Introduction à la vérification de programmes

Polytech Paris-Saclay, PEIP 2, Informatique Option S3

Thibaut Benjamin, Henry Saudubray, Philippe Volte–Vieira

11 Décembre 2025

Cours 1

Objectifs et modalités

Déroulé du cours

- 3 cours/TD
- 2 TP
- Un contrôle lors du dernier TP

Question centrale

- Dans le cours d'info S3, vous avez vu comment on pouvait analyser la complexité d'un programme pour déterminer combien de temps il allait mettre.

- Dans ce cours on va se poser la question de comment s'assurer qu'un programme est correct.

Pourquoi pas juste le tester ?

- Si le lance mon programme de tri sur un tableau et qu'il me renvoie bien le tableau trié
  Mon programme est-il correct, ou est-ce qu'il contient une erreur, mais j'ai eu de la chance de ne pas tomber dessus ?

- Si je lance un programme et qu'il est encore en train de tourner
  Est-ce j'ai fait une boucle infinie, ou est-ce qu'il faut juste que j'attende encore ?

- Dans certaines situations, on ne veux pas vraiment lancer des tests
  Guidage d'une fusée, gestion d'une centrale nucléaire...

Deux grands axes

- Le programme ne contient pas de bugs
  Qu'est-ce qu'un bug ?

- Le programme fait bien ce qu'il est censé faire
  Comment définir ce que le programme est censé faire, et comment le vérifier ?

Absence de bug

- Pas de boucles infinies (Séance 1 : les variants)
- Pas de dépassements arithmétiques
- Pas de d'accès à des zones mémoires non initialisées ou libérées
- Pas de déréférencement de pointeur nul

Le programme fait bien ce qu'il est censé faire

- Spécification : définir ce que le programme est censé faire
  - Séance 2 : Les contrats de fonction

- Vérification : le programme implémente bien sa spécification
  - Parfois partiellement automatisée : Séance 3 : Les invariants, model checking, solveurs smt, interprétation abstraite
  - Parfois manuelle Les assistants de preuve interactifs.

Séance du jour : Variants de boucle et terminaison des programmes

- La notion de variant de boucle
- Exercices de calcul de variants
- Bonus pour aller plus loin : problème de l'arrêt, ordres bien fondés

Variants de boucles et terminaison des programmes

Premier exemple

```
int main (){
  for (int i = 0; i < 10; i++){
    cout << "valeur de i: " << i << endl;
  }
  return 0;
}
```

comment justifier que la boucle va toujours s'arrêter ?

Et pour celle-ci ?

```
int main (){
  for (int i = 0; i < 10; i++){
    i = i-1;
  }
  return 0;
}
```

Et pour celle-ci ?

```c
int main (){
  for (int i = 0; i < 10; i++){
    i = i-1;
  }
  return 0;
}
```

Cette boucle est infinie ! Il faut trouver un critère qui permette de distinguer les deux !

La notion de variant

- Un variant est en entier positif qui décroît à chaque passage au point d'entrée dans la boule.

- Toute suite d'entier décroissante est nécessairement finie, donc on fait au plus un nombre fini de passages dans la boucle !

Premier exemple de variant

```
int main (){
  for (int i = 0; i < 10; i++){
    //@variant 10-i;
    cout << "valeur de i: " << i << endl;
  }
  return 0;
}
```

La quantité `10-i` est toujours positive et décroit à chaque passage au point où le variant est indiqué.

Premier exemple de variant

```
int main (){
  for (int i = 0; i < 10; i++){
    //@variant 10-i;
    cout << "valeur de i: " << i << endl;
  }
  return 0;
}
```

La quantité `10-i` est toujours positive et décroit à chaque passage au point où le variant est indiqué.

Démonstration avec Why3

Exemple de programme qui ne termine pas

```
int main (){
  for (int i = 0; i < 10; i++){
    //@variant 10-i;
    i = i-1;
  }
  return 0;
}
```

La quantité `10-i` est toujours positive mais ne décroit pas à chaque passage.

Exemple de programme qui ne termine pas

```
int main (){
  for (int i = 0; i < 10; i++){
    //@variant 10-i;
    i = i-1;
  }
  return 0;
}
```

La quantité `10-i` est toujours positive mais ne décroit pas à chaque passage.

Démonstration avec Why3

Affichage des valeurs d'un tableau

Exercice

Donner un variant pour la boucle dans le programme suivant :

```
int afficherTableau (int n, int tab){
  for (int i = 0; i < n; i++){
    cout << tab[i] << " ";
  }
  return 0;
}
```

Sommes partielles d'un tableau

Exercice

Donner un variant pour les boucles dans le programme suivant :

```
void sommesPartielles(int n, int tab[]){
  for(int i = 0; i < n; i++){
    int somme = 0;
    for (int j = 0; j <= i; j++){
      somme = somme + tab[j];
    }
    cout << somme << " ";
  }
}
```

La fonction fusion du tri fusion

Exercice
Donner un variant pour les boucles dans le programme suivant :

```
void fusion (int tab[], int gauche, int milieu, int droite){
  int annexe [droite - gauche];
  int i = gauche, j = milieu, k = 0;
    while (i < milieu && j < droite) {
        if (tab[i] <= tab[j]) {
            annexe[k] = tab[i]; i++; k++;
        }
        else {
            annexe[k] = tab[j]; j++; k++;
        }
    } // reste de la fonction de fusion
}
```

Enumération des nombres triangulaires

Exercice
Donner un variant pour les boucles dans le programme suivant :

```
void nombresTriangulaires(int n) {
  int tr = 0, i = 1;
    while (tr < n) {
        tr = tr + i;
        i++;
        cout << i << " ";
    }
}
```

Pour prouver ce variant, il faudra s'appuyer sur une propriété qui reste vraie le long de l'exécution du programme.

Le PGCD, version itérative

Exercice
Donner un variant pour les boucles dans le programme suivant :

```
int pgcd (int a, int b){
  while(a != b){
    if (a > b) { a = a - b; }
    else{ b = b - a; }
  }
  return a;
}
```

Pour prouver ce variant, il faudra s'appuyer sur une propriété qui reste vraie le long de l'exécution du programme.

Le PGCD, version itérative

Exercice

Donner un variant pour les boucles dans le programme suivant :

```
void course (int n){
  int participant [3] = {0,0,0};
  while(participant[0] < n &&
        participant[1] < n &&
        participant[2]<n){
    participant[rand()%3] ++;
  }
}
```

Pour prouver ce variant, il faudra s'appuyer sur une propriété qui reste vraie le long de l'exécution du programme.

Bonus : Le problème de l’arrêt

Pourquoi s'embêter ?

- Pourquoi y-a-t-il besoin de donner un variant un variant à la main ? Est-ce que le programme ne pourrait pas deviner tout seul qui est le variant ?

- Plus généralement : on voudrait programme qui prend en entrée n'importe quel programme avec des entrées et décide si ce programme termine sur ces entrées.
  Problème de l'arrêt

- Un tel programme ne peut pas exister
  Le progblème de l'arrêt est indécidable.

Pourquoi un tel programme n'existe pas

```
//On suppose que l'on a une fonction
bool arret (string programme, string argument)

void contradiction (string programme){
  if (!arret(programme, programme)){
    return;
  }
  else
    while(true){}
}
```

Que fait `contradiction(contradiction)` ?

Bonus : Les ordres bien fondés

Variants et ordres bien fondés

- Il n'existe pas de suite infinie décroissante d'entiers positifs

- On peut généraliser cela. Un ordre bien fondé est un ensemble équipé d'une relation d'ordre pour laquelle il n'existe pas de suite infinie décroissante.

- Plutôt que de donner des variants entiers positifs, on peut les donner pour n'importe quel ordre bien fondé.

Exemple (pour la culture) : les suites de Goodstein

- Une suite de Goodstein
  - On part d'un nombre écrit en binaire

$$a_m 2^m + \ldots + a_1 2^1 + a_0 2^0$$

  - On change tous les 2 en 3 et on soustrait 1

$$a_m 3^m + \ldots + a_1 3^1 + a_0 3^0 - 1$$

  - On répète l'opération, on change tous les 3 en 4 et on soustrait 1, etc.

- Théorème : Toutes les suites de Goodstein finissent par tomber à 0
  - Si on écrit un programme pour calculer tous les éléments d'une suite de Goodstein, et que l'on veut prouver que ce programme termine :
  - On est obligé d'introduire des variants dans les ordinaux.