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]**.

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**].

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 ?

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.

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.

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