Algorithmique et Analyse d'Algorithmes

L3 Info

Cours 4 : Logique de Hoare

Benjamin Wack

2021 – 2022

La dernière fois

- Invariant, précondition, postcondition
- Terminaison d'un algorithme
- Drapeau Hollandais

Aujourd'hui

- Système formel
- Logique de Hoare
- Dichotomie

Motivations

La méthode des invariants fonctionne mais ne passe pas à l'échelle :

- L'invariant doit être « deviné ».
- La précondition aussi pose parfois problème.
- Et surtout la démonstration elle-même n'est pas automatisable.

Solution proposée :

- un système de déduction **formel**
- annotation du programme **en partant de la fin**

- Progrès vers l'automatisation
- Pas totale cependant : la preuve d'algorithme est un problème indécidable
- Possibilité d'utiliser des *assistants de démonstration* (Why3 par exemple)

Plan

La logique de Hoare (1969)
- Formalisation du langage
- Sémantique, système de déduction
- Les règles de Hoare

Algorithme classique : recherche dichotomique

Le langage de programmation

Afin de pouvoir raisonner formellement, on fixe une syntaxe restreinte sur le langage de « programmation » utilisé :

- expressions E
  constituées de variables, de constantes et d'opérations ($+$, $*$, ...)
- expressions booléennes B
  constituées de comparaisons et de connecteurs (`and`, `or`, ...)
- une instruction I peut être :
  - une affectation `x` $:= E$
  - une séquence I_1 ; I_2
  - une conditionnelle `if` B `then` I_1 `else` I_2
  - une boucle `while` B `do` I

On peut alors raisonner **par cas** et **par induction** sur la forme du programme considéré.

- Pas de *for* (utiliser *while*)
- Pas de structures de données (mais possibilité d'étendre le langage)
- Pas d'appel de fonction (et donc pas de récursivité)

Le langage des propriétés

Les propriétés que nous exprimons à propos des données et des résultats sont généralement des **formules de logique du premier ordre** :

- connecteurs logiques *non*, *et*, *ou*, $\Rightarrow$
- quantificateurs $\forall$, $\exists$
- opérations et prédicats usuels sur les données ($+$, $*$, $<$, $=$...)

La plupart des variables sont partagées par le programme et les propriétés, certaines ne sont utilisées que dans les propriétés.

Attention

Une propriété des variables peut être vraie ou fausse à un point donné de l'exécution d'un programme, et selon les données initiales.

Ne pas confondre

expression booléenne évaluée dans une exécution du programme

propriété utilisée dans la démonstration

Ne pas confondre

assertion utilisée dans le test de programme, évaluée systématiquement et qui lève une exception si elle est fausse

propriétés (parfois appelées assertions !) sans se prononcer *a priori* sur leur valeur de vérité

Triplet de Hoare

Tony Hoare (1934-)

- Tri rapide
- Compilateur pour ALGOL-60
- Spécification formelle de langages (y compris multi-processus)
- Turing Award 1980

On travaille sur le langage minimal défini la semaine dernière (affectation, séquence, conditionnelle, boucle `while` + quelques opérations).

Triplet de Hoare

Un triplet de Hoare est de la forme $\{P\}$ `I` $\{Q\}$ où :

- P et Q sont des formules logiques comportant éventuellement des variables de `I`
- `I` est une instruction de notre langage de programmation

Nota Bene : dans sa présentation originale, Hoare écrivait plutôt P {`I`} Q, ce qu'on retrouve dans certains documents.

L'intention derrière le système

Le triplet $\{P\}$ `I` $\{Q\}$ exprime la propriété suivante :

> Si P (précondition) est vérifiée avant l'exécution de I,
> alors Q (postcondition) est vérifiée après son exécution.

Quelques triplets

$\{true\}$ `x := 3` $\{x = 3\}$

$\{x > 0\}$ `x := x + 1` $\{x > 0\}$

$\{x > 0\}$ `x := x - 1` $\{x < 2\}$

$\{x = n\}$ `x := x + 1` $\{x = n + 1\}$

$\{true\}$ `if x<0 then x := -x else skip` $\{x > 0\}$

Un tel triplet peut être valide ou non.

Validité et démonstration

Le triplet $\{P\}$ `I` $\{Q\}$ est dit valide si :

partant de **tout état mémoire** qui vérifie P,
`I` produit un état mémoire dans lequel Q est vérifié.

Problème : il y a une infinité d'états mémoire à considérer.

On construit donc les triplets de Hoare à l'aide d'un système de déduction, qui ne se base **que** sur la structure de `I`.

Théorème de correction de la logique de Hoare

Si

on peut déduire un triplet $\{P\}$ `I` $\{Q\}$ dans le système de Hoare

alors

pour tout état mémoire qui vérifie P, l'exécution de `I` produit un état mémoire dans lequel Q est vérifié.

(La démonstration sort du cadre de ce cours.)

Système formel de déduction

Une règle est constituée :

- d'**hypothèses** ou **prémisses** $H_1, \ldots, H_n$
- d'une **conclusion**
- éventuellement on étiquette une règle par son nom

$$\frac{H_1 \ldots H_n}{C}\ R$$

Exemple : Introduction de la conjonction $\frac{A \quad B}{A \ et \ B}$

Système de déduction = ensemble fini, fixé de règles

Preuve dans un système formel

Un arbre de preuve est formé de plusieurs règles, les conclusions des unes formant les prémisses des autres :

$$\frac{H_1 \qquad \cdots \qquad H_n}{A}$$

prouve A sous les hypothèses $H_1, \ldots, H_n$.

- Un **axiome** est une règle sans prémisse.
- Un **arbre complet** est un arbre de preuve dont toutes les feuilles sont des axiomes.

Démontrer A dans le système choisi, c'est exhiber un arbre complet dont la conclusion est A.

Une règle simple : la séquence

$$\frac{\{P\}\ \mathtt{I}\ \{Q\} \qquad \{Q\}\ \mathtt{J}\ \{R\}}{\{P\}\ \mathtt{I}\ ;\ \mathtt{J}\ \{R\}}$$

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \{\mathtt{x} > 0\}\ \mathtt{x := x + 1}\ \{\mathtt{x} > 1\} & \{\mathtt{x} > 1\}\ \mathtt{y := 2 * x}\ \{\mathtt{y} > 2\}\end{array}}{\{\mathtt{x} > 0\}\ \mathtt{x := x + 1\ ;\ y := 2 * x}\ \{\mathtt{y} > 2\}}$$

Attention

L'arbre de preuve ainsi construit n'est pas encore complet.

Autre règle simple : la conditionnelle

$$\frac{\{P \text{ et } C\}\ \mathtt{I}\ \{Q\} \qquad \{P \text{ et non } C\}\ \mathtt{J}\ \{Q\}}{\{P\}\ \mathtt{if\ C\ then\ I\ else\ J}\ \{Q\}}$$

$$\frac{\{\mathtt{y} \geq 0\}\ \mathtt{x := y}\ \{\mathtt{x} \geq 0\} \qquad \{\mathtt{y} < 0\}\ \mathtt{x := 0}\ \{\mathtt{x} \geq 0\}}{\{\mathit{true}\}\ \mathtt{if\ y} \geq \mathtt{0\ then\ x := y\ else\ x := 0}\ \{\mathtt{x} \geq 0\}}$$

Remarque : on prouve la **même** chose sous les **mêmes** hypothèses (excepté la condition booléenne) : pas réaliste en général.

L'affectation : ça se complique

Quelle règle pour `a := b + x` ?

- postcondition $\{\mathtt{a} = \mathtt{b} + \mathtt{x}\}$:
  oui mais ne tient pas compte d'une éventuelle précondition
- si la précondition est $\{\mathtt{z} > 0\}$: **conserver** cette information
- si la précondition est $\{\mathtt{a} = 3\}$: **oublier** cette information
- si la précondition est $\{\mathtt{b} = 0\}$: on voudrait en **déduire** $\mathtt{a} = \mathtt{x}$

Plus compliqué : `x := x + 2`

- postcondition $\{\mathtt{x} = \mathtt{x} + 2\}$: **n'a pas de sens**
- si la précondition est $\{\mathtt{x} = 4\}$: on voudrait **déduire** $\{\mathtt{x} = 6\}$
- si la précondition est $\{\mathtt{x} > 0\}$: on voudrait **déduire** $\{\mathtt{x} > 2\}$

Vers une règle pour l'affectation

Ce qu'on voudrait

- Établir un lien **logique** entre la précondition et la postcondition
- **Exploiter** le changement de valeur de la variable `x` affectée

Idée :

- Utiliser une même propriété P en précondition et en postcondition
- Substituer `x` par sa nouvelle valeur

~~$\{P\}$ `x := E` $\{P[x \leftarrow E]\}$~~ : **Cette règle est incorrecte**

Essai sur un exemple

- Soit l'instruction `x := x - 1`
- Soit la précondition $P : x \geq 0$

Alors cette règle donne $\{x \geq 0\}$ `x := x - 1` $\{x \geq 1\}$

Règle de l'affectation (axiome)

$$\frac{}{\{Q[\mathtt{x} \leftarrow \mathtt{E}]\}\ \mathtt{x}\ \mathtt{:=}\ \mathtt{E}\ \{Q\}}$$

Raisonnement arrière : pour que Q soit vraie après cette affectation, il faut qu'elle soit déjà vraie pour la valeur que va prendre $\mathtt{x}$.

$$\frac{}{\{\mathtt{b} + \mathtt{x} = \mathtt{x}\}\ \mathtt{a}\ \mathtt{:=}\ \mathtt{b}\ \mathtt{+}\ \mathtt{x}\ \{\mathtt{a} = \mathtt{x}\}}$$

$$\frac{}{\{\mathtt{x} + 2 > 2\}\ \mathtt{x}\ \mathtt{:=}\ \mathtt{x}\ \mathtt{+}\ 2\ \{\mathtt{x} > 2\}}$$

Deux remarques

- Il faut pouvoir manipuler les (in)égalités.
- Ne marche que si une variable est toujours manipulée explicitement par son nom : pas d'aliasing, pas de pointeurs, pas d'effets de bord.

Exemple : échange de deux variables

$$\{\mathtt{x} = a \ et \ \mathtt{y} = b\} \ \ \mathtt{t} := \mathtt{x} \ ; \ \mathtt{x} := \mathtt{y} \ ; \ \mathtt{y} := \mathtt{t} \ \ \{\mathtt{x} = b \ et \ \mathtt{y} = a\}$$

$$\frac{\dfrac{\{\mathtt{y} = b \ et \ \mathtt{x} = a\}\mathtt{t} := \mathtt{x}\{\mathtt{y} = b \ et \ \mathtt{t} = a\} \quad \{\mathtt{y} = b \ et \ \mathtt{t} = a\}\mathtt{x} := \mathtt{y}\{\mathtt{x} = b \ et \ \mathtt{t} = a\}}{\{\mathtt{x} = a \ et \ \mathtt{y} = b\}\mathtt{t} := \mathtt{x} \ ; \ \mathtt{x} := \mathtt{y}\{\mathtt{x} = b \ et \ \mathtt{t} = a\}} \quad \{\mathtt{x} = b \ et \ \mathtt{t} = a\}\mathtt{y} := \mathtt{t}\{\mathtt{x} = b \ et \ \mathtt{y} = a\}}{\{\mathtt{x} = a \ et \ \mathtt{y} = b\}\mathtt{t} := \mathtt{x} \ ; \ \mathtt{x} := \mathtt{y} \ ; \ \mathtt{y} := \mathtt{t}\{\mathtt{x} = b \ et \ \mathtt{y} = a\}}$$

Règle du `while`

$$\frac{\{P \text{ et } C\}\ \texttt{I}\ \{P\}}{\{P\}\ \texttt{while C do I}\ \{P \text{ et non } C\}}$$

(P est l'invariant de la boucle)

$$\frac{\dfrac{\dfrac{}{\{\texttt{x} < \texttt{b}\}\ \texttt{x := x + 1}\ \{\texttt{x} \leq \texttt{b}\}}}{\{\texttt{x} \leq \texttt{b} \text{ et } \texttt{x} < \texttt{b}\}\ \texttt{x := x + 1}\ \{\texttt{x} \leq \texttt{b}\}}}{\{\texttt{x} \leq \texttt{b}\}\ \texttt{while x < b do x := x + 1}\ \{\texttt{x} \leq \texttt{b} \text{ et } \texttt{x} \geq \texttt{b}\}}$$

$$\{\texttt{x} \leq \texttt{b}\}\ \texttt{while x < b do x := x + 1}\ \{\texttt{x} = b\}$$

Règles logiques

Soit à démontrer :

$$\frac{\dfrac{}{\{\mathtt{y} \geq 0\}\ \mathtt{x := y}\ \{\mathtt{x} \geq 0\}}\ (\mathit{axiome}) \qquad \dfrac{}{\{\mathtt{y} < 0\}\ \mathtt{x := 1}\ \{\mathtt{x} \geq 0\}}\ ???}{\{\mathit{true}\}\ \mathtt{if\ y} \geq \mathtt{0\ then\ x := y\ else\ x := 1}\ \{\mathtt{x} \geq 0\}}$$

Renforcement de la précondition

$$\frac{P \Rightarrow P' \qquad \{P'\}\ \mathtt{I}\ \{Q\}}{\{P\}\ \mathtt{I}\ \{Q\}}$$

$$\frac{\mathtt{y} < 0 \Rightarrow 1 \geq 0 \qquad \{1 \geq 0\}\ \mathtt{x := 1}\ \{\mathtt{x} \geq 0\}}{\{\mathtt{y} < 0\}\ \mathtt{x := 1}\ \{\mathtt{x} \geq 0\}}$$

Et réciproquement...

Affaiblissement de la postcondition

$$\frac{\{P\}\ \mathtt{I}\ \{Q\} \qquad Q \Rightarrow Q'}{\{P\}\ \mathtt{I}\ \{Q'\}}$$

Si $A \Rightarrow B$ la condition A est dite plus forte que B.
la condition B est plus faible que A.

On cherchera donc à prouver des triplets avec préconditions faibles et postconditions fortes (ensuite relâchées par les règles logiques).

Ces implications sont **à prouver** :

- à la main
- par d'autres règles d'inférence (déduction naturelle...)
- à l'aide d'un assistant de démonstration (logiciel)

En pratique

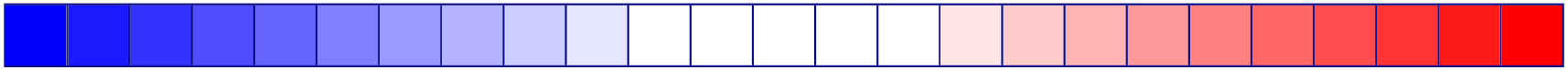
Le système de Hoare est complexe à utiliser :

- règles lourdes à appliquer
- taille des arbres de preuve
- mieux adapté à une preuve automatisée

On en retiendra cependant quelques idées pour les « petites » preuves :

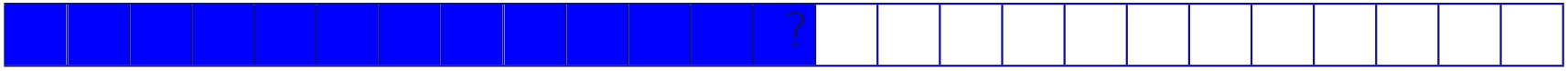
- Décomposer le programme par induction structurelle
- Annoter chaque étape du programme avec une propriété
- Remonter de la postcondition vers la précondition
- Exploiter *non* C en sortie du while

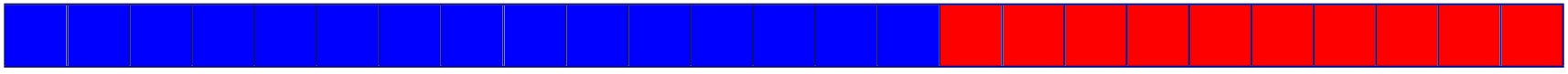
Problème et intuition



Où est la frontière ?

Littéralement, *dichotomie* = « couper en deux »





Bleu/rouge = par exemple inférieur ou supérieur à un élément recherché, **mais pas seulement**

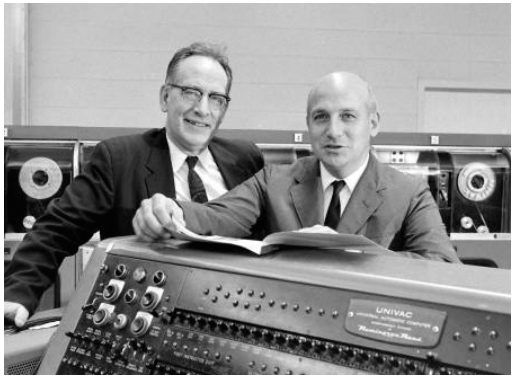
Algorithme

DICHOTOMIE(T, v)

Données : Un tableau T indexé de 1 à N
rangé en ordre croissant
Une valeur v

Résultat : Un indice r tel que T[r] = v
r = -1 s'il n'en existe pas

```
r := -1
g := 1
d := N
while r < 0  &&  g ≤ d
    m := (g + d) / 2
    if t[m] < v
        g := m + 1
    else if t[m] > v
        d := m - 1
    else
        r := m
```



J. Mauchly
(1907-1980)

Démontrons :

- Correction
- Complétude

Correction :

Si on trouve un élément alors il s'agit de v.

```
r := -1                                   { −1 < 0 : rien à démontrer }
g := 1
d := N
while r < 0  &&  g ≤ d                    { invariant : Si r ≥ 0 alors t[r] = v }
    m := (g + d) / 2
    if t[m] < v                           (r ne change pas)
        g := m + 1
    else if t[m] > v                      (r ne change pas)
        d := m - 1
    else                                  { Si m ≥ 0 alors t[m] = v }
        r := m
                                          { Si r' ≥ 0 alors t[r'] = v }
                                          { Si r ≥ 0 alors t[r] = v }
```

Ici l'invariant est identique à la postcondition.

Complétude :

Si v est présent dans le tableau alors on le trouve. Contraposée : Si on ne trouve pas v alors il est absent du tableau.

Précondition : $\forall i, j, \quad i \leq j \Rightarrow t[i] \leq t[j]$ $\underline{\forall i, j, \quad i \leq j \Rightarrow t[i] \leq t[j]}$

```
r := -1                                   {∀i < 1, t[i] < v et ∀i > N, t[i] > v}
g := 1
d := N
while r < 0 && g ≤ d        {inv : ∀i < g, t[i] < v et ∀i > d, t[i] > v}
    m := (g + d) / 2
    if t[m] < v  t[m] < v
                                {∀i < m + 1, t[i] < v}      (d ne change pas)
        g := m + 1
    else if t[m] > v                                        {idem pour d}
        d := m - 1
    else                                                    {g et d inchangés}
        r := m
                          {∀i < g', t[i] < v et ∀i > d', t[i] > v}
{invariant et (r ≥ 0 ou g > d)}  ⇒  { Si r < 0 alors ∀i, t[i] ≠ v }
```


Ici l'invariant est une spécialisation de la postcondition.

Complexité

Soit k l'entier (unique) tel que $2^{k-1} < N \leq 2^k$.

- Initialement $d - g = N - 1 \leq 2^k$
- À chaque itération $d' - g' \leq \frac{d-g}{2}$
- Donc à l'itération i on a $d - g \leq \frac{2^k}{2^i}$
- Et lorsque $d - g \leq 1$ il reste au pire 2 itérations

Comme chaque itération s'effectue en temps constant, le coût de l'algorithme est de l'ordre de $k = \log_2(N)$.

En pratique

Pour les systèmes critiques

Démontrer qu'une spécification est réalisable
Écrire un programme qui la respecte
} sont les mêmes activités.

⇒ **Extraction de programme fonctionnel dans les assistants de démonstration** (par exemple CompCert)

Pour la programmation « au quotidien »

- Toujours écrire ses boucles en ayant en tête un invariant
- et un variant

« Recettes » pour construire un invariant

- Prendre (une partie de) la postcondition
- Spécialiser la postcondition (avec une des variables du programme)
- Renforcer l'invariant si nécessaire

Test ou preuve ?

"Testing shows the presence, not the absence of bugs"
E. W. Dijkstra

Le test :

- valide une **implantation** plutôt qu'un algorithme
- permet rapidement d'éliminer des « bugs »
- peut être utilisé en cours de développement
- fait apparaître les limites du modèle

La preuve :

- fournit une **garantie** incontestable sur le fond de l'algorithme
- mais n'élimine pas (complètement) les erreurs de programmation
- nécessite des outils formels pour une utilisation à grande échelle

En résumé

Aujourd'hui

- Un **système formel** permet d'automatiser un(e partie du) raisonnement
- La **logique de Hoare** permet de démontrer qu'un programme vérifie une spécification
- On part des postconditions et on cherche les préconditions adaptées
- Programmation par contrat : preuve et programmation simultanées

La prochaine fois

- Type abstrait
- Pile, file, file à priorité
- Expression bien parenthésée