NPS52-83-010

NAVAL POSTGRADUATE SCHOOL
Monterey, California

A COMPUTER SCIENCE VERSION OF GÖDEL'S THEOREM

Bruce J. MacLennan

August 1983

Approved for public release; distribution unlimited

Prepared for:

Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

[illegible]

DAVID K. HSIAO, Chairman
Department of Computer Science

WILLIAM M. TOLLES
Dean of Research

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-83-010	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Computer Science Version of Gödel's Theorem		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Bruce J. MacLennan		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61152N: RR000-01-10 N0001482WR20043
11. CONTROLLING OFFICE NAME AND ADDRESS Chief of Naval Research Arlington, VA 22217		12. REPORT DATE August 1983
		13. NUMBER OF PAGES 15
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Gödel's Theorem, halting problem, incompleteness theorem, computability, formalism, decidability, paradoxes, decision problems, foundations of mathematics, logic

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We present a simplified proof of Gödel's theorem by appealing to well-known programming concepts. The significance of Gödel's result to computer science, mathematics and logic is discussed.

A Computer Science Version of Gödel's Theorem

B. J. MacLennan

Computer Science Department

Naval Postgraduate School

Monterey, CA 93940

Abstract:

We present a simplified proof of Gödel's theorem by appealing to well-known programming concepts. The significance of Gödel's result to computer science, mathematics and logic is discussed.

1. Introduction

Most computer scientists have heard of Gödel's Incompleteness Theorem, and many have seen it "proved." Yet, Gödel's theorem usually remains a mystery. The proof, as it's generally presented, is almost incomprehensible. Students usually come away with a feeling that they've somehow been tricked. They would probably ignore the theorem altogether, but they've been told that it's very important, that it sets limits on logic and rational thought. It is therefore particularly unfortunate that both the theorem and the proof are presented so mysteriously.

Part of the difficulty of the theorem is that it was written in the early 1930's, before the widespread use of general purpose computers. Gödel's Theorem is a theorem about computers — which was formulated before there were many computers. By using ideas familiar to any programmer, Gödel's Theorem can be made easily understandable. There is no excuse for doing without these concepts.

Section 2 presents an informal proof of Gödel's theorem based on ideas from programming. Section 3 generalizes the theorem to other questions of decidability, and Section 4 relates the results to symbolic logic. Finally, Section 5 discusses some wider implications of the theorem.

2. The Halting Problem

Before proving Gödel's theorem, I will prove an equivalent result due to Turing: The Halting Problem, which asks, "For any given program and any given input, is there is a way of determining whether that program will halt when given that input?" For many programs this can be decided on a case by case basis. What is wanted is a *general* procedure that, given any description of a program P, and any input x, will tell us whether P halts when given x. In particular, we want a *program* that can decide the halting question.

Before embarking on a search for a decision procedure for the halting problem, it is wise to ask if such a procedure is possible. The classic approach tc determine whether something is possible or not assumes its existence and shows that this assumption leads to a contradiction. Self-contradictory things don't exist.

The proof I'll discuss is related to the famous Liar's Paradox of Epimenides:

This statement is a lie.

Is the above statement true or false? If we assume it's true, then we must conclude that it's false (since it's a lie); if we assume it's false, then we must conclude it's true (since it's not a lie). Put simply, this statement contradicts itself. Since this statement can be neither true nor false, we must conclude that it makes no factual claim at all. It was considerations of paradoxes like this that led Gödel and Turing to their proofs.

In the statement of the theorem and its proof we will use a Pascal-like programming language called L. You will see, however that the result does not depend on the language used, and remains true for any realistic programming language.

Definition: A procedure, called 'Halts', is called a *decision procedure for the halting problem in L* if, given (1) any string p representing a program P in L, and (2) any string x, Halts(p,x) = **true** if $P(x)$ halts, and Halts(p,x) = **false** if $P(x)$ goes into an infinite loop.

Theorem: The decision procedure Halts for the halting problem in L is impossible.

Proof: Assume that the procedure Halts exists.

Halts(p,x) tells us whether the program P described by p will halt when applied to the string x. Since p is itself a string, and we want to know what will happen when P is applied to any string, we can legitimately ask what would happen if P were applied to p, that is, if $x=p$. This is answered by the procedure call Halts(p,p).

Now, consider the following definition of the procedure Q in L:

```
procedure Q(p) is
 definition of Halts
begin
 if Halts(p,p) then
  label: goto label
 else return;
end Q;
```

The meaning of "*definition of* Halts" is that the definition of Halts is to be placed at that position in the definition of Q; this permits Q to make use of Halts. The behavior of Q is as follows: if $P(p)$ halts, then Q goes into an infinite loop; otherwise Q immediately returns.

The character string above, which defines the procedure Q, is a program in L. Call this character string q:

$$q \;=\; \text{'}\textbf{procedure}\ \mathrm{Q(p)}\ \ldots\ \textbf{end}\ \mathrm{Q;}\text{'}$$

Since q is a string representing a program in L, it is a legitimate input to the procedure Q. Therefore we can consider the behavior of $Q(q)$. Let's trace the steps. $Q(q)$ first asks if Halts(q,q). Since Halts is a decision procedure for the halting problem, it returns a Boolean (**true** or **false**) result indicating whether the procedure represented by q will halt when applied to q. But the procedure represented by q is Q itself, so Halts(q,q) asks whether $Q(q)$ halts.

Now, either $Q(q)$ halts or it doesn't. If it halts, then Halts(q,q) returns **true**, and $Q(q)$ goes into an infinite loop. This contradicts our assumption that $Q(q)$ halts. Therefore, let's suppose the opposite, that $Q(q)$ doesn't halt. Then, Halts(q,q) returns **false** and $Q(q)$ immediately returns, i.e., halts. This again contradicts our assumption. Since there are only two possibilities ($Q(q)$ halts or it doesn't) and both lead to a contradiction, we conclude that our assumption, that the procedure Halts is possible, was contradictory. Therefore there can be no such procedure as Halts. *QED.*

We have skimmed over several issues in the proof of the halting theorem. In view of its remarkable nature, we will reconsider them. In particular, a proof by contradiction shows that *at least one* of the assumptions is contradictory; it doesn't tell us which one. In this case we made two major assumptions: that it is possible to write Halts in L and that it is possible to write Q in L. Maybe Halts can be written but Q can't.

I haven't been too specific about what is and isn't legal in the language L. I have said that L is typical in the sense that anything we prove about L will hold for any realistic language. In particular, observe that to be able to define Q, we only need to be able to do the following things:

- Call another procedure (e.g. Halts).
- Do a conditional test.
- Go into an infinite loop.
- Return immediately (i.e., *not* go into an infinite loop).

These are things that can be done in any realistic programming language. Hence we must conclude that there is no decision procedure for the halting problem for any realistic programming language.

Are there any other hidden assumptions in our proof? Yes, we have also assumed that our language can handle arbitrarily long character strings. Since programs can be arbitrarily long, this is necessary if we are to be able to write a Halts procedure that's applicable to all programs. What about languages that don't have character strings? The proof still goes through if there's any data type (say, integers) that is equivalent to arbitrary bit strings. The reason is that a character string is just a string of bits. It should be noted, however, that the proof does *not* go through if any resource, either space or time, is limited to a particular amount. Since all real computers have such limitations, the theorem *only* applies to idealized computers with unbounded resources.

3. Other Decision Problems

Having found that there is no decision procedure for the halting problem, a natural question to ask is: What other questions *can't* be decided by a program? The answer is: almost any property you care to name. Suppose you want to write a decision procedure Does_D(p,x) that, for any behavior D, tells you if p does D when applied to x. Then consider this procedure definition:

```
procedure Q(p) is
 definition of Does_D
begin
 if Does_D(p,p) then
   don't do D
 else do D
end Q;
```

In the case of the Halting Problem D was 'halts', so *don't do* D was accomplished by 'label: **goto** label' and *do* D was accomplished by '**return**'

Call this string q. By the same reasoning as before you can see that $Q(q)$ *does* D if and only if $Q(q)$ *doesn't* do D. Hence we have a contradiction. The only way the proof can fail to go through is if the language is so weak that it is not possible to write code that *does D* and *doesn't do D*. For any realistic programming language, and property D of interest, it will be possible to write Q in the language.

There are some properties of programs that *can* be decided by programs. For example suppose we wanted to write a function Halts_Quickly(p,x) which determines if p halts within 100 years when applied to x. Intuitively it seems like we ought to be able to write such a function: we just run $P(x)$ until it halts or until 100 years are up, whichever occurs first. Thus, Halts_Quickly(p,x) always returns an answer, although we may have to wait 100 years to get it!

But, can't we write a paradoxical procedure Q that runs less than 100 years only if it doesn't run less than 100 years? Let's try and do this. If Halts_Quickly(p,p) is true, we will go into an infinite loop (looping for over 100 years would be adequate); otherwise we will return immediately. The resulting program is:

```
procedure Q(p) is
  ... definition of Halts_Quickly ...
begin
  if Halts_Quickly(p,p) then loop forever
  else return;
end Q;
```

Now consider the application $Q(q)$. Does it halt quickly or not?

Suppose $Q(q)$ halts within 100 years; then, Halts_Quickly(q,q) returns **true**, and $Q(q)$ loops forever. This contradicts our assumption that $Q(q)$ halts quickly.

Now suppose $Q(q)$ does not halt within 100 years; then Halts_Quickly(q,q) is **false**, and $Q(q)$ returns immediately. Does this lead to a contradiction? No, since the running time of $Q(q)$ includes the time necessary to execute Halts_Quickly(q,q). This could well be more than 100 years (as it would be if we implemented it in the naive way described previously). Hence there is no contradiction.

This does not mean we *can* write Halts_Quickly, only that it does not lead to a contradiction in the same way as Halts. On the other hand, our informal implementation discussion shows that Halts_Quickly could actually be programmed.

Thus, the halting theorem does *not* say that it is impossible for a program to decide *any* property of all programs. In fact it is the *unlimited* properties (e.g., halts *eventually)* which are undecidable; *limited* properties (e.g., halts *within 100 years)* are often decidable.

4. Symbolic Logic

You have probably heard that Gödel's theorem has something to do with the limitations of logic; you may have even heard that it sets bounds on rational thought. Yet, I have not said anything about logic in the preceding sections. I have only discussed the limitations of what programs can tell us about other programs. To understand the connection between programs and logic, it will be necessary to discuss some topics in the foundations of mathematics.

A major concern of mathematicians in the 19th century was *rigor*, standards of proof by which mathematicians could ensure that their theorems were true. Mathematicians were anxious to secure the foundations of mathematics, to ensure that there were no contradictions implicit in mathematical theories. Mathematicians divided themselves into several, often antagonistic, schools depending on the approach to foundations that they took. One of the most important of these schools, founded by the famous mathematician David Hilbert (1862-1946), was called *formalism*.

It had long been observed that deductive reasoning was often *formal;* that is, the correctness of a deduction could be decided on the basis of the form of the argument without reference to the meanings of the terms used in it. This is particularly true of mathematical proofs: they are often accomplished by successively transforming a formula into new formulas. These transformations are simple mechanical rearrangements of symbols. For example, we can prove $x+1=5$ implies $x=4$ by these transformations:

$$\begin{aligned} x+1=5 &\Rightarrow (x+1)-1 = (4+1)-1 \\ &\Rightarrow x+(1-1) = 4+(1-1) \\ &\Rightarrow x+0 = 4+0 \\ &\Rightarrow x=4 \end{aligned}$$

The formalists believed that all mathematics could be reduced to *formal systems*. A formal system has two parts: (1) a set* of *initial strings*, and (2) a finite set of computable *transformation rules*. By applying the transformation rules to the initial strings, a formal system generates a set of *derived strings*. As you can see, a formal system is a primitive sort of program, in which (1) the transformation rules are the operations and statements of the program, (2) the initial strings are the input data, and (3) the derived strings are the intermediate and output data that result from applying the program to the input data.

The formalists believed that they could make mathematics rigorous by reducing it to a formal system. By letting the initial strings of a formal system represent the mathematical axioms, and the transformation rules represent the deductive rules of inference, the derived strings would represent just those theorems that could be deduced from the axioms. Since the transformation rules were mechanical symbol manipulation operations, independent of the meaning of the symbols, it seemed that this approach would eliminate both non-rigorous ideas and the use of intuition from mathematical proofs.

Another important goal of the formalist program was to establish the *consistency* of mathematics. *Consistency* means that it is not possible to prove both a proposition 'P' and its negation 'not P'. In other words, consistency means that there are no contradictions inherent in the axioms.

* The set of initial strings is required to be *recursive*, i.e., a computer program can decide whether a given string is an initial string or not.

Mathematicians are also concerned with the *completeness* of their axioms: how many axioms must be included in order to prove all true theorems? In other words, if a theorem is true, and can be expressed in the symbols of the formal system, it should be possible to derive it from the initial strings (axioms) by using the transformation rules (rules of inference).

Formal systems gave the formalists hope that they would be able to prove the consistency and completeness of some formalization of mathematics. This is because formal systems are themselves mathematical objects. Therefore, it is possible to prove things about formal systems by using mathematical techniques. In particular, they wished to show the consistency and completeness of mathematics by using rules of inference that were so simple that no one could object to them. They wanted to use *finitistic* rules, that is, rules that could be executed on a computer.

You may have already realized the dubiousness of this proposition. A formal system is essentially a programming language, and we know that it's impossible for a program to decide much of anything about a reasonably powerful programming language. Hence, if we conceive of mathematics as a formal system (i.e., programming language), then we can see that it will be impossible for mathematics to decide (prove) much of interest about any reasonably powerful formal system, including itself. Hence, it's not likely to be possible to use mathematics to prove the consistency and completeness of mathematics. This is in fact the case.

Let's consider how we could prove this result more carefully. Suppose the formal mathematical language is powerful enough to express the theorem 'p halts on x', for any program p (in some programming language) and for any input x. Then we know that this formal mathematical system must be incomplete, because otherwise we could either prove or disprove 'p halts on x' for each program p and input x. This would solve the halting problem, which we just showed is impossible. Hence, the central issue is whether our formal mathematical language is powerful enough to state the theorem

'p halts on x'.

Next I argue that the theorem 'p halts on x' can be stated in any reasonably powerful formal mathematical language. In particular, if we can talk about integers, sequences and sets of integers, and the arithmetic operations, then we can express 'p halts on x'.

First, since 'p halts on x' is a proposition about programs, it must be possible to talk about programs mathematically. A program in any programming language can always be written as a finite sequence of characters, and characters can be thought of as small integers (say, integers in the range 0 to 255 in the ASCII code), so any program can be represented by a finite sequence of integers, which is a mathematical object. If we assume that our programs take strings as inputs and return strings as outputs, then the inputs and outputs of programs can also be represented as sequences of integers.

We now know how to represent the program p and the input x; how do we express 'p halts on x'? What this means is that if p is applied to x, then some output y will result. Now, if we had a function 'apply (p,x)' that returned the result of applying p to x (assuming it halts) then we could express 'p halts on x' mathematically:

$$p \text{ halts on } x \text{ if and only if there exists a } y \text{ such that } y = \text{apply}(p,x)$$

Notice that 'apply' is just a mathematical function that takes two sequences of integers and returns a sequence of integers.

The 'apply' function is essentially an interpreter for our programming language. Can one define an interpreter using just mathematics? LISP programmers will realize that the answer is "yes," because programming in pure LISP is essentially programming in mathematics, and a LISP interpreter can be written in about twenty-five lines of LISP. If you are not familiar with LISP you may take a little more convincing.

Think about the way a program executes: at each stage of its execution it takes the values of a number of variables and computes new values for a (possibly overlapping) set of variables. This alteration in the variables is determined by the current instruction being executed in the program. Further, each instruction designates another instruction as the next one to execute.

Now, the memory that contains all the variables in a program is just an array of bits, so the memory can be represented as a sequence of integers. Also, the *instruction-pointer*, which designates the next instruction in the program to execute, is just a sequence of bits, so it can also be made part of this sequence of integers. Notice that this sequence of integers, which we call the *state* of the program, is all that changes from one step of the program to the next.

In order to avoid a lot of very tedious details, I have to skip some of the finer points of this argument. Therefore, suppose that we have defined a mathematical function 'Next_State' such Next_State(p,s) is the state resulting from executing one step of the program p in state s. (The instruction to execute is determined by the instruction-pointer part of s.) It should be fairly clear that it is possible to define such a function, because most programming language instructions just change a few variables, i.e., replace a few elements of the sequence of integers representing the state. Control-flow instructions (e.g., **goto**) just change the part of the state representing the instruction-pointer.

Assuming we have the Next_State function, it is easy to complete the argument. To do this we need to define the idea that, given states s and s', executing the program in state s will eventually lead to a state s'. This just means that it is possible to get from s to s' by executing zero or more steps in the program, i.e., we can get from s to s' by zero or more applications of Next_State. This is easy to express mathematically as a recursive definition:

Program p eventually takes s to s' if and only if

either $s = s'$

or there is a state s'' such that

$s'' = \text{Next\_State}(p,s)$ and program p eventually takes s' to s'

It is now simple to express the equation $y = \text{apply}(p,x)$:

$$y = \text{apply}(p,x) \text{ if and only if program } p \text{ eventually takes } x' \text{ to } y$$

where x' is x with the initial instruction-pointer appended in the proper place.

This completes the argument that the halting problem can be expressed in a reasonably powerful formal mathematical system. Thus, a reasonably powerful *formal* mathematical system, if it's consistent, must also be incomplete.

5. Relevance

Gödel's Incompleteness Theorem is often misinterpreted. In popular accounts it is often represented as a proof of the inherent limitations of logic and mathematics. In some quarters it is used as an excuse for irrationality and mysticism. Thus it is important to consider the relevance of Gödel's theorem, in particular to logic and mathematics.

Gödel's theorem has great relevance to computer science. In its form as the Halting Problem and its extensions, it tells us that we should not try to find algorithms that can infallibly decide certain questions about any given program. Thus, Gödel's theorem puts ultimate limits on the capabilities of computers.

What is the relevance of Gödel's theorem to logic and mathematics? It's major effect has been to destroy the formalist program in mathematics. A primary goal of that program was to prove, using mathematics, the consistency and completeness of mathematics; Gödel showed that this cannot be done. Gödel also showed that many other formal logical systems, such as three-valued logic, also suffer from the incompleteness property. This should not surprise us, since any reasonably powerful formal system is equivalent to a programming language, and so suffers from the undecidability theorems.

Does this then indicate an inherent and unavoidable limitation to logic and mathematics? Only if you are a formalist. In essence, Gödel's theorem says that formalism cannot capture all of the power of mathematics and logic.

Let's consider an example of the latter case. Consider Fermat's Last Theorem, which states that there are no integers x, y, and z such that for some $n>2$, $x^n+y^n=z^n$. This theorem has been neither proved nor disproved, although there is ample empirical evidence of its truth. Suppose that Fermat's Last Theorem were known to be undecidable in some reasonably powerful formal mathematical system. We would then have to conclude that the theorem is true. How can this be? Suppose (contrary to fact) that it were false. Then there would exist integers a, b, c, and $k>2$ such that $a^k+b^k=c^k$. Now, in all reasonably powerful formal mathematical systems the following is a valid deduction:

1. $k>2$

2. $a^k+b^k=c^k$

3. Therefore, there exists x, y, z, and $n>2$ such that $x^n+y^n=z^n$

The latter proposition is the contradictory of Fermat's Last Theorem. Hence, if Fermat's Last Theorem is false, then it will be decidable in any reasonably powerful formal mathematical system. Conversely, if Fermat's Last Theorem is undecidable in any reasonably powerful formal mathematical system, then the theorem must be true. *QED.*

What have we done here? It seems that we've taken an undecidable theorem and decided it. More precisely, we've proved a theorem that was not provable within the system. Sometimes this kind of proof is called "meta-mathematical" or "meta-logical," but these are misnomers. The prefix 'meta-', suggests that some sort of unusual process has been used. In fact we've just used the plain, old, garden-variety, Aristotelian logic that's been known for 2500 years. Our proof would be clear to Euclid.

Thus, there is nothing superior or esoteric about our meta-logical proof. It would be more accurate to call the deductions of a formal logical system *sub-logical* proofs.

The conclusion we should draw is that formal systems are not very good models for either mathematics or logic. Formal systems model situations where propositions can be deduced from other propositions without regard for the meanings of the terms in those propositions. One of the premises of formalism is that all mathematical truths can be derived by applying this formal deductive process to a fixed set of axioms (initial strings). This has never been the case in mathematics. On the contrary, although there is little debate on the laws of algebra or the calculus, there is an interminable debate on the choice of the axioms upon which to found mathematics.

The problem is that since the time of Euclid mathematics has been viewed as a purely *deductive* science in which theorems are deduced from given axioms and definitions. In Euclid's time these axioms and definitions were considered self-evident; the formalists considered them arbitrary. Thus, the axioms and definitions are considered the foundation of the edifice of mathematics.

In this deductive view mathematics is very different from the *inductive* sciences like physics and chemistry. In these the basic laws are neither self-evident nor arbitrary. Rather, they are the result of a lengthy process of scientific investigation. They are not the foundations of these sciences; they are the capstones.

In fact mathematics is more like the other sciences than is generally acknowledged. For eons man has known that apples drop from trees; it took Newton to explain this in terms of the more basic ideas of mass and gravity. Similarly, for eons man has known that 2+2=4; mathematicians are still proposing explanations of this fact in terms of more basic ideas. Mathematics, like the other sciences, proceeds by a combination of induction and deduction.

In summary, Gödel's Incompleteness Theorem is not a theorem about logic or mathematics; it is a theorem about programming. It places limits on the capabilities of computers, not on the capabilities of mathematics or logic. Contrary to the common notion that it demonstrates the impotence of reason, it is actually a sterling example of the power of reason.

6. Bibliography

1. Nagel, E. and Newman, J. R., *Gödel's Proof*, New York University Press, 1958.

   This is the most readable traditional account of Gödel's proof. It skips many of the tedious details, although anyone familiar with programming ought to be able to see how they could be filled in.

2. Hoare, C. A. R., and Allison, D. C. S., Incomputability, *ACM Computing Surveys 4*, 3 (September 1972), 169-178.

   This is the only other presentation of Gödel's proof that I am aware of that makes use of programming concepts to simplify the proof.

3. Kline, M., *Mathematics: The Loss of Certainty*, Oxford University Press, 1980.

   In this book a well-known mathematician traces the gradual destruction, culminating in Gödel's proof, of the mathematicians' belief that they have a special, non-empirical path to the truth.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52Hq Department of Computer Science Naval Postgraduate School Monterey, CA 93943	40
Professor Bruce J. MacLennan, Code 52Ml Department of Computer Science Naval Postgraduate School Monterey, CA 93943	12
Dr. Robert Grafton Code 433 Office of Naval Research 800 N. Quincy Arlington, VA 22217	1
A. Dain Samples Computer Science Division - EECS University of California at Berkeley Berkeley, CA 94720	1