C B Jones

THE AXIOMATIC METHOD:

DATA MANIPULATION

C.A.R. HOARE.

Dec 68

Summary.

This paper illustrates the manner in which the axiomatic method may be applied to the rigorous definition of programming languages. It deals with the definition of the primitive types of operand, and the operations which may be performed upon them. This is the area in which programming languages are normally most machine-dependent; and yet the axiomatic approach gives a largely machine-independent solution of the problem. Furthermore, the solution appears to apply to broad classes of language and many varieties of implementation. It seems, therefore, highly suited for languge standardisation purposes.

Introduction.

There are many purposes for which the description of a programming language is required, and it is usual to construct different types of description for each purpose. For example:

a teaching manual, for those learning to program;

a reference manual, explaining more advanced details to the initiated programmer;

a programmers' guide, describing the use of a particular implementation;

an implementors' manual, containing suitable instructions and hints;

even the language compiler itself may be regarded as a form of language description - indeed, the only one which is meaningful to a machine.

This paper deals with yet another purpose for which a language description is required, namely the standardisation of the language across many implementations; and a form of description which is suitable for this purpose is rather different from the forms which are familiar from their use in other contexts. A programming language standard, like standards for other artefacts, is intended to lay down a set of necessary properties of a certain class of products (implementations); and these conditions must be satisfied by any product which claims to conform to the standard. Most standards will leave the product designer considerable freedom in specifying its non-essential features, and will even permit variation within predefined limits for the essential design parameters. Similarly, a language description formulated for language standardisation purposes must permit reasonable variation of implementation details, so that the language can be successfully implemented on a variety of computers, using a variety of implementation techniques. However, this variety must be constrained within certain bounds, since otherwise it would not be possible to write programs which could be accepted and run successfully on all implementations.

Most descriptions of a standard are carefully and fairly unambiguously expressed in an ordinary natural language, supplemented by a few formulae, tables, and diagrams; and they use terminology which is familiar and widely understood in the relevant field. However, in the field of programming languages, there has not yet developed any established terminology, as will be revealed by a comparison of the descriptions of existing and proposed language standards. Furthermore, the larger programming languages are probably more complicated in their structure than any other product which has ever been considered as a candidate for standardisation. A need has therefore been expressed (1) for some rigorous or even formal technique for describing a language in a manner suitable for standardisation. It has been suggested in a previous paper (2) that the axiomatic method, as practiced by geometers, mathematicians, and logicians, may be well suited for this purpose.

In the construction of axiom sets modelling the essential features of programming languages, the following design criteria are relevant:

1. The axioms should not place constraints on language implementors which might be unacceptable for certain hardware designs or configurations of equipment.
2. The axioms should be sufficiently deterministic for the programmer to be able to secure the results which he wants on any implementation which satisfies the axioms.
3. The axioms should be formulated in such a way that many of them will apply unchanged to many data types in many programming languages. In this way it should be possible to gain an insight into the genuine similarities and differences between the various languages.

4. The axioms should be reasonably independent of each other, so that the language designer may freely discuss one axiom or a small group of axioms in isolation, without fear of unexpected interactions and alterations in other parts of the language.

5. The primitive (undefined) terms in the axioms should correspond with our pre-existing intuitive understanding of the behaviour of programming languages, and the axioms should recommend themselves as "self-evident" to one who is familiar with programming. This will ease the task of checking the correctness of the axioms themselves, and of using them in constructing proofs concerning the correctness of programs.

6. The number of primitive concepts and axioms should be kept to a reasonable minimum.

2. Data Types (General).

The easiest and most obvious application of the axiomatic method is in the definition of the primitive operations on simple values of the various types. This is also the area in which there is the greatest need for a machine-independent techniques for controlling machine-dependent phenomena. Sets of axioms which define the structure and behaviour of a simple type will bear a close resemblance to axiom sets already familiar to mathematical logicians, with changes made necessary only by the finitude of computer representations of the values concerned. A suggestion for using axiom sets for defining the primitive operations of a programming language was independently made by Laski. (3)

A data type may be regarded in the traditional manner of abstract mathematics as a set of objects over which certain operations are defined. In general, on a computer, the built-in operations take their operands and deliver results within the same data type; this corresponds to the familiar mathematical practice of defining closed systems, and giving an axiomatic description of the properties of the operators which are defined within that system and not outside it. The concept of a type-transfer function corresponds to the concept of a homeomorphism between closed systems, preserving some part of the structure of the mapped system. Thus languages which permit operations on operands of "mixed" type are preferably regarded as merely definitional abbreviations of a language in which all type-transfer functions have been made explicit.

In many branches of abstract mathematics, it is usual to define certain very general properties of every system in general (e.g. the axioms of closure in topology) and then to supplement these by specifying additional properties for certain specific systems of interest. In the same way, it is possible to list certain general properties of the operators (+, x, etc.) which apply to all data types, and then to give a few additional axioms to differentiate the individual types (real, integer, etc.). Thus many of the axioms will be independent not only of any particular language, and of any particular machine, but also independent of any particular data type.

2.1 Ordering.

A characteristic feature of most data types is that there is defined over them a partial or total ordering relationship, which we shall denote $\leq$ In the case of most arithmetic data types, this ordering corresponds to the natural numeric ordering; in the case of complex numbers, it must be interpreted as an ordering between the moduli of the numbers. The axioms of ordering are familiar:

O1. $x \leq x$ (reflexivity)

O2. $x \leq y \wedge y \leq z \supset x \leq z$ (transitivity)

O3. $x \leq y \vee y \leq x$ (total ordering)

The following axiom is not vali . for complex numbers:

O4. $x \leq y \wedge y \leq x \quad x = y$

The following definition will be found useful:

O5. $x < y =_{df} x \leq y \wedge \neg y \leq x$

2.2 Arithmetic.

Floating point arithmetic is well-known to be only an approximation to the arithmetic of mathematics and analysis; in other words, it satisfies only a selection of the familiar and traditional axioms. For example, in the case of addition and multiplication, commutativity is the only property which may be immediately claimed for computer arithmetic:

A1. $x+y = y+x$

A2. $x\times y = y\times x$

In order to reestablish the familiar identities of associativity and distributivity, it is necessary to introduce the concept of approximate equaly ($\doteq$). The exact interpretation of approximate equality will depend on the characteristics of each particular data type: for example, in the case of integers, approximate equality is exactly equivalent to genuine identity. However, even when approximate equality is used, it is necessary to make a stipulation that all operands and results in the following axioms exist and are strictly positive:

A3. $x=y \supset x\doteq y$

A4. $x \;\; y \;\; x$

A5. $(x+y)+z \doteq x+(y+z)$ (associativity of addition)

A6. $(x\times y)\times z \doteq x\times(y\times z)$ (associativity of multiplication)

A7. $x\times(y+z) \doteq x\times y + x\times z$ (distributivity)

A8. $(x-y)+y \doteq x$

A9. $(x/y)\times y \doteq x$

Axiom A9 does not apply to integers, for which the / operator is not defined. In order to characterise the properties of arithmetic on nonpositive numbers, it is necessary to provide rules for converting it to positive arithmetic, by suitably distributing the monadic negation operator:

A10. $-(-x)=x$

A11. $x+(-y)=x-y = -(y-x)$

A12. $(-x)-y = -(x+y)$

A13. $x\times(-y) = -(x\times y)$

A14. $x\doteq y \equiv -x\doteq -y$

In all these axioms, the existence of the terms is a precondition of the validity of the equation. In formalising logical deductions from these axioms, it will be helpful to adopt a logical system which permits use of a functional notation, but does not assume that an expression necessarily has a result: although if the result exists, it will be unique. A statement which refers to a non-existent object or a non-existent operation may be regarded, therefore, as vacuously true.

2.3 Constants.

In each data type there are certain elements which uniquely satisfy certain propositions. It is usual to designate these elements by the same conventional symbols for all types, even though the types are disjoint, and the element denoted is in fact different for each type. Thus there is a systematic ambiguity for these symbols as there is for the operator symbols. In a programming language, the chosen symbols are called "constants". The commonly encountered constants are zero and unity; also it is useful to name the maximum and minimum values, with respect to the ordering relationship defined over the type. These "constants", when accessed from within a program, can provide a useful "environment enquiry".

K_1. $x+0 = x$

K_2. $x\times 0 = 0$

K_3. $x\times 1 = x$

K_4. $x\leq max$

K_5. $min\leq x$

K_6. $0 < 1$

In types for which a given constant does not exist, the axioms which mention it may be regarded as vacuously true.

2.4 Finitude.

In proving theorems relating to elements of any given type it is essential to be able to characterise the totality of elements of that type in an exhaustive but exclusive fashion. In the case of integers, this is commonly done by an axiom of induction; and we use here a very similar method to enable inductive proof techniques to be applied to elements of the other types as well. First, a successor relationship is defined: x is succeeded by y (xSy) if it is strictly less than y, but there is no element between x and y. Then the set S*x is defined as the set containing x and its successor(s), and the successor(s) of its successor(s), and so on. Finally, it is possible to state that $x \leq y$ only if y belongs to the set S*x.

F1. $xSy =_{df} x<y \wedge \neg \exists z(x<z \wedge z<y)$

F2. $x \in S^*x$

F3. $y \in S^*x \wedge ySz \supset z \in S^*x$

F4. The only elements of S*x are as given by rules F2 and F3.

F5. $x<y \supset y \in S^*x$

Axiom F5 effectively distinguishes computer arithmetic from the arithmetic of the real continuum, in which the only member of S*x is x itself.

Taken in conjunction with K_4 and K_5, axiom F5 effectively asserts the finitude of each data type; since it implies that by starting with min and repeatedly taking all successors of what we have already reached, we will eventually, in a finite number of steps, reach every element of the type, including max; and this has no successors.

3. Particular data types.

In addition to satisfying the general axioms listed in section 2, each data type will satisfy further axioms, distinct from those satisfied by other types. These axioms relate mainly to the interpretation of the approximate equality ($\doteq$) and the successor relationship (S).

3.1 Integers.

For the integer type, it has been stated that approximate equality cannot be regarded as distinct from identity; furthermore, the successor of a number is obtained by adding one to it:

I_1 $x \doteq y \supset x=y$

I_2 $xSy \supset y=x+1$

These axioms, in addition to those which have gone before, seem sufficient to characterise all the machine-independent aspects of computer arithmetic on integers. However, certain machine-dependent aspects can be specified by means of supplementary axioms. For example, if a machine uses "sign-plus-modulus" representations, it will satisfy the first of the following axioms; but if it uses "twos complement" it will satisfy the second:

I_3 $max = -min$

I_3' $min = -max-1$

An even more important choice in an implementation of integer arithmetic is the treatment of overflow. In some implementations, an operation which overflows will yield no result whatsoever; and in others, an overflowing operation will yield a result computed by modulo arithmetic. A distinction between these two types of implementation may be made by a choice of one of the following:

I_4 $min = max+1$

I_4' $y = x+1 \supset xSy$

3.2 Floating point.

For floating point arithmetic, the concept of approximation is of vital importance. In fact, each computer "real" number may be regarded as an approximation for a certain range of genuine real numbers which neighbour on it. In standardised floating point arithmetic, the width of this range is roughly proportional to the absolute magnitude of the number. In axiomatising the concept of a "range" surrounding each element, it is desirable not to have to appeal to the existence of objects outside the data type itself. This can be achieved by postulating that the floating point numbers have a certain "density"; i.e., that each number is sufficiently "close" to its successor. This can be done by postulating a floating point constant known as "span" which has the property that there is a distinct number lying between any number and the result of multiplying it by span. It is obvious that the smaller the value of span, the more dense and accurate in general is the floating point representation. Thus an implementor will try to quote smallest possible value of span, which will therefore usually be a number just greater than unity; in fact, probably the successor of unity. The relevant axioms are

R1 $\quad 0<x \wedge xSy \supset y \leq x \times \text{span}$

The concept of approximate equality may now be explained in terms of one side of the equality being within the "span" of the other:

R2 $\quad 0<x \wedge 0<y \wedge x \approx y \supset x<y \times \text{span} \wedge y<x \times \text{span}$

These three axioms, together with those given in section 2, appear to characterise most of the machine-independent properties of standardised floating point arithmetic, ie those properties which are independent of the range of exponent, length of mantissa, and number base of standardisation.

Axioms for defining floating point arithmetic have previously been proposed by A. vanWijngaarden; these axioms, however, do not describe a closed system, since they depend on the preexistence of the mathematics of the real continuum.

3.3 Fractions.

Most computers, and several programming languages, permit the programmer to use fixed point arithmetic on fractions between -1 and +1. In fractional arithmetic, the concept of approximation is also relevant: but in this case, the size of the range of approximation surrounding each number is a constant, independent of the magnitude of the number. In fact, this range is equal to the distance between any two numbers, ie., the "step" which takes one number into its successor. These facts are summarised in the following two axioms:

P1. $\quad xSy \supset y=x+\text{step}$

P2. $\quad x<0 \wedge y<0 \wedge x \approx y \supset x \leq y+\text{step} \wedge y \leq x+\text{step}$

In fractional arithmetic, multiplication must be approximative, since, if the result is to be of the same precision as the operands, there must be some truncation or round-off error. Consequently associativity and distributivity rules for multiplication are approximate. However, addition and subtraction are completely accurate, and therefore the normal rules of associativity and cancellation may be established as identities:

P3. $\quad (x+y)+z = x+(y+z)$

P4. $\quad (x-y)+y = (x+y)-y = x$

3.4 Complex Arithmetic.

One method of defining complex arithmetic is to construct a complex number as an ordered pair of reals (known as its real part and its imaginary part), and then to use the familiar rules to give a method of computing the results of the arithmetic operations, for example:

real part (x+y) = real part (x) + real part (y)

imaginary part (x+y) = imaginary part (x) + imaginary part (y)

However, there is some advantage in giving an independent axiomatisation of complex arithmetic, for the following reasons:

1. An implementation of complex arithmetic may prefer to adopt a representation of complex numbers which is not the same as a pair of reals. For example, it may be economical to represent them as a pair of mantissae sharing the same exponent.

2. Even if complex numbers have been implemented using the standard constructive technique, it is interesting to have an independent criterion for acceptability, and to attempt to prove that the construction is valid, in the sense that it satisfies this criterion.

Complex arithmetic satisfies all the axioms for real arithmetic except O4, provided that the notation $x \leq y$ is interpreted as the ordering relation on the <u>moduli</u> of the two numbers. A distinguishing characteristic of complex numbers is that there is an operator ι with the following properties, comparable to those of the negation operator A10–A14:

C1. $\iota(\iota x) = -x$

C2. $\iota(x+y) = (\iota x)+(\iota y)$

C3. $x \times (\iota y) = \iota(x \times y)$

C4. $x \leq \iota x \leq x$

3.5 Characters.

In some computer languages, the concept of a character is accepted as a primitive data type, or as the primitive constituent of structures such as strings. There are no intuitively appealing operations which can be performed upon characters; but it is useful to define a total ordering on the character set, and to ensure that the set is finite. This can be done by postulating only the axioms O1–O5, F1–F5, and if desired, K_4 and K_5.

3.6 Boolean Algebra.

The familiar Boolean type of ALGOL 60 can be seen to satisfy all the general axioms of section 2, provided that following interpretations are given:

1. $x \leq y$ stands for $x \supset y$
2. $x+y$ stands for $x \vee y$ and $x \times y$ stands for $x \wedge y$
3. approximate equality is interpreted as exact quality
4. min=0 stands for <u>false</u> and max=1 stands for <u>true</u>

Note that A7–A13 are vacuous for the Boolean type, since subtraction, negation, and division are not Boolean operations.

The following supplementary axioms appear to be valid -

B1. $x \doteq y \supset x = y$

B2. $x+1 = 1$

B3. $x+(\neg x) = 1$

B4. $x \times (\neg x) = 0$

B5. $x+(y \times z) = (x+y) \times (x+z)$

B6. $\min = 0 \wedge \max = 1$

B7. $x = 0 \vee x = 1$

In more recent programming languages, the programmer is given the capability of performing logical operations on words of the store, taking advantage of the fact that a word is an array of bits, and that hardware instructions are available to operate simultaneously on all the bits of the word. These operations are often regarded as highly machine-dependent, since their results depend rather more immediately upon wordlength than is the case with the normal range of integers. However, if the axiomatic method is used, there is no problem in leaving the wordlength indeterminate, and yet prescribing in every other way the behaviour of the values under the usual operations. In fact, the relevant axioms are exactly those of Boolean Algebra. These can be obtained (with some redundancy) by omitting exactly two axioms, namely O3 (total ordering) and B7, and adopting the convention that 0 stands for an all-zero word and 1 stands for a word consisting of all one bits.

It can be shown that if 0 has exactly N successors, there will be $2\uparrow N$ elements in the Boolean algebra. The number N is of course, equal to the number of bits in the word of the computer on which the algebra is implemented.

4. Transfer functions.

In the previous sections, each set of axioms has been intended to apply only to values from the single type in question. In order to deal with operations on values of mixed type, most programming languages specify certain standard "transfer" functions, which map the elements of one type into another. These transformations may be defined axiomatically in a manner similar to other homeomorphisms between structured sets in mathematics.

4.1 Integer to floating point.

The transfer of integers to floating point is known as "float", and has the following properties:

$$\text{T1 float } (0) = 0$$

$$\text{T2 float } (1) = 1$$

$$\text{T3 float } (i + j) \doteq \text{float } (i) + \text{float } (j)$$

In an implementation in which the size of the floating point mantissa is greater than that of an integer, the approximate equality of the third axiom may usually be strengthened to full equality. In this case, the three axioms give a complete characterisation of the float function.

4.2 Floating point to integer.

The primitive transfer function is taken here to be the "entier" function of ALGOL 60.

$$\text{T4 } x-1 \leq \text{float}(\text{entier}(x)) \leq x$$

4.3 Relational Operators.

In a language such as ALGOL 60, the relational operators < and = are defined to give as result a value of type Boolean. Thus the following axioms are required

$$\text{T5 } (x=y)=1 \equiv x=y$$

$$\text{T6 } (x\leq y)=1 \equiv x\leq y$$

Note that the bracketed occurrences of the = and < signs are new symbols representing dyadie transfer functions. They have a different meaning to the other occurrences; and consequently the axioms are not so tantologous as they appea

It is unfortunate that in some implementations of integer arithmetic, axiom 6 is not true, and must be replaced by

$$\text{T7 } (x\leq y)=1 \equiv 0<y-x$$

This gives a different answer (or fails to give an answer at all) in the case of intege overflow.

4.4 Integer to Boolean Algebra.

One of the useful transfer functions between integers and the elements of a Boolean algebra is one which ranges over the successors of the zero of the algebra. This function could be named "bit", and in a normal implementation it would map an integer i onto a word which had a 1-bit in the i^{th} bit position, and zeroes elsewhere. A more machine-independant description may be given as follows

T8 $1 \leq i \leq \text{wordlength} \supset 0 \text{ S } \text{unitset}(i)$

T9 $i \neq j \supset \text{unitset}(i) \times \text{unitset}(j) = 0$

Wordlength is assumed to be an integer constant (environment enquiry), stating the number of bits in the word of the computer.

Conclusion.

This paper has illustrated the use of the axiomatic method on characterising the primitive operations and operands of a programming language. About 60 axioms have been quoted, covering some five important data types and some ten relations and operations defined upon their elements. In addition five examples of transfer functions have been given.

No attempt has been made to prove consistency or completeness of the axiom sets; consistency is verified by any accurate implementation of the operations; and completeness would be contrary to our primary objective. Even the usual studies of independence of axioms is not wholly relevant if the purpose of the axioms is to demonstrate the essential similarities and differences between the various types of data, and the way in which they are treated in various languages.

However, further studies could profitably be devoted to the investigation of the range of possible models for the machine-independent axioms, and the discovery of elegant supplementary axioms which categorically define each of these models. It may also be possible to find additional machine-independent axioms, particularly for real and complex arithmetic, to assist a numerical analyst to prove the validity of his algorithms.

References.

1 I S O

2 Hoare, C.A.R. The Axiomatic Method (unpublished)

3 Laski, J.G. Sets and other types ALGOL Bulletin

4 vanWijngaarden. BIT.