Incomputability

C. A. R. HOARE AND D. C. S. ALLISON

The Queen's University of Belfast,
Belfast, Northern Ireland

Russell's logical paradox, formulated in terms of English adjectives, is considered as a convenient starting point for this discussion of incomputability. It is shown to be impossible, under a wide variety of circumstances, to program a function which will determine whether another function written in the same programming language will terminate. The theory of types is introduced in an attempt to evade the paradox Finally, it is shown that any language containing conditionals and recursive function definitions, which is powerful enough to program its own interpreter, cannot be used to program its own *terminates* function.

Key words and phrases: incomputability, logical paradox, termination, recursion, programming languages, theory of types, interpreter.

CR categories: 1.3, 5 27

1. INTRODUCTION

This paper attempts to stimulate the interest of practicing programmers in some of the ideas and methods of the theory of computation. It explores the fundamental limitations on the power of programming languages and computers. Other practical limitations, such as cost, are not covered in this paper. Nevertheless, programmers have been known to attempt solutions to problems which are probably unsolvable; the existence of such problems should be of interest to all programmers.

The purpose of the theory of computation (like all theories) is to prove results of the greatest possible generality—not just about languages like LISP or FORTRAN, but about all languages and all computers, even those not yet invented. This raises the following difficulty: to prove a problem *can* be solved in a given language, one is only required to write the program and prove that it works; but in order to prove that a program *cannot* be written requires a more indirect method of proof.

The proof method that we will use is the proof by contradiction (reductio ad absurdum). We first assume that the problem *can* be solved, and that a program to solve it is available. Using this program as a subroutine, we construct another program, and prove that this constructed program: 1) must terminate, when executed; 2) cannot give the answer "yes"; 3) cannot give the answer "no"; and 4) must give either the answer "yes" or "no."

This is obviously impossible; therefore our original assumption must have been fallacious, and we conclude that the subroutine cannot be written in the language in question. For this reason whenever we begin a proof with the words "Let us assume that . . . " the reader should be warned not to place too much faith in the assumption.

The constructions on which these proofs are based will, of course, seem to be very artificial. The reason for this is that they have been developed solely for the purpose of the proof, and have deliberately been kept very simple. The proofs also are alarmingly short, and for that reason may seem less

CONTENTS

convincing. Nevertheless, the reader can assure himself of the flawlessness of the proofs by rigorously checking every step.

In this paper we draw on the programming language LISP [7] for writing most of the example programs. We have chosen this language for the following reasons: 1) it is fairly widely known, 2) it is a small and simple language, 3) it is easy to represent LISP programs as data (*S*-expressions) which can then be manipulated, 4) it is relatively easy to write an interpreter of the language in LISP itself, and 5) it might initially seem to be easy to write a program which tells whether another program will terminate or not.

For the purposes of this paper, only a "reading" knowledge of LISP is required; it is not necessary to have programming experience in it.

2. THE RUSSELL PARADOX

There is a well-known paradox in logic, discovered by Russell [1], that Grelling [2], [3] formulated in terms of English adjectives:

> An adjective in the English language is said to be autological if it truly applies to itself. For example, "polysyllabic" has many syllables, and is polysyllabic; it is therefore, by definition, autological.

An adjective which is not autological is said to be heterological. For example, "long" is not a long word; it does not apply to itself, and is therefore plainly heterological. The following table gives some further examples (it is obvious that most adjectives will be heterological).

Autological	Heterological
short	abstract
understandable	monosyllabic
English	French

We now ask the question: is "heterological" autological or heterological? There are two possible answers:

> 1) Suppose we answer that it is autological. Then, by definition, it applies to itself. If "heterological" applies to itself, it must (by the definition of heterological) be heterological. This

shows that our answer is self-contradictory.

2) Suppose we answer that "heterological" is heterological. This obviously means that it applies to itself. By definition, it is therefore autological. This again contradicts our answer.

Thus both possible answers to the original question inevitably lead to self-contradiction. What is the solution to the question? Philosophers have advanced many ingenious ideas, but the simplest solution is just not to give any answer at all to such a question. Thus the only conclusion that we can draw from the paradox is that no answer can be given to this question.

3. THE COMPUTER'S REACTION TO THE RUSSELL PARADOX

We will now examine how a computer handles a question such as the Russell paradox. We first define the predicate "heterological" in a language that the computer can understand, for example LISP[4], a list processing language. The definition is:

heterological $(p) = \neg\ (p(p))$.

This definition tells the computer that to find out whether p is heterological it must be applied to itself $[p(p)]$, and the answer must be negated. Next, we ask the computer to evaluate the function call

heterological (heterological)

i.e., to answer *true* or *false*.
The computer will then try to compute

$\neg$ (heterological (heterological)).

In order to do this the computer must first evaluate the expression in brackets; which is by definition:

$\neg$ (heterological (heterological))

To accomplish this, it must first evaluate ..... and so on. The calculation never finishes; the computer continually tries to compute the expression in brackets, but each time it finds the same difficulty. Thus, the computer will never give any answer at all to the question—which is, as we have already suggested, the only sensible response!

4. THE PROBLEM OF TERMINATION

The Russell paradox has shown that there are some questions which a computer will be unable to answer; and that some easily definable functions lead to infinite computations when a computer tries to evaluate them.

Let us investigate the factorial function defined by

$\text{fac}(x) = \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x * \text{fac}(x-1)$

and consider what answers the computer will give when asked to evaluate fac(0), fac(3), fac(904379621) and fac(−1). This is an example of definition by recursion in which the name of the function being defined (fac) is used in its own definition. This appears to violate a basic principle of valid definition; a word cannot be defined in terms of itself. Nevertheless, this form of recursive definition is now widely accepted in mathematics, because of certain basic characteristics of functions. The essential characteristic of a function is that it specifies for each value in some domain a particular value in its range; a function may therefore be regarded as fully defined if we know what this specified value is in every case. A recursive definition of a function, in spite of its apparent circularity, is quite powerful enough to achieve this, as the following example illustrates:

$\text{fac}(0) = \textbf{if } 0 = 0 \textbf{ then } 1 \textbf{ else } 0 \times \text{fac}(-1)$
$= 1$
$\text{fac}(3) = \textbf{if } 3 = 0 \textbf{ then } 1 \textbf{ else } 3 \times \text{fac}(2)$
$\text{fac}(2) = \textbf{if } 2 = 0 \textbf{ then } 1 \textbf{ else } 2 \times \text{fac}(1)$
$\text{fac}(1) = \textbf{if } 1 = 0 \textbf{ then } 1 \textbf{ else } 1 \times \text{fac}(0)$
$\text{fac}(0) = \textbf{if } 0 = 0 \textbf{ then } 1 \textbf{ else } 0 \times \text{fac}(-1)$

giving $\text{fac}(3) = 3 \times 2 \times 1 \times 1 = 6$. We note that fac(−1) in the first and last lines is never evaluated since the first proposition is true.

In the evaluation of (fac(904379621)), the computer will undoubtedly fail to give the answer, since the numbers involved in the calculation are too large. In fact, the

answer is a number with several hundred million decimal digits, and apart from the expense, it would take the fastest modern computers many years to perform the multiplications. There are therefore practical reasons why present computers will never give an answer to the third question; although it is possible to envisage advances in computer technology that would put this calculation within the bounds of practicality. In the evaluation of (fac(−1)), the computer would again fail to give an answer, and would go on computing until it was stopped (or broke down). In this case, however, even the advance of computer technology could do nothing to save the situation. The fastest possible computers could never complete the calculation, but would continue multiplying by larger and larger negative numbers, ad infinitum. In such a case, we say that the computation is *nonterminating*, or that the application of the function fac to a negative argument is *undefined*.

Consider a quotient function defined as follows:

$$\text{quot}(x, y) = \textbf{if } x < y \textbf{ then } 0 \textbf{ else } 1 + \text{quot}(x-y, y)$$

This performs a division of two positive integers (x/y) by the simple but inefficient technique of counting how many times the divisor can be subtracted from the dividend. Let us investigate what response the computer will give when asked to evaluate quot(7, 3), quot(910763492, 2), and quot(17, 0). In the first case the answer is quite simple, i.e. quot(7, 3) = 2, since:

$$\text{quot}(7, 3) = \textbf{if } 7 < 3 \textbf{ then } 0 \textbf{ else } 1 + \text{quot}(4, 3);$$
$$\text{quot}(4, 3) = \textbf{if } 4 < 3 \textbf{ then } 0 \textbf{ else } 1 + \text{quot}(1, 3), \text{ and}$$
$$\text{quot}(1, 3) = \textbf{if } 1 < 3 \textbf{ then } 0 \textbf{ else } 1 + \text{quot}(-2, 3).$$

However, in the case of quot(910763492, 2), although the computation will theoretically terminate, it is possible that the integer limit of the computer will be exceeded. This machine limitation is of no profound significance since it is always possible to use a machine with larger word length, or revert to multilength working. Finally, the third case (quot(17, 0)) shows how the computer will loop forever. This nontermination corresponds to the familiar mathematical understanding that the illegal operation of dividing by zero has no defined result.

Of course, if a programmer wishes to obtain the result of a calculation, it can only be an error on his part if he programs the calculation in such a way that it does not terminate. In the case of operations "built in" to the hardware of the computer, there is usually a check against illegality. For example, an attempt to divide by zero is usually trapped, the program is immediately halted, and a message is printed out that an illegal operation has been attempted. However, if the programmer defines a function that does not terminate, there is no such helpful message printed out. The computation is merely continued until it is stopped or the machine breaks down. Furthermore, at this point the programmer may not know whether or not a few seconds more computation time might have produced the desired result. Thus, he cannot always tell whether the reason for the computer's failure to give an answer was merely the practical one of insufficient time allowed, or the theoretical one of trying to obtain a result that is in principle undefined.

This suggests that it might be a good idea to design and write a program that will answer this very question and thus help other programmers to find faults in their programs. Let us call the function that will determine whether the program terminates "terminates (f, x)," and let us further suppose that it gives the answer *true* if the computation $f(x)$ will eventually (in theory) terminate, and the answer *false* if it will never terminate. Of course, the function "terminates" itself should be designed always to terminate; otherwise its general usefulness would be seriously impaired.

If the function "terminates" is applied to some of the examples we have previously investigated, then the computer should give the following results:

$$\text{terminates (fac, 0)} = \textit{true}$$
$$\text{terminates (fac, 3)} = \textit{true}$$
$$\text{terminates (fac, 9807671932)} = \textit{true}$$
$$\text{terminates (fac, } -1) = \textit{false}$$

The third example presents the theoretical answer,"*true.*" The actual answer may be "*false*" because of the integer limit of the computer.

5. THE FUNDAMENTAL THEOREM

If we could succeed in programming the "terminates" function, we could use it to reprogram the problem of heterology. A function is said to be autological if it terminates when applied to itself and delivers the value *true*, and heterological otherwise. Thus if the function does not terminate when applied to itself it is always heterological. If it does terminate when applied to itself it will be heterological if and only if it delivers the value *false*. Thus we can program the function as follows:

$$\text{hetero}(f) = \textbf{if terminates } (f, f) \textbf{ then } \neg\, f(f) \textbf{ else true} \quad (1)$$

Consider the application of this function to itself, i.e., hetero(hetero). By the definition of "hetero" in (1) we have:

$$\text{hetero(hetero)} = \textbf{if terminates (hetero, hetero) then } \neg\, \text{hetero(hetero) else true} \quad (2)$$

To evaluate this expression we reason as follows:

1) "hetero", as defined in (1) *always* terminates, independent of what function f is given as the argument, since:
   a) by specification "terminates" always terminates;
   b) "true" and "¬" always terminate since one of them is constant, and the other is a primitive operator defined for all arguments;
   c) $f(f)$ is evaluated only when it is already known that terminates (f, f) is true, i.e., the evaluation of $f(f)$ terminates.
2) Since "hetero" terminates for all f, it terminates when applied to itself. Consequently (by the specification of "terminates") terminates (hetero, hetero) = *true*. Therefore by the definition of the conditional:

$$\textbf{if terminates (hetero, hetero) then } \neg\, \text{hetero(hetero) else true} = \neg\, \text{hetero(hetero)}. \quad (3)$$

Combining (2) and (3) we have:

$$\text{hetero(hetero)} = \neg\, \text{hetero(hetero)}. \quad (4)$$

This time it is not possible to evade the contradiction by merely stating that when "hetero" is applied to itself the computation never terminates; for we have already established that "hetero" must terminate provided that the "terminates" function works as specified. We must therefore seek another method for resolving this mathematical contradiction.

The correct procedure for resolving a mathematical contradiction is to question at least one of the assumptions on which it is based. The biggest assumption that underlies the contradiction given above is that we have succeeded in programming a function "terminates" with the assumed specification. The denial of this assumption is widely regarded as the fundamental theorem of incomputability. We restate it clearly [5], [6].

THEOREM: It is impossible in LISP to program a function t with the following properties:
   a) it takes two arguments, f and x;
   b) it takes the value *true* when the application of f to x is a terminating computation and *false* when it is a nonterminating computation:
   c) the function t itself terminates for all arguments f and x.

This impossibility theorem should be compared with other celebrated impossibility theorems in mathematics and physics.

1) It is impossible to find two whole numbers m and n such that $m^2 = 2n^2$.
2) It is impossible to find a fraction m/n which expresses the ratio between the radius and the circumference of a circle.
3) It is impossible to trisect an angle using only a ruler and a compass.
4) It is impossible to build a perpetual motion machine.
5) It is impossible to accurately observe both the position and the speed of an electron.

The significance of this result is two-fold. First, it shows that it is possible to rigorously specify and define a function which is impossible to program. Thus, there is at least one well-defined task which a computer is, in principle, incapable of ever performing. As we have seen, there are many other tasks which are just too lengthy for practical performance; nevertheless, the existence of a theoretically unsolvable problem is a fact that places an ultimate limitation on the power of a computer.

The second significant point is that the incomputability of the "terminates" function can be used to prove that many other potentially useful functions can never be programmed. For example, it is not possible to write a program that will take as parameter a mathematical proposition and decide whether it is a theorem or not. To prove this, we first assume the contrary, namely that we *can* write a program to test whether a mathematical proposition is a theorem or not; and then using this program as a subroutine, we show how to write the "terminates" function. Since we know it is impossible to write a "terminates" function in this case, we conclude that the mathematical theorem-tester can never be programmed.

For example, note that a mathematical theorem can be represented in a programming language by a Boolean function which always terminates and delivers the answer *true* for all values of all its parameters. (For simplicity let us consider only theorems which do not assert existence.) Consider the theorem

$$x + y = y + x$$

which can be expressed as the function "commutative" defined:

$$\text{commutative}(x, y) = (x + y = y + x).$$

Now assume we have a function theorem(f) which always terminates, and delivers the value *true* if and only if f is a Boolean function which *always* delivers the value true. If this assumption is valid, we can readily define a terminates function:

terminates (f, x) = theorem
(**if** $f(x)$ **then true else true**).

The argument of "theorem" (i.e., **if** $f(x)$ **then true else true**) terminates exactly when $f(x)$ terminates, and delivers the value *true* whenever it terminates. Thus "theorem" delivers the value *true* whenever $f(x)$ terminates, and the value *false* whenever it doesn't. Since we know that such a function can never be programmed, it follows that the "theorem" function can never be programmed.

6. WEAKER PROGRAMMING LANGUAGES

In the derivation of the fundamental theorem we used a rather powerful programming language (LISP) in order to write the "hetero" function and derive the necessary contradiction. The theorem should therefore be qualified to apply only to languages which contain at least: 1) recursive function definition; and 2) arbitrary functions as parameters of functions. The question therefore arises whether the theorem also applies to other languages, either more or less powerful. If we define a more powerful language as being capable of programming every function which can be expressed in the less powerful language we can easily show that the theorem applies; if it is capable of programming the "terminates" function then it will also be capable of programming the "hetero" function, and the resulting contradiction shows that the theorem still applies.

Therefore, perhaps surprisingly, the only hope of evading the impossibility theorem is to devise a *less* powerful programming language in which the "hetero" function could not be programmed at all. This can certainly be done for trivially simple languages. For example, suppose that the language did not permit any functions except constants; i.e., each function delivers a fixed value independent of the values of its parameters. All functions programmed in this language obviously must terminate for every value of their argument. Thus the "terminates" function can be programmed as the function which always gives the result *true*

terminates (f, x) = *true*

which is, as it should be, a constant function. However, the "hetero" function cannot be programmed in the language, and thus no contradiction arises. Therefore we have found a weaker programming language to which the fundamental theorem does not apply.

Of course, a programming language which permits only functions defined as constants is absurd, and completely useless as a means of programming a computer. However, it is possible to devise a subset of ALGOL for which the terminates function is always *true*. In this subset there would have to be:

a) no **go to** statements (or only those that jump forward);
b) no recursion;
c) no **while** loops; and
d) in a **for** loop, the counting variable, step, and the limit must be simple variables which cannot possibly be changed inside the body of the loop.

This weakened version of ALGOL 60 has been carefully defined to ensure that a suitably clever compiler can be constructed to check whether or not a program satisfies the restrictions; so that nonterminating programs will be rejected before execution has begun. This is most important both in theory and in practice, since it is futile to define a language in which nonterminating programs are excluded if it is then impossible to tell whether a given program text is expressed in accordance with the rules of the language or not.

In view of the importance of avoiding nonterminating programs, considerable investigation has been made into powerful programming languages in which nonterminating programs cannot be expressed. Most practical programs can readily be formulated in such languages, but unfortunately not all. For example, in our ALGOL subset we write a function:

integer procedure *mu(f);*
Boolean procedure *f;*

which delivers a result equal to the smallest positive integer n such that $f(n) = true$. Since there is no indication given in advance of how many numbers will have to be tested, it is impossible to set up the necessary loop. In fact, such a program will be impossible to write in any language which permits only terminating functions, because there is always a risk that the *mu* function will fail to terminate; that is, when its actual parameter is a function which always gives the answer *false*.

If we do not wish to exclude all nonterminating programs, we can design a language which prohibits the construction of curious functions such as "hetero," and thereby leaves open the possibility of writing a "terminates" function. (Some readers may wish to skip or skim through the remainder of this section.)

The curious feature of the "hetero" function is that it contains an application of function *f to itself*. This appears to be the basic reason for the paradox, and would be unusual in any practical program. We therefore want to design a language in which no call of a procedure is allowed to pass itself as a parameter, either directly or indirectly. One way of enforcing this restriction is based on the theory of types, which was formulated by Russell to evade similar difficulties in set theory. A type in ALGOL 60 is a range of possible values for variables; for example, the Boolean type comprises the truth values *true* and *false*; and each variable has a type associated with it by the programmer. We can also associate with each function a type, which specifies not only the type of its result, but also the types of *all* its parameters. For instance, the type of a function

exclusive or (x, y)

would be:

function of a pair of Booleans
onto a Boolean.

A compiler can check every call of this function for the correct number of parameters of the correct type; any text which violated this check would not be accepted as a program in our language.

This strict type discipline prevents our paradox by incorporating the fact that when a procedure has a procedure parameter, the definition of the type of the procedure must contain the expected type of its parameter. This increased strictness of type specifica-

tion is required in ALGOL 68, but not in ALGOL 60. Thus the function:

$$g(f) = f \text{ (true)}$$

has as its type

Boolean function of one parameter (being a function of Boolean onto Boolean).

Again, the conformity of any actual parameter to the expected type can be checked by a compiler. For example, g (not) would be accepted (and when evaluated yield *false*); but the following would be rejected:

g(or)—parameter function expects two parameters, not one;
g(cos)—parameter function defined on reals, not Booleans;
$g(g)$—parameter function defined on a function, not on Boolean.

The first two examples represent obvious programming errors, and far from complaining that the language is too weak, most programmers would appreciate a compiler that rejects these functions. However the third example, the one we are currently most interested in, is really just as meaningless as the others, as can readily be seen by applying the definition of g twice:

$$g(g) = g(\textbf{true}) = \textbf{true}(\textbf{true}).$$

The attempt to apply a constant as if it were a function is an obvious absurdity. We can readily see that the function "hetero" can never be directly programmed in any language which forces the programmer to specify fully the types of all functions and parameters. The proof of this assertion is based on the fact that the type of an actual parameter must always have a *shorter* definition than that of the procedure to which it is passed—indeed the former must be properly contained in the latter. Thus, it is impossible to ascribe a type to the formal parameter of "hetero" in a manner which would permit this parameter to be applied to itself.

By adopting an approach based on the theory of types, we are able to see that the response of the computer to the question "is 'heterological' heterological?" is not merely to say nothing, but to reply explicitly to the effect that: "Your question contains a contextual type error; it is ungrammatical and therefore meaningless. I shall not attempt to give an answer."

This is in many ways a highly satisfactory solution to the paradox. Nevertheless, it does make the language a bit cumbersome, and poses certain difficulties in the construction of some general classes of program, including the "terminates" function. For instance, in the declaration of "terminates (f, x)", one would have to declare the types of f and x, and one would therefore need a separate "terminates" function for each valid pair of arguments.

The question now arises whether our strictly typed language is capable of writing any of its own "terminates" functions; the simplest functions being of the following type:

Boolean function of two parameters, one being a Boolean function of one Boolean parameter and the other being a Boolean.

The result is that regardless of the care with which the type constraints have been formulated, it is still possible (under certain rather general conditions) to reformulate the "hetero" function, even in a strictly typed language, and thus reestablish the incompatibility theorem for the "terminates" function. This is demonstrated in the following section.

7. THE FINAL REFINEMENT

Let us consider a language even simpler and apparently less powerful than the strictly typed language described above; in fact, a language which does not allow procedures as parameters of procedures at all. Checking type constraints for such a language is exceptionally easy. At first consideration, this restriction appears rather severe, and apparently prohibits the programming of the "terminates" function from the beginning, since this function itself requires a function as the first of its two parameters. Nevertheless, it is possible to specify an

analogous function "terminates*" which takes as its first parameter not a function itself, but some ordinary data item which may be regarded as *representing* a function.

The representation of a portion of program as a data item can be achieved in both high and low level programming languages. For example, in both ALGOL and FORTRAN the data item could be an array which contains all the characters of the program; in machine code the data item might be a binary program in the form of a block of bit patterns in the store. On the other hand, in a list processing language such as LISP the data item will be an ordinary list which is the *S*-expression (see [8] for a definition) corresponding to the function which it is required to test for termination.

If we define $R(f)$ to be the representation of a function f within some data space, then we can specify the "terminates*" function as follows:

$$\text{terminates*}\ (z, x)$$

yields result *true* if $z = R(f)$ for some f, and if the computation $f(x)$ terminates; and yields the result *false* otherwise. Applying the function "terminates*" to some of the examples involving the factorial function we see that terminates* (R(fac), 0) is *true* and terminates* (R(fac), −1) is *false*. On the other hand, terminates* (0, 0) is probably *false*, since 0 is probably not the representation of any function.

Thus it appears that in a language that does not permit functions as arguments it may still be possible to program a useful version of the "terminates" function. Nevertheless, it would be wise to check the impossibility of programming a version of "hetero" in this language. At first, this might appear to be ruled out, because the construction ¬ $f(f)$ is illegal in the language. However, it might be possible to construct some analog of this expression which has the same unfortunate effect.

Let us specify a Boolean function "interpret" which takes as arguments two items of data, z and y. If z is the representation of some Boolean function f, interpret (z, y) delivers the same Boolean value as $f(y)$; otherwise it gives the value *false*. Furthermore, if $f(y)$ does not terminate, then neither does interpret $(R(f), y)$.

If the function "interpret" works correctly, we have for all f and y

$$\text{interpret}\ (R(f), y) = f(y).$$

In particular we have:

$$\text{interpret}\ (R(f), R(f)) = f(R(f)).$$

Using the "interpret" function we can now define a function "hetero*" which is closely analogous to "hetero." A function is said to be hetero* if, when applied to its own *representation*, it either fails to terminate or terminates and delivers the value *false*, i.e.,

$$\text{hetero*}(z) = \textbf{if}\ \text{terminates*}(z, z)\ \textbf{then}\ \neg\ \text{interpret}\ (z, z)\ \textbf{else}\ \textit{true}$$

We now consider the question: "is 'hetero*' heterological?" i.e.,

$$\text{hetero*}\ (R(\text{hetero*})) = \textbf{if}\ \text{terminates}\ (R(\text{hetero*}), R(\text{hetero*}))\ \textbf{then}\ \neg\ \text{interpret}\ (R(\text{hetero*}), R(\text{hetero*}))\ \textbf{else}\ \textit{true}$$

Now "hetero*" always terminates, because: 1) "terminates*" always terminates (by specification); and 2) if interpret (z, z) is entered, then "terminates*" must have been *true*. Therefore the function of which z is a representation, when applied to z, must terminate. Consequently, by the specification of "interpret", interpret (z, z) must terminate.
Consequently, terminates* $(R(\text{hetero*}), x)$ is *true* for all x, and in particular:

$$\text{terminates*}\ (R(\text{hetero*}), R(\text{hetero*})) = \textit{true}.$$

Therefore,

$$\textbf{if}\ \text{terminates*}\ (R(\text{hetero*}), R(\text{hetero*}))\ \textbf{then}\ \neg\ \text{interpret}\ (R(\text{hetero*}), R(\text{hetero*}))\ \textbf{else}\ \textit{true} = \neg\ \text{interpret}\ (R(\text{hetero*}), R(\text{hetero*})).$$

But by the definition of "interpret" this equals

$$\neg\ \text{hetero*}\ (R(\text{hetero*})).$$

We have again derived the contradiction

$$\text{hetero}^*\ (R(\text{hetero}^*)) = \neg\ \text{hetero}^*\ (R(\text{hetero}^*))$$

and again this contradiction cannot be resolved by the nontermination of hetero* (R(hetero*)), since we have already shown that "hetero* " always terminates.

It is important to realize that for any programmed function f, $R(f)$ is a demonstrable item of data. In particular, if "terminates* " and "interpret" are programmable, then R(terminates*) and R(interpret) could actually be written, or punched onto cards and fed into a computer.

The conclusion to be drawn from this contradiction is that in a language which includes conditionals and function definitions with parameters either
a) it is impossible to program "terminates"; or
b) it is impossible to program "interpret"
This may be formulated as a theorem.

THEOREM: Any language containing conditionals and recursive function definitions which is powerful enough to program its own interpreter cannot be used to program its own "terminates" function.

The question now arises whether a specified language is powerful enough to write its own interpreter. If the language in question can write its own termination function, then it cannot write its own interpreter; but if it can write its own interpreter, then it cannot write its own terminates function. Therefore only one of these two programs is necessary to decide the question.

In LISP it has been shown [7] that it is possible to program the "interpret" function and hence that it is impossible to program the "terminates" function. Conversely, we can see that our ALGOL subset (section 6) can program its own "terminates* " function (i.e., the constant function always delivering *true*). Consequently, we know that the language is not powerful enough to write its own interpreter. Intuitively, we know it is going to be difficult to write an interpreter; we do not know in advance how many steps the execution of the program is going to take, and therefore we cannot set up the necessary loop. The prohibition against backward jumps prevents constructing an interpreter on the traditional step-by-step basis. The theorem proved in this paper provides an absolute proof to support our intuition; without this proof, it would be very difficult to prove an impossibility of this kind.

REFERENCES

1. RUSSELL, B. *Introduction to mathematical philosophy.* Allen and Unwin, 1919.
   A pleasantly readable work for those who wish to explore more deeply the relations between programming and the foundations of mathematics.

2 DE LONG, H. *A profile of mathematical logic* Addison Wesley, Reading, Mass. 1970.
   An excellent survey which contains much useful historical and bibliographic material. In particular, the Russell paradox and variations are discussed in some detail.

3 GRELLING, K "The logical paradoxes." *Mind,* **45,** (1936), 481–486.
   Although the paradox of the adjective "heterological" was formulated by Grelling in 1908, this later article contains a useful commentary on a number of logical paradoxes.

4 FOSTER, J M. *List processing.* Macdonald, 1967.
   A very readable monograph which deals with the techniques of list processing and compares some of the languages which have been developed especially for lists e g , LISP, SLIP, COMIT

5 DAVIS, M *Computability and unsolvability* McGraw-Hill, New York, 1958
   The first few chapters are particularly useful in the context of incomputability.

6 STRACHEY, C "An impossible program." Letter to the Editor, *The Computer Journal* **8,** (1965), 313.
   A short proof of the impossibility of writing a program which will determine whether any other program will terminate or loop

7. MCCARTHY, J. "Recursive functions of symbolic expressions and their computation by machine, part 1 " *Comm. ACM* **3,** 4 (April 1960) 184–195
   The formalism for defining functions recursively is shown to be useful both as a programming language (LISP) and as a vehicle for developing a theory of computation This paper should be studied in detail before any work in LISP is attempted

8 WOODWARD, P M AND JENKINS, D P. "Atoms and lists." *The Computer Journal* **4,** (1961), 47–55.
   This is an attempt to describe McCarthy's work on LISP in a form which is more understandable to the non-specialist