How can we reason about software? The birth of program logics

Xavier Leroy

Collège de France, chair of software sciences
xavier.leroy@college-de-france.fr

Course outline

A review of three articles that started it all:

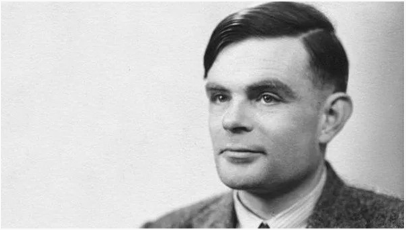
- Alan Turing, *Checking a large routine*, 1949.
- Robert W. Floyd, *Assigning meanings to programs*, 1967.
- C. A. R. Hoare, *An axiomatic basis for computer programming*, 1969.

The discovery:

Checking a large routine

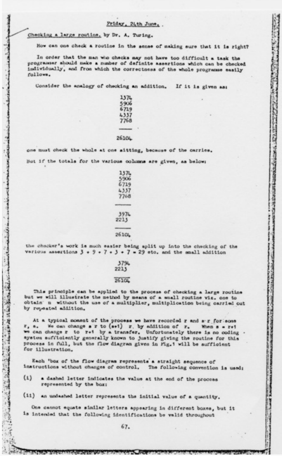
Alan Turing, 1949

Alan Mathison Turing, 1912–1954



1931-36	Cambridge: studies mathematics
1936	Publishes the founding paper of computability theory
1936-38	Princeton: Ph.D. with A. Church
1939-44	Bletchley Park: breaking German ciphers
1945-47	Cambridge: design of the ACE programmable computer
1948-50	Manchester: the Mark 1 programmable computer (Ferranti); “Turing’s test” in artificial intelligence.
1951-53	Manchester: mathematical biology; morphogenesis.

A pioneering article



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.

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

1374
5906
6719
4337
7768

26104

one must check the whole at one sitting, because of the carries.

But if the totals for the various columns are given, as below:

1374
5906
6719
4337
7768

3974
2213

26104

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

3794
2213

26104

This principle can be applied to the process of checking a large routine but we will illustrate the method by means of a small routine viz. one to obtain n without the use of a multiplier, multiplication being carried out by repeated addition.

At a typical moment of the process we have recorded r and s r for some r, s. We can change s r to (s+1) r, by addition of r. When s = r+1 we can change r to r+1 by a transfer. Unfortunately there is no coding system sufficiently generally known to justify giving the routine for this process in full, but the flow diagram given in Fig.1 will be sufficient for illustration.

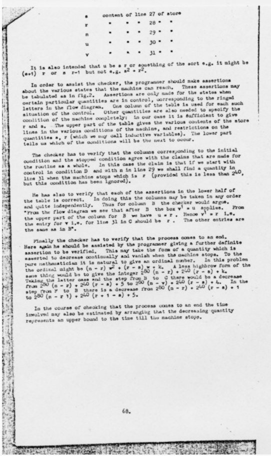
Each 'box of the flow diagram represents a straight sequence of instructions without changes of control. The following convention is used:

(i) a dashed letter indicates the value at the end of the process represented by the box:

(ii) an undashed letter represents the initial value of a quantity.

One cannot equate similar letters appearing in different boxes, but it is intended that the following identifications be valid throughout

67.



s content of line 27 of store
r " " " 28 " "
n " " " 29 " "
u " " " 30 " "
v " " " 31 " "

It is also intended that u be s r or something of the sort e.g. it might be (s+1) r or s r-1 but not e.g. $s^2 + r^2$.

In order to assist the checker, the programmer should make assertions about the various states that the machine can reach. These assertions may be tabulated as in fig.2. Assertions are only made for the states when certain particular quantities are in control, corresponding to the ringed letters in the flow diagram. One column of the table is used for each such situation of the control. Other quantities are also needed to specify the condition of the machine completely: in our case it is sufficient to give r and s. The upper part of the table gives the various contents of the store lines in the various conditions of the machine, and restrictions on the quantities s, r (which we may call inductive variables). The lower part tells us which of the conditions will be the next to occur.

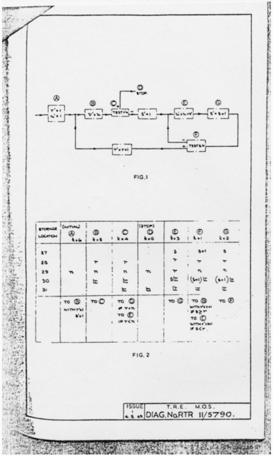
The checker has to verify that the columns corresponding to the initial condition and the stopped condition agree with the claims that are made for the routine as a whole. In this case the claim is that if we start with control in condition D and with n in line 29 we shall find a quantity in line 31 when the machine stops which is r (provided this is less than 2^{40}, but this condition has been ignored).

He has also 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. Thus for column B the checker would argue. "From the flow diagram we see that after B the box $v' = u$ applies. From the upper part of the column for B we have u = r. Hence $v' = r$ i.e. the entry for v i.e. for line 31 in C should be r. The other entries are the same as in B".

Finally the checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops. To the pure mathematician it is natural to give an ordinal number. In this problem the ordinal might be $(n - r)\,\omega^2 + (r - s)\,\omega + k$. A less highbrow form of the same thing would be to give the integer $2^{80}(n - r) + 2^{40}(r - s) + k$. Taking the latter case and the step from B to C there would be a decrease from $2^{80}(n - r) + 2^{40}(r - s) + 5$ to $2^{80}(n - v) + 2^{40}(r - s) + 4$. In the step from F to B there is a decrease from $2^{80}(n - r) + 2^{40}(r - s) + 1$ to $2^{80}(n - r - 1) + 2^{40}(r + 1 - s) + 5$.

In the course of checking that the process comes to an end the time involved may also be estimated by arranging that the decreasing quantity represents an upper bound to the time till the machine stops.

68.



Talk given at the inaugural conference of the EDSAC computer, Cambridge University, june 1949. The manuscript was commented, and republished by F.L. Morris and C.B. Jones in *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.

Decomposing verification in elementary steps

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

```
 1374
 5906
 6719
 4337
 7768
-----
26104
```

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

```
 1374
 5906
 6719
 4337
 7768
-----
 3974
2213
-----
26104
```

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

```
 3974
2213
-----
26104
```

Turing's program: the factorial function

Compute $n!$ using additions only.

Two nested loops.

```
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;
}
```

The flowchart for the program

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

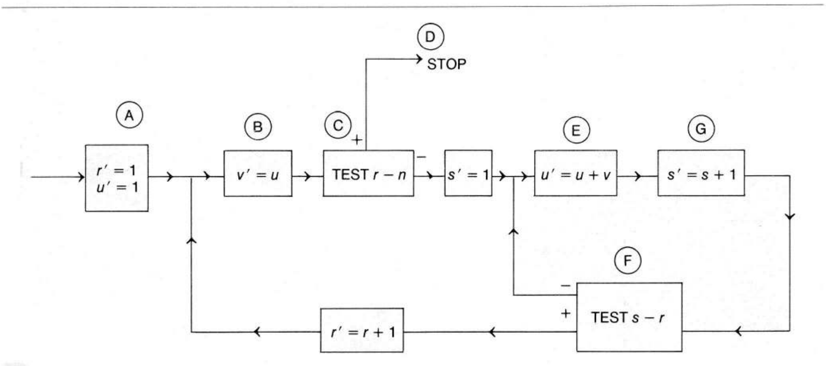


Figure 1 (Redrawn from Turing's original)

(The notation u/u' denotes the value of u before/after the block).

Logical assertions

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

The assertions document not only which memory location contains which abstract variable, but also relations between these variables.

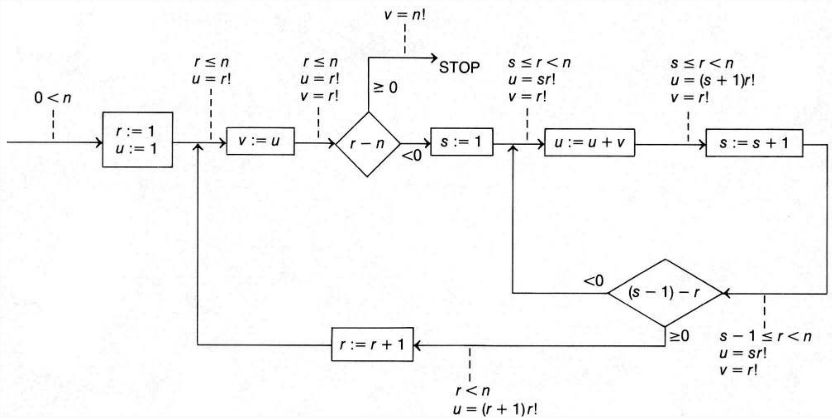
STORAGE LOCATION	(INITIAL) Ⓐ $k = 6$	Ⓑ $k = 5$	Ⓒ $k = 4$	(STOP) Ⓓ $k = 0$	Ⓔ $k = 3$	Ⓕ $k = 1$	Ⓖ $k = 2$
27					s	$s + 1$	s
28		r	r		r	r	r
29	n	n	n	n	n	n	n
30		$\underline{\lvert r}$	$\underline{\lvert r}$		$s\underline{\lvert r}$	$(s + 1)\underline{\lvert r}$	$(s + 1)\underline{\lvert r}$
31			$\underline{\lvert r}$	$\underline{\lvert n}$	$\underline{\lvert r}$	$\underline{\lvert r}$	$\underline{\lvert r}$
	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)

(The notation $\underline{\lvert n}$ means “n factorial”.)

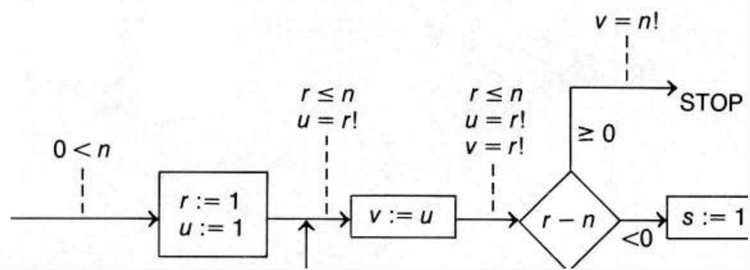
Logical assertions on the flowchart

In the modern notation (introduced by Floyd in 1967), we write the assertions directly on the edges of the flowchart.



Verification

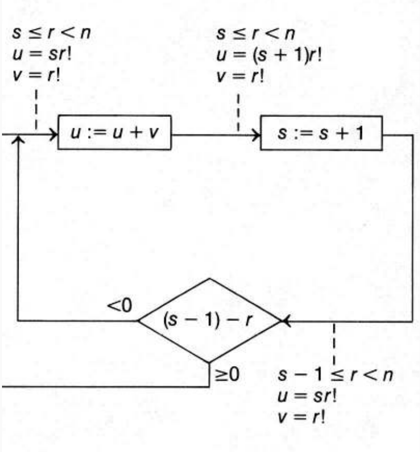
"The checker has to verify that the columns corresponding to the initial condition and the stopped condition agree with the claim that are made for the routine as a whole."



$$r \leq n \land u = r! \land v = r! \land r - n \geq 0 \implies v = n!$$

Verification

"[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."



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

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

Verifying termination

> *"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 suggests taking the ordinal $(n-r)\omega^2 + (r-s)\omega + k$, which corresponds to lexicographic ordering on $(n-r, r-s, k)$.

More pragmatically, he suggests $2^{80}(n-r) + 2^{40}(r-s) + k$.

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

Rediscovery and formalization:
Assigning meanings to programs
Robert W. Floyd, 1967

Robert W Floyd, 1936–2001

1953	B.A. in liberal arts, U. Chicago	
1958	B.S. in physics, U. Chicago	
195?-61	Computer programmer, Illinois I.T.	(syntax analysis)
1962-64	Senior scientist, Computer Associates	(compilers)
1965-67	Associate professor, Carnegie I.T.	(algorithms, semantics)
1968-91	Professor, Stanford	(algorithms)
1978	Turing award	

A fundamental paper

Robert W. Floyd

ASSIGNING MEANINGS TO PROGRAMS[1]

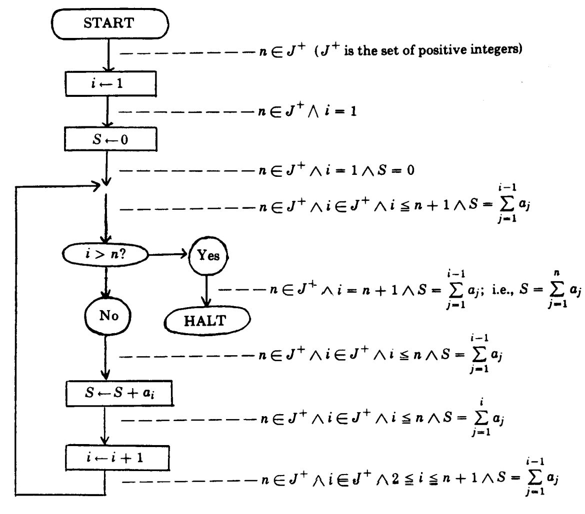
Introduction. This paper attempts to provide an adequate basis for formal definitions of the meanings of programs in appropriately defined programming languages, in such a way that a rigorous standard is established for proofs about computer programs, including proofs of correctness, equivalence, and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

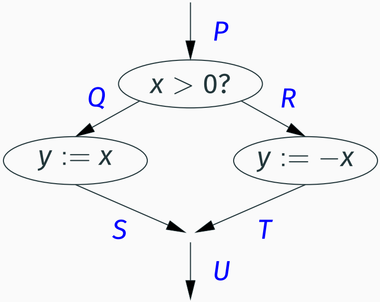
The return of logical assertions

18 years later, Floyd rediscovers Turing's idea: annotate a flowchart with logical assertions.



From verification conditions…

Floyd formalizes the verification conditions:
logical implications that guarantee the logical consistency of the assertions annotating the program.



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

Annotated program $\longrightarrow$ Verification conditions

… to the semantics of the programming language

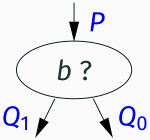
annotated program $\xrightarrow[\text{semantics}]{\text{formal}}$ verification conditions

Floyd notices that the rules transforming an annotated program into verification conditions constitute a semantics of the programming language.

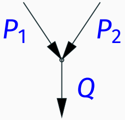
It's the birth of formal semantics!

> *"[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"*

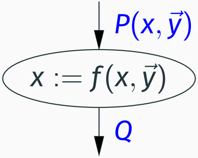
Verification conditions for flowcharts



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

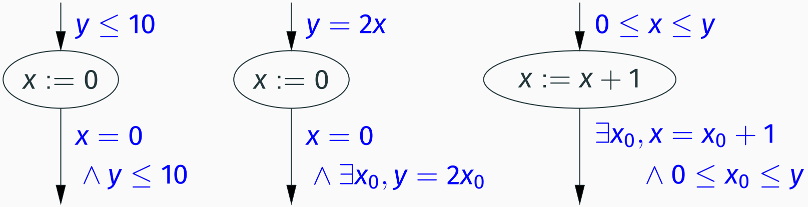


$$P_1 \vee P_2 \Rightarrow Q$$

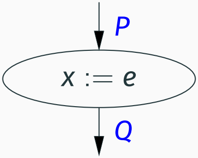


$$(\exists x_0, x = f(x_0, \vec{y}) \wedge P(x_0, \vec{y})) \Rightarrow Q$$

Examples:



General case:



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

Generic rules

Notations:

c	command (fragment of a program)
$\vec{P}$	preconditions (one per entry in c)
$\vec{Q}$	postconditions (one per exit out of c)
$V_c(\vec{P}; \vec{Q})$	verification conditions for $\vec{P}$, c, $\vec{Q}$

Consequence: if $V_c(\vec{P}; \vec{Q})$ and $\vec{P}' \Rightarrow \vec{P}$ and $\vec{Q} \Rightarrow \vec{Q}'$, then $V_c(\vec{P}'; \vec{Q}')$.

Conjunction: if $V_c(\vec{P}; \vec{Q})$ and $V_c(\vec{P}', \vec{Q}')$ then $V_c(\vec{P} \land \vec{P}'; \vec{Q} \land \vec{Q}')$.

Disjunction: if $V_c(\vec{P}; \vec{Q})$ and $V_c(\vec{P}', \vec{Q}')$ then $V_c(\vec{P} \lor \vec{P}'; \vec{Q} \lor \vec{Q}')$.

Existential quantification: if $V_c(\vec{P}; \vec{Q})$ then $V_c(\exists x.\ \vec{P}; \exists x.\ \vec{Q})$.

Semantic soundness

If the verification condition $V_c(P_1 \ldots P_n; Q_1 \ldots Q_m)$ holds
c executes from initial state s to final state s' (exit number j)
the initial state s satisfies one of the preconditions P_i

then

the final state s' satisfies postcondition Q_j.

Easy to prove for the flowchart rules.

Corollary: if the program starts in an initial state satisfying its precondition P, and if it terminates, then the final state satisfies its postcondition Q.

Strongest verifiable consequence

Floyd conjectures that the verification condition $V_c(\vec{P}; \vec{Q})$ can always be written as

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

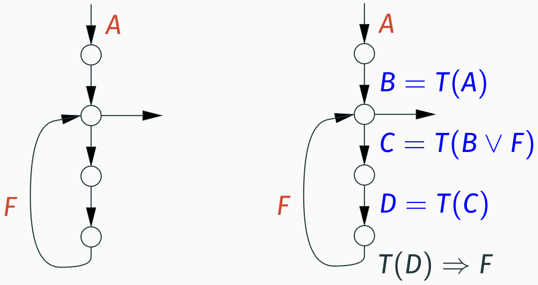
where $T_c(P)$ is the strongest postcondition for command c with precondition P.

For example, in the case of flowcharts, we have

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

Towards automation

Using T, we can complete a partially-annotated flowchart.



“ 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. ”

Other contributions of Floyd's paper

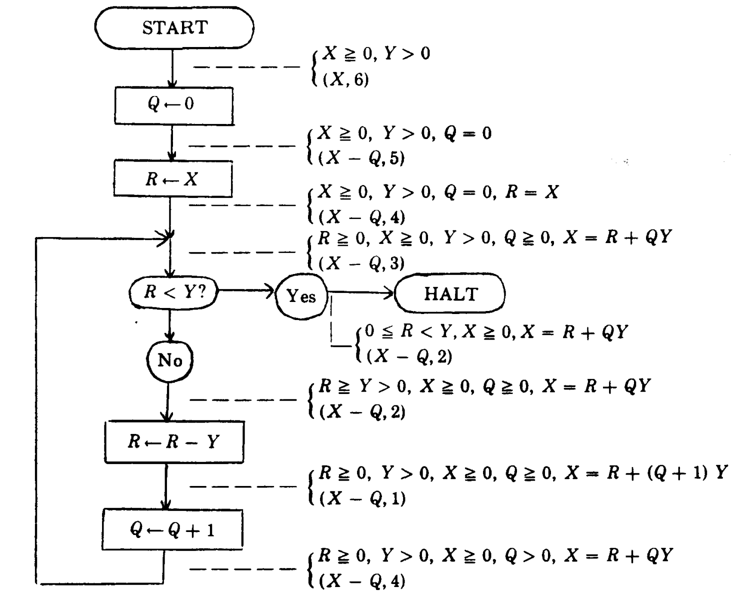
A partial definition of V_c for structured commands in the style of Algol (sequences, `if`/`then`/`else`, `for` loops).

A discussion of completeness for the definition of V_c (see next lecture).

A method to verify termination:

- To each edge of the flowchart, associate a function values of variables $\rightarrow$ well-founded set W (e.g. W = tuples of integers with lexicographic ordering)
- Check that these functions decrease at each transition.

Final example in the paper: Euclidean division



An axiomatic basis for computer programming

C. A. R. Hoare, 1969

Sir Charles Antony Richard Hoare, 1934–

1952-55	B.A. in philosophy, Oxford
1956-57	Serves in the Royal Navy
1958	Master in statistics, Oxford
1959	Works with Kolmogorov at Lomonossov university, Moscow
1960–67	Works at Elliot Brothers: compiling Algol; Quicksort.
1968–76	Professor, University of Belfast.
1977–	Professor, University of Oxford
1980	Turing award
1999–	Principal researcher, Microsoft Research, Cambridge
2000	Knighthood

A seminal paper

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

A principled position: the axiomatic approach

An axiomatic approach makes it possible to specify programs and define programming languages without specifying everything.

Hoare's example: arithmetic overflows (in unsigned integer arithmetic).

Error:	$\texttt{MAX} + 1$ halts the program
Saturation:	$\texttt{MAX} + 1 = \texttt{MAX}$
Modulo:	$\texttt{MAX} + 1 = 0$

Axiomatizing computer arithmetic

Hoare states 9 axioms that hold in $\mathbb{N}$ but also in the three kinds of machine arithmetic:

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	$x \times 1 = x$	

He shows that these axioms suffice to verify Euclidean division.

A notation: "Hoare triples"

To axiomatize programs, Hoare introduces the notation

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

$\nearrow$ precondition $\uparrow$ program $\nwarrow$ 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".*

The notation universally used today:

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

$\nearrow$ precondition $\uparrow$ command $\nwarrow$ postcondition

A contribution: the rules for a structured language

Instead of flowcharts, Hoare considers control structures in the style of Algol 60.

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

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

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

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

Hoare's rule for assignment

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

"Backward" reasoning style: the postcondition Q determines the precondition.

Example

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

Contrast with the "forward" style of Floyd's rule:

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

Hoare's rule for iteration

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

The precondition P must be a loop invariant:
true at the beginning of the loop body c at every iteration;
re-established at the end of the body c for the next iteration.

Example (counted loop)

$x := 0;$

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

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

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

$\texttt{done}$

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

Final example in the paper: Euclidean division

$r := x;$

$q := 0;$

$\texttt{while}\ y \leq r\ \texttt{do}$

$\quad r := r - y;$

$\quad q := q + 1$

$\texttt{done}$

Line number	Formal proof	Justification
1	$\mathbf{true} \supset x = x + y \times 0$	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	$\mathbf{true}\ \{r := x\}\ x = r + y \times 0$	D1 (1, 2)
5	$\mathbf{true}\ \{r := x;\ \ q := 0\}\ x = r + y \times q$	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	$x = r + y \times q\ \{\mathbf{while}\ y \leqslant r\ \mathbf{do}\ (r := r-y;\ \ q := 1+q)\}\ \neg y \leqslant r \wedge x = r + y \times q$	D3 (10)
12	$\mathbf{true}\ \{((r := x;\ \ q := 0);\ \ \mathbf{while}\ y \leqslant r\ \mathbf{do}\ (r := r-y;\ \ q := 1+q))\}\ \neg y \leqslant r \wedge x = r + y \times q$	D2 (5, 11)

The manifesto of deductive verification

A discussion of all that remains to be done:

- Verify termination and absence of run-time errors.
- More arithmetic (incl. floating point), arrays, records, procedures, functions, recursion, `goto`, pointers.

An advocacy of program verification

- Testing is expensive.
- Error is very expensive.
- Documentation; portability.

Some quotes

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.

Some quotes

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.

Some quotes

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.

Summary

Summary so far

As early as 1969, the general principles of deductive verification have already been set in the works of Floyd and Hoare.

Much work remains:

- 1970's and 1980's: deeper understanding of the foundations for "Hoare logic". ($\rightarrow$ lecture #2)
- 1990's and 2000's: implementation within deductive verification tools

The next major turning point in the area takes place around year 2000…