Mathématiques pour l'informatique

Fabrice Rossi

29 février 2012

Table des matières

1 Introduction

Objectifs du cours :
- utiliser des techniques et outils mathématiques pour analyser des programmes ;
- comprendre les fondements mathématiques de l'informatique.

Thèmes :

analyse de programmes : montrer qu'un programme est correct, étudier son coût

induction : aller plus loin que l'induction sur les entiers (induction structurelle) pour analyser des programmes plus complexes

automates : modéliser les états d'un programme, analyser des chaînes de caractères

2 Rappels de logique

2.1 Prédicat

Un prédicat est une formule logique portant éventuellement sur des variables libres. Étant données les valeurs des variables libres, on peut calculer la valeur du prédicat qui est soit **vrai** soit **faux** (si on parle du prédicat, vraie ou fausse si on parle de la valeur).

Exemple : $P = x > 0$ est un prédicat portant sur la variable x. Si x est négatif, alors P vaut faux. On note souvent $P(x)$ un tel prédicat pour insister sur le fait qu'il est construit à partir de la variable libre x. La variable peut ensuite être liée dans une construction plus complexe.

2.2 Connecteurs logiques

Pour construire un prédicat, on utilise des connecteurs logiques, essentiellement :

$\wedge$ **et logique** : $a \wedge b$ est vrai si et seulement si a est vrai et b est vrai ;

$\vee$ **ou logique** : $a \vee b$ est vrai si a est vrai ou si b est vrai ;

$\neg$ **négation** : $\neg a$ est vrai si et seulement si a est faux.

On utilise aussi des connecteurs définis à partir des précédents :

$\Rightarrow$ **implication** : $a \Rightarrow b$ est vrai si et seulement si a implique b, c'est-à-dire si $\neg a \vee b$ est vrai ;

$\Leftrightarrow$ **équivalence** : $a \Leftrightarrow b$ est vrai si et seulement si a implique b et b implique a, c'est-à-dire si $(a \wedge b) \vee (\neg a \wedge \neg b)$ est vrai.

2.3 Quantificateurs

On utilise aussi deux quantificateurs qui permettent d'introduire des variables liées (on ne spécifie pas la valeur d'une variable liée pour connaître la valeur d'un prédicat) :

$\exists$ **il existe** : le prédicat $\exists x,\ P(x)$ est vrai si et seulement si on peut trouver une valeur y telle que $P(y)$ soit vrai ;

$\forall$ **quel que soit** : le prédicat $\forall x,\ P(x)$ est vrai si et seulement si pour toute valeur y, $P(y)$ est vrai.

3 Preuve de programme

3.1 Motivations

Garantir le bon fonctionnement d'un programme, notamment dans des situations critiques (transport, santé, énergie). C'est le seul moyen d'obtenir une garantie forte (par opposition à des tests, par exemple).

3.2 Exemple d'introduction

3.2.1 calcul d'une somme

On écrit un programme qui calcule $\sum_{i=1}^{n} i^2$, par exemple :

```
public static int somme(int n) {
    int result = 0;
    for(int i = 1; i <= n; i++) {
        result += i*i;
    }
    return result;
}
```

Trois niveaux d'analyse :

1. se **convaincre intuitivement** que le programme fonctionne, c'est-à-dire qu'il calcule effectivement ce qu'il doit calculer. On précise à ce niveau les limites évidentes du programme (par exemple que se passe-t-il ici pour `n<0` ?) ;
2. **prouver mathématiquement** que le programme fonctionne, dans les limites identifiées ;
3. tenir compte de la **réalité informatique**, par exemple ici des limites sur les `int`.

On s'intéresse essentiellement au deuxième niveau d'analyse dans ce cours.

3.2.2 technique fondamentale : la preuve par récurrence

Rappel sur les **prédicats** : un prédicat est une propriété d'objets mathématiques, qui possède une valeur de vérité (vraie ou faux, selon que les objets considérés vérifient ou non la propriété décrite). Un prédicat est en général représenté par une formule logique.

Preuve par récurrence d'une propriété sur les entiers, $P(n)$. Si :

– $P(0)$ est vraie, et si

– pour tout $n \geq 0$, $P(n)$ est vraie implique $P(n+1)$ est vraie,

alors

– pour tout $n \geq 0$, $P(n)$ est vraie.

C'est un **axiome** de l'arithmétique de Peano : il est non trivial car une preuve doit être de longueur finie.

Exemple : montrer que $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

3.2.3 application à l'exemple

La stratégie classique consiste à montrer par récurrence qu'une boucle réalise quelque chose en s'appuyant sur le fait qu'une exécution de n tours d'une boucle est la même chose qu'une exécution de $n-1$ tours suivi par l'exécution d'un tour supplémentaire.

Ici, on montre que `result` contient $\sum_{i=1}^{n} i^2$ après n tours de boucle (propriété $P(n)$). En effet, après un tour, `result` contient 1 car le premier tour se fait avec `i=1`, et donc $P(1)$ est vraie.

Supposons $P(n)$ vraie, soit donc que `result` contient $\sum_{i=1}^{n} i^2$. Au tour suivant de la boucle, `i` contient $n+1$ et donc `result` contient $\sum_{i=1}^{n} i^2 + (n+1)^2 = \sum_{i=1}^{n+1} i^2$. Donc $P(n+1)$ est vraie.

Donc, par récurrence, $P(n)$ est donc vraie pour tout $n \geq 1$.

Remarque importante : on s'appuie sur le fait que `i` contient `n` au début du n-ème tour de la boucle. Comment le prouver ?

3.3 Formalisation

On s'inspire de la logique de Hoare et les triplets de Hoare.

3.3.1 triplet de Hoare

On cherche à montrer la correction d'un programme, c'est-à-dire formellement la correction d'un triplet de Hoare.

Un triplet est de la forme $\{P\}$ `programme` $\{Q\}$, représentation dans laquelle P et Q sont des prédicats sur les variables du `programme`.

P représente les **préconditions**, propriétés vraies avant l'exécution du `programme`. Q représente les **postconditions**, propriétés vraies après l'exécution du `programme`.

Exemple :

{ `n` ≥ 1 }

```
result = 0
i = 1
while(i <= n) {
    result = result + i * i
```

```
    i = i +1
}
```

{ `result` $= \sum_{i=1}^{\texttt{n}} i^2$ }

3.3.2 correction partielle et correction totale

Dès qu'un programme contient une boucle, il peut ne jamais s'arrêter (terminer).

La correction *partielle* correspond au cas des programmes qui peuvent ne pas se terminer, alors quand la correction *totale* exige la terminaison.

Un triplet de Hoare $\{P\}$ `programme` $\{Q\}$ est **vrai** si pour tout état initial du `programme` qui vérifie P et tel que le `programme` se termine, alors Q est vraie après l'exécution. Le programme est **partiellement correct** par rapport à P et Q.

Pour la correction *totale* on note le triplet $< P >$ `programme` $< Q >$. Le triplet est vrai si pour tout état initial du `programme` qui vérifie P, le programme se termine et que Q est vraie après l'exécution. Le programme est alors **totalement correct** par rapport à P et Q.

Que dire de notre triplet d'exemple ?

3.3.3 quelques exemples simples

traiter quelques exemples simples, du type instruction seule, puis `if then else`.

3.4 Invariant de boucle

3.4.1 principe

La difficulté principale des preuves réside dans la gestion des boucles. On la traite en général par la notion d'**invariant de boucle**. C'est un prédicat qui est vrai *avant* la boucle et qui, s'il est vrai avant un tour de boucle, le reste *après* ce tour de boucle. C'est donc une sorte de traduction informatique du principe de récurrence.

3.4.2 triplet de Hoare

Techniquement si le triplet { I et `b` } `programme` { I } est vrai, alors le triplet suivant est vrai : { I } `while (b) programme` { I et non `b` }.

Dans cette écriture, I est l'invariant de boucle. **Attention** on ne traite pas ici la terminaison : on procède en général avec un **variant** de boucle, c'est-à-dire une variable entière dont le contenu décroît strictement à chaque tour le boucle et donc la positivité est exigée dans la condition de la boucle.

3.4.3 exemple

reprendre l'exercice précédent sous la forme d'invariant de boucle.

3.4.4 digression : le problème de l'arrêt

Le traitement de la correction totale des boucles est difficile notamment car on ne peut pas déterminer automatiquement (c'est-à-dire par un programme) si un programme s'arrête.

- argument de Turing
  Considérons au contraire qu'il existe un programme A qui étant donné un autre programme B et des entrées pour ce programme, E, répond 1 si B s'arrête quand on lui demande de travailler sur E, 0 sinon.
  On définit alors un programme D qui travaillant sur une entrée E réalise la chose suivante : si A(E,E)=1 alors on fait une boucle infinie, sinon on s'arrête. Le paradoxe vient de A(D,D). En effet, si A(D,D)=1, alors D(D) ne s'arrête pas, contrairement à ce que dit A. Si au contrainre A(D,D)=0, alors D(D) s'arrête, contraitement à sa définition.

On en déduit par l'absurde que A ne peut pas exister.

– argument de Chaitin
Chaitin propose un argument plus simple au sens où A détermine si un programme B s'arrête (ou non) en considérant les données comme partie du programme.
On considère alors un programme P qui énumére tous les programmes de taille inférieure à 10 fois sa propre taille (notons que P lui même fait partie de cette liste, c'est toute l'astuce). Il applique A à chaque programme et exécute les programmes qui terminent. Il transforme la sortie de chaque programme qui termine en entier et produit comme résultat le plus petit entier qui n'est pas obtenu par ce procédé.
Par construction, P fait partie de la liste car il s'arrête. Mais il doit se faire tourner lui-même et doit donc noter son propre résultat, alors que ce résultat ne doit justement pas faire partie de la liste des résultats considérés...

3.5 Exercices simples

3.5.1 factorielle par boucle

Analyser la correction du triplet de Hoare suivant :

$\{ \texttt{n} \geq 1 \}$

```
result = 1
i = 2
while(i <= n) {
    result = result * i
    i = i +1
}
```

$\{ \texttt{result = n!} \}$

3.5.2 fibonacci par boucle

On considère la suite de Fibonacci définie par :

$$u_1 = u_2 = 1$$
$$u_{n+2} = u_{n+1} + u_n \quad \text{pour } n \geq 1$$

On considère la postcondition Q donnée par $\texttt{result} = u_{\texttt{n}}$. Donner une précondition P telle que le programme suivant soit totalement correct par rapport à P et Q (ce qu'on montrera).

```
result = 1
ancien = 1
i = 3
while(i <= n) {
    tmp = result
    result = result + ancien
    ancien = tmp
    i = i + 1
}
```

3.5.3 PGCD

On considère l'algorithme d'Euclide pour le calcul du PGCD de `a` et `b` :

```
x = a
y = b
while(y != 0) {
```

```
    r = reste de la division de x par y
    x = y
    y = r
}
```

Analyser la correction de ce programme.

3.6 Récursivité

3.6.1 fonction récursive

Une fonction est **récursive** si sa définition fait appel à elle-même. Elle doit contenir une condition d'arrêt : pour certaines valeurs de ces paramètres, la fonction ne doit pas faire appel à elle-même.

Exemple :

```
public static int identité(int n) {
    if(n==0) {
        /* condition d'arrêt */
        return 0;
    } else {
        return 1 + identité(n-1);
    }
}
```

Que peut-on dire cette fonction en terme de terminaison ?

3.6.2 analyse des fonctions récursives

Même principe que pour les autres programmes, mais plus proche des preuves par récurrence. Pour l'exemple précédent, on prouve facilement par récurrence que `identité(n)=n` pour `n` ≥ 0.

3.6.3 discussion sur la mise en oeuvre concrète

Parler ici de pile d'appels et du coût associé.

3.6.4 exercices

Reprendre factorielle et Fibonacci en version récursive. Bien mettre en avant le coût de Fibonacci récursif naïf (illustrer par un arbre d'appel).

3.6.5 récursivité terminale

Une fonction est **récursive terminale** si les seuls appels récursifs qu'elle contient constituent les dernières instructions de la branche d'exécution correspondante. En d'autres termes, chaque appel récursif est de la forme `return f(...)`.

La récursivité terminale est efficace car elle est équivalente à une boucle. Exemple simple :

```
public static int factorielle(int n) {
  return fact(n,1);
}

public static int fact(int n,int acc) {
    if (n <= 1) {
        return acc;
    } else {
```

```
        return fact(n - 1, acc * n);
    }
}
```

Montrer (mathématiquement) que ça fonctionne comme prévu en exercice.

3.7 Exercices (généraux)

Voir aussi la feuille d'exercices.

3.7.1 arithmétique

- exponentiation rapide récursive (terminale ou non)

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ (x^2)^{\frac{n}{2}} & \text{si } n \text{ est pair} \\ x(x^2)^{\frac{n-1}{2}} & \text{si } n \text{ est impair} \end{cases}$$

- Fibonacci en récursivité terminale

3.7.2 tableaux

- recherche du minimum d'un tableau (version récursive, terminale ou pas)
- méthode de Horner pour évaluer un polynôme en un point
- tri par insertion

4 Complexité(s)/coût(s) d'un programme

4.1 Motivations

Prédire les ressources nécessaires à l'exécution d'un programme (mémoire et temps). Concrètement : une tâche android ne peut pas prendre plus de 5 secondes...

4.2 Principes

Même idée générale que pour la preuve de programme : on cherche ici des résultats théoriques et exacts (relativement à un modèle d'exécution des programmes). Le but n'est pas de chronométrer un programme, mais de donner une idée (approximative) du temps d'exécution attendu en fonction de la taille du problème traité.

Modèle d'exécution :

- exécution séquentielle des instructions
- toutes les instructions coûtent environ la même chose, c'est-à-dire prennent environ le même temps

Naïf car :

- on ne tient pas compte des coûts d'accès à la mémoire
- on ne tient pas compte des opérations très coûteuses (calcul de exponentielle, par exemple)
- on ne tient pas compte du parallélisme (automatique ou pas)

L'objectif est d'obtenir des résultats asymptotiques, c'est-à-dire quand la taille des données tend vers l'infini.

4.3 Exemple simple

4.3.1 calcul d'une somme

On reprend le calcul de $\sum_{i=1}^{n} i^2$:

```
public static int somme(int n) {
    int result = 0;
    for(int i = 1; i <= n; i++) {
        result += i*i;
    }
    return result;
}
```

4.3.2 analyse

Opérations :

1. initialisation de `result` : 1
2. initialisation de `i` : 1
3. pour chaque tour de la boucle (`n` tours) :
   (a) une comparaison : 1
   (b) une addition : 1
   (c) une multiplication : 1
   (d) `i++` : 1
4. une comparaison finale (sortie de la boucle)
   Donc au total : `3 + 4 * n` opérations. Le **coût** du programme est **linéaire**.

4.4 Types d'analyse

4.4.1 notation asymptotique

On utilise les notations asymptotiques classiques :
- $O(f(n))$: dominé par $af(n)$
- $\Omega(f(n))$: minoré par $af(n)$
- $\Theta(f(n))$: encadré par $af(n)$ et $bf(n)$

4.4.2 types

On distingue différents types d'analyse :
- **cas le pire** : on considère le coût engendré par les données les plus défavorables en terme de temps d'exécution
- **en moyenne** : on considère le coût moyen relativement à une distribution sur les données
- **amortie** : coût dans le cas le pire analysé sur une séquence d'opérations (hors programme)

4.5 Exemples plus réalistes

4.5.1 max d'un tableau

Analyse du programme suivant :

```
x = tab[0]
for(int j=1; j<tab.length; j++) {
  if(tab[j] > x) {
    x = tab[j]
  }
}
```

On compte :

1. deux initialisations (`x` et `j`)

2. `tab.length-1` tours de la boucle
   (a) deux comparaisons `j<tab.length` et `tab[j] > x`
   (b) un incrément `j++`
   (c) une affectation dans certains cas
3. une comparaison pour la sortie de la boucle

Ici on fait une analyse dans le cas le pire : on compte toujours l'affectation. On obtient alors `3 + 4*(tab.length-1)`. Asymptotiquement, on a un coût linéaire en $\Theta(n)$ où n est la taille du tableau.

4.5.2 tri par insertion

Étudier un tri par insertion classique. Objectif :

1. analyse dans le cas le pire
2. arrondi et simplification dans le calcul

4.5.3 calcul de la suite de Fibonacci

Étudier le coût de la version récursive et de la version itérative. Cela conduit à la recherche de l'expression par une formule close de la suite de Fibonacci. On étudie l'équation caractéristique de la récurrence :

$$x^2 - x - 1 = 0,$$

puis on cherche une expression de la forme $ar_1^n + br_2^n$ où r_1 et r_2 sont les racines de l'équation.

4.6 Théorème maître

4.6.1 exemple du tri fusion

Présentation du tri fusion (algorithme récursif). Démonstration de la relation de récurrence sur le coût $T(n) = 2T(n/2) + \Theta(n)$ avec $T(1) = \Theta(1)$.

Analyse de la version simplifiée $T(n) = 2T(n/2) + n$ et $T(2) = 2$ pour les puissance de 2 uniquement (par récurrence).

4.6.2 le théorème

Soit $a \geq 1$ et $b > 1$ des constantes, soit $f(n)$ une fonction et soit $T(n)$ telle que $T(n) = aT(n/b) + f(n)$. Dans cette formule n/b représente soit la partie entière de n/b, soit son « plafond » (partie entière par excès). On a

1. si $f(n) = O(n^{\log_b a - \epsilon})$ pour un $\epsilon > 0$ constant, alors $T(n) = \Theta(n^{\log_b a})$.
2. si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \log n)$.
3. si $f(n) = \Omega(n^{\log_b a + \epsilon})$ pour un $\epsilon > 0$ constant et si $af(n/b) \leq cf(n)$ pour un $c < 1$ constant (et à partir d'un certain rang sur n) alors $T(n) = \Theta(f(n))$.

Exemple d'application au tri fusion, puis à la recherche dichotomique dans un tableau trié.

5 Induction structurelle

5.1 Motivation

En informatique, on définit souvent des structures récursives. Par exemple :
– une liste est soit une liste vide, soit de la forme (tête, sous-liste) ;
– un arbre binaire est soit vide, soit de la forme (racine, sous-arbre 1, sous-arbre 2).
Comment justifier ces définitions ? Comment prouver des propriétés sur ces structures ?

5.2 Formalisation

5.2.1 version très théorique

On se donne un ensemble U et :

1. un sous-ensemble B de U, la **base**
2. des **règles**, c'est-à-dire des fonctions de U^k dans U. Chaque fonction possède une arité k, c'est-à-dire un nombre de paramètres

la base et les règles forment un **schéma d'induction** qui permet de construire l'ensemble E des objets qui nous intéressent. E est défini de plusieurs façons équivalentes, notamment :

1. E est le plus petit sous-ensemble de U contenant B et stable par les règles (c'est-à-dire tel que pour toute règle f d'arité k et pour tout k-uplet $(e_1, \ldots, e_k)$ d'éléments de E, $f(e_1, \ldots, e_k)$ est dans E.
2. E est l'ensemble des objets finiment constructibles à partir de B par les règles, c'est-à-dire l'ensemble des objets e tels qu'il existe une suite d'objets $e_1, \ldots, e_p$ dans E avec $e_p = e$ telle que chaque e_i est soit dans B, soit construit comme image par une règle de certaines de ses prédécesseurs dans la suite finie.

En pratique, on considère des règles avec paramètres pour se simplifier la vie (cf la suite).

5.2.2 induction structurelle

Le principal intérêt de E est qu'il permet des preuves par **induction structurelle**. Soit une propriété P définie sur U. Si

1. pour tout $e \in B$, $P(e)$ est vraie
2. pour toute règle f, si f est d'arité k et si le k-uplet $(e_1, \ldots, e_k)$ d'éléments de U vérifie $P(e_i)$ est vraie pour tout i, alors $P(f(e_1, \ldots, e_k))$ est vraie

alors P est vraie sur E tout entier.

De ce fait, on peut mettre en place des définitions par récurrence structurelle et prouver des propriétés par induction structurelle. On admet en particulier que pour définir une fonction de E^k dans un ensemble quelconque, il suffit de la définir sur la base et sur les images d'éléments de E par chacun des règles.

5.3 Exemple des entiers

5.3.1 formalisation

On prend $U = \mathbb{R}$, $B = \{0\}$ et comme unique règle s d'arité un définie par $s(n) = n + 1$. On montre alors que ce schéma d'induction engendre $E = \mathbb{N}$. On constate en effet que $\mathbb{N}$ contient la base et est stable par la règle. On montre ensuite facilement par récurrence sur la longueur de la construction des éléments de E qu'ils sont tous dans $\mathbb{N}$.

5.3.2 utilisation

Considérons la fonction suivante f définie sur $E \times E$ par récurrence structurelle :
- $f(0, e) = e$ pour tout $e \in E$
- $f(s(e_1), e_2) = s(f(e_1, e_2))$

On veut montrer que s coïncide en fait avec la somme sur les entiers, c'est-à-dire la propriété $f(a, b) = a + b$. On constate que c'est vrai pour $f(0, e)$. On doit alors montrer que la propriété est stable par la règle s. On montre d'abord que pour tout e fixé, et tout n, $f(n, e) = n + e$, par récurrence induction structurelle sur n. Soit donc $n = s(p)$ avec $f(p, e) = p + e$. Alors $f(n, e) = s(f(p, e)) = s(p + e) = p + e + 1 = n + e$ (en utilisant dans l'ordre l'hypothèse inductive, puis la définition de s deux fois). Comme $f(0, e) = e = 0 + e$, on a bien que pour tout $n \in E$, $f(n, e) = n + e$ (par induction structurelle). Comme la preuve est vraie pour tout e, on obtient bien ce qu'on voulait...

5.4 Exemple des listes

5.4.1 formalisation

On considère les listes d'entiers (on évite soigneusement la définition de U) :
- $[]$ est la liste vide (base)
- on a une règle paramétrique :: telle que si k est un entier (le paramètre) et l une liste, alors $k :: l$ est une liste

L'interprétation intuitive est que k est mis au début de la liste.

5.4.2 opérations

On peut définir la longueur par induction de la façon suivante :
- $|[]| = 0$
- $|[k :: l]| = 1 + |l|$

De même on définit la concaténation de deux listes par :
- $[] + l = l$
- $|[k :: l] + p| = k :: (l + p)$

5.4.3 propriété

On montre que la longueur de la concaténation est la somme des longueurs, par induction structurelle.
- on fixe l et on $|[] + l| = |l| = |[]| + |l|$.
- on suppose donné p telle que $|p + l| = |l| = |p| + |l|$, on a alors $|[k :: p] + l| = |k :: (p + l)| = 1 + |p + l| = 1 + |p| + |l| = |[k :: p]| + |l|$, ce qui montre la stabilité de la propriété relativement à la règle de construction.

On en déduit par induction structurelle que pour tout l et tout p, $|p + l| = |p| + |l|$.

5.4.4 exercice

Définir par induction le miroir d'une liste et montrer que la longueur est préservée par cette opération.

5.5 Exemple des arbres binaires

5.5.1 formalisation

On considère les arbres binaires sur les entiers :
- $\emptyset$ comme l'arbre vide
- la règle de construction paramétrique T telle que si k est un entier, d et g deux arbres, alors $T(k, g, d)$ est un arbre.

5.5.2 opérations

Définir par induction :
- la hauteur de l'arbre
- son nombre de noeuds
- son nombre de feuilles

Montrer des liens entre les quantités ainsi définies.

6 Automates