An Axiomatic Basis for Computer Programming

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

October, 1969
Volume 12, Number 10
pp. 576–580

On several occasions, Hoare has written a paper that changes the way people think about some aspect of programming. In this paper he proposed notation for associating assertions to key points in a program so that the proof method pointed out by Floyd in 1967 could be applied in modern programming languages. This paper is the source of the now familiar notation \{P\}S\{Q\}, meaning "if proposition P is true when control is at the beginning of statement S, then proposition Q will be true when control is at the end of statement S." This notation has been refined by others and applied to many other cases of program verification.

—P. J. D.

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: 6.0, 4.21, 4.22, 5.20, 5.21, 5.22, 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 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 involves, 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 1 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:

\[ x = z + y \times 0 \]
\[ y \leq r \Rightarrow r + y \times q = (r - y) + y \times (1 + q) \]

The proof of the second of these is:

\[ A5 \quad (r - y) + y \times (1 + q) = (r - y) + (y \times 1 + y \times q) \]
\[ A9 \quad = (r - y) + (y + y \times q) \]
\[ A8 \quad = (r - y) + y \times q \]
\[ A6 \quad = r + y \times q \quad \text{provided} \quad y \leq r \]

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) Static 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 if not 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:

\[ A10. \forall x (y < x) \]

where all finite arithmetics satisfy:

\[ A10. \forall x \forall y (x < y) \]

where "\( \forall y \)" 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 \( x + 1 \):

\[ A11a. \forall x \forall y \forall z (x + y = z) \quad \text{(strict interpretation)} \]
\[ A11b. \max + 1 = \max \quad \text{(firm boundary)} \]
\[ A11c. \max + 1 = 0 \quad \text{(modulo arithmetic)} \]

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

| A1 | x + y = y + x | addition is commutative |
| A2 | x * y = y * x | multiplication is commutative |
| A3 | (x + y) + z = x + (y + z) | addition is associative |
| A4 | (x * y) * z = x * (y * z) | multiplication is associative |
| A5 | x * (y + z) = x * y + x * z | multiplication distributes through addition |
| A6 | x - x = 0 | addition cancels subtraction |

| A7 | x + 0 = x |
| A8 | x * 1 = x |
| A9 | x * 0 = 0 |

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

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 specify particular values for each variable but will 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 operational 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 success must 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 (B), we introduce a new notation:

\[ P \Rightarrow Q \, \text{R} \]

This may be interpreted "If the assertion \( P \) is true before initiation of a program \( Q \), then \( Q \) will be true on its completion." If there are no preconditions imposed, we write true \[ Q \, \text{R} \].

The treatments given below are essentially due to Floyd [5] and are 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 axioms 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 assignment \( P(x) \) which is to be true of \( (\text{the value of } x) \) after the assignment is made must also be true of \( (\text{the value of } x) \) after 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:

\[
\text{D0 Axiom of Assignment} \quad P(x := f) \Rightarrow P
\]

where \( x \) is a variable identifier; \( f \) is an expression; \( P \) 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.

If this can be proved in our formal system, we use the familiar logical symbol for therefrom: \[ \vdash P \Rightarrow Q \]

3.2. Rules of Consequence

In addition to the deductive rules required, and the familiar rules of operational precedence which have been used wherever possible to improve legibility, there are at least one rule of inference, which permits the derivation 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 \( Z \) is also 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:

\[
\text{D1 Rule of Consequence} \quad \vdash P(Q) \Rightarrow (R \land S) \Rightarrow \vdash P(S)
\]

\[
\text{D2 Rule of Composition} \quad \vdash P(Q) \Rightarrow (R \land S) \Rightarrow \vdash P(R \land S)
\]

3.3. Rules 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: \( Q_1 ; Q_2; \ldots; Q_n \). 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 \( Q_1 ; Q_2 \); \( Q_1 ; (Q_2 ; (Q_3 ; \ldots)) \). The removal of the brackets of this nest may be regarded as convention based on the associativity of the "c-"operator," in the same way as brackets are removed from an arithmetic expression \( a + (b + (c + \ldots)) \).

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:

\[
\text{D3 Rule of Composition} \quad \vdash P(Q) \Rightarrow (R \land S) \Rightarrow \vdash P(Q) \Rightarrow P(R \land S)
\]

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 \( B \) is met. A simple way of expressing such an iteration is to adapt the ALOX 60 while notation:

\[ \text{while } B \text{ do } S \]

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 be true after any number of iterations of \( S \) (even no iterations). Furthermore, it is known that the terminating 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 \( S \).

\[
\text{D4 Rule of Iteration} \quad \vdash P \land R \Rightarrow \vdash P \land \text{while } B \text{ do } S \Rightarrow \neg B \land P
\]

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 a by \( y \). All variables are assumed to range over a set of nonnegative integers congruent to the axioms listed in Table I. For simplicity we use the trivial but inefficient method of successive subtraction. The proposed program is:

\[ \langle r := x ; q := 0 \rangle \text{ while } y \leq r \langle r := r - y ; q := q + 1 \rangle \]

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:

\[ \text{true } \langle Q \rangle \Rightarrow \forall r \leq x \wedge \forall y \times q \]

where \( Q \) stands for the program displayed above. This expresses a necessary (but not sufficient) condition for the "correct" behavior 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 rule 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 data types, the size of storage, or an operating system time limit. Thus the notation "PQRP" should be interpreted as meaning "provided that the program successfully terminates, the property of the results 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 depend on knowledge of many implementation-dependent features, for example, the size and speed of the computer, the number of variables, and the choice of overflow technique. Apart from proofs of the univalence 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, overlays, definition files, input, output, declarations, subroutines, parameters, recursion, and parallel execution. Even the characteristic 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.

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 components. When the implementation is unrealized, 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 powerful, proof techniques become available, and even then will not be easy. But the practical advantages of program proofs will presumably outweigh the difficulties, in view of the increasing costs of programming error. As a result, a "proof" of a program is to try it out in particular cases and to modify it if the results produced do not correspond to its intentions. After he has found a reasonably wide variety of example cases on which the program seems to work, he believes that the program is correct.

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 been taken into use is often far greater. Frequently 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 insurable—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 research and integrity, but a serious practical 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 affect 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 property 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 algorithms, 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 computer, and the user of another computer will be puzzled and general and expensive surprises can result when attempting to transfer it to another machine. However, presence of a machine-dependent property is always 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 inquiries; 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 axioms, for example, one of the versions of AII (Section 2). In the latter case, the axioms must be explicitly quoted as one of the proofs of the correctness of the program.

The program can still with complete confidence, be transferred to any other machine which happens to satisfy the same machine-dependent axioms; but if it becomes necessary to transfer it to an implementation which does not, then all the phases 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 resolution 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 programs of high caliber; and may be applicable only to quite simple programs designed in areas where reliability can be purchased at a reasonable price.
be described by a few "self-evident" axioms from which both 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 to reflect certain generalizations which are quite naturally and directly, without the nuisance of detail which usually accompanies algorithmic descriptions. Finally, axioms can be formulated in a manner largely independent of each other, so that the user can work freely on one axiom or group of axioms without fear of unexpected interaction with other parts of the language.

Acknowledgments. Many axiomatized treatments of computer programs have appeared in [1, 2]. Statements 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 formal proofs is clearly emphasized in [8], 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 prove expression 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 incorporating partial 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 a slight connection with any other system of axioms which remain 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.

EDUCERER November, 1968, revised May, 1969

REFERENCES

2. WANG, H.: A Note on the Number of logical operators, Journal of the Association for Computing Machinery 4, (1957), 399-412.

Letter to the Editor, Vol. 9, No. 4, April 1969, p. 243.

Algorithm and Formula

Editor:
We are making this communication intentionally short to leave as much room as possible for the answers.


T. WANGZSNER J. FRANKLIN

Inход, Beach, California

Algorithm and Formula

Eleven

Letter to the Editor, Vol. 9, No. 9, September 1969, pp. 653-654

Algorithm and Program Information and Data

Editor:
The preceding letter by Dr. Huber defines "algorithm" in terms of programming languages. I would like to take a slightly different point of view, however. As far as I can see, there are essentially two approaches to the definition of an algorithm, and each of these approaches is necessary. I will call one an "algorithmic" and the other a "programmatic" approach. We can say that an algorithm is a program for solving a problem. We can also say that a program is a specification of an algorithm. In this paper, I will concentrate on the algorithmic approach. I will use the term "algorithm" to refer to a description of a problem-solving process. I will use the term "program" to refer to a computer program for the execution of an algorithm.

I will now try to explain how the notion of "algorithm" can be mathematically formulated along these lines. Let us say that a computational method consists of a set Q (finite or infinite) of "states," containing a subset X of "inputs" and a subset Y of "outputs," and a function F from X into Y. (These quantities are usually further restricted to be computable, in some sense that corresponds to what human beings can comprehend.) A computationally feasible method defines a computation for each X in X, as follows: Let q = q, and define f(q) = F(q). If q = q, then define f(q) = Y, (since we can be considered to be a sentence in a formal language in which there is no need for more symbols.) Each such computation is called an algorithm in a formal language. In this way we can consider abstraction algorithms from particular programs. We have used the word "computational" in the same sense as "data processing," "symbol manipulation," or more generally "computation." There seems to be no real difference between the words "information" and "data" much like that between "algorithm" and "program." When a scientist conducts an experiment and he is measuring the value of some quantity, he has four things present, each of which is often called "information:" (a) the true value of the quantity being measured (b) the accuracy of the measuring device (c) the representation of the value in some formal language (d) the process by which the value is computed. It would seem that the word "data" would be most appropriately applied to (c) and the word "information" when used to mean (a) and (b) should be further qualified by stating what kind of information is mean.

DONALD E. KRUTH
California Institute of Technology
Pasadena, California 91109

Algorithm and Program Information and Data

Editor:
The preceding letter by Dr. Huber defines "algorithm" in terms of programming languages. I would like to take a slightly different point of view, however. As far as I can see, there are essentially two approaches to the definition of an algorithm, and each of these approaches is necessary. I will call one an "algorithmic" and the other a "programmatic" approach. We can say that an algorithm is a program for solving a problem. We can also say that a program is a specification of an algorithm. In this paper, I will concentrate on the algorithmic approach. I will use the term "algorithm" to refer to a description of a problem-solving process. I will use the term "program" to refer to a computer program for the execution of an algorithm.

I will now try to explain how the notion of "algorithm" can be mathematically formulated along these lines. Let us say that a computational method consists of a set Q (finite or infinite) of "states," containing a subset X of "inputs" and a subset Y of "outputs," and a function F from X into Y. (These quantities are usually further restricted to be computable, in some sense that corresponds to what human beings can comprehend.) A computationally feasible method defines a computation for each X in X, as follows: Let q = q, and define f(q) = F(q). If q = q, then define f(q) = Y, (since we can be considered to be a sentence in a formal language in which there is no need for more symbols.) Each such computation is called an algorithm in a formal language. In this way we can consider abstraction algorithms from particular programs. We have used the word "computational" in the same sense as "data processing," "symbol manipulation," or more generally "computation." There seems to be no real difference between the words "information" and "data" much like that between "algorithm" and "program." When a scientist conducts an experiment and he is measuring the value of some quantity, he has four things present, each of which is often called "information:" (a) the true value of the quantity being measured (b) the accuracy of the measuring device (c) the representation of the value in some formal language (d) the process by which the value is computed. It would seem that the word "data" would be most appropriately applied to (c) and the word "information" when used to mean (a) and (b) should be further qualified by stating what kind of information is mean.

DONALD E. KRUTH
California Institute of Technology
Pasadena, California 91109