Avant de commencer : attention semaines prochaines

Échange de cours

- Cours d'ALM jeudi 5 octobre de 9h45 à 11h15 en amphi 022
- Cours d'algo lundi 9 octobre de 8h30 à 10h en amphi 022

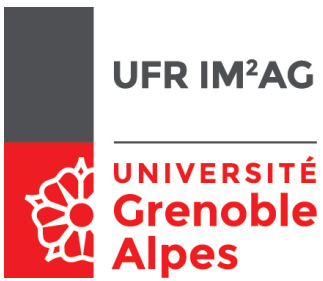
Autrement dit :
deux cours d'ALM la semaine prochaine
deux cours d'algo la semaine suivante

Algorithmique et Analyse d'Algorithmes

L3 Info

Cours 4 : Logique de Hoare

Benjamin Wack



2017- 2018

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*

Plan

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.

Sémantique et validité

Pour décider si un triplet est valide, il faut définir formellement ce qu'est une exécution de `I` :

Sémantique opérationnelle d'une instruction

C'est l'effet de cette instruction sur un **état mémoire**.
Elle est définie par une règle pour chaque construction du langage,
par exemple $E \xrightarrow{I_1 ; I_2} G$ si $E \xrightarrow{I_1} F$ et $F \xrightarrow{I_2} G$.

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

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

Système formel de déduction

Démontrer chaque triplet en référence à la sémantique opérationnelle est infaisable en pratique : on recourt à un système de déduction formel.

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 : Conjonction en déduction naturelle $\frac{A \quad B}{A \wedge B}\ (\wedge I)$

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 :

$$\dfrac{\dfrac{}{H_1} \quad \cdots \quad \dfrac{}{H_n}}{A}$$

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.

Légitimité du système

Un tel système de déduction est un pur jeu d'écriture.
Il n'est intéressant que s'il permet de démontrer des choses « vraies ».

Théorème

La logique de Hoare est correcte vis-à-vis de la sémantique opérationnelle.

Autrement dit :

Si

on peut démontrer 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.)

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

$$\frac{\{P \wedge C\}\ \mathtt{I}\ \{Q\} \qquad \{P \wedge \neg C\}\ \mathtt{J}\ \{Q\}}{\{P\}\ \mathtt{if\ C\ then\ I\ else\ J}\ \{Q\}}$$

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

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

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

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

Règle du `while`

$$\frac{\{P \wedge C\}\ \mathtt{I}\ \{P\}}{\{P\}\ \mathtt{while\ C\ do\ I}\ \{P \wedge \neg C\}}$$

$$\frac{\dfrac{\dfrac{}{\{\mathtt{x} < \mathtt{b}\}\ \mathtt{x := x + 1}\ \{\mathtt{x} \leq \mathtt{b}\}}}{\{\mathtt{x} \leq \mathtt{b} \wedge \mathtt{x} < \mathtt{b}\}\ \mathtt{x := x + 1}\ \{\mathtt{x} \leq \mathtt{b}\}}}{\{\mathtt{x} \leq \mathtt{b}\}\ \mathtt{while\ x < b\ do\ x := x + 1}\ \{\mathtt{x} \leq \mathtt{b} \wedge \mathtt{x} \geq \mathtt{b}\}}$$

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

Règles logiques

Soit à démontrer :

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

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

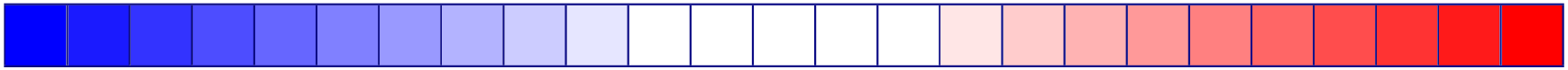
Le système de Hoare est complexe à utiliser :

- 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 :

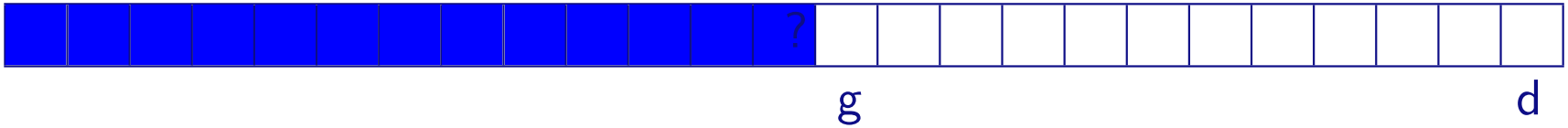
- Décomposer le programme par induction structurelle
- Annoter chaque étape du programme avec une propriété
- Remonter de la postcondition vers la précondition
- Exploiter $\neg C$ en sortie du while

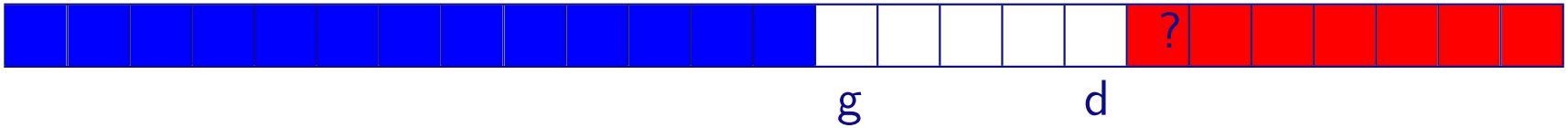
Problème et intuition

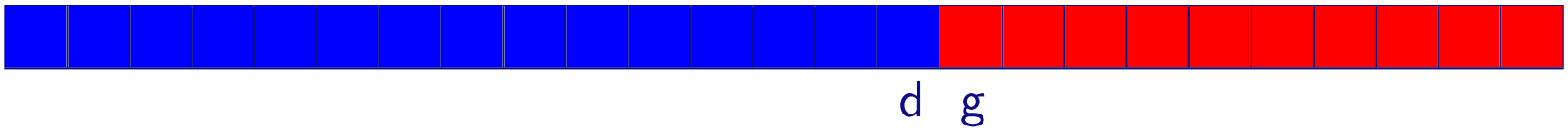


Où est la frontière ?

Littéralement, *dichotomie* = « couper en deux »







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

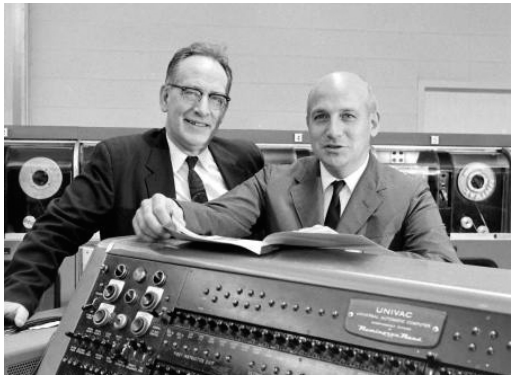
Algorithme

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)

Correction :

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

$\{-1 \geq 0 \Rightarrow t[-1] = v\}$

r := -1

g := 1

d := N

$\{invariant : r \geq 0 \Rightarrow t[r] = v\}$

while r < 0 && g ≤ d

$\{r \geq 0 \Rightarrow t[r] = v\}$

m := (g + d) / 2

$\{r \geq 0 \Rightarrow t[r] = v \wedge t[m] < v\}$

if *t[m] < v*

g := m + 1

$\{r \geq 0 \Rightarrow t[r] = v \wedge t[m] > v\}$

else if *t[m] > v*

d := m - 1

$\{m \geq 0 \Rightarrow t[m] = v \wedge t[m] = v\}$

else

r := m

$\{r \geq 0 \Rightarrow t[r] = v\}$

$\{r \geq 0 \Rightarrow t[r] = v\}$

Ici l'invariant est identique à la postcondition.

Complétude :

Si v est présent dans le tableau alors on le trouve.

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

```
r := -1                                   {∀i < 1, t[i] < v ∧ ∀i > N, t[i] > v}
g := 1
d := N
while r < 0 && g ≤ d    {inv : ∀i < g, t[i] < v ∧ ∀i > d, t[i] > v}
    m := (g + d) / 2
    if t[m] < v
        {t[m] < v ∧ ∀i < m + 1, t[i] < v ∧ ∀i > d, t[i] > v}
        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 ∧ ∀i > d, t[i] > v}
{invariant ∧ (r ≥ 0 ∨ g > d)}   ⇒   {r < 0 ⇒ ∀i, t[i] ≠ v}
```

Ici l'invariant est une généralisation de la postcondition.

Sûreté :

On ne dépasse pas les bornes du tableau.

Lemme : $\forall g, d, \quad g \leq d \;\Rightarrow\; g \leq (g+d)/2 \leq d$

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

Cas particulier : on cherche ici à ce que la propriété soit vraie à chaque itération.

Complexité

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

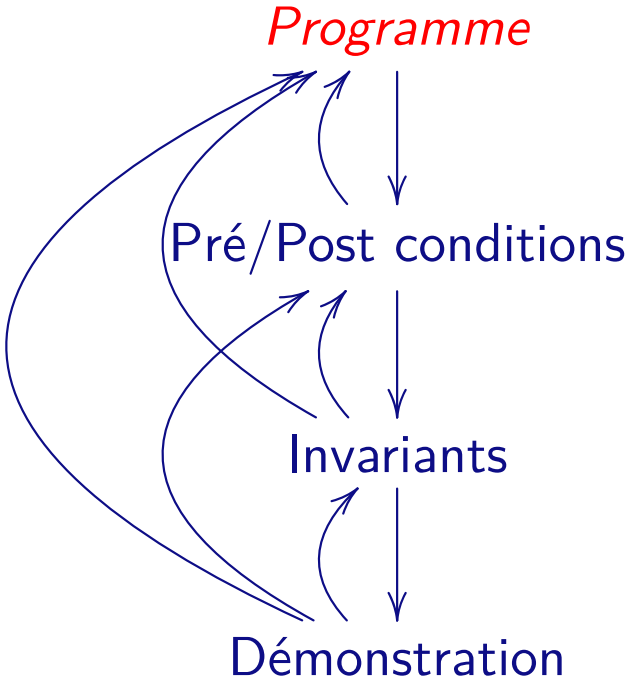
- Initialement $d - g = N - 1 \leq 2^k$
- À chaque itération $d' - g' \leq \frac{d-g}{2}$
- Donc à l'itération i on a $d - g \leq \frac{2^k}{2^i}$
- Et lorsque $d - g \leq 1$ il reste au pire 2 itérations

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

Programmation et preuve

Trois styles de développement :

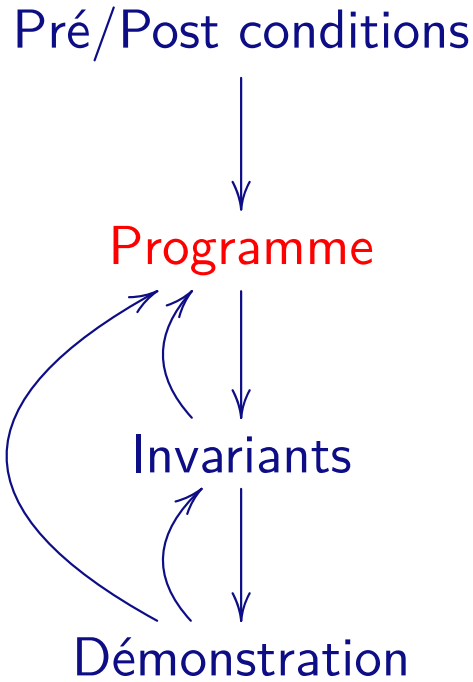
Preuve a posteriori



Programmation et preuve

Trois styles de développement :

Preuve constructive

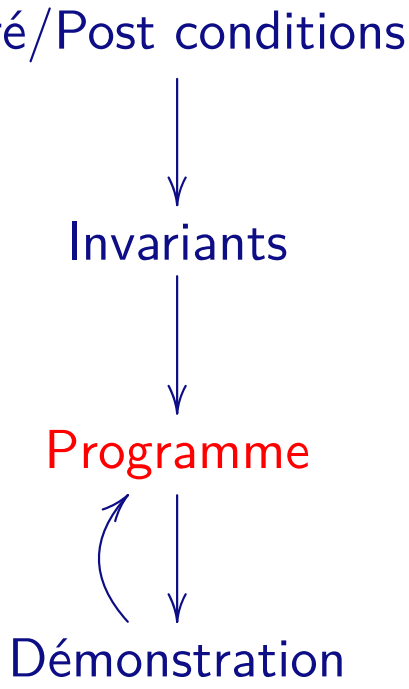


Dijkstra : *To prove the correctness of a given program, was in a sense putting the cart before the horse. A much more promising approach turned out to be letting correctness proof and program grow hand in hand : with the choice of the structure of the correctness proof one designs a program for which this proof is applicable.*

Programmation et preuve

Trois styles de développement :

Programmation par contrat



Dérivation de programme : le code est obtenu à partir de la spécification par l'application de règles systématiques

En pratique

Pour les systèmes critiques

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

$\Rightarrow$ **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
- Généraliser la postcondition (avec une des variables du programme)
- Renforcer l'invariant si nécessaire

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

Autres règles

Instruction `skip` :

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}$$

permet de construire une règle pour `if C then I` :

$$\frac{\{P \wedge C\}\ \texttt{I}\ \{Q\} \qquad \dfrac{\{P \wedge \neg C\}\ \texttt{skip}\ \{P \wedge \neg C\} \qquad P \wedge \neg C \Rightarrow Q}{\{P \wedge \neg C\}\ \texttt{skip}\ \{Q\}}}{\{P\}\ \texttt{if C then I else skip}\ \{Q\}}$$

$$\frac{\{P \wedge C\}\ \texttt{I}\ \{Q\} \qquad P \wedge \neg C \Rightarrow Q}{\{P\}\ \texttt{if C then I}\ \{Q\}}$$

Extensions possibles : pointeurs, appels de procédures, parallélisme...

Règle du `while` pour la correction totale

Un programme miraculeux

$$\dfrac{\dfrac{\overline{\mathit{true} \wedge \mathtt{x} \neq \sqrt{2} \Rightarrow \mathit{true}} \qquad \overline{\{\mathit{true}\}\ \mathtt{skip}\ \{\mathit{true}\}}}{\{\mathit{true} \wedge x \neq \sqrt{2}\}\ \mathtt{skip}\ \{\mathit{true}\}}}{\{\mathit{true}\}\ \mathtt{while\ (x\ !=}\ \sqrt{2}\mathtt{)\ do\ skip}\ \{\mathit{true} \wedge \mathtt{x} = \sqrt{2}\}}$$

Remplacez la condition par la propriété qui vous plaît : ce programme calcule ce qu'on veut... quand il termine.

$$\dfrac{\{P \wedge C \wedge \mathtt{V} = n\}\ \mathtt{I}\ \{P \wedge 0 \leq \mathtt{V} < n\}}{\{P\}\ \mathtt{while\ C\ do\ I}\ \{P \wedge \neg C\}}$$

Généralement notée avec d'autres crochets : $\langle P \rangle \mathtt{I} \langle Q \rangle$