Logiques de programmes: quand la machine raisonne sur ses logiciels

Introduction

Xavier Leroy

2021-03-04

Collège de France, chaire de sciences du logiciel
xavier.leroy@college-de-france.fr

Comment s’assurer qu’un logiciel fait ce qu’il est censé faire ?

Les méthodes classiques

Test

- Exécuter le programme sur des entrées bien choisies.
- Comparer les comportements observés aux comportements attendus.

Revues

- Relire attentivement le code, les tests, les documents de conception, …

Analyses

- Étudier mathématiquement certains aspects du programme : précision numérique, complexité en temps ou en espace, etc.
- Sur le papier ou assisté par des outils d'analyse statique.

Les limites du test

Testing shows the presence, not the absence of bugs.

(E. W. Dijkstra, 1969)

One ne teste qu'un petit nombre des comportements possibles du programme. Certains bugs se déclenchent très rarement !

Exemple (propagation de retenue dans les codes crypto)

Ajoute `2 * ta * tb` à `c2:c1:c0` en ≪optimisant≫ les retenues.

```
BN_UMULT_LOHI(t0,t1,ta,tb);
t2 = t1+t1; c2 += (t2<t1)?1:0;
t1 = t0+t0; t2 += (t1<t0)?1:0;
c0 += t1; t2 += (c0<t1)?1:0;
c1 += t2; c2 += (c1<t2)?1:0;
```

Les limites des revues de code

Given enough eyeballs, all bugs are shallow.

(Eric Raymond, 1999)

Fatigue, étourderie, distraction des relecteurs.

Certains codes sont moins relus que d'autres (p.ex. les *hot fixes*).

Exemple (le bug `goto fail`)

```
if ((err=SSLHashSHA1.update(&hashCtx,&signedParams)) != 0)
    goto fail;
    goto fail;
if ...
...
fail: return err;
```

Les limites des analyses de code

Beware of bugs in the above code;
I have only proved it correct, not tried it.
(Donald E. Knuth, 1977)

Risques d'erreurs dans l'analyse faite à la main et d'*unsoundness* dans les analyseurs statiques.

Décalage possible entre l'analyse et le ≪vrai≫ programme ou ses conditions d'exécution.

Exemple (Ariane 501)

Débordement dans conversion flottant 64 bits $\rightarrow$ entier 16 bits.

Une analyse menée dans le contexte d'Ariane 4 avait montré que cette quantité BH ≪tenait≫ dans 16 bits. Analyse invalide dans le contexte d'Ariane 5.

La vérification déductive (aussi appelée *program proof*)

Établir, par le raisonnement logique, des propriétés vraies de *toutes* les exécutions possibles du programme.

Au contraire d'autres ≪méthodes formelles≫, les propriétés établies vont jusqu'à la correction complète vis-à-vis de la spécification.

Intérêts pratiques :

- Obtenir des garanties plus fortes
  que celles atteignables par test et revues.
- Trouver des bugs introuvables autrement.

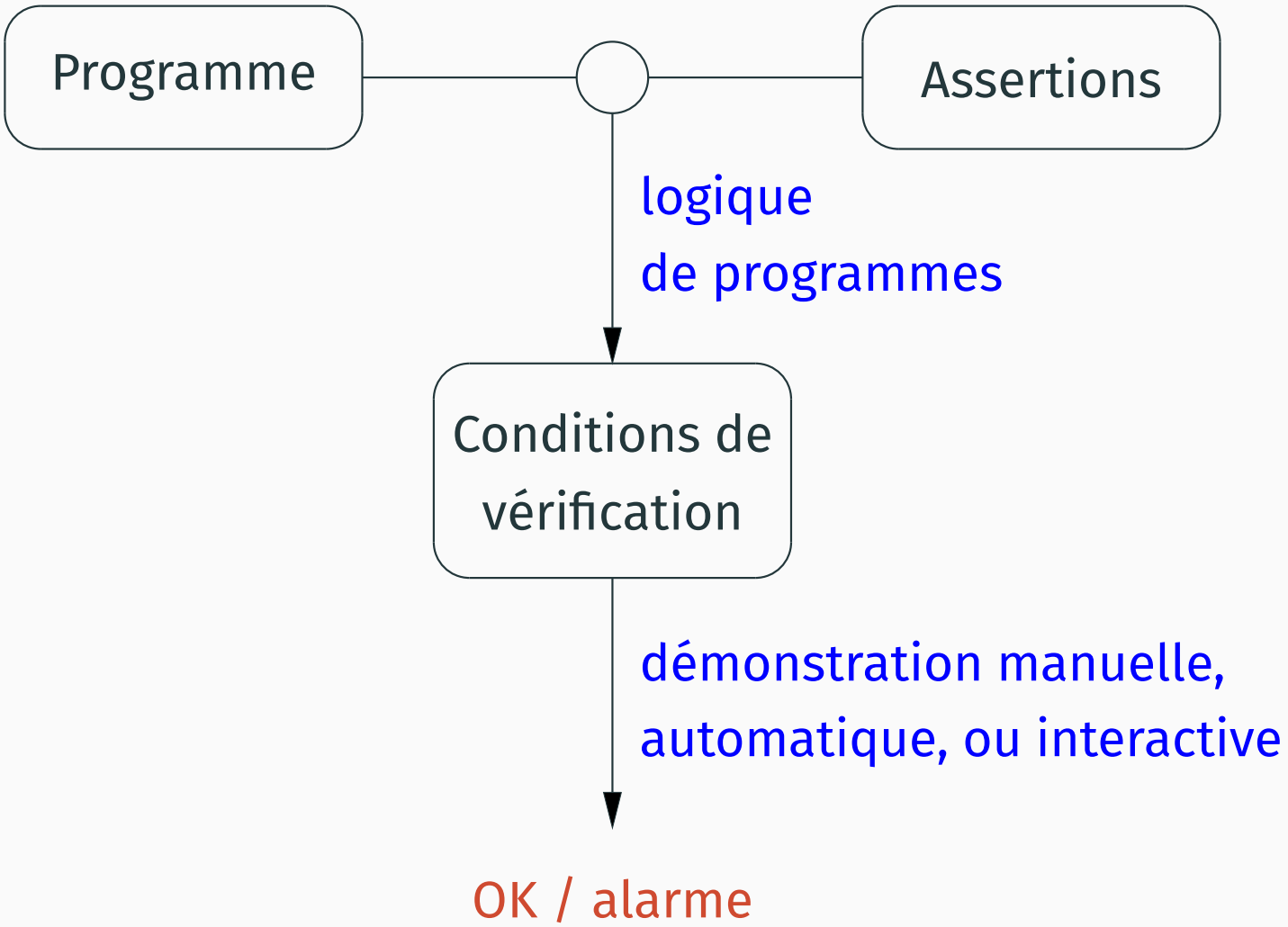
Les logiques de programmes

Une logique de programmes fournit un langage de spécification et des principes de raisonnement sur les comportements du programme.

Les spécifications se présentent généralement comme des assertions logiques portant sur le programme :

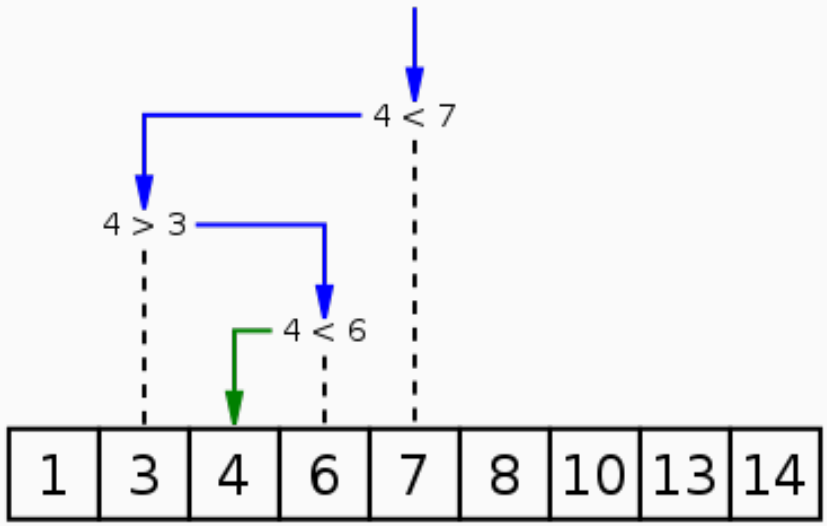
- préconditions : hypothèses sur les entrées
  (paramètres de fonction ; valeurs initiales des variables)
- postconditions : garanties sur les sorties
  (résultats de fonction ; valeurs finales des variables)
- invariants : garanties sur l'état en un point du code
  (invariants de boucles, de structures de données, …)

Logiques de programmes et vérification déductive



À la chasse au bug :
La recherche dichotomique

La recherche dichotomique




```
l = 0; h = a.length - 1;
while (l <= h) {
  m = (l + h) / 2;
  if (a[m] == v) return m;
  if (a[m] < v) h = m - 1; else l = m + 1;
}
return -1;
```

Une longue histoire

```
l = 0; h = a.length - 1;
while (l <= h) {
  m = (l + h) / 2;
  if (a[m] == v) return m;
  if (a[m] < v) h = m - 1; else l = m + 1;
}
return -1;
```

1946 John Mauchly, *Moore School Lectures*

1960 Derrick H. Lehmer publie l'algorithme moderne

1986 Jon Bentley, *Programming pearls*, chapitre 4

2004 Rapport de bug : `java.util.Arrays.binarySearch()` *will throw an* `ArrayIndexOutOfBoundsException` *if the array is large.*

2006 Joshua Bloch, *Nearly All Binary Searches and Mergesorts are Broken*.

La source du bug : un débordement arithmétique

```
m = (l + h) / 2;
```

On a $0 \leq$ `l` $\leq$ `h` $<$ `a.length`.

`l + h` peut déborder si `a.length` est suffisamment grand.

En Java, `l + h` est alors négatif, donc `m` aussi, et `a[m]` lève une exception «accès hors bornes».

En C, on a un «comportement indéfini» ou on continue avec `m` incorrect.

Un correctif simple : `m = l + (h - l) / 2;`

Pourquoi ce bug est difficile à trouver ?

Test :

- On teste rarement sur de très grosses données.
- Il faut une machine 64 bits et plusieurs Go de RAM pour faire apparaître ce bug.

Revues :

- La formule $(l + h)/2$ est familière.
- Risque d' ≪optimiser≫ $l + (h - l)/2$ en $(l + h)/2$.

Analyses :

- Une analyse d'intervalles de variation peut signaler l'erreur.

Démo

Vérification déductive de la recherche dichotomique avec l'outil Frama-C.

Le cours et le séminaire

Objectifs du cours

Comprendre les principes des logiques de programmes et les développements récents dans ce domaine.

Leitmotiv : quelles logiques pour quels traits des langages de programmation ?

(variables, pointeurs, parallélisme, ordre supérieur, etc)

Objectifs du séminaire

Illustrer l'utilisation de logiques de programmes dans des outils de vérification de qualité industrielle.

Montrer de nouveaux problèmes de vérification et de nouvelles idées pour les aborder.

La suite du cours

04/03	La naissance des logiques de programmes
11/03	Variables et boucles : la logique de Hoare
18/03	Pointeurs et structures de données : la logique de séparation
25/03	Parallélisme à mémoire partagée : la logique de séparation concurrente
01/04	Extensions de la logique de séparation : permissions fractionnaires, état fantôme, algèbres de ressources, …
08/04	Logiques pour la mémoire partagée faiblement cohérente
15/04	Logiques pour les langages fonctionnels et l'ordre supérieur

Le séminaire

11/03 Loïc Correnson (CEA).
Les logiques de programmes à l'épreuve du réel : tours et détours avec Frama-C/WP.

18/03 Yannick Moy (Adacore).
Preuve auto-active de programmes en SPARK

25/03 Bart Jacobs (K. U. Leuven).
VeriFast : Semi-automated modular verification of concurrent C and Java programs using separation logic

01/04 François Pottier (Inria).
Raisonner à propos du temps en logique de séparation

08/04 Jacques-Henri Jourdan (CNRS).
Protocoles personnalisés en logique de séparation : ressources fantômes et invariants dans la logique Iris

15/04 Philippa Gardner (Imperial College London).
Gillian : Compositional Symbolic Testing and Verification

Logiques de programmes, premier cours

Comment raisonner sur un logiciel?
La naissance des logiques de programmes

Xavier Leroy

2021-03-04

Collège de France, chaire de sciences du logiciel
`xavier.leroy@college-de-france.fr`

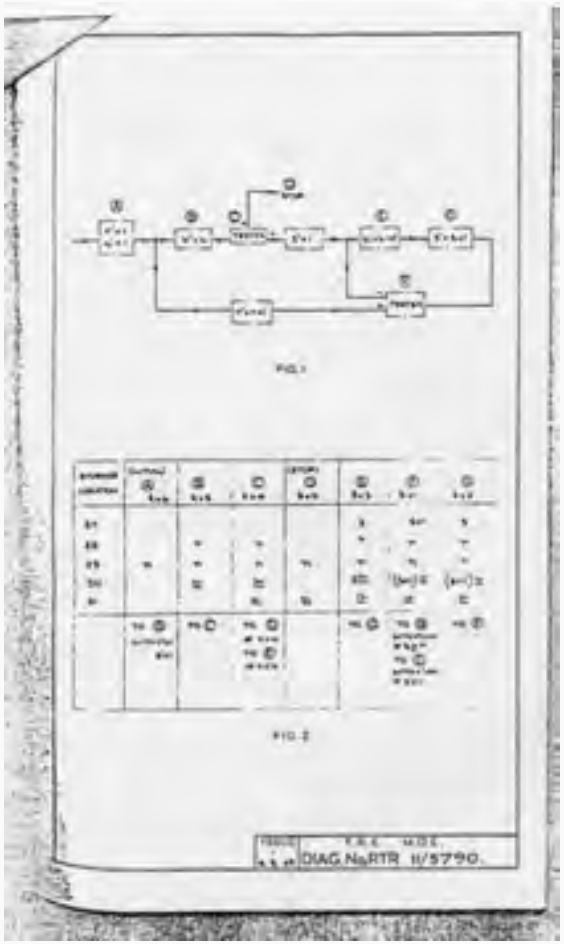
Plan du cours

Une revue de trois articles qui ont fondé le domaine :

- Alan Turing, *Checking a large routine*, 1949.
- Robert W. Floyd, *Assigning meanings to programs*, 1967.
- C. A. R. Hoare, *An axiomatic basis for computer programming*, 1969.

La découverte :
Checking a large routine
Alan Turing, 1949

Un article précurseur



Exposé à la *inaugural conference of the EDSAC computer*, Cambridge University, juin 1949. Le tapuscrit a été corrigé, commenté et republié par F.L. Morris et C.B. Jones dans *Annals of the History of Computing*, 6, 1984.

How can one check a routine? That is the question!

Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

Décomposer la vérification en étapes élémentaires

Consider the analogy of checking an addition. If it is given as:

$$\begin{array}{r} 1374 \\ 5906 \\ 6719 \\ 4337 \\ 7768 \\ \hline 26104 \end{array}$$

one must check the whole at one sitting, because of the carries. But if the totals for the various columns are given, as below:

$$\begin{array}{r} 1374 \\ 5906 \\ 6719 \\ 4337 \\ 7768 \\ \hline 3974 \\ 2213 \\ \hline 26104 \end{array}$$

the checker's work is much easier being split up into the checking of the various assertions $3 + 9 + 7 + 3 + 7 = 29$ etc. and the small addition

$$\begin{array}{r} 3974 \\ 2213 \\ \hline 26104 \end{array}$$

Le programme de Turing : calcul de la factorielle

Calculer $n!$ en n'utilisant que des additions.

Deux boucles imbriquées.

```
int fac (int n)
{
    int s, r, u, v;
    u = 1;
    for (r = 1; r < n; r++) {
        v = u; s = 1;
        do {
            u = u + v;
        } while (s++ < r);
    }
    return u;
}
```

L'organigramme du programme

≪Unfortunately there is no coding system sufficiently generally known to justify giving the routine for this process.≫

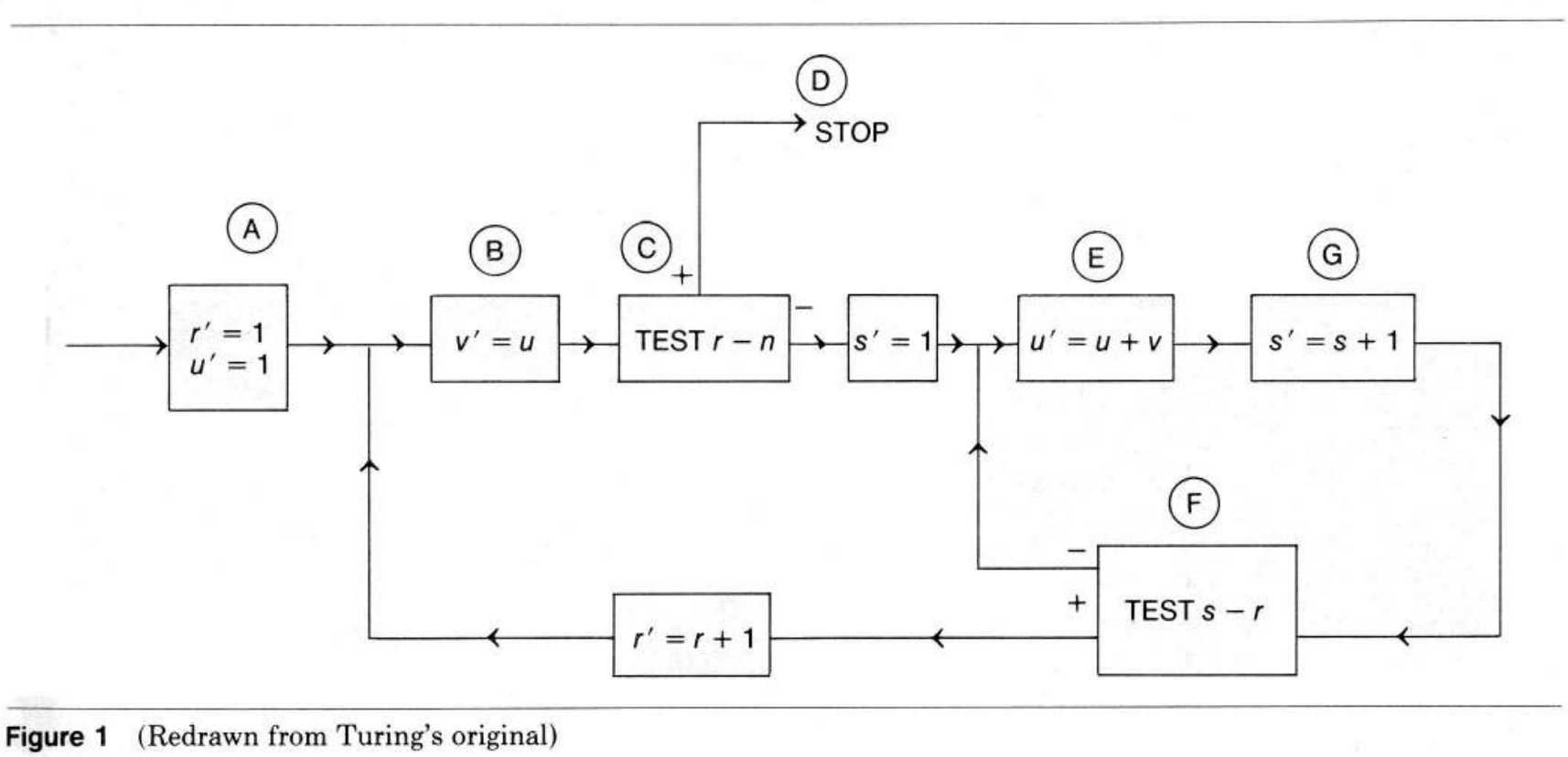


Figure 1 (Redrawn from Turing's original)

(La notation u/u' dénote la valeur de u avant/après le bloc).

Les assertions logiques

> ≪*In order to assist the checker, the programmer should make assertions about the various states that the machine can reach.*≫

Documente non seulement quelle case mémoire contient quelle variable abstraite, mais surtout les relations entre ces variables.

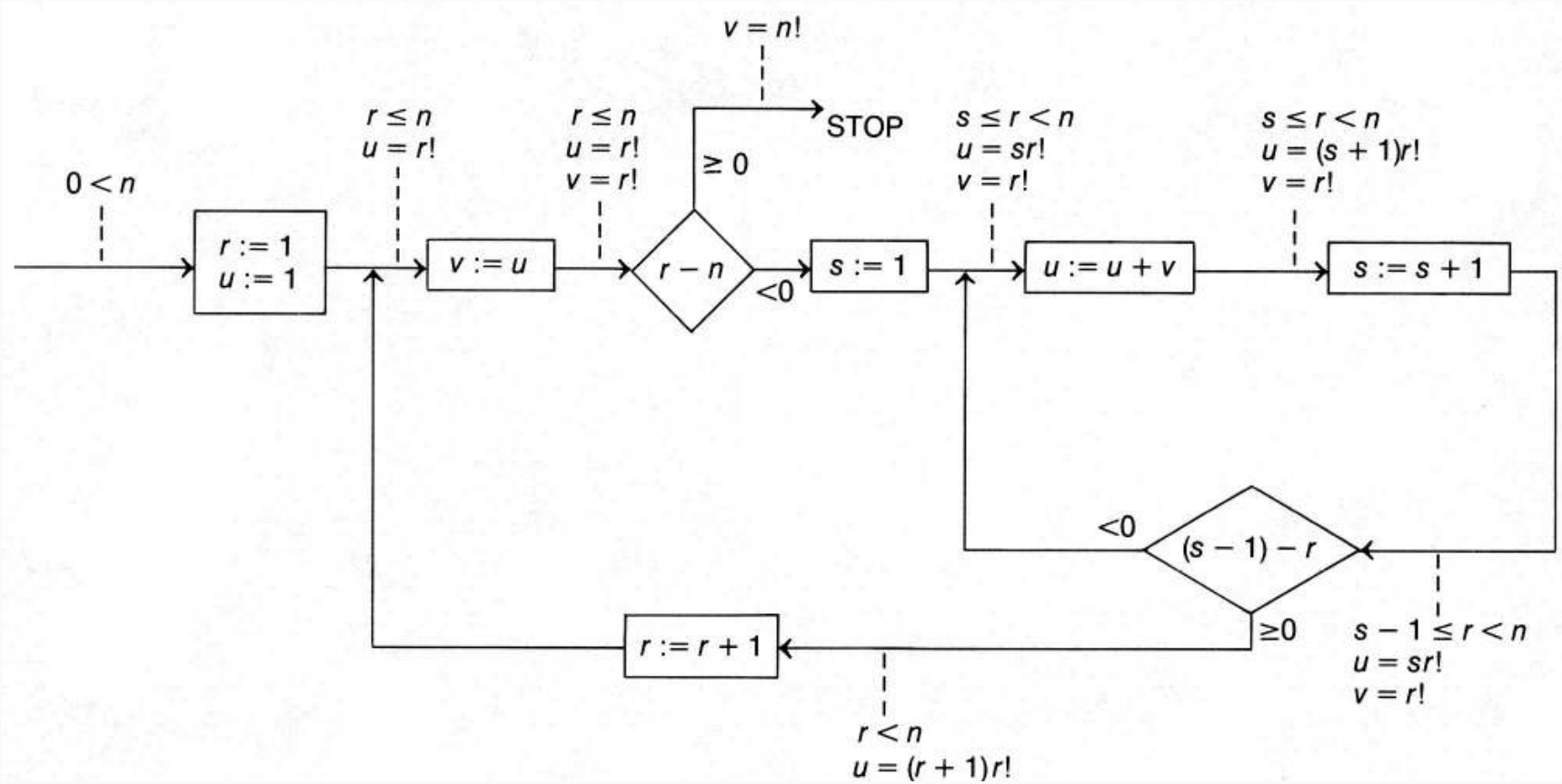
STORAGE LOCATION	(INITIAL) Ⓐ $k = 6$	Ⓑ $k = 5$	Ⓒ $k = 4$	(STOP) Ⓓ $k = 0$	Ⓔ $k = 3$	Ⓕ $k = 1$	Ⓖ $k = 2$
27 28 29 30 31	 n	r n $\lfloor r$	r n $\lfloor r$ $\lfloor r$	 n $\lfloor n$	s r n $s \lfloor r$ $\lfloor r$	$s+1$ r n $(s+1) \lfloor r$ $\lfloor r$	s r n $(s+1) \lfloor r$ $\lfloor r$
	TO Ⓑ WITH $r' = 1$ $u' = 1$	TO Ⓒ	TO Ⓓ IF $r = n$ TO Ⓔ IF $r < n$		TO Ⓖ	TO Ⓑ WITH $r' = r + 1$ IF $s \geq r$ TO Ⓔ WITH $s' = s + 1$ IF $s < r$	TO Ⓕ

Figure 2 (Redrawn from Turing's original)

(La notation $\lfloor n$ signifie ≪factorielle n≫.)

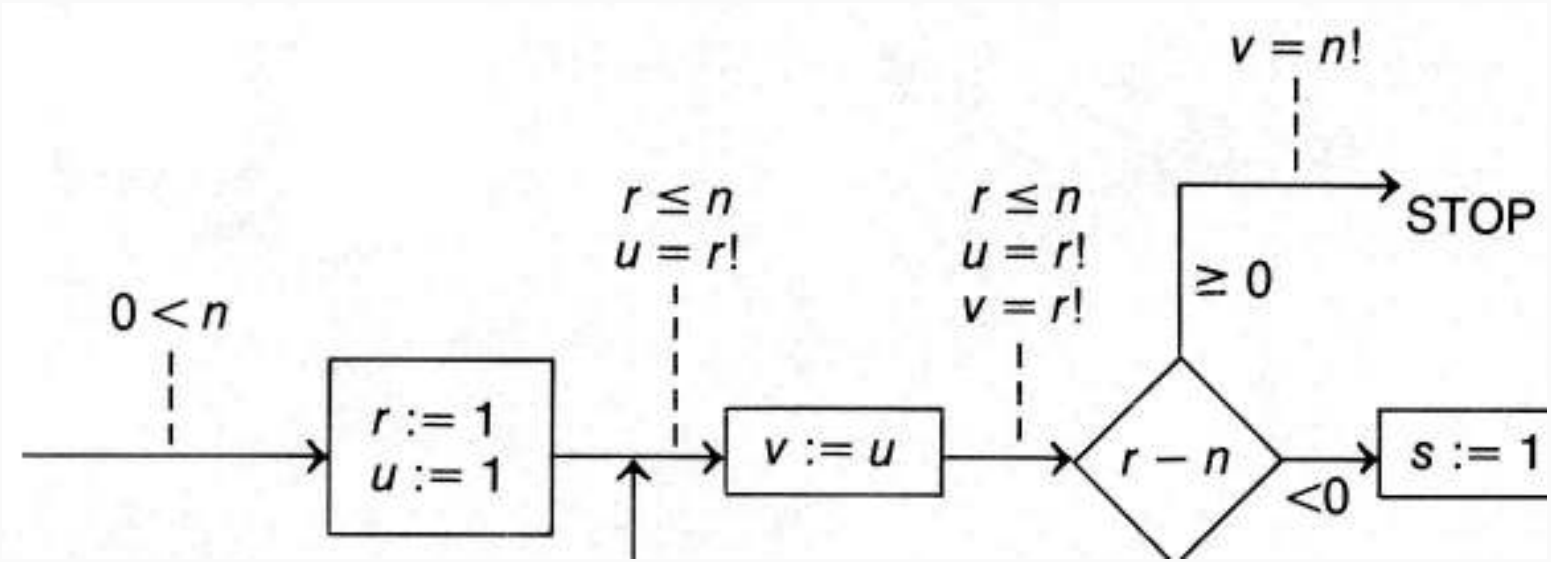
Les assertions logiques directement sur l'organigramme

Dans la notation moderne (introduite par Floyd en 1967), on note les assertions directement sur les arcs de l'organigramme.



La vérification

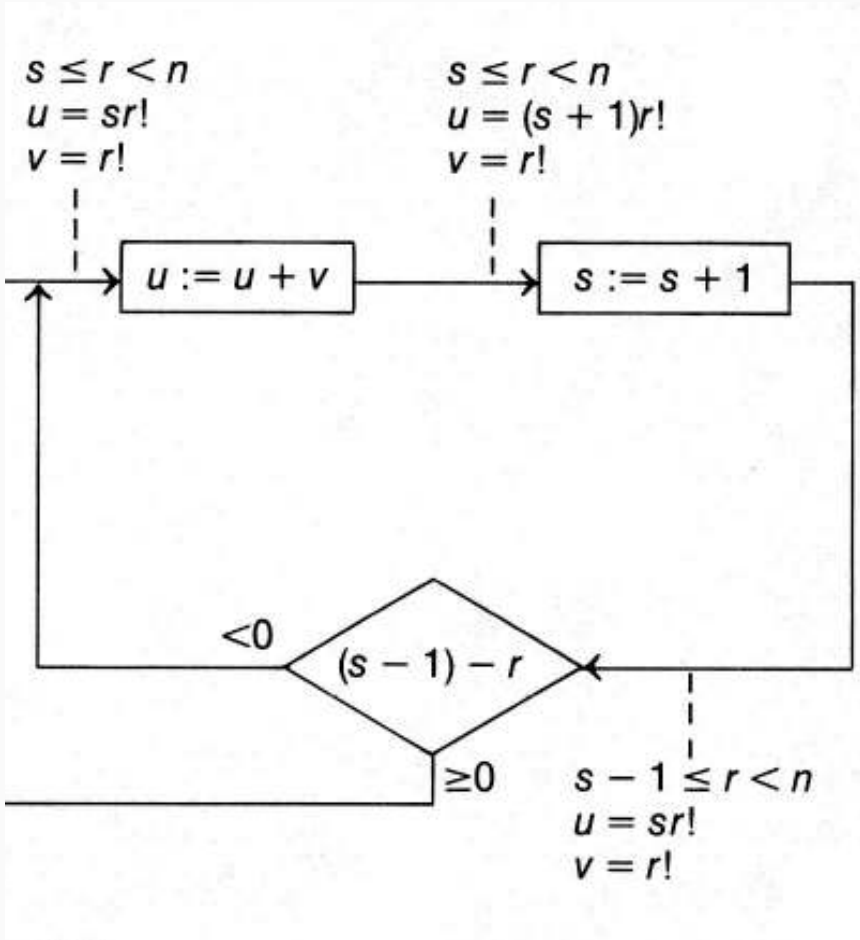
≪The checker has to verify that the columns corresponding to the initial condition and the stopped condition agree with the claim that are made for the routine as a whole.≫



$$r \leq n \wedge u = r! \wedge v = r! \wedge r - n \geq 0 \Longrightarrow v = n!$$

La vérification

≪[The checker] also has to verify that each of the assertions in the lower half of the table is correct. In doing this the columns may be taken in any order and quite independently.≫



$$s \le r < n \wedge u = sr! \wedge v = r!$$
$$\Downarrow$$
$$s \le r < n \wedge u + v = (s+1)r! \wedge v = r!$$

$$s - 1 \le r < n \wedge u = sr! \wedge v = r! \wedge (s-1) - r < 0$$
$$\Downarrow$$
$$s \le r < n \wedge u = sr! \wedge v = r!$$

Vérifier la terminaison

> ≪*Finally the checker has to verify that the process comes to an end. Here again he should be assisted by the programmer [...]. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops.*≫

Turing propose de prendre l'ordinal $(n - r)\omega^2 + (r - s)\omega + k$, ce qui correspond à l'ordre lexicographique sur $(n - r, r - s, k)$.

Plus prosaïquement il suggère $2^{80}(n - r) + 2^{40}(r - s) + k$.

STORAGE LOCATION	(INITIAL) Ⓐ $k = 6$	Ⓑ $k = 5$	Ⓒ $k = 4$	(STOP) Ⓓ $k = 0$	Ⓔ $k = 3$	Ⓕ $k = 1$	Ⓖ $k = 2$

La redécouverte et la formalisation : *Assigning meanings to programs* Robert W. Floyd, 1967

Un article fondateur

Robert W. Floyd

ASSIGNING MEANINGS TO PROGRAMS[1]

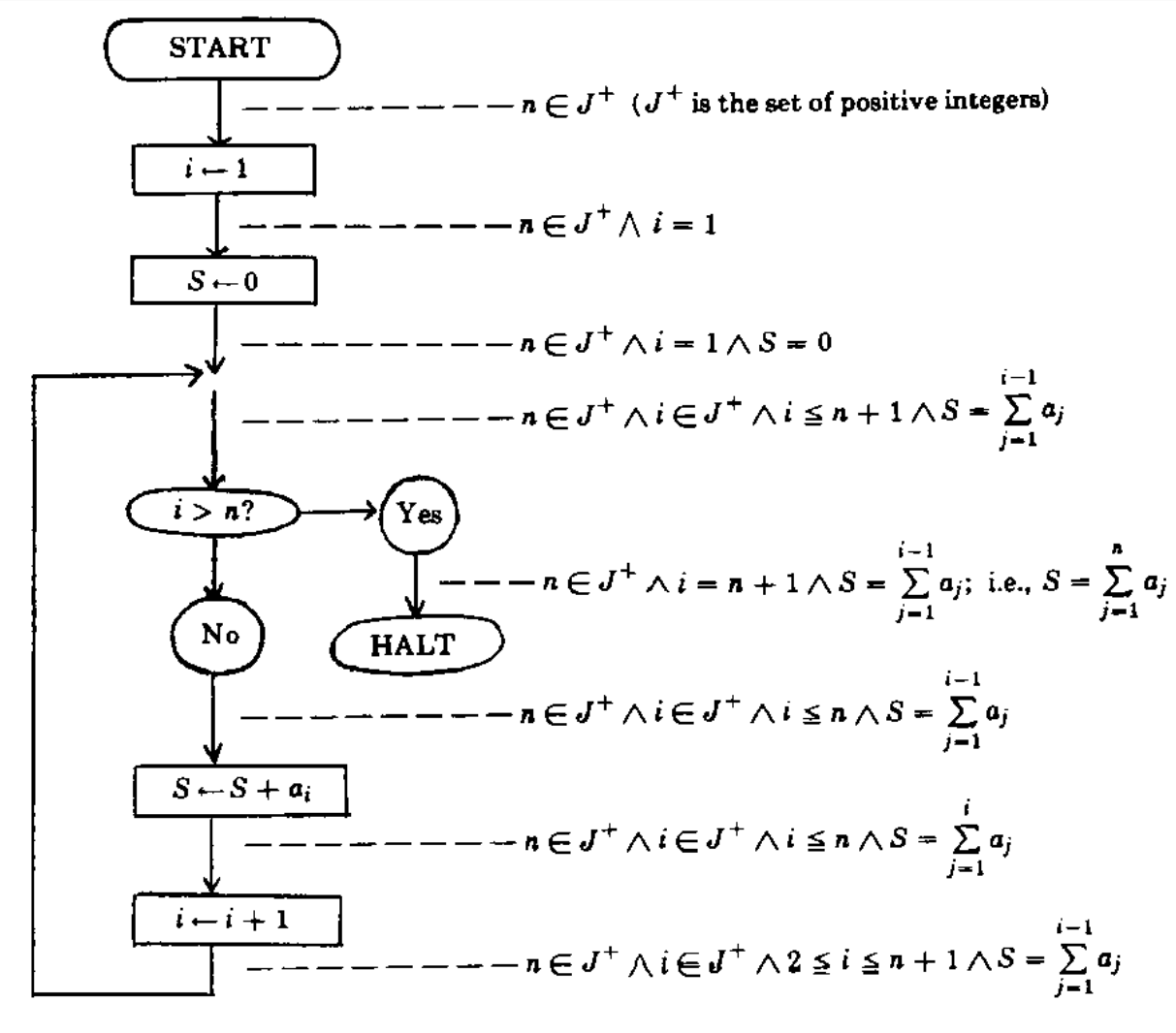
Introduction. This paper attempts to provide an adequate basis for formal definitions of the meanings of programs in appropriately defined programming languages, in such a way that a rigorous standard is established for proofs about computer programs, including proofs of correctness, equivalence, and termination. The basis of our approach is the notion of

Mathematical Aspects of Computer Science, 1967, 14 pages.

Proceedings of Symposium on Applied Mathematics, vol 19, AMS.

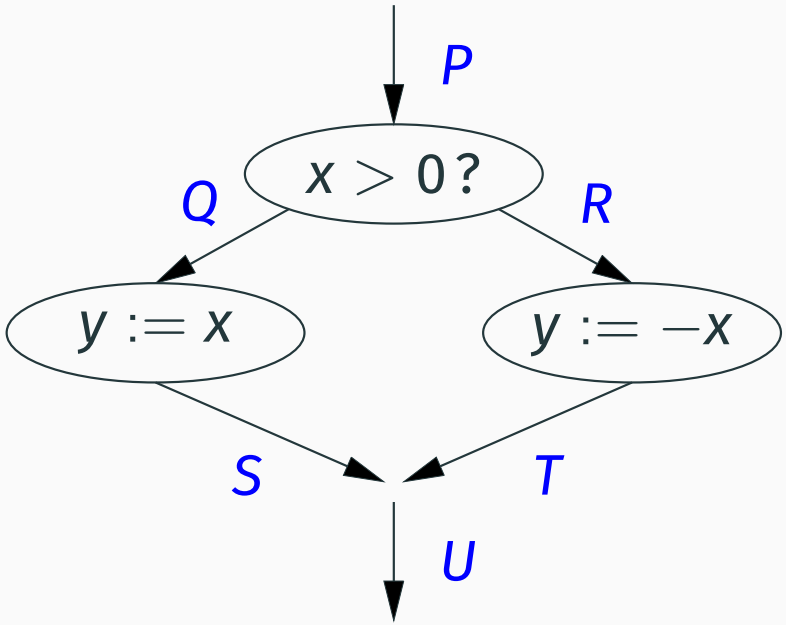
Le retour des assertions logiques

18 ans après, Floyd redécouvre l'idée de Turing :
annoter un organigramme par des assertions logiques.



Des conditions de vérification…

Floyd formalise les conditions de vérification :
les implications logiques qui garantissent la cohérence des assertions qui annotent le programme.



$$P \wedge x > 0 \Rightarrow Q$$
$$P \wedge x \leq 0 \Rightarrow R$$
$$\exists y_0, Q[y \leftarrow y_0] \wedge y = x \Rightarrow S$$
$$\exists y_0, R[y \leftarrow y_0] \wedge y = -x \Rightarrow T$$
$$S \vee T \Rightarrow U$$

Programme annoté $\longrightarrow$ Conditions de vérification

… à la sémantique du langage de programmation

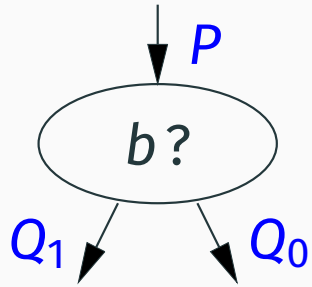
programme annoté $\xrightarrow[\text{formelle}]{\text{sémantique}}$ conditions de vérification

Floyd observe que les règles qui transforment un programme annoté en conditions de vérification constituent une sémantique du langage de programmation utilisé.

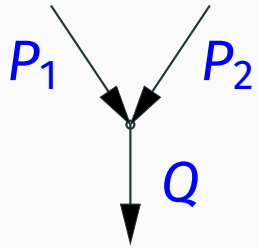
C'est la naissance de la sémantique formelle !
(Cf. mon cours ≪Sémantiques mécanisées≫ en 2019).

> ≪*[T]he proposal that the semantics of a programming language may be defined independently of all processors for that language, by establishing standards of rigor for proofs about programs in the language, appear to be novel*≫

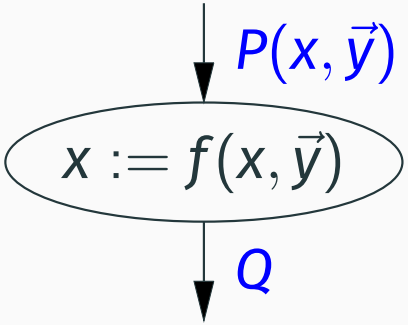
Conditions de vérification pour les organigrammes



$$P \wedge b \Rightarrow Q_1$$
$$P \wedge \neg b \Rightarrow Q_0$$



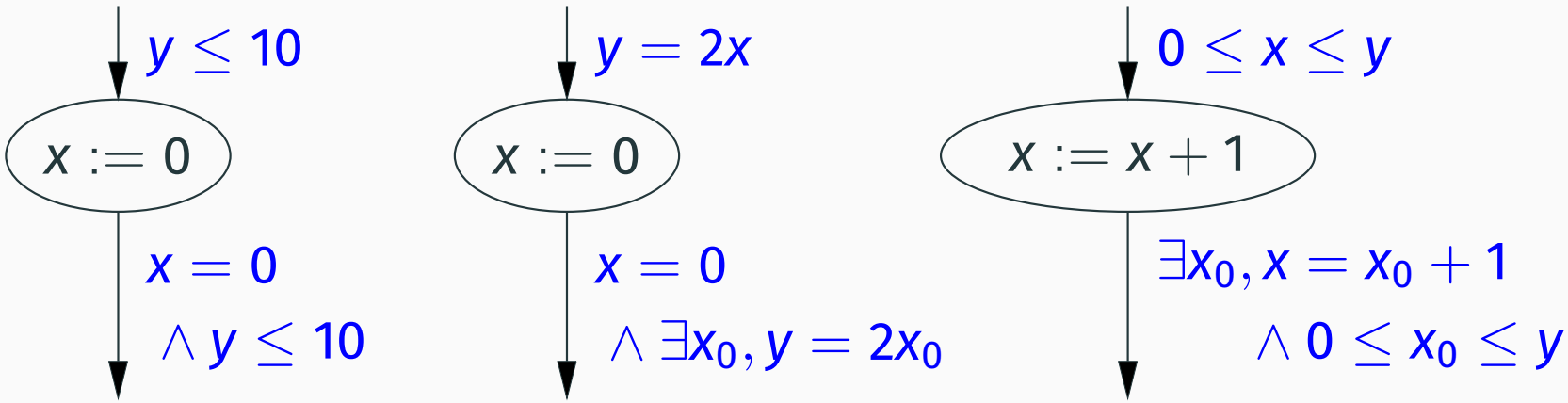
$$P_1 \vee P_2 \Rightarrow Q$$



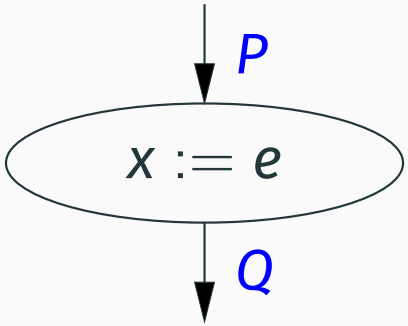
$$(\exists x_0, x = f(x_0, \vec{y}) \wedge P(x_0, \vec{y})) \Rightarrow Q$$

La règle de Floyd pour l'affectation

Exemples :



Le cas général :



$$(\exists x_0, x = e[x \leftarrow x_0] \wedge P[x \leftarrow x_0]) \Rightarrow Q$$

Quelques règles générales

Notations :

c	commande (morceau de programme)
$\vec{P}$	préconditions (une par entrée dans c)
$\vec{Q}$	postconditions (une par sortie de c)
$V_c(\vec{P}; \vec{Q})$	conditions de vérifications pour $\vec{P}$, c, $\vec{Q}$

Conséquence : si $V_c(\vec{P}; \vec{Q})$ et $\vec{P}' \Rightarrow \vec{P}$ et $\vec{Q} \Rightarrow \vec{Q}'$, alors $V_c(\vec{P}'; \vec{Q}')$.

Conjonction : si $V_c(\vec{P}; \vec{Q})$ et $V_c(\vec{P}', \vec{Q}')$ alors $V_c(\vec{P} \wedge \vec{P}'; \vec{Q} \wedge \vec{Q}')$.

Disjonction : si $V_c(\vec{P}; \vec{Q})$ et $V_c(\vec{P}', \vec{Q}')$ alors $V_c(\vec{P} \vee \vec{P}'; \vec{Q} \vee \vec{Q}')$.

Quantification existentielle : si $V_c(\vec{P}; \vec{Q})$ alors $V_c(\exists x.\ \vec{P}; \exists x.\ \vec{Q})$.

Correction sémantique

Si la condition de vérification $V_c(P_1 \ldots P_n; Q_1 \ldots Q_m)$ est vraie

c s'exécute d'un état initial s vers un état final s' (sortie j)

l'état initial s satisfait l'une des préconditions P_i

alors

l'état final s' satisfait la postcondition Q_j.

Facile à vérifier pour les règles des organigrammes.

Corollaire : si le programme démarre dans un état initial satisfaisant sa précondition P, et s'il termine, alors l'état final satisfait sa postcondition Q.

Plus forte conséquence vérifiable

Floyd conjecture que la condition de vérification $V_c(\vec{P}; \vec{Q})$ peut toujours s'écrire sous la forme

$$T_c(P_1 \vee \cdots \vee P_n) \Rightarrow \vec{Q}$$

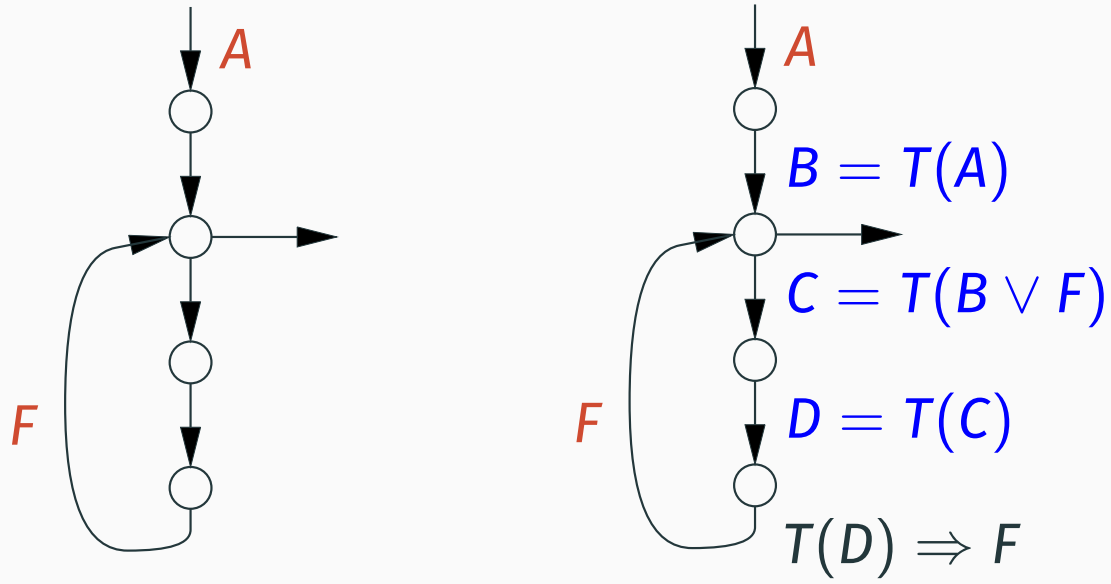
où $T_c(P)$ est la plus forte postcondition de la commande c avec la précondition P.

Par exemple, pour les organigrammes, on a

$$T_{x:=e}(P) = \exists x_0,\ x = e[x \leftarrow x_0] \wedge P[x \leftarrow x_0]$$
$$T_{\texttt{test}(b)}(P) = (P \wedge b, P \wedge \neg b)$$

Vers l'automatisation

En utilisant T, on peut compléter un organigramme partiellement annoté.



≪ *This fact offers the possibility of automatic verification of programs, the programmer merely tagging entrances and one edge in each innermost loop; the verifying program would extend the interpretation and verify it, if possible, by mechanical theorem-proving techniques.* ≫

Autres contributions de l'article

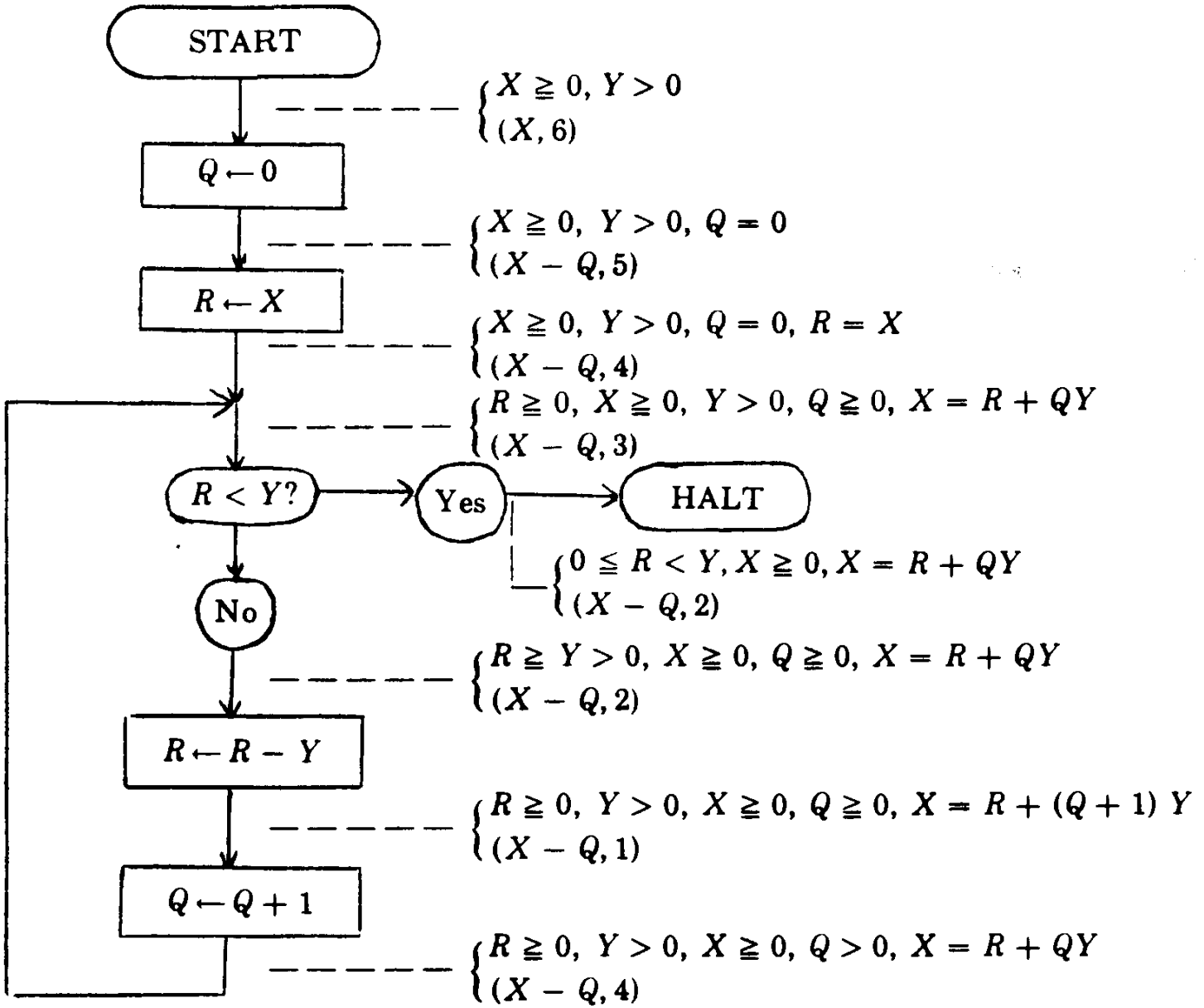
Un début de définition de V_c pour des commandes structurées à la manière d'Algol (séquence, if/then/else, boucle `for`).

Une discussion de la complétude de la définition de V_c
(voir le prochain cours).

Une méthode pour vérifier la terminaison :

- Associer à chaque arc de l'organigramme une fonction
  état des variables $\rightarrow$ ensemble bien fondé W
  (p.ex. W = n-uplets d'entiers + ordre lexicographique)
- Vérifier que ces fonctions décroissent sur chaque transition.

Exemple final de l'article : la division euclidienne



Le manifeste :

An axiomatic basis for computer programming

C. A. R. Hoare, 1969

Un article séminal

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

Communications of the ACM 12(10), 1969

Une position de principe : l'approche axiomatique

Une approche axiomatique permet de spécifier les programmes et de définir les langages sans pour autant tout spécifier.

Exemple de Hoare : les débordements arithmétiques (en arithmétique entière non signée).

Erreur :	$\texttt{MAX} + 1$ arrête le programme
Saturation :	$\texttt{MAX} + 1 = \texttt{MAX}$
Modulo :	$\texttt{MAX} + 1 = 0$

Axiomatiser l'arithmétique des ordinateurs

Hoare donne 9 axiomes qui sont vrais dans $\mathbb{N}$ mais aussi dans les trois types d'arithmétique machine :

A1	$x + y = y + x$	addition is commutative
A2	$x \times y = y \times x$	multiplication is commutative
A3	$(x + y) + z = x + (y + z)$	addition is associative
A4	$(x \times y) \times z = x \times (y \times z)$	multiplication is associative
A5	$x \times (y + z) = x \times y + x \times z$	multiplication distributes through addition
A6	$y \leqslant x \supset (x - y) + y = x$	addition cancels subtraction
A7	$x + 0 = x$	
A8	$x \times 0 = 0$	
A9	$x \times 1 = x$	

Il montre que ces axiomes suffisent à vérifier le programme de la division euclidienne.

Une notation : les ≪triplets de Hoare≫

Pour axiomatiser les programmes, Hoare introduit la notation

$$P \ \{ Q \} \ R$$

précondition — programme — postcondition

This may be interpreted "If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion".

La notation utilisée partout aujourd'hui :

$$\{ P \} \ c \ \{ Q \}$$

précondition — commande — postcondition

Une contribution : les règles pour un langage structuré

Plus d'organigrammes, mais des structures de contrôle dans le style d'Algol 60.

$$\{\, Q[x \leftarrow e] \,\}\ x := e\ \{\, Q \,\} \quad \text{(affectation)}$$

$$\frac{\{\, P \,\}\ c\ \{\, Q \,\} \quad Q \Rightarrow Q'}{\{\, P \,\}\ c\ \{\, Q' \,\}}\ \text{(conséquence 1)} \qquad \frac{P' \Rightarrow P \quad \{\, P \,\}\ c\ \{\, Q \,\}}{\{\, P' \,\}\ c\ \{\, Q \,\}}\ \text{(conséquence 2)}$$

$$\frac{\{\, P \,\}\ c_1\ \{\, Q \,\} \qquad \{\, Q \,\}\ c_2\ \{\, R \,\}}{\{\, P \,\}\ c_1; c_2\ \{\, R \,\}}\ \text{(composition)}$$

$$\frac{\{\, P \wedge b \,\}\ c\ \{\, P \,\}}{\{\, P \,\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{\, P \wedge \neg b \,\}}\ \text{(itération)}$$

La règle de Hoare pour l'affectation

$$\{\, Q[x \leftarrow e] \,\}\ x := e\ \{\, Q \,\}$$

Style «en arrière» : la postcondition Q détermine la précondition.

Exemple

$$\begin{array}{lll} \{\, 0 = 0 \wedge y \leq 10 \,\} & x := 0 & \{\, x = 0 \wedge y \leq 10 \,\} \\ \{\, 1 \leq x + 1 \leq 10 \,\} & x := x + 1 & \{\, 1 \leq x \leq 10 \,\} \end{array}$$

À comparer au style «en avant» de la règle de Floyd :

$$\{\, P \,\}\ x := e\ \{\, \exists x_0, x = e[x \leftarrow x_0] \wedge P[x \leftarrow x_0] \,\}$$

La règle de Hoare pour l'itération

$$\frac{\{\, P \wedge b \,\}\ c\ \{\, P \,\}}{\{\, P \,\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{\, P \wedge \neg b \,\}}\ \text{(itération)}$$

La précondition P doit être un invariant de boucle :
vrai au début du corps de boucle c à chaque itération ;
rétabli à la fin du corps c pour l'itération suivante.

Exemple (boucle comptée)

$x := 0;$

$\{\, 0 \leq x \leq 10 \,\}$

$\texttt{while}\ x < 10\ \texttt{do}$

$\{\, 0 \leq x \leq 10 \wedge x < 10 \,\}\ x := x + 1\ \{\, 0 \leq x \leq 10 \,\}$

$\texttt{done}$

$\{\, 0 \leq x \leq 10 \wedge \neg(x < 10) \,\} \Rightarrow \{\, x = 10 \,\}$

Dernier exemple de l'article : la division euclidienne

$r := x;$

$q := 0;$

`while` $y \leq r$ `do`

$\quad r := r - y;$

$\quad q := q + 1$

`done`

Line number	Formal proof	Justification
1	**true** $\supset x = x + y \times 0$	Lemma 1
2	$x = x + y \times 0\{r := x\}x = r + y \times 0$	D0
3	$x = r + y \times 0\ \{q := 0\}\ x = r + y \times q$	D0
4	**true** $\{r := x\}\ x = r + y \times 0$	D1 (1, 2)
5	**true** $\{r := x;\ \ q := 0\}\ x = r + y \times q$	D2 (4, 3)
6	$x = r + y \times q \wedge y \leqslant r \supset x =$ $(r-y) + y \times (1+q)$	Lemma 2
7	$x = (r-y) + y \times (1+q)\{r := r-y\}x =$ $r + y \times (1+q)$	D0
8	$x = r + y \times (1+q)\{q := 1+q\}x =$ $r + y \times q$	D0
9	$x = (r-y) + y \times (1+q)\{r := r-y;$ $q := 1+q\}\ x = r + y \times q$	D2 (7, 8)
10	$x = r + y \times q \wedge y \leqslant r\ \{r := r-y;$ $q := 1+q\}\ x = r + y \times q$	D1 (6, 9)
11	$x = r + y \times q\ \{$**while** $y \leqslant r$ **do** $(r := r-y;\ \ q := 1+q)\}$ $\neg y \leqslant r \wedge x = r + y \times q$	D3 (10)
12	**true** $\{((r := x;\ \ q := 0);$ **while** $y \leqslant r$ **do** $(r := r-y;\ \ q := 1+q))\}\ \neg y \leqslant r \wedge x =$ $r + y \times q$	D2 (5, 11)

Le manifeste de la vérification déductive

Une discussion de tout ce qui reste à faire :

- Vérifier la terminaison et l'absence d'erreurs à l'exécution.
- Arithmétiques (virgule flottante), tableaux, enregistrements, procédures, fonctions, récursion, `goto`, pointeurs.

Un argumentaire : pourquoi vérifier les programmes ?

- Coût élevé du test.
- Coût très élevé de l'erreur.
- Documentation ; portabilité.

Quelques extraits

When the correctness of a program, its compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with a confidence limited only by the reliability of electronics.

Quelques extraits

The cost of error in certain types of program may be almost incalculable—a lost spacecraft, a collapsed building, a crashed aeroplane, or a world war. Thus, the practice of program proving is not only a theoretical pursuit, followed in the interest of academic responsibility, but a serious recommendation for the reduction of the costs associated with programming error.

Quelques extraits

However, program proving, certainly at present, will be difficult even for programmers of high caliber; and may be applicable only to quite simple program designs. As in other areas, reliability can be purchased only at the price of simplicity.

Point d’étape

Point d'étape

Dès 1969, les grandes lignes de la vérification déductive sont déjà tracées dans les travaux de Floyd et de Hoare.

Beaucoup de travail reste à faire :

- Années 1970, 1980 : approfondissement des bases de la ≪logique de Hoare≫ ($\rightarrow$ cours n° 2)
- À partir des années 1990 : mise en pratique sous forme d'outils de vérification déductive ($\rightarrow$ séminaires n° 1 et 2)

Le prochain grand tournant du domaine sera vers 2000…