

TD Feuille 1 — Sémantiques des langages de programmation

Exercice 1 (Ordre d'évaluation et *Undefined Behavior*). On compare la sémantique de l'affectation "`t[++x] = x+b;`" en Java et C. Dans les deux cas, on rappelle qu'une telle affectation est une expression. Dans le cas de Java, les expressions sont évaluées de gauche à droite. Dans le cas du C, le choix de l'ordre d'évaluation est laissé au compilateur.

▷ **Question 1.** Décrire d'abord la sémantique Java de cette affectation dans le contexte : `int x=0, t[]=new int[]{ 3, 5 }, b=17;`. Et ensuite en changeant ce contexte avec `x = 1;`

▷ **Question 2.** Décrire maintenant la sémantique C de cette affectation dans le contexte : `int x=0, t[]={ 3, 5 }, b=17;`. Et ensuite en changeant ce contexte avec `x = 1;`

▷ **Question 3.** Dans le contexte de typage "`int x, t[2], b;`", un compilateur C aux optimisations très agressives pourrait transformer l'instruction "`if (x>=0) t[++x]=x+b;`" en "`if (x>=0) {x=1; t[1]=b;}`". Expliquez pourquoi ce serait correct. Est-ce qu'un compilateur Java aurait le droit de faire une transformation similaire?

▷ **Question 4.** Quels sont les avantages/inconvénients de ne pas fixer l'ordre d'évaluation comme en C? Quels sont les avantages/inconvénients des comportements indéfinis du C vis-à-vis des erreurs à l'exécution à la Java?

Exercice 2 (Boucles For). En Python, le nombre maximal d'itérations d'une boucle `for` se déduit directement de ses bornes : elles sont évaluées une fois pour toutes avant l'entrée dans la boucle. Au contraire, en C, le `for` est un raccourci syntaxique pour une boucle `while`.

▷ **Question 1.** Décrire le comportement de chaque itération ci-dessous pour la sémantique Python à gauche, et la sémantique C à droite. En Python, ces itérations ont lieu dans le contexte "`n=6`"; et en C, dans le contexte "`int i, n=6;`".

<pre>1. for i in range(1, 6): i = i-1 print(i) print("i=", i)</pre>	<pre>for (i=1; i<6; i++){ i = i-1; printf("%d\n", i); } printf("i = %d\n", i);</pre>
<pre>2. for i in range(1, n): n = i+2 print(n) print("i =", i)</pre>	<pre>for (i=1; i<n; i++){ n = i+2; printf("%d\n", n); } printf("i = %d\n", i);</pre>
<pre>3. for i in range(6, n): print(i) print("i =", i)</pre>	<pre>for (i=6; i<n; i++){ printf("%d\n", i); } printf("i = %d\n", i);</pre>

► **Question 2.** On imagine qu'on veut étendre le C avec une syntaxe alternative des boucles `for` de la forme "`for i in e1..e2 { S }`" où i est un nom de variable (supposée déjà déclarée de type `int` dans le contexte), e_1 et e_2 sont deux expressions de type `int`, et S une séquence d'instructions.

On demande d'écrire une transformation—implémentable dans le parseur du compilateur—qui encode cette boucle dans le reste du langage, sans utiliser de boucle `for` (il faudra utiliser une boucle `while` à la place). Implémenter successivement les deux sémantiques décrites ci-dessus : d'abord la sémantique C, ensuite la sémantique Python (comme en Python, il faudra évaluer e_1 avant e_2).

Exercice 3 (Passages de paramètres). On s'intéresse dans cet exercice à définir la sémantique de différents modes de passage de paramètres.

Soit "`void p(T x){ S }`" la définition d'une procédure p avec x le paramètre formel de type T et S le corps de la fonction, dans un langage ayant la même syntaxe que le langage C. On considère les différentes sémantiques par dépliages suivantes pour l'appel "`p(v)`", avec v une variable (ou plus généralement une *lvalue*) :

passage par copie l'appel est déplié en "`{ T x = v; S }`"

passage par adresse l'appel est déplié en "`{ T *x = &v; S[x ← *x] }`"

où la notation $S[x ← t]$ désigne le texte obtenu en remplaçant dans S les occurrences libres de x par le terme t et sans capture des variables libres de t par une variable liée de S .

On considère le programme suivant :

```

1 struct toto { int a[1000]; } n;
2
3 void p(struct toto x){
4     {
5         int x = n.a[0];
6         n.a[0] = ++x;
7     }
8     x.a[0]++;
9     printf("x: %d\n", x.a[0]);
10 }
11
12 void main() {
13     n.a[0] = 2;
14     p(n);
15     printf("n: %d\n", n.a[0]);
16 }

```

► **Question 1.** Appliquer chacun de ces modes de passage à l'exemple ci-dessus. Quelle est la différence à l'exécution ?

► **Question 2.** Quels sont les avantages/inconvénients de ces modes de passage de paramètre ? Quel est celui du C ?

Exercice 4 (Garanties sémantiques et raisonnements formels sur les programmes). On considère ci-dessous une fonction C (à gauche) et son “analogue” en Rust (à droite). Elles sont spécifiées chacune dans une logique de programme dédiée : Frama-C (<https://frama-c.com/>) à gauche, et Creusot (<https://github.com/creusot-rs/creusot>) à droite. On va ici comparer le raisonnement formel dans ces deux systèmes, et expliquer pourquoi la version Frama-C est plus verbeuse.

<pre> /*@ requires 0 <= x && 0 < y && @ \valid(q) && \valid(r); @ assigns *q, *r; @ ensures x == *q * y + *r && @ 0 <= *r < y; */ void div(int x, int y, int *q, int *r) { *q = x / y; *r = x - *q * y; } </pre>	<pre> #[requires(0i32 <= x && 0i32 < y)] #[ensures(x == ^q * y + ^r && 0i32 <= ^r && ^r < y)] fn div(x: i32, y: i32, q: &mut i32, r: &mut i32){ *q = x / y; *r = x - *q * y; } </pre>
--	---

Dans les deux cas “requires” est une *précondition* que le contexte d’appel doit satisfaire sur l’état initial (avant l’appel). Le “ensures” est une *postcondition* satisfaite par l’état final (cf. CMs). Côté Rust, les paramètres q et r sont des pointeurs - qui sont des *emprunts mutables*. Dans la sémantique du langage, un tel pointeur est nécessairement non nul et alloué, et tant qu’il est *actif*, il garantit un accès exclusif en écriture à la donnée (autrement dit, il ne peut pas être alié à un autre pointeur actif). Dans la postcondition en Creusot, ^q et ^r représentent les valeurs pointées en fin d’appel (alors que *q et *r représenteraient leur valeur avant l’appel).

► **Question 1.** Le prouveur Creusot valide la spécification donnée ci-dessus (ce qui garantit qu’elle est correcte). Par contre, si on modifie la précondition sur y en `0i32 <= y`, la validation échoue. Pourquoi l’implémentation est incorrecte dans ce cas ? Y a-t-il un moyen de la corriger pour la rendre correcte ?

Dans la postcondition Frama-C, *q et *r représentent aussi les valeur pointées en fin d’appel (il faudrait écrire `\old(*q)` et `\old(*r)` pour désigner les valeurs avant l’appel). La clause `assigns` représente un sur-ensemble des cases mémoires modifiées par l’appel de la fonction (contrairement à Rust, en C, il est en général difficile d’en calculer statiquement une sur-approximation précise). En l’absence de cette clause, Frama-C considère par exemple que les variables globales du contexte d’appel peuvent être modifiées par l’appel (ce qui invalide certaines preuves sur le contexte d’appel). Enfin, les préconditions `\valid(q)` et `\valid(r)` indiquent que q et r sont des pointeurs non-nuls et alloués, avec des permissions en lecture et écriture (cette garantie étant implicite avec les emprunts mutables de Rust).

► **Question 2.** Malgré toutes ces précisions, Frama-C ne valide pas la spécification C. Exhibez un contre-exemple, qui montre que l’implémentation est incorrecte. Y a-t-il un moyen de la corriger pour la rendre correcte ? Sinon, comment faut-il corriger la spécification pour éliminer ce contre-exemple ?