The Axiomatic Basis of Computer Programming

Summary

This paper attempts 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.

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 underly 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 confines itself to a very simple language, which is effectively a subset of all current procedure-oriented languages, and yet is theoretically as powerful as any of them, in the sense of being able to program any computable function.

Line number	Formal Proof	Justification
1.	$\underline{true}\ \{r := x\}\ \exists r_0 (r = x \wedge \underline{true})$	D0
2	$\exists r_0 (r = x \wedge \underline{true})\ \{q := 0\}\ \exists q_0 (q = 0 \wedge \exists r_0 (r = x \wedge \underline{true}))$	D0
3.	$\exists q_0 (q = 0 \wedge \exists r_0 (r = x \wedge \underline{true})) \supset x = r + y \times q$	Lemma 1
4.	$\exists r_0 (r = x \wedge \underline{true})\ \{q := 0\}\ x = r + y \times q$	D1 (2,3)
5.	$\underline{true}\ \{(r := x;\ q := 0)\}\ x = r + y \times q$	D2 (1,4)
6.	$x = r + y \times q \wedge y \leq r\ \{r := r - y\}\ \exists r_0 (r = r_0 - y \wedge x = r_0 + y \times q \wedge y \leq r_0)$	D0
7	$\exists r_0 (r = r_0 - y \wedge x = r_0 + y \times q \wedge y \leq r_0)\ \{q := 1 + q\}\ \exists q_0 (q = 1 + q_0 \wedge \exists r_0 (r = r_0 - y \wedge x = r_0 + y \times q_0 \wedge y \leq r_0))$	D0
8	$\exists q_0 (q = 1 + q_0 \wedge \exists r_0 (r = r_0 - y \wedge x = r_0 + y \times q_0 \wedge y \leq r_0)) \supset x = r + y \times q$	Lemma 2
9	$\exists r_0 (r = r_0 - y \wedge x = r_0 + y \times q \wedge y \leq r_0)\ \{q := 1 + q\}\ x = r + y \times q$	D1 (7,8)
10	$x = r + y \times q \wedge y \leq r\ \{(r := r - y;\ q := 1 + q)\}\ x = r + y \times q$	D2 (6,9)
11.	$x = r + y \times q\ \{\underline{while}\ y \leq r\ \underline{do}\ (r := r - y;\ q := 1 + q)\}\ \neg y \leq r \wedge x = r + y \times q$	D3 (8)
12	$\underline{true}\ \{((r := x;\ q := 1 + q);\ \underline{while}\ y \leq r\ \underline{do}\ (r := r - y;\ q := 1 + q))\}\ \neg y \leq r \wedge x = r + y \times q$	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 1 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 A12 and A13.

3. Lemma 2 follows directly from the theorem proved in section 2.

Table 3.

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 1 is rather a small selection of axioms relevant to integers.

table 1

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	$y\le x\supset(x-y)+y=x$	addition cancels subtraction
A7	$x+0=x$	
A8	$x\times 0=0$	
A9	$x\times 1=x$	

From this incomplete set of axioms it is possible to deduce such simple theorems as:

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

$$y\le r_0\supset r_0+y\times q_0=(r_0-y)+y\times(1+q_0)$$

The proof of the second of these is:

$$\begin{aligned}(r_0-y)+y\times(1+q_0)&=(r_0-y)+(y\times 1+y\times q_0) && \text{A5}\\ &=(r_0-y)+(y+y\times q_0) && \text{A9}\\ &=((r_0-y)+y)+y\times q_0 && \text{A3}\\ &=r_0+y\times q_0 \quad \text{provided } y\le r_0 && \text{A6}\end{aligned}$$

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; and furthermore, it is largely independent of the choice of technique applied in the event of "overflow"; for example:

(1) the 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 2 by addition and multiplication tables for a trivially small model, in which 0, 1, 2, and 3 are the only integers represented.

(1) the 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

table 2

It is interesting to note that in the different systems satisfying axioms A1-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:

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

whereas all finite arithmetics satisfy:

$$A10_F \quad \forall x.\ (x \leq max)$$

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:

$A11_S$	$\neg \exists x\,(x = \max + 1)$	(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, 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 shall use the normal notations of mathematical logic to express these assertions; the only symbols which may be novel to programmers familiar with an ALGOL-like language are the quantifiers:

$\forall x.\, P$ (P is true for all x)

$\exists x.\, P$ (there is an x such that P).

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. In order 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 <u>true</u> $\{Q\}R$.

3.1 Axiom of assignment.

Assignment is undoubtedly the most characteristic feature of programming languages, and one that most clearly distinguishes practical programming from other branches of mathematics. However, the axiom governing assignment is rather complex, and will be introduced here in a gradual fashion.

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.

Let us suppose first of all that the variable x does <u>not</u> enter into the expression f. Then it is obviously true to state:

<u>true</u> $\{x := f\}\ x = f$. (where f does not contain x) (1)

Furthermore, if P is any assertion which does not mention the variable x at all, the assignment to x cannot affect the truth of P, so we get:

$P\{x := f\}\ x = f \wedge P$ (where f and P do not contain x) (2)

Now let us consider an assertion $P(x)$, which <u>does</u> say something about the value of x. Thus when a new assignment is made to x, P will be no longer true of x. However P is still true of the <u>previous</u> value of x, which we shall denote x_0. Thus at least there <u>exists</u> a value satisfying P, and we write:

$P(x)\ \{x := f\}\ \exists x_0 \left(x = f \wedge P(x_0)\right)$ (where f does not contain x) (3)

Finally, if we allow f also to contain occurrences of x, we must remember that in computing $f(x)$, it is the <u>previous</u> value of x which is used. Thus we must change the x in $f(x)$ into x_0, as we did in the case of $P(x)$. Thus the axiom assumes the form

$$P(x)\{x:=f(x)\}\ \exists x_0\left(x=f(x_0)\wedge P(x_0)\right) \qquad (4)$$

It appears that (4) is the most general result obtainable, and that the previous three assertions can be directly deduced from it as special cases. Giving a more formal description of (4) we obtain:

D0 Axiom of Assignment

$$\vdash P\{x:=f\}\ \exists x_0\left(x=f_0\wedge P_0\right)$$

where x is a variable
f is an expression
x_0 is a variable not free in f or P
f_0 and P_0 are obtained from f and P by substituting x_0 for all free 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 very 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.

3.2 Rule 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. This can be more formally expressed:

D1 Rule of Consequence

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

It is this rule which makes it possible to deduce the special cases from the general axiom of assignment.

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: $(Q_1; Q_2; \dots Q_n)$. 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 $(Q_1;(Q_2;(\dots(Q_{n-1};Q_n)\dots)))$. 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 $(t_1+(t_2+(\dots(t_{n-1}+t_n)\dots)))$.

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

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

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 <u>while</u> notation:

<u>while</u> B <u>do</u> 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.

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

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 1. For the sake of simplicity, we use the trivial but inefficient method of successive subtraction. The proposed program is:

$$((r := x;\ q := 0);\ \underline{\text{while}}\ y \le r\ \underline{\text{do}}\ (r := r - y;\ q := 1 + q))$$

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 (ie. $x = r + y \times q$). Furthermore, the remainder is less than the divisor. These properties may be expressed formally:

$$\underline{\text{true}}\ \{Q\}\ \neg\, y \le r \wedge x = r + y \times q$$

where Q stands for the program displayed above.

A formal proof of this theorem is given in table 3. 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.

When stapling, please put the tables at the end of the text.

Table 1

insert 11 multiplication signs

one $\supset$ sign (A6)

The second 1 in A9 should be deleted.

Table 2

insert 3 multiplication signs.

Table 3

line 1 insert . $\wedge$ sign (upside-down v will do)

2 insert 3 $\wedge$ signs

3 insert 2 $\wedge$ signs and X and $\supset$

4 insert $\wedge$ and X

5 insert X

6 insert 3 $\wedge$ signs and 2 X

7. insert 5 $\wedge$ signs and 2 X ; also pull back the 90 on the last line.

8 3 $\wedge$ and 2 X and $\supset$

9 2 $\wedge$ and 2 X

10. 1 $\wedge$ and 2 x

11 1 $\wedge$ and 2 X and one $\neg$; also change 8 to 10

12 1 $\wedge$ and 1 x and one $\neg$

page 1 9 X signs and one $\supset$

page 2 lines 17-19 please move left to the margin. Also capital T for "table"

line 22 delete "in"

line 35 put "c" for "s"

line 27 insert $\neg$ ~~& (A + C_1)~~

line 37 insert $\neg$

page 3 insert 3 $\wedge$ signs

page 4 line 8 insert $\vdash$ and $\wedge$

line 23 should be "text conforms" — ie delete "con" from "context"

line 32 3 $\vdash$ signs

line 41 3 $\vdash$ signs and $\supset$

line 55 insert ", thus:"

page 5 line 7 3 $\vdash$ signs
carry back the "$Q_2)\}R$" from next line.

line 36 two $\vdash$, two $\wedge$ and one $\neg$

53 X

56 $\wedge$ and X

page 7 line 51 insert , at end

line 65 replace "al" by "ot"
it should read "annotated"

page 9 line 15 ~~rule~~ insert "s" after "provide"

Page 9 line 50 to 51

should read

5. R.M. Burstall Proving properties of programs by structural induction.

Experimental Programming Reports: No 17 (Feb 1968)

DMIP Edinburgh.

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 either to prove their absence in each case before applying the appropriate proof technique; or else (before attempting the proof, to) translate all function calls of the program into procedure calls with explicit sequencing. 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 timelimit. 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 non-terminating 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, and recursion. 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 make 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. Even the characterisation of integer arithmetic is far from complete

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 may 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. ~~Even electronic reliability is likely to increase if the engineer no longer has the excuse of program or software error to avoid dealing with occasional errors.~~

The practice of supplying proofs for non-trivial 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.

Even then, it is notorious that [illegible] well tested programs still contain errors. The cost of removing errors discovered after a program has gone into use is often greater, particularly in the case of items of computer manufacturers' 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 literally astronomical — 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 on how to use it, and what it accomplishes, and secondly to assist in further development when it becomes necessary to ~~modify~~ update a program to meet changing circumstances, or to improve it in the light of increasing knowledge. The most rigorous method of formulating the purpose of a subroutine and 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 certainly be difficult even for programmers of high calibre; 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 underly 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, and possibly

In effect, this amounts to accepting the axioms and rules of inference as the ultimately definitive formal specification of the language, [illegible]

~~Apart from giving an immediate, [illegible] this approach to formal language~~ definition is that it seems to be the simplest and most readily comprehensible of all techniques proposed for this purpose. It bears the same relationship to an algorithm for defining or implementing a language as the criterion of program correctness does to a program itself. Thus it is the only known type of description which is equally suitable

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.

In effect, this is equivalent to accepting the axioms and rules of inference as the ultimately definitive specification of the meaning of the language for study by the user as well as the implementor of a language. ~~By bridging this ever widening gap in communication by means of a single provably consistent document, the axiomatic method of programming language definition recommends itself as the ultimate solution to this problem.~~ 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 standardisation 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 is given in section 2.

N.P.

Another of the objectives of formal language definition ~~which may be furthered by the axiomatic approach~~ 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 ~~to express the intentions of a language designer~~ may lead to similar advantages in the area of "semantics", since it seems likely that a language which can 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.

7. Acknowledgements.

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 approach of ~~the present~~ paper most closely follows the ideas of [8,9]. The importance of a mechanism for leaving parts of a language undefined, and the use of axioms to achieve this end, appear to have been first suggested by the present author.

The importance of program proofs is clearly emphasised in [9], and an informal technique for providing them is described. The suggestion that the specification of ~~formal~~ proof techniques provide 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 other more skilled hands.

References:

1. Yu. I Yanov Logical Operator Schemes.
Kybernetica I Moscow 1958

2. S. Igarashi. An Axiomatic Approach to Equivalence Problems of Algorithms with Applications. PhD Thesis 1964.
Rep. Compt. Centre Univ Tokyo 1 (1968) 1-101

3. J.W. de Bakker Axiomatics of simple assignment statements.
M.R. 94 Mathematisch Centrum, Amsterdam. June 1968

4. J. McCarthy Towards a Mathematical Theory of Computation
Proc IFIP Congress 1962, North Holland, 1963.

5. R. Burstall

6. A. van Wijngaarden Numerical Analysis as an Independent Science.
BIT (1966) Vol 6 pp 66-81

7. J. Laski Sets and Other Types
ALGOL Bulletin 27 (1968)

8. R.W. Floyd Assigning Meanings to Programs

9. P. Naur. Proof of Algorithms by General Snapshots
BIT (1966) Vol 6 pp 310-316