Les points délicats de la programmation à objets.

Didier Rémy
2001 - 2002

http://cristal.inria.fr/˜remy/mot/5/
http://www.enseignement.polytechnique.fr/profs/informatique/Didier.Remy/mot/5/

Slide 1

Cours	**Exercices**
1. Les mécanismes de liaison 2. Héritage multiple et wrappers 3. Récursion, self, self-type et le clonage 4. Le Sous-typage 5. Les méthodes binaires	1. Inoubliable 2. Surcharge 3. Les tuyaux 4. Sous-typage 5. Sauvegarde 6. Démon

Les mécanismes de liaison.

Les différents mécanismes de liaison

Slide 2

Le mécanisme de liaison décrit l'effet d'une définition de variable et de son utilisation dans un programme.

La liaison statique donne à la variable une valeur dans une portée statique, *i.e.* connue à la compilation.

La liaison dynamique a une portée dynamique ; une telle liaison peut être affectée pendant le déroulement du programme par une autre définition, changeant le sens des prochaines utilisations de cette variable.

La liaison tardive (ou retardée) est un mécanisme propre à la programmation avec objets : les appels récursifs entre les méthodes d'un même objet sont déterminés à la création de l'objet et non de la classe.

La surcharge permet d'avoir plusieurs définitions simultanées associées à une même variable (ou méthode). Le choix de la définition à considérer dépend du type (statique ou dynamique) du (ou des) arguments passés à la fonction.

Liaison statique

Slide 3

Dans Ocaml, la liaison est statique : la valeur associée à une variable est fixée définitivement à la première définition de cette variable ; elle ne peut pas être affectée par des liaisons ultérieures.

Attention ! Une liaison peut en cacher une autre ! mais ne peut pas la changer.

```
let x = 1;;
```

val x : int = 1

```
let f y = x + y;;
```

val f : int -> int = <fun>

```
f 0;;
```

− : int = 1

```
let x = 3;;
```

val x : int = 3

```
f 0;;
```

− : int = 1

La plupart des langages de programmation utilisent (heureusement) la liaison statique.

Liaison dynamique

Slide 4

La liaison dynamique signifie que la valeur d'une variable est prise à l'exécution et utilise la dernière définition de cette variable.

La liaison dynamique peut être simulée par la liaison statique en remplaçant les variables par des références vers des variables.

```
(* la première fois *)
let x = ref 1;;
```

val x : int ref = {contents=1}

```
let f y = !x + !y;;
```

val f : int ref -> int = <fun>

```
f (ref 0);;
```

– : int = 1

```
(* les fois suivantes *)
x := 3; x;;
```

– : int ref = {contents=3}

```
f (ref 0);;
```

– : int = 3

Lisp (par exemple le Lisp de Emacs) utilise la liaison dynamique. C'est une erreur de jeunesse...

Liaison dynamique (danger)

Slide 5

Si toutes les liaisons sont dynamiques, c'est comme si toutes les variables étaient des références.

On ne peut plus garantir aucun invariant... sans avoir le programme tout entier.

La liaison statique limite les effets globaux à des objets mutables. Une fonction ne contenant pas de variables mutables possède une sémantique indépendante de son contexte d'utilisation. On dit qu'elle est *référentiellement transparente*.

Liaison tardive

Slide 6

La liaison *tardive* (ou *retardée*) est un mécanisme propre aux langages à objets, qui consiste à retarder la liaison jusqu'au moment de la création des objets. (Attention, ce mécanisme est parfois appelé *à tord*, liaison dynamique).

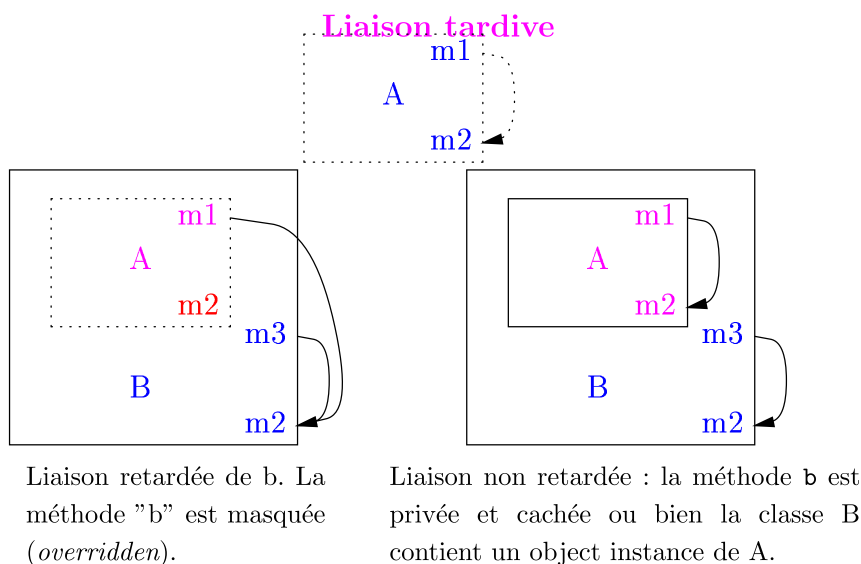
Dans une classe `A`, une méthode `m1` peut appeler une méthode `m2` de la même classe en envoyant un message `m2` à `self`. Cela simule un appel récursif, mais le *câblage* de cet appel ne sera réalisé qu'au moment de la fabrication de l'objet.

En effet, une sous-classe `B` de `A` peut redéfinir la méthode `m2`. Dans un objet de `B`, c'est la nouvelle définition de `m2` qui sera appelé par la méthode `m1`.

La liaison tardive est au cœur de la programmation avec objets, de son expressivité , mais aussi de ses difficultés.

Liaison tardive

Slide 7



Liaison retardée de b. La méthode "b" est masquée (*overridden*).

Liaison non retardée : la méthode `b` est privée et cachée ou bien la classe B contient un object instance de A.

Liaison tardive (danger)

Slide 8

La liaison tardive limite l'aspect dynamique aux classes. De plus si les classes ne sont pas de première classe, une classe n'a qu'un nombre fini de classes parentes connues statiquement.

L'effet retardé de la liaison est plus contrôlable que celui de la liaison dynamique... à condition de connaître les classes parentes. Aussi

- Lorsqu'on ignore l'implémentation d'une classe parente, il est impossible de connaître l'effet de la redéfinition d'une méthode sur les autres méthodes, ce qui peut avoir des conséquences très inattendues.
- La liaison tardive est délicate, souvent difficile à contrôler, mais c'est sans doute le prix à payer pour l'expressivité. L'héritage repose sur l'optimisme (que la classe parente fera bien ce que l'on imagine à la seule vue de son interface).

Liaison tardive (variables)

Slide 9

En Ocaml, les variables d'instances sont héritées mais ne sont jamais en liaison tardive.

L'utilisation d'une variable d'instance dans une méthode réfère à la variable d'instance définie au-dessus (éventuellement dans une classe parente) et la plus proche.

La redéfinition d'une variable d'instance ne change pas la valeur de la précédente mais revient à faire une nouvelle définition qui sera utilisée par les méthodes définies en dessous (éventuellement dans une sous-classe)

```
class i  = object               val x = 1 method i  = x end
class ii = object inherit i val x = 2 method ii = x end;;
let p = new ii in p#i, p#ii;;
```

− : int ∗ int = 1, 2

Liaison tardive (méthode privées)

En Ocaml, les méthodes visibles sont toujours en liaison tardive

Seules les méthodes privées peuvent être cachées. En effet, les méthodes publiques peuvent avoir été utilisées récursivement ; les cacher permettrait de les redéfinir avec un type incompatible.

Slide 10

Exercice 1 *Construire un tel exemple et vérifier qu'il n'est pas typable en Ocaml.* Réponse □

Les méthodes privées peuvent être cachées : leur liaison est alors résolue à ce moment là : une redéfinition ultérieur créera une nouvelle méthode de même nom, mais indépendante.

Les méthodes privées peuvent aussi être rendues publiques.

Méthode privée ≡ **pas encore** utilisée "en public"
(*e.g.* pas encore d'appel externe)

Liaison tardive (contrôle)

Comment arrêter (résoudre) la liaison tardive dans une classe ?

En Java

- Une méthode finale ne peut plus être redéfinie
  (cela produira une Erreur de type).
- On peut appeler la méthode d'une classe particulière.

Slide 11

En Ocaml

- Utiliser une fonction plutôt qu'une méthode
  (on utiliserait une méthode statique en Java).
- Utiliser une méthode privée cachée auxiliaire.
- Remplacer l'héritage par la délégation : créer un objet de la classe parente A dans la sous-classe B.

Liaison tardive (protection préventive)

Slide 12

Pour se prémunir contre le risque de cassé un invariant de la méthode `a` en redéfinition une méthode `b` dans une sous-classe, on peut utiliser :

– une méthode privée (figure de gauche), ou

– une fonction auxiliaire (figure de droite) ≈ méthode statique :

```
class type ab = object method a : int method b : int end;;
```

```
class méthode_cachée : ab =
  object (self)
    method a = 2 * self # b'
    method private b' = 1
    method b = self # b'
  end;;
```

```
class liaison_cachée =
  let b' = 1 in
  object (self)
    method a = 2 * b'
    method b = b' self
  end;;
```

Liaison tardive (protection préventive)

Slide 13

Une classe héritée ne peut plus changer le comportement de `a` par inadvertance, *i.e.* en redéfinissant celui de `b`.

```
class droite =
  object (self)
    inherit a
    method c = 10 + self#b
    method b = 3
  end;;
let x = new a and y = new b in x#a, y#a, y#c;;
```

– : int ∗ int ∗ int = 2, 2, 13

Liaison tardive (protection curative)

Relais : Si la classe d'origine est exposée, on peut en obtenir une version protégée empêchant la redéfinition de `b` de briser les invariants de la classe d'origine en fabriquant une classe **relais** où les méthodes **délèguent** leur exécution aux méthodes d'une instance de la classe d'origine.

Slide 14

```
class exposée =
  object (self)
    method a = 2 * self # b
    method b = self # a
  end;;
```

```
class protégée =
  object (self)
    val relais = new a
    method a = relais#a
    method c = 10 + self#b
    method b = 3
  end;;
```

```
let x = new a and y = new b in x#a, y#a, y#c;;
```

− : int ∗ int ∗ int = 2, 2, 3

Liaison tardive (Exemple)

Slide 15

Solution préventive

```
class type blindé = object
  method clé : string -> bool
  method toc_toc : string -> string
end
class portail mot_de_passe : blindé = object (self)
  method private clef (s : string) = (s = mot_de_passe)
  method toc_toc k =
   if self # clef k then "entrez!" else "sortez!"
  method clé s = self#clef s
end;;
```

La redéfinition de la méthode `clé` dans une sous classe n'affectera pas la version privée `clé_cachée`.

Liaison tardive (Exemple)

Solution préventive (variante)

```
class portail mot_de_passe =
  let clef (s : string) = s = mot_de_passe in
  object (self)
    method toc_toc k = if clef k then "oui!" else "non!"
    method clé s = clef s
  end;;
```

Slide 16

Structure stratifiée

```
class porte = object
  val parent = new portail "c'est moi!"
  method toc_toc = parent#toc_toc
  method clé s =
    parent#clé s or parent#clé (String.uppercase s)
end;;
```

Liaison surchargée

La surcharge est présente dans certains langages avec ou sans objets, avec ou sans mécanisme de sous-typage. Il n'y a pas de surcharge en Ocaml, il y en a en Java.

La liaison surchargée est statique mais une variable peut avoir plusieurs définitions simultanées avec des types différents.

Slide 17

La surcharge est souvent réservée aux fonctions, et seulement sur les types des arguments. La résolution est alors effectuée en fonction du type des arguments.

Liaison surchargée (résolution)

La résolution de la surcharge, consiste à choisir quelle définition de la variable utiliser. Ce choix est fait à chaque utilisation de la variable et dépend de son contexte.

Résolution statique

Slide 18

La résolution statique utilise les types connus à la compilation. Les types des valeurs n'ont pas besoin d'être passés à l'exécution.

Résolution dynamique

La résolution dynamique utilise les types à l'exécution, ce qui oblige à les conserver.

En Java, une partie de la surcharge est résolue dynamiquement, mais uniquement avec l'information de type statique.

Liaison surchargée (exemple en Java)

Slide 19

```
class Point {
  int x = 0;
  Point (int x0) { x = x0; }
}
class Bipoint extends Point {
  int y = 0;
  Bipoint (int x0) { super (x0); }
  Bipoint (int x0, int y0) { super (x0); y = y0; }
}
```

On peut créer des bipoints en passant un ou deux arguments et le constructeur de classe correspondant sera appelé :

```
Bipoint p = Bipoint(1); Bipoint q = Bipoint(1,2)
```

Dans le premier cas, le point aura pour abscisse sa valeur par défaut `0`.

Liaison surchargée (avantage)

Slide 20

Surcharge statique

La surcharge est le plus souvent statique.

Dans ce cas, elle n'augmente pas l'expressivité, mais la convivialité. Elle permet de donner le même nom à plusieurs variantes d'une même fonction.

Surcharge dynamique

Elle augmente l'expressivité du langage.

Par exemple, une autre approche de la programmation avec objets traite les méthodes comme des fonctions surchargées. L'envoi d'un message est alors résolu dynamiquement en regardant le type des objets reçus en argument à chaque appel de méthode.

Liaison surchargée (danger)

Slide 21

Sémantique typée

La sémantique du programme est déterminée par le typage. Si la surcharge est statique, le typage n'est qu'une approximation de la valeur. Au cours du calcul le typage devient plus précis.

Par exemple, il est fréquent de remplacer un programme par le programme obtenu après une étape d'évaluation, *i.e.* faire de l'évaluation partielle, manuellement ou automatiquement.

Dans le cas de la surcharge résolu statiquement, cette transformation n'est plus valable sans précautions : maintenir artificiellement une information de typage approchée (ce qui n'est pas toujours possible).

Liaison surchargée (danger)

Évaluation partielle

L'évaluation partielle consiste à remplacé un appel de fonction connu statiquement par le résultat de cet appel.

Slide 22

```
class Programme {
  static boolean surcharge (Object x) { return true; }
  static int surcharge (String s) { return 42; }
  static void eval (Object x)
    { System.out.println(surcharge(x)); }
  static void call () { eval("hello"); }
}
```

Ici `hello` est passé à la méthode `eval`, donc coercé en un `Object` et passé à `surcharge` avec le type statique `Objet`. Le compilateur choisi donc la première définition et `Programme.call` retourne `true`

Liaison surchargée (danger)

Version spécialisée incorrecte

Slide 23

```
class Programme_partiellement_evalue {
  ...
  static void call () {
    System.out.println(surcharge("hello"));
  }
}
```

Ici `hello` est directement passé à `surcharge` avec le type `String`. Le compilateur choisit donc la deuxième définition.

Pourtant, l'intuition du programmeur en écrivant `eval` est (à tord) de propager mentallement la surcharge conduisant au programme ci-dessus.

Surcharge (danger)

Version spécialisée correcte

Slide 24

```
class Programme_partiellement_evalue {
  ...
  static void call () {
    Object x = "hello";
    System.out.println(surcharge(x));
  }
}
```

Ici `hello` est directement passé à `surcharge` avec le type `Object`, comme dans la version de référence. Le compilateur choisit donc la première définition.

Liaison surchargée (exercice)

Slide 25

```
class A           { int bin(A y) { return 1; } }
class B extends A { int bin(B y) { return 2; } }

A aa = new A(); B bb = new B(); A ab = bb;
```

Exercice 2 *Quelle est la valeur retournée par* ***x.bin(y)*** *pour chacune des 9 combinaisons possibles de "(x,y)" ?* Réponse

Écrire un programme Java qui permet de vérifier les cas ci-dessus (en calculant les 9 combinaisons). Réponse

On considére maintenant la définition suivante :

```
class A         { int bin(A x){ return 1; } }
class B extends A {
  int bin(A x) { return 3; }
  int bin(B x) { return 2; }
}
```

Quelle est la différence "d'effet" entre les deux méthodes définies dans la classe B. Réponse

Répondre à la première question mais avec la définition ci-dessus. Réponse

Écrire une variante qui retourne toujours (et seulement) `2` *lorsque les deux objets sont de la classe* `B` *(et seulement* `1` *autrement)* Réponse

La surcharge peut être éliminée statiquement en choisissant des nom non-ambigü (par exemple en suffixant les nom par le type des arguments) : écrire une version du programme de la question 2 qui n'utile pas la surcharge. Réponse

Slide 26

Quelle version de `bin_A` *ou* `bin_B` *faut-il utiliser pour les différents appels (9 combinaisons possibles) pour être le plus précis possible ? (On indiquera par une prime si la version exécutée est celle définie dans la class A ou dans la class B).* Réponse □

Héritage multiples, et wrappers (class mixins).

Héritage simple *v.s.* multiple

Avec l'héritage simple, une classe a au plus un parent. La hiérarchie des classes forme donc un arbre.

Avec l'héritage multiple, une classe peut avoir plusieurs parents. La hiérarchie des classes forme un graphe acyclique.

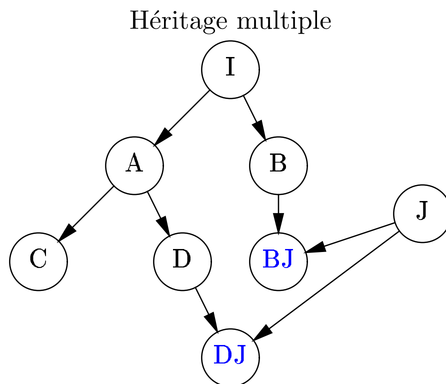
C'est une généralisation naturelle de l'héritage simple, mais elle rend la compilation plus difficile.

Slide 27

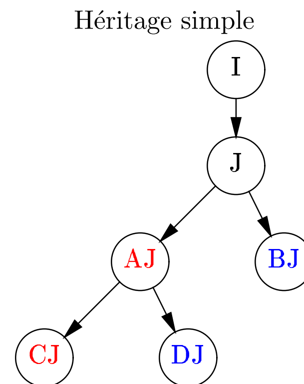
Comparaison

Slide 28

Héritage multiple



Héritage simple



Extension des classes B et D par J, a posteriori, sans modification des autres classes.

Les classes A,B,C et D sont pollués par la classe J. Il faut changer le code des classes A et B.

Quelques exemples

La sauvegarde

Construction du groupe des entiers

Constructeurs de classes (wrappers)

Les pipes

Slide 29

Les pipes

Écriture

```
class virtual writer = object (self)
  method virtual put_char : char -> unit
  method put_string s =
    for i = 0 to String.length s - 1
    do self#put_char s.[i] done
end;;
class fileout filename = object
  inherit writer
  val chan = open_out filename
  method put_char c = output_char chan c
end;;
```

Slide 30

Héritage multiple (pipes)

Slide 31

Lecture

```
class virtual reader = object (self)
  method virtual get_char : char
  method get_string n =
    let s = String.create n in
    for i = 0 to n - 1 do s.[i] <- self#get_char done;
    s
end;;
class filein filename = object
  inherit reader
  val chan = open_in filename
  method get_char = input_char chan
end;;
```

Héritage multiple (pipes)

Slide 32

Pipe

```
class pipe = object
  val q = Queue.create ()
  inherit writer()
  inherit reader()
  method put_char c = Queue.add c q
  method get_char =
    try Queue.take q with
      Queue.Empty -> raise End_of_file
end
```

Exercice 3 *Reprendre l'exemple des* pipes *en le rendant plus réaliste, par exemple, ajouter une fonction d'écriture et de lecture des entiers.* □

Les wrappers (class mixins)

Slide 33

Les *mixins* (de classe) ou *wrappers* permettent d'abstraire des classes par rapport à d'autres classes. Par exemple, on écrirait

```
class w (arg : arg_type) = body end
class b = w c
```

Ce n'est pas directement possible en Ocaml au niveau du langage de classe. Parfois, on peut obtenir le partage désiré en utilisant des modules (expressif mais lourd) ou l'héritage multiple (léger, mais limité), ou une combinaison des deux. Toutefois, aucune solution n'est parfaite car le typage reste limité.

Utilisation des modules

```
module W (Arg : sig class class_arg : class_arg_type end) =
  struct open Arg class resultat = body end
class b = Wrapper(struct class arg = c end).resultat
```

`b` ne voit que les méthodes de `arg_type` et de `body` (pas de `arg`).

Exemple

Slide 34

```
module Secure (Arg :
              sig class c : object
                method retrait : float -> float
                method dépôt : float -> unit method solde : float
              end end) =
  struct
    let interdits_bancaire = ref []
    let gelé x = List.mem x !interdits_bancaire
    class c = object (self)
      inherit Arg.c as super
      method retrait x =
        if gelé (self :> Arg.c) then raise (Failure "interdit bancaire")
        else super#retrait x
    end
  end;;
module Secure_compte = Secure (struct class c = compte end);;
class secure_compte = Secure_compte.c;;
```

Wrappers (simulation de l'héritage multiple)

Slide 35

Les wrappers permettent de simuler l'héritage multiple (dans les langages qui n'en ont pas) :

```
class c1 = body1
class c2 = inherit c0 body2
class c3 = object inherit c1 inherit c2 body3 end
```

est implémenté par :

```
class c1 (s) = object inherit s body1 end
class c2 (s) = object inherit c0(s) body2 end
class c3 (s) = object inherit c2(c1(s)) body3 end
```

et en remplaçant `new c1` par `new c1(object end)`.

Si le type de `s` *doit être fixé, ce qui est en général le cas, cette solution est restreinte et ne remplace pas l'héritage multiple.*

Wrapper *v.s.* héritage multiple

Slide 36

Les wrappers

– Il faut prévoir le besoin d'abstraction.
– Il faut connaître l'interface de l'argument (les composantes non spécifiées qui peuvent être oubliées seront cachées).
+ Avantage : le câblage (liaison de super, overriding) peut-être fait dans le wrapper et être partagé.
+ Les variables de la classe parente sont visibles.

L'héritage multiple

+ Rien à prévoir
+ Il y a toujours une classe parente hypothétique (héritage)
– Le câblage doit être réalisé après coup à chaque utilisation
– Les variables ne peuvent pas être virtuelles

Les deux sont complémentaires, avec un recouvrement important.

Récursion, self, son type et le clonage..

Self (lui-même)

Slide 37

Self désigne pendant l'exécution d'une méthode l'objet qui a appelé cette méthode. En Java cet objet est désigner par le mot clé `this`. En Ocaml, on doit déclarer en tête de la classe une variable pour désigner cet objet. Nous parlerons de `self` ici quelque soit le mécanisme de liaison.

"self" peut donc 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 d'exécuter une méthode, ce qui réalise la récursion. Parce que cette méthode est prise dans l'objet à l'exécution (du moins en théorie, le compilateur étant libre de compiler l'appel autrement pourvu que son effet soit indiscernable) et non dans la classe, un message à self effectue une liaison tardive.

Le type de self en Ocaml

Slide 38

Dans une classe, le type de self est celui d'un objet ayant toutes les méthodes de la classe et peut-être d'autres (par exemple celles ajoutées dans une sous-classe).

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

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

En Ocaml, le type de self est polymorphe; il contient une variable de rangée. On peut voir le type de self d'une classe `c` en faisant :

```
fun x -> (x :> c);;
```

– : (< m : 'a; .. > as 'a) -> c = <fun>

Ici, c'est un type récursif `'a` d'un objet qui a une méthode `m` qui retourne un objet de type `'a`.

Le type de self (suite)

Slide 39

En Ocaml, le type polymorphe de self lui permet dans une sous-classe d'être une instance de celui de la classe parente.

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

class cd : object ('a) method m : 'a method n : int end

```
fun x -> (x :> cd);;
```

– : (< m : 'a; n : int; .. > as 'a) -> cd = <fun>

Aussi self dans un objet de la sous-classe aura bien le type de la sous-classe et non celui de la classe parente.

```
new m1 # m1;;
```

– : m1 = <obj>

```
new m1_m2 # m1;;
```

– : m1_m2 = <obj>

Le type de self en Java

Slide 40

En java, il n'y pas de type pour self (this) et on lui donne le type de la classe courante : en particulier, une méthode qui retourne `this` garde dans une classe héritée le type de la classe parente.

```
class C {
    C () { }
    C m() { return this; }
}
```

```
class CD extends C {
    CD () { }
    int n() { return 2; }
}
```

```
class Error {
    static CD cd = new CD();
    static int x = (cd.m()).n();
}
```

Exercice 4 *Vérifier qu'en Ocaml la classe `Error` serait correctement typée. (on écrira le code Ocaml correspondant)* □

Types récursifs

Slide 41

Les types des objets sont récursifs, car une méthode doit pouvoir retourner un objet du même type.

En Ocaml un type récursif inféré est forcément un type objet et décrit par une équation `(< m1 : int; m2 : 'a > as 'a)` avec le mot clé `as`. Ici, la variable `'a` sert uniquement à décrire la récursion : elle n'est pas polymorphe car il n'y a pas d'autre variable dans le membre gauche.

Par contre, le type ouvert `(< m1 : int; m2 ; 'a; ..> as 'a)` est polymorphe, car `..` représente une variable de type anonyme.

En Ocaml, il existe aussi des types récursifs déclarés :

```
type 'a liste = Vide | Cellule of 'a * 'a liste
```

Types récursifs

En java les types peuvent aussi être récursifs[a] mais il sont toujours déclarés :

```
class Point {
  int x;
  Point (int x0) { x = x0; }
  Point self () { return this; }
}
```

Slide 42

Cette classe définit un type d'objet `Point` ayant un champ `x` de type `int` et une méthode `self` de type `Point`.

[a]Tout langage de programmation intéressant a des structures de données récursives, donc des types récursifs pour les décrire.

Le type de self (avantages)

Le type de self permet de traiter correctement
– les méthodes qui retournent self.
– les méthodes binaires.

Overriding La construction `{< l_1 = e_1; ... l_p = e_p >}` permet de retourner une copie de self dans lesquels les variables `l_i` sont liée aux valeurs résultant de l'évaluation de `e_i`.

Slide 43

Application : il permet de créer une nouvel objet du même type que self.

En effet les constructeurs de la classe retournent un objet de la classe courante et ils feront de même dans une sous-classe. (La construction `Oo.copy` ne permet pas de changer les variables d'instances.)

Clonage

Un résumé des opérations de clonage et de leurs effets

```
class démon =
  let population = ref 0 in
  let créateur() = Random.int 999999 in
  let combine x y =
    let m = créateur() in (x land m) lor (y land (lnot m)) in
```

Slide 44

```
  object (moi : 'en_personne)

    val au_delà = ref 0
    method pensée x = au_delà := x
    method intuition = !au_delà

    val mutable gènes = créateur()
    method patrimoine = gènes

    method même = moi
```

```
    method copie = moi # contrôle; Oo.copy moi
    method clone = moi # contrôle; {< >}

    method reproduction (x : 'en_personne) =
      moi # contrôle ; {< gènes = combine gènes x#patrimoine >}
    method mutation = gènes <- moi # mute

    method private mute = gènes lxor (1 lsl (créateur()))
    method private contrôle =
      if !population < 10000 then incr population
      else raise (Failure "surpopulation")
    initializer moi # contrôle
  end;;
```

Slide 45

Exercice 5 (demon) *On considère la classe* `démon` *ci-dessus.*

Quel est la différence entre la variable d'instance mutable `gènes` *et la variable d'intance non mutable* `au_delà` *dont le contenu est une référence ?* Réponse

Mettre en évidence le comportement de `au_delà` *sur une exemple*

permettant de communiquer entre un certains groupes d'objets que l'on précisera. Réponse

Quel est le rôle `population` *?* Réponse

Quel est la différence entre les méthodes `copie` *et* `clone` *?* Réponse

Pourquoi `mutation` *n'appelle pas* `contrôle` *?* Réponse

Les démons peuvent-ils être en surpopulation ? Qui a-t-il de remarquable dans ce cas ? Réponse

Quel est la différence entre les méthode `copie` *et* `même` *?* Réponse

Slide 46

À quoi sert la méthode `même` *?* Réponse

Quel est la différence entre `combine` *et* `mute` *? Est-ce que* `mute` *pourrait être une fonction auxiliaire comme combine ? Quel est le point commun.* Réponse □

Exercice 6 *(Voir l'exercice sur la sauvegarde)* □

Le Sous-typage.

Qu'est-ce que c'est ?

Le sous-typage est un affaiblissement de l'information statique de type qui sous certaines conditions permet de voir des objets de types différents sous un même type.

La relation de sous-typage est toujours fermée par reflexivité et transitivité.

Slide 47

Sous-typage structurel Les types sont construits librement à partir de types de base, et la relation de sous-typage est définie une fois pour toute et ne dépend que de la structure des types.

Sous-typage déclaré Les types sont vus comme des atomes et la relation de sous-typage est construite au fur et à mesure de la définition de nouveaux types par des déclarations manuelles où automatiques.

Bien sûr, le typeur vérifie que les déclaration de sous-typage sont permines.

On peut combiner sous-typage structurel et déclaré.

(Mais c'est autant un cumul des difficultés que des avantages.)

Abbréviations de types

Les abbréviations de types sont transparentes : le sous-typage entre types abbrégés est donc structurel ou par nom mais indépendemment du mécanisme d'abbréviation.

Slide 48

Intuition

Considérons, par exemple, une relation de sous-typage structurelle entre types-enregistrements.

Sous-typage en largeur

Un enregistrement avec plus de champs peut toujours être utilisé à la place d'un enregistrement avec moins de champs. Au niveau des types, on peut donc cacher sans risque d'erreurs un nombre arbitraire de champs.

Slide 49

La relation de sous-typage correspond à l'inclusion des champs.

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.

Contra-variance

Co-variance On dit que l'opérateur des listes est contra-variant, parce qu'il propage la relation de sous-typage entre les éléments en une relation entre des listes de ces éléments dans le même sens.

Contra-variance Cette propagation est inversée dans le cas des consommateurs : Si un élément de type `A` peut être utilisé comme un élément de type `B`, alors une fonctions dont l'argument est de type `B` peut être coercée en une fonction dont l'argument est de type `A`.

Slide 50

En effet, un argument de type `A` peut être vu par sous-typage comme un argument de type `B`, donc passé à la fonction.

Le type des fonctions

Il est co-variant à droite et contra-variant à gauche.

Non-variance

La propagation de la relation au travers d'un constructeur n'est pas possible dans certains cas. Ces types n'ont qu'eux-mêmes pour sous-types. Le constructeur de type est dit non-variant.

Par exemple, imaginons qu'on définisse le type suivant :

```
type 'a transformation = ('a -> 'a)
```

Slide 51

Le constructeur de type `transformation` est non-variant, car pour qu'une `A transformation` soit plus petit qu'une `B transformation`, il faudrait que `A -> A` soit plus petit que `B -> B`, *i.e.* que `A` soit à la fois plus grand et plus petit que `B`. Ce qui n'est vrai que pour `A` égal à `B` (ici, on suppose que la relation de sous-typage est anti-symmétrique).

Les types mutables

Pour des raisons similaires, le type `'a buffer` défini par

```
type 'a buffer = ('a -> unit) * (unit -> 'a)
```

est non-variant.

Une référence est un tampon à un élément. Il se comporte comme un objet avec deux méthodes de types respectifs `'a -> unit` et `unit -> 'a`. Ainsi le constructeur de référence `'a ref` est non-variant.

Slide 52

Plus généralement, une définition de type est co-variante (resp. contra-variante) en une variable si toutes les occurrences de cette variable sont co-variantes (resp. contra-variantes)

Par défaut, *i.e.* en l'absence d'information, un constructeur de type doit être considéré comme non-variant.

Sous-typage en Ocaml

Le sous-typage est structurel. Il est définit formellement comme la plus petite relation réflexive et transitive fermée par les opérations suivantes :
– Si $A' < A$ et $B < B'$ alors $A \to B < A' \to B'$,
– Si $A_i < B_i$ alors $\langle \ell_1 : A_1; ...\ell_k : A_k; ...l_n : A_n \rangle < \langle \ell_1 : B_1; ...\ell_k : B_k \rangle$
On retrouve la contra-variance à gauche de la flèche et la co-variance à droite de la flèche, plus le sous-typage en largeur pour les types objets.

Slide 53

Comme les types objets ne montrent pas les types des variables, il n'y a pas de cas particulier à faire pour les champs mutables.

Si un champ peut être lu et écrit de l'extérieur, cela revient à donner deux fonctions de lecture et d'écriture de variances opposées, ce qui rend le type de l'objet non-variant par rapport au type du champ mutable.

Exercice 7 *Définir une classe paramétrique* `['a] cell` *des cellules (version objet des références) de type* `'a'`*.* Réponse
Expliquer pourquoi le type `'a cell` *est non-variant par rapport à* `'a`*.* Réponse

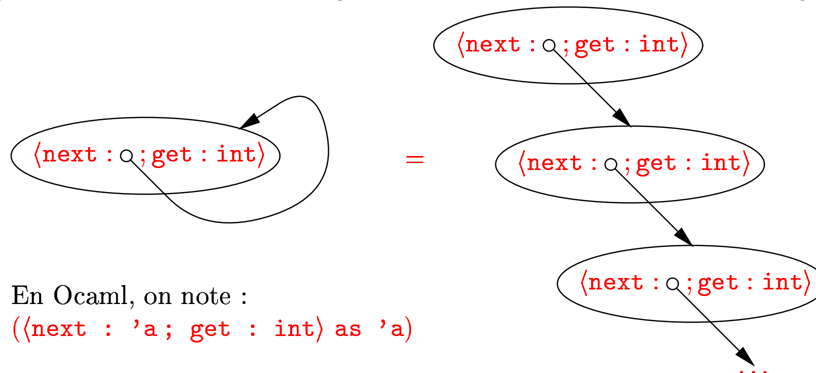
Vérifier expérimentalement (en essyant différentes coercions) que c'est bien le cas. Réponse
Vérifier qu'en cachant certaines méthodes (donc en perdant certaines fonctionnalités), la classe redevient co-variante ou contra-variante. Réponse
Le type des objets d'une classe héritée de `['a] cell` *peut-il être sous-type de* `[a'] cell` *(si oui, donner un exemple) ?* Réponse □

Slide 54

Types récursifs

Les types récursifs sont des arbres infinis réguliers. Un arbre régulier est un arbre qui a un nombre fini de sous-arbres. Il est représentable de façon finie en indiquant le partage de certains sous-arbres par un nommage.

Slide 55



En Ocaml, on note :
`(⟨next : 'a ; get : int⟩ as 'a)`

Types-récursifs (déroulement)

La notation avec partage cache l'arbre déroulé. Le déroulement est toujours possible (c'est la bonne façon de comprendre).

En particulier, $(\alpha \rightarrow \beta)$ `as` α est égal à $((\alpha \rightarrow \beta)$ `as` $\alpha) \rightarrow \beta$ mais aussi à $((\alpha \rightarrow \beta) \rightarrow \beta)$ `as` α qui font tous deux apparaître une occurrence négative de β.

Slide 56

Lorsque la variable muette α utilisée pour dénoter la récursion apparaît en position négative, alors toutes les occurrences de l'arbre désigné par α se répètent positivement et négativement.

Si α apparaît négativement dans t, alors t n'admet pas d'autres sous-types que lui-même.

Par exemple, un objet de la forme $\langle \ell_0 : \alpha \rightarrow \tau_0; \ell_1 : \tau_1; ...\ell_n : \tau_n \rangle$ `as` α ne sera jamais sous-type de $\langle \ell_0 : \alpha \rightarrow \tau_0; \ell_i : \tau_i \rangle$. Il faut d'abord cacher toutes les méthodes binaires.

Perte d'information

Le sous-typage est une perte d'information de type irréversible (*i.e.* non recouvrable statiquement).

C'est bien : cela permet de voir des objets de différentes classes avec une interface commune, et donc de les mettre ensemble dans une même boîte.

Slide 57

C'est embêtant : lorsque l'on retire des objets d'un telle boîte, on ne voit plus que leur interface commune, on ne peut donc plus obtenir de comportement spécifique.

Perte d'information (limitation)

Accepter la perte d'information

On n'a pas besoin de recouvrer les vrais types des objets.

Pour recouvrer l'information perdue

Utilisation de types concrets (variantes).

Slide 58

Type-case : on peut tester l'appartenance à une classe ou le respect d'une interface.

Pour éviter de perdre de l'information

Utilisation du polymorphisme (à la place ou en combinaison avec le sous-typage).

Exemple en Ocaml

Imaginons deux classes `a` et `b` dont ont veux mettre les objets dans une liste avec l'interface `c`.

Avec perte d'information :

```
let (<<) t h = (h :> c) :: t;;
let tous_ensemble = [] << a1 << b1 << a2;;
let send_à_tous x = x#mc in
List.map send_à_tous tous_ensemble;;
```

Slide 59

Avec types concrets :

```
type 'a as_c = A of a | B of b;;
let (<<) t h = h :: t;;
let chacun_pour_soi = [] << A a1 << B b1 << A a2;;
let send_à_chacun =
  function A x -> x#ma | B y -> y#mb in
List.map send_à_chacun chacun_pour_soi;;
```

Solution mixte

Un usage mixte, à la fois homogène et hétérogène est aussi possible :

```
type 'a as_c = A of a | B of b;;
let cA x = (x :> c), A x and cB x = (x :> c), B x;;
let tous_pour_un = [] << cA a1 << cB b1 << cA a2;;
let send_à_tous (x,_) = x#mc in
List.map send_à_tous tous_pour_un;;
let send_à_chacun =
  function _, A x -> x#ma | _, B y -> y#mb in
List.map send_à_chacun tous_pour_un;;
```

Slide 60

Perte d'information (limitation)

En Ocaml, une fonction qui prend en argument un objet avec une méthode `m` appelle cette méthode et retourne son argument :

```
let f x = x#m; x;;
```

```
val f : (< m : 'b; .. > as 'a) -> 'a = <fun>
```

Slide 61

Cette fonction n'utilise pas le sous-typage, mais le polymorphisme : *Pour tout objet possédant une méthode `m`, elle retourne un objet du même type de son argument*

Avec du sous-typage, sans polymorphisme, on pourrait prendre `< m : 'b >` pour le type de l'argument, mais dans ce cas, on ne peut retourner que le type `< m : 'b>`.

```
f : < m : 'b > -> < m : 'b >
```

Une solution pour retrouver l'information est le polymorphisme contraint (plus difficile) : `(f : All ('a <: <m : 'b >) 'a -> 'a)`

Utilisation de type case

Cela revient à un mécanisme de typage dynamique. Les objets doivent alors porter leur types à l'exécution, ce qui peut être coûteux.

En java, on utilise la primitive `instanceof` combinée avec `cast`.

```
if (c instanceof A) ... (A) c ...
else ... c ...
```

Slide 62

Cette combinaison est sûre (ne produit pas d'exception). Les autres usages du cast sont plus dangereux (risque d'exception).

En Ocaml, les valeurs ne portent pas leur type à l'exécution et il n'y a pas de type-case.

Le sous-typage n'est pas de l'héritage

Sous-typage n'implique pas héritage

En Ocaml, les types sont structurels, donc le sous-typage ne dépend pas de la façon dont la classe est construite : prendre par exemple deux copies de la même classe produisant deux classes différentes avec la même interface.

Slide 63

En java, une clause `implement` crée une relation de sous-typage sans relation d'héritage.

Héritage n'implique pas sous-typage

En Ocaml, une classe qui hérite d'une autre ayant une méthode binaire ne crée pas de relation sous-typage entre leur type.

Les méthodes binaires.

Qu'est-ce qu'une méthode binaire ?

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

Slide 64

```
class point x0 =
  object (self)
    val x = abs x0
    method getx = x
    method inf p =
      p#getx < x
    method max p =
      if self#inf p then p
      else self
  end;;
```

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

Une méthode binaire a comme argument un objet du même type que `self`. Par extension, une méthode est binaire si son type contient une occurrence contra-variante du type de self.

Autres exemples

L'union sur les ensembles.

La concaténation des chaînes de caractères

Append sur les listes.

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

Slide 65

Quelle particularité ?

Difficulté Les méthodes binaires posent plusieurs difficultés :
– 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 qu'un méthode binaire reste visible.

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

Slide 66

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.

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.

En conséquence, elle n'est pas héritée comme une méthode binaire : elle n'accepte plus en argument qu'un objet d'une classe parente.

Pour retrouver le bon type dans une sous-classe, il faut la redéfinir. Ou alors, il faudra utiliser des conversions de type, avec risque d'échec à l'exécution.

Slide 67

Méthodes binaires en Ocaml

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

```
class point_coloré x0 y0 =
  object (self)
    inherit point x0 as super
    val y = abs y0
    method gety = y
    method inf p =
      x + y <
        p#getx + p#gety
  end;;
```

```
class point_coloré :
  int -> int ->
  object ('a)
    val x : int
    val y : int
    method getx : int
    method gety : int
    method inf : 'a -> bool
    method max : 'a -> 'a
  end
```

Slide 68

Ingrédients nécessaires

Il faut typer la classe parente en supposant que self est le type d'un objet d'une sous-classe arbitraire, et sans perdre d'information sur le type de self.

Pour cela, il faut utiliser du polymorphisme, que ce soit avec variable de rangée ou avec des contraintes de sous-typage (voir la fin de la partie précédente).

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

Slide 69

Sous-typage des méthodes binaires

En supposant, `p = new point 0` et `q = new point_coloré 0 0` et en oubliant temporairement le typage (on ne pourra donc pas les tester en Ocaml) quelles sont les combinaisons de `x.max y` qui s'évaluerait quand même correctement ?

Réponse

Slide 70

La méthode `max` de la classe `point_coloré` lit indirectement, par un appel à la méthode `inf` variable `y` de son argument donc son argument doit être de la classe `point_coloré`.

x \ y	p	q
p	OK	OK
q	×	OK

Il est donc incorrect de dire que `q` peut être vu avec l'interface d'un point coloré, sinon, on pourrait appeler sa méthode max avec `p`.

Difficulté

Le problème ci-dessus n'est pas facilement détectable, car l'appel à la méthode `gety` est indirect.

Le problème vient du fait que le type de self représente à la fois
(1) les méthodes implémentées,
(2) les méthodes utilisées récursivement.
(3) les méthodes utilisées dans un objet du même type que self.

Slide 71

Ce sont les méthodes (3) qu'il est dangereux de cacher en présence de méthodes binaires. Par exemple, `getx` ou `gety`.

En effet, les méthodes
(1) peuvent être oubliées : elles ne seront simplement pas disponibles. (par exemple `max`)
(2) sont prises dans l'objet et non dans l'argument, et donc toujours présentes. (par exemple `inf`)

Externaliser les méthodes binaires

Slide 72

On peut externaliser les méthodes binaires.

```
class petit_point x0 =
  object (self)
    val x = abs x0
    method getx = x
  end;;
let inf p q = p#getx < q#getx;;
```

val inf : < getx : 'a; .. > -> < getx : 'a; .. > -> bool = <fun>

```
let max p q = if inf p q then p else q;;
```

val max : (< getx : 'b; .. > as 'a) -> 'a -> 'a = <fun>

C'est plus une dérobade qu'une solution.

Méthodes binaires et objets amis

Slide 73

Typiquement, une méthode binaire a besoin de déstructurer son argument pour l'utiliser. L'argument doit donc montrer une partie de sa représentation. Ainsi, une classe définissant une méthode binaire doit typiquement révéler la représentation de ses objets.

(Penser par exemple à l'opération d'union sur les ensembles.)

Si l'on veut par ailleurs cacher cette représentation entre tous les objets de la même classe, on va rendre tous les objets de la même classe "amis" en un utilisant un module (voir objets amis)

Exemple

Slide 74

```
module type Point = sig
  type t
  class c : int -> int ->
    object ('a)
      method getx : t
      method inf : 'a -> bool
      method max : 'a -> 'a
    end
end;;
```

```
module Point : Point =
  struct
    type t = int
    class c x = point x
  end;;
```

Conclusion

Slide 75

La programmation avec objets est un paradigme idéal pour les données munies d'opérations unaires ou externes.

– Elle devient plus délicate et moins bien adaptée dès qu'il y a besoin d'utiliser des méthodes binaires.
– En générale, celles-ci détruisent la symétrie, par exemple `plus x y` devient `x#plus y`.

D'autres approches des objets qui traitent les méthodes comme des fonctions surchargées définies en dehors des objets sont mieux appropriées ici, mais ces approchent présentent d'autres difficultés.

1 Solutions des exercices

Exercice 1, page 10

```
class type p_sans_m =
  object method n : int end;;
class p : p_sans_m =
  object (self) method m = 1 method n = 1 + self#m end;;
```

Characters 23−75:
The class type object method m : int method n : int end
is not matched by the class type p_sans_m
The public method m cannot be hidden

Si la classe `p` était typable, alors on pourrait construire la classe `q` puis l'objet `cassé` suivant donc l'appel de la méthode `m` lancerait un calcul qui aboutirait à `1 + true`.

```
class q = object inherit p method m = true end
let cassé = new q in cassé # n;;
```

Exercice 2, page 25

La méthode `bin` est définie avec les types suivants :

Type de x	A	B	
Type de y	A	A	B
Résultat	1	1	2

La seule combinaison qui retourne `2` est telle que l'objet `x` et l'argument `y` soient tous les deux de type `B` donc de la classe `B`, soit `bb.bin(bb)`.

Autrement, `x` ou `y` est (éventuellement coercé en une valeur) de type `A` et le résultat retourné est `1`. Soit finalement

$x \setminus y$	*aa*	*ab*	*bb*
aa	1	1	1
ab	1	1	1
bb	1	1	2

Exercice 2 (continued)

```
class A          { int bin(A x){ return 1; } }
class B extends A { int bin(B x){ return 2; } }

class AB{
  static A aa = new A();
  static B bb = new B();
  static A ab = bb;
  static void print (String s, int r){
    System.out.print(s); System.out.println(r);
  }
  public static void main (String argv[]) {
    print ("aa.bin(aa)=",aa.bin(aa));
    print ("aa.bin(ab)=",aa.bin(ab));
    print ("aa.bin(bb)=",aa.bin(bb));
    print ("ab.bin(aa)=",ab.bin(aa));
    print ("ab.bin(ab)=",ab.bin(ab));
    print ("ab.bin(bb)=",ab.bin(bb));
    print ("bb.bin(aa)=",bb.bin(aa));
    print ("bb.bin(ab)=",bb.bin(ab));
    print ("bb.bin(bb)=",bb.bin(bb));
  }
```

```
}
```

Exercice 2 (continued)

La méthode première définition `bin(A x)` override (remplace) la définition de la classe `A` alors que la deuxième définition `bin(B x)` surcharge (complète) la définition précédente.

Exercice 2 (continued)

Lorsque le sujet (`x`) est de la classe `B` et que dans le premier cas, on la réponse n'était pas 2. En effet, c'était c'est la méthode de la classe `A` qui était appelé, maintenant ce sera la méthode de `B` qui a le même type que dans classe `A` donc qui prend un argument de type `A`. Ainsi

$x \setminus y$	aa	ab	bb
aa	1	1	1
ab	3	3	3
bb	3	3	2

Exercice 2 (continued)

```
class A         { int bin(A x){ return 1; } }
class B extends A{
  int bin(A x){
    if (x instanceof B) return bin((B)(x));
    else return super.bin(x);
  }
  int bin(B x){ return 2; }
}

class AB{
  static A aa = new A();
  static B bb = new B();
  static A ab = bb;
  static void print (String s, int r){
    System.out.print(s); System.out.println(r);
  }
  public static void main (String argv[]) {
    print ("aa.bin(aa)=",aa.bin(aa));
    print ("aa.bin(ab)=",aa.bin(ab));
    print ("aa.bin(bb)=",aa.bin(bb));
    print ("ab.bin(aa)=",ab.bin(aa));
    print ("ab.bin(ab)=",ab.bin(ab));
    print ("ab.bin(bb)=",ab.bin(bb));
    print ("bb.bin(aa)=",bb.bin(aa));
    print ("bb.bin(ab)=",bb.bin(ab));
    print ("bb.bin(bb)=",bb.bin(bb));
  }
}
```

Le résultat est toujours 2 dès que `x` et `y` sont tous deux de la class `B`, indépendemment de leur type statique. En effet dès que `x` est de la classe `B`, la méthode `bin` est choisie dans la classe `B` indépendemment du type (statique) de `x`. Par ailleurs, les deux implémentations de `bin` dans la classe `B` retournent 2 dès que `y` est un objet de la classe `B`.

Exercice 2 (continued)

```
class A         { int bin_A (A x){ return 1; } }
class B extends A{
```

```
  int bin_A (A x){ return 3; }
  int bin_B (B x){ return 2; }
}
```

Exercice 2 (continued)

$x \setminus y$	aa	ab	bb
aa	bin_A	`bin_A`	bin_A
ab	bin'_A	bin'_A	bin'_A
bb	bin'_A	bin'_A	bin'_B

Exercice 5, page 45

Chacune est allouée dans chaque objet. La seule différence observable est au travers de l'opératino de copie soit par `Oo.copy` ou les différentes formes de copie fonctionnelle `{< ... >}`. La référence `au_delà` sera partagée entre tous les objets issus d'un même objet par clonage ou copie ; inversement la variable mutable `gènes` sera départagée à chaque copie et donc possèse une valeur propre indépendante dans chaque objet.

Exercice 5 (continued)

Puisque la variable `au_delà` est partagée par tous les objets issus du même objet par clonage ou copie, elle permet de communiquer entre ceux-ci par “effet de bord”.

```
let d = new démon in
let d' = d#clone in
let d_to_d' x = d#pensée x; d' # intuition in
d_to_d' 497;;
```

Exercice 5 (continued)

Cette variable joue le rôle d'une variable de classe. Cette référence est partagée par tous les objets de la même classe.

Exercice 5 (continued)

Le comportement de ces deux méthodes est strictement équivalent.

Exercice 5 (continued)

La mutation ne crée par un nouveau démon, mais modifie un démon existant.

Exercice 5 (continued)

Oui, en dupplicant des démons de l'extérieur avec `Oo.copy` ce qui ne passe pas par le contrôle des population. Dans ce cas, il y a forcément des jumeaux.

Exercice 5 (continued)

Copie retourne un nouvel objet indépendant de l'original (nouvelle identité), alors que `même` retourne strictement le même objet.

Exercice 5 (continued)

À rien ! Puisque `x # même` retourne toujours `x` !

Exercice 5 (continued)

A la différence de `combine qui est une fonction auxiliaire,` mute" qui est une méthode est visible dans les sous-classes, elle peut être modifiée par liaison tardive, et a accès aux variables d'instances. Pour devenir une fonction auxilliaire il faudrait que `mute` soit abstraite par rapport à la variable d'instance `gènes`. Elles en en commun de n'est pas visibles dans les objets de la classe.

Exercice 7, page 53

```
class ['a] cell x =
  object
    val mutable contents : 'a = x
    method get = x
    method set x = contents <- x
  end;;
```

Exercice 7 (continued)

Parce que le paramètre `'a` apparaît à la fois en position positive dans la méthode `get` et négative dans la méthode `set`.

Exercice 7 (continued)

```
fun x -> (x : < a : int > cell :> < > cell);;
fun x -> (x : < > cell :> < a : int > cell);;
```

Exercice 7 (continued)

Si on cache la méthode set, le type `['a] cell` devient co-variant.

```
class ['a] cell_set x =
  object
    val mutable contents : 'a = x
    method get = x
    method private set x = contents <- x
  end;;
fun x -> (x : < a : int > cell_set :> < > cell_set);;
```

Inversement, si on cache la méthode `get`, le type `['a] cell` devient contra-variant.

```
class ['a] cell_get x =
  object
    val mutable contents : 'a = x
    method private get = x
    method set x = contents <- x
  end;;
fun x -> (x : < > cell_get :> < a : int > cell_get);;
```

Exercice 7 (continued)

Oui, car `'a` n'est pas le type de self, mais d'un paramètre. Il ne s'agit pas d'une méthode binaire. Un exemple sans intérêt, mais qui montre que tout se passe bien :

```
class ['a] cellule x =
  object (self)
    inherit ['a] cell x
    method consulte = self#set (self#get)
  end;;
```

On vérifie que `cellule` est sous-type de `cell` (mais pas l'inverse).

```
fun x -> (x : 'a cellule :> 'a cell);;
fun x -> (x : 'a cell :> 'a cellule);;
```