[illegible]

THE AXIOMATIC METHOD

Program Execution

DEC 68

C. A. R. HOARE

Summary.

This paper illustrates the manner in which the axiomatic method may be applied to the rigorous definition of a programming language. It deals with the dynamic aspects of the behaviour of a program, which is an aspect considered to be most far removed from traditional mathematics. However, it appears that the axiomatic method not only shows how programming is closely related to traditional branches of logic and mathematics, but also formalises the techniques which may be used to prove the correctness of a program over its intended area of application.

1. Introduction.

Any successful description of a programming language must obviously provide a technique for discovering the meaning of all programs expressed in that language. The meaning of a program may be defined by specifying the effect of executing the program in every possible environment in which it can be run. Thus it appears necessary to give a comprehensive description of every possible environment (machine state), and then to give a method for working out the comprehensive description of the environment that results from executing a given program. The main objections to this technique are as follows:

(1) The environment is usually so large and complex, and the algorithm so time consuming, that the only practical way of finding the meaning of the program is by running it on a computer. Even this is impractical if we wish to examine the effect of the program in every possible environment: or worse, if we have doubts about the accuracy of the implementation.

(2) Unless the programming language is inordinately complicated, it seems that a program itself is much more concise and comprehensibe description of a transformation on an environment than the equivalent treatment embodied in the language definition.

(3) The method does not help in the formulation of proofs that a program will perform correctly in any of the circumstances in which it may be applied.

In this paper, we take a slightly different and possibly more fruitful approach to programming language definition. We do not insist on any complete environmental description either at the beginning or the end of program execution, or at any intermediate stage; rather we permit the programmer himself to describe only those features of the environment which he considers relevant for the successful use of his program. In other words, he will make certain quite general statements about those properties and relationships holding between the values of variables at the time at which the program is invoked; and he will define the purpose of the program by making other general statements about the environment which will result on completion of the program. A person may be regarded as having a "reading" understanding of a programming language if he is able to check whether the result of executing any program of the language will in fact satisfy the claims made by the programmer whenever it is applied in an environment for which the declared preconditions hold. This may be done either by constructing or by checking a proof that this is the case. A person may be regarded as having a "writing" knowledge of the language if he is in general capable of considering the desired properties of the result of executing a program, and constructing a program which can be proved to have the desired result, subject, if necessary, to stated preconditions. If this approach is taken, the meaning of a programming language can be fully specified by describing a method of checking proofs that an arbitrary program written in language satisfies a given set of design objectives.

This approach appears to avoid the drawbacks associated with previous methods. It also appears to offer the promise of eliminating the expense of traditional program testing and the danger of using incorrect programs. Thus it may represent the same magnitude of advance in Computer Science as the axiomatic geometry of Euclid compared with the crude land-measurement of the ancient Egyptians.

2. Basic Features.

In order to describe the preconditions and results of successful use of a program, we will adopt the normal notational conventions of propositional and predicate logic. To express a program, we will use mainly the notations of ALGOL 60. But for the assertion of a relationship between the two, it will be necessary to introduce a new and unfamiliar notation: P{Q}R, where P and R are assertions and Q is a part of a program, will assert that if precondition P is true when Q is initiated, the statement R will be true on successful termination of Q.

If this assertion can be proved as a theorem from the axioms, we write $\vdash P\{Q\}R$.

Most of the formal material in the following sections takes the form of rules of inference rather than axioms. The majority of pure axioms relate to the non-dynamic properties of the simple operations and operands of the language, and have been treated in a previous paper.

The following rules of inference are obvious extensions of the rule of deduction:

1. If $P' \supset P$ and $P\{Q\}R$ then $P'\{Q\}R$

2. If $R \supset R'$ and $P\{Q\}R$ then $P\{Q\}R'$

Furthermore, the following rule is intuitively appealing and useful:

3. If $P\{Q\}R$ and $P'\{Q\}R'$ then $P \wedge P'\{Q\} R \wedge R'$.

2.1 Assignment.

Assignment is undoubtedly the most characteristic feature of programming, and that which distinguishes it most clearly from more traditionally "timeless" branches of mathematics. However, Floyd has shown how it can be treated in a completely static fashion, by a simple logical technique of substitution of free variables. This "freezing" of time-dependent phenomena enables more powerful logical and mathematical techniques to be applied, in the same way as the realisation that a function of time could be regarded as a static completed entity makes it possible to deal with integral equations and boundary problems, in a manner which would be inconceivable to those who regard such a function as definable only in a step-by-step fashion.

Consider the assignment statement $x := f$,

where x is a simple variable identifier
and f is an expression, possibly containing x.

Suppose that an assertion P is true before the assignment is executed; we are interested in finding the strongest statement P' which is true after execution of the assignment. Obviously, if the variable x does not occur at all in the statement P or in the expression f, it is valid to state that

$$P\{x:=f\}x=f \wedge P$$

If the variable x appears (free) in P but not in f, we can remove the possibility of such free occurrences by prefixing P by the existential quantifier $(\exists x)$. Since $P \supset \exists x P$ is a theorem we can state

$$P~~\vdash \exists x.P~~\{x:=f\}\ x=f \wedge \exists x.P$$

Finally, if x occurs free in f (as in the assignment $x:=x+1$) we can perform the assignment in two stages e.g. $t:=f$; $x:=t$, where t is an arbitrary variable which does not occur in f or P.
Consequently, as we have seen above,

$$P\{t:=f\}\ t=f \wedge P \qquad (1)$$

and therefore for the second assignment we get

$$t=f \wedge P\{x:=t\}\ x=t \wedge \exists x,\ t=f \wedge P \qquad (2)$$

But it is a theorem of predicate calculus that

$$x=t \wedge \exists x.t=f \wedge P \supset \exists x_o(x=f_o \wedge P_o) \qquad (3)$$

where f_o and P_o are like f and P except that they contain free occurrences of x_o wherever the latter contain free occurrences of x.

Combining the results (1), (2), (3), it becomes obvious that

$$\vdash P\{x:=f\}\exists x_o(x=f_o \wedge P_o)$$

This statement avoids the awkwardness of introducing the fresh unbound variable t on every assignment, and Floyd has proved that it is the strongest summary of the effects of an assignment statement that can be given.

In order to incorporate this discovery into a formal axiom, it is necessary to adopt the technique of postulating an infinite set of axioms, each of which exhibits the appropriate form. So we obtain

1. $\vdash P\{x:=f\}\exists x_o(x=f_o \wedge P_o)$ (Axiom of Assignment)

where x is a variable

x_o is a variable not free in f or P

f is an expression

f_o is obtained from f by substituting x_o for free occurrences of x

P_o is obtained from P by substituting x_o for free occurrences of x.

2.2 Composition.

A program normally consists of a sequence of statements which are executed one after the other in the stated sequence. The notation (Q;Q') is used here to indicate a program or portion of program which specifies the execution of Q followed by the execution of Q'. If Q and Q' were functions, (Q;Q') would be known as their functional composition; and in fact it obviously satisfies the same associativity principle, in that the effect of the program (Q;Q');Q" must always be identical to that of Q;(Q';Q"). This is the justification for removing the brackets around pairs of statements in a language like ALGOL 60.

The rule of inference associated with composition is rather obvious and has already been used in the informal reasoning of the previous section. If it can be proved that P' is true on completion of Q, and whenever P' is true on initiation of Q' we can prove that P" is true on its termination, then we can assert categorically that P" will always be true after executing (Q;Q). In more formal terms:

2. $\vdash$ If P{Q}P' and $\vdash$P'{Q'}P" then $\vdash$P{Q;Q'}P" (rule of composition)

2.3 Conditionals.

A fundamental feature of computers is their ability to decide on a course of action in accordance with the truth or falsity of a condition. In machine code and in primitive programming languages, such a test is normally associated with a "jump" instruction; but in more advanced programming languages, the programmer is invited to specify a pair of (possibly compound) statements, say Q and Q', only one of which is ever executed. The selection is made in accordance with the truth or falsity of a proposition B. In the construction:

<u>if</u> B <u>then</u> Q <u>else</u> Q'

the statment Q is initiated whenever B is true and the statement Q' whenever B is false. Since we do not normally know the value of B until the program is under execution it is obvious that any assertion made about the consequences of executing the whole conditional must be true no matter which of the limbs has been selected. The strongest assertion which has this property is the logical disjunction of the two assertions R and R' which can be shown to be true when the limb Q and the limb Q' are selected. Thus we have the following rule of inference:

3. If $\vdash P \wedge B\{Q\}R$ and $\vdash P \wedge \neg B\{Q'\}R'$ then $\vdash$P{<u>if</u> B <u>then</u> Q <u>else</u> Q'}R∨R'

In most languages which permit a sophisticated conditional construction, an option is given to the programmer to omit the second alternative after (and including) the <u>else</u>; in which case, when the condition is false, no action takes place. The omitted <u>else</u> clause may be regarded as a definitional abbreviation of "<u>else</u> <u>null</u>", where null is a null statement, performing no action. In other words, any proposition which is true before executing a null statement remains true after its execution. This simple rule may be embodied in the axiom scheme:

4. $\vdash P\{\underline{\text{null}}\}P$

2.4 Loops.

The essential feature of a stored program computer, - indeed the only feature which makes it worth while to store the program at all, - is its capability of executing the same section of program repeatedly. A section of program capable of repeated execution is known as a loop. Two parts of a loop can be distinguished:

1. A continuation condition B: if this is true on initiation of any repetition of the loop, the execution of the loop continues; whereas if it is false, the loop terminates immediately.

2. A statement S; this forms the body of the loop, and is executed none or more times in accordance with the status of the condition B. The simplest notation for a loop incorporating these two parts is:

while B do S

Every other form of loop, DO statement, for statement can be ultimately defined in terms of this simple construction.

The loop is the first programming feature for which the construction of proofs of program characteristics is non-trivial. In fact, it demands that one find some proposition P whose truth can never be affected by execution of the body of the loop S. Then obviously, the truth of P will not be affected, no matter how many times S is executed; and if it is true before the loop starts, it will still be true when the loop terminates. Furthermore, at that stage it is known that B is false. These considerations lead to the following slightly more accurate formulation:

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

2.5 Declarations and blocks.

Most modern programming languages permit the programmer to make an arbitrary choice of identifiers for his variables. These identifiers are usually regarded as "local" to some part of the program (block, procedure, subroutine); and if the same identifier is introduced in an entirely disjoint portion of the program, it is regarded as in fact a different variable. The arbitrary nature of the choice of a variable identifier is illustrated by the fact that the identifier may be replaced systematically throughout the relevant portion of program by some other suitably chosen variable name, without making any difference whatsoever to the meaning of the program.

In a language such as ALGOL 60 or PL/I, which allows the nesting of the scopes of declared variables, the rules for arbitrary substitution are slightly more elaborate; since a distinction must be made between "free" and "bound" occurrences of the same identifier. If a declaration (implicit or explicit) is regarded as a form of "quantifying" or "binding" occurrence of the declared identifier, these rules are exactly the same as those which govern the substitution of variables in logic. This appeal to a familiar concept greatly simplifies the treatment of declarations.

The main feature about a block in which a variable is declared is that it is impossible to refer in any way to that variable when outside the block. In other words, it is not possible to carry out of the block any information about the nature or value of the variable. This can be assured if the assertion R describing the result of executing a block contains no free occurrences of any identifier declared in the head of the block. Similarly, a description of the conditions which hold on entry to a block must not be allowed to refer in any way to a local variable of the block. Thus we can formulate the following rule of inference:

6. If P{Q}R and neither P nor R contain free occurrences of the variable v, then

P{(d v;Q)}R

where d is a declarator.

This definition deals with a bracketed block in which only one variable is declared. The usual ALGOL block with

begin ... **end** brackets and multiple declarations may be regarded as an abbreviation; thus

begin $\underline{d}_1$ $v_{11}, v_{12}, \ldots; \underline{d}_2$ v_{21} ...;...; S **end** stands for

$(\underline{d}_1 v_{11};(\underline{d}_1 v_{12};(\ldots(\underline{d}_2 v_{21};(\ldots;(S)..))\ldots)))$

2.6 General Reservations.

The axioms quoted above apply to a relatively simple language, in which the evaluation of all expressions, and of Boolean expressions in particular can never have any side-effects. Many modern languages permit function calls to feature in expressions, and do not prohibit these function calls from affecting the truth of propositions which were valid before their execution. Such languages should be regarded as providing merely an alternative notation for a language in which such function calls have been replaced by an appropriate sequence of procedure (subroutine) calls, as is suggested by the ALGOL 60 definition. If the major purpose of using a highlevel language is to permit the easy verification of the validity of programs expressed in the language, it is doubtful whether the use of functional notation for procedures with side-effects is a genuine advantage.

Another general feature of the axioms quoted above is that the proof of a theorem about the results of a program gives no guarantee whatsoever that the program will ever actually produce those results; in fact it may never terminate in the normal way. The failure to terminate could be due to one of many causes:

1. The evaluation of an expression demands the performance of an operation which, as a result of so-called "overflow", fails to deliver a result.

2. The execution of the program demands more time than allowed by the operating system time limit (time overflow).

3. The execution of the program demands more space than can be made available to it (space overflow).

4. The program contains an infinite loop.

Although much attention has been given by theorists to the fourth possibility, it is in fact ~~of~~ often less practical significance than the other three. On the other hand, in spite of theoretical difficulties, it is usually fairly easy to prove termination by some technique similar to that suggested by Floyd. Proofs that a program will not encounter one of the other three overflow conditions are often based on the known characteristics of a particular implementation, and can obviously never be achieved in a wholly machine-independent fashion unless careful use has been made of environment enquiries. However, it is to be hoped that any good implementation of a language will always clearly inform the user of a program when one of these conditions has occurred; so in fact, all proofs from the axioms quoted above should be regarded as proofs of a certain conditional assertion:

"If the program terminates successfully (ie.without an overflow message) then its results will have the desired properties". In many circumstances, this weaker assertion is quite good enough, since the program is found to terminate successfully on most of the occasions it is needed, and on the remaining occasions the results are disregarded and so cannot cause serious loss. No!

However, in certain kinds of real time control programs, operating systems, etc. the effect of even a detected failure can be exceptionally serious; in these cases, the programmer has a duty to prove that overflows cannot occur, even if he has to rely on machine-dependent characteristics to do so. This may be one of the more inescapable reasons why real-time programming is so resistant to machine-independent programming techniques.

3. Data Structures.

The preceding section deals with the assignment of simple values only to simple variables. These values are assumed to belong to some primitive type which is provided by the language, and defined by a suitable axiom set. The programmer is invited to consider these values as completely unstructured objects, since to regard them as a structure of (say) binary digits would be to introduce both unnecessary profusion of detail and a high degree of machine-dependence.

However, in practice, a programmer often requires to group together certain information into a structure, which can be regarded in some way as an independent entity in its own right. For example, in ALGOL 60 the programmer can group together a sequence of variables in an array, whose elements he can access and assign to individually. In languages such as COBOL and PL/I, the programmer has access to other means of defining and using data structures.

It is helpful in a theoretical approach to separate the definition a structured type from the declaration of an actual instance of the structure, in the same way as the type integer is distinct from any particular integer variable or value. A structured type may be regarded as the set of all values which exhibit the structure. For example a particular structured type could be defined as "the set of all real arrays with subscripts ranging from 1 to 24"; in ALGOL 60, the declaration

real array A, B, C [1:24];

may be regarded as declaring three variables whose value belongs to the given structured type.

3.1 Arrays.

The only structuring method provided by ALGOL 60 is the array. In general, a single-dimensional array A may be regarded as a mapping of one set (a range of integers) onto the elements of a particular type (e.g. integers, or reals, or Booleans). The element A [i] is the value corresponding to the integer i in the mapping A. A declaration of the form real array A[m:n] may be regarded as stating the domain of the mapping A to be the set of integers between m and n (inclusive), and the range of the mapping to be a subset of floating point numbers. In other words A is a member of the "power set" X^Y, where X is the set of reals, and Y is the set of integers between m and n.

In more general languages, the range of an array may be any data type, and its domain may be other than a contiguous sequence of integers. For example, a two-dimensional array may be regarded as a mapping between pairs of integers and (say) reals. It is therefore appropriate to axiomatise the more general concept of the power set X^Y, where X and Y are arbitrary sets of values.

P1. $x \in X^Y \supset \forall_y (y \in Y \supset x[y] \in X)$

This axiom correctly fails to say anything about x[y] in the case where y is not within the appropriate "subscript range".

In order to fully characterise a class of structur~~al~~ed objects, it is necessary to state the conditions of equality of elements of the class. This is usually a trivial matter:

2. $a,b \in X^Y \supset .\ a=b \equiv \forall y(y\in Y \supset a[y]=b[y])$

Finally we wish to define that transformation on an array which consists in assigning a new value to exactly one of its components. The resulting array is designated by $(A|i|x)$, where i is the "subscript" of the changed element, and x is its new value

3. $A\in X^Y \wedge y \in Y \wedge x\in X \supset (A|y|x) \in X^Y$

4. $(A|y|x)[z] = A[z]$ for $z\neq y$

5. $(A|y|x)[y] = x$

Now it is a simple matter to define the effect of assignment to a subscripted variable:

$$P\{x[i]:=f\} \exists x_o,\ x=(x_o|\,i_o|\,f_o)\wedge P_o$$

where i_o, f_o, P_o are like i, f, P, except that they contain free occurrences of x_o wherever i, f, P contain free occurrences of x.

3.2 Cartesian Products.

In languages such as COBOL, JOVIAL, and PL/I, the array-structuring mechanism is supplemented by a technique which permits the elements of the structure to be accessed by means of named selectors ("field identifiers") rather than by computed subscripts. In suitable cases, this can afford many advantages:

1. The elements of the structure may be of different types, whereas array elements must be all of the same type.

2. Access to an element is entirely free from the risk of subscript error, and no run-time check is required.

3. In the interests of conserving space, the elements can often be packed together in a single computer word.

The "type" of a structured value of this kind may be defined by specifying the types of the individual elements of the structure. If X and Y are types (possibly the same) then XxY is the type of all ordered pairs (x,y), where x belongs to X and y belongs to Y. If X and Y are both floating point types, then XxY is the set of all ordered pairs of floating point numbers. Any element (x,y) of this type may be regarded as representing the position of a point in 2-dimensional space, by means of its Cartesian coordinates. Thus the type XxY may fairly be called the "Cartesian Product" of the types X and Y; and the term Cartesian Product has been adopted even in cases where X and Y are not floating point.

In this section we give axioms for Cartesian products of any number of "dimensions". This implies that each axiom in fact stands for an infinity of axioms, one for each required dimensionality. Thus the use of dots involves no breach of rigour. The first axiom defines the range of elements of a Cartesian product and the second defines the conditions of identity:

1. $a\in(X\times Y\times\ldots) \equiv \exists x,y,.. (a=(x,y,..)\wedge x\in X\wedge y\in Y\wedge \ldots)$

2. $(x,y,..) = (x',y',\ldots)\equiv x=x'\wedge y=y'\wedge \ldots$

If a group of elements of various types have been "packed" together as a single element of the Cartesian Product type, the programmer will require some facility to "unpack" the constituent elements again, so as to inspect them and operate upon them individually. A suitable notation for achieving s is:

$$(x,y,\ldots) := f$$

where x is a simple variable of type X,

y is a simple variable of type Y,

---

f is an expression yielding a value of type (XxYx...).

The relevant axiom to describe the effect of this statement is modelled on that for normal assignment:

$$P\{(x,y,\ldots) := f\} \exists\, x_o, y_o, \ldots ((x,y,\ldots) = f_o \wedge P_o)$$

where f_o, P_o are like f,P, except that they contain free occurrences of x_o, y_o,..wherever the latter contain free occurrences of x,y,...

3.3 Set Unions.

In machine code, it is possible to use the same storage area at different times to hold values of different types and different structures. Highlevel languages such as FORTRAN permit the programmer to achieve the same effect by EQUIVALENCE statements, or in COBOL by the REDEFINES verb. The use (or misuse) of this facility permits machine-dependent effects to be obtained, and a programming language description must be careful to leave these effects undefined.

To represent this feature of computers and programming, we introduce the concept of a Union of types X,Y,... which may be written X+Y+... The elements of the union must not be regarded as identical with the elements of any of the types X,Y,...; in fact a Union type should be a closed system, in the same way as other types. However, for each alternative Z of (X+Y +Z+...) there is a one-one mapping (transfer function) T_Z which maps a particular element of the union type onto an element of the type Z. Thus if we start with an element z of type Z we can obtain an element of the union type, namely $T_Z^{-1}z$. If we now apply T_z, we will get back the element $T_Z T_Z^{-1} z = z$ itself. However, if we apply the wrong transfer function, say T_X, there is no way of telling which element of X (if any) we will obtain. Thus the effect can only be understood in machine-dependent terms.

The relevant axioms for the Union Type are:

1. $a \in (X+Y+\ldots) \equiv T_x a \in X \vee T_y a \in Y \vee \ldots$
2. $a, b \in (X+Y+\ldots) \supset a = b \equiv T_x a = T_x b \wedge T_y a = T_y b \wedge \ldots$
3. $a \in (X+Y+\ldots Z \ldots) \supset T_X T_X^{-1} a = a \wedge T_y T_y^{-1} a = a \wedge \ldots$
4. $z \in Z \supset T_z^{-1} z \in (X+Y+\ldots+Z+\ldots)$

The function T_z^{-1} for any Z corresponds to the PL/I function UNSPEC which maps a value of any type onto the unspecified machine representation of that value.

Some languages which permit the use of a Union type also provide a mechanism for the programmer to enquire exactly which type the current value of a union variable belongs to. This implies that an implementation actually records, as part of the value of the variable, a marker quantity, indicating the type of the value of the rest of the variable: this certainly has advantages, but it is not usually done in any COBOL, FORTRAN or PL/I implementation. To discriminate s marker, a suitable notation is similar to that for the conditional 'case") construction:

<u>consider</u> u <u>when</u> X <u>then</u> Q_1
<u>when</u> Y <u>then</u> Q_2
- - - - - - - -

where Q_1 is a statement to be executed when $T_x u \epsilon X$

Q_2 is a statement to be executed when $T_y u \epsilon Y$

The relevant axiom is also modelled on the conditional rule:

If $P \wedge u \epsilon X\{S_1\}Q_1$
and $P \wedge u \epsilon Y\{S_2\}Q_2$, etc.,

then $P \wedge u \epsilon (X+Y+\ldots)\{$<u>consider</u> u <u>when</u> X <u>then</u> Q_1
<u>when</u> Y <u>then</u> Q_2
---------------- $\}Q_1 \vee Q_2 \vee \ldots$

3.4 <u>Streams</u>.

In representing the actions of input and output, it is useful to introduce yet another structuring mechanism, the <u>stream</u>. This includes and unifies many features usually associated with files, stacks, lists, and strings. A stream is an arbitrarily long sequence of elements of some type, with at most one element (the current element) accessible at any one time. There is assumed to be a means of testing whether a stream is "empty" (end-of-file condition). The basic operations on streams are:

output: Add an element of appropriate type at the end of the stream
input : Remove an element from the beginning of the stream
backspace: Remove an element from the end of the stream.

If S is a stream, and x is a potential element of the stream, we write $(S||x)$ for the stream which results from outputting x at the end of S; and we write $(x||S)$ for the stream consisting of x followed in sequence by the elements of S. We write X^* as the type of the stream, all of whose elements are members of X. The following axioms apply.

1. $\text{nullstream} \epsilon X^*$
2. $x \epsilon X \wedge S \epsilon X^* \supset (x||S) \epsilon X^* \wedge (S||x) \epsilon X^*$
3. The only members of X^* are as given by rules 1 and 2.
4. $(x||S) = (x'||S') \equiv x=x' \wedge S=S'$
5. $(S||x) = (S'||x') \equiv x=x' \wedge S=s'$

We are now in a position to define axioms for input and output:

6. $P\{$<u>output</u> f <u>on</u> $S\}\exists S_o\ (S=(S_o||f_o) \wedge P_o)$
7. $P\{$<u>input</u> x <u>on</u> $S\}\exists S_o, x_o (S_o=(x||S) \wedge P_o)$
8. $P\{$<u>backspace</u> x <u>on</u> $S\}\exists S_o, x_o (S_o=(S||x) \wedge P_o)$

4. <u>Advanced features</u>.

The previous sections contain a set of axioms suitable for defining fairly simple languages, which are nevertheless sufficiently powerful to express practical algorithms in reasonably concise and efficient form. This section explores some of the more advanced features of modern languages, with a view to elucidating their axiomatic basis.

4.1 Procedures.

As defined in ALGOL 60, a procedure call is exactly equivalent to the insertion of a (possibly modified) copy of the procedure body in the place of the call. This means that the results of a procedure call are identical to those of the procedure body, provided the preconditions are the same. Thus if we introduce the notation

n=B

to signify that n is a procedure identifier of a procedure with body B, then we can readily adduce the following rule of inference:

1. if ⊢P{B}Q then ⊢P∧n=B{call n}Q

Now all that is required is some means for establishing the truth of "n=B". This is done in ALGOL-like languages by a procedure declaration in the head of a block. Thus we postulate axioms of the form:

2. If ⊢P∧n=B{Q}R then

   ⊢P{(procedure n;B;Q)}R

where n is not free in P or R. This approach is similar to that for other declarations (2.5), and invites a similar convention for abbreviation.

In most languages, some means is provided for passing parameters to a procedure at time of its invocation. There are many different mechanisms for parameter passing, but the name parameter mechanism seems the simplest and most general, in terms of which many of the other mechanisms can be defined. If we use the convenience of lambda-notation, to express functional abstraction, the following axiom appears to be sufficient:

λ(x',y',..).B][x,y,...]

3. if ⊢P{B}Q then ⊢P∧n=λ(x',y',..)B{call n(x,y,....)}Q

In the second assertion the occurrences of x,y,...within B are assumed to be "bound" by the occurrences within the preceding lambda-brackets, whereas the occurrences of x,y,...as actual parameters are free, as are the occurrences of x,y,...within P and Q.

The relationship between the procedure identifier n and the procedure λ(x,y,...)B is established by declaration:

4. If ⊢P∧n=λ(x,y,...)B{Q}R

   then ⊢P{procedure n(x,y,...); B; Q}R

The axioms given above may be applied successfully to procedures in a language like FORTRAN, which makes no attempt to define the effect of recursive procedure call. To make it possible to prove truths about recursive procedures, a slightly more complicated rule of inference is required, just as a slightly more complex storage control mechanism is required to implement them. In this more complex formulation, the successful working of the internal recursive calls may be assumed in the proof that the procedure body as a whole will be successful:

5. If from ⊢P{call n(x,y,...)}Q it can be proved that P{B}Q

   then ⊢P∧n=λ(x,y,..)B{call n(x',y',...)}Q

The possibility of mutual recursion adds yet further complexity to the situation. The axiom above must be strengthened still further:

6. if from ⊢P_A{call a(x,y,...)}R_A

   and ⊢P_B{call b(x',y',...)}R_B

   [illegible]

   it can be proved that

   ⊢P_A{A}R_A and ⊢P_B{B}R_B and ....

[illegible]

then $P_A \wedge a=\lambda(x,y,\ldots)A \wedge b=\lambda(x',y',\ldots)B \ldots\{$<u>call</u> $a(x,y,\ldots)\}R_A$.

Mutual recursion also requires a strengthening of the rule for procedure declarations:

7. If $\vdash P \wedge a=\lambda(:..)A \wedge b=\lambda(\ldots)B..\{Q\}R$

   and a, b, ... are not free in P or R,

   then $\vdash P\{($<u>procedure</u> $a(\ldots);A;$<u>procedure</u> $b(\ldots);B;\ldots;Q)\}R$

4.2 Record Handling.

The axioms and rules of inference given so far do not make any particular assumption about the storage allocation method used in an implementation. All the axioms except those permitting recursion, and equally true of FORTRAN, which generally uses static allocation, and of ALGOL, for which a stacked technique is often used. It is only the recursive axiom which differentials between the two approaches.

In addition to the stack, several modern languages permit an even greater degree of flexibility in the dynamic allocation of storage, and for deallocation they use possibly some entirely invisible "garbage collection" process. In such a language, the set of storage elements ("records") which have been allocated at any given moment of program execution form a finite set C, but the program at any time may add a new element to the set, and store a "reference" to this new element in a suitable pointer (or reference) variable r. The statement which achieves this effect may be given the form:

r <u>new</u> C

This statement obviously changes the current values of the variable r and of the class C; its effect may be described in a manner similar to assignment:

$P\{r$ <u>new</u> $C\}\exists r_o, C_o.\ C= C_o+\{r\} \wedge r \notin C_o \wedge P_o$

where P_o is like P except that it contains free occurrences of r_o and C_o wherever P contains free occurrences of r and C.

4.3. Parallelism.

Certain modern languages claim to permit the programmer to specify that two or more portions of program are to be executed in parallel with each other. The nature and implementation of this feature are still problematic; and its rigorous definition is rather difficult, since parallelism apparently permits the specification of programs with inherently unpredictable results. Since it is very difficult to prove anything about the properties of unpredictable programs, it would not be surprising to find that any axioms which claimed to apply in such cases would be very complicated, and almost impossible to understand and use.

If the main purpose of using a programming language is to obtain programs which can fairly readily be proved to work, it would seem to be desirable to restrict the facility for invoking parallelism to cases in which the unpredictability is absent, or at least reasonably circumscribed. In the case of parallel streams of computation, this can be achieved by ensuring that the parallel streams contain no free variables in common; or at least that no assignment or input/output operation is carried out on any variables which they have in common. Introducing the notation (Q//Q') to indicate that Q and Q' are to be executed in parallel:

1. If $\vdash P\{Q\}R$ and $\vdash P'\{Q'\}R'$,
   and no left-hand variable of Q is free in $P'\{Q'\}R'$
   and no left-hand variable of Q' is free in $P\{Q\}R$
   then $P \wedge P'\{(Q//Q')\}R \wedge R'$

is assumed that the definition of "left-hand" variables of a piece of rogram is achieved by a purely syntactic scan, very similar to that used to efine the "free" variables of a text.

This form of parallelism may be used quite successfully by a programmer to achieve concurrency of computation and input/output. It does, of course, completely exclude any possibility of indeterminacy, since it does not permit any communication whatsoever between the parallel processes. A useful extension to the simple facility would be to introduce such communication, provided this still permits proofs to be constructed about the performance of the program. It appears that this can be achieved by using the input/output mechanism, and linking the output of one parallel process directly into the input of another. However, further discussion of this topic would be out of place in this paper.

4.4. Jumps.

It is notorious that ALGOL programs containing any significant number of go to statements are remarkably difficult to understand and verify. It would seem that proofs of the correctness of such programs will be equally difficult to construct, and the axioms governing such proofs will be correspondingly complicated. This complexity is mainly due to the "linearity" of program texts when treated by conventional syntactic techniques. If the program is organised as a flowchart, there is no problem with jumps, as Floyd has shown. Furthermore, there is not much difficulty in the case of a "flat" language such as FORTRAN or machine code which do not have any "nesting" of statements, nor do they permit jumping out of a block. In such a language, a fairly simple approach may be taken.

Let us deal first with a sequence of instructions T which contains exactly one conditional jump instruction and one label.

Let X be the sequence of statements up to "if B then go to A"

Y be the sequence of statements following "if B then go to A"

Z be the sequence of statements following the label "A":

W be the sequence of statements ~~following~~ up to the label "A".

1. If $\vdash P\{X\}Q$ and $\vdash Q \wedge B\{Z\}R$ and $\vdash Q \wedge \neg B\{Y\}R'$ and $P\{W\}O'$ and $Q\{Y\}R$

then $P\{T\}R \vee R'$

If there are more than one conditional jump to A in a sequence T, the preconditions of rule 1 must apply to every one of them. If there is more than one label, yet further complexities arise. The unravelling of them is left to a reader with a sufficient enthusiasm for jumping.

Conclusion.

The paper has adduced a number of axioms which define the dynamic properties of a simple but complete and powerful programming language. It is a considerably smaller language than those which are currently fashionable. However, many of the features of more sophisticated languages may conveniently be regarded as definitional abbreviations of material which could have been expressed more diffusely in the primitive language. The introduction of useful definitional abbreviations, and the proof of powerful metatheorems about them, is one of the main avenues for advance in mathematics; and one would hope to see the same possibility for development in programming languages, - particularly those which permit the user to define new notations and new interpretations for existing symbols.

We have not made any attempt to prove the consistency, completeness, dependence of the proposed axiom set. A more urgent task is actually se the axiomatic proof technique to develop a library of useful software se correctness has been put beyond reasonable doubt. It is only by ilding on a foundation of reliable programs and theorems that we can hope to extricate ourselves from our present software difficulties, and progress in our capability to use a computer effectively as a partner in problem-solving activity.

References:

C. A. R. Hoare	The Axiomatic Method
C. A. R. Hoare	The Axiomatic Method: Data Manipulation
R. W. Floyd	Assigning Meanings to Programs
P. Naur	Proof of algorithms by generalised snapshots