

Reunions de rentree

Licence 2

Mardi 6 septembre de 14h à 15h en amphi POINCARÉ Bat A22

Université Bordeaux 1
Licence Sciences, Technologies, Santé
J1INAW01 : Informatique Théorique 1

Pierre Castéran

22 novembre 2011

└ Introduction

 └ Premier Cours

Premier Cours

Important !

- ▶ Ce cours est conforme à la nouvelle habilitation de la Licence,
- ▶ Cours, TDs, examens, seront **différents** de ceux des années précédentes,
- ▶ Transparents, sources des programmes, exercices, solutions seront sur le site www.labri.fr/perso/casteran/IT1/ et disponibles au service des polycopiés (à partir de la semaine prochaine).
- ▶ Contrôle continu : **devoir maison**

À propos du contenu de ce cours

- ▶ L'intitulé **Informatique Théorique 1** est hérité d'anciennes habilitations de la Licence,
- ▶ Le nouveau contenu considère cependant la théorie seulement comme un **outil**
- ▶ dont la connaissance est indispensable à certains types de métiers émergents, liés aux problèmes de sécurité logicielle.

À propos du contenu de ce cours

- ▶ L'intitulé **Informatique Théorique 1** est hérité d'anciennes habilitations de la Licence,
- ▶ Le nouveau contenu considère cependant la théorie seulement comme un **outil**
- ▶ dont la connaissance est indispensable à certains types de métiers émergents, liés aux problèmes de sécurité logicielle.
- ▶ Ces métiers présenteront un intérêt tant que le coût (humain, financier) de bugs ou failles de sécurité sera supérieur au coût de développement (**énergie, transports, cartes à puce, communication, etc.**)

Métiers concernés

- ▶ Côté « Clients » : expression précise des besoins logiciels

Métiers concernés

- ▶ Côté « Clients » : expression précise des besoins logiciels
- ▶ Développement :
 - ▶ Proposition de solutions (exemple : ClearSy)
 - ▶ Développement d'outils logiciels

Métiers concernés

- ▶ Côté « Clients » : expression précise des besoins logiciels
- ▶ Développement :
 - ▶ Proposition de solutions (exemple : ClearSy)
 - ▶ Développement d'outils logiciels
- ▶ Recherche et Innovation :
 - ▶ Théorie de la programmation, logique, probabilités,
 - ▶ Création et amélioration d'outils informatique

Après IT1 ?

- ▶ Une option « Preuves d'algorithmes » en L3,
- ▶ Master d'informatique, Algorithmes et Méthodes Formelles, Génie Logiciel,
- ▶ Master Européen en Vérification Logicielle (Bordeaux, Bruxelles, Copenhague, Munich)

Positionnement international

L'Europe, la France, le LaBRI sont bien placées dans la recherche et la diffusion de ces méthodes formelles.

Positionnement international

L'Europe, la France, le LaBRI sont bien placées dans la recherche et la diffusion de ces méthodes formelles. Exemple : Écoles d'Été (Chine : 2009, 2010, **2011**, Inria : 2010, 2011).



Suzhou, Chine, 13-21 Août 2011. Enseignants de Bordeaux, Inria (Sophia-Antipolis), Rennes, USTC, Microsoft Research, Yale.

Contenu de ce cours

- ▶ Phases du développement d'un composant logiciel

Contenu de ce cours

- ▶ Phases du développement d'un composant logiciel
- ▶ Risques possibles (exemples)

Contenu de ce cours

- ▶ Phases du développement d'un composant logiciel
- ▶ Risques possibles (exemples)
- ▶ Notion de spécification

Contenu de ce cours

- ▶ Phases du développement d'un composant logiciel
- ▶ Risques possibles (exemples)
- ▶ Notion de spécification
- ▶ Aide au développement de programmes corrects

Contenu de ce cours

- ▶ Phases du développement d'un composant logiciel
- ▶ Risques possibles (exemples)
- ▶ Notion de spécification
- ▶ Aide au développement de programmes corrects
- ▶ Confiance dans ces techniques

Contenu de ce cours

- ▶ Phases du développement d'un composant logiciel
- ▶ Risques possibles (exemples)
- ▶ Notion de spécification
- ▶ Aide au développement de programmes corrects
- ▶ Confiance dans ces techniques
- ▶ Rappels et compléments de logique

Phases de la vie d'un programme

I : Expression d'un besoin : Exemples

- ▶ Pilotage d'un métro automatique,
- ▶ Signalisation ferroviaire : éviter les collisions,
- ▶ Logiciel embarqué (avionique)
- ▶ Logiciel des cartes à puces,
- ▶ Sécurité des navigateurs,
- ▶ Conception de bibliothèques pour un langage, de plug-ins, etc.

Phases de la vie d'un programme

I : Expression d'un besoin : Exemples

- ▶ Pilotage d'un métro automatique,
- ▶ Signalisation ferroviaire : éviter les collisions,
- ▶ Logiciel embarqué (avionique)
- ▶ Logiciel des cartes à puces,
- ▶ Sécurité des navigateurs,
- ▶ Conception de bibliothèques pour un langage, de plug-ins, etc.

Risques

Les besoins peuvent être mal exprimés, incomplets ou entachés d'ambiguïté

Phases de la vie d'un programme

II : Traduction en langage formel (adapté aux outils de développement)

- ▶ C'est la phase la plus risquée (incompréhension entre deux corps de métier totalement différents)

Phases de la vie d'un programme

II : Traduction en langage formel (adapté aux outils de développement)

- ▶ C'est la phase la plus risquée (incompréhension entre deux corps de métier totalement différents)
- ▶ **Solution :** Disposer d'un langage commun entre les « clients » et les informaticiens.

Phases de la vie d'un programme

II : Traduction en langage formel (adapté aux outils de développement)

- ▶ C'est la phase la plus risquée (incompréhension entre deux corps de métier totalement différents)
- ▶ **Solution :** Disposer d'un langage commun entre les « clients » et les informaticiens.
- ▶ **Choix de la Méthode B :**
 - ▶ Langage des maths élémentaires : théorie simple des ensembles, arithmétique, logique du premier ordre,
 - ▶ Méthodologie : construction d'un modèle compréhensible et accepté par le client.

Phases de la vie d'un programme

III : Passage d'un modèle à un programme effectif

- ▶ Entre le modèle compréhensible par le client, et un programme de performances acceptables, on procède à une suite d'optimisations.
- ▶ Le risque d'introduction de bugs peut être réduit par des outils +- interactifs (exemples **Atelier B**, **Rodin**, **Frama-C**, etc.)

Phases de la vie d'un programme

IV : Phases finales

- ▶ Production de code compilable
- ▶ Compilateurs certifiés (**Compcert**),
- ▶ Vérification du matériel.

Notion de spécification : Un exemple

On considère un tableau de nombres entiers, par exemple

2	4	15	8	-7	6	5	8	7	-10
---	---	----	---	----	---	---	---	---	-----

Notion de spécification : Un exemple

On considère un tableau de nombres entiers, par exemple

2	4	15	8	-7	6	5	8	7	-10
---	---	----	---	----	---	---	---	---	-----

On veut écrire une fonction (en Java) permettant de calculer la position d'un nombre dans ce tableau.

Soyons plus précis !

En Java, il est de tradition d'indexer les cases d'un vecteur de 0 à $n - 1$, n étant la longueur du vecteur.

0	1	2	3	4	5	6	7	8	9
2	4	15	8	-7	6	5	8	7	-10

Par exemple, si la valeur recherchée est 15, la fonction doit retourner l'indice 2 (troisième case du vecteur).

Soyons plus précis !

En Java, il est de tradition d'indexer les cases d'un vecteur de 0 à $n - 1$, n étant la longueur du vecteur.

0	1	2	3	4	5	6	7	8	9
2	4	15	8	-7	6	5	8	7	-10

Par exemple, si la valeur recherchée est 15, la fonction doit retourner l'indice 2 (troisième case du vecteur). Une ignorance de cette convention peut provoquer des erreurs d'interprétation des résultats

Soyons plus précis ! (2)

0	1	2	3	4	5	6	7	8	9
2	4	15	8	-7	6	5	8	7	-10

Si la valeur recherchée est 8, que doit retourner la fonction ?

- ▶ 3 ?
- ▶ 7 ?
- ▶ 3 ou 7 (peu importe) ?
- ▶ {3, 7 } ?

Soyons plus précis ! (3)

0	1	2	3	4	5	6	7	8	9
2	4	15	8	-7	6	5	8	7	-10

Si la valeur recherchée est 88, que doit retourner la fonction ?

- ▶ 0 ?

Soyons plus précis ! (3)

0	1	2	3	4	5	6	7	8	9
2	4	15	8	-7	6	5	8	7	-10

Si la valeur recherchée est 88, que doit retourner la fonction ?

► 0? **NON!!!!**

Soyons plus précis ! (3)

0	1	2	3	4	5	6	7	8	9
2	4	15	8	-7	6	5	8	7	-10

Si la valeur recherchée est 88, que doit retourner la fonction ?

- ▶ 0? **NON!!!!**
- ▶ -1?

Soyons plus précis ! (3)

0	1	2	3	4	5	6	7	8	9
2	4	15	8	-7	6	5	8	7	-10

Si la valeur recherchée est 88, que doit retourner la fonction ?

- ▶ 0? **NON!!!!**
- ▶ -1? Pourquoi pas, à justifier!

Soyons plus précis ! (3)

0	1	2	3	4	5	6	7	8	9
2	4	15	8	-7	6	5	8	7	-10

Si la valeur recherchée est 88, que doit retourner la fonction ?

- ▶ 0? **NON!!!!**
- ▶ -1? Pourquoi pas, à justifier!
- ▶ \emptyset ?

Soyons plus précis ! (3)

0	1	2	3	4	5	6	7	8	9
2	4	15	8	-7	6	5	8	7	-10

Si la valeur recherchée est 88, que doit retourner la fonction ?

- ▶ 0? **NON!!!!**
- ▶ -1? Pourquoi pas, à justifier!
- ▶ \emptyset ? OK, mais si la fonction retourne toujours un ensemble d'indices
- ▶ Lever une exception?

Soyons plus précis ! (3)

0	1	2	3	4	5	6	7	8	9
2	4	15	8	-7	6	5	8	7	-10

Si la valeur recherchée est 88, que doit retourner la fonction ?

- ▶ 0? **NON!!!!**
- ▶ -1? Pourquoi pas, à justifier!
- ▶ \emptyset ? OK, mais si la fonction retourne toujours un ensemble d'indices
- ▶ Lever une exception? Pourquoi pas!

Spécifications formelles

On utilise le langage des mathématiques élémentaires, supposé connu des acteurs de la vie d'un logiciel.

- ▶ Ensembles, relations, fonctions,
- ▶ Arithmétique simple,
- ▶ Logique du premier ordre : \vee , \wedge , \Rightarrow , \neg , $=$, \forall , \exists , etc.

Formalisation de notre exemple

Notations (cf **B**) :

- ▶ $\mathbb{N} = \{0, 1, 2, 3, \dots\}$: nombres entiers naturels :
- ▶ $\mathbb{N}_p = \{i \in \mathbb{N} | p \leq i\}$
- ▶ \mathbb{Z} : nombres entiers (relatifs)
- ▶ $n..p = \{i \in \mathbb{Z} | n \leq i \leq p\}$
- ▶ **Note** : Si $p < n$, alors $n..p = \emptyset$.

Représentation mathématique d'un tableau

Exemple : Considérons un tableau a :

0	1	2	3	4	5	6	7	8	9
2	4	15	8	-7	6	5	8	7	-10

Représentation mathématique d'un tableau

Exemple : Considérons un tableau a :

0	1	2	3	4	5	6	7	8	9
2	4	15	8	-7	6	5	8	7	-10

Le tableau a peut être représenté par une fonction *totale*
 $a : 0..9 \rightarrow \mathbb{Z}$:

- $a(0) = 2, a(1) = 4, \dots, a(9) = -10$

Représentation mathématique d'un tableau

Exemple : Considérons un tableau a :

0	1	2	3	4	5	6	7	8	9
2	4	15	8	-7	6	5	8	7	-10

Le tableau a peut être représenté par une fonction *totale*
 $a : 0..9 \rightarrow \mathbb{Z}$:

- ▶ $a(0) = 2, a(1) = 4, \dots, a(9) = -10$
- ▶ On peut dire aussi que a est une *fonction partielle* de \mathbb{N} dans \mathbb{Z}
 - ▶ de domaine $\text{dom}(a) = 0..9$
 - ▶ de codomaine $\text{ran}(a) = \{-10, -7, 2, 4, 5, 6, 7, 8, 15\}$.

Représentation mathématique d'un tableau

Exemple : Considérons un tableau a :

0	1	2	3	4	5	6	7	8	9
2	4	15	8	-7	6	5	8	7	-10

Le tableau a peut être représenté par une fonction *totale*
 $a : 0..9 \rightarrow \mathbb{Z}$:

- ▶ $a(0) = 2, a(1) = 4, \dots, a(9) = -10$
- ▶ On peut dire aussi que a est une *fonction partielle* de \mathbb{N} dans \mathbb{Z}
 - ▶ de domaine $\text{dom}(a) = 0..9$
 - ▶ de codomaine $\text{ran}(a) = \{-10, -7, 2, 4, 5, 6, 7, 8, 15\}$.
- ▶ Notons qu'une expression comme $a(-1)$ ou $a(42)$ n'a aucun sens.

Notations utiles

Soit $f : E \rightarrow F$ une fonction (partielle)

- ▶ Si $X \subseteq E$, $f(X) = \{y \in F \mid \exists x \in X, f(x) = y\}$
- ▶ Si $Y \subseteq F$, $f^{-1}(Y) = \{x \in E \mid \exists y \in Y, f(x) = y\}$

Si E est un ensemble fini, $\text{card}(E)$ « cardinal de E » dénote son nombre d'éléments.

Exemples

- ▶ $a(0..4) = \{2, 4, 8, 15\}$
- ▶ $a^{-1}(\{8\}) = \{3, 7\}$
- ▶ $\text{card}(a^{-1}(\{8\})) = 2$

Spécification : Premier essai

Précondition

- ▶ Soient $n \in \mathbb{N}_1$,
- ▶ $a : 0..(n - 1) \rightarrow \mathbb{Z}$,
- ▶ $x \in \mathbb{Z}$.

Spécification : Premier essai

Précondition

- ▶ Soient $n \in \mathbb{N}_1$,
- ▶ $a : 0..(n - 1) \rightarrow \mathbb{Z}$,
- ▶ $x \in \mathbb{Z}$.

Postcondition

L'appel de fonction `search(x, a)` renvoie un résultat `\res` tel que :

- ▶ $\res \in 0..(n - 1) \quad \wedge \quad a(\res) = x$

Spécification : Premier essai

Précondition

- ▶ Soient $n \in \mathbb{N}_1$,
- ▶ $a : 0..(n - 1) \rightarrow \mathbb{Z}$,
- ▶ $x \in \mathbb{Z}$.

Postcondition

L'appel de fonction `search(x, a)` renvoie un résultat `\res` tel que :

- ▶ $\res \in 0..(n - 1) \quad \wedge \quad a(\res) = x$

Raté !

Cette spécification ne précise aucun comportement dans le cas où x n'apparaît pas dans le tableau.

Spécification : deuxième essai

Précondition

Identique à l'exemple précédent.

Spécification : deuxième essai

Précondition

Identique à l'exemple précédent.

Postcondition

- ▶ $\forall i \in 0..n - 1, a(i) = x \Rightarrow \text{\textcolor{red}{res}} = i$
- ▶ $(\forall i \in 0..n - 1, a(i) \neq x) \Rightarrow \text{\textcolor{red}{res}} = -1$

Spécification : deuxième essai

Précondition

Identique à l'exemple précédent.

Postcondition

- ▶ $\forall i \in 0..n - 1, a(i) = x \Rightarrow \text{\textcolor{red}{res}} = i$
- ▶ $(\forall i \in 0..n - 1, a(i) \neq x) \Rightarrow \text{\textcolor{red}{res}} = -1$

Encore raté !

- ▶ Considérer l'exemple :

0	1	2	3	4	5	6	7	8	9
2	4	15	8	-7	6	5	8	7	-10

Une spécification correcte

Précondition

- ▶ Soient $n \in \mathbb{N}_1$,
- ▶ $a : 0..(n - 1) \rightarrow \mathbb{Z}$,
- ▶ $x \in \mathbb{Z}$.

Une spécification correcte

Précondition

- ▶ Soient $n \in \mathbb{N}_1$,
- ▶ $a : 0..(n - 1) \rightarrow \mathbb{Z}$,
- ▶ $x \in \mathbb{Z}$.

Postcondition

L'appel de fonction `search(x, a)` renvoie un résultat `\res` tel que :

$$\text{\res} \in 0..(n - 1) \wedge a(\text{\res}) = x$$

∨

$$\text{\res} < 0 \wedge \forall i \in 0..n - 1, a(i) \neq x$$

Une réalisation en Java

```
static int search(int x, int a[])
    // returns a position of x in the array a (0 based)
    // returns -1 if x doesn't occur in a
{
    int i;
    for(i = 0; i < a.length ; i++)
        if (a[i] == x)
            return i;
    return -1;
}
// amélioration : utiliser des exceptions (plus tard)
```

Une autre réalisation

```
static int search(int x, int a[])
    // returns a position of x in the array a (0 based)
{
    int i;
    for(i = a.length-1; i >=0 ; i--)
        if (a[i] == x)
            break;
    return i;
}
```

Résumé

Dans les exemples précédents, une spécification de programme est composée de :

- ▶ Une *précondition*, composée d'une ou plusieurs propositions décrivant quelles propriétés doivent satisfaire les arguments, et garantissant l'exécution sans problème du programme,
- ▶ Une *postcondition* composée d'une ou plusieurs propositions qui doivent être vérifiées après exécution du programme.
- ▶ Dans le cas d'une fonction, le mot clef \res désigne le résultat retourné par la fonction.

Remarque importante

La bonne exécution du programme et la correction des résultats n'est garantie que si la précondition est vérifiée (toutes les propositions la composant sont vraies).

Exemple : racine carrée entière

Précondition

$$n \in \mathbb{N}$$

Postcondition

$$\text{\res} \in \mathbb{N} \wedge \text{\res}^2 \leq n < (\text{\res} + 1)^2$$

```
public static int isqrt(int n) {  
    int r = 0;  
    int sqr = 0;  
    while (sqr <= n) {  
        sqr += 2 * r + 1;  
        r++;  
    }  
    return r-1;  
}
```

Cette réalisation est correcte, même si `isqrt(-45)` renvoie `-1`.

Autre exemple ; division entière

Précondition

$$a, b \in \mathbb{N} \wedge b > 0$$

Postcondition

$$\text{\textbackslash res} \in \mathbb{N} \wedge b \times \text{\textbackslash res} \leq a < b \times (\text{\textbackslash res} + 1)$$

```
public static int div(int a, int b) {  
    int q = 0;  
    while (b <= a) {  
        q ++;  
        a -= b;  
    }  
    return q;  
}
```

Correct, même si l'appel `div(4,0)` provoque un bouclage et l'appel `div(-6,2)` renvoie le résultat -1 .

Comparaison de pré[post]-conditions

Définition

Soient P et Q deux propositions. On dit que Q est *plus faible* que P si on peut prouver l'implication $P \Rightarrow Q$. On dit aussi que P est *plus forte* que Q .

Comparaison de pré[post]-conditions

Définition

Soient P et Q deux propositions. On dit que Q est *plus faible* que P si on peut prouver l'implication $P \Rightarrow Q$. On dit aussi que P est *plus forte* que Q .

Exemple

$$Q : n \in \mathbb{N}_1 \wedge a : 0..n-1 \rightarrow \mathbb{Z} \wedge x \in \mathbb{Z}$$

$$\begin{aligned} P : \quad & n \in \mathbb{N}_1 \wedge a : 0..n-1 \rightarrow \mathbb{Z} \wedge \\ & x \in \mathbb{Z} \wedge \\ & \forall i \in 0..n-2, a(i) \leq a(i+1) \end{aligned}$$

Exemple

Reprenons l'exemple de spécification de la recherche d'un élément dans un tableau, et prenons une précondition plus forte :

Précondition : $n \in \mathbb{N}_1 \wedge a : 0..n - 1 \rightarrow \mathbb{Z} \wedge$
 $x \in \mathbb{Z} \wedge$
 $\forall i \in 0..n - 2, a(i) \leq a(i + 1)$

Postcondition : $\text{\color{red}\textbackslash res} \in 0..(n - 1) \wedge a(\text{\color{red}\textbackslash res}) = x$
 \vee
 $\text{\color{red}\textbackslash res} < 0 \wedge \forall i \in 0..n - 1, a(i) \neq x$

La fonction de recherche vue précédemment est conforme à cette nouvelle spécification.

On peut, pour cette spécification : « recherche d'un élément dans un tableau trié », proposer une réalisation plus efficace, par *recherche dichotomique*.

Par exemple, cherchons 15 dans ce tableau

0	1	2	3	4	5	6	7	8	9
-12	-4	-4	1	7	7	15	18	77	100

On peut, pour cette spécification : « recherche d'un élément dans un tableau trié », proposer une réalisation plus efficace, par *recherche dichotomique*.

Par exemple, cherchons 15 dans ce tableau

0	1	2	3	4	5	6	7	8	9
-12	-4	-4	1	7	7	15	18	77	100

0	1	2	3	4	5	6	7	8	9
-12	-4	-4	1	7	7	15	18	77	100

On peut, pour cette spécification : « recherche d'un élément dans un tableau trié », proposer une réalisation plus efficace, par *recherche dichotomique*.

Par exemple, cherchons 15 dans ce tableau

0	1	2	3	4	5	6	7	8	9
-12	-4	-4	1	7	7	15	18	77	100

0	1	2	3	4	5	6	7	8	9
-12	-4	-4	1	7	7	15	18	77	100

0	1	2	3	4	5	6	7	8	9
-12	-4	-4	1	7	7	15	18	77	100

└ Notion de spécification

```
static int search(int x, int a[]) {  
    int result = -1;  
    int lo = 0;  
    int hi = a.length - 1;  
    int m = hi/2;  
    while (x >= a[lo] & x <= a[hi])  
        if (a[m] == x) return m;  
        else if (a[m] > x) {  
            hi = m - 1;  
            m = (lo + m)/2; }  
        else {  
            lo = m + 1;  
            m = (m + hi)/2;  
        }  
    return result;  
}
```

└ Notion de spécification

En revanche, cette fonction ne réalise pas la spécification dont la précondition est plus faible (ne suppose pas que le tableau est trié).

Exercice : trouver un contre-exemple !

En résumé ...

Si l'on renforce la précondition et/ou l'on affaiblit la postcondition, on peut augmenter le nombre de réalisations possibles.

En résumé ...

Si l'on renforce la précondition et/ou l'on affaiblit la postcondition, on peut augmenter le nombre de réalisations possibles.

Autrement dit

Si une fonction/un programme est conforme à une spécification, il reste conforme si l'on renforce la précondition et/ou l'on affaiblit la postcondition.

Il est très fréquent qu'une instruction ou une suite d'instructions d'un programme modifie la valeur de certaines variables.

```
if ( y < x ) {  
    int z = x;  
    x = y;  
    y = z;  
}
```

Postcondition

$$x = \min(\text{\oldx}, \text{\oldy}) \wedge y = \max(\text{\oldx}, \text{\oldy})$$

Dans les conditions, on utilise les conventions du langage mathématique. le signe $=$ désigne l'égalité et non l'affectation

Autre exemple

```
{  
    int z = p;  
    p = n;  
    n = z;  
}
```

n = \old{p} \wedge p = \old{n}

Autre exemple

```
{  
    int z = p;  
    p = n;  
    n = z;  
}
```

n = \old{p} \wedge p = \old{n}

Modification d'une structure de donnée

Précondition:

$$n : \mathbb{N}_1 \wedge a : 0..n-1 \rightarrow \mathbb{Z} \wedge i, j \in 0..n-1$$

Modification d'une structure de donnée

Précondition:

```
n : N1 ∧ a : 0..n - 1 → Z ∧ i, j ∈ 0..n - 1
{
    int x = a[i];
    a[i] = a[j];
    a[j] = x;
}
```

Modification d'une structure de donnée

Précondition:

$$n : \mathbb{N}_1 \wedge a : 0..n-1 \rightarrow \mathbb{Z} \wedge i, j \in 0..n-1$$

```
{
```

```
    int x = a[i];
```

```
    a[i] = a[j];
```

```
    a[j] = x;
```

```
}
```

Postcondition:

$$\begin{aligned} a(i) &= \text{\textcolor{red}{old}}\,a(j) \wedge a(j) = \text{\textcolor{red}{old}}\,a(i) \wedge \\ \forall k \in (0..n-1) \setminus \{i, j\}, \quad a(k) &= \text{\textcolor{red}{old}}\,a(k) \end{aligned}$$

Deuxième Cours

Note

Certains de ces transparents ont été améliorés ou corrigés depuis leur impression lancée la semaine du 5/09.

- ▶ On reprendra l'impression ici.
- ▶ Noter les corrections dans
<http://www.labri.fr/perso/casteran/IT1>

Raisonnement sur des programmes impératifs

Voir Wikipedia

http://fr.wikipedia.org/wiki/Logique_de_Hoare

http://fr.wikipedia.org/wiki/Charles_Antony_Richard_Hoare

On va étudier la *logique de Hoare* afin de pouvoir prouver des propriétés sur le comportement de programmes simples.

Nous commencerons par des programmes **très** simples, puis rajouterons des traits réalistes (exceptions, fonctions, etc.)

Les triplets de Hoare

Définition

Un **triplet de Hoare** est une formule de la forme $\{P\} S \{Q\}$, où P et Q sont des assertions et S est une instruction. P et Q sont respectivement la **précondition** et la **postcondition** du triplet.

Les triplets de Hoare

Définition

Un **triplet de Hoare** est une formule de la forme $\{P\} S \{Q\}$, où P et Q sont des assertions et S est une instruction. P et Q sont respectivement la **précondition** et la **postcondition** du triplet.

Signification

1. Si P est vraie avant l'exécution de S

Les triplets de Hoare

Définition

Un **triplet de Hoare** est une formule de la forme $\{P\} S \{Q\}$, où P et Q sont des assertions et S est une instruction. P et Q sont respectivement la **précondition** et la **postcondition** du triplet.

Signification

1. Si P est vraie avant l'exécution de S
2. et si S s'exécute normalement (sans boucler ni lever une exception),

Les triplets de Hoare

Définition

Un **triplet de Hoare** est une formule de la forme $\{P\} S \{Q\}$, où P et Q sont des assertions et S est une instruction. P et Q sont respectivement la **précondition** et la **postcondition** du triplet.

Signification

1. Si P est vraie avant l'exécution de S
2. et si S s'exécute normalement (sans boucler ni lever une exception),
3. alors Q est vraie après l'exécution de S .

Les triplets de Hoare

Définition

Un **triplet de Hoare** est une formule de la forme $\{P\} S \{Q\}$, où P et Q sont des assertions et S est une instruction. P et Q sont respectivement la **précondition** et la **postcondition** du triplet.

Signification

1. Si P est vraie avant l'exécution de S
2. et si S s'exécute normalement (sans boucler ni lever une exception),
3. alors Q est vraie après l'exécution de S .

Avec 2) On parle de **correction partielle**.

Exemples

Les triplets ci-dessous sont *corrects* :

{ $x \leq y$ } $x = \max(x, y); \{x = y\}$

{true} $x = y; y = x; \{x = y\}$

{ $x \in \mathbb{Z} \wedge x < 42$ } $x++; \{x \leq 42\}$

{ $x \in \mathbb{Z} \wedge y \in \mathbb{Z}$ } $\text{while}(x \neq y) y++; \{x = y\}$
(correction partielle!).

Troisième Cours

À propos des triplets *

Ordre d'écriture

Dans un triplet $\{P\} S \{Q\}$,

Q désigne le but à atteindre, S un moyen d'atteindre ce but, et P des conditions suffisantes pour que S permette d'atteindre cet objectif.

À propos des triplets *

Ordre d'écriture

Dans un triplet $\{P\} S \{Q\}$,

Q désigne le but à atteindre, S un moyen d'atteindre ce but, et P des conditions suffisantes pour que S permette d'atteindre cet objectif.

L'ordre d'écriture d'un tel triplet est donc naturellement de la droite vers la gauche.

Exemple *

$\{a \in \mathbb{N} \wedge b \in \mathbb{N}_1\}$

```
int q = 0;  
int r = a;  
while (b <= r) {  
    q ++;  
    r -= b;  
}  
{b q ≤ a < b(q + 1)}
```

Exemple (Suite) *

$\{a \in \mathbb{N} \wedge b \in \mathbb{N}_1\}$

```
int q = 0;  
int r = a;  
while (b <= r) {  
    q ++;  
    r -= b;  
}  
{bq ≤ a < b(q + 1)}
```

Exemple (Fin) *

$\{a \in \mathbb{N} \wedge b \in \mathbb{N}_1\}$

```
int q = 0;  
int r = a;  
while (b <= r) {  
    q ++;  
    r -= b;  
}  
{b q ≤ a < b(q + 1)}
```

Précondition la plus faible

Certains outils fonctionnent de la façon suivante :

1. On écrit la postcondition Q ,

Précondition la plus faible

Certains outils fonctionnent de la façon suivante :

1. On écrit la postcondition Q ,
2. On écrit l'instruction S ,

Précondition la plus faible

Certains outils fonctionnent de la façon suivante :

1. On écrit la postcondition Q ,
2. On écrit l'instruction S ,
3. Le système calcule une précondition, $\text{wp}(S, Q)$, appelée **précondition la plus faible** associée à Q et à S .

Précondition la plus faible

Certains outils fonctionnent de la façon suivante :

1. On écrit la postcondition Q ,
2. On écrit l'instruction S ,
3. Le système calcule une précondition, $\text{wp}(S, Q)$, appelée **précondition la plus faible** associée à Q et à S .
4. si l'on peut prouver $P \Rightarrow \text{wp}(S, Q)$, (**autrement dit P est plus forte que $\text{wp}(S, Q)$**),

Précondition la plus faible

Certains outils fonctionnent de la façon suivante :

1. On écrit la postcondition Q ,
2. On écrit l'instruction S ,
3. Le système calcule une précondition, $\text{wp}(S, Q)$, appelée **précondition la plus faible** associée à Q et à S .
4. si l'on peut prouver $P \Rightarrow \text{wp}(S, Q)$, (**autrement dit P est plus forte que $\text{wp}(S, Q)$**),
5. alors le triplet $\{P\} S \{Q\}$ est correct.

Précondition la plus faible

Certains outils fonctionnent de la façon suivante :

1. On écrit la postcondition Q ,
2. On écrit l'instruction S ,
3. Le système calcule une précondition, $\text{wp}(S, Q)$, appelée **précondition la plus faible** associée à Q et à S .
4. si l'on peut prouver $P \Rightarrow \text{wp}(S, Q)$, (**autrement dit P est plus forte que $\text{wp}(S, Q)$**),
5. alors le triplet $\{P\} S \{Q\}$ est correct.

Voir Wikipedia “weakest precondition”.

*

On peut aussi **annoter** le code avec des assertions. Il faut prouver que ces assertions sont vraies chaque fois que l'exécution du programme passe par le point associé.

```
//@ a ∈ N ∧ b ∈ N1
int q = 0;
int r = a;
//@ r ∈ N ∧ a = bq + r
while (b <= r) {
    //@ r ∈ N ∧ a = bq + r ∧ b ≤ r
    q++;
    a -= b;
}
//@ bq ≤ a < b(q + 1)
```

Attention aux confusions !

Certains langages (*C, Java, ...*) proposent un *mécanisme d'assertions* :

```
assert (0 < i & i < a.length -1) ;  
if (a[i] < a[i+1] )  
{ ...  
}
```

Attention aux confusions !

Certains langages (*C, Java, ...*) proposent un *mécanisme d'assertions* :

```
assert (0 < i & i < a.length -1) ;  
if (a[i] < a[i+1] )  
{ ...  
}
```

Ce mécanisme est fondamentalement différent des annotations du transparent précédent.

Attention aux confusions ! (2)

- ▶ les assertions introduites par assert sont vérifiées **dynamiquement** (à chaque exécution),

Attention aux confusions ! (2)

- ▶ les assertions introduites par assert sont vérifiées **dynamiquement** (à chaque exécution),
- ▶ leur validité relève du « Jusqu'ici, tout va bien »,

Attention aux confusions ! (2)

- ▶ les assertions introduites par `assert` sont vérifiées **dynamiquement** (à chaque exécution),
- ▶ leur validité relève du « **Jusqu'ici, tout va bien** »,
- ▶ les annotations introduites par `//@` sont prouvées une fois pour toutes et pour toute exécution (quelle que soit la valeur des paramètres d'une fonction, par exemple).

Attention aux confusions ! (3)

- ▶ On peut appliquer assert seulement sur des expressions booléennes calculables en un temps raisonnable,
- ▶ Les annotations peuvent être *n'importe quelle proposition logique.*

//@ $\forall i \ j \in 0..n-1, \ i \neq j \implies a(i) \neq a(j)$

Attention aux confusions ! (3)

- ▶ On peut appliquer assert seulement sur des expressions booléennes calculables en un temps raisonnable,
- ▶ Les annotations peuvent être *n'importe quelle proposition logique*.

//@ $\forall i \ j \in 0..n-1, \ i \neq j \implies a(i) \neq a(j)$

- ▶ Le **calcul** d'une valeur booléenne est **automatique**
- ▶ La **preuve** d'une **proposition logique** demande souvent une intervention humaine. D'où le développement d'**assistants à la preuve** interactifs : Coq, Isabelle, HOL4, ACL2, PVS, Rodin, Atelier B, etc.

Remarques

Les deux fonctionnements suivants sont envisageables :

- ▶ Le programme à prouver existe déjà, on le décore d'assertions qu'on espère correctes et suffisantes,

Remarques

Les deux fonctionnements suivants sont envisageables :

- ▶ Le programme à prouver existe déjà, on le décore d'assertions qu'on espère correctes et suffisantes, ou
- ▶ On développe simultanément code exécutable et assertion (plus aisé).

Remarques

Les deux fonctionnements suivants sont envisageables :

- ▶ Le programme à prouver existe déjà, on le décore d'assertions qu'on espère correctes et suffisantes, ou
- ▶ On développe simultanément code exécutable et assertion (plus aisé).

L'obtention d'un code certifié peut être assez longue, et demander pas mal d'essais. L'assistance d'un logiciel spécialisé est extrêmement utile.

Un langage simple

- ▶ Types de valeurs : entiers, booléens
- ▶ arithmétique : $+$, $*$, $-$, \leq , $<$, $=$, \neq , etc.
- ▶ opérateurs booléens : and, or, not
- ▶ Instructions :
 - ▶ **skip** (instruction vide)
 - ▶ $x := e$
 - ▶ $S_1 ; S_2$
 - ▶ **if** B **then** S_1 **else** S_2 **endif**
 - ▶ **while** B **do** S **done**

Notes

On s'éloigne de la syntaxe de Java ou C pour être cohérent avec la plupart des documents. On laisse provisoirement de côté break, return, throw, les jeux dangereux avec les pointeurs, etc.

Exemple (en Java)

```
int p = n;
long y = x;
long a = 1;
while (p > 0)
    if (p % 2 == 0) {
        p /= 2;
        y = y * y;
    }
    else {
        p--;
        a *= y;
    }
}
```

L'instruction centrale, avec nos notations simplifiées

```
p := n ; y := x ; a := 1 ;
while p > 0
do
  if p % 2 = 0
    then
      p := p/2 ;
      y := y * y
    else
      p := p-1;
      a := a * y
    endif
done
```

$$\{n \in \mathbb{N} \wedge x \in \mathbb{Z}\}$$

```
p := n ; y := x ; a := 1 ;
```

```
while p > 0
```

```
do
```

```
if p % 2 = 0
```

```
then
```

```
    p := p/2 ;
```

```
    y := y * y
```

```
else
```

```
    p := p-1;
```

```
    a := a * y
```

```
endif
```

```
done
```

$$\{a = x^n\}$$

$\{n \in \mathbb{N} \wedge x \in \mathbb{Z}\}$

```
p := n ; y := x ; a := 1 ;
```

```
while p > 0
```

```
do
```

```
if p % 2 = 0
```

```
then
```

```
    p := p/2 ;
```

```
    y := y * y
```

```
else
```

```
    p := p-1;
```

```
    a := a * y
```

```
endif
```

```
done
```

 $\{a = x^n\}$

(On néglige pour le moment les problèmes de débordement)

Utilisation des triplets

- ▶ Nous allons présenter des règles de construction de triplets corrects,

Utilisation des triplets

- ▶ Nous allons présenter des règles de construction de triplets corrects,
- ▶ en explorant systématiquement toutes les formes d'instructions.

Utilisation des triplets

- ▶ Nous allons présenter des règles de construction de triplets corrects,
- ▶ en explorant systématiquement toutes les formes d'instructions.
- ▶ Ce sera l'occasion de revoir un peu les règles du raisonnement mathématique.

L'instruction vide

L'instruction vide **skip** ne change pas l'état de la mémoire. En Java ou en C, elle s'écrit **{}**.

Cette instruction est surtout utile dans des conditionnelles **if then else endif**.

L'instruction vide

L'instruction vide **skip** ne change pas l'état de la mémoire. En Java ou en C, elle s'écrit **{}**.

Cette instruction est surtout utile dans des conditionnelles **if then else endif**.

La règle ci-dessous exprime que si l'état de la mémoire vérifie une assertion **P**, alors **P** est vérifiée après toute exécution de **skip** :

$$\overline{\{P\} \text{ skip } \{P\}}$$

L'affectation $x := e$

Prenons un exemple :

{????} $x := y + 5 \{x > 10\}$

L'affectation $x := e$

Prenons un exemple :

{????} $x := y + 5 \{x > 10\}$

Intuitivement, la précondition devrait être $y > 5$

L'affectation $x := e$

Prenons un exemple :

$$\{ \text{????} \} \quad x := y + 5 \quad \{ x > 10 \}$$

Intuitivement, la précondition devrait être $y > 5$ autrement dit
 $y + 5 > 10$

L'affectation $x := e$

Prenons un exemple :

$$\{????\} x := y + 5 \{x > 10\}$$

Intuitivement, la précondition devrait être $y > 5$ autrement dit
 $y + 5 > 10$

Règle pour l'affectation

$$\overline{\{P[e/x]\}} x := e \overline{\{P\}}$$

$P[e/x]$ dénote l'assertion P dans laquelle toutes les occurrences libres de x ont été remplacées par e .

Exemples

{????} $y := x*x \{y \geq 81\}$

Exemples

```
{?????} y := x*x {y ≥ 81}  
{x2 ≥ 81} y := x*x {y ≥ 81}
```

Exemples

{?????} $y := x*x \{y \geq 81\}$

$\{x^2 \geq 81\} y := x*x \{y \geq 81\}$

{?????} $y := 36 \{\exists n \in \mathbb{Z}, n^2 = y\}$

Exemples

{?????} $y := x*x \{y \geq 81\}$

$\{x^2 \geq 81\} y := x*x \{y \geq 81\}$

{?????} $y := 36 \{\exists n \in \mathbb{Z}, n^2 = y\}$

$\{\exists n \in \mathbb{Z}, n^2 = 36\} y := 36 \{\exists n \in \mathbb{Z}, n^2 = y\}$

Exemples

{?????} $y := x*x \{y \geq 81\}$

$\{x^2 \geq 81\} y := x*x \{y \geq 81\}$

{?????} $y := 36 \{\exists n \in \mathbb{Z}, n^2 = y\}$

$\{\exists n \in \mathbb{Z}, n^2 = 36\} y := 36 \{\exists n \in \mathbb{Z}, n^2 = y\}$

{?????} $x := 2*x \{\exists n \in \mathbb{Z}, 2 \times n = x\}$

Exemples

{?????} $y := x*x \{y \geq 81\}$
 $\{x^2 \geq 81\} y := x*x \{y \geq 81\}$

{?????} $y := 36 \{\exists n \in \mathbb{Z}, n^2 = y\}$
 $\{\exists n \in \mathbb{Z}, n^2 = 36\} y := 36 \{\exists n \in \mathbb{Z}, n^2 = y\}$

{?????} $x := 2*x \{\exists n \in \mathbb{Z}, 2 \times n = x\}$
 $\{\exists n \in \mathbb{Z}, 2 \times n = 2 \times x\} x := 2*x \{\exists n \in \mathbb{Z}, 2 \times n = x\}$

Exemples

$\{????\} \quad y := x*x \quad \{y \geq 81\}$
 $\{x^2 \geq 81\} \quad y := x*x \quad \{y \geq 81\}$

$\{????\} \quad y := 36 \quad \{\exists n \in \mathbb{Z}, \quad n^2 = y\}$
 $\{\exists n \in \mathbb{Z}, \quad n^2 = 36\} \quad y := 36 \quad \{\exists n \in \mathbb{Z}, \quad n^2 = y\}$

$\{????\} \quad x := 2*x \quad \{\exists n \in \mathbb{Z}, \quad 2 \times n = x\}$
 $\{\exists n \in \mathbb{Z}, \quad 2 \times n = 2 \times x\} \quad x := 2*x \quad \{\exists n \in \mathbb{Z}, \quad 2 \times n = x\}$

Mais pas :

$\{\exists y \in \mathbb{Z}, \quad y > y + 1\} \quad x := y + 1 \quad \{\exists y \in \mathbb{Z}, \quad y > x\}$

Exemples

$\{????\} \quad y := x*x \quad \{y \geq 81\}$

$\{x^2 \geq 81\} \quad y := x*x \quad \{y \geq 81\}$

$\{????\} \quad y := 36 \quad \{\exists n \in \mathbb{Z}, \quad n^2 = y\}$

$\{\exists n \in \mathbb{Z}, \quad n^2 = 36\} \quad y := 36 \quad \{\exists n \in \mathbb{Z}, \quad n^2 = y\}$

$\{????\} \quad x := 2*x \quad \{\exists n \in \mathbb{Z}, \quad 2 \times n = x\}$

$\{\exists n \in \mathbb{Z}, \quad 2 \times n = 2 \times x\} \quad x := 2*x \quad \{\exists n \in \mathbb{Z}, \quad 2 \times n = x\}$

Mais pas :

$\{\exists y \in \mathbb{Z}, \quad y > y + 1\} \quad x := y + 1 \quad \{\exists y \in \mathbb{Z}, \quad y > x\}$

$\{\exists z \in \mathbb{Z}, \quad z > y + 1\} \quad x := y + 1 \quad \{\exists z \in \mathbb{Z}, \quad z > x\}$ OK

Règle de Composition

$$\frac{\text{????}}{\{P\} \text{ S ; T } \{Q\}}$$

Règle de Composition

On décompose l'exécution de $S;T$ en une exécution de S suivie d'une exécution de T .

Règle de Composition

$$\frac{\{P\} \text{ S } \{R\} \quad \{R\} \text{ T } \{Q\}}{\{P\} \text{ S ;T } \{Q\}}$$

Exemple

$$\frac{\{2y = 2^{x+1}\} \ y := 2*y \ \{ \quad \quad \quad \} \quad \{ \quad \quad \quad \} \ x := x+1 \ \{y = 2^x\}}{\{2y = 2^{x+1}\} \ y := 2*y ; \ x := x+1 \ \{y = 2^x\}}$$

Exemple

$$\frac{\{2y = 2^{x+1}\} \ y := 2*y \ \{y = 2^{x+1}\} \quad \{y = 2^{x+1}\} \ x := x+1 \ \{y = 2^x\}}{\{2y = 2^{x+1}\} \ y := 2*y \ ; \ x := x+1 \ \{y = 2^x\}}$$

Remarque

L'assertion du milieu (*R* dans la règle de composition) n'apparaît pas dans la conclusion. Il faut parfois trouver quelle est la bonne assertion. C'est une des causes de l'impossibilité d'automatiser toutes les preuves.

Règle de la conséquence

On reprend notre dernière inférence :

$$\frac{\dots}{\{2y = 2^{x+1}\} \ y := 2*y \ ; \ x := x+1 \ \{y = 2^x\}}$$

Règle de la conséquence

On reprend notre dernière inférence :

$$\frac{\dots}{\{2y = 2^{x+1}\} \ y := 2*y \ ; \ x := x+1 \ \{y = 2^x\}}$$

En fait on voulait prouver :

$$\frac{\dots}{\{y = 2^x\} \ y := 2*y \ ; \ x := x+1 \ \{y = 2^x\}}$$

Règle de la conséquence

On reprend notre dernière inférence :

$$\frac{\dots}{\{2y = 2^{x+1}\} \ y := 2*y \ ; \ x := x+1 \ \{y = 2^x\}}$$

En fait on voulait prouver :

$$\frac{\dots}{\{y = 2^x\} \ y := 2*y \ ; \ x := x+1 \ \{y = 2^x\}}$$

Intuitivement cela doit être possible, car l'égalité $y = 2^x$ implique $2y = 2^{x+1}$

On peut appliquer la **règle de la conséquence** :

$$\frac{P \Rightarrow P_1 \quad \{P_1\} \text{ S } \{Q_1\} \quad Q_1 \Rightarrow Q}{\{P\} \text{ S } \{Q\}}$$

En clair, on peut renforcer la précondition et affaiblir la postcondition dans un triplet correct.

On peut appliquer la **règle de la conséquence** :

$$\frac{P \Rightarrow P_1 \quad \{P_1\} \text{ S } \{Q_1\} \quad Q_1 \Rightarrow Q}{\{P\} \text{ S } \{Q\}}$$

En clair, on peut renforcer la précondition et affaiblir la postcondition dans un triplet correct.

On retrouvera les mêmes problèmes d'automatisation. Comment inventer les assertions P_1 et Q_1 à partir de P et Q ?

Exemple (renforcement de la précondition)

$$\frac{(arithmétique)}{y = 2^x \implies 2y = 2^{x+1} \quad \{2y = 2^{x+1}\} \ y := 2*y ; \ x := x+1 \quad \{y = 2^x\}} \quad \dots$$

$$\{y = 2^x\} \ y := 2*y; \ x := x+1 \quad \{y = 2^x\}$$

Exemple (renforcement de la précondition)

$$\frac{(arithmétique)}{y = 2^x \implies 2y = 2^{x+1} \quad \{2y = 2^{x+1}\} \ y := 2*y ; \ x := x+1 \ \{y = 2^x\}} \quad \dots$$
$$\{y = 2^x\} \ y := 2*y ; \ x := x+1 \ \{y = 2^x\}$$

Note

C'est dans les renforcements de précondition et les affaiblissements de postcondition qu'on utilisera les règles de la logique.

Autre exemple

On prouve (exercice) le triplet ci-dessous

$$\{y = b \wedge x = a\} \ z := y ; \ y := x ; \ x := z \ \{x = b \wedge y = a\}$$

Autre exemple

On prouve (exercice) le triplet ci-dessous

$$\{y = b \wedge x = a\} \ z := y ; \ y := x ; \ x := z \ \{x = b \wedge y = a\}$$

Par la règle de la conséquence, on obtient

$$\{x = a \wedge y = b\} \ z := y ; \ y := x ; \ x := z \ \{a \neq b \implies x \neq y\}$$

Règle de la Conditionnelle

$$\frac{\{P \wedge b\} \ S \ \{Q\} \quad \{P \wedge \neg b\} \ T \ \{Q\}}{\{P\} \text{ if } b \text{ then } S \text{ else } T \text{ endif } \ {Q}}$$

On n'exécute S que si b est vraie, et T que si b est fausse.

Règle de la Conditionnelle

$$\frac{\{P \wedge b\} \ S \ \{Q\} \quad \{P \wedge \neg b\} \ T \ \{Q\}}{\{P\} \text{ if } b \text{ then } S \text{ else } T \text{ endif } \ {Q}}$$

On n'exécute S que si b est vraie, et T que si b est fausse.

Exemple (x, y, z variables dans \mathbb{Z})

$\{\text{true}\} \text{ if } x < y \text{ then } z := y \text{ else } z := x \text{ endif } \{z = \max(x, y)\}$

Règle de la Conditionnelle

$$\frac{\{P \wedge b\} \ S \ \{Q\} \quad \{P \wedge \neg b\} \ T \ \{Q\}}{\{P\} \text{ if } b \text{ then } S \text{ else } T \text{ endif } \ \{Q\}}$$

On n'exécute S que si b est vraie, et T que si b est fausse.

Exemple (x, y, z variables dans \mathbb{Z})

$$\frac{\{x < y\} \ z := y \ \{z = \max(x, y)\} \quad \{\neg x < y\} \ z := x \ \{z = \max(x, y)\}}{\{\text{true}\} \text{ if } x < y \text{ then } z := y \text{ else } z := x \text{ endif } \ \{z = \max(x, y)\}}$$

Exercice

Soient x, y, z trois variables dans \mathbb{Z} , et l'instruction \mathcal{S} ci-dessous

```
if y < x
then
    z := x ;
    x := y ;
    y := z
else
    skip
endif
```

Montrer que le triplet $\{\text{true}\} \mathcal{S} \{x \leq y\}$ est correct.

Exercice

On considère des variables sur \mathbb{N} . Montrer que le triplet suivant est correct :

```
{  $x^n = a y^p$  }  
if p = 0  
then  
    skip  
else  
    if p % 2 = 0  
    then  
        y := y*y ; p := p/2  
    else  
        a := a*y ; p := p-1  
    endif  
endif  
{  $x^n = a y^p$  }
```

Traitement de la boucle **while**

Prenons un exemple :

```
{ n ∈ N }  
a := 1 ; p := n ; y := x ;  
while p > 0  
do  
    if p % 2 = 0  
        then  
            y := y*y ; p := p/2  
        else  
            a := a*y ; p := p-1  
        endif  
    done  
{ a = xn }
```

On rappelle que l'exécution d'une boucle **while b do S done** consiste en un nombre (inconnu à l'avance de répétitions de la séquence :

1. évaluation du booléen b ,
2. si b s'évalue en **true**, exécution du corps de boucle S ,

On sort de la boucle quand b s'évalue en **false** à l'étape 1.

On rappelle que l'exécution d'une boucle **while** b do S done consiste en un nombre (inconnu à l'avance de répétitions de la séquence :

1. évaluation du booléen b ,
2. si b s'évalue en **true**, exécution du corps de boucle S ,

On sort de la boucle quand b s'évalue en **false** à l'étape 1.

Notons que l'exécution d'une boucle peut très bien ne jamais se terminer : Exemple : **while** $x \geq 0$ **do** $x := x+1$ **done**.

Invariant de boucle

On appelle **Invariant de boucle** toute assertion correcte au début de chaque exécution du corps de boucle.

```
a := 1 ; p := n ; y := x ;
while p > 0
{ Invariant: p ∈ N ∧ xn = a yp }
do
  if p % 2 = 0
  then
    y := y*y ; p := p/2
  else
    a := a*y ; p := p-1
  endif
done
```

Règle pour la boucle **while**

$$\frac{\{P \wedge b\} \; S \; \{P\}}{\{P\} \text{ while } b \text{ do } S \text{ done } \{P \wedge \neg b\}}$$

Règle pour la boucle **while**

$$\frac{\{P \wedge b\} \; S \; \{P\}}{\{P\} \text{ while } b \text{ do } S \text{ done } \{P \wedge \neg b\}}$$

Commentaires :

On sort de la boucle (si on en sort ...) avec la propriété P (supposée vraie avant l'exécution de la boucle, et stable par l'execution du corps de boucle) et $\neg b$ (seul moyen de sortir de la boucle).

Suite de l'exemple de calcul de x^n

On choisit comme invariant de boucle l'assertion

$$\{ p \in \mathbb{N} \wedge x^n = a y^p \}.$$

Suite de l'exemple de calcul de x^n

On choisit comme invariant de boucle l'assertion

$$\{ p \in \mathbb{N} \wedge x^n = a y^p \}.$$

On prouve le triplet

```
{ p > 0    $\wedge$  p ∈  $\mathbb{N}$   $\wedge$   $x^n = a y^p$ }  
if p % 2 = 0  
then  
    y := y*y ; p := p/2  
else  
    a := a*y ; p := p-1  
endif  
{p ∈  $\mathbb{N}$   $\wedge$   $x^n = a y^p\}$ 
```

Par la règle du **while**, on obtient le triplet :

```
{ $p \in \mathbb{N} \wedge x^n = a y^p$ }  
while p > 0  
  {Invariant:  $p \in \mathbb{N} \wedge x^n = a y^p$ }  
  do  
    if p % 2 = 0  
      then  
        y := y*y ; p := p/2  
      else  
        a := a*y ; p := p-1  
      endif  
    done  
  {  $p \in \mathbb{N} \wedge x^n = a y^p \wedge \neg(p > 0)$  }
```

En affaiblissant la postcondition, on a :

```
{ $p \in \mathbb{N} \wedge x^n = ay^p$ }  
while p > 0  
  {Invariant:  $p \in \mathbb{N} \wedge x^n = ay^p$ }  
  do  
    if p % 2 = 0  
      then  
        y := y*y ; p := p/2  
      else  
        a := a*y ; p := p-1  
      endif  
    done  
  { $x^n = ay^0 = a$ }
```

En utilisant la règle de composition, on obtient

$$\{n \in \mathbb{N} \wedge x^n = 1 \times x^n\}$$

a := 1 ; p := n ; y := x ;

$$\{p \in \mathbb{N} \wedge x^n = ay^p\}$$

while p > 0

do ...

done

$$\{x^n = ay^0 = a\}$$

Par la règle de la conséquence, on obtient finalement :

```
{n ∈ N}  
a := 1 ; p := n ; y := x ;  
while p > 0  
do ...  
done  
{ xn = a }
```

Exercice

Prouver le triplet ci-dessous (en précisant bien quelles règles sont appliquées)

$\{x \in \mathbb{N} \wedge y \in \mathbb{N}\}$

```
a := 0 ;
k := y ;
while k > 0
do
  a := a + x ;
  k := k - 1
done
{a = xy}
```

Terminaison et Correction totale

Rappelons la règle pour la boucle **while** :

$$\frac{\{P \wedge b\} \; S \; \{P\}}{\{P\} \text{ while } b \text{ do } S \text{ done } \{P \wedge \neg b\}}$$

Sa conclusion énonce que si l'exécution de la boucle se termine normalement, alors la postcondition $P \wedge \neg b$ est garantie.

Terminaison et Correction totale

Rappelons la règle pour la boucle **while** :

$$\frac{\{P \wedge b\} \ S \ \{P\}}{\{P\} \text{ while } b \text{ do } S \text{ done } \{P \wedge \neg b\}}$$

Sa conclusion énonce que si l'exécution de la boucle se termine **normalement**, alors la postcondition $P \wedge \neg b$ est garantie.
Cette règle ne sert pas à garantir la terminaison de la boucle

Exemple

```
int result := -1; int lo := 0; int hi := a.length - 1;
int m := hi/2; boolean found := false;
while ¬ found ∧ lo <= hi do
    if a[m] == x
        then result := m ; found := true
    else if a[m] > x
        then
            hi := m - 1; m := (lo + m)/2
        else
            lo := m + 1; m := (m + hi)/2
    endif
done
```

Exemple

```
int result := -1; int lo := 0; int hi := a.length - 1;
int m := hi/2; boolean found := false;
while ¬ found ∧ lo <= hi do
    if a[m] == x
        then result := m ; found := true
    else if a[m] > x
        then
            hi := m - 1; m := (lo + m)/2
        else
            lo := m + 1; m := (m + hi)/2
    endif
done
```

Démo :/src : script demoBug.txt

Exemple

```
int result := -1; int lo := 0; int hi := a.length - 1;
int m := hi/2; boolean found := false;
while ¬ found ∧ lo <= hi do
    if a[m] == x
        then result := m ; found := true
    else if a[m] > x
        then
            hi := m - 1; m := (lo + m)/2
        else
            lo := m + 1; m := (m + hi)/2
    endif
done
```

Démo : .. /src : script demoBug.txt Boucle par exemple pour rechercher 71 dans 1, 2, 2, 4, 7, 11, 18, 22, 31.

└ Terminaison et Correction totale

0	1	2	3	4	5	6	7	8
1	2	2	4	7	11	18	22	31

└ Terminaison et Correction totale

0	1	2	3	4	5	6	7	8
1	2	2	4	7	11	18	22	31

0	1	2	3	4	5	6	7	8
1	2	2	4	7	11	18	22	31

└ Terminaison et Correction totale

0	1	2	3	4	5	6	7	8
1	2	2	4	7	11	18	22	31

0	1	2	3	4	5	6	7	8
1	2	2	4	7	11	18	22	31

0	1	2	3	4	5	6	7	8
1	2	2	4	7	11	18	22	31

└ Terminaison et Correction totale

0	1	2	3	4	5	6	7	8
1	2	2	4	7	11	18	22	31

0	1	2	3	4	5	6	7	8
1	2	2	4	7	11	18	22	31

0	1	2	3	4	5	6	7	8
1	2	2	4	7	11	18	22	31

0	1	2	3	4	5	6	7	8
1	2	2	4	7	11	18	22	31

└ Terminaison et Correction totale

0	1	2	3	4	5	6	7	8
1	2	2	4	7	11	18	22	31

0	1	2	3	4	5	6	7	8
1	2	2	4	7	11	18	22	31

0	1	2	3	4	5	6	7	8
1	2	2	4	7	11	18	22	31

0	1	2	3	4	5	6	7	8
1	2	2	4	7	11	18	22	31

0	1	2	3	4	5	6	7	8
1	2	2	4	7	11	18	22	31

$$m = 7 \quad \text{lo} = \text{hi} = 8$$

Analyse du problème

- ▶ Une ignorance de l'arithmétique élémentaire (division par 2), fait que la propriété $lo \leq m \leq hi$ n'est pas maintenue,
- ▶ La taille du sous-tableau à explorer : $hi - lo + 1$ peut ne pas décroître (ici elle reste à 1 durant le bouclage).

Analyse du problème

- ▶ Une ignorance de l'arithmétique élémentaire (division par 2), fait que la propriété $lo \leq m \leq hi$ n'est pas maintenue,
- ▶ La taille du sous-tableau à explorer : $hi - lo + 1$ peut ne pas décroître (ici elle reste à 1 durant le bouclage).
- ▶ **Solution :** forcer cette quantité (appelée **variant**) à décroître strictement à chaque tour de boucle ...

Analyse du problème

- ▶ Une ignorance de l'arithmétique élémentaire (division par 2), fait que la propriété $lo \leq m \leq hi$ n'est pas maintenue,
- ▶ La taille du sous-tableau à explorer : $hi - lo + 1$ peut ne pas décroître (ici elle reste à 1 durant le bouclage).
- ▶ **Solution :** forcer cette quantité (appelée **variant**) à décroître strictement à chaque tour de boucle ...
- ▶ ...et vérifier quelle est toujours positive ou nulle.

└ Terminaison et Correction totale

└ Quatrième Cours

Quatrième Cours

Voir le pb du DM.

Réponses à quelques questions posées en TD

A quoi ça sert ?

- ▶ À écrire moins de programmes faux,
- ▶ à écrire plus facilement des programmes corrects
- ▶ à corriger ou valider des programmes écrits par d'autres.

Réponses à quelques questions posées en TD

A quoi ça sert ?

- ▶ À écrire moins de programmes faux,
- ▶ à écrire plus facilement des programmes corrects
- ▶ à corriger ou valider des programmes écrits par d'autres.

Écrire un tas de triplets de Hoare est fastidieux

Oui, mais le “calcul de Hoare” est justement un calcul, comme le calcul d’une dérivée, l’arithmétique, le calcul matriciel. On apprend d’abord les règles, on les comprend. Dans une phase ultérieure, on saute les étapes *triviales*.

Réponses à quelques questions posées en TD (2)

Les outils professionnels de vérification fonctionnent t-ils comme ça ?

Le calcul de triplets est entièrement automatisé. Les triplets les plus faciles à prouver sont prouvés automatiquement. Une technique des plus utilisées est le calcul de **plus faible précondition** basé sur le calcul de Hoare.

Réponses à quelques questions posées en TD (2)

Les outils professionnels de vérification fonctionnent t-ils comme ça ?

Le calcul de triplets est entièrement automatisé. Les triplets les plus faciles à prouver sont prouvés automatiquement. Une technique des plus utilisées est le calcul de **plus faible précondition** basé sur le calcul de Hoare.

Pourquoi faut-il alors apprendre/comprendre la théorie ?

Parce que ces outils ne peuvent automatiquement :

- ▶ inventer les invariants de boucle
- ▶ appliquer dans les cas non triviaux la règle de conséquence
- ▶ prouver dans tous les cas la terminaison des programmes

Réponses à quelques questions posées en TD (3)

Quand peut-on dire qu'un résultat est "trivial" ?

Tout dépend de la situation :

- ▶ S'il suffit d'un simple calcul pour vérifier (avec ou sans machine)
- ▶ Si c'est l'application directe d'un résultat connu (qu'on peut citer)
- ▶ Au cours d'une discussion (ex. oral), si on peut répondre à une demande de justification.
- ▶ Si on est vraiment sûr(e) de soi.

Contre-exemple

"*Si $x \in \mathbb{Z}$, alors $2x \leq x^2$* "

Autre exemple :

{ $y < x$ }

$z := x ; x := y ; y := z$

{ $x < y$ }

Au lieu d'expliciter trois applications de la règle d'affectation et deux de la règle de composition, on peut écrire de façon concise :

{ $y < x$ }

$z := x ;$

{ $y < z$ }

$x := y ;$

{ $x < z$ }

$y := z$

{ $x < y$ }

Autre exemple :

{ $y < x$ }

$z := x ; x := y ; y := z$

{ $x < y$ }

Au lieu d'expliciter trois applications de la règle d'affectation et deux de la règle de composition, on peut écrire de façon concise :

{ $y < x$ }

$z := x ;$

{ $y < z$ }

$x := y ;$

{ $x < z$ }

$y := z$

{ $x < y$ }

Et enfin l'écrire directement dans sa première forme **sans se tromper.**

Où trouver de la documentation ?



Une sélection de tutoriaux (en anglais et en français) est accessible sur www.labri.fr/perso/casteran/IT1 (mise à jour en permanence).

Un peu de méthodologie

- ▶ En réalité, la preuve d'un programme se réduit à mettre des annotations correctes aux endroits **importants** d'un programme.
- ▶ Les logiciels de vérification s'appuient sur ces annotations et le calcul de Hoare pour essayer d'automatiser au maximum la vérification.

Un peu de méthodologie

- ▶ En réalité, la preuve d'un programme se réduit à mettre des annotations correctes aux endroits **importants** d'un programme.
- ▶ Les logiciels de vérification s'appuient sur ces annotations et le calcul de Hoare pour essayer d'automatiser au maximum la vérification.

Ce qu'il reste à préciser

- ▶ Quels sont ces « points importants » ?
- ▶ Comment écrire les **bonnes** annotations ?

Un peu de méthodologie

- ▶ En réalité, la preuve d'un programme se réduit à mettre des annotations correctes aux endroits **importants** d'un programme.
- ▶ Les logiciels de vérification s'appuient sur ces annotations et le calcul de Hoare pour essayer d'automatiser au maximum la vérification.

Ce qu'il reste à préciser

- ▶ Quels sont ces « points importants » ?
- ▶ Comment écrire les **bonnes** annotations ?

On va se baser sur une grande variété d'exemples (en Cours et en TD)

Le problème des invariants de boucle

```
{P}  
while b  
  {Invariant: Inv}  
  do S  
  done  
{Q}
```

Problème : trouver *Inv*

En appliquant les règles du **while** et de la conséquence, on peut être un peu plus précis :

```
{P}  
while b  
  {Invariant: Inv}  
  do   S  
  done  
{Q}
```

Il faut prouver les implications

- ▶ Initialisation : $P \implies \text{Inv}$
- ▶ Sortie de boucle : $\text{Inv} \wedge \neg b \implies Q$
- ▶ Maintien de l'invariant à chaque itération : $\{\text{Inv} \wedge b\} \text{ S } \{\text{Inv}\}$.

Exemple

On considère le code suivant (calcul de $\lfloor \log_2(a) \rfloor$)

```
n := 0 ;
p := 2 ;
while p <= a
do
    p := 2 * p ;
    n := n + 1
done
```

Il n'est pas interdit de commencer par des **tests** !

a	n	p
16	0	2

Il n'est pas interdit de commencer par des **tests** !

a	n	p
16	0	2
16	1	4

Il n'est pas interdit de commencer par des **tests** !

a	n	p
16	0	2
16	1	4
16	2	8

Il n'est pas interdit de commencer par des **tests** !

a	n	p
16	0	2
16	1	4
16	2	8
16	3	16

└ Un peu de méthodologie

Il n'est pas interdit de commencer par des **tests** !

a	n	p
16	0	2
16	1	4
16	2	8
16	3	16
16	4	32

On n'oublie pas le calcul d'un logarithme par défaut :

a	n	p
31	0	2

On n'oublie pas le calcul d'un logarithme par défaut :

a	n	p
31	0	2
31	1	4

On n'oublie pas le calcul d'un logarithme par défaut :

a	n	p
31	0	2
31	1	4
31	2	8

On n'oublie pas le calcul d'un logarithme par défaut :

a	n	p
31	0	2
31	1	4
31	2	8
31	3	16

On n'oublie pas le calcul d'un logarithme par défaut :

a	n	p
31	0	2
31	1	4
31	2	8
31	3	16
31	4	32

On n'oublie pas le calcul d'un logarithme par défaut :

a	n	p
31	0	2
31	1	4
31	2	8
31	3	16
31	4	32

On peut maintenant entamer une formalisation :

└ Un peu de méthodologie

On appelle **axiome** une hypothèse exprimant une propriété portant sur des paramètres **constants** d'un programme, et dont la validité supposée reste constante lors de toute exécution.

On appelle **axiome** une hypothèse exprimant une propriété portant sur des paramètres **constants** d'un programme, et dont la validité supposée reste constante lors de toute exécution.

À éviter :

Axiom $a \in \mathbb{N}_1$

```
while a > 0
do
  a := a-1;
done
```

On suppose $a \in \mathbb{N}_1$ (**axiome**)

```
{ true}
n := 0 ; p := 2 ;
{ n = 0  $\wedge$  p = 2 }
while p <= a
{Invariant: Inv }
do p := 2 * p ; n := n + 1
done
{  $2^n \leq a < 2^{n+1}$ }
```

On suppose $a \in \mathbb{N}_1$ (**axiome**)

```

{ true}
n := 0 ; p := 2 ;
{ n = 0  $\wedge$  p = 2 }
while p <= a
{Invariant: Inv }
do   p := 2 * p ; n := n + 1
done
{  $2^n \leq a < 2^{n+1}$ }

```

On doit avoir les propriétés suivantes :

$$n = 0 \wedge p = 2 \implies \text{Inv}$$

$$\text{Inv} \wedge a < p \implies 2^n \leq a < 2^{n+1}$$

et le triplet $\{\text{Inv} \wedge p \leq a\} \ p := 2 * p ; n := n + 1 \ \{\text{Inv}\}$.

Première idée

On propose la formule $p = 2^{n+1}$ comme invariant de boucle.
deux sur les trois preuves à effectuer sont faciles :

$$n = 0 \wedge p = 2 \implies p = 2^{n+1}$$

$$\{p = 2^{n+1} \wedge p \leq a\} \quad p := 2 * p ; n := n + 1 \quad \{p = 2^{n+1}\}.$$

Première idée

On propose la formule $p = 2^{n+1}$ comme invariant de boucle.
deux sur les trois preuves à effectuer sont faciles :

$$n = 0 \wedge p = 2 \implies p = 2^{n+1}$$

$$\{p = 2^{n+1} \wedge p \leq a\} \quad p := 2 * p ; n := n + 1 \quad \{p = 2^{n+1}\}.$$

Mais l'implication

$$p = 2^{n+1} \wedge a < p \implies 2^n \leq a < 2^{n+1}$$

est fausse. Prendre par exemple $a = 31$, $p = 1024$, $n = 9$.

On prend un invariant plus fort

La propriété $2^n \leq a$ semble confirmée par les tests précédents.

On prend un invariant plus fort

La propriété $2^n \leq a$ semble confirmée par les tests précédents.

On peut essayer de l'ajouter à l'invariant :

Axiome: $a \in \mathbb{N}_1$

```
{ n = 0  $\wedge$  p = 2 }
while p <= a
{Invariant: p =  $2^{n+1}$   $\wedge$   $2^n \leq a$ }
do p := 2 * p ; n := n + 1
done
{  $2^n \leq a < 2^{n+1}$ }
```

Les trois conditions à vérifier sont prouvables facilement.

- ▶ Initialisation :

$$n = 0 \wedge p = 2 \implies p = 2^{n+1} \wedge 2^n \leq a$$

- ▶ Sortie de boucle :

$$p = 2^{n+1} \wedge 2^n \leq a \wedge a < p \implies 2^n \leq a < 2^{n+1}$$

- ▶ Maintien de l'invariant :

$$\{p = 2^{n+1} \wedge 2^n \leq a \wedge p \leq a\}$$

$p := 2 * p ; n := n + 1$

$$\{p = 2^{n+1} \wedge 2^n \leq a\}$$

Exercice

Considérer cette autre version du calcul du logarithme entier par défaut (La division par 2 est une division entière.)

Axiome: $a \in \mathbb{N}_1$

```
b := a; n := 0;  
while b > 1  
do  b := b/2 ;  n := n + 1  
done  
{  $2^n \leq a < 2^{n+1}$ }
```

Est-elle correcte ? Si oui, donner un invariant de boucle suffisant pour prouver sa correction.

Un autre exemple

```
static final int FAILURE = -1;

static int search(int x, int a[])
    // returns a position of x in the array a (0 based)
    // returns -1 if x doesn't occur in a
{
    int i = 0;
    boolean found = false;
    while (i < a.length & !found)
        if (a[i] == x)
            found = true;
        else
            i++;
    return found ? i : FAILURE;
}
```

Axiomes : $n \in \mathbb{N}_1$

$a : 0..n-1 \rightarrow \mathbb{Z}$
 $x \in \mathbb{Z}$

```
i := 0; found := false;
while (i < a.length ∧ ¬ found) do
    if a[i] = x
        then found := true
    else i := i+ 1
    endif
done
{(found = true ∧ i ∈ 0..n-1 ∧ a(i) = x) ∨
 (found = false ∧ ∀ k ∈ 0..n-1, a(k) ≠ x ) }
```

Problème principal à résoudre

```
{i = 0  $\wedge$  found = false}
while (i < n  $\wedge$   $\neg$  found)
{Invariant: ???}
do
    if a[i] = x
        then found := true
        else i := i+ 1
        endif
    done
{(found = true  $\wedge$  i  $\in$  0..n-1  $\wedge$  a(i) = x)  $\vee$ 
 (found = false  $\wedge$   $\forall$  k  $\in$  0..n-1, a(k)  $\neq$  x ) }
```

Quelques conseils pour trouver un invariant

Procéder en plusieurs étapes

Un invariant de boucle est une formule logique (une **proposition**) qui doit satisfaire les trois propriétés suivantes :

- ▶ Initialisation : vraie à la première entrée dans la boucle
- ▶ Sortie de boucle : doit impliquer la postcondition associée à la boucle,
- ▶ maintien de l'invariant lorsque l'on procède à une itération (c.-à-d. lorsque l'expression booléenne associée à la boucle est vraie).

Quelques conseils pour trouver un invariant

Procéder en plusieurs étapes

Un invariant de boucle est une formule logique (une **proposition**) qui doit satisfaire les trois propriétés suivantes :

- ▶ Initialisation : vraie à la première entrée dans la boucle
- ▶ Sortie de boucle : doit impliquer la postcondition associée à la boucle,
- ▶ maintien de l'invariant lorsque l'on procède à une itération (c.-à-d. lorsque l'expression booléenne associée à la boucle est vraie).

Un invariant se présente souvent sous la forme d'une conjonction $\{\text{Inv}_1 \wedge \text{Inv}_2 \wedge \dots \wedge \text{Inv}_n\}$. On peut construire ces composants un par un.

Quelques conseils pour trouver un invariant (2)

Lorsque le programme peut comporter des opérations « à risque » (erreurs à l'exécution), exprimer que ces situations d'erreurs ne se produisent pas.

Quelques conseils pour trouver un invariant (2)

Lorsque le programme peut comporter des opérations « à risque » (erreurs à l'exécution), exprimer que ces situations d'erreurs ne se produisent pas.

```
{i = 0 ∧ found = false}
while (i < n ∧ ¬ found)
{Invariant: i ∈ 0..n - 1 }
do
  if a[i] = x
    then found := true
  else i := i + 1
  endif
done
```

└ Un peu de méthodologie

On remarque qu'on peut avoir $i = n$ dans ce programme.

On remarque qu'on peut avoir $i = n$ dans ce programme.

On répare ...

```
{i = 0 ∧ found = false}
while (i < n ∧ ¬ found)
{Invariant: i ∈ 0..n }
do
    {i ∈ 0..n - 1 }
    if a[i] = x
    then found := true
    else i := i+ 1
    endif
done
{i ∈ 0..n ∧ (i = n ∨ found = true)}
```

L'invariant est trop faible :

La proposition $\{i \in 0..n \wedge (i = n \vee \text{found} = \text{true})\}$

ne permet pas d'inférer la postcondition du programme :

$$\begin{aligned} &\{(\text{found} = \text{true} \wedge i \in 0..n - 1 \wedge a(i) = x) \vee \\ &(\text{found} = \text{false} \wedge \forall k \in 0..n - 1, a(k) \neq x)\} \end{aligned}$$

Quelques conseils pour trouver un invariant (3)

Exprimer le rôle des variables.

Quelques conseils pour trouver un invariant (3)

Exprimer le rôle des variables.

- ▶ Rôle de i : Les cases du tableau qui précèdent i ne contiennent pas x .
 $\forall k \in 0..n - 1, a(k) = x \implies i \leq k$

Quelques conseils pour trouver un invariant (3)

Exprimer le rôle des variables.

- ▶ Rôle de i : Les cases du tableau qui précèdent i ne contiennent pas x .
 $\forall k \in 0..n - 1, a(k) = x \implies i \leq k$
- ▶ Rôle de $found$: Si la variable $found$ a pour valeur **true**, alors la case courante contient x .
 $found = \text{true} \implies i < n \wedge a(i) = x$

Quelques conseils pour trouver un invariant (3)

Exprimer le rôle des variables.

- ▶ Rôle de i : Les cases du tableau qui précèdent i ne contiennent pas x .
 $\forall k \in 0..n - 1, a(k) = x \implies i \leq k$
- ▶ Rôle de $found$: Si la variable $found$ a pour valeur **true**, alors la case courante contient x .
 $found = \text{true} \implies i < n \wedge a(i) = x$
- ▶ **Attention**, l'invariant suivant serait faux :
 $found = \text{true} \iff (i < n \wedge a(i) = x)$

On propose donc comme invariant la conjonction des trois propositions précédentes :

$$i \in 0..n$$

$$(\forall k \in 0..n - 1, a(k) = x \implies i \leq k)$$

$$\text{found} = \text{true} \implies i < n \wedge a(i) = x$$

Utilité de l'invariant (sortie de boucle)

Il faut montrer :

$$i \in 0..n$$

$$(\forall k \in 0..n-1, a(k) = x \implies i \leq k)$$

$$\text{found} = \text{true} \implies i < n \wedge a(i) = x$$

$$\neg(i < n \wedge \neg\text{found})$$

$$(\text{found} = \text{true} \wedge i \in 0..n-1 \wedge a(i) = x) \vee$$

$$(\text{found} = \text{false} \wedge \forall k \in 0..n-1, a(k) \neq x)$$

On procède à quelques simplifications :

$$i \in 0..n$$

$$(\forall k \in 0..n-1, a(k) = x \implies i \leq k)$$

$$\text{found} = \text{true} \implies i < n \wedge a(i) = x$$

$$i = n \vee \text{found}$$

$$(\text{found} = \text{true} \wedge i \in 0..n-1 \wedge a(i) = x) \vee$$

$$(\text{found} = \text{false} \wedge \forall k \in 0..n-1, a(k) \neq x)$$

Premier cas : $i = n$

On en déduit **found=false**,

Premier cas : $i = n$

On en déduit $\text{found} = \text{false}$,

$$i \in 0..n$$

$$(\forall k \in 0..n-1, a(k) = x \implies i \leq k)$$

$$\text{found} = \text{true} \implies i < n \wedge a(i) = x$$

$$i = n$$

$$\text{found} = \text{false}$$

$$(\text{found} = \text{false} \wedge \forall k \in 0..n-1, a(k) \neq x)$$

Second cas : $\text{found} = \text{true}$.

$$i \in 0..n$$

$$\text{found} = \text{true} \implies i < n \wedge a(i) = x$$

$$\text{found} = \text{true}$$

$$(\text{found} = \text{true} \wedge i \in 0..n - 1 \wedge a(i) = x) \vee$$

$$(\text{found} = \text{false} \wedge \forall k \in 0..n - 1, a(k) \neq x)$$

CQFD

Initialisation

$$\frac{i = 0 \wedge \text{found} = \text{false}}{i \in 0..n \wedge (\forall k \in 0..n - 1, a(k) = x \implies i \leq k) \wedge (\text{found} = \text{true} \implies i < n \wedge a(i) = x)}$$

Initialisation

$$\frac{i = 0 \wedge \text{found} = \text{false}}{i \in 0..n \wedge (\forall k \in 0..n - 1, a(k) = x \implies 0 \leq k) \wedge \text{found} = \text{true} \implies i < n \wedge a(i) = x}$$

Maintien de l'invariant

$\{i \in 0..n\}$

$\{\text{forall } k \in 0..n-1, \ a(k) = x \implies i \leq k\}$

$\{\text{found} = \text{true} \implies i < n \wedge a(i) = x\}$

$\{i < n \wedge \neg \text{found}\}$

if $a[i] = x$

then $\text{found} := \text{true}$

else $i := i + 1$

endif

$\{i \in 0..n\}$

$\{\text{forall } k \in 0..n-1, \ a(k) = x \implies i \leq k\}$

$\{\text{found} = \text{true} \implies i < n \wedge a(i) = x\}$

On simplifie un peu ...

{ $i \in 0..n$ }

{ $\forall k \in 0..n-1, a(k) = x \implies i \leq k$ }

{ $found = \text{true} \implies i < n \wedge a(i) = x$ }

{ $i < n \wedge \neg found$ }

if $a[i] = x$

then $found := \text{true}$

else $i := i + 1$

endif

{ $i \in 0..n$ }

{ $\forall k \in 0..n-1, a(k) = x \implies i \leq k$ }

{ $found = \text{true} \implies i < n \wedge a(i) = x$ }

Branche **then**

```
{forall k ∈ 0..n-1, a(k) = x ⇒ i ≤ k}
{i < n ∧ ¬ found}
{a(i) = x}
found := true
{∀ k ∈ 0..n-1, a(k) = x ⇒ i ≤ k}
{found = true ⇒ i < n ∧ a(i) = x}
```

Branche **else**

{forall $k \in 0..n-1$, $a(k) = x \implies i \leq k\}$

{ $i < n \wedge \neg found$ }

{ $a(i) \neq x$ }

$i := i+1$

{ $\forall k \in 0..n-1$, $a(k) = x \implies i \leq k\}$

{ $found = \text{true} \implies i < n \wedge a(i) = x\}$

Finir en TD.

Le code est donc correct.

└ Terminaison

 └ Cinquième cours

Cinquième cours : terminaison

Correction partielle, correction totale

Comme on a vu en TD, le calcul de Hoare vu jusqu'à présent ne traite que de la correction partielle.

Correction partielle, correction totale

Comme on a vu en TD, le calcul de Hoare vu jusqu'à présent ne traite que de la correction partielle.

$\{n, p \in \mathbb{Z}\}$

while $n \neq p$ **do**

$n := n+1$

done

$\{n = p\}$

Correction partielle, correction totale

Comme on a vu en TD, le calcul de Hoare vu jusqu'à présent ne traite que de la correction partielle.

$\{n, p \in \mathbb{Z}\}$

while $n \neq p$ **do**

$n := n+1$

done

$\{n = p\}$

$\{n \in \mathbb{N}_1\}$

while $0 < n$ **do**

$n := n+1$

done

$\{2 = 3\}$

Correction totale

On reprend l'exemple précédent avec une légère modification de sa précondition :

$\{n, p \in \mathbb{Z} \wedge n \leq p\}$

while $n \neq p$

$\{n \leq p\}$

do

$n := n+1$

done

$\{n = p\}$

Correction totale

On reprend l'exemple précédent avec une légère modification de sa précondition :

$$\{n, p \in \mathbb{Z} \wedge n \leq p\}$$

while $n \neq p$

$$\{n \leq p\}$$

do

$n := n+1$

done

$$\{n = p\}$$

On constate que la quantité $p - n$ décroît strictement à chaque itération.

Correction totale

On reprend l'exemple précédent avec une légère modification de sa précondition :

```
{n, p ∈ ℤ ∧ n ≤ p}
```

```
while n ≠ p
```

```
{n ≤ p }
```

```
do
```

```
    n := n+1
```

```
done
```

```
{n = p}
```

On constate que la quantité $p - n$ décroît strictement à chaque itération.

Par l'invariant $n \leq p$, cette quantité reste dans \mathbb{N} .

Correction totale

On reprend l'exemple précédent avec une légère modification de sa précondition :

```
{n, p ∈ ℤ ∧ n ≤ p}
```

```
while n ≠ p
```

```
{n ≤ p }
```

```
do
```

```
    n := n+1
```

```
done
```

```
{n = p}
```

On constate que la quantité $p - n$ décroît strictement à chaque itération.

Par l'invariant $n \leq p$, cette quantité reste dans \mathbb{N} . Si le programme bouclait, on obtient une suite infinie d'entiers naturels strictement décroissante, ce qui est impossible (on l'admet provisoirement).

Règles de Hoare pour la correction totale

Définition

Un **triplet de Hoare pour la correction totale** est une formule de la forme $\langle P \rangle S \langle Q \rangle$, où P et Q sont des assertions et S est une instruction. P et Q sont respectivement la **précondition** et la **postcondition** du triplet.

Règles de Hoare pour la correction totale

Définition

Un **triplet de Hoare pour la correction totale** est une formule de la forme $\langle P \rangle S \langle Q \rangle$, où P et Q sont des assertions et S est une instruction. P et Q sont respectivement la **précondition** et la **postcondition** du triplet.

Signification

1. Si P est vraie avant l'exécution de S ,

Règles de Hoare pour la correction totale

Définition

Un **triplet de Hoare pour la correction totale** est une formule de la forme $\langle P \rangle S \langle Q \rangle$, où P et Q sont des assertions et S est une instruction. P et Q sont respectivement la **précondition** et la **postcondition** du triplet.

Signification

1. Si P est vraie avant l'exécution de S ,
2. alors S s'exécute normalement (sans boucler ni lever une exception),

Règles de Hoare pour la correction totale

Définition

Un **triplet de Hoare pour la correction totale** est une formule de la forme $\langle P \rangle S \langle Q \rangle$, où P et Q sont des assertions et S est une instruction. P et Q sont respectivement la **précondition** et la **postcondition** du triplet.

Signification

1. Si P est vraie avant l'exécution de S ,
2. alors S s'exécute normalement (sans boucler ni lever une exception),
3. et Q est vraie après l'exécution de C .

Règle de la boucle **while** avec correction totale

$$\frac{\langle P \wedge b \wedge \text{variant} = z \rangle \ S \ \langle P \wedge \text{variant} < z \wedge \text{variant} \in \mathbb{N} \rangle}{\langle P \rangle \text{ while } b \text{ do } S \text{ done } \langle P \wedge \neg b \rangle}$$

- ▶ *variant* est une expression, appelée le **variant** de la boucle,
- ▶ *z* est une variable non utilisée ailleurs dans le programme (la renommer dans le cas contraire).
- ▶ Les triplets prouvant la correction totale sont écrits avec des chevrons au lieu d'accolades.

Règles

Les règles pour la correction totale sont les mêmes que pour la correction partielle (en changeant les $\{P\} S \{Q\}$ en $\langle P \rangle S \langle Q \rangle$), sauf la règle du **while**, où l'on prend la règle avec variant.

Remarque

Si S ne contient aucune boucle, alors la correction de $\{P\} S \{Q\}$ implique celle de $\langle P \rangle S \langle Q \rangle$.

Concrètement, on prouve la correction partielle comme précédemment, puis on prouve la terminaison des boucles.

Reprendons l'exemple du calcul du logarithme

On suppose $a \in \mathbb{N}_1$ (**axiome**)

```
{ true}
n := 0 ; p := 2 ;
while p <= a
{Invariant:  $p = 2^{n+1} \wedge 2^n \leq a$  }
{Variant: ??? }
do p := 2 * p ; n := n + 1
done
{  $2^n \leq a < 2^{n+1}$  }
```

└ Terminaison

└ Cinquième cours

Le variant doit être une expression dont la valeur est un entier naturel et qui décroît à chaque tour de boucle.

└ Terminaison

└ Cinquième cours

Le variant doit être une expression dont la valeur est un entier naturel et qui décroît à chaque tour de boucle.

```
{ true}  
n := 0 ; p := 2 ;  
while p <= a  
{Invariant:  $p = 2^{n+1} \wedge 2^n \leq a$  }  
{Variant:  $a - 2^n$  }  
do p := 2 * p ; n := n + 1  
done  
{  $2^n \leq a < 2^{n+1}$  }
```

└ Terminaison

└ Cinquième cours

Il suffit de montrer le triplet suivant :

$$\langle p = 2^{n+1} \wedge 2^n \leq a \wedge p \leq a \wedge a - 2^n = z \\ p := 2 * p ; \\ n := n + 1 \\ a - 2^n < z \wedge a - 2^n \in \mathbb{N} \rangle$$

└ Terminaison

└ Cinquième cours

Or on a trivialement :

$$\langle a - 2^{n+1} < z \wedge a - 2^{n+1} \in \mathbb{N} \rangle$$

p := 2 * p ;

n := n + 1

$$\langle a - 2^n < z \wedge a - 2^n \in \mathbb{N} \rangle$$

Et l'implication :

$$p = 2^{n+1} \wedge 2^n \leq a \wedge p \leq a \wedge a - 2^n = z$$

\implies

$$a - 2^{n+1} < z \wedge a - 2^{n+1} \in \mathbb{N}$$

Exercice

Prouver la correction totale du programme suivant de calcul du plus grand diviseur de a et b (entiers naturels strictement positifs).

```
x := a ; y := b ;
while x ≠ y
do
    if x < y
    then
        y := y - x
    else
        x := x - y
    endif
done
```

Ensembles et Relations Bien Fondés

On veut justifier la règle de correction totale du **while** :

$$\frac{\langle P \wedge b \wedge \text{variant} = z \rangle \; S \; \langle P \wedge \text{variant} < z \wedge \text{variant} \in \mathbb{N} \rangle}{\langle P \rangle \; \text{while } b \text{ do } S \text{ done } \langle P \wedge \neg b \rangle}$$

Ensembles et Relations Bien Fondés

On veut justifier la règle de correction totale du **while** :

$$\frac{\langle P \wedge b \wedge \text{variant} = z \rangle \ S \ \langle P \wedge \text{variant} < z \wedge \text{variant} \in \mathbb{N} \rangle}{\langle P \rangle \text{ while } b \text{ do } S \text{ done } \langle P \wedge \neg b \rangle}$$

Théorème : l'ordre strict $<$ sur \mathbb{N} est **bien fondé** :

Il n'existe aucune suite infinie strictement décroissante

$x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} .

Ensembles et Relations Bien Fondés

On veut justifier la règle de correction totale du **while** :

$$\frac{\langle P \wedge b \wedge \text{variant} = z \rangle \ S \ \langle P \wedge \text{variant} < z \wedge \text{variant} \in \mathbb{N} \rangle}{\langle P \rangle \ \text{while } b \text{ do } S \text{ done } \langle P \wedge \neg b \rangle}$$

Théorème : l'ordre strict $<$ sur \mathbb{N} est **bien fondé** :

Il n'existe aucune suite infinie strictement décroissante

$x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} .

Ce théorème s'applique à la suite des valeurs successives du variant.

Preuve de la bonne fondation de $<$ sur \mathbb{N}

L'ensemble \mathbb{N} peut être défini par les 5 **axiomes de Peano**.

Selon Wikipédia

1. L'élément appelé zéro et noté 0, est un entier naturel.
2. Tout entier naturel n a un unique successeur, noté $s(n)$ ou S_n .
3. Aucun entier naturel n'a 0 pour successeur.
4. Deux entiers naturels ayant même successeur sont égaux.
5. Si un ensemble d'entiers naturels contient 0 et contient le successeur de chacun de ses éléments, alors cet ensemble est égal à \mathbb{N} .

Le cinquième axiome fonde le **principe de récurrence**.

Démonstration par récurrence simple

Soit P un prédictat défini sur \mathbb{N} : Pour prouver la proposition $\forall n \in \mathbb{N}, P(n)$, il suffit de :

- ▶ Prouver $P(0)$,
- ▶ Prouver que pour tout $n \in \mathbb{N}$, $P(n) \implies P(n + 1)$ est vraie.

On peut essayer de prouver que \mathbb{N} , muni de la relation $<$, est bien fondé :

- ▶ Preuve par récurrence ; On considère la propriété $P(n)$:
« Il n'existe aucune suite infinie strictement décroissante $n = x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} »
- ▶ $P(0)$ trivial.

On peut essayer de prouver que \mathbb{N} , muni de la relation $<$, est bien fondé :

- ▶ Preuve par récurrence ; On considère la propriété $P(n)$:
« Il n'existe aucune suite infinie strictement décroissante $n = x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} »
- ▶ $P(0)$ trivial.
- ▶ Soit $n \in \mathbb{N}$, tel que $P(n)$. Montrons $P(n + 1)$.
 - ▶ Supposons qu'il existe une suite infinie $n + 1 = x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N}

On peut essayer de prouver que \mathbb{N} , muni de la relation $<$, est bien fondé :

- ▶ Preuve par récurrence ; On considère la propriété $P(n)$:
« Il n'existe aucune suite infinie strictement décroissante $n = x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} »
- ▶ $P(0)$ trivial.
- ▶ Soit $n \in \mathbb{N}$, tel que $P(n)$. Montrons $P(n + 1)$.
 - ▶ Supposons qu'il existe une suite infinie $n + 1 = x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N}
 - ▶ la sous-suite $x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} est strictement décroissante et infinie.

On peut essayer de prouver que \mathbb{N} , muni de la relation $<$, est bien fondé :

- ▶ Preuve par récurrence ; On considère la propriété $P(n)$:
« Il n'existe aucune suite infinie strictement décroissante $n = x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} »
- ▶ $P(0)$ trivial.
- ▶ Soit $n \in \mathbb{N}$, tel que $P(n)$. Montrons $P(n + 1)$.
 - ▶ Supposons qu'il existe une suite infinie $n + 1 = x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N}
 - ▶ la sous-suite $x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} est strictement décroissante et infinie. **impasse...**

Analyse du blocage

Le prédicat P précédent est mal choisi pour utiliser une récurrence simple.

Analyse du blocage

Le prédicat P précédent est mal choisi pour utiliser une récurrence simple.

Deux solutions :

1. Choisir un autre prédicat,

Analyse du blocage

Le prédicat P précédent est mal choisi pour utiliser une récurrence simple.

Deux solutions :

1. Choisir un autre prédicat, ou
2. Utiliser un principe de récurrence plus adapté.

Analyse du blocage

Le prédicat P précédent est mal choisi pour utiliser une récurrence simple.

Deux solutions :

1. Choisir un autre prédicat, ou
2. Utiliser un principe de récurrence plus adapté.

En fait, la seconde solution est de la même nature que la première.

Au lieu de considérer une solution *ad hoc*, on tente d'obtenir des solutions génériques.

Preuve correcte de la bonne fondation de $<$ sur \mathbb{N} (solution *ad hoc*)

- ▶ Preuve par récurrence simple ; On considère la propriété $Q(n)$:

Il n'existe aucune suite infinie strictement décroissante $x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} avec $x_1 \leq n$.

Preuve correcte de la bonne fondation de $<$ sur \mathbb{N} (solution *ad hoc*)

- ▶ Preuve par récurrence simple ; On considère la propriété $Q(n)$:
Il n'existe aucune suite infinie strictement décroissante $x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} avec $x_1 \leq n$.
- ▶ $Q(0)$ trivial.
- ▶ Soit $n \in \mathbb{N}$, tel que $Q(n)$ est vraie. Montrons $Q(n + 1)$.
 - ▶ Supposons qu'il existe une suite infinie $x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} avec $x_1 \leq n + 1$.

Preuve correcte de la bonne fondation de $<$ sur \mathbb{N} (solution *ad hoc*)

- ▶ Preuve par récurrence simple ; On considère la propriété $Q(n)$:
Il n'existe aucune suite infinie strictement décroissante $x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} avec $x_1 \leq n$.
- ▶ $Q(0)$ trivial.
- ▶ Soit $n \in \mathbb{N}$, tel que $Q(n)$ est vraie. Montrons $Q(n + 1)$.
 - ▶ Supposons qu'il existe une suite une suite infinie $x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} avec $x_1 \leq n + 1$.
 - ▶ la sous-suite $x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} est strictement décroissante et infinie, et on a $x_2 \leq n$.

Preuve correcte de la bonne fondation de $<$ sur \mathbb{N} (solution *ad hoc*)

- ▶ Preuve par récurrence simple ; On considère la propriété $Q(n)$:
Il n'existe aucune suite infinie strictement décroissante $x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} avec $x_1 \leq n$.
- ▶ $Q(0)$ trivial.
- ▶ Soit $n \in \mathbb{N}$, tel que $Q(n)$ est vraie. Montrons $Q(n + 1)$.
 - ▶ Supposons qu'il existe une suite infinie $x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} avec $x_1 \leq n + 1$.
 - ▶ la sous-suite $x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} est strictement décroissante et infinie, et on a $x_2 \leq n$.
 - ▶ ce qui est impossible par l'hypothèse de récurrence.

CQFD.

Solution générique

Principe de récurrence complète

Soit P un prédicat défini sur \mathbb{N} : Pour prouver la proposition $\forall n \in \mathbb{N}, P(n)$, il suffit de :

- ▶ Prouver $P(0)$,
- ▶ Prouver que pour tout $n \in \mathbb{N}$, si $P(n)$ est vraie pour tout $p \leq n$, alors $P(n + 1)$ est vraie.

En fait, le principe de récurrence complète **se dérive** du principe de récurrence simple, en considérant le prédictat

$$Q(n) = \forall p \in \mathbb{N}, p \leq n \implies P(p)$$

En fait, le principe de récurrence complète **se dérive** du principe de récurrence simple, en considérant le prédictat

$$Q(n) = \forall p \in \mathbb{N}, p \leq n \implies P(p)$$

Démonstration à faire en TD.

Preuve correcte de la bonne fondation de $<$ sur \mathbb{N} (par récurrence complète)

- ▶ On considère la propriété $P(n)$:

Il n'existe aucune suite infinie strictement décroissante $n = x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} .

Preuve correcte de la bonne fondation de $<$ sur \mathbb{N} (par récurrence complète)

- ▶ On considère la propriété $P(n)$:

Il n'existe aucune suite infinie strictement décroissante $n = x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} .

- ▶ $P(0)$ trivial.
- ▶ Soit $n \in \mathbb{N}$, tel que $P(p)$ est vraie pour tout $p \leq n$. Montrons $P(n + 1)$.
 - ▶ Supposons qu'il existe une suite infinie $n + 1 = x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N}

Preuve correcte de la bonne fondation de $<$ sur \mathbb{N} (par récurrence complète)

- ▶ On considère la propriété $P(n)$:

Il n'existe aucune suite infinie strictement décroissante $n = x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} .

- ▶ $P(0)$ trivial.
- ▶ Soit $n \in \mathbb{N}$, tel que $P(p)$ est vraie pour tout $p \leq n$. Montrons $P(n + 1)$.
 - ▶ Supposons qu'il existe une suite infinie $n + 1 = x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N}
 - ▶ la sous-suite $x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} est strictement décroissante et infinie, et on a $x_2 \leq n$.

Preuve correcte de la bonne fondation de $<$ sur \mathbb{N} (par récurrence complète)

- ▶ On considère la propriété $P(n)$:

Il n'existe aucune suite infinie strictement décroissante $n = x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} .

- ▶ $P(0)$ trivial.
- ▶ Soit $n \in \mathbb{N}$, tel que $P(p)$ est vraie pour tout $p \leq n$. Montrons $P(n + 1)$.
 - ▶ Supposons qu'il existe une suite infinie $n + 1 = x_1 > x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N}
 - ▶ la sous-suite $x_2 > \dots > x_n > x_{n+1} > \dots$ dans \mathbb{N} est strictement décroissante et infinie, et on a $x_2 \leq n$.
 - ▶ ce qui est impossible par l'hypothèse de récurrence.

CQFD.

Un autre exemple

On veut montrer que le programme de recherche dans un tableau vu précédemment possède la propriété de correction totale. Il suffit donc de trouver un variant pour la boucle.

```
while (i < n ∧ ¬ found)
{Invariant:
 i ∈ 0..n ∧
(forall k ∈ 0..n - 1, a(k) = x ⇒ i ≤ k) ∧
found = true ⇒ i < n ∧ a(i) = x}
do
  if a[i] = x
    then found := true
    else i := i + 1
  endif
done
```

On peut proposer $n - i + 1 - b2n(\text{found})$ avec

$$b2n(\text{true}) = 1$$

$$b2n(\text{false}) = 0$$

On vérifie bien que, *sous les hypothèses formées par l'invariant et le test de la boucle while*, le variant reste dans \mathbb{N} et décroît strictement

On peut être plus subtil

```
while (i < n ∧ ¬ found)
{Invariant:
 i ∈ 0..n ∧
(forall k ∈ 0..n - 1, a(k) = x ⇒ i ≤ k) ∧
found = true ⇒ i < n ∧ a(i) = x}
{Variant: n - i}
do
    if a[i] = x
    then found := true
    else i := i + 1
    endif
done
```

À chaque itération, le variant reste dans \mathbb{N} et décroît strictement ou le booléen $i < n \wedge \neg \text{found}$ devient faux, et la boucle se termine.

Problèmes posés par l'application de la règle de correction totale du **while**

- ▶ L'invariant et le variant sont parfois difficiles à trouver.
- ▶ On peut faciliter la recherche du variant si l'on prend un domaine plus complexe que \mathbb{N}

Exemple (src/Countdown.java)

```
{z ∈ N}  
x := z ; y := z;  
while x > 0 | y > 0 do  
    // ...  
    if y > 0  
        then y := y -1  
    else  
        y := z ; x := x- 1  
    endif  
done
```

Exemple (src/Countdown.java)

```
{z ∈ N}  
x := z ; y := z;  
while x > 0 | y > 0 do  
    // ...  
    if y > 0  
        then y := y -1  
    else  
        y := z ; x := x- 1  
    endif  
done
```

Variant ?

Exemple (src/Countdown.java)

```
{z ∈ N}  
x := z ; y := z;  
while x > 0 | y > 0 do  
    // ...  
    if y > 0  
        then y := y -1  
    else  
        y := z ; x := x- 1  
    endif  
done
```

Variant ? $x + y$?

Exemple (src/Countdown.java)

```
{z ∈ N}  
x := z ; y := z;  
while x > 0 | y > 0 do  
    // ...  
    if y > 0  
        then y := y -1  
    else  
        y := z ; x := x- 1  
    endif  
done
```

Variant ? $x + y$? Non : (exemple, avec $z = 9$, passage de $x = 8, y = 0$ à $x = 7, y = 9$)

Exemple (src/Countdown.java)

```
{z ∈ N}  
x := z ; y := z;  
while x > 0 | y > 0 do  
    // ...  
    if y > 0  
        then y := y -1  
    else  
        y := z ; x := x- 1  
    endif  
done
```

Variant ? $x + y$? Non : (exemple, avec $z = 9$, passage de $x = 8, y = 0$ à $x = 7, y = 9$) $(z + 1)x + y$?

Exemple (src/Countdown.java)

```
{z ∈ N}  
x := z ; y := z;  
while x > 0 | y > 0 do  
    // ...  
    if y > 0  
        then y := y -1  
    else  
        y := z ; x := x- 1  
    endif  
done
```

Variant ? $x + y$? Non : (exemple, avec $z = 9$, passage de $x = 8, y = 0$ à $x = 7, y = 9$) ($z + 1)x + y$? OK, mais hors de portée de certains automatismes.

On généralise la règle du **while**

Soit $(E, <)$ un ensemble bien fondé :

$$\frac{\langle P \wedge b \wedge \text{variant} = z \rangle \ S \ \langle P \wedge \text{variant} < z \wedge \text{variant} \in E \rangle}{\langle P \rangle \ \mathbf{while} \ b \ \mathbf{do} \ S \ \mathbf{done} \ \langle P \wedge \neg b \rangle}$$

Propriétés des ensembles bien fondés

- ▶ Exemples, contre-exemples
- ▶ Propriétés importantes
- ▶ Exemples empruntés à la programmation

rappels

- Soit E un ensemble. Une **relation** sur E est un sous-ensemble du produit cartésien $E \times E$.

rappels

- ▶ Soit E un ensemble. Une **relation** sur E est un sous-ensemble du produit cartésien $E \times E$.
- ▶ Notations usuelles : R , S , $x R y$ au lieu de $(x, y) \in R$, etc.
- ▶ Une relation R est **irréflexive** si l'on a $\forall x \in E, \neg x R x$.

rappels

- ▶ Soit E un ensemble. Une **relation** sur E est un sous-ensemble du produit cartésien $E \times E$.
- ▶ Notations usuelles : R , S , $x R y$ au lieu de $(x, y) \in R$, etc.
- ▶ Une relation R est **irréflexive** si l'on a $\forall x \in E, \neg x R x$.
- ▶ Une relation R est **transitive** si l'on a
$$\forall x y z \in E, x R y \wedge y R z \implies x R z.$$

rappels

- ▶ Soit E un ensemble. Une **relation** sur E est un sous-ensemble du produit cartésien $E \times E$.
- ▶ Notations usuelles : R , S , $x R y$ au lieu de $(x, y) \in R$, etc.
- ▶ Une relation R est **irréflexive** si l'on a $\forall x \in E, \neg x R x$.
- ▶ Une relation R est **transitive** si l'on a
$$\forall x y z \in E, x R y \wedge y R z \implies x R z.$$
- ▶ Une relation à la fois irréflexive et transitive est appelée **ordre strict**.

rappels

- ▶ Soit E un ensemble. Une **relation** sur E est un sous-ensemble du produit cartésien $E \times E$.
- ▶ Notations usuelles : R , S , $x R y$ au lieu de $(x, y) \in R$, etc.
- ▶ Une relation R est **irréflexive** si l'on a $\forall x \in E, \neg x R x$.
- ▶ Une relation R est **transitive** si l'on a
$$\forall x y z \in E, x R y \wedge y R z \implies x R z.$$
- ▶ Une relation à la fois irréflexive et transitive est appelée **ordre strict**.
- ▶ Notations usuelles pour un ordre strict : $<$, \subset , \prec , \sqsubset , $<_A$, etc.

Exemples, contre-exemples

- ▶ On a déjà démontré que l'ordre strict $<$ sur \mathbb{N} est bien fondé.

Exemples, contre-exemples

- ▶ On a déjà démontré que l'ordre strict $<$ sur \mathbb{N} est bien fondé.
- ▶ En revanche, l'ordre strict $<$ sur \mathbb{Z} n'est pas bien fondé.

Exemples, contre-exemples

- ▶ On a déjà démontré que l'ordre strict $<$ sur \mathbb{N} est bien fondé.
- ▶ En revanche, l'ordre strict $<$ sur \mathbb{Z} n'est pas bien fondé.
Considérer la suite $0 > -1 > -2 > -3 > \dots$.
- ▶ L'ensemble des nombres rationnels positifs n'est pas bien fondé pour $<$.

Exemples, contre-exemples

- ▶ On a déjà démontré que l'ordre strict $<$ sur \mathbb{N} est bien fondé.
- ▶ En revanche, l'ordre strict $<$ sur \mathbb{Z} n'est pas bien fondé.
Considérer la suite $0 > -1 > -2 > -3 > \dots$
- ▶ L'ensemble des nombres rationnels positifs n'est pas bien fondé pour $<$.
Considérer la suite
 $1 > 1/2 > 1/4 > 1/8 > \dots > 1/2^n > 1/2^{n+1} > \dots$

Exemples, contre-exemples

- ▶ On a déjà démontré que l'ordre strict $<$ sur \mathbb{N} est bien fondé.
- ▶ En revanche, l'ordre strict $<$ sur \mathbb{Z} n'est pas bien fondé.
Considérer la suite $0 > -1 > -2 > -3 > \dots$
- ▶ L'ensemble des nombres rationnels positifs n'est pas bien fondé pour $<$.
Considérer la suite
 $1 > 1/2 > 1/4 > 1/8 > \dots > 1/2^n > 1/2^{n+1} > \dots$
- ▶ L'ensemble des chaînes de caractères n'est pas bien fondé pour l'ordre alphabétique.

Exemples, contre-exemples

- ▶ On a déjà démontré que l'ordre strict $<$ sur \mathbb{N} est bien fondé.
- ▶ En revanche, l'ordre strict $<$ sur \mathbb{Z} n'est pas bien fondé.
Considérer la suite $0 > -1 > -2 > -3 > \dots$
- ▶ L'ensemble des nombres rationnels positifs n'est pas bien fondé pour $<$.
Considérer la suite
 $1 > 1/2 > 1/4 > 1/8 > \dots > 1/2^n > 1/2^{n+1} > \dots$
- ▶ L'ensemble des chaînes de caractères n'est pas bien fondé pour l'ordre alphabétique.
Considérer la suite "oups" $>$ "ouupps" $>$ "ouuppps" $>$...

Revenons à notre programme de Compte à rebours.

```
{z ∈ N}  
x := z ; y := z;  
while x > 0 | y > 0 do  
    // ...  
    if y > 0  
        then y := y -1  
        else  
            y := z ; x := x- 1  
        endif  
    done
```

Revenons à notre programme de Compte à rebours.

```
{z ∈ N}  
x := z ; y := z;  
while x > 0 | y > 0 do  
    // ...  
    if y > 0  
        then y := y -1  
    else  
        y := z ; x := x- 1  
    endif  
done
```

On va se créer un outil pour prouver la terminaison de la boucle.

Ordre lexicographique

Définition

Soit $<_A$ [resp. $<_B$] un ordre strict sur A [resp. B]. On définit le **produit lexicographique** de $<_A$ et $<_B$ comme la relation sur $A \times B$ définie par : par :

$$\forall x, y, z, y <_B z \implies (x, y) <_{\text{lex}} (x, z)$$

$$\forall x, y, z, t, x <_A z \implies (x, y) <_{\text{lex}} (z, t)$$

Ordre lexicographique

Définition

Soit $<_A$ [resp. $<_B$] un ordre strict sur A [resp. B]. On définit le **produit lexicographique** de $<_A$ et $<_B$ comme la relation sur $A \times B$ définie par : par :

$$\forall x, y, z, y <_B z \implies (x, y) <_{\text{lex}} (x, z)$$

$$\forall x, y, z, t, x <_A z \implies (x, y) <_{\text{lex}} (z, t)$$

Exemple

On considère le produit lexicographique de $(\mathbb{N}, <)$ par lui-même.

On a par exemple

$$(1, 2) <_{\text{lex}} (1, 4) <_{\text{lex}} (1, 10^{978}) <_{\text{lex}} \dots <_{\text{lex}} (2, 0) <_{\text{lex}} (3, 0) \dots$$

Propriétés de l'ordre lexicographique

Théorème

Si $<_A$ et $<_B$ sont des ordres stricts, alors leur produit lexicographique est un ordre strict.

Preuve

Propriétés de l'ordre lexicographique

Théorème

Si $<_A$ et $<_B$ sont des ordres stricts, alors leur produit lexicographique est un ordre strict.

Preuve

Exercice.

Théorème

Si $<_A$ et $<_B$ sont bien fondés, alors leur produit lexicographique est bien fondé.

Preuve (par l'absurde)

1. On suppose que le produit lexicographique de $(A, <_A)$ par $(B, <_B)$ n'est pas bien fondé.

Théorème

Si $<_A$ et $<_B$ sont bien fondés, alors leur produit lexicographique est bien fondé.

Preuve (par l'absurde)

1. On suppose que le produit lexicographique de $(A, <_A)$ par $(B, <_B)$ n'est pas bien fondé.
2. Il existe donc une suite infinie
$$(a_1, b_1) >_{\text{lex}} (a_2, b_2) >_{\text{lex}} \dots (a_i, b_i) >_{\text{lex}} (a_{i+1}, b_{i+1}) \dots$$

Théorème

Si $<_A$ et $<_B$ sont bien fondés, alors leur produit lexicographique est bien fondé.

Preuve (par l'absurde)

1. On suppose que le produit lexicographique de $(A, <_A)$ par $(B, <_B)$ n'est pas bien fondé.
2. Il existe donc une suite infinie
$$(a_1, b_1) >_{\text{lex}} (a_2, b_2) >_{\text{lex}} \dots (a_i, b_i) >_{\text{lex}} (a_{i+1}, b_{i+1}) \dots$$
3. Deux cas sont à considérer :
 - ▶ La suite des a_i est stationnaire à partir d'un rank k (c.-à-d.
 $\forall l \in k.. \infty, a_l = a_k$,
 - ▶ Pour tout k , il existe l tel que $l > k$ et $a_l <_A a_k$.

Suite de la preuve

1. ▶ Dans le premier cas, la suite b_k, b_{k+1}, \dots est infinie et strictement décroissante, ce qui est impossible, car $(B, <_B)$ est bien fondé.

Suite de la preuve

1.
 - ▶ Dans le premier cas, la suite b_k, b_{k+1}, \dots est infinie et strictement décroissante, ce qui est impossible, car $(B, <_B)$ est bien fondé.
 - ▶ Dans le second cas, pour tout $k \in \mathbb{N}$, il existe un indice $l > k$ tel que $a_l <_A a_k$.

Suite de la preuve

1.
 - ▶ Dans le premier cas, la suite b_k, b_{k+1}, \dots est infinie et strictement décroissante, ce qui est impossible, car $(B, <_B)$ est bien fondé.
 - ▶ Dans le second cas, pour tout $k \in \mathbb{N}$, il existe un indice $l > k$ tel que $a_l <_A a_k$.
De même, il existe $m > l$ tel que $a_m <_A a_l$, etc.

Suite de la preuve

1.
 - ▶ Dans le premier cas, la suite b_k, b_{k+1}, \dots est infinie et strictement décroissante, ce qui est impossible, car $(B, <_B)$ est bien fondé.
 - ▶ Dans le second cas, pour tout $k \in \mathbb{N}$, il existe un indice $l > k$ tel que $a_l <_A a_k$.
De même, il existe $m > l$ tel que $a_m <_A a_l$, etc.
En itérant ce processus, on construit une suite infinie strictement décroissante dans A , ce qui est impossible !
2. Fin de la preuve par l'absurde. Le produit lexicographique de $(A, <_A)$ par $(B, <_B)$

Remarques sur le théorème précédent

- ▶ Il est abstrait,

Remarques sur le théorème précédent

- ▶ Il est abstrait,
- ▶ donc réutilisable dans beaucoup de contextes différents

Remarques sur le théorème précédent

- ▶ Il est abstrait,
- ▶ donc réutilisable dans beaucoup de contextes différents
- ▶ L'effort consacré à sa preuve peut être rentable.

Exemple d'application

```
{z ∈ N}  
x := z ; y := z;  
while x > 0 | y > 0 do  
    // ...  
    if y > 0  
        then y := y -1  
        else  
            y := z ; x := x- 1  
        endif  
    done
```

On propose comme variant le couple (x, y) .

```
{z ∈ N}  
x := z ; y := z;  
while x > 0 | y > 0 do  
    // ...  
    if y > 0  
        then y := y -1  
    else  
        y := z ; x := x- 1  
    endif  
done
```

À chaque tour de boucle , le variant décroît strictement pour l'ordre lexicographique sur $\mathbb{N} \times \mathbb{N}$.

```
{z ∈ N}  
x := z ; y := z;  
while x > 0 | y > 0 do  
    // ...  
    if y > 0  
        then y := y -1  
        else  
            y := z ; x := x- 1  
        endif  
    done
```

À chaque tour de boucle , le variant décroît strictement pour l'ordre lexicographique sur $\mathbb{N} \times \mathbb{N}$. Comme cet ordre est bien fondé, la boucle termine.

```
{z ∈ N}  
x := z ; y := z;  
while x > 0 | y > 0 do  
    // ...  
    if y > 0  
        then y := y -1  
        else  
            y := z ; x := x- 1  
        endif  
    done
```

À chaque tour de boucle , le variant décroît strictement pour l'ordre lexicographique sur $\mathbb{N} \times \mathbb{N}$. Comme cet ordre est bien fondé, la boucle termine. Exercice : appliquer explicitement les règles appropriées du calcul de Hoare.

Cours 6 : Ensembles et relations

- ▶ Définitions (rappels ?)
- ▶ Applications principales
- ▶ Relations bien fondées et terminaison
- ▶ Applications à la certification de programmes

Définitions

Soient E et F deux ensembles. Une **relation** de E vers F est un sous-ensemble R du produit cartésien $E \times F$. On note $x R y$ pour $(x, y) \in R$.

Définitions (suite)

Dans les définitions ci-dessous, on suppose $E = F$.

- ▶ R est **réflexive** si $x R x$ pour tout x dans E .
- ▶ R est **irréflexive** si la proposition $\forall x \in E, \neg(x R x)$ est vraie.
- ▶ R est **symétrique** si la proposition $\forall x y \in E, x R y \implies y R x$ est vraie.
- ▶ R est **antisymétrique** si la proposition
 $\forall x y \in E, x R y \wedge y R x \implies x = y$ est vraie.
- ▶ R est **transitive** si la proposition
 $\forall x y z \in E, x R y \wedge y R z \implies x R z$ est vraie.

Definitions (suite)

- ▶ R est un **préordre** si R est réflexive et transitive.
- ▶ R est une **relation d'ordre** si R est un préordre et R est antisymétrique.
- ▶ R est un **ordre strict** si R est transitive et irréflexive.
- ▶ R est **bien fondée** si et seulement s'il n'existe aucune suite infinie $x_1, x_2, \dots, x_n, \dots$ telle que $x_{i+1} \mathcal{R} x_i$ pour tout i .
- ▶ R est une **relation d'équivalence** si R est réflexive, symétrique, et transitive.

Exemples et Applications Principales

Préordres

- ▶ La relation “divise” sur \mathbb{Z}
- ▶ L’accessibilité dans un graphe
- ▶ Simulation

Ordres

- ▶ La relation “divise” sur \mathbb{N}
- ▶ L’accessibilité dans un graphe sans cycle
- ▶ L’inclusion \subseteq dans $\mathcal{P}(A)$
- ▶ L’ordre alphabétique sur les chaînes ASCII

Ordres stricts

- ▶ L'ordre $<$ dans \mathbb{N} , dans \mathbb{Z} , dans \mathbb{R} , etc.
- ▶ L'inclusion \subsetneq dans $\mathcal{P}(A)$
- ▶ Héritage simple (Java)

Relations d'équivalence

- ▶ L'égalité sur n'importe quel ensemble E ,
- ▶ Représentations équivalentes d'une même valeur
- ▶ $S \equiv S'$ ssi $\forall P Q, \{P\} \in \{Q\} \iff \{P\} \in S' \{Q\}$

Relations et Ensembles Bien Fondés

- ▶ Rappels,
- ▶ Utilité : terminaison de programmes impératifs et récursifs
- ▶ Principe de récurrence transfinie
- ▶ Bibliothèque de relations bien fondées
- ▶ Exemples

Rappels

Soit E un ensemble et \mathcal{R} une relation sur E . On dit que E est bien fondé pour \mathcal{R} si et seulement s'il n'existe aucune suite infinie $x_1, x_2, \dots, x_n, \dots$ telle que $x_{i+1} \mathcal{R} x_i$ pour tout i .

Définition

On dit aussi que \mathcal{R} est bien fondée sur E , ou que (E, \mathcal{R}) est bien fondé.

Remarque

\mathcal{R} n'est pas forcément une relation d'ordre strict. Par exemple les relations suivantes sont bien fondées :

- ▶ $\{(2, 4)\}$
- ▶ $\{(i, i + 1) | i \in \mathbb{N}\}$

Utilisation des ensembles bien fondés : terminaison des programmes impératifs

Soit $(E, <)$ un ensemble bien fondé :

$$\frac{\langle P \wedge b \wedge \text{variant} = z \rangle \mathrel{S} \langle P \wedge \text{variant} < z \wedge \text{variant} \in E \rangle}{\langle P \rangle \text{ while } b \text{ do } S \text{ done } \langle P \wedge \neg b \rangle}$$

Utilisation des ensembles bien fondés : terminaison des programmes récursifs

```
public static int fact(int n) {  
    return n <= 0 ? 1 : n * fact(n-1);  
}
```

$$0 < n \implies 0 \leq n-1 < n$$

```
public static int fib(int n) {  
    return n <= 1 ? 1 : fib(n-1) + fib(n-2);  
}
```

$$1 < n \implies 0 \leq n-2 < n-1 < n.$$

Utilisation des ensembles bien fondés : certification des programmes

Exemple

```
public static int fib(int a, int b, int n) {  
    return n == 0 ? a : fib(b, a+b, n-1);  
}
```

```
public static int fib (int n) {  
    assert (n >= 0);  
    return fib(1,1,n);  
}
```

Propriétés des ensembles bien fondés

Théorème

Soit $(E, <)$ un ensemble bien fondé non vide.

Alors il existe au moins un élément minimal de E pour la relation $<$.

Rappel

Un élément x de E est *minimal* s'il n'existe aucun élément y tel que $y < x$.

Preuve (par l'absurde)

Soit $(E, <)$ un ensemble bien fondé non vide.

1. On suppose la propriété fausse,

Preuve (par l'absurde)

Soit $(E, <)$ un ensemble bien fondé non vide.

1. On suppose la propriété fausse, c'est à dire
 $\forall x \in E, \exists y \in E, y < x.$
2. Puisque E n'est pas vide, il existe un élément $a \in E$.
3. On construit une suite d'éléments de E :
 - ▶ $x_0 = a$

Preuve (par l'absurde)

Soit $(E, <)$ un ensemble bien fondé non vide.

1. On suppose la propriété fausse, c'est à dire
 $\forall x \in E, \exists y \in E, y < x.$
2. Puisque E n'est pas vide, il existe un élément $a \in E$.
3. On construit une suite d'éléments de E :
 - ▶ $x_0 = a$
 - ▶ Soit $n \in \mathbb{N}$, il existe un élément $y \in E$ tel que $y < x_n$.

Preuve (par l'absurde)

Soit $(E, <)$ un ensemble bien fondé non vide.

1. On suppose la propriété fausse, c'est à dire
 $\forall x \in E, \exists y \in E, y < x.$
2. Puisque E n'est pas vide, il existe un élément $a \in E$.
3. On construit une suite d'éléments de E :
 - ▶ $x_0 = a$
 - ▶ Soit $n \in \mathbb{N}$, il existe un élément $y \in E$ tel que $y < x_n$. On pose $x_{n+1} = y$.
4. On construit donc une suite $x_0 > x_1 > \dots > x_n > x_{n+1} > \dots$, ce qui est impossible.
5. C.Q.F.D.

Catalogue d'ensembles bien fondés

La connaissance des théorèmes suivants permet de ne pas revenir trop souvent à la définition initiale.

Théorème

Tout ordre strict sur un ensemble **fini** est bien fondé.

Catalogue d'ensembles bien fondés

La connaissance des théorèmes suivants permet de ne pas revenir trop souvent à la définition initiale.

Théorème

Tout ordre strict sur un ensemble **fini** est bien fondé.

Preuve

Soit E un ensemble fini $n = \text{card}(E)$, et \prec un ordre strict sur E .

Catalogue d'ensembles bien fondés

La connaissance des théorèmes suivants permet de ne pas revenir trop souvent à la définition initiale.

Théorème

Tout ordre strict sur un ensemble **fini** est bien fondé.

Preuve

Soit E un ensemble fini $n = \text{card}(E)$, et \prec un ordre strict sur E . On suppose qu'il existe une suite infinie $x_1 \succ x_2 \succ \dots x_i \succ x_{i+1} \succ \dots$ dans E .

Catalogue d'ensembles bien fondés

La connaissance des théorèmes suivants permet de ne pas revenir trop souvent à la définition initiale.

Théorème

Tout ordre strict sur un ensemble **fini** est bien fondé.

Preuve

Soit E un ensemble fini $n = \text{card}(E)$, et \prec un ordre strict sur E . On suppose qu'il existe une suite infinie $x_1 \succ x_2 \succ \dots x_i \succ x_{i+1} \succ \dots$ dans E .

Il existe i et j tels que $1 \leq i < j \leq n + 1$ tels que $x_i = x_j$ (**principe des tiroirs**).

Catalogue d'ensembles bien fondés

La connaissance des théorèmes suivants permet de ne pas revenir trop souvent à la définition initiale.

Théorème

Tout ordre strict sur un ensemble **fini** est bien fondé.

Preuve

Soit E un ensemble fini $n = \text{card}(E)$, et \prec un ordre strict sur E . On suppose qu'il existe une suite infinie $x_1 \succ x_2 \succ \dots x_i \succ x_{i+1} \succ \dots$ dans E .

Il existe i et j tels que $1 \leq i < j \leq n + 1$ tels que $x_i = x_j$ (**principe des tiroirs**).

Comme \prec est transitive, on a $x_i \prec x_i = x_j$, ce qui est impossible (car \prec est irréflexive). **C.Q.F.D.**

Exemples

- ▶ L'ordre alphabétique strict sur les caractères ASCII est bien fondé.
- ▶ Soit E un ensemble fini ; L'inclusion stricte \subsetneq définie sur $\mathcal{P}(E)$ est bien fondée.

Exemples

- ▶ L'ordre alphabétique strict sur les caractères ASCII est bien fondé.
- ▶ Soit E un ensemble fini ; L'inclusion stricte \subsetneq définie sur $\mathcal{P}(E)$ est bien fondée.

Note

L'hypothèse de finitude est nécessaire dans l'exemple précédent
(Exercice)

Théorème

Soient R et S deux relations sur E . Si $R \subseteq S$ et S est bien fondée, alors R est bien fondée.

Preuve en TD

Théorème

Soient R et S deux relations sur E . Si $R \subseteq S$ et S est bien fondée, alors R est bien fondée.

Preuve en TD

Exemple

Soit la relation “est un diviseur strict” sur \mathbb{N} .

Si n est un diviseur strict de p , alors $n < p$.

Donc la relation considérée est bien fondée.

Théorème (de l'image réciproque)

Soient E et F deux ensembles, R une relation sur E et S une relation sur F . Soit une fonction $f : E \rightarrow F$. Si on a :

- ▶ $\forall x y \in E, x R y \implies f(x) S f(y)$
- ▶ (F, S) bien fondée

Alors (E, R) est bien fondé.

Preuve en TD

Exemple

L'ensemble des chaînes ASCII, avec la relation “ s est plus courte que s' ”.

Cours No 7

Théorème (Rappel)

Si \langle_A et \langle_B sont bien fondés, alors leur produit lexicographique est bien fondé sur $A \times B$.

Exemple : La fonction d'Ackermann

```
public static int ack(int m, int n) {  
    // m, n >= 0  
    return m == 0 ?  
        n + 1 :  
        (n == 0 ?  
            ack(m-1, 1) :  
            ack (m - 1, ack (m, n-1)));  
}
```

Le calcul de $\text{ack}(m, n)$ termine toujours (si $m, n \in \mathbb{N}$).

Le calcul de $\text{ack}(m, n)$ termine toujours (si $m, n \in \mathbb{N}$).

Preuve

Les appels récursifs $\text{ack}(m-1, 1)$, $\text{ack}(m, n-1)$, et
 $\text{ack}(m - 1, \text{ack}(m, n-1))$ ont des arguments strictement inférieurs
à (m, n) dans le produit lexicographique de $(\mathbb{N}, <)$ par lui-même.
Ce dernier étant bien fondé, l'évaluation de $\text{ack}(m, n)$ termine
toujours.

Remarque

Le calcul peut être très long. Si $m=4$ et $n= 2$, le résultat est
 $2^{65536} - 3$. Cette fonction est utilisée dans l'évaluation de
performances de compilateurs (benchmarks), de même que la
fonction de Fibonacci.

Exemples (suite)

Soit A un alphabet fini, pourvu d'un ordre strict sur les caractères. L'ensemble des chaînes de caractères de longueur donnée n est bien fondé pour l'ordre alphabétique strict.

Preuve par récurrence sur n

- ▶ Si $n = 0$, la relation considérée est vide, donc bien fondée.

Exemples (suite)

Soit A un alphabet fini, pourvu d'un ordre strict sur les caractères. L'ensemble des chaînes de caractères de longueur donnée n est bien fondé pour l'ordre alphabétique strict.

Preuve par récurrence sur n

- ▶ Si $n = 0$, la relation considérée est vide, donc bien fondée.
- ▶ On suppose la propriété vraie au rang n .

Exemples (suite)

Soit A un alphabet fini, pourvu d'un ordre strict sur les caractères. L'ensemble des chaînes de caractères de longueur donnée n est bien fondé pour l'ordre alphabétique strict.

Preuve par récurrence sur n

- ▶ Si $n = 0$, la relation considérée est vide, donc bien fondée.
- ▶ On suppose la propriété vraie au rang n .
- ▶ On associe à toute chaîne de longueur $n + 1$ un couple composé de sa première lettre et du reste de la chaîne.

Exemples :

$$\text{"aabb"} \rightsquigarrow (\text{'a'}, \text{"abb"})$$

$$\text{"aabc"} \rightsquigarrow (\text{'a'}, \text{"abc"})$$

Soient deux chaînes s et t de longueur $n + 1$. s est strictement inférieure à t si et seulement si le couple associé à s est strictement inférieur au couple associé à t pour le produit lexicographique :

- ▶ de l'ordre sur les caractères (bien fondé)
- ▶ par l'ordre sur les chaînes de longueur n (bien fondé par l'hypothèse de récurrence).

On applique donc deux théorèmes :

- ▶ Celui sur le produit lexicographique,
- ▶ Le théorème sur l'image réciproque

L'ensemble des chaînes de caractères de longueur bornée (inférieure ou égale à un entier n donné) est bien fondé pour l'ordre alphabétique strict.
(deux preuves).

L'ensemble des chaînes de caractères de longueur finie n'est pas bien fondé pour l'ordre alphabétique strict.
(déjà vu)

Un exemple un peu plus élaboré

On considère les chaînes « triées » par ordre décroissant, par exemple "**zzyuuttttrd**"

Question

L'ordre alphabétique est-il bien fondé pour cette catégorie de chaînes ?

On remarque que l'ordre considéré est très complexe :

- ▶ Entre "z" et "y", il y a une infinité de mots, par exemple tous les mots de la forme " y^nx^p "
- ▶ Entre "yyxx" et "yyx", il y a une infinité de mots de la forme "yx x^nb^n "

En conclusion, on ne peut pas borner la longueur d'une suite strictement décroissante, ni la longueur des chaînes qui la composent.

Un résultat non intuitif/trivial

Théorème

L'ensemble des chaînes de caractères « triées » est bien fondé pour l'ordre alphabétique.

Un résultat non intuitif/trivial

Théorème

L'ensemble des chaînes de caractères « triées » est bien fondé pour l'ordre alphabétique.

Preuve

Les remarques sur la complexité de l'ordre considéré laissent penser que la simple récurrence sur \mathbb{N} ne suffira pas. On va donc devoir construire des outils plus puissants.

Principe de Récurrence Transfinie

- ▶ Soit $(E, <)$ un ensemble bien fondé,
- ▶ Soit P un prédictat défini sur E tel que
 - ▶ pour tout $x \in E$,
 - ▶ si $P(y)$ est vraie pour tout $y < x$,
 - ▶ alors $P(x)$ est vraie.
- ▶ Alors, on en déduit la proposition

$$\forall x \in E, P(x)$$

Ce principe est la généralisation de toutes les formes de récurrence usuelles. On l'appelle aussi Principe de récurrence bien fondée. Noter l'anglicisme “induction”.

Preuve de ce principe

On rappelle les hypothèses :

- ▶ Soit $(E, <)$ un ensemble bien fondé,
- ▶ Soit P un prédictat défini sur E tel que
 - ▶ pour tout $x \in E$,
 - ▶ si $P(y)$ est vraie pour tout $y < x$,
 - ▶ alors $P(x)$

1. On considère l'ensemble $X = \{x \in E \mid \neg P(x)\}$,
2. S'il est vide, c'est gagné.
3. Sinon, soit $y \in X$.

1. Par définition de X , on a $\neg P(y)$,
2. Par hypothèse (et par contraposition), on déduit qu'il existe un z dans E tel que $y > z \wedge \neg P(z)$

1. Par définition de X , on a $\neg P(y)$,
2. Par hypothèse (et par contraposition), on déduit qu'il existe un z dans E tel que $y > z \wedge \neg P(z)$
3. En itérant ce processus, on construit une suite

$$x_1 = x > x_2 = y > x_3 > \dots x_n > \dots$$

1. Par définition de X , on a $\neg P(y)$,
2. Par hypothèse (et par contraposition), on déduit qu'il existe un z dans E tel que $y > z \wedge \neg P(z)$
3. En itérant ce processus, on construit une suite
$$x_1 = x > x_2 = y > x_3 > \dots x_n > \dots$$
4. Ce qui est impossible car $(E, <)$ est bien fondé.

1. Par définition de X , on a $\neg P(y)$,
2. Par hypothèse (et par contraposition), on déduit qu'il existe un z dans E tel que $y > z \wedge \neg P(z)$
3. En itérant ce processus, on construit une suite
$$x_1 = x > x_2 = y > x_3 > \dots x_n > \dots$$
4. Ce qui est impossible car $(E, <)$ est bien fondé.
5. Donc X est vide, et par conséquent tous les éléments de E satisfont P .

Remarque

Une démonstration par récurrence transfinie se fait de façon « descendante », à la différence de la récurrence sur \mathbb{N} , qui peut être ascendante ou descendante.

On revient aux chaînes triées

Soit une chaîne « triée ». On considère sa première lettre, ainsi que le nombre d'occurrences de cette lettre. Appelons “mesure” d'une chaîne cette information.

Exemples

- ▶ "zzy" : ("z",2)
- ▶ "zyyyyyyyyy" : ("z",1)
- ▶ "yyyyyyyyyyyyyyyytttttt" : ("y",20)

On revient aux chaînes triées

Soit une chaîne « triée ». On considère sa première lettre, ainsi que le nombre d'occurrences de cette lettre. Appelons “mesure” d'une chaîne cette information.

Exemples

- ▶ "zzy" : ("z",2)
- ▶ "zyyyyyyyyy" : ("z",1)
- ▶ "yyyyyyyyyyyyyyyytttttt" : ("y",20)

Intérêt : Le produit lexicographique de l'ordre sur les caractères ASCII et de l'ordre strict sur \mathbb{N} est bien fondé.

On peut envisager une récurrence transfinie sur cet ordre.

La preuve

On prouve le lemme suivant :

Pour tout caractère c et tout entier naturel n , il n'existe aucune suite strictement décroissante issue d'une chaîne de mesure (c, n) .

On note $P(c, n)$ cette propriété.

- ▶ Soient (c, n) , on suppose que $P(c, n)$ est vraie pour tout couple (c', p) inférieur à (c, n) .

- ▶ Soient (c, n) , on suppose que $P(c, n)$ est vraie pour tout couple (c', p) inférieur à (c, n) .
- ▶ Considérons une suite infinie $s_i (i \in \mathbb{N})$ strictement décroissante de chaînes, telle que la mesure de s_1 soit (c, n) .

- ▶ Soient (c, n) , on suppose que $P(c, n)$ est vraie pour tout couple (c', p) inférieur à (c, n) .
- ▶ Considérons une suite infinie $s_i (i \in \mathbb{N})$ strictement décroissante de chaînes, telle que la mesure de s_1 soit (c, n) .
 - ▶ Soit il existe une chaîne s_i dont la mesure est inférieure à (c, n) , et on peut appliquer l'hypothèse de récurrence. On a donc une contradiction.

- ▶ Soient (c, n) , on suppose que $P(c, n)$ est vraie pour tout couple (c', p) inférieur à (c, n) .
- ▶ Considérons une suite infinie $s_i (i \in \mathbb{N})$ strictement décroissante de chaînes, telle que la mesure de s_1 soit (c, n) .
 - ▶ Soit il existe une chaîne s_i dont la mesure est inférieure à (c, n) , et on peut appliquer l'hypothèse de récurrence. On a donc une contradiction.
 - ▶ Soit toutes les chaînes sont de mesure (c, n) , et en coupant les n premiers caractères de chaque chaîne on obtient une suite infinie de chaînes de mesure inférieure à (c, n) , ce qui est impossible.
- ▶ Donc $P(c, n)$ est prouvée.
- ▶ **C.Q.F.D.**

Quelques exemples pas trop évidents

La suite de Syracuse

```
{n ∈ N1}  
while n > 1 do  
    if n % 2 = 0  
        then n := n/2  
    else n := 3*n + 1  
    endif  
done
```

Démo : src/Syracuse.java

Suites de Goodstein

- ▶ On prend un nombre naturel, exprimé sous forme d'une *décomposition héréditaire* en base b .

$$1026 = 2^{2^{2^1+1}+2^1} + 2^1$$

Suites de Goodstein

- ▶ On prend un nombre naturel, exprimé sous forme d'une *décomposition héréditaire* en base *b*.

$$1026 = 2^{2^{2^1+1}+2^1} + 2^1$$

- ▶ A chaque étape, on incrémente la base, et on enlève 1.

$$\begin{aligned} 3^{3^{3^1+1}+3^1} + 3^1 - 1 &= 3^{3^{3^1+1}+3^1} + 2 \\ &= 11972515182562019788602740026717047105683 \end{aligned}$$

Suites de Goodstein

- ▶ On prend un nombre naturel, exprimé sous forme d'une *décomposition héréditaire* en base *b*.

$$1026 = 2^{2^{2^1+1}+2^1} + 2^1$$

- ▶ A chaque étape, on incrémente la base, et on enlève 1.

$$\begin{aligned} 3^{3^{3^1+1}+3^1} + 3^1 - 1 &= 3^{3^{3^1+1}+3^1} + 2 \\ &= 11972515182562019788602740026717047105683 \end{aligned}$$

- ▶ On continue : $4^{4^{4^1+1}+4^1} + 1$, $5^{5^{5^1+1}+5^1}$, etc.

Le cas (plus simple) de G_4

On peut choisir une structure de donnée appropriée :

$$\begin{aligned} 4 &= 2^2 \\ 3^3 - 1 &= 3^2 \times 2 + 3 \times 2 + 2 \\ &\quad 4^2 \times 2 + 4 \times 2 + 1 \\ &\quad 5^2 \times 2 + 5 \times 2 + 0 \\ 6^2 \times 2 + 6 \times -1 &= 6^2 \times 2 + 6 \times 1 + 5 \\ &\dots \end{aligned}$$

Un item de la suite sera représenté par un quadruplet (b, a_2, a_1, a_0)

Démo G4silent.java essaie de calculer la longueur de la suite.

Une première certitude

La suite issue de 4 est finie.

Une première certitude

La suite issue de 4 est finie.

- ▶ On prend comme variant le triplet (a_2, a_1, a_0) .

Une première certitude

La suite issue de 4 est finie.

- ▶ On prend comme variant le triplet (a_2, a_1, a_0) .
- ▶ A chaque étape ce triplet décroît strictement pour l'ordre lexicographique sur $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$.

Une première certitude

La suite issue de 4 est finie.

- ▶ On prend comme variant le triplet (a_2, a_1, a_0) .
- ▶ A chaque étape ce triplet décroît strictement pour l'ordre lexicographique sur $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$.
- ▶ Or cet ordre est bien fondé.

Une première certitude

La suite issue de 4 est finie.

- ▶ On prend comme variant le triplet (a_2, a_1, a_0) .
- ▶ A chaque étape ce triplet décroît strictement pour l'ordre lexicographique sur $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$.
- ▶ Or cet ordre est bien fondé.
- ▶ Donc la suite finit par s'arrêter sur $(0, 0, 0)$.

Peut-on *calculer* la longueur de la suite ?

Une première certitude

La suite issue de 4 est finie.

- ▶ On prend comme variant le triplet (a_2, a_1, a_0) .
- ▶ A chaque étape ce triplet décroît strictement pour l'ordre lexicographique sur $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$.
- ▶ Or cet ordre est bien fondé.
- ▶ Donc la suite finit par s'arrêter sur $(0, 0, 0)$.

Peut-on *calculer* la longueur de la suite ? Apparemment pas en faisant des tests sur machine !

Première idée

On trace les valeurs successives de la suite :

Le programme G4.java imprime pour chaque étape

- ▶ La base b (Numéro courant de l'item)
- ▶ le triplet (a_2, a_1, a_0)

Première idée

On trace les valeurs successives de la suite :

Le programme G4.java imprime pour chaque étape

- ▶ La base b (Numéro courant de l'item)
- ▶ le triplet (a_2, a_1, a_0)

On n'est pas vraiment avancé !

Meilleure idée

On trace uniquement les étapes significatives : le “chiffre” a_0 passe par 0.

Le programme G4zero.java réalise cette restriction.

Calcul de quelques items de la suite (parmi les premiers)

5 (2, 2, 0)

6 (2, 1, 5)

11 (2, 1, 0)

23 (2, 0, 0)

47 (1, 23, 0)

95 (1, 22, 0)

191 (1, 21, 0)

Détection de régularités $3 \times 2^n - 1$ ($i, j, 0$).

Preuve d'invariants (par récurrence)

Lemme 1

Si le $3 \times 2^n - 1$ -ième terme de la suite est $(i, j + 1, 0)$, alors le $3 \times 2^{n+1} - 1$ -ième est $(i, j, 0)$,

Preuve d'invariants (par récurrence)

Lemme 1

Si le $3 \times 2^n - 1$ -ième terme de la suite est $(i, j + 1, 0)$, alors le $3 \times 2^{n+1} - 1$ -ième est $(i, j, 0)$,

Preuve

En effet, le 3×2^n -ème terme est $(i, j, 3 \times 2^n - 1)$,

Preuve d'invariants (par récurrence)

Lemme 1

Si le $3 \times 2^n - 1$ -ième terme de la suite est $(i, j + 1, 0)$, alors le $3 \times 2^{n+1} - 1$ -ième est $(i, j, 0)$,

Preuve

En effet, le 3×2^n -ème terme est $(i, j, 3 \times 2^n - 1)$, puis après $3 \times 2^n - 1$ itérations, on aura $(i, j, 0)$.

Or $3 \times 2^n - 1 + 3 \times 2^n - 1 + 1 = 3 \times 2^{n+1} - 1$.

Preuve d'invariants (par récurrence)

Lemme 1

Si le $3 \times 2^n - 1$ -ième terme de la suite est $(i, j + 1, 0)$, alors le $3 \times 2^{n+1} - 1$ -ième est $(i, j, 0)$,

Preuve

En effet, le 3×2^n -ème terme est $(i, j, 3 \times 2^n - 1)$, puis après $3 \times 2^n - 1$ itérations, on aura $(i, j, 0)$.

$$\text{Or } 3 \times 2^n - 1 + 3 \times 2^n - 1 + 1 = 3 \times 2^{n+1} - 1.$$

Lemme 2

Si le $3 \times 2^n - 1$ -ème terme de la suite est $(i, j, 0)$, alors le $3 \times 2^{n+j} - 1$ -ème est $(i, 0, 0)$,

Lemme 3

Si le $3 \times 2^n - 1$ -ème terme de la suite est $(i + 1, 0, 0)$, alors le $3 \times 2^{n+3 \times 2^n} - 1$ -ème est $(i, 0, 0)$,

En effet, on a l'évolution suivante :

- Le 3×2^n -ème terme est $(i, 3 \times 2^n - 1, 3 \times 2^n - 1)$

Lemme 3

Si le $3 \times 2^n - 1$ -ème terme de la suite est $(i + 1, 0, 0)$, alors le $3 \times 2^{n+3 \times 2^n} - 1$ -ème est $(i, 0, 0)$,

En effet, on a l'évolution suivante :

- ▶ Le 3×2^n -ème terme est $(i, 3 \times 2^n - 1, 3 \times 2^n - 1)$
- ▶ Après $3 \times 2^n - 1$ étapes, on obtient le
 $3 \times 2^n + 3 \times 2^n - 1 = 3 \times 2^{n+1}$ -ème terme : $(i, 3 \times 2^n - 1, 0)$

Lemme 3

Si le $3 \times 2^n - 1$ -ème terme de la suite est $(i + 1, 0, 0)$, alors le $3 \times 2^{n+3 \times 2^n} - 1$ -ème est $(i, 0, 0)$,

En effet, on a l'évolution suivante :

- ▶ Le 3×2^n -ème terme est $(i, 3 \times 2^n - 1, 3 \times 2^n - 1)$
- ▶ Après $3 \times 2^n - 1$ étapes, on obtient le
 $3 \times 2^n + 3 \times 2^n - 1 = 3 \times 2^{n+1}$ -ème terme : $(i, 3 \times 2^n - 1, 0)$
- ▶ En appliquant le lemme 2, on obtient que le
 $3 \times 2^{n+1+3 \times 2^n - 1} - 1$ -ème terme est $(i, 0, 0)$.

En utilisant ces lemmes et quelques calculs, on obtient un résumé de la suite :

$$3 \quad (2, 2, 2)$$

$$4 \quad (2, 2, 1)$$

$$5 = 3 \times 2^1 - 1 \quad (2, 2, 0)$$

En utilisant ces lemmes et quelques calculs, on obtient un résumé de la suite :

$$3 \quad (2, 2, 2)$$

$$4 \quad (2, 2, 1)$$

$$5 = 3 \times 2^1 - 1 \quad (2, 2, 0)$$

$$3 \times 2^3 - 1 \quad (2, 0, 0)$$

En utilisant ces lemmes et quelques calculs, on obtient un résumé de la suite :

3	(2, 2, 2)
4	(2, 2, 1)
$5 = 3 \times 2^1 - 1$	(2, 2, 0)
$3 \times 2^3 - 1$	(2, 0, 0)
$3 \times 2^{27} - 1$	(1, 0, 0)

En utilisant ces lemmes et quelques calculs, on obtient un résumé de la suite :

3	(2, 2, 2)
4	(2, 2, 1)
$5 = 3 \times 2^1 - 1$	(2, 2, 0)
$3 \times 2^3 - 1$	(2, 0, 0)
$3 \times 2^{27} - 1$	(1, 0, 0)
$3 \times 2^{27+3 \times 2^{27}} - 1$	(0, 0, 0)

En utilisant ces lemmes et quelques calculs, on obtient un résumé de la suite :

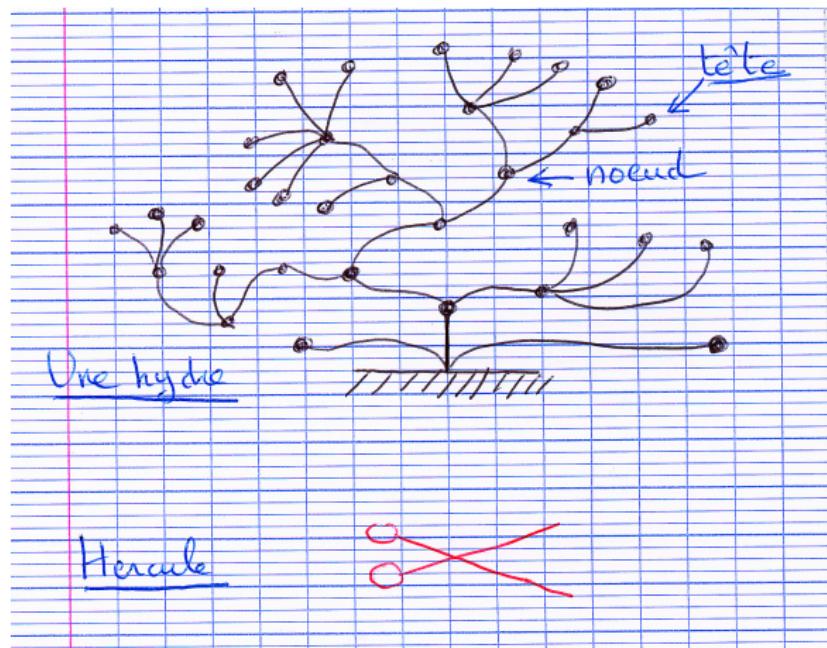
3	(2, 2, 2)
4	(2, 2, 1)
$5 = 3 \times 2^1 - 1$	(2, 2, 0)
$3 \times 2^3 - 1$	(2, 0, 0)
$3 \times 2^{27} - 1$	(1, 0, 0)
$3 \times 2^{27+3 \times 2^{27}} - 1$	(0, 0, 0)

La suite de Goostein issue de 4 a donc $3 \times 2^{402653211} - 1$ termes !

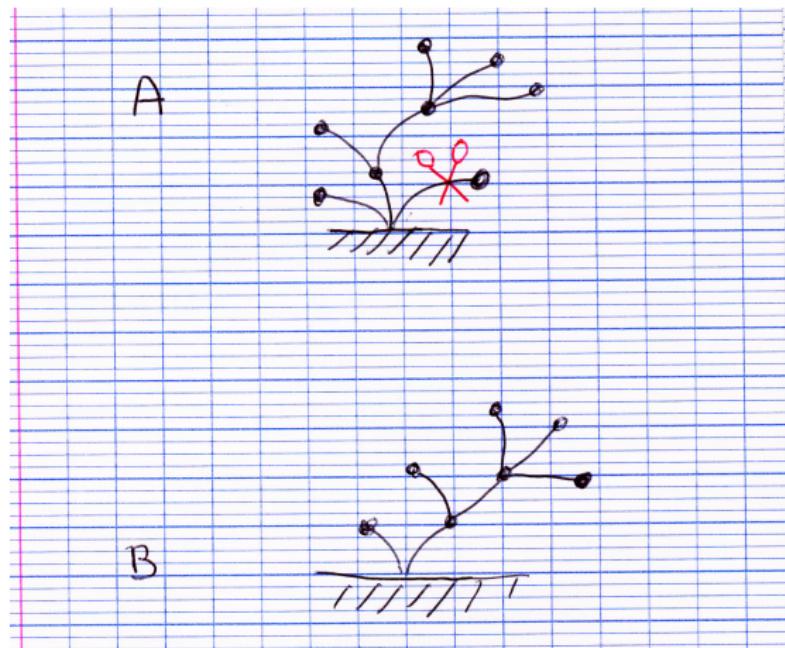
Conclusions à tirer de cet exemple

- ▶ Coopération entre calcul et preuve,
- ▶ Les tests peuvent donner une idée fausse du comportement d'un programme

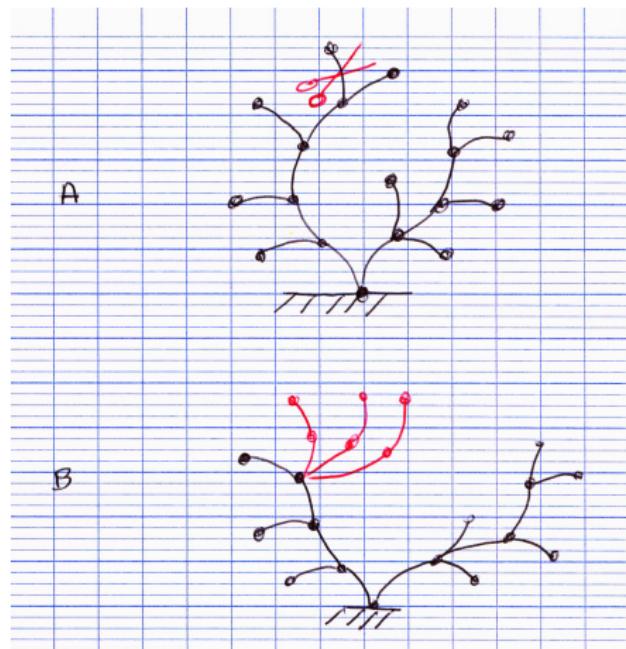
Batailles d'Hydre



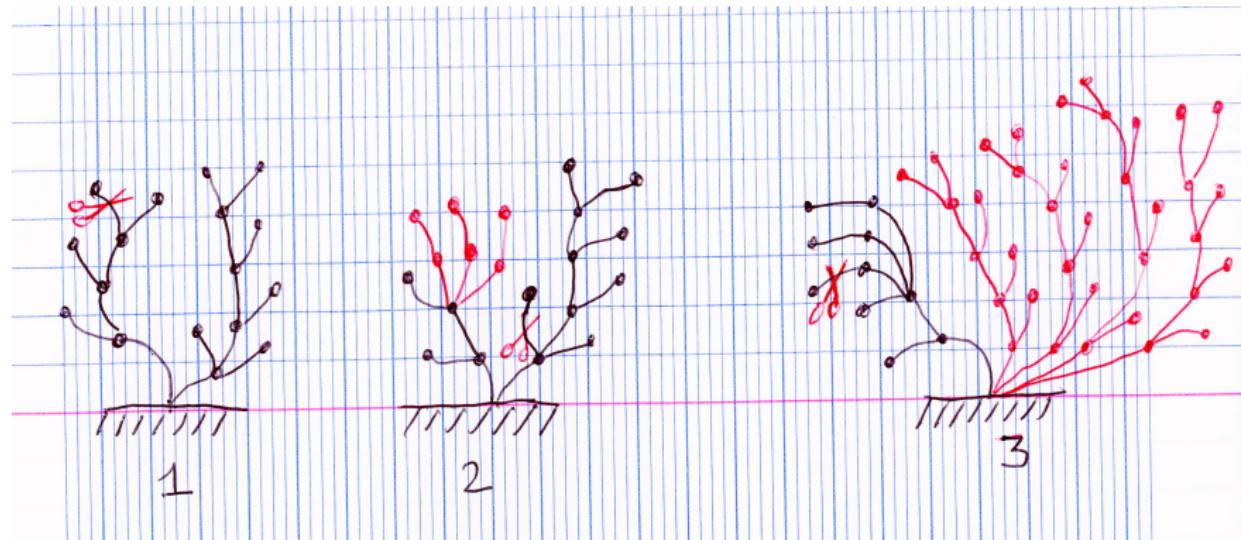
Décapitation : premier cas



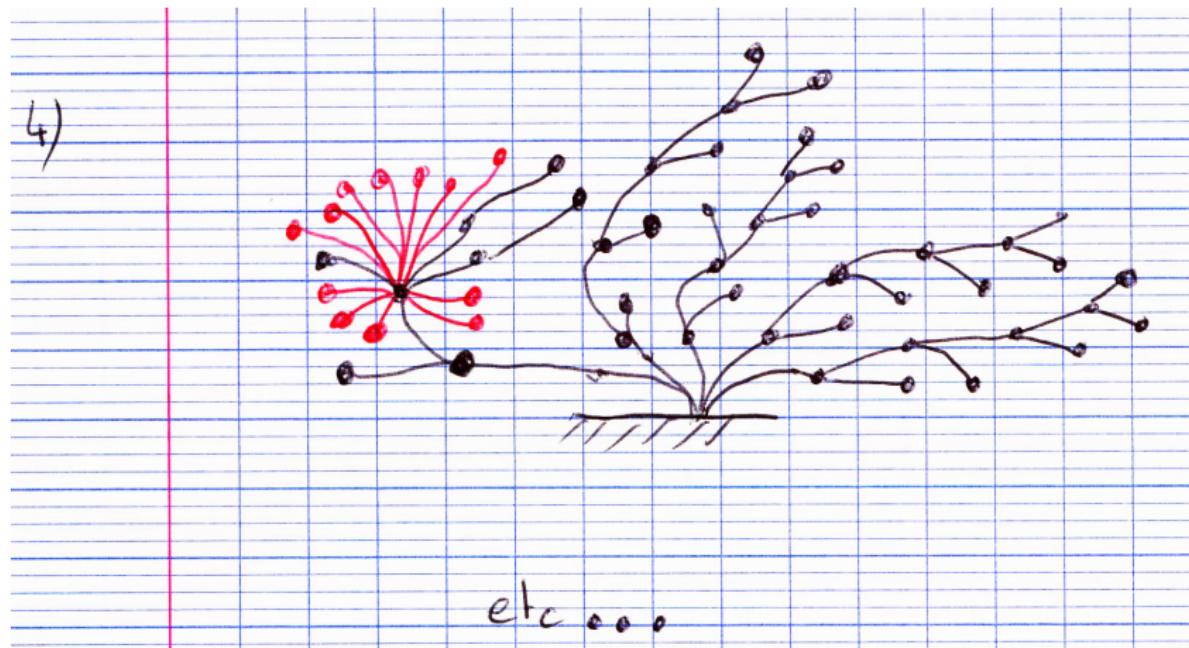
Décapitation : second cas



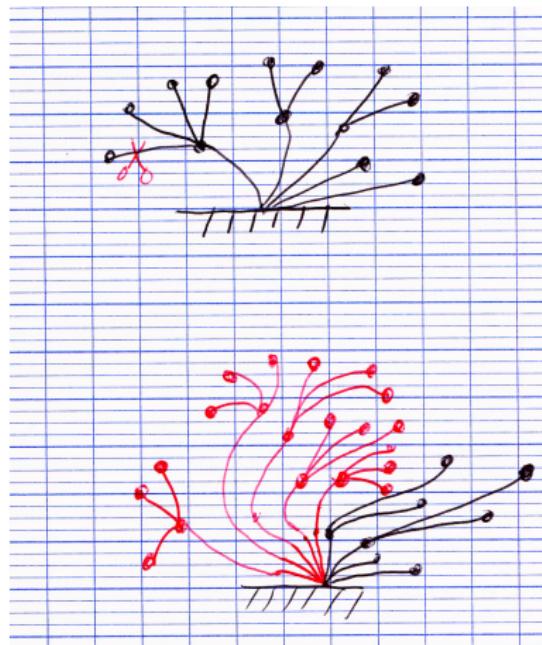
Un combat (début)



Un combat (suite)

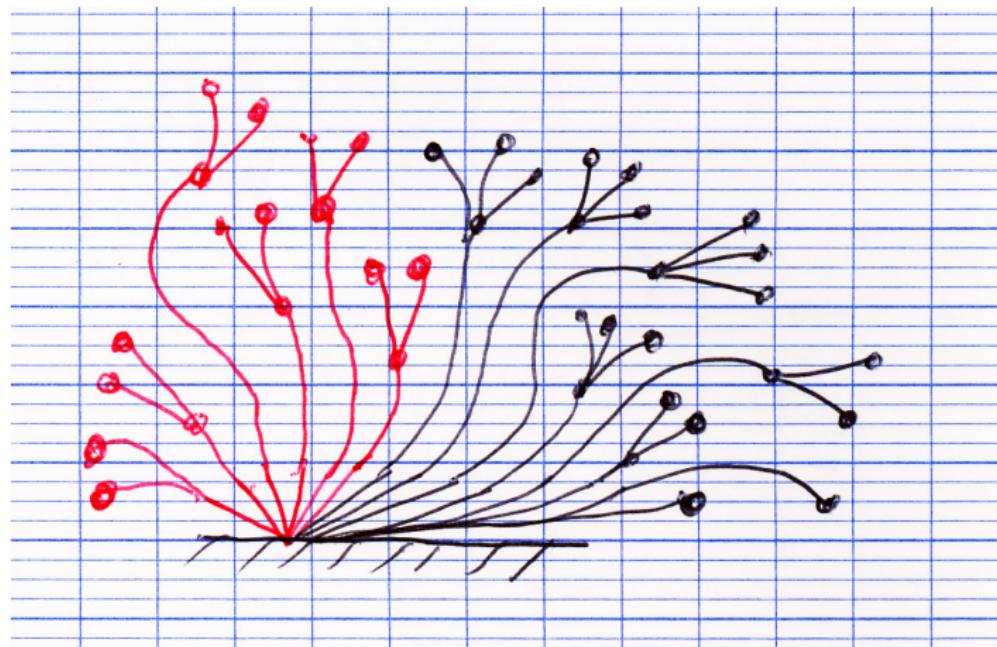


Étude d'une espèce particulière d'hydre (tentacules de longueur ≤ 2)



└ Ensembles et Relations Bien Fondés

└ Hercule contre l'Hydre

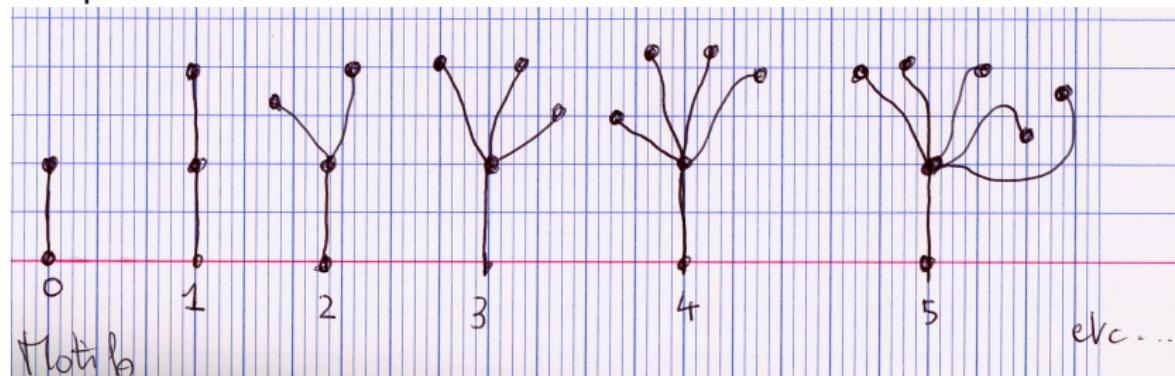


└ Ensembles et Relations Bien Fondés

└ Hercule contre l'Hydre

On veut prouver qu'Hercule finit toujours par vaincre une telle hydre.

Pour ce faire, on commence par associer un nombre entier à chaque motif élémentaire :



On constitue des « sacs » de nombres :

$$\begin{aligned}\{\{4, 2, 2, 0, 0\}\} &> \{\{3, 3, 3, 3, 3, 2, 2, 0, 0\}\} \\ &> \{\{3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 0, 0\}\} \\ &> \{\{3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 0, 0\}\}\end{aligned}$$

Il « suffit » de montrer que l'ensemble des sacs est bien fondé.
Pour ça, on a besoin d'outils mathématiques.

Il « suffit » de montrer que l'ensemble des sacs est bien fondé.

Pour ça, on a besoin d'outils mathématiques.

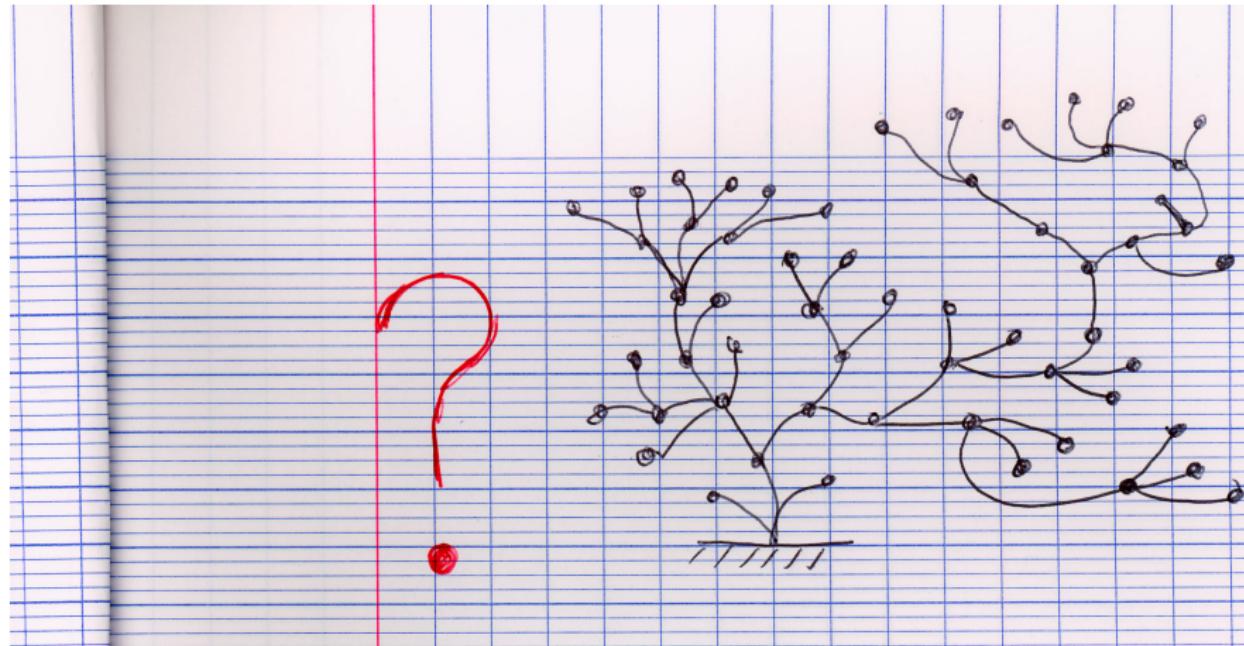
On commence par donner une version très générale de la récurrence, appelée **récurrence transfinie**.

Note

En anglais, « récurrence » se dit « induction »

- ▶ L'ensemble des sacs est bien fondé (se prouve en raisonnant sur le plus gros nombre d'un sac et combien de fois il apparaît).
- ▶ A chaque tour, le sac associé à l'Hydre diminue.

Et pour une hydre quelconque ?



Théorème : Hercule finit toujours par vaincre l'hydre, quelle que soit sa stratégie de croissance.

Preuve en Coq :

- ▶ Bibliothèque sur les ordinaux (dont bonne fondation d' ϵ_0) : 2510 lignes
 1. En premier lieu, preuve directe (inspirée de la preuve de Tait)
 2. Simplification : collaboration avec une équipe du LRI.
- ▶ Association d'une suite strictement décroissante d'ordinaux à l'évolution d'une hydre : 939 lignes.

Remarque pour les mathématiciens

La complexité des problèmes de terminaison précédents peut se mesurer à l'aide des ordinaux de Cantor :

Compte à rebours : ω^3

Hydre dont les tentacules sont de longueur ≤ 2 : ω^ω

Hydre quelconque : ϵ_0 (plus grand que $\omega^\omega, \omega^{\omega^\omega}, \omega^{\omega^{\omega^\omega}}, \omega^{\omega^{\omega^{\omega^\omega}}} \dots$)

Cours 8

```
/* a[0] to a[n-1] is the array to sort */
int iPos;
int iMin;
for (iPos = 0; iPos < n; iPos++) {
    /* find the min element in the unsorted a[iPos .. n-1] */
    iMin = iPos;
    for (i = iPos+1; i < n; i++) {
        if (a[i] < a[iMin]) {
            iMin = i;
        }
    }
    if ( iMin != iPos ) {
        swap(a, iPos, iMin);
    }
} /* (c) Wikipedia */
```

Problèmes posés

- ▶ deux boucles imbriquées,
- ▶ structures de contrôles différentes du **while**,
- ▶ sécurité des accès aux tableaux (en **C, Java**)
- ▶ modification d'une structure de donnée : **swap**, affectation $a[i] = \dots$
- ▶ utilisation de fonctions ou procédures

Accès aux éléments d'un tableau, et fonctions partielles

Certaines instructions (principalement d'affectation) peuvent provoquer des problèmes à l'exécution :

- ▶ Parfois ignorés (**C**),
- ▶ Provoquent des erreurs (**Java**)

```
x = a[i] + 3;
```

```
y = x / z;
```

```
a[i]++;
```

Pour vérifier à l'avance qu'aucune de ces affectations ne provoquera de bugs ou d'erreurs à l'exécution, on peut renforcer la règle de l'affectation. On note **def(*e*)** la proposition exprimant que l'évaluation de l'expression *e* ne pourra pas provoquer d'erreur (pas d'accès illicite à un tableau, de division par zéro, d'arithmetic overflow, etc.)

Pour vérifier à l'avance qu'aucune de ces affectations ne provoquera de bugs ou d'erreurs à l'exécution, on peut renforcer la règle de l'affectation. On note $\text{def}(e)$ la proposition exprimant que l'évaluation de l'expression e ne pourra pas provoquer d'erreur (pas d'accès illicite à un tableau, de division par zéro, d'arithmetic overflow, etc.)

Si l'expression e présente un tel risque, la règle de l'affectation devient : $\{0 \leq \text{def}(e) \wedge P[e/x]\} \times := e \{P\}$

Exemple :

$\{0 \leq i < n \wedge a[i] \neq 0 \wedge 10000/a[i] \geq 0\} \times := 10000/a[i] \{x \geq 0\}$

Autre exemple

$$\{y^2 + 1 \neq 0 \wedge 100/(y^2 + 1) \geq 0\}$$

y := y * y + 1;

$$\{y \neq 0 \wedge 100/y \geq 0\}$$

z := 100/y

$$\{z \geq 0\}$$

Autre exemple

$$\{y^2 + 1 \neq 0 \wedge 100/(y^2 + 1) \geq 0\}$$

$y := y * y + 1;$

$$\{y \neq 0 \wedge 100/y \geq 0\}$$

$z := 100/y$

$$\{z \geq 0\}$$

En revanche, on ne peut pas prouver le triplet

$$\{y > 0\} \quad y := y - 1; z := 1000/y \quad \{z \geq 0\}$$

car $y > 0$ n'implique pas $y - 1 \neq 0 \wedge 1000/(y - 1) \geq 0$.

Exercices

1

Soit a un tableau d'entiers, de taille n , Démontrer que le code suivant ne provoque pas d'erreur d'indice de tableau à l'exécution :

```
i := 0; while i < n do x := x * a[i]; i := i + 1 done
```

2

Montrer qu'il n'en est pas de même si on remplace $i < n$ par $i \leq n$ dans la condition d'arrêt de la boucle.

Le problème de l'affectation $a[i] := e$

Il faut bien sûr ajouter une précondition $0 \leq i < n$, où n est la taille du tableau.

On peut proposer la règle suivante :

$$\{0 \leq i < n \wedge P[e/a[i]]\} \quad a[i] := e \quad \{P\}$$

On suppose que la tableau a est de longueur $n \geq 2$. Le triplet suivant serait valide.

$$\{\forall j, 0 \leq j < n - 1 \implies a[j] \leq a[j + 1]\}$$

$a[1] := a[0] - 1;$

$$\{\forall j, 0 \leq j < n - 1 \implies a[j] \leq a[j + 1]\}$$

La règle proposée n'est donc pas correcte.

Analyse du problème

Les règles de déduction comme celles de Hoare associent aux variables des objets mathématiques afin de raisonner sur ces objets.

Analyse du problème

Les règles de déduction comme celles de Hoare associent aux variables des objets mathématiques afin de raisonner sur ces objets. Un tableau n'est pas un objet mathématique, puisqu'on peut le modifier au cours du temps.

Analyse du problème

Les règles de déduction comme celles de Hoare associent aux variables des objets mathématiques afin de raisonner sur ces objets. Un tableau n'est pas un objet mathématique, puisqu'on peut le modifier au cours du temps.

Solution : considérer une version mathématique des tableaux (et d'autres structures : files, arbres de recherche, etc.)

Wikipedia, toujours

Extrait de l'article : Structure de données persistante

En informatique, une structure de données persistante est une structure de données qui préserve ses versions antérieures lorsqu'elle est modifiée ; une telle structure est observationnellement immuable, car ses opérations ne la modifient pas en place (de manière visible) mais renvoient au contraire de nouvelles structures.

Les structures qui ne sont pas persistantes sont dites éphémères.

Ce type de structures de données est particulièrement courant en programmation logique et fonctionnelle. Dans un programme purement fonctionnel, où toute donnée est immuable, les structures de données sont automatiquement totalement persistantes. Les structures persistantes peuvent aussi être obtenues en utilisant des modifications en place des données et peuvent alors être parfois plus efficaces, en temps ou en espace, que leurs contreparties purement fonctionnelles.

Tableaux persistants

- ▶ Un tableau persistant est *immuable*
- ▶ Si a est un tableau de taille n , i un indice compris entre 0 et $n - 1$, x une valeur, alors le tableau obtenu en remplaçant la i -ème case par x est un *nouveau tableau* noté $a[i := x]$.

Tableaux persistants

- ▶ Un tableau persistant est *immutable*
- ▶ Si a est un tableau de taille n , i un indice compris entre 0 et $n - 1$, x une valeur, alors le tableau obtenu en remplaçant la i -ème case par x est un *nouveau tableau* noté $a[i := x]$.

L'affectation $a[i] := x$ est alors un synonyme de $a := a[i := x]$. La variable affectée est donc a et non $a[i]$.

Tableaux persistants

- ▶ Un tableau persistant est *immutable*
- ▶ Si a est un tableau de taille n , i un indice compris entre 0 et $n - 1$, x une valeur, alors le tableau obtenu en remplaçant la i -ème case par x est un *nouveau tableau* noté $a[i := x]$.

L'affectation $a[i] := x$ est alors un synonyme de $a := a[i := x]$. La variable affectée est donc a et non $a[i]$.

La règle associée à ce type d'affectation est donc :

$$\{0 \leq i < n \wedge P[a[i := e]]\} \quad a[i] := e \quad \{P\}$$

Exemple

On ne peut plus prouver le triplet faux :

$$\{\forall j, 0 \leq j < n - 1 \implies a[j] \leq a[j + 1]\}$$

$a[1] := a[0] - 1;$

$$\{\forall j, 0 \leq j < n - 1 \implies a[j] \leq a[j + 1]\}$$

Exemple

On ne peut plus prouver le triplet faux :

$$\{\forall j, 0 \leq j < n - 1 \implies a[j] \leq a[j + 1]\}$$

$a[1] := a[0] - 1;$

$$\{\forall j, 0 \leq j < n - 1 \implies a[j] \leq a[j + 1]\}$$

En effet, posons $b = a[1 := a[0] - 1]$ Le triplet correct est :

$$\{\forall j, 0 \leq j < n - 1 \implies b[j] \leq b[j + 1]\}$$

$a[1] := a[0] - 1;$

$$\{\forall j, 0 \leq j < n - 1 \implies a[j] \leq a[j + 1]\}$$

Exemple

On ne peut plus prouver le triplet faux :

$$\{\forall j, 0 \leq j < n - 1 \implies a[j] \leq a[j + 1]\}$$

$a[1] := a[0] - 1;$

$$\{\forall j, 0 \leq j < n - 1 \implies a[j] \leq a[j + 1]\}$$

En effet, posons $b = a[1 := a[0] - 1]$ Le triplet correct est :

$$\{\forall j, 0 \leq j < n - 1 \implies b[j] \leq b[j + 1]\}$$

$a[1] := a[0] - 1;$

$$\{\forall j, 0 \leq j < n - 1 \implies a[j] \leq a[j + 1]\}$$

Or, on n'a pas l'implication :

$$\forall j, 0 \leq j < n - 1 \implies a[j] \leq a[j + 1]$$

\implies

$$\forall j, 0 \leq j < n - 1 \implies b[j] \leq b[j + 1]$$

(un contre-exemple simple suffit à s'en persuader)

Les tableaux persistants (suite)

Pour écrire des raisonnements corrects, il faut définir l'opération de mise à jour $a[i := x]$:

Axiomes

Soit a un tableau de taille $n > 0$, i un indice tel que $0 \leq i < n$, et x une valeur. Soit $b = a[i := x]$.



$$b[i] = x$$



$$\forall j, 0 \leq j < n \wedge i \neq j \implies b[j] = a[j]$$

Un exemple simple

Montrer que si le tableau a est trié, alors le code suivant maintient cette propriété.

```
a[1] := a[0]
```

Un exemple simple

Montrer que si le tableau a est trié, alors le code suivant maintient cette propriété.

```
a[1] := a[0]
```

La post-condition et la pré-condition sont la formule

$$\forall i, 0 \leq i < n - 1 \implies a[i] \leq a[i + 1]$$

Soit $b = a[1 := a[0]]$.

Il faut donc vérifier que la pré-condition implique la formule

$$\forall i, 0 \leq i < n - 1 \implies b[i] \leq b[i + 1]$$

On a $b[1] = a[0]$ et $b[i] = a[i]$ pour tout $i \in (0..n - 1) \setminus \{1\}$

1. Soit $i \in 0..n - 2$,
2. si $i = 0$, alors $b[i] = a[0] = b[1] = b[i + 1]$
3. si $i = 1$, alors $b[i] = a[0] \leq a[2] = b[i + 1]$
4. si $i \geq 2$, alors $b[i] = a[i] \leq a[i + 1] = b[i + 1]$

Limitations de cette approche

*In computing, **aliasing** describes a situation in which a data location in memory can be accessed through different symbolic names in the program. Thus, modifying the data through one name implicitly modifies the values associated to all aliased names, which may not be expected by the programmer. As a result, aliasing makes it particularly difficult to understand, analyze and optimize programs. Aliasing analysers intend to make and compute useful information for understanding aliasing in programs.*

[Wikipedia](#), article **Array Aliasing**

Conséquences

- ▶ Malgré sa portée limitée, la formalisation des tableaux par des objets persistants permet de certifier de nombreux modules et algorithmes.
- ▶ Pour les programmes à écrire, les outils de développement et les langages appropriés peuvent faciliter l'obtention de programmes corrects et raisonnablement efficaces,
- ▶ Pour les programmes déjà (mal) écrits, utiliser des outils d'analyse genre **Frama-C**.

On reprend le tri par sélection

Pour pallier la complexité du code (boucles imbriquées), on va privilégier une **approche descendante** : construction progressive du code à partir de la spécification.

On reprend le tri par sélection

Pour pallier la complexité du code (boucles imbriquées), on va privilégier une **approche descendante** : construction progressive du code à partir de la spécification.

Remarque

La spécification d'un tri ne tient pas compte de la **méthode** utilisée : quicksort, bubble sort, selection sort, etc. Celle-ci apparaîtra lors de **raffinements** (transitions de la spécification abstraite vers la réalisation concrète).

Spécification d'un tri (sur tableau d'entiers)

Soit t un tableau d'entiers de taille n .

{ $a = t$ }

tri de a

{ a est trié et

a exactement les mêmes éléments que t

(avec la même multiplicité) }

Définitions

- ▶ a est trié si et seulement si $\forall i \in 0..n - 2, a[i] \leq a[i + 1]$
- ▶ $a \equiv t$ si et seulement si $\forall x \in \mathbb{Z}, \text{card}(t^{-1}(x)) = \text{card}(a^{-1}(x))$

Définitions

- ▶ a est trié si et seulement si $\forall i \in 0..n - 2, a[i] \leq a[i + 1]$
- ▶ $a \equiv t$ si et seulement si $\forall x \in \mathbb{Z}, \text{card}(t^{-1}(x)) = \text{card}(a^{-1}(x))$

Spécification

{ $a = t$ }

tri de a

{ a est trié et $a \equiv t$ }

$$\text{Inv}(a, i) \quad =_{\text{def}} \quad a \equiv t \wedge 0 \leq i \leq n \wedge \\ (\forall j \in 0..i-2, a[j] \leq a[j+1]) \wedge \\ (\forall j \in 0..i-1, \forall k \in i..n-1, a[j] \leq a[k])$$

```
{ a = t }
i:=0;
while i < n -1
{ Inv(a, i) }
do
// echanger a[i] et a[m], avec m position de la valeur
// minimum du segment a[i]..a[m - 1]
i := i+1
done
```

```
{ a = t }
i:=0;
while i < n-1
{ Inv(a,i) }
do
    // calcul de m ne doit pas modifier a et i
    { m est une position du plus petit élément de
      a[i..n - 1]}
    // echanger a[i] et a[m]
    i := i+1
done
{a est trié et a ≡ t}
```

On s'occupe de l'échange de deux cases

Soit a un tableau de taille n .

```
procedure swap(a,i,j) {
    if i ≠ j
        then {
            int x := a[i];
            a[i] := a[j];
            a[j] := x
        }
    else
        skip
    endif
```

```
{i, j ∈ 0..n − 1 ∧ P[permuter(a, i, j)/a] }  
swap(a, i, j);  
{ P }
```

permuter(*a, i, j*) est le tableau *b* de taille *n* tel que

$$\begin{aligned} b[i] &= a[j] \\ b[j] &= a[i] \\ \forall k \in 0..n-1 \setminus \{i, j\}, \quad b[k] &= a[k] \end{aligned}$$

(Preuve en exercice). On prouve également l'équivalence
 $\text{permuter}(a, i, j) \equiv a$.

Suite du raisonnement/ développement

On suppose qu'on est capable de calculer la position m du minimum des valeurs $a[i] \dots a[n - 1]$ sans modifier les valeurs de a et i . On pourra donc assurer l'invariant après ce calcul.

```
while i < n-1
{ Inv(a, i) }
do
    // calcul de m ne doit pas modifier a et i
    { m est une position du plus petit élément de
      a[i..n - 1]}
    { Inv(a, i) }
    // échanger a[i] et a[m]
    i := i+1
done
```

Preuve de l'invariant : initialisation

On a clairement :

$$a = t \quad \implies$$

$$(a \equiv t \wedge 0 \leq 0 \leq n \wedge \\ (\forall j \in 0.. - 2, a[j] \leq a[j + 1]) \wedge \\ (\forall j \in 0.. - 1, \forall k \in i..n - 1, a[j] \leq a[k]))$$

Preuve de l'invariant : fin de la boucle

Lorsque $i = n$, l'invariant devient :

$$\begin{aligned} & (a \equiv t \wedge 0 \leq n \leq n \wedge \\ & (\forall j \in 0..n-2, a[j] \leq a[j+1]) \wedge \\ & (\forall j \in 0..n-1, \forall k \in n..n-1, a[j] \leq a[k])) \end{aligned}$$

Cette proposition implique que a est une version triée de t .

Maintien de l'invariant

il suffit de prouver (sous l'hypothèse de correction du calcul de m).

```
while i < n-1
{ Inv(a,i) }
do
    // calcul de  $m$  ne doit pas modifier  $a$  et  $i$ 
    {  $m$  est une position du plus petit élément de
         $a[i..n - 1]$ }
    { Inv(a,i) }
    // échanger  $a[i]$  et  $a[m]$ 
    i := i+1
done
```

Calcul de m

On utilise des variables locales.

```
int find_minimum(a, i)
{ int m := i;
  int k := i +1 ;
  while k < n do
    if a[k] < a[m]
      then m := k
      else skip
      endif;
  k:= k+1;
  done;
  return k
}
```

Correction totale

- ▶ Fonction `find_minimum` : variant : $n - k$
- ▶ Boucle **while** : variant : $n - i$

Remarques importantes

- ▶ Au lieu de prendre un programme tel quel, on l'a développé en même temps que sa preuve
- ▶ Dans le traitement de `find_minimum`, on a procédé en deux étapes :
 1. On suppose qu'on peut écrire cette fonction, sans qu'elle modifie certaines variables du programme, et on utilise cette propriété dans le reste de la preuve,
 2. On écrit cette fonction, puis on prouve sa correction et qu'elle satisfait bien la propriété annoncée.

```
{ a = t }
i:=0;
while i < n-1
do
    m := find_minimum(a,i);
    swap(a, i, m);
    i := i+1
done
{a est trié et a ≡ t}
```

Comment marchent les outils ?

- ▶ On considère un exemple simple,
- ▶ Les détails seront donnés dans les UE de L3 et M1.

Comment marchent les outils ?

- ▶ On considère un exemple simple,
- ▶ Les détails seront donnés dans les UE de L3 et M1.

```
{ a ∈ ℤ ∧ n ∈ ℕ }  
a := 1 ; p := n;  
while p > 0  
do  
    if p % 2 = 0  
        then  
            y := y*y ; p := p/2  
        else  
            a := a*y ; p := p-1  
        endif  
done  
{ a = xn }
```

Pour la correction partielle, l'utilisateur ajoute un invariant.

Pour la correction partielle, l'utilisateur ajoute un invariant.

```
{ a ∈ ℤ ∧ n ∈ ℕ }  
a := 1 ; p := n;  
while p > 0  
Invariant: { xn = a yp }  
do  
    if p % 2 = 0  
        then  
            y := y*y ; p := p/2  
        else  
            a := a*y ; p := p-1  
        endif  
    done  
{ a = xn }
```

Le système ne propose pas de triplets, mais des **obligations de preuves**

Pour la règle de l'affectation un triplet $\{P\} \ x := e \ \{Q\}$ propose par exemple l'obligation $P \longrightarrow Q[e/x]$.

Le système ne propose pas de triplets, mais des **obligations de preuves**

Pour la règle de l'affectation un triplet $\{P\} \ x := e \ \{Q\}$ propose par exemple l'obligation $P \longrightarrow Q[e/x]$.

$Q[e/x]$ est appelée la **précondition la plus faible** associée à $x := e$ et à Q .

Le système ne propose pas de triplets, mais des **obligations de preuves**

Pour la règle de l'affectation un triplet $\{P\} \ x := e \ \{Q\}$ propose par exemple l'obligation $P \longrightarrow Q[e/x]$.

$Q[e/x]$ est appelée la **précondition la plus faible** associée à $x := e$ et à Q .

Ce schéma s'utilise pour tous les types d'instruction.

Première obligation engendrée : initialisation du while :

$$\begin{aligned} a \in \mathbb{Z} \wedge n \in \mathbb{N} \\ \implies \\ x^n = a1^n \end{aligned}$$

Cette obligation peut être résolue par un solveur automatique.

Deuxième obligation engendrée : sortie du while :

$$\begin{aligned}x^n = a y^p \wedge p = 0 \\ \implies a = x^n\end{aligned}$$

Cette obligation peut être résolue par un solveur automatique.

Les autres obligations de preuve proviennent de l'analyse du triplet

$$\{ x^n = a y^p \wedge p > 0 \}$$

if $p \% 2 = 0$

then

$y := y * y$; $p := p / 2$

else

$a := a * y$; $p := p - 1$

endif

$$\{ x^n = a y^p \}$$

Les autres obligations de preuve proviennent de l'analyse du triplet

$$\{ x^n = a y^p \wedge p > 0 \}$$

if $p \% 2 = 0$

then

$y := y * y ; p := p / 2$

else

$a := a * y ; p := p - 1$

endif

$$\{ x^n = a y^p \}$$

Deux cas seront étudiés selon la parité de p .

- ▶ Pour p pair :

$$x^n = ay^p \wedge p > 0 \wedge p \% 2 = 0$$

 \implies

$$x^n = a(y^2)^{p/2}$$

- ▶ Pour p impair :

$$x^n = ay^p \wedge p > 0 \wedge p \% 2 = 1$$

 \implies

$$x^n = (ay)y^{p-1}$$

Ces obligations peuvent se prouver automatiquement ou facilement, suivant le(s) prouveur(s) utilisé(s).

Correction totale

On ajoute un variant :

```
{ a ∈ ℤ ∧ n ∈ ℕ }  
a := 1 ; p := n;  
while p > 0  
    Invariant: {  $x^n = ay^p$  }  
    Variant: { p }  
    do  
        if p % 2 = 0  
        then  
            y := y*y ; p := p/2  
        else  
            a := a*y ; p := p-1  
        endif  
    done  
{ a =  $x^n$  }
```

Les obligations engendrées sont :

- ▶ Pour p pair :

$$x^n = ay^p \wedge p > 0 \wedge p \% 2 = 0$$

$$\implies$$

$$0 \leq p/2 < p$$

- ▶ Pour p impair :

$$x^n = ay^p \wedge p > 0 \wedge p \% 2 = 1$$

$$\implies$$

$$0 \leq p - 1 < p$$

Notes sur l'usage des générateurs d'obligations de preuves

- ▶ Si toutes les obligations de preuves générées à partir d'un programme et de sa spécification sont résolues, tout va bien.

Notes sur l'usage des générateurs d'obligations de preuves

- ▶ Si toutes les obligations de preuves générées à partir d'un programme et de sa spécification sont résolues, tout va bien.
- ▶ sinon,

Notes sur l'usage des générateurs d'obligations de preuves

- ▶ Si toutes les obligations de preuves générées à partir d'un programme et de sa spécification sont résolues, tout va bien.
- ▶ sinon,
 - ▶ Votre programme peut être faux.

Notes sur l'usage des générateurs d'obligations de preuves

- ▶ Si toutes les obligations de preuves générées à partir d'un programme et de sa spécification sont résolues, tout va bien.
- ▶ sinon,
 - ▶ Votre programme peut être faux. L'inspection des obligations de preuves paraissant absurde (genre $n \leq n - 1$) peut aider la correction (ici, un indice de boucle mal géré).

Notes sur l'usage des générateurs d'obligations de preuves

- ▶ Si toutes les obligations de preuves générées à partir d'un programme et de sa spécification sont résolues, tout va bien.
- ▶ sinon,
 - ▶ Votre programme peut être faux. L'inspection des obligations de preuves paraissant absurde (genre $n \leq n - 1$) peut aider la correction (ici, un indice de boucle mal géré).
 - ▶ votre programme est juste, mais

Notes sur l'usage des générateurs d'obligations de preuves

- ▶ Si toutes les obligations de preuves générées à partir d'un programme et de sa spécification sont résolues, tout va bien.
- ▶ sinon,
 - ▶ Votre programme peut être faux. L'inspection des obligations de preuves paraissant absurde (genre $n \leq n - 1$) peut aider la correction (ici, un indice de boucle mal géré).
 - ▶ votre programme est juste, mais
 - ▶ Vos invariants sont faux à l'initialisation d'une boucle.

Notes sur l'usage des générateurs d'obligations de preuves

- ▶ Si toutes les obligations de preuves générées à partir d'un programme et de sa spécification sont résolues, tout va bien.
- ▶ sinon,
 - ▶ Votre programme peut être faux. L'inspection des obligations de preuves paraissant absurde (genre $n \leq n - 1$) peut aider la correction (ici, un indice de boucle mal géré).
 - ▶ votre programme est juste, mais
 - ▶ Vos invariants sont faux à l'initialisation d'une boucle.
réfléchir !

Notes sur l'usage des générateurs d'obligations de preuves

- ▶ Si toutes les obligations de preuves générées à partir d'un programme et de sa spécification sont résolues, tout va bien.
- ▶ sinon,
 - ▶ Votre programme peut être faux. L'inspection des obligations de preuves paraissant absurde (genre $n \leq n - 1$) peut aider la correction (ici, un indice de boucle mal géré).
 - ▶ votre programme est juste, mais
 - ▶ Vos invariants sont faux à l'initialisation d'une boucle.
réfléchir !
 - ▶ Vos invariants sont trop faibles, soit pour impliquer la post-condition, soit pour prouver qu'ils sont maintenus à chaque itération :

Notes sur l'usage des générateurs d'obligations de preuves

- ▶ Si toutes les obligations de preuves générées à partir d'un programme et de sa spécification sont résolues, tout va bien.
- ▶ sinon,
 - ▶ Votre programme peut être faux. L'inspection des obligations de preuves paraissant absurde (genre $n \leq n - 1$) peut aider la correction (ici, un indice de boucle mal géré).
 - ▶ votre programme est juste, mais
 - ▶ Vos invariants sont faux à l'initialisation d'une boucle.
réfléchir !
 - ▶ Vos invariants sont trop faibles, soit pour impliquer la post-condition, soit pour prouver qu'ils sont maintenus à chaque itération : Renforcer ces invariants par des propositions qui vous paraissent vraies, puis relancer l'analyse automatique du programme.
 - ▶ Vos invariants sont corrects mais restent non prouvés.

Notes sur l'usage des générateurs d'obligations de preuves

- ▶ Si toutes les obligations de preuves générées à partir d'un programme et de sa spécification sont résolues, tout va bien.
- ▶ sinon,
 - ▶ Votre programme peut être faux. L'inspection des obligations de preuves paraissant absurde (genre $n \leq n - 1$) peut aider la correction (ici, un indice de boucle mal géré).
 - ▶ votre programme est juste, mais
 - ▶ Vos invariants sont faux à l'initialisation d'une boucle. réfléchir !
 - ▶ Vos invariants sont trop faibles, soit pour impliquer la post-condition, soit pour prouver qu'ils sont maintenus à chaque itération : Renforcer ces invariants par des propositions qui vous paraissent vraies, puis relancer l'analyse automatique du programme.
 - ▶ Vos invariants sont corrects mais restent non prouvés. Utiliser plusieurs prouveurs automatiques, puis en cas d'échec, des assistants interactifs à la preuve

À consulter

- ▶ Documentations de Why3, Frama-C, Rodin, Atelier B (générateurs d'obligations de preuves)
- ▶ Articles sur “precondition la plus faible” (weakest precondition)
- ▶ Solveurs automatiques : Alt-Ergo, “Simplify theorem prover” , etc.
- ▶ Assistants à la preuve : Coq, Isabelle, HOL, PVS