DOI:10.1145/1562764.1562779

C.A.R. Hoare

Viewpoint

Retrospective: An Axiomatic Basis for Computer Programming

C.A.R. Hoare revisits his past Communications *article on the axiomatic approach to programming and uses it as a touchstone for the future.*

THIS MONTH MARKS the 40th anniversary of the publication of the first article I wrote as an academic.[a] I have been invited to give my personal view of the advances that have been made in the subject since then, and the further advances that remain to be made. Which of them did I expect, and which of them surprised me?

Retrospective (1969–1999)

My first job (1960–1968) was in the computer industry; and my first major project was to lead a team that implemented an early compiler for ALGOL 60. Our compiler was directly structured on the syntax of the language, so elegantly and so rigorously formalized as a context-free language. But the semantics of the language was even more important, and that was left informal in the language definition. It occurred to me that an elegant formalization might consist of a collection of axioms, similar to those introduced by Euclid to formalize the science of land measurement. My hope was to find axioms that would be strong enough to enable programmers to discharge their responsibility to write correct and efficient programs. Yet I wanted them to be weak enough to permit a variety of efficient implementation strategies, suited to the particular characteristics of the widely varying hardware architectures prevalent at the time.

I expected that research into the axiomatic method would occupy me for my entire working life; and I expected that its results would not find widespread practical application in industry until after I reached retirement age. These expectations led me in 1968 to move from an industrial to an academic career. And when I retired in 1999, both the positive and the negative expectations had been entirely fulfilled.

The main attraction of the axiomatic method was its potential provision of an objective criterion of the quality of

C.A.R. Hoare attending the NATO Software Engineering Techniques Conference in 1969.

PHOTOGRAPH BY ROBERT M. McCLURE

a Hoare, C.A.R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct. 1969), 576–580.

a programming language, and the ease with which programmers could use it. For this reason, I appealed to academic researchers engaged in programming language design to help me in the research. The latest response comes from hardware designers, who are using axioms in anger (and for the same reasons as given above) to define the properties of modern multicore chips with weak memory consistency.

One thing I got spectacularly wrong. I could see that programs were getting larger, and I thought that testing would be an increasingly ineffective way of removing errors from them. I did not realize that the success of tests is that they test the programmer, not the program. Rigorous testing regimes rapidly persuade error-prone programmers (like me) to remove themselves from the profession. Failure in test immediately punishes any lapse in programming concentration, and (just as important) the failure count enables implementers to resist management pressure for premature delivery of unreliable code. The experience, judgment, and intuition of programmers who have survived the rigors of testing are what make programs of the present day useful, efficient, and (nearly) correct. Formal methods for achieving correctness must support the intuitive judgment of programmers, not replace it.

My basic mistake was to set up proof in opposition to testing, where in fact both of them are valuable and mutually supportive ways of accumulating evidence of the correctness and serviceability of programs. As in other branches of engineering, it is the responsibility of the individual software engineer to use all available and practicable methods, in a combination adapted to the needs of a particular project, product, client, or environment. The best contribution of the scientific researcher is to extend and improve the methods available to the engineer, and to provide convincing evidence of their range of applicability. Any more direct advocacy of personal research results actually excites resistance from the engineer.

Progress (1999–2009)

On retirement from University, I accepted a job offer from Microsoft Research in Cambridge (England). I was surprised to discover that assertions, sprinkled more or less liberally in the program text, were used in development practice, not to prove correctness of programs, but rather to help detect and diagnose programming errors. They are evaluated at runtime during overnight tests, and indicate the occurrence of any error as close as possible to the place in the program where it actually occurred. The more expensive assertions were removed from customer code before delivery. More recently, the use of assertions as contracts between one module of program and another has been incorporated in Microsoft implementations of standard programming languages. This is just one example of the use of formal methods in debugging, long before it becomes possible to use them in proof of correctness.

I did not realize that the success of tests is that they test the programmer, not the program.

In 1969, my proof rules for programs were devised to extract easily from a well-asserted program the mathematical 'verification conditions', the proof of which is required to establish program correctness. I expected that these conditions would be proved by the reasoning methods of standard logic, on the basis of standard axioms and theories of discrete mathematics. What has happened in recent years is exactly the opposite of this, and even more interesting. New branches of applied discrete mathematics have been developed to formalize the programming concepts that have been introduced since 1969 into standard programming languages (for example, objects, classes, heaps, pointers). New forms of algebra have been discovered for application to distributed, concurrent, and communicating processes. New forms of modal logic and abstract domains, with carefully restricted expressive power, have been invented to simplify human and mechanical reasoning about programs. They include the dynamic logic of actions, temporal logic, linear logic, and separation logic. Some of these theories are now being reused in the study of computational biology, genetics, and sociology.

Equally spectacular (and to me unexpected) progress has been made in the automation of logical and mathematical proof. Part of this is due to Moore's Law. Since 1969, we have seen steady exponential improvements in computer capacity, speed, and cost, from megabytes to gigabytes, and from megahertz to gigahertz, and from megabucks to kilobucks. There has been also at least a thousand-fold increase in the efficiency of algorithms for proof discovery and counterexample (test case) generation. Crudely multiplying these factors, a trillion-fold improvement has brought us over a tipping point, at which it has become easier (and certainly more reliable) for a researcher in verification to use the available proof tools than not to do so. There is a prospect that the activities of a scientific user community will give back to the tool-builders a wealth of experience, together with realistic experimental and competition material, leading to yet further improvements of the tools.

For many years I used to speculate about the eventual way in which the results of research into verification might reach practical application. A general belief was that some accident or series of accidents involving loss of life, perhaps followed by an expensive suit for damages, would persuade software managers to consider the merits of program verification.

This never happened. When a bug occurred, like the one that crashed the maiden flight of the Ariane V spacecraft in 1996, the first response of the manager was to intensify the test regimes, on the reasonable grounds that if the erroneous code had been exercised on test, it would have been easily corrected before launch. And if the issue ever came to court, the defense of 'state-of-the-art' practice would always prevail. It was clearly a mistake to try to frighten people into changing their ways. Far more effective is the incentive of reduction in cost. A recent report from the U.S. Department of Commerce has suggested that the cost of programming error to the world economy is measured in tens

of billions of dollars per year, most of it falling (in small but frequent doses) on the users of software rather than on the producers.

The phenomenon that triggered interest in software verification from the software industry was totally unpredicted and unpredictable. It was the attack of the hacker, leading to an occasional shutdown of worldwide commercial activity, costing an estimated $4 billion on each occasion. A hacker exploits vulnerabilities in code that no reasonable test strategy could ever remove (perhaps by provoking race conditions, or even bringing dead code cunningly to life). The only way to reach these vulnerabilities is by automatic analysis of the text of the program itself. And it is much cheaper, whenever possible, to base the analysis on mathematical proof, rather than to deal individually with a flood of false alarms. In the interests of security and safety, other industries (automobile, electronics, aerospace) are also pioneering the use of formal tools for programming. There is now ample scope for employment of formal methods researchers in applied industrial research.

Prospective (2009–)

In 1969, I was afraid industrial research would dispose such vastly superior resources that the academic researcher would be well advised to withdraw from competition and move to a new area of research. But again, I was wrong. Pure academic research and applied industrial research are complementary, and should be pursued concurrently and in collaboration. The goal of industrial research is (and should always be) to pluck the 'low-hanging fruit'; that is, to solve the easiest parts of the most prevalent problems, in the particular circumstances of here and now. But the goal of the pure research scientist is exactly the opposite: it is to construct the most general theories, covering the widest possible range of phenomena, and to seek certainty of knowledge that will endure for future generations. It is to avoid the compromises so essential to engineering, and to seek ideals like accuracy of measurement, purity of materials, and correctness of programs, far beyond the current perceived needs of industry or popularity in the marketplace. For this reason, it is only scientific research that can prepare mankind for the unknown unknowns of the forever uncertain future.

The phenomenon that triggered interest in software verification from the software industry was totally unpredicted and unpredictable.

So I believe there is now a better scope than ever for pure research in computer science. The research must be motivated by curiosity about the fundamental principles of computer programming, and the desire to answer the basic questions common to all branches of science: what does this program do; how does it work; why does it work; and what is the evidence for believing the answers to all these questions? We know in principle how to answer them. It is the specifications that describes what a program does; it is assertions and other internal interface contracts between component modules that explain how it works; it is programming language semantics that explains why it works; and it is mathematical and logical proof, nowadays constructed and checked by computer, that ensures mutual consistency of specifications, interfaces, programs, and their implementations.

There are grounds for hope that progress in basic research will be much faster than in the early days. I have already described the vastly broader theories that have been proposed to understand the concepts of modern programming. I have welcomed the enormous increase in the power of automated tools for proof. The remaining opportunity and obligation for the scientist is to conduct convincing experiments, to check whether the tools, and the theories on which they are based, are adequate to cover the vast range of programs, design patterns, languages, and applications of today's computers. Such experiments will often be the rational reengineering of existing realistic applications. Experience gained in the experiments is expected to lead to revisions and improvements in the tools, and in the theories on which the tools were based. Scientific rivalry between experimenters and between tool builders can thereby lead to an exponential growth in the capabilities of the tools and their fitness to purpose. The knowledge and understanding gained in worldwide long-term research will guide the evolution of sophisticated design automation tools for software, to match the design automation tools routinely available to engineers of other disciplines.

The End

No exponential growth can continue forever. I hope progress in verification will not slow down until our programming theories and tools are adequate for all existing applications of computers, and for supporting the continuing stream of innovations that computers make possible in all aspects of modern life. By that time, I hope the phenomenon of programming error will be reduced to insignificance: computer programming will be recognized as the most reliable of engineering disciplines, and computer programs will be considered the most reliable components in any system that includes them.

Even then, verification will not be a panacea. Verification technology can only work against errors that have been accurately specified, with as much accuracy and attention to detail as all other aspects of the programming task. There will always be a limit at which the engineer judges that the cost of such specification is greater than the benefit that could be obtained from it; and that testing will be adequate for the purpose, and cheaper. Finally, verification cannot protect against errors in the specification itself. All these limits can be freely acknowledged by the scientist, with no reduction in enthusiasm for pushing back the limits as far as they will go. C

C.A.R. Hoare (thoare@microsoft.com) is a principal researcher at Microsoft Research in Cambridge, U.K., and Emeritus Professor of Computing at Oxford University.

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.

KEY WORDS AND PHRASES: axiomatic method, theory of programming' proofs of programs, formal language definition, programming language design, machine-independent programming, program documentation
CR CATEGORY: 4.0, 4.21, 4.22, 5.20, 5.21, 5.23, 5.24

1. Introduction

Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language. For illustrative purposes, this paper is confined to a very simple language, which is effectively a subset of all current procedure-oriented languages.

2. Computer Arithmetic

The first requirement in valid reasoning about a program is to know the properties of the elementary operations which it invokes, for example, addition and multiplication of integers. Unfortunately, in several respects computer arithmetic is not the same as the arithmetic familiar to mathematicians, and it is necessary to exercise some care in selecting an appropriate set of axioms. For example, the axioms displayed in Table I are rather a small selection of axioms relevant to integers. From this incomplete set of axioms it is possible to deduce such simple theorems as:

$$x = x + y \times 0$$

$$y \leqslant r \supset r + y \times q = (r - y) + y \times (1 + q)$$

The proof of the second of these is:

A5 $(r - y) + y \times (1 + q)$

$= (r - y) + (y \times 1 + y \times q)$

A9 $= (r - y) + (y + y \times q)$

A3 $= ((r - y) + y) + y \times q$

A6 $= r + y \times q$ provided $y \leqslant r$

The axioms A1 to A9 are, of course, true of the traditional infinite set of integers in mathematics. However, they are also true of the finite sets of "integers" which are manipulated by computers provided that they are confined to *nonnegative* numbers. Their truth is independent of the size of the set; furthermore, it is largely independent of the choice of technique applied in the event of "overflow"; for example:

(1) Strict interpretation: the result of an overflowing operation does not exist; when overflow occurs, the offending program never completes its operation. Note that in this case, the equalities of A1 to A9 are strict, in the sense that both sides exist or fail to exist together.

(2) Firm boundary: the result of an overflowing operation is taken as the maximum value represented.

(3) Modulo arithmetic: the result of an overflowing operation is computed modulo the size of the set of integers represented.

These three techniques are illustrated in Table II by addition and multiplication tables for a trivially small model in which 0, 1, 2, and 3 are the only integers represented.

It is interesting to note that the different systems satisfying axioms A1 to A9 may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, infinite arithmetic satisfies the axiom:

$A10_I$ $\neg \exists x \forall y \quad (y \leqslant x)$,

where all finite arithmetics satisfy:

$A10_F$ $\forall x \quad (x \leqslant \max)$

where "max" denotes the largest integer represented.

Similarly, the three treatments of overflow may be distinguished by a choice of one of the following axioms relating to the value of max + 1:

$A11_S$	$\neg \exists x \quad (x = \max + 1)$	(strict interpretation)
$A11_B$	$\max + 1 = \max$	(firm boundary)
$A11_M$	$\max + 1 = 0$	(modulo arithmetic)

Having selected one of these axioms, it is possible to use it in deducing the properties of programs; however,

* Department of Computer Science

TABLE I

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	$x \times 1 = x$	

TABLE II

1. Strict Interpretation

+	0	1	2	3
0	0	1	2	3
1	1	2	3	*
2	2	3	*	*
3	3	*	*	*

×	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	*	*
3	0	3	*	*

* nonexistent

2. Firm Boundary

+	0	1	2	3
0	0	1	2	3
1	1	2	3	3
2	2	3	3	3
3	3	3	3	3

×	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	3
3	0	3	3	3

3. Modulo Arithmetic

+	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

×	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

these properties will not necessarily obtain, unless the program is executed on an implementation which satisfies the chosen axiom.

3. Program Execution

As mentioned above, the purpose of this study is to provide a logical basis for proofs of the properties of a program. One of the most important properties of a program is whether or not it carries out its intended function. The intended function of a program, or part of a program, can be specified by making general assertions about the values which the relevant variables will take *after* execution of the program. These assertions will usually not ascribe particular values to each variable, but will rather specify certain general properties of the values and the relationships holding between them. We use the normal notations of mathematical logic to express these assertions, and the familiar rules of operator precedence have been used wherever possible to improve legibility.

In many cases, the validity of the results of a program (or part of a program) will depend on the values taken by the variables before that program is initiated. These initial preconditions of successful use can be specified by the same type of general assertion as is used to describe the results obtained on termination. To state the required connection between a precondition (P), a program (Q) and a description of the result of its execution (R), we introduce a new 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." If there are no preconditions imposed, we write **true** $\{Q\}R$.[1]

The treatment given below is essentially due to Floyd [8] but is applied to texts rather than flowcharts.

3.1. Axiom of Assignment

Assignment is undoubtedly the most characteristic feature of programming a digital computer, and one that most clearly distinguishes it from other branches of mathematics. It is surprising therefore that the axiom governing our reasoning about assignment is quite as simple as any to be found in elementary logic.

Consider the assignment statement:

$$x := f$$

where

x is an identifier for a simple variable;

f is an expression of a programming language without side effects, but possibly containing x.

Now any assertion $P(x)$ which is to be true of (the value of) x *after* the assignment is made must also have been true of (the value of) the expression f, taken *before* the assignment is made, i.e. with the old value of x. Thus if $P(x)$ is to be true after the assignment, then $P(f)$ must be true before the assignment. This fact may be expressed more formally:

D0 Axiom of Assignment

$\vdash P_0\{x := f\}P$

where

x is a variable identifier;

f is an expression;

P_0 is obtained from P by substituting f for all occurrences of x.

It may be noticed that D0 is not really an axiom at all, but rather an axiom schema, describing an infinite set of axioms which share a common pattern. This pattern is described in purely syntactic terms, and it is easy to check whether any finite text conforms to the pattern, thereby qualifying as an axiom, which may validly appear in any line of a proof.

[1] If this can be proved in our formal system, we use the familiar logical symbol for theoremhood: $\vdash P\{Q\}R$

3.2. Rules of Consequence

In addition to axioms, a deductive science requires at least one rule of inference, which permits the deduction of new theorems from one or more axioms or theorems already proved. A rule of inference takes the form "If $\vdash X$ and $\vdash Y$ then $\vdash Z$", i.e. if assertions of the form X and Y have been proved as theorems, then Z also is thereby proved as a theorem. The simplest example of an inference rule states that if the execution of a program Q ensures the truth of the assertion R, then it also ensures the truth of every assertion logically implied by R. Also, if P is known to be a precondition for a program Q to produce result R, then so is any other assertion which logically implies P. These rules may be expressed more formally:

D1 Rules of Consequence

If $\vdash P\{Q\}R$ and $\vdash R \supset S$ then $\vdash P\{Q\}S$

If $\vdash P\{Q\}R$ and $\vdash S \supset P$ then $\vdash S\{Q\}R$

3.3. Rule of Composition

A program generally consists of a sequence of statements which are executed one after another. The statements may be separated by a semicolon or equivalent symbol denoting procedural composition: $(Q_1 ; Q_2 ; \cdots ; Q_n)$. In order to avoid the awkwardness of dots, it is possible to deal initially with only two statements $(Q_1 ; Q_2)$, since longer sequences can be reconstructed by nesting, thus $(Q_1 ; (Q_2 ; (\cdots (Q_{n-1} ; Q_n) \cdots)))$. The removal of the brackets of this nest may be regarded as convention based on the associativity of the ";-operator", in the same way as brackets are removed from an arithmetic expression $(t_1 + (t_2 + (\cdots (t_{n-1} + t_n) \cdots)))$.

The inference rule associated with composition states that if the proven result of the first part of a program is identical with the precondition under which the second part of the program produces its intended result, then the whole program will produce the intended result, provided that the precondition of the first part is satisfied.

In more formal terms:

D2 Rule of Composition

If $\vdash P\{Q_1\}R_1$ and $\vdash R_1\{Q_2\}R$ then $\vdash P\{(Q_1 ; Q_2)\}R$

3.4. Rule of Iteration

The essential feature of a stored program computer is the ability to execute some portion of program (S) repeatedly until a condition (B) goes false. A simple way of expressing such an iteration is to adapt the Algol 60 **while** notation:

$$\textbf{while } B \textbf{ do } S$$

In executing this statement, a computer first tests the condition B. If this is false, S is omitted, and execution of the loop is complete. Otherwise, S is executed and B is tested again. This action is repeated until B is found to be false. The reasoning which leads to a formulation of an inference rule for iteration is as follows. Suppose P to be an assertion which is always true on completion of S, provided that it is also true on initiation. Then obviously P will still be true after any number of iterations of the statement S (even no iterations). Furthermore, it is known that the controlling condition B is false when the iteration finally terminates. A slightly more powerful formulation is possible in light of the fact that B may be assumed to be true on initiation of S:

D3 Rule of Iteration

If $\vdash P \wedge B\{S\}P$ then $\vdash P\{\textbf{while } B \textbf{ do } S\} \neg B \wedge P$

3.5. Example

The axioms quoted above are sufficient to construct the proof of properties of simple programs, for example, a routine intended to find the quotient q and remainder r obtained on dividing x by y. All variables are assumed to range over a set of nonnegative integers conforming to the axioms listed in Table I. For simplicity we use the trivial but inefficient method of successive subtraction. The proposed program is:

$$((r := x;\ \ q := 0);\ \ \textbf{while } y \leqslant r \textbf{ do } (r := r - y;\ \ q := 1 + q))$$

An important property of this program is that when it terminates, we can recover the numerator x by adding to the remainder r the product of the divisor y and the quotient q (i.e. $x = r + y \times q$). Furthermore, the remainder is less than the divisor. These properties may be expressed formally:

$$\textbf{true } \{Q\}\ \neg y \leqslant r \wedge x = r + y \times q$$

where Q stands for the program displayed above. This expresses a necessary (but not sufficient) condition for the "correctness" of the program.

A formal proof of this theorem is given in Table III. Like all formal proofs, it is excessively tedious, and it would be fairly easy to introduce notational conventions which would significantly shorten it. An even more powerful method of reducing the tedium of formal proofs is to derive general rules for proof construction out of the simple rules accepted as postulates. These general rules would be shown to be valid by demonstrating how every theorem proved with their assistance could equally well (if more tediously) have been proved without. Once a powerful set of supplementary rules has been developed, a "formal proof" reduces to little more than an informal indication of how a formal proof could be constructed.

4. General Reservations

The axioms and rules of inference quoted in this paper have implicitly assumed the absence of side effects of the evaluation of expressions and conditions. In proving properties of programs expressed in a language permitting side effects, it would be necessary to prove their absence in each case before applying the appropriate proof technique. If the main purpose of a high level programming language is to assist in the construction and verification of correct programs, it is doubtful whether the use of functional notation to call procedures with side effects is a genuine advantage.

Another deficiency in the axioms and rules quoted above

is that they give no basis for a proof that a program successfully terminates. Failure to terminate may be due to an infinite loop; or it may be due to violation of an implementation-defined limit, for example, the range of numeric operands, the size of storage, or an operating system time limit. Thus the notation "$P\{Q\}R$" should be interpreted "provided that the program successfully terminates, the properties of its results are described by R." It is fairly easy to adapt the axioms so that they cannot be used to predict the "results" of nonterminating programs; but the actual use of the axioms would now depend on knowledge of many implementation-dependent features, for example, the size and speed of the computer, the range of numbers, and the choice of overflow technique. Apart from proofs of the avoidance of infinite loops, it is probably better to prove the "conditional" correctness of a program and rely on an implementation to give a warning if it has had to abandon execution of the program as a result of violation of an implementation limit.

Finally it is necessary to list some of the areas which have not been covered: for example, real arithmetic, bit and character manipulation, complex arithmetic, fractional arithmetic, arrays, records, overlay definition, files, input/output, declarations, subroutines, parameters, recursion, and parallel execution. Even the characterization of integer arithmetic is far from complete. There does not appear to be any great difficulty in dealing with these points, provided that the programming language is kept simple. Areas which do present real difficulty are labels and jumps, pointers, and name parameters. Proofs of programs which made use of these features are likely to be elaborate, and it is not surprising that this should be reflected in the complexity of the underlying axioms.

TABLE III

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)

NOTES

1. The left hand column is used to number the lines, and the right hand column to justify each line, by appealing to an axiom, a lemma or a rule of inference applied to one or two previous lines, indicated in brackets. Neither of these columns is part of the formal proof. For example, line 2 is an instance of the axiom of assignment (D0); line 12 is obtained from lines 5 and 11 by application of the rule of composition (D2).
2. Lemma 1 may be proved from axioms A7 and A8.
3. Lemma 2 follows directly from the theorem proved in Sec. 2.

5. Proofs of Program Correctness

The most important property of a program is whether it accomplishes the intentions of its user. If these intentions can be described rigorously by making assertions about the values of variables at the end (or at intermediate points) of the execution of the program, then the techniques described in this paper may be used to prove the correctness of the program, provided that the implementation of the programming language conforms to the axioms and rules which have been used in the proof. This fact itself might also be established by deductive reasoning, using an axiom set which describes the logical properties of the hardware circuits. 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 the electronics.

The practice of supplying proofs for nontrivial programs will not become widespread until considerably more powerful proof techniques become available, and even then will not be easy. But the practical advantages of program proving will eventually outweigh the difficulties, in view of the increasing costs of programming error. At present, the method which a programmer uses to convince himself of the correctness of his program is to try it out in particular cases and to modify it if the results produced do not correspond to his intentions. After he has found a reasonably wide variety of example cases on which the program seems to work, he believes that it will always work. The time spent in this program testing is often more than half the time spent on the entire programming project; and with a realistic costing of machine time, two thirds (or more) of the cost of the project is involved in removing errors during this phase.

The cost of removing errors discovered after a program has gone into use is often greater, particularly in the case of items of computer manufacturer's software for which a large part of the expense is borne by the user. And finally, 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 interests of academic respectability, but a serious recommendation for the reduction of the costs associated with programming error.

The practice of proving programs is likely to alleviate some of the other problems which afflict the computing world. For example, there is the problem of program documentation, which is essential, firstly, to inform a potential user of a subroutine how to use it and what it accomplishes, and secondly, to assist in further development when it becomes necessary to update a program to meet changing circumstances or to improve it in the light of increased knowledge. The most rigorous method of formulating the purpose of a subroutine, as well as the conditions of its proper use, is to make assertions about the values of variables before and after its execution. The proof of the correctness of these assertions can then be used as a lemma in the proof of any program which calls the subroutine. Thus, in a large program, the structure of the whole can be clearly mirrored in the structure of its proof. Furthermore, when it becomes necessary to modify a program, it will always be valid to replace any subroutine by another which satisfies the same criterion of correctness. Finally, when examining the detail of the algorithm, it seems probable that the proof will be helpful in explaining not only *what* is happening but *why*.

Another problem which can be solved, insofar as it is soluble, by the practice of program proofs is that of transferring programs from one design of computer to another. Even when written in a so-called machine-independent programming language, many large programs inadvertently take advantage of some machine-dependent property of a particular implementation, and unpleasant and expensive surprises can result when attempting to transfer it to another machine. However, presence of a machine-dependent feature will always be revealed in advance by the failure of an attempt to prove the program from machine-independent axioms. The programmer will then have the choice of formulating his algorithm in a machine-independent fashion, possibly with the help of environment enquiries; or if this involves too much effort or inefficiency, he can deliberately construct a machine-dependent program, and rely for his proof on some machine-dependent axiom, for example, one of the versions of A11 (Section 2). In the latter case, the axiom must be explicitly quoted as one of the preconditions of successful use of the program. The program can still, with complete confidence, be transferred to any other machine which happens to satisfy the same machine-dependent axiom; but if it becomes necessary to transfer it to an implementation which does not, then all the places where changes are required will be clearly annotated by the fact that the proof at that point appeals to the truth of the offending machine-dependent axiom.

Thus the practice of proving programs would seem to lead to solution of three of the most pressing problems in software and programming, namely, reliability, documentation, and compatibility. 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.

6. Formal Language Definition

A high level programming language, such as ALGOL, FORTRAN, or COBOL, is usually intended to be implemented on a variety of computers of differing size, configuration, and design. It has been found a serious problem to define these languages with sufficient rigour to ensure compatibility among all implementors. Since the purpose of compatibility is to facilitate interchange of programs expressed in the language, one way to achieve this would be to insist that all implementations of the language shall "satisfy" the axioms and rules of inference which underlie proofs of the properties of programs expressed in the language, so that all predictions based on these proofs will be fulfilled, except in the event of hardware failure. In effect, this is equivalent to accepting the axioms and rules of inference as the ultimately definitive specification of the meaning of the language.

Apart from giving an immediate and possibly even provable criterion for the correctness of an implementation, the axiomatic technique for the definition of programming language semantics appears to be like the formal syntax of the ALGOL 60 report, in that it is sufficiently simple to be understood both by the implementor and by the reasonably sophisticated user of the language. It is only by bridging this widening communication gap in a single document (perhaps even provably consistent) that the maximum advantage can be obtained from a formal language definition.

Another of the great advantages of using an axiomatic approach is that axioms offer a simple and flexible technique for leaving certain aspects of a language *undefined*, for example, range of integers, accuracy of floating point, and choice of overflow technique. This is absolutely essential for standardization purposes, since otherwise the language will be impossible to implement efficiently on differing hardware designs. Thus a programming language standard should consist of a set of axioms of universal applicability, together with a choice from a set of supplementary axioms describing the range of choices facing an implementor. An example of the use of axioms for this purpose was given in Section 2.

Another of the objectives of formal language definition is to assist in the design of better programming languages. The regularity, clarity, and ease of implementation of the ALGOL 60 syntax may at least in part be due to the use of an elegant formal technique for its definition. The use of axioms may lead to similar advantages in the area of "semantics," since it seems likely that a language which can

(*Continued on p. 583*)

by Lowe. In addition, we define $F(j) = \sum_{i=1}^{j-1} f(i)$ and write $\lfloor x \rfloor$ for the greatest integer not exceeding x.

In a packed list file, the bucket which contains the first element of list j will have its first

$$F(j) - C\lfloor F(j)/C \rfloor \tag{1}$$

positions occupied by lists $j - 1$, $j - 2$, $\cdots$. For any practical file, when j is not a small integer, (1) behaves as a random variable uniformly distributed between 0 and C. In other words, the start of a list is independent of bucket boundaries. It is easy to see that the expected number of accesses required to retrieve list j is $f(j)/C + 1$.

Hence we have

$$T_r = t_s\left\{\sum_{j=1}^{N} (f(j)/C + 1)p(j)\right\},$$
$$\therefore \quad T_r/t_s = 1 + \sum_{j=1}^{N} f(j)p(j)/C, \tag{2}$$

since $\sum_{j=1}^{N} p(j) = 1$. Equation (2) corresponds to (11) in [1].

The assumptions on $f(j)$ and $p(j)$ in [1] may be substituted into (2). The first two assumptions yield

$$T_r/t_s = 1 + S/NC, \tag{3}$$

and the third assumption, for large N, yields approximately

$$T_r/t_s = 1 + (lnN + \gamma)^{-2}\pi^2/6. \tag{4}$$

These equations should be compared with the right-hand inequalities of (13) and (24) in [1].

RECEIVED MARCH 1969; REVISED JUNE 1969

REFERENCES

1. LOWE, THOMAS C. The influence of data-base characteristics and usage on direct-access file organization. *J. ACM 15*, 4 (Oct. 1968), 535–548.

C. A. R. HOARE—*cont'd from page 580*

be described by a few "self-evident" axioms from which proofs will be relatively easy to construct will be preferable to a language with many obscure axioms which are difficult to apply in proofs. Furthermore, axioms enable the language designer to express his general *intentions* quite simply and directly, without the mass of detail which usually accompanies algorithmic descriptions. Finally, axioms can be formulated in a manner largely independent of each other, so that the designer can work freely on one axiom or group of axioms without fear of unexpected interaction effects with other parts of the language.

Acknowledgments. Many axiomatic treatments of computer programming [1, 2, 3] tackle the problem of proving the equivalence, rather than the correctness, of algorithms. Other approaches [4, 5] take recursive functions rather than programs as a starting point for the theory. The suggestion to use axioms for defining the primitive operations of a computer appears in [6, 7]. The importance of program proofs is clearly emphasized in [9], and an informal technique for providing them is described. The suggestion that the specification of proof techniques provides an adequate formal definition of a programming language first appears in [8]. The formal treatment of program execution presented in this paper is clearly derived from Floyd. The main contributions of the author appear to be: (1) a suggestion that axioms may provide a simple solution to the problem of leaving certain aspects of a language undefined; (2) a comprehensive evaluation of the possible benefits to be gained by adopting this approach both for program proving and for formal language definition.

However, the formal material presented here has only an expository status and represents only a minute proportion of what remains to be done. It is hoped that many of the fascinating problems involved will be taken up by others.

RECEIVED NOVEMBER, 1968; REVISED MAY, 1969

REFERENCES

1. YANOV, YU I. Logical operator schemes. *Kybernetika 1*, (1958).
2. IGARASHI, S. An axiomatic approach to equivalence problems of algorithms with applications. Ph.D. Thesis 1964. Rep. Compt. Centre, U. Tokyo, 1968, pp. 1–101.
3. DE BAKKER, J. W. Axiomatics of simple assignment statements. M.R. 94, Mathematisch Centrum, Amsterdam, June 1968.
4. MCCARTHY, J. Towards a mathematical theory of computation. Proc. IFIP Cong. 1962, North Holland Pub. Co., Amsterdam, 1963.
5. BURSTALL, R. Proving properties of programs by structural induction. Experimental Programming Reports: No. 17 DMIP, Edinburgh, Feb. 1968.
6. VAN WIJNGAARDEN, A. Numerical analysis as an independent science. *BIT 6* (1966), 66–81.
7. LASKI, J. Sets and other types. ALGOL Bull. 27, 1968.
8. FLOYD, R. W. Assigning meanings to programs. Proc. Amer. Math. Soc. Symposia in Applied Mathematics, Vol. 19, pp. 19–31.
9. NAUR, P. Proof of algorithms by general snapshots. *BIT 6* (1966), 310–316.