Introduction à la sémantique des objets en Deca (et Java)

version du 18 avril 2023 à 17:16

Contrairement à **[Gencode]** et les diapos du stage 2A, ce TD ne va pas indiquer *comment* on compile des programmes Deca objet. Il va plutôt "*rappeler*" le comportement attendu de tels programmes : ce qu'on appelle la *sémantique* de ces programmes. La sémantique des objets en Deca correspond fidèlement à celle de Java, modulo quelques simplifications (qui sont détaillées dans la correction du dernier exercice). Ce TD se contente donc d'expliciter ce que tout programmeur Java est censé bien connaître.

A Retenir En Deca (comme en Java), un objet est une *adresse* dans une zone spéciale de la mémoire appelé le *tas* : les comparaisons `==` et `!=` (resp. affectations) sur des objets (resp. variables, paramètres ou champ d'un type d'objet) correspondent donc toujours à des comparaisons (resp. copies) de simples adresses. Dans la sémantique simplifiée de Deca, on considère qu'une donnée allouée dans le tas n'est jamais désallouée. [1] Un objet a un type qui spécifie comment le programme peut utiliser l'objet, notamment les *champs* et les *méthodes* qu'il offre. Il y a en fait deux notions de type :

type statique c'est le type associé à l'objet par **[SyntaxeContextuelle]** : autrement dit, le type de l'objet vu par le compilateur.

type dynamique c'est le type X avec lequel l'objet a été créé dans le tas à l'exécution, à partir de l'expression "**new** $X()$". [2]

Ces deux types ne coincident généralement pas, comme illustré sur le programme de la figure 1.

```
1  class A { int x = 15; }
2
3  class B extends A { float x = ((A)(this)).x * 0.5; }
4
5  {
6    A oA; B oB;
7    print("Votre choix (0 ou 1):");
8    if (readInt() == 0) {
9      oA = new A();
10     oB = null;
11   } else {
12     oB = new B();
13     oA = oB;
14   }
15   println(oA.x);
16   if (oA == oB) { println(oB.x); }
17 }
```

FIGURE 1 – Typage dynamique dépendant de l'utilisateur

Exercice 1 (Type statique versus type dynamique). Sur le programme de la figure 1, indiquer les types statiques et dynamiques des variables `oA` et `oB`.

1. La machine virtuelle Java, elle, désalloue les objets lorsqu'ils ne sont plus accessibles par le programme (à partir d'une adresse de la pile, ou d'un autre objet accessible) : c'est ce qu'on appelle le *Glanage de Cellule* (ou *Garbage Collecting* en anglais). Du point de vue sémantique, ça ne fait donc pas trop de différences avec la sémantique de Deca, sauf que Java offre la possibilité au programmeur de définir une méthode `finalize` appelée à la discrétion du GC, juste avant la destruction de l'objet.

2. Dans le cas particulier de l'objet **null** (qui correspond à une adresse spéciale jamais allouée), le type dynamique est un type spécial noté null dans **[SyntaxeContextuelle]**.

Correction
La variable `oA` (resp. `oB`) est de *type statique* `A` (resp. `B`) : le type statique est donc indépendant de l'exécution du programme. Par contre, le *type dynamique* dépend du choix de l'utilisateur. Si l'utilisateur tape 0, alors `oA` (resp. `oB`) sera de type dynamique `A` (resp. `null`). Sinon, `oA` et `oB` seront de même type dynamique `B` : ce qui découle du fait, que ces deux variables représentent le *même* objet (i.e. la même adresse du tas).

A Retenir La sémantique de Deca (comme celle de Java) garantit l'invariant suivant au programmeur

Le type dynamique d'un objet est toujours un sous-type de son type statique.

En effet, sur une exécution donnée, le type dynamique d'un objet donné est fixé une fois pour toute (via le `new`). Par contre, son type statique change au gré des *conversions de type statique* (*cast* en anglais) implicites ou explicites du programme. Le cast vers un super-type (appelé *upcast*, c'est le cas de tous les casts implicites) préserve cet invariant sans problème. Le cast vers un sous-type (appelé *downcast*) engendre une vérification à l'exécution pour garantir l'invariant ci-dessus : une erreur à l'exécution empêche toute violation de l'invariant.

Exercice 2 (Casts implicites). Indiquez toutes les conversions implicites de type statique sur le programme de la figure 1.
NB : un tel cast correspond à un *assign_compatible* ou un *type_binary_op* qui acceptent des types non identiques dans les règles (3.28) et (3.33) de **[SyntaxeContextuelle]**.

Correction
Voici la liste de tous les casts implicites :

ligne 3 du type `int` vers `float` (en opérande gauche du `*`) ;
ligne 10 du type `null` vers le type `B` ;
ligne 13 du type `B` vers le type `A` ;
ligne 16 dans le test "`oA == oB`" où `oA` et `oB` sont castés sur le type `Object`, le super-type de tous les types d'objet.

Par contre, le cast "`(A)(this)`" ligne 3 est explicite (même si c'est un upcast à l'exécution).

Exercice 3 (Sélection statique des champs). L'exemple de la figure 1 montre une classe `B` qui définit un champ `x`, déjà défini dans sa classe mère `A`. Les deux champs coexistent donc au sein des objets de type dynamique `B` : à l'exécution le champ `x` est selectionné en fonction du *type statique*. En déduire le comportement de ce programme à l'exécution : qu'affiche-t-il si l'utilisateur tape 0 ? qu'affiche-t-il s'il tape 1 ?

Correction
Si l'utilisateur tape 0, il voit :

```
Votre choix (0 ou 1):0
15
```

Le `println` de la ligne 16 n'est pas exécuté (puisque la condition du `if` a répondu `false`).

Par contre, si l'utilisateur tape 1, il voit :

```
Votre choix (0 ou 1):1
15
7.50000e+00
```

Notons que le upcast explicite "`(A)(this)`" ligne 3 est nécessaire pour pouvoir faire référence au champ `x` hérité de la classe `A` au sein de la classe `B`.

A Retenir Un objet est donc un pointeur sur un enregistrement (*record* en anglais) stocké dans le *tas*, qui contient d'une part les *champs* de l'objet, et d'autre part le *type dynamique* de l'objet. Le type dynamique est l'adresse d'une table (typiquement stockée en zone statique du programme ou dans la fenêtre de pile du bloc principal) qui contient d'une part pour chaque nom de méthode l'adresse du code à exécuter, et d'autre part le pointeur vers son super-type immédiat[3]. En effet, d'une part, lors d'un appel de méthode "`o.m(...)`", l'adresse du code à exécuter est sélectionnée en fonction du type dynamique de cet objet `o`.[4] D'autre part, lors d'un cast explicite ou d'un `instanceof`, le programme parcourt la liste chaînée des super-types du type dynamique pour connaître l'ensemble des super-types possibles de l'objet. Par contre, comme on l'a vu précédemment, la sélection des champs se base uniquement sur le *type statique*.[5]

Exercice 4 (Sélection dynamique des méthodes). L'exemple de la figure 2 montre une classe `B` qui définit une méthode `equals`, déjà définie dans sa classe mère `A` (et en fait déjà définie au niveau de `Object`). En Deca, un objet ne peut avoir qu'une seule méthode d'un nom donné : la méthode `equals` pour les objets de type dynamique `B` *redéfinit* (*overrides* en anglais) celle héritée de la classe `A`. En déduire le comportement de ce programme à l'exécution.

Correction
Pour clarifier, on préfixe ici les noms de champs et de méthodes par le nom de la classe qui les définit : `A.x` vs `B.x` et `A.equals` vs `B.equals`. Comme `test` est définie dans la `A`, même si elle est héritée dans `B`, on écrit `A.test`. Exécution commentée :

```
{
  A oB=new B(), oA=new A();
    // oB -> {type=B; A.x=3.5; B.x=11}  et  oA ->  {type=A; A.x=3.5}
  oA.test(oB);
    // appelle A.test sur this=oA et o=oB
    // qui elle-même appelle A.equals sur this=oA et o=oB
    // qui elle-même appelle A.equalsA sur this=oA et a=oB
    // qui elle-même retourne true
  oB.test(oA);
    // appelle A.test sur this=oB et o=oA
    // qui elle-même appelle appelle B.equals sur this=oB et o=oA
    // qui retourne false (car le instanceof répond false).
  oB.test(new B());
    // crée un objet oB' -> {type=B; A.x=3.5; B.x=11}
    // et appelle A.test sur this=oB et o=oB'
    // qui elle-même appelle appelle B.equals sur this=oB et o=oB'
    // qui elle-même appelle B.equalsB sur this=oB et b=oB'
    // qui elle-même appelle A.equalsA sur this=oB et a=oB'
    // qui elle-même retourne true
}
```

3. Par convention, on signifie qu'`Object` n'a pas de super-type immédiat avec l'adresse nulle : c'est le type noté 0 dans [**SyntaxeContextuelle**], à ne pas confondre avec le type null (type de l'objet **null**).

4. Un appel de méthode correspond donc à deux déréférencements de pointeur suivi d'un saut indirect : d'abord, on déréférence l'objet `o` (avec un déplacement) pour connaître son type dynamique, ensuite on déréférence le type dynamique (avec un déplacement) pour accéder à l'adresse de la méthode ; enfin, il y a un saut indirect pour exécuter la méthode elle-même.

5. Un accès de champ correspond donc à seul déréférencement de pointeur : on déréférence l'objet (avec un déplacement) pour récupérer la valeur du champ stockée dans le tas.

```
1  class A {
2    float x = 3.5;
3
4    boolean equalsA(A a) {
5      print("A:", x, ";");
6      return a != null && a.x==x;
7    }
8
9    boolean equals(Object o) {
10     return o instanceof A && equalsA((A)(o));
11   }
12
13   void test(Object o) {
14     if (equals(o)) { println("Eq"); }
15     else { println("Diff"); }
16   }
17 }
18
19 class B extends A {
20   int x = 11;
21
22   boolean equalsB(B b) {
23     print("B:", x, ";");
24     return equalsA(b) && b.x==x;
25   }
26
27   boolean equals(Object o) {
28     return o instanceof B && equalsB((B)(o));
29   }
30 }
31
32 {
33   A oB=new B(), oA=new A();
34   oA.test(oB);
35   oB.test(oA);
36   oB.test(new B());
37 }
```

FIGURE 2 – Sélection dynamique des méthodes vs sélection statique des champs

Au final, chacun des appels à `test` provoque l'affichage d'une seule ligne (un seul `println`), et on obtient donc les trois lignes suivantes :

```
A:3.50000e+00;Eq
Diff
B:11;A:3.50000e+00;Eq
```

Exercice 5 (Surcharge de méthode). On considère le programme Deca obtenue à partir de la figure 2 en remplaçant tous les identificateurs `equalsA` par `equals`. Est-ce que ce nouveau programme est accepté par les compilateurs Deca ? Si oui, donner le comportement à l'exécution. Sinon, indiquer la règle de [**SyntaxeContextuelle**] qui rejette le programme.

Correction
Non, le programme est rejeté par [**SyntaxeContextuelle**] à cause de la règle (2.7). Dans la déclaration de la classe `A`, la méthode `equals` déclarée à la ligne 4 (qui était `equalsA` sur la figure 2) n'a pas une signature identique à celle de sa super-classe `Object`. Autrement dit dans la règle (2.7), c'est le test $sig = sig_2$ qui échoue, où sig est le profil des arguments de la méthode en train d'être définie, et sig_2 celui de la classe héritée. De plus, même si `equals` n'avait pas été déjà définie dans `Object`, alors c'est l'opérateur $\oplus$ de la règle (2.6) qui interdit de définir deux méthodes de même nom dans la même classe.

A Retenir Ici, Deca apporte une simplification notable vis-à-vis de Java. En effet, Java accepte qu'un objet ait deux méthodes qui portent le même nom : c'est ce qu'on appelle la *surcharge* (*overloading* en anglais) à bien distinguer de la redéfinition de méthode au cours de l'héritage. Les méthodes de même nom sont distinguées par le type (statique) de leurs arguments. La redéfinition opère uniquement sur les méthodes qui ont exactement le même profil d'arguments. La sélection de méthode est alors plus complexe. La méthode à exécuter est d'abord choisie selon le *type dynamique* de l'objet surlequel elle est invoquée (comme en Deca), et si plusieurs méthodes (avec des profils d'arguments différents) sont possibles, alors on choisit celle qui minimise la "longueur" des chaînes de upcast à réaliser sur le *type statique* des arguments. Sur l'équivalent Java de l'exemple de la figure 2, ce mécanisme fait que le renommage de `equalsA` par `equals` préserve le comportement du code dans le bloc principal. Mais un tel renommage ne préserve pas toujours les comportements : par exemple, si on renomme simulaténement `equalsA` et `equalsB` par `equals`, alors l'appel à `oB.test(new B())` ligne 36 provoque un appel récursif sans fin ligne 24. De plus, en Java, il reste des appels de méthode ambiguës. Par exemple, dans la méthode `bar` de la classe ci-dessous :

```
class A {

  void foo(A x, Object y) {
    System.out.println("foo1");
  }

  void foo(Object x, A y) {
    System.out.println("foo2");
  }

  void bar(){
    this.foo(this,this);
  }

}
```

Ces ambiguïtés sont détectées par le compilateur Java qui signale typiquement l'erreur suivante dans la méthode `bar` :

```
Error: reference to foo is ambiguous
    this.foo(this,this);
         ^
  both method foo(A,Object) in A and method foo(Object,A) in A match
1 error
```

En pratique, le programmeur peut lever l'ambiguïté en faisant un cast explicite à la main. Par exemple, pour choisir `foo2`, il écrit :

```
  void bar(){
    this.foo((Object)this,this);
  }
```