An Early Program Proof by Alan Turing

F. L. MORRIS AND C. B. JONES

The paper reproduces, with typographical corrections and comments, a 1949 paper by Alan Turing that foreshadows much subsequent work in program proving.

*Categories and Subject Descriptors: D.2.4 [**Software Engineering**]—correctness proofs; F.3.1 [**Logics and Meanings of Programs**]—assertions; K.2 [**History of Computing**]—software*
General Terms: Verification
Additional Key Words and Phrases: A. M. Turing

Introduction

The standard references for work on program proofs attribute the early statement of direction to John McCarthy (e.g., McCarthy 1963); the first workable methods to Peter Naur (1966) and Robert Floyd (1967); and the provision of more formal systems to C. A. R. Hoare (1969) and Edsger Dijkstra (1976). The early papers of some of the computing pioneers, however, show an awareness of the need for proofs of program correctness and even present workable methods (e.g., Goldstine and von Neumann 1947; Turing 1949).

The 1949 paper by Alan M. Turing is remarkable in many respects. The three (foolscap) pages of text contain an excellent motivation by analogy, a proof of a program with two nested loops, and an indication of a general proof method very like that of Floyd. Unfortunately, the paper is made extremely difficult to read by the large number of transcription errors. For example, all instances of the factorial sign (Turing used $\underline{|n}$) have been omitted in the commentary, and ten other identifiers are written incorrectly. It would appear to be worth correcting these errors and commenting on the proof from the viewpoint of subsequent work on program proofs.

Turing delivered this paper in June 1949, at the inaugural conference of the EDSAC, the computer at Cambridge University built under the direction of Maurice V. Wilkes. Turing had been writing programs for an electronic computer since the end of 1945—at first for the proposed ACE, the computer project at the National Physical Laboratory, but since October 1948 for the Manchester prototype computer, of which he was deputy director. The references in his paper to 2^{40} are reflections of the 40-bit "lines" of the Manchester machine storage system.

The following is the text of Turing's 1949 paper, corrected to the best of our ability to what we believe Turing intended. We have made no changes in spelling, punctuation, or grammar.

Authors' Addresses: F. L. Morris, School of Computer and Information Science, 313 Link Hall, Syracuse University, Syracuse, NY 13210. C. B. Jones, Department of Computer Science, The University, Manchester MI3 9PL, England.
© 1984 AFIPS 0164-1239/84/020139-143$01.00/00

Turing Text

Friday, 24th June [1949]

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:

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

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

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

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 $\underline{|n}$ without the use of a multiplier, multiplication being carried out by repeated addition.

At a typical moment of the process we have recorded $\underline{|r}$ and $s\underline{|r}$ for some r, s. We can change $s\underline{|r}$ to $(s + 1)\underline{|r}$ by addition of $\underline{|r}$. 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.

F. L. Morris *was graduated in mathematics from Harvard College in 1964 and received the Ph.D. in computer science from Stanford University in 1972. He has been employed at the universities of Essex and Edinburgh, and since 1975 has taught at Syracuse University. The collaboration with C. B. Jones reported here was done while visiting the Programming Research Group, Oxford, during 1980–1981.*

C. B. Jones *began his career working on standard commercial computing projects. A period in operations research was followed by a change to "systems programming." In IBM's Product Test Group the realization dawned that quality could only be achieved in design. The first step toward "formal methods" came with an assignment to the IBM Vienna Laboratory in 1968. A sabbatical in Oxford from 1979–1981 led to the current position of professor of computing science at Manchester University.*

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

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

It is also intended that u be $s\underline{|r}$ or something of the sort e.g. it might be $(s + 1)\underline{|r}$ or $s\underline{|r - 1}$ 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 A and with n in line 29 we shall find a quantity in line 31 when the machine stops which is $\underline{|n}$ (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 $u = \underline{|r}$. Hence $v' = \underline{|r}$ i.e. the entry for v i.e. for line 31 in C should be $\underline{|r}$. 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 $2^{80}(n - r) + 2^{40}(r - s) + 4$. 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.

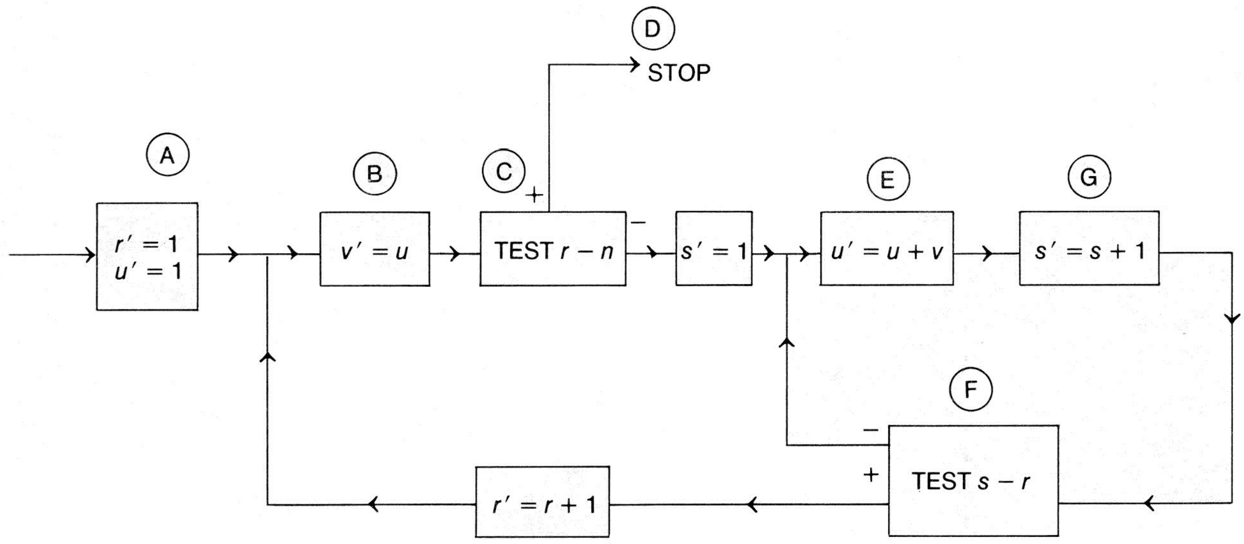

Figure 1 (Redrawn from Turing's original)

Conference Discussion *(from page 70 of the conference report)*

Prof. Hartree said that he thought that Dr Turing had used the terms "induction" and "inductive variable" in a misleading sense since to most mathematicians induction would suggest "mathematical induction" whereas the process so called by von Neumann and Turing often consisted of repetition without logical connection. Prof. Newman suggested that the term "recursive variable" should be used. Dr Turing, however, still thought that his original terminology could be justified.

Comments

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

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		$\lfloor r$	$\lfloor r$		$s\lfloor r$	$(s + 1)\lfloor r$	$(s + 1)\lfloor r$
31			$\lfloor r$	$\lfloor n$	$\lfloor r$	$\lfloor r$	$\lfloor r$
	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)

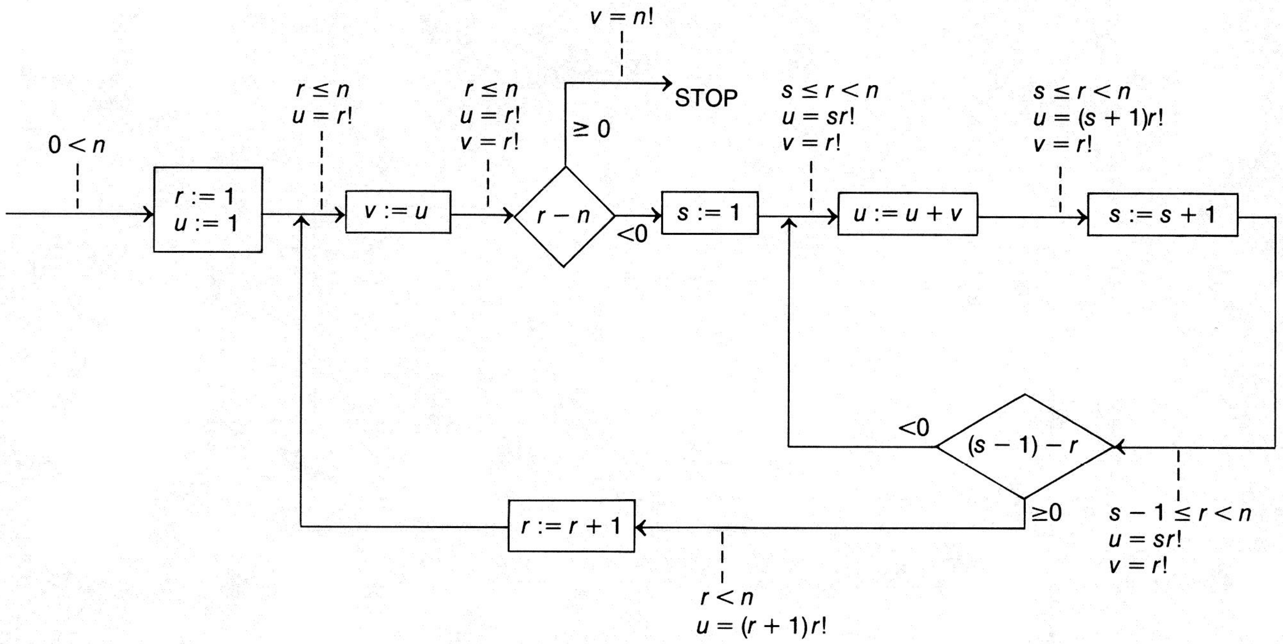

Figure A

other, whereas the format of Figure 2 tends to restrict one to giving an explicit expression for the value of each variable of interest. Thus it is possible to express, for example, the inequality $r \le n$, which strictly speaking is necessary for inferring the $v = n!$ claim at D from $v = r!$ (holding at C) and $r \ge n$ (shown by arrival at D from C). (Note that Turing speaks of giving, in the upper part of Figure 2, "restrictions on the quantities s, r"; these do not appear, however.)

2. In Figure 1 the contents of the individual boxes (e.g., "$r' = r + 1$") are best regarded as specifications to be met by coding: "achieve that r on exit is one more than r on entry." The corresponding assignment statement in Figure A ("$r := r + 1$") is to be thought of as a directly executable statement; the level of necessary representation of quantities and implementation of operations lying below the atomic statements of Figure A is entirely ignored. In particular, the Floyd notation makes no use of primed variables; every use of a variable in an expression, whether in a box or in an assertion, is to be understood as referring to the current value.

The most striking discrepancy between the two versions of the flowchart arises form this last point. Turing chooses to regard the box at G ("$s' = s + 1$") as having no effect on the values of his variables, but instead as causing location 27 to contain $s + 1$ in place of s, an outcome that in Floyd's notation one would have no means of expressing. As is clear from the remarks in Figure 2, the test at F is meant to compare r with the unincremented value of s. Just how this test is to be implemented, s being no longer the contents of any location, is presumably left to the coder's ingenuity.

Turing's convention here—that the increase of s need not coincide with execution of the box "$s' = s + 1$"—cannot be regarded as happily chosen; indeed, the notation of Figure 1 must probably be considered as potentially ambiguous standing on its own, because there seems to be no clear rule about when the addition of a prime to a letter makes a difference. We conjecture, however, that the flow diagram (Figure 1) was drawn just for the occasion, because "there is no coding system sufficiently generally known," and that what Turing had in mind to be passed between the programmer and the checker was the actual code of a routine, marked with letters A, B, . . . , together with an equivalent of Figure 2. There would then be no appearance of inconsistency between the code corresponding to box G, incrementing the contents of location 27, and the behavior of the variable s, belonging solely to the assertions, which increased—as might seem more natural to the programmer—at the point of closure of the loop it controlled.

An additional, minor, remark on the proof concerns the intended domain of the program. It would appear to compute factorial zero correctly, but the assertions are not framed so as to prove this. The necessary changes would be tedious but not difficult. In Figure A we have made explicit the assumption that n is strictly positive.

One final observation on Turing's style of proof concerns his termination argument. To facilitate the simultaneous handling of both loops, what is in effect lexicographic ordering between ordered pairs of expressions is used. In the style of Floyd (1967), the termination of the loops would be considered separately.

In spite of the early date of Turing's paper and the insight shown, we know of no evidence that the paper influenced the later contributors to the ideas of program proofs.

Acknowledgments

Martin Campbell-Kelly, Andrew Hodges, Bill Wadge, Stanley Winkler, and Mike Woodger provided valuable suggestions on the presentation of this paper.

REFERENCES

Dijkstra, E. W. 1976. *A Discipline of Programming.* Englewood Cliffs, N.J., Prentice-Hall.

Floyd, R. W. 1967. "Assigning Meaning to Programs." *Proc. of Symposia in Appl. Math. 19.* (Also in S. T. Schwartz (ed.), *Mathematical Aspects of Computer Science*, Providence, American Mathematical Society, 1967.)

Goldstine, H. H., and J. von Neumann. 1947. "Planning and Coding of Problems for an Electronic Computing Instrument." Report of U.S. Ord. Dept. In A. Taub (ed.), *Collected Works* of J. von Neumann, New York, Pergamon, Vol. 5, 1965, pp. 80–151.

Hoare, C. A. R. October 1969. An axiomatic basis for computer programming. *Comm. ACM 12*, 10, 576–580.

McCarthy, J. 1963. "A Basis for a Mathematical Theory of Computation." In P. Braffort and D. Hirschberg (eds.), *Computer Programming and Formal Systems*, Amsterdam, North-Holland, 1967, pp. 33–70.

Naur, P. 1966. Proof of algorithms by general snapshots. *BIT 6*, 4, 310–316.

Turing, A. M. 1949. "Checking a Large Routine." In *Report of a Conference on High Speed Automatic Calculating Machines*, Univ. Math. Lab., Cambridge, pp. 67–69.