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

Xavier Leroy
2021-03-04

Collège de France, chair of software sciences
xavier.leroy@college-de-france.fr
A review of three articles that started it all:

- Alan Turing, *Checking a large routine*, 1949.
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.
A pioneering article

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

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:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1374</td>
<td>5906</td>
<td>6719</td>
</tr>
<tr>
<td>4337</td>
<td>7768</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>26104</td>
</tr>
</tbody>
</table>

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1374</td>
<td>5906</td>
<td>6719</td>
</tr>
<tr>
<td>4337</td>
<td>7768</td>
<td></td>
</tr>
<tr>
<td>3974</td>
<td>2213</td>
<td>26104</td>
</tr>
</tbody>
</table>

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3974</td>
<td>2213</td>
</tr>
<tr>
<td>26104</td>
<td></td>
</tr>
</tbody>
</table>
Turing's program: the factorial function

Compute $n!$ using additions only.

Two nested loops.

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

(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.

---

The contributors to the conference discussion were M. H. A. Newman, then professor of pure mathematics at Manchester University, who had played a leading part in setting up the Manchester computer project, and D. R. Hartree, then professor of mathematical physics at Cambridge University, who had been a moving force both at the NPL and at Cambridge.

We now turn to a discussion of Turing’s proof method. Present methods might combine Turing’s Figures 1 and 2 into a flowchart that includes the assertions. Figure A is an annotated flowchart in the style of Floyd (1967). Two significant differences between Figure A and Turing’s presentation may be observed.

1. In the Floyd style, assertions may be any propositions relating the values of the variables to each other.

---

(The notation $|n$ means “$n$ factorial”.)
In the modern notation (introduced by Floyd in 1967), we write the assertions directly on the edges of the flowchart.
“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.”

\[ r \leq n \land u = r! \land v = r! \land r - n \geq 0 \implies v = n! \]
“[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.”

\[
\begin{align*}
\text{Verification} \\
\begin{array}{ll}
\text{s} \leq r < n & \wedge u = sr! \wedge v = r! \\
\downarrow \\
\begin{array}{l}
s \leq r < n \\
\wedge u + v = (s + 1)r! \wedge v = r!
\end{array}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{ll}
s - 1 \leq r < n & \wedge u = sr! \wedge v = r! \\
\downarrow \\
\begin{array}{l}
s - 1 \leq r < n \\
\wedge (s - 1) - r < 0
\end{array}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{ll}
s \leq r < n & \wedge u = sr! \wedge v = r!
\end{array}
\end{align*}
\]
“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 \(2^{80}(n - r) + 2^{40}(r - s) + k\).
Rediscovery and formalization:
Assigning meanings to programs
Robert W. Floyd, 1967
<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1953</td>
<td>B.A. in liberal arts</td>
<td>U. Chicago</td>
</tr>
<tr>
<td>1958</td>
<td>B.S. in physics</td>
<td>U. Chicago</td>
</tr>
<tr>
<td>1957-61</td>
<td>Computer programmer</td>
<td>Illinois I.T. (syntax analysis)</td>
</tr>
<tr>
<td>1962-64</td>
<td>Senior scientist</td>
<td>Computer Associates (compilers)</td>
</tr>
<tr>
<td>1965-67</td>
<td>Associate professor</td>
<td>Carnegie I.T. (algorithms, semantics)</td>
</tr>
<tr>
<td>1968-91</td>
<td>Professor</td>
<td>Stanford (algorithms)</td>
</tr>
<tr>
<td>1978</td>
<td>Turing award</td>
<td></td>
</tr>
</tbody>
</table>
ASSIGNING MEANINGS TO PROGRAMS

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

1 This work was supported by the Advanced Projects Agency of the Office of the Secretary of Defense (SD-146).

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.

![Flowchart of program t() compute S = 1:1-1 OJ (n 0)](image-url)
Floyd formalizes the verification conditions: logical implications that guarantee the logical consistency of the assertions annotating the program.

\[
\begin{align*}
\text{Annotated program} & \quad \longrightarrow \quad \text{Verification conditions} \\
\end{align*}
\]
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!
(See my 2019 lectures “Mechanized 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

\[ x := f(x, \bar{y}) \]

\[ (\exists x_0, x = f(x_0, \bar{y}) \land P(x_0, \bar{y})) \Rightarrow Q \]
Floyd’s rule for assignment

Examples:

\[ x := 0 \]
\[ y \leq 10 \]
\[ x = 0 \]
\[ y = 2x \]
\[ x = 0 \]
\[ 0 \leq x \leq y \]
\[ \exists x_0, x = x_0 + 1 \]
\[ \exists x_0, x = x_0 + 1 \]

General case:

\[ x := e \]
\[ P \]
\[ Q \]
\[ (\exists x_0, x = e[x \leftarrow x_0] \land P[x \leftarrow x_0]) \Rightarrow Q \]
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, then $c$ 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 $Q$. 
Floyd conjectures that the verification condition $V_c(\vec{P}; \vec{Q})$ can always be written as

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

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

For example, in the case of flowcharts, we have

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

$$T_{\text{test}(b)}(P) = (P \land b, P \land \neg b)$$
Towards automation

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

```
A

F
```

```
A
B = T(A)
C = T(B \lor F)
D = T(C)

T(D) \Rightarrow F
```

“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, \texttt{if/then/else}, \texttt{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

\[
\begin{align*}
\text{START} & \quad \{ X \geq 0, Y > 0 \} \\
Q \leftarrow 0 & \quad \{ X \geq 0, Y > 0, Q = 0 \} \\
R \leftarrow X & \quad \{ X \geq 0, Y > 0, Q = 0, R = X \} \\
R < Y? & \quad \{ R \geq 0, X \geq 0, Y > 0, Q \geq 0, X = R + QY \} \\
\text{Yes} & \quad \{ R < Y, X \geq 0, X = R + QY \} \\
\text{No} & \quad \{ R \geq Y > 0, X \geq 0, Q \geq 0, X = R + QY \} \\
R \leftarrow R - Y & \quad \{ R \geq 0, Y > 0, X \geq 0, Q \geq 0, X = R + (Q + 1)Y \} \\
Q \leftarrow Q + 1 & \quad \{ R \geq 0, Y > 0, X \geq 0, Q > 0, X = R + QY \}
\end{align*}
\]

\text{HALT}

References
The manifesto:
An axiomatic basis for computer programming
C. A. R. Hoare, 1969
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
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
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:** $\text{MAX} + 1$ halts the program
- **Saturation:** $\text{MAX} + 1 = \text{MAX}$
- **Modulo:** $\text{MAX} + 1 = 0$
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  | $y \leq x \Rightarrow (x - y) + y = x$ | 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.
To axiomatize programs, Hoare introduces the notation

\[ P \{ Q \} R \]

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:

\[ \{ P \} c \{ Q \} \]
Instead of flowcharts, Hoare considers control structures in the style of Algol 60.

\[
\begin{align*}
\{ Q[x \leftarrow e] \} \ x := e \ \{ Q \} \quad \text{(assignment)}
\end{align*}
\]

\[
\begin{align*}
\{ P \} \ c \ \{ Q \} \quad Q \Rightarrow Q' & \quad & P' \Rightarrow P & \quad \{ P \} \ c \ \{ Q \} \\
\{ P \} \ c \ \{ Q' \} & \quad \{ P' \} \ c \ \{ Q \} & \quad \{ P' \} \ c \ \{ Q \}
\end{align*}
\]

\[
\begin{align*}
\{ P \} \ c_1 \ \{ Q \} \quad \{ Q \} \ c_2 \ \{ R \} & \quad \text{(composition)}
\end{align*}
\]

\[
\begin{align*}
\{ P \} \ c_1 ; \ c_2 \ \{ R \} \\
\{ P \ \land \ b \} \ c \ \{ P \} & \quad \text{(iteration)}
\end{align*}
\]

\[
\begin{align*}
\{ P \} \ \text{while } b \ \text{do } c \ \{ P \ \land \ \neg b \}
\end{align*}
\]
Hoare’s rule for assignment

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

“Backward” reasoning style: the postcondition \( Q \) determines the precondition.

Example

\begin{align*}
\{ 0 = 0 \land y \leq 10 \} & \quad x := 0 \quad \{ x = 0 \land y \leq 10 \} \\
\{ 1 \leq x + 1 \leq 10 \} & \quad x := x + 1 \quad \{ 1 \leq x \leq 10 \}
\end{align*}

Contrast with the “forward” style of Floyd’s rule:

\begin{align*}
\{ P \} \ x := e \ \{ \exists x_0, x = e[x \leftarrow x_0] \land P[x \leftarrow x_0] \}
\end{align*}
Hoare’s rule for iteration

\[
\{ P \land b \} \ c \ \{ P \} \quad \text{(iteration)}
\]

\[
\{ P \} \ \text{while } b \ \text{do } c \ \{ P \land \lnot b \} 
\]

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)**

\[
x := 0; \quad \{ 0 \leq x \leq 10 \} \\
\text{while } x < 10 \ \text{do} \quad \{ 0 \leq x \leq 10 \land x < 10 \} \ x := x + 1 \ \{ 0 \leq x \leq 10 \} \\
\text{done} \quad \{ 0 \leq x \leq 10 \land \lnot (x < 10) \} \implies \{ x = 10 \} 
\]
Final example in the paper: Euclidean division

\[ r := x; \]
\[ q := 0; \]
\[ \textbf{while } y \leq r \textbf{ do} \]
\[ r := r - y; \]
\[ q := q + 1 \]
\[ \textbf{done} \]
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.
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.
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
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”.
  (→ lecture #2)

- 1990’s and 2000’s: implementation within deductive verification tools
  (→ seminars #1 and #2)

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