaml(suite)

$$\begin{aligned} Type_{self}(c) &= < \texttt{m} : Type_{self}(c); \rho_1 > \\ Type_{self}(cd) &= < \texttt{n} : \texttt{int}; \rho_2 > \end{aligned}$$

Contrainte : $Type_{self}(c) = Type_{self}(cd)$

Cela n'est possible que si :

$$\begin{aligned} \rho_1 &\mapsto < \texttt{n} : \texttt{int}; \rho_3 > \\ \rho_2 &\mapsto < \texttt{m} : Type_{self}(cd); \rho_3 > \end{aligned}$$

Reemplaçons ρ_1 et ρ_2 dans :

$$\begin{aligned} Type_{self}(cd) &= < \texttt{n} : \texttt{int}; \rho_2 > \\ \Rightarrow &\quad < \texttt{n} : \texttt{int}; \texttt{m} : Type_{self}(cd); \rho_3 > \end{aligned}$$

$$\begin{aligned} Type_{self}(c) &= < \texttt{m} : Type_{self}(c); \rho_1 > \\ \Rightarrow &\quad < \texttt{m} : Type_{self}(c); \texttt{n} : \texttt{int}; \rho_3 > \end{aligned}$$

Après héritage, le type qui avait self dans m (extensible grâce à la variable de rangée ρ_1), est étendu par toutes les méthodes de ocd.

self dans le type hérité de m, correspondra bien au type de la sous-classe et non à celui de la classe parente.

Sous-typage

Qu'est-ce que c'est ?

C'est une forme de polymorphisme obtenue par *affaiblissement* de l'information des types (perte d'information), qui permet de voir deux types différents comme étant de même type.

Plusieurs approches :

Sous-typage structurel Les types sont construits à partir de types de base. Leur compatibilité et la relation de sous-typage ne dépend que de la structure des types.

Sous-typage nominal Les types sont vus comme des atomes (leurs noms), et la relation de sous-typage est construite au fur et à mesure des nouvelles déclarations. Les règles de typage structurel entre les classes ne sont pas nécessairement toutes respectées (c'est le cas de la majorité des langages).

Intuition

On dit que B est un *sous-type* de A, noté

$$B \preceq A$$

si toute valeur de type B peut être employée *sans provoquer d'erreur de typage*, dans un contexte où une expression de type A est attendue.

"Règle de Remplacement" :
$B \preceq A$ si *une valeur de type B peut remplacer une valeur de type A sans danger pour le typage.*

Intuition de sous-typage structurel

Sous-typage en largeur : Un objet avec plus de méthodes peut toujours être utilisée à la place d'un objet avec moins de méthodes. On peut "oublier" les méthodes "en trop" sans risque d'erreurs : les méthodes attendues sont présentes.

La relation de sous-typage correspond à l'inclusion des ensembles des méthodes pour chaque type d'objet

Sous-typage en profondeur : Si un élément de type A peut être utilisé comme un élément de type B, alors :

• une liste d'éléments de type A peut aussi être utilisée comme une liste d'éléments de types B.

• un objet avec une méthode m de type A, peut être utilisée comme un objet avec une méthode m de type B.

Sous-typage en largeur : inclusion de méthodes (approximation)

Une valeur de type B peut remplacer une valeur de type A, si B *possède au moins toutes les méthodes de* A.

Notons $Meth(t)$ l'ensemble des méthodes définies sur le type t. On peut reformuler la définition précédente par :

$$B \preceq A \quad \textbf{ssi} \quad Meth(A) \subseteq Meth(B) \qquad (Inclusion)$$

Règle de “remplacement” de sous-types

– si $B \preceq A$

– et $b : B$

– alors $b : A$

autrement dit, si $b : B$ et que B est un sous-type de A, alors b peut être vu de deux manières (polymorphisme) :

– comme un objet de type B,
– mais *aussi* comme un objet de type A.

Remarque : Cette règle nous dit que si $B \preceq A$, le type de B peut être remplacé par celui de A !

Polymorphisme, compatibilité

Le sous-typage est une forme de polymorphisme : deux types différents sont vus comme étant *compatibles*.

Compatibilité entre types : si deux types sont égaux, ou si l'un peut être *utilisé* à la place de l'autre.

Pourquoi le sous-typage est sûr ?

Supposons :

- une classe A avec une méthode M_A de type T,
- une fonction $f(x : A) = x.M_A$
- Son type est donc $f : A \to T$,

Par ailleurs, on suppose qu'une autre classe B est telle que $B \preceq A$, et un objet $b : B$,

L'appel $f(b)$ est-il correctement typé ?

Oui, car

- par la règle de "remplacement",
  si $b : B$ et $B \preceq A \Longrightarrow b : A$,

- si $b : A$, et sachant que $f : A \to T$,
  alors $f(b)$ est bien typé.

Le sous-typage est un typage sûr

Le fait de passer $b : B$ à f, qui attend un argument de type A, ne produit pas d'erreurs à l'exécution :

- en effet, $B \preceq A$
  $\Rightarrow Meth(A) \subseteq Meth(B)$
  $\Rightarrow M_A \in Meth(B)$,

- donc M_A est une méthode de b,

- et l'exécution du code de f ne produira pas d'erreur :
  $f(b) \Rightarrow (x.M_A \text{ où } x = b) \Rightarrow b.M_A$.

Le comportément de f, qu'il soit appelé avec une valeur de type A, ou avec une valeur de type B, est *sûr* du point de vue du typage.

Sous-typage en largeur (Ocaml)

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

class pointColore (i,c) =
object
 inherit point i as super
 val c = c
 method get_couleur = c
 method print =  super#print ;
   print_string (", "^c)
end
```

En Ocaml, peut-on dire : `pointColore` $\preceq$ `point` ?

Exemple de sous-typage en largeur (Ocaml)

`pointColore` possède plus de méthodes que `point` : $Meth(\texttt{point}) \subseteq Meth(\texttt{colorpoint})$.

Nous pouvons demander au compilateur Ocaml de vérifier que `pointColore` $\preceq$ `point`.

```
# let cp1 = new pointColore (1, "Rouge") ; ;

# (cp1 : pointColore :> point) ; ;
- : point = <obj>
```

La relation inverse est fausse : `point` n'est pas un sous-type de `pointColore` :

```
# ((new point 1) : point :> pointColore) ; ;
Type point = < getx : int ; move : int -> unit ; print : unit >
is not a subtype of type
  pointColore =
< get_couleur : string ; getx : int ;
  move : int -> unit ; print : unit >
```

Point en Java

```
class Point {
    int x ;
    Point (int x0) { x = x0 ; }
    int getx(){ return x ;}
    void move (int d) { x = x+d ; }
    void print () {System.out.println(this.getx()) ;}
}
```

```
class PointColore extends Point {
    String c ;
    PointColore (int x0, String c0) { super(x0) ; c = c0 ; }
    String getCouleur () { return c ; }
    void print () {
        System.out.println(this.getx() + " , ") ;
        System.out.println(this.getCouleur()) ;
    }
}
```

En Java, peut-on dire : `PointColore` $\preceq$ `Point` ?

Point en Java (suite)

Pour tester le sous-typage, la classe `TestPoint` déclare une variable p de type Point. Il suffit de lui donner une valeur pointColore et voir si cela compile.

```
public class TestPoint{
    public static void main (String arg[]) {
        Point p;
        PointColore q = new PointColore (2,"Rouge");
        p = q;
        System.out.println(p.getx());
    }
}
% javac TestPoint.java
% java TestPoint
2
```

q : PointColore peut être vu comme un Point. Pour Java, PointColore est bien un sous-type de Point.

Sous-typage en profondeur

Lorsque un type d'objet contient des méthodes avec *des types d'objets en argument ou en résultat*, le sous-typage doit considérer :

- non seulement que les méthodes de l'objet sont présentes dans le sous-type,
- mais aussi que les types de ces méthodes sont en relation de sous-typage dans les deux objets.

Supposons deux classes a et b avec les interfaces :

```
class a :
object
  method m : point
end
```

```
 class b :
object
  method m : pointColore
end
```

Peut-on dire $\mathtt{b} \preceq \mathtt{a}$?

$\Rightarrow$ Il faut vérifier que le type de m dans b est un sous-type de celui de m dans a :

$$\begin{array}{lll} \mathtt{b} \preceq \mathtt{a} & \text{ssi} & Type(\mathtt{m_b}) \preceq Type(\mathtt{m_a}) \\ & \text{ssi} & \mathtt{pointColore} \preceq \mathtt{point} \end{array}$$

Définition du sous-typage structurel ($t \preceq t'$)

Soient t et t' deux types d'objets :

- $t =< m_1 : \tau_1; \ldots; m_n : \tau_n >$
- $t' =< m_1 : \tau'_1; \ldots; m_n : \tau'_n; \ldots; m_k : \tau'_k >, k \geq n$

alors t' est un sous-type de t (noté $t' \preceq t$)

ssi

- pour tout i dans [1..n], on a $\tau'_i \preceq \tau_i$

□

Cette définition englobe le typage en largeur, et en profondeur :

- Le sous-type t' contient au moins les mêmes noms des méthodes que le super-type t (car $k \geq n$).

- Pour toutes ces méthodes, il est demandé que leurs types soient en relation de sous-typage : $\tau'_i \preceq \tau_i$

Sous-typage en profondeur (Ocaml)

b est un sous-type de a ?

```
class a =
object
  method m = new point 1
end
```

```
class b =
object
  method m = new pointColore (1, "Vert")
end
```

Oui, car pour la seule méthode commune, son type dans chaque classe est :

$$\mathtt{m}_a : \mathtt{point}$$
$$\mathtt{m}_b : \mathtt{pointColore}$$

et nous savons déjà que $\mathtt{point} \preceq \mathtt{pointColore}$.

Nous demandons au typeur de vérifier ceci :

```
# let ob = (new b) in (ob : b :> a) ; ;
- : a = <obj>
```

Sous-typage en profondeur (Java)

Java réalise du sous-typage en profondeur :

```
class A {
    A () { }
    Point m() { return new Point(1) ; }
}

class B extends A {
    B () { }
    PointColore m() { return new PointColore(2,"Vert") ; }
}

class TestB{
    public static void main(String [] args){
        B b = new B() ;
        Point c = b.m() ;
        System.out.println(c.getx()) ;
    }
}
```

Ce programme est compilé et exécuté correctement.

```
% javac TestB.java
% java TestB
2
```

Sous-typage de classes paramétrées

Les cercles paramétrés par le type de points (du centre).

```
class ['a] cercle (c : 'a) =
   object
     constraint 'a = #point
     val mutable center = c
     method center = center
     method move = (center#move : int -> unit)
   end

# let cpc = new cercle (new pointColore (2, "Rouge")) ; ;
val cpc : pointColore cercle = <obj>

# (cpc : pointColore cercle :> point cercle) ; ;
- : point cercle = <obj>
```

`pointColore cercle` $\preceq$ `point cercle` :

Ces deux types d'objets ont les mêmes méthodes, dont seule `center` change de type :

$$\texttt{center}_{(\texttt{pointColore cercle})} : \texttt{pointColore}$$
$$\texttt{center}_{(\texttt{point cercle})} : \texttt{point}$$

Et nous avons `pointColore` $\preceq$ `point`.

Sous-typage $\neq$ héritage

(a) Sous-typage n'implique pas héritage

• En Ocaml, pour décider du sous-typage on examine les structures des types : le sous-typage ne dépend pas de la façon dont les classes sont construites (i.e : de la clause `inherits`. Deux types d'objet peuvent être de sous-types sans que l'un hérite de l'autre. Par exemple, en Ocaml, `b` n'hérite pas de `a`, mais `b` $\preceq$ `a` :

```
class a =
object
  method m = new point 1
end

class b =
object
  method m = new pointColore(1, "Vert")
end
```

```
class a :
object
   method m : point
end

class b :
object
  method m : pointColore
end
```

• En java, une clause `implement` crée une relation de sous-typage sans réaliser d'héritage. Dans ce cas précis, Java compare les structures des types pour vérifier le bien fondé du sous-typage entre classe et interface.

Sous-typage ≠ héritage (suite)

(a) Héritage n'implique pas sous-typage

Avec du sous-typage structurel, une classe A avec une *méthode binaire m, n'admet aucun sous-type*.

Si une autre classe C hérite de A, et donc de m, C ne sera pas un sous-type de A !!

Qu'est-ce qu'une méthode binaire ?

C'est une méthode qui prend un argument du même type que self.

Les méthodes binaires

Une méthode binaire est une méthode qui combine self avec un argument du même type que self.

```
class point x0 =
  object (self : 'a)
    val x = abs x0
    method getx = x
    method equal(p : 'a) = x = p#getx
  end

class point :
  int ->
  object ('a)
    val x : int
    method equal : 'a -> bool
    method getx : int
end
```

`equal` est une méthode binaire : elle compare l'objet courant (self) avec un objet extérieur de même type.

`equal` : $Type_{self}$(`point`) $\rightarrow$ `bool`

Exemples de méthodes binaires

- Addition, soustraction, maximum, minimum, etc.
- Concaténation de chaînes, de listes.
- Opérations sur les ensembles (union, intersection).

Plus généralement, toutes les opérations binaires sur les types données implémentés dans un style objet...

Difficulté des méthodes binaires :

- Il est difficile de leur préserver un type correct dans une sous-classe.
- Le type des objets avec une méthode binaire n'admet pas de sous-type (tant que la méthode binaire reste visible).

Solutions

La plus répandue consiste à ignorer le problème, et donc à ne pas hériter correctement des méthodes binaires (Java).

Pour résoudre ce problème, il faut utiliser un système de type sophistiqué (Ocaml).

Une autre solution consiste à sortir la méthode binaire de la classe.

Héritage $\not\Rightarrow$ sous-typage

Supposons :

– `point` possède une méthode binaire de type

$$\texttt{equal}_{\texttt{point}} : \texttt{point} \rightarrow \texttt{bool}$$

– `pointCouleur` hérite de cette méthode, qui devient de type :

$$\texttt{equal}_{\texttt{pointCouleur}} : \texttt{pointCouleur} \rightarrow \texttt{bool}$$

On peut montrer (par sous-typage structurel) :

$$\begin{array}{rccc} & \texttt{pointCouleur} \rightarrow \texttt{bool} & \not\leq & \texttt{point} \rightarrow \texttt{bool} \\ \Rightarrow & \mathit{Type}(\texttt{equal}_{\texttt{pointCouleur}}) & \not\leq & \mathit{Type}(\texttt{equal}_{\texttt{point}}) \\ \Rightarrow & \texttt{pointCouleur} & \not\leq & \texttt{point} \end{array}$$

• `pointCouleur` est une sous-classe de `point`.

• `pointCouleur` n'est pas un sous-type de `point`.

`pointCouleur` ne peut pas être sous-type de `Point`

Pourquoi ?

- C'est sémantiquement incorrecte : on pourrait comparer des points non comparables.

- Mais surtout : cela produirait une erreur à l'exécution *due au typage* $\Rightarrow$ le typage statique deviendrait non sûr ...

Nous allons supposer que `pointCouleur` $\preceq$ `point` et montrer que cela produirait une erreur à l'exécution.

Cependant, nous ne pouvons faire cette démonstration dans aucun langage connu :

- La plupart contournent soignseument les méthodes binaires (on verra comment, en Java).

- En Ocaml, elles sont autorisées et correctement typés, mais sans passer par le sous-typage, ce qui fait qu'elles ne provoquent pas d'erreur !

Java hypothétique

Pour faire notre démonstration nous devons nous placer dans un langage hypothétique où l'on fait du sous-typage structurel et rien d'autre.

Nous nous plaçons dans un Java hypothétique où :

- le sous-typage est structurel,
- le type de self est correctement calculé : il change pendant l'héritage,
- il n'y a pas de surcharge. Une méthode redéfinie dans une sous-classe écrase la méthode de la classe parente.

Méthodes binaires en Java

```
class Point {
    int x ;
    Point (int x0) { x = x0 ; }
    int getx(){ return x ;}
    boolean equal(Point p) { return this.getx() == p.getx() ;}
}

class PointColore extends Point {
    String c ;
    PointColore (int x0, String c0) { super(x0) ; c = c0 ; }
    String getCouleur () { return c ; }
    boolean equal(PointColore p) {
         return ((this.getCouleur() == p.getCouleur()) &&
                 super.equal(p)) ;}
}

public class TestePointBinaire{
    static boolean breakIt(Point p){
        Point t = new Point(1) ;
        return p.equal(t) ;
    }
    public static void main (String arg[]) {
        PointColore pc = new PointColore (1,"Rouge") ;
        boolean res = breakIt(pc) ;
        if (res) {System.out.println("Egaux") ;}
        else  {System.out.println("Differents") ;}
    }
}
```

Typage hypothétique

D'après notre typage hypothétique :

$$\texttt{Point} : \begin{cases} \texttt{getx} & : \texttt{int} \\ \texttt{equal} & : \texttt{Point} \rightarrow \texttt{boolean} \end{cases}$$

$$\texttt{PointCouleur} : \begin{cases} \texttt{getx} & : \texttt{int} \\ \texttt{getCouleur} & : \texttt{String} \\ \texttt{equal} & : \texttt{PointCouleur} \rightarrow \texttt{boolean} \end{cases}$$

Par ailleurs, la méthode statique **`breakIt`**, a pour type :

$$\texttt{breakIt} : \texttt{Point} \rightarrow \texttt{boolean}$$

Notre hypothèse est que `PointColore` $\preceq$ `Point`.

Donc, on pourra appliquer la fonction **`breakIt`** sur un objet de type `PointColore`.

Exécution hypothétique

```
class Point {
 boolean equal(Point p) { return this.getx() == p.getx() ;}
}
class PointColore extends Point {
 boolean equal(PointColore p) {
   return ((this.getCouleur() == p.getCouleur()) && super.equal(p)) ;}
}
...
  static boolean breakIt(Point p){
      Point t = new Point(1) ;
      return p.equal(t) ;
  }
  public static void main (String arg[]) {
      PointColore pc = new PointColore (1,"Rouge") ;
      boolean res = breakIt(pc) ;
    }
}
```

Soit `pc = new PointColore (1,"Rouge")` :

`breakIt(pc)` $\Rightarrow$

$\Rightarrow$ `p.equal(t)` où `p=pc` et `t=new Point(1)`

$\Rightarrow$ `pc.equal(t)`

$\Rightarrow$ `(pc.getCouleur == t.getCouleur) && super.equal(t)`

$\Rightarrow$ `("Rouge" == (new Point(1)).getCouleur)`
`&& super.equal(t)`

$\Rightarrow$ **Erreur** car `t=new Point(1)` n'a pas de méthode `getCouleur`

Exécution hypothétique (suite)

Si `pc =new PointColore (1,"Rouge")` :

`breakIt(pc)` $\Rightarrow$
$\Rightarrow$ `p.equal(t)` où `p=pc` et `t=new Point(1)`
$\Rightarrow$ `pc.equal(t)`
$\Rightarrow$ `(pc.getCouleur == t.getCouleur) && super.equal(t)`
$\Rightarrow$ `("Rouge" == (new Point(1)).getCouleur)`
`&& super.equal(t)`
$\Rightarrow$ *Erreur* car `t=new Point(1)` n'a pas de méthode `getCouleur`

Lors de l'exécution de `p.equal(t)`, on a :

- Comme `p = pc` , par liaison tardive, le code exécuté pour `equal` est celui de `pointCouleur`.
- Dans celui-ci, on envoi le message `getCouleur` à `t`, qui étant une instance de `Point` ne possède pas de méthode `getCouleur`,
- une erreur est déclenchée pendant l'exécution.

Conclusion : il n'est pas sûr du point de vue du typage de considérer `pointCouleur` $\preceq$ `Point`

Méthodes binaires en Java

En java, une méthode binaire est typée en donnant à self le type de la classe dans laquelle elle est définie.

Par conséquent, elle n'est pas héritée comme une méthode binaire : elle n'accepte plus en argument qu'un objet de la classe parente d'origine.

Ce mécanisme, couplé avec la surcharge, fait qu'il y a deux méthodes `equal` dans la classe `PointColore` :

- celle de la classe parente, de type `Point` → `boolean`, et qui ne fait pas appel à `getCouleur`.
- celle de la classe courante, de type `PointColore` → `boolean`

⇒ L'appel `breakIt` s'exécute sans erreur, car c'est la méthode de la classe parente qui est choisie ...

Exécution en vrai Java

`pc =new PointColore (1,"Rouge")` :

`breakIt(pc)` $\Rightarrow$
$\Rightarrow$ `p.equal(t)` où `p=pc` et `t=new Point(1)`
$\Rightarrow$ `pc.equal(t)`

- `p = pc` est instance de `PointCouleur`. Comme l'argument de `pc.equal(t)` est de type `Point`, c'est la méthode equal de `Point` qui est exécuté.

- Cette méthode ne compare que les coordonnées de `Point`. Elle s'exécute donc correctement.

  $\Rightarrow$ `pc.equal(t)`
  $\Rightarrow$ `pc.x== t.getx()`
  $\Rightarrow$ `true`

Le résultat (true) nous dit que le point `new Point(1)` "est égal" au point de couleur `new PointCouleur (1,''Vert'')`.

On a comparé uniquement les coordonnées (comparaison incomplète) et de plus la méthode invoquée n'est pas celle de l'objet courant !

Ce n'est pas vraiement le comportément auquel on s'attend.

Méthodes binaires en Java (conclusion)

Le non-typage de self + surcharge avec les méthodes parentes, permet d'éviter une erreur à l'exécution.

Mais c'est au prix d'une sémantique étrange :

- la liaison tardive est effectuée "parfois" en présence de méthodes binaires, et d'autres fois, c'est la méthode de la classe parente qui est exécutée.

- Des objets non comparables le deviennent (points et points colorés).

Un typage plus satisfaisant devrait toujours autoriser la liaison tardive, mais interdire le mélange d'objets incomparables via les méthodes binaires ....

Méthodes binaires en Ocaml

```
class point x0 =
  object (self : 'a)
    val x = abs x0
    method getx = x
    method equal(p : 'a) = x = p#getx
  end

class pointCouleur (x0,c0) =
 object(self :'a)
   inherit point x0 as super
   val c = (c0 : string)
   method get_couleur = c
   method equal(p : 'a) =
      (c = p#get_couleur) && super#equal(p)
 end

let breakit(p : #point) =
  let t = new point 1 in
  p#equal(t);;
```

Typage des méthodes binaires en Ocaml

Les méthodes binaires sont correctement typées grâce à l'utilisation du polymorphisme des variables de rangée.

En particulier, le type de `equal` dans PointCouleur n'est pas un sous-type, mais une *instance polymorphe* de celui de equal dans `Point`.

Conséquence : le typage de `breakit`, impose que p et t soient uniformément typés dans son corps :

```
let breakit(p : #point) =
  let t = new point 1 in
  p#equal(t) ; ;
```

Et puisque dans son corps `t = new point 1`, alors `p` doit être de type `point` fermé.

```
val breakit : point -> bool = <fun>
```

Typage des méthodes binaires en Ocaml

En Ocaml, on ne peut comparer un point t qu'avec un autre point p. Les mélanges autorisés en Java (mais contournés de facto, au prix d'un comportement baroque), sont ici interdits d'eemblée par le typage :

```
# let pc = new pointCouleur (2,"Vert") in
  breakit(pc) ; ;
This expression has type pointCouleur but is here used with type point
Only the first object type has a method get_couleur

# let p = new point 2 in
  breakit(p) ; ;
- : bool = false

# let pc1 = new pointCouleur (2,"Vert") and
  pc2 = new pointCouleur (2,"Bleu") in pc1#equal(pc2) ; ;
- : bool = false
```

Ainsi, le typage d'Ocaml permet :

• d'utiliser les méthodes binaires avec liaison tardive de la façon attendue,

• tout en interdisant les mélanges d'objets non comparables produisant, soit des erreurs d'exécution, soit des résultats sémantiquement étranges.

Conclusion

Le typage des objets est délicat.

- Indépendemment du bon typage de self, le problème le sous-typage des objets avec des méthodes binaires reste un problème difficile.

- Chaque langage tente de traiter, ou la plupart des fois, de contourner les difficultés, avec un choix de solutions plus ou moins compréhensibles par le programmeur.

- Ces choix étant souvent différentes, cela ajoute à la difficulté dans la compréhension des langages objets.

- Ocaml et Java proposent des solutions différentes à ces problèmes, chacune défendable dans l'esprit propre à chaque langage.

- Un des objectifs de ce cours est de mettre ces problèmes en lumière, dans le but d'améliorer la compréhension, et donc le bon usage des langages objets.