Introduction à la vérification de programmes

Polytech Paris-Saclay, PEIP 2, Informatique Option S3

Thibaut Benjamin, Henri Saudubray, Philippe Volte–Vieira

5 Janvier 2026

Cours 3

Récapitulatif

Dans les cours précédents, nous avons vu :

- La méthode du variant pour prouver la terminaison des boucles.

- Les contrats de fonctions comprenant
  - Les pré-conditions : ce qu'il faut vérifier à chaque appel de fonction
  - Les variants : pour prouver le terminaison des fonctions récursives
  - Les post-conditions : ce qui est vrai à la sortie de la fonction.

Séance du jour

- Les invariants de boucles : pour prouver ce que fait une boucle.

- Une preuve d'un algorithme de tri

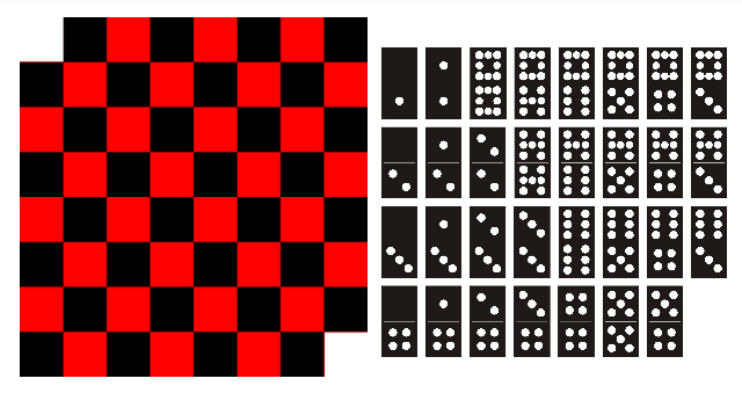
Les invariants de boucle

Les invariants de boucle

Introduction aux invariants

Un exemple informel

On considère un échiquier dans lequel on a enlevé deux coins opposés. On souhaite recouvrir cet échiquier complètement avec des dominos, qui doivent recouvrir deux cases, sans se chevaucher. Est-ce que c'est possible ?



Solution : Un invariant

- Chaque un domino on recouvre une case rouge et une case noire.
  Notons N_k le nombre de case noires non recouvertes à l'étape k et R_k le nombre de cases rouges non recouvertes à l'étape k.

- On a donc $N_{k+1} - R_{k+1} = N_k - R_k$: La quantité $N - R$ est un *invariant* : elle garde la même valeur tout le long de la procédure.

- On démarre avec aucune case recouverte : $N_0 - R_0 = 2$. On ne peut jamais recouvrir toutes les cases, car cela impliquerait d'atteindre un état avec $N_k - R_k = 0 \neq 2$.

Analyse de la méthode de l'invariant

- On a remarqué que chaque nouvelle étape de notre programme satisfait une condition.
- Cela nous a permis de trouver un invariant.
- Grâce à cet invariant, on a pu transmettre de l'information du début à la fin du programme, pour prouver l'impossibilité de la question.

Les invariants en informatique

- En informatique, on va utiliser des invariants lorsque l'on a des boucles.

- Un invariant est une proposition qui satisfait deux critères :
  - Elle est vraie au moment de la première entrée dans la boucle.
  - Si elle est vraie au début d'une itération, elle est aussi vraie au début de la suivante.

- Cela implique que la propriété est vraie à chaque entrée dans la boucle. C'est une technique de raisonnement par récurrence.

Les invariants de boucle

Premiers exemples : invariants pour prouver les variants

Les invariants en Why3

- En Why3, on peut déclarer un invariant associé à une boucle à l'aide du mot-clé `//@invariant`

- Il est souvent nécessaire pour Why3 d'avoir un invariant afin de prouver les variants.

Un premier exemple, l'addition naive

```
int additionBete (int a, int b){
  //@requires b>0;
  while (b != 0){
    //@variant b;
    //@invariant b>=0;
    a = a + 1;
    b = b - 1;
  }
  return a;
}
```

Démo en Why3

Variant et invariant

- Pourquoi l'invariant est-il nécessaire pour prouver le variant dans cet exemple ?

- Comment aurait-on pu modifier le programme pour prouver le variant sans avoir besoin d'un invariant ?

- Dans les exercices suivants on reprend les exemples de variants vus dans les cours précédents et on ajoute les invariants nécéssaires pour les prouver.

Exercice : Enumération des nombres triangulaires

Donner un variant et un invariant permettant de prouver le variant

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

Démo en Why3

Exercice : PGCD version itérative

Donner un variant, avec des préconditions et un invariant permettant de prouver le variant

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

Démo en Why3

Rappel : PGCD récursif

```
int pgcd (int a, int b){
    //@requires a > 0;
    //@requires b > 0;
    //@variant a + b;
    if (a == b){ return a; }
    else if (a > b){ return pgcd(b, a-b); }
    else { return pgcd (a, b-a); }
}
```

Démo en Why3

Comparer la réponse à l'exercice précédent avec cette spécification.

Les invariants de boucle

Utilisation des variants pour prouves les post-conditions

Principe général

- La méthode de l'invariant ne permet pas seulement de prouver des variants, elle donne un moyen général de caractériser ce qu'il se passe lors d'une boucle.

- On peut utiliser un invariant pour aider à prouver les post-conditions des fonctions.

Usage des labels

- Attention : En Why3 un invariant est nécessairement une condition booléenne : ca ne peut pas être un entier comme dans l'exemple de l'introduction.

- Pour construire ces booléens, on utilise souvent des « labels » grâce au mot clé `//@label`

- Si on a définit un label `L`, on peut écrire `at(e,L)` pour faire référence à la valeur qu'avait l'expression `e` au point de programme marqué du label `L`.

Exercice : Spécification complète de l'addition naive

On reprend le code du début. Ajouter une postcondition qui spécifie le résultat que renvoie cette fonction, et un invariant permettant de prouver cette post-condition.

```
int additionBete (int a, int b){
  //@requires b>0;
  while (b != 0){
    //@variant b;
    //@invariant b>=0;
    a = a + 1;
    b = b - 1;
  }
  return a;
}
```

Exercice : Somme des éléments d'un tableau

Sujet sur Why3

On suppose une fonction logique de somme partielles d'un tableau :

1. Définir une fonction logique `somme(int a[])` qui renvoie la somme des éléments du tableau `a`.

2. Spécifier le programme donné, à l'aide de ces fonctions.

Exercice : nombres triangulaires

Sujet sur Why3

1. Définir une fonction logique `triangulaire(int n)` qui renvoie la valeur du $n^{\text{ème}}$ nombre triangulaire.

2. A l'aide de cette fonction spécifier complètement le programme donné.
   Attention : Why3 ne sera pas capable de prouver l'invariant de boucle, même si ce dernier est correct.

Exercice : plus petit nombre triangulaire plus grand qu'une borne

Sujet sur Why3

1. Définir une fonction logique `triangulaire(int n)` qui renvoie la valeur du $n^{\text{ème}}$ nombre triangulaire.

2. Spécifier le programme donné pour prouver que le résultat est un nombre triangulaire plus grand que `n`.

3. Affiner la spécification pour certifier que le résultat est le plus petit nombre triangulaire plus grand que `n`.

Les invariants de boucle

Preuve d'un algorithme de tri itératif

Objectif

- La dernière fois, on avait montré que la version récursive du tri par instertion renvoie toujours un tableau trié (sans prouver la fonction d'insertion).

- Aujourd'hui, on va faire la même chose pour la version itérative du tri par insertion.

Prédicats utilisés

- Définir un prédicat `trieJusque(int a[], int n)` qui dit que le tableau `a` est trié jusqu'à l'indice `n` exclus.

- Définir un prédicat `trie(int a[])` qui dit que le tableau `a` est trié.

Fonction d'insertion

Rappeler les spécifications de la fonction d'insertion, garantissant qu'elle trie le tableau

```
void insertion (int tab[], int n){
    // CODE DE LA FONCTION
}
```

Tri par insertion

Donner les spécifications (contrat, variant et invariant) pour la fonction de tri par insertion itérative suivante, garantissant que le tableau obtenu à la fin est trié.

```
void triInsertion(int tab[], int n){
    for(int i = 1; i <= n; i++){
        insertion(tab,i);
    }
}
```

Démo en Why3