Vérification de programmes par interprétation abstraite

Marc Chevalier

Équipe Antique

14 septembre 2020

Quoi ? Pourquoi ?

Soit. Mais comment ?

Le passé et l'avenir

Quoi ? Pourquoi ?

Soit. Mais comment ?

Le passé et l'avenir

À quoi sert l'interprétation abstraite ?

Une méthode pour prouver des propriétés sur des programmes. Par exemple :

- pas de comportement indéfinis,
- la spécification sur le résultat est satisfaite,
- temps d'exécution maximum, chemin d'exécution constant,
- ...tout autre propriété sémantique à laquelle on peut penser.

Pourquoi ? – Quand les logiciels fautent

Figure 1 – Ariane 5, 4 juin 1996

Pourquoi ? – Quand les logiciels fautent

Figure 2 – Ariane 5, 40s plus tard

Pourquoi ? – Le coût des bugs

Les bugs ont différentes conséquences fâcheuses :

- Morts (Patriot MIM-104, Toyota, machines de radiothérapie Therac-25)
- Beaucoup d'argent : Ariane 5 (charge utile : $\$370 \cdot 10^6$), $\$60 \cdot 10^9$/an aux USA (NIST en 2002)
- Confidentialité (Heartbleed)
- ...

Pourquoi ? – Ce qu'on fait d'habitude

Comment les développeurs croient qu'ils peuvent éviter les bugs :

- Tests
- Langage de haut niveau
- Style de code strict

Et pourtant, Ariane 5 s'est crashée... « And here, poor fool[s], with all [their] lore, [they] stand no wiser than before ».

Quoi ? Pourquoi ?

Soit. Mais comment ?

Le passé et l'avenir

L'idée

- Vérifier une exécution : test, limité.
- Vérifier toutes les exécutions d'un coup : ça marche, mais c'est pas calculable.
- Calculer une sur-approximation de toutes les exécutions : correct, mais pas complet.

Tous les comportements possibles sont dans notre sur-approximation (et sans doute plus).

Un exemple

```
1  int f(int x)
2  {                    // x ∈ [−2^31; 2^31 − 1]
3      y = abs(x);      // y ∈ [0; 2^31 − 1] ∨ x = −2^31
4      z = y + 1;       // z ∈ [1; 2^31 − 1] ∨ y = 2^31 − 1
5      return 1/z;      // 0 ∉ [1; 2^31 − 1] ⇒OK !
6  }
```

Généralisons

Soit $(D, \subseteq)$ un trop gros ensemble partiellement ordonné (avec de bonnes propriétés) : typiquement, des ensembles d'environnements mémoire.

$$[\![P]\!] = f_1 \circ \cdots \circ f_n$$

On veut $c \subseteq$ specification en tout point de programme.

Généralisons

Domaine abstrait :

- $(D^\sharp, \subseteq^\sharp)$: un ensemble raisonnable (p. ex. $\overline{\mathbb{Z}}^2$)
- $\gamma : D^\sharp \to D$: concrétisation (p. ex. $(a, b) \mapsto \{x \in \mathbb{Z} \mid a \leqslant x \leqslant b\}$)

Correct si en tout point de programme $c \subseteq \gamma(a)$: on ne rate aucun comportement en exécutant dans l'abstrait (mais on perd de la précision).

Opérateur abstrait correct : $f_i \circ \gamma \subseteq \gamma \circ f_i^\sharp$.

Et on veut $\gamma(a) \subseteq$ specification.

Généralisons

$$\begin{array}{ccc} [-1;1] & & \\ \gamma\downarrow & & \\ \{-1,0,1\} & \xrightarrow{\times 2} & \{-2,0,2\} \end{array} \qquad \begin{array}{ccc} [-1;1] & \xrightarrow{(\times 2)^\sharp} & [-2;2] \\ & & \downarrow\gamma \\ & & \{-2,-1,0,1,2\} \end{array}$$

On a tous les résultats possibles en exécutant dans l'abstrait.

Au quotidien

N'importe quelle méthode pour déduire une propriété sur un système sans tout connaitre :

- Règle des signes pour la multiplication :

$\times$	$+$	-
$+$	$+$	-
-	-	$+$

- Dépouillement

L'incomplétude

```
1  /*@ requires -10 <= x <= 10; */
2  int g(int x)
3  {                        // x ∈ [−10, 10]
4      int y = x;           // y ∈ [−10, 10]
5      int z = x * y;
6      /* z ∈ Interval({a × b | a ∈ [−10, 10], b ∈ [−10, 10]})
7      z ∈ [−100, 100]
8      */
9      int t = z + 1; // t ∈ [−99, 101]
10     return 1/t;    // 0 ∈ [−99, 101] ⇒ Alarme !
11 }
```

Mais ce programme est trivialement sûr.

Qu'est ce qui s'est passé ? Ce domaine abstrait ne peut pas comprendre la relation entre `x` et `y`.

L'incomplétude au quotidien

Parfois, une connaissance partielle est insuffisante :

- La règle des signes pour l'addition :

+	+	-
+	+	?
-	?	-

- Comptage des votes sans majorité absolue

D'autres domaines

- Numérique :
  Non-relationnel :
  - Modulo : $x_i \equiv c_i[n_i]$
  - Bitwise : $x_i = 0?1??100010111????$
  - Sign : $x_i < 0$, $x_i > 0$, $x_i \leqslant 0$ ...

  Relationnel :
  - Polytope : $\sum a_i x_i \leqslant c_i$
  - Octogone : $\pm x_i \pm x_j \leqslant c_i$

  Et combinaisons de domaines :
  $x \equiv [4;5][10] \wedge x \in [\![0,4]\!] \Rightarrow x \equiv 4[10] \wedge x \in [\![4,4]\!]$
- Mémoire : une variable pointe sur une autre, structures mémoire, logique de séparation...
- Partitionnement : $(x > 0 \Rightarrow ...) \wedge (x \leqslant 0 \Rightarrow ...)$
- N'importe quel domaine ad hoc nécessaire.

Quoi ? Pourquoi ?

Soit. Mais comment ?

Le passé et l'avenir

L'interprétation abstraite à l'épreuve de la réalité

Les bons côtés :

- Ça marche sur du code déjà écrit.
- Ça marche pour de vrai : Astrée (A340, A380)
- Plutôt automatique (quand on a les domaines qui vont bien)
- Le développeur qui connait son code peut aisément aider l'analyseur

Et les mauvais :

- Incomplétude
- Beaucoup de travail si les domaines existants ne sont pas assez expressifs
- Certaines propriétés sont très difficiles à prouver avec cette méthode (p. ex. terminaison).

Mon œuvre – L'assembleur

Cas d'étude : l'OS d'une plateforme d'accueil dans les avions, à la frontière du monde de confiance (contrôle de vol) et du monde sauvage (potentiellement hostile).

Nous voulons prouver des propriétés de sécurité : isolation de la mémoire, les applications hébergées ne montent pas en privilège (ring 3)...

Ces propriétés ne sont pas visibles du C (principalement à propos des registres du CPU) : assembleur inline $\Rightarrow$ analyser des mélanges de C et d'assembleur.

Mon œuvre – L'assembleur

```
1  int a = 42;
2  int f() {
3      int b = 1337;
4      register int c;
5      asm {
6          mov EAX, a
7          add EAX, b[EBP]
8          mov c, EAX
9      }
10     return c;
11 }
```

Imaginez des appels sans retours, des retours sans appels, des appels de C depuis l'assembleur, et tout type d'interaction possible.

Mon œuvre – L'assembleur

Surtout du C : on doit analyser de l'assembleur dans une analyse conçue pour le C.

En quoi le x86 est si différent du C :

- Sauts inter-fonctions vs blocs et sauts locaux,
- Sauts dynamiques vs CFG statique,
- Registres type-agnostique vs Programmes statiquement typés,
- Registres et pile... vs Indépendant de l'architecture et de l'implémentation.

Mon œuvre – Les variables fantômes

En plus de l'assembleur, les logiciels de bas niveau font des choses étonnantes (couper des pointeurs en morceaux).

Il faut utiliser des variables fantômes (des quantités qui n'apparaissent pas vraiment dans le programme) tout en faisant coopérer des domaines.

Mon œuvre – Les variables fantômes

```
1  int* f(int* input) {
2      int high = input >> 16;
3      int low = input & 0xffff;
4      input = 0;
5      int* output = low | (high << 16);
6      return output;
7  }
```

Facile, si on se souvient de la valeur originale de `input`.

L'avenir

En général, les domaines dépendent fortement du cas d'étude : il faut construire une colossale collection de domaines complets et d'interfaces avec des outils externes.

Pour les OS, la pagination est un problème majeur. On peut aussi citer l'ordonnancement, le matériel externe...