C B Jones

THE AXIOMATIC METHOD

PART I

C.A.R. HOARE.

~~Oct 1968~~

DEC 67

Summary.

This paper points a distinction between two fundamentally differing approaches to the definition of mathematical foundations and of computational algorithms; the two approaches are known as "constructivist" and "axiomatic". It is suggested that the axiomatic approach may be more suited to the rigorous definition of programming languages, particularly for purposes of standardisation.

1. Introduction.

Most current approaches (1,2,3) to formal definition of languages are based on a machine-like construction of some convenient degree of abstraction; this passes through a succession of states in accordance with more or less deterministic transition rules, laid down by the statements of the definition. Such an approach may be characterised as mechanistic, constructivist, or algorithmic. The axiomatic approach, on the other hand, does not rely on the concept of a machine, whether concrete or abstract; and it does not lay down any particular algorithm which must be used to execute a program in the given language. Rather, it uses axioms to state certain very general properties which every algorthmic implementation of the language must possess in order to qualify for that title. The axiomatic approach tends to give subtle and implicit rather than explicit definitions, but nevertheless, it has advantages similar to those encountered in the formalisation of mathematics and in the design and description of general purpose programs.

2. Foundations of Mathematics.

The distinction between the axiomatic and constructivist methods is clearly apparent in the study of the foundations of mathematics. For example, consider the approach to the formalisation of the arithmetic of natural numbers. There are several constructivist proposals on the subject:

1. Frege (4) defined a natural number as a set of sets with the same number of elements.
2. The CUCH (5a) suggests that a natural number is a functional, which transforms a function f into another function g, where the result of applying g is the same as that of applying f a certain number of times.
3. Van Wijngaarden (5b) constructs an integer in terms of its decimal representation as a sequence of digits.

The question now arises whether these are valid interpretations of our "intuitive" concept of natural numbers? In order to answer this question, we will need to have an <u>independent</u> criterion for the meaning of natural numbers. In fact, such an independent criterion was supplied by Peano (6) in the form of <u>axioms</u>. The proponent of any constructivist definition of natural numbers is obliged to prove that they satisfy the axioms of Peano. Thus it is the axioms, not the constructions, which supply the definitive formalisation of the concept of a natural number. This fact is widely recognised among modern algebraists, who value their right to define groups, rings, fields, etc., in purely axiomatic terms; and would regard constructions which satisfy the axioms merely as instances, examples, illustrations, or models, of that which they are really interested in.

A close analogy can be drawn with the definition of integer arithmetic in a programming language. An iteger in ALGOL 60 can be constructed (implemented) in many ways:

1. As a collection of bistable electrical states in a binary computer.
2. As a rotary cog position in a calculating machine.
3. As a position of beads on an abacus.

The question now arises, whether these are valid implementations of ALGOL integers? If the formal definition of ALGOL were expressed in terms of a selected one of the techniques, then it would be impossible to prove that any of the other techniques was valid, unless we had an independent criterion of the properties which it must share with the selected technique. But if we have such a criterion, then there is now no need to enshrine one of the techniques in the formal definition; why can't we use the criterion itself? As before, the criterion could be expressed as a number of axioms, similar to those of Peano, but adapted in light of the finitude of computer arithmetic.

3. Program Description.

The distinction between the axiomatic and constructivist techniques in the description of programming languages may also be illustrated by different approaches to the description of particular algorithms. Suppose, for example, we wish to talk about sorting programs. There are many ways in which we could describe such programs:

Informal Comment:

The procedure SORT sorts an integer array A[m:n] using an integer-valued function of integers, namely f, to define a sequencing on elements of A.

Axiomatic Approach:

On exit from SORT, the array A will have been transformed to an array A' with the following properties:

1. There is a one-one mapping P [m:n→ m:n] such that A'[P(i)] = A[i] for all i between m and n.
2. f (A'[i])≤f(A'[j]) for all i,j such that m≤i≤j≤n.

Algorithmic method:

```
Procedure  SORT (A,f); integer array A; integer procedure f;
    begin integer   i,j,k w,fw;
        for  i:=m+1 step 1 until n do
        begin w:=A[i] ;  fw:=f(w);
            for j:= i-1 step - 1 until m do
                if f(A[j])>fw then A[j+1]  :=A[j];
                    else begin  A[j] :=w;
```

```
                go to L
            end j;
A[m]:=w;
L: end i

end SORT;
```

Each of these three description methods is useful in different circumstances. The informal comment is essential to instruct a programmer how to use a sorting procedure within his program; and the algorithm is essential to actually cause the sorting to take place in a computer. The axioms are in some ways more similar to the informal comment, and they might be chosen as a means of establishing a standard to which many different sorting algorithms are intended to conform. The use of an actual algorithm to define a standard suffers from the following defects:

1. It contains an error; the statement "A[j]:=w" should read "A[j+1]:=w" such errors are characteristic of the algorithmic descriptions of complex processes.
2. If f (A[i]) = f (A[j]) then these elements retain the order which they possessed before the beginning of the procedure. This is probably not an essential requirement on all sorting techniques.
3. The procedure is very inefficient when the array A is large, or the function f laborious to compute.

The axioms seem to offer corresponding advantages for standardisation purposes:

1. The axioms are in general much simpler and more self-evident than any algorithm which implements them; and they are therefore less prone to error.
2. The axioms enable the standardiser to leave indeterminate the effects of cases in which he does not mind what the result is, and yet he can define exactly the degree of determinacy that he needs.
3. The implementor can take advantage of permitted indeterminacy to design an algorithm which is very much more efficient than the one quoted.

4. Language Standardisation.

A high level programming language should be capable of being implemented on computers with widely differing designs and architectures. If the language is to be successful as a standard, it is essential that working programs should be capable of transfer from one machine to another without any fear of producing incorrect results. That this is unachievable with existing languages only shows how far we have yet to go.

However, it is not good enough for a language to standardise too rigidly, since otherwise it will be impossible to implement it efficiently on differing hardware designs. It is essential to isolate implementation dependencies (for example, accuracy of floating point), and leave the implementor sufficient freedom to make the best choice for his machine. That is why it is not practicable to standardise a language on a particular implementation of it. A successful language description technique will ensure that the limits of the undefinedness in implementation-dependent areas can be defined with sufficient determinacy to enable the user to obtain the desired results from his program on all implementations. Thus the purpose of a formal definition is to establish the correct degree of standardisation of a language across many implementations.

A set of axioms, it is believed, can be formulated to express exactly the properties which must be displayed by every implementation of the language, and it can do so in a simple and subtle manner. For example, it can express a quite elegant formulation of the properties of real arithmetic which one would like to require of every implementation of the language, without prescribing any particular accuracy of floating point representation. Mechanistic interpretations of a language tend to omit any description of real arithmetic, or else to describe it in a way which puts unacceptable constraints on a practical implementation.

Of course, the axiomatic definition of a language standard cannot, for practical reasons, prevent an implementation from possessing features extraneous to the definition of the language - for example an implementor must choose some particular technique of floating point rounding or truncation. Such a choice can often itself be expressed by means of supplementary axioms; and thus many implementations may share all the standard axioms, but differ in a choice of certain specific axioms; just as Euclidean and non-Euclidean geometries share most of their axioms, but differ in a choice of the parallel postulate.

The use of axioms to describe the semantics of a programming language may be found to have other advantages besides that of assisting in standardisation. These potential advantages are described in the following sections.

5. Proof Construction.

The conventional technique 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 it provides results which do not correspond to his intentions. After he has found a reasonably wide variety of example cases on which his program works, he believes that it will always work. This technique is adequate for small programs, in well understood applications; but for very large programs, or for new application areas, it is altogether less satisfactory; and when dealing with interrupts and parallel actions, the usefulness of traditional program testing methods is very suspect.

Quite apart from practical difficulties, there are serious theoretical objections to a technique which so closely resembles the attempt by a mathematician to prove a theorem by showing that it is true of the first thousand numbers he thought of testing it on. For this reason it has been suggested [7,8,9] that the best

way of making sure that a program works is to prove that it works, in the same way that a mathematician proves a theorem. The proof can then be read and checked by other programmers, or even published for the scrutiny of the learned world. It is only in this way that a sound basis can be laid for the science of programming.

Techniques for proving the correctness of programs at present must rely on an intuitive understanding of the meaning of the program itself. However, the axiomatic method might provide a set of axioms, theorems and metatheorems which can be used quite directly in the construction of proofs about programs. The axioms themselves should be reasonably "self-evident" as are the axioms of many of the branches of mathematics; and the primitives of the system should correspond closely to the intuitive understanding of the language by programmers. It will therefore be easier to discover and formulate proofs than if the basic axioms and concepts were purely arbitary.

If the practice of providing proofs with programs becomes widely accepted, it will lead to a solution to the problem of program interchange between differing implementations. A proof of program correctness which uses only the axioms of the language will ensure the success of program interchange; whereas if the program takes advantage of implementation-dependent features the proof will need to use supplementary axioms particular to an implementation, and then successful interchange cannot be guaranteed.

6. Comparative linguistics.

At present, programming languages are described in a wide variety of different ways. The proposed definition of the standards for ALGOL, FORTRAN, COBOL and PL/I are radically different in their style and content, in spite of great similarities in the languages which they describe. It is consequently almost impossible to identify the similarities and isolate the differences between them.

It is hoped that a more rigorous method of language definition will provide a common technique for description of all languages, and thereby form a basis for the comparative examination and evaluation of existing languages and of new language proposals. If this hope is to be fulfilled, it is necessary that a reasonable proportion of all the statements of a language definition shall apply unchanged to other similar languages. Thus attention can be concentrated on those statements which are different, and one can begin to understand and explain these differences. The axiomatic method seems rather promising in this respect, since mathematicians already have experience in the choice of independent axiom sets, in which a change to a few of the axioms will not affect the validity of the rest; for example, the axioms of Euclidean geometry have been devised in such a way that a choice of two non-Euclidean geometries may be obtained by modifying a single axiom.

7. Language design.

The regularity and clarity of the syntax of ALGOL 60 (as compared with, say, COBOL) is one of its most attractive features; and is of genuine assistance to the programmer in the avoidance of trivial but tiresome syntactic errors. This regularity is mainly explained by the strong desire of the designers of the language for simplicity and perspicuity; but at least part of the credit must be given to the use of Backus Normal Form, which made it easy to define constructions which correspond to our intuitive ideas of regularity and simplicity, and a lot more difficult to define a language which offends these ideas.

In the same way, it is hoped that a technique for formal definition of semantics will encourage the design of languages which are semantically simple and regular. If the axiomatic method of language definition is adopted, then it would seem likely that a language which could be described in few "self-evident" axioms will be more appealing on grounds of simplicity than one which requires many axioms. The minimisation of axiom sets is an activity in which mathematicians and logicians are already skilled.

There are three further grounds for supposing that the axiomatic method will prove a useful tool in the design of programming languages:

1. It is extremely flexible; there is no assertion which a language designer cannot easily propose as an axiom.
2. It enables the language designer clearly and simply to express his general intentions about a language, without risk of confusion with a mass of barely relevant detail.
3. Axioms can be formulated in a manner largely independent of each other, so that the reader can consider each axiom, or small group of axioms, in isolation from the whole mass of statements required to define the whole language.

8. The roleof constructivism.

One of the main objections to the axiomatic method is that it is too easy! This objection is well expressed by Russell [10], who held that,

> "The method of 'postulating' what we want has many advantages; they are the same as the advantages of theft over honest toil."

This objection certainly has valid grounds. In constructing axioms, there are two hidden dangers: first, that the axioms will be contradictory, and second, that they may not be sufficiently deterministic to serve their purpose.

One of the best methods of proving that these dangers have been avoided is to construct a model which satisfies the axioms, and prove that the model is deterministic to the required degree. In the case of a programming language, this model might very well be an abstract machine passing through a series of states in accordance with certain transition rules. Thus the axiomatic method still requires the support of constructivist techniques to check consistency and completness. But if the claims of this paper justified, the construction can be thrown away as soon as it has served its purpose, and the axiom set will remain as the ultimate arbiter on matters of language definition.

9. Conclusion.

A suggestion has been made that the axiomatic method may be suited to the rigorous definition of programming languages, and contribute to the goals of standardisation, language comparison, evaluation and design, and also assist in constructing proofs about the correctness of programs.

10. Acknowledgements.

The ideas of this paper have been influenced by fruitful discussions with M. Woodger, P.J. Landin, and members of the language definition group at the IBM Research Laboratory, Vienna.

This paper is published by kind permission of the Director of the National Computing Centre.

1. Landin, P.J. - Mechanical Evaluation of Expressions, The Computer Journal 6.4, January, 1964.

2. IBM Vienna Research Laboratory - Formal Definition of PL/I (TR 25.071) December, 1966.

3. de Bakker - Formal Definition of Algorithmic Languages MR 74, Matematisch Centrum, Amsterdam, May, 1965.

4. Frege, G. - Die Grundlagen der Arithmetik, 1884.

5. Formal Language Description Languages, North Holland, 1966.

   (a) Van Wijngaarden, A. - Recursive Definition of Syntax and Semantics.

   (b) Bohm, C. - The CUCH as a Formal and Description Language.

6. Peano, G. - Sul concetti di numero, Rivista di Matematica Vol. 1 1891.

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

8. Dijkstra, E.W. - On the Design of Machine-Independent Programming Languages M.R.34. Matematisch Centrum, Amsterdam, October, 1961.

9. Naur, P. - Proof of Algorithms by General Snapshots BIT. 1966 Vol. 6.

10. Russell, B. - Introduction to Mathematical Philosphy, Allen & Unwin, 1919.