Gödel's Incompleteness Theorem for Computer Users

Stephen A. Fenner*

November 16, 2007

Abstract

We sketch a short proof of Gödel's Incompleteness theorem, based on a few reasonably intuitive facts about computer programs and mathematical systems. We supply some background and intuition to the result, as well as proving related results such as the Second Incompleteness theorem, Rosser's extension of the Incompleteness theorem, the Fixed Point theorem, and Löb's theorem.

1 Introduction

In 1931, Kurt Gödel proved that any commonly used mathematical system is incomplete, that is, there are statements expressible in the system which are neither provable nor refutable in the system.[1] Such statements are known as *undecidable*. Further, he showed that no commonly used mathematical system can prove itself to be free of contradictions. Gödel's surprising and profound results disturbed much of the mathematical community then, and they remain unsettling today. Any of several celebrated open questions in mathematics—Goldbach's Conjecture, the Generalized Riemann Hypothesis, and the Twin Primes Conjecture, to name a few—may remain unsolved forever. Moreover, we may never know (there is no general procedure for knowing) whether or not a given open problem is decidable. Finally, we have no way of proving mathematically that the systems we use will never derive a contradiction.

Gödel's original 1931 paper spells out his proof in meticulous detail. Most of this detail, however, is concerned with clarifying two threads of contemporaneous research: (1) formalized mathematics, and (2) computation. Large chunks of the proof are concerned, on the one hand, with how formal mathematical syntax can be algorithmically manipulated, and on the other hand, how the notion of algorithm itself admits a rigorous mathematical description. Electronic computers did not exist at the time, and ideas about what is "computable" or "algorithmic" were in their infancy. No doubt, Gödel was extremely careful with these

*Computer Science and Engineering Department, University of South Carolina, Columbia, SC 29208 USA. Email: fenner@cse.sc.edu.

[1] A refutable statement is one whose negation is provable.

concepts in no small part because people at the time had little good intuition about them. The downside of so much detail is that it obscures the conceptually simple crux of the proof.

The current ubiquity of electronic computers and networks in our everyday lives has raised our collective intuition about computation considerably since the 1930s. Computer programmers certainly have a good sense of what computing can accomplish, but even people who just use computers routinely have a pretty good idea as well. This intuition can help us cut through most of the tedious detail in Gödel's original proof, quickly and neatly exposing its core idea. The current exposition will attempt to do this.

2 Preliminary Intuitions

I suggest that if you feel you have sufficient background, you skip this entire section on first reading and go straight to Section 4, except for skimming the four definitions of Computational Completeness, Computational Formalizability, Consistency, Computational Soundness, and Completeness later in this section. Section 4 contains the key idea of Gödel's result (cast in terms of computations), which can be understood with very little preparation.

2.1 Formal Mathematics

Mathematics strives to investigate abstract ideas using reasoning that can be agreed upon universally. Mathematicians communicate their results using various systems of notation and of proof. Theorems and proofs written in math journal articles usually mix informal prose, more formal symbolic expressions, and possibly diagrams. The prose and diagrams are useful to save time by appealing to certain *clear* intuitions on the part of the reader. Intuitions, regardless of how clear they may be to the writer of a proof, may not be shared by all readers, however. In principle, it should be possible to replace all appeals to intuition in a proof by a succession of mathematical formulas, adhering to strict syntactical rules, so that they can be checked for correctness by someone with no intuition for the subject whatsoever. (This strictly formal approach is never done in practice—at least not yet—because formally expressing theorems and proofs would produce formulas so long and cumbersome that checking them without the aid of intuition would tax one's patience, to say the least.) Thus to avoid any crucial dependence on intuition, which may not be universally held, mathematical communication is limited to what can ultimately be expressed completely formally, using a *formal mathematical system*.

A formal mathematical system consists of two things:

1. syntactical rules for expressing mathematical statements as strings of symbols over some alphabet of symbols—not unlike the rules describing correct computer programs written in some programming language—and

2. syntactical rules for establishing the truth of certain statements of the system. Truths established in the system are the *theorems* of the system.

The second component usually breaks down into two parts: axioms and rules of inference. Axioms are basic statements in the system that are taken to be true *a priori*. For example, the formula $(x + y) + z = x + (y + z)$ might be an axiom expressing the fact that addition is associative. The system should have rules for deciding whether or not a given statement is an axiom based solely on its syntax, disregarding any possible meaning intended for the statement. A rule of inference determines (again, based solely on syntax) when a mathematical statement can be inferred as a newly established truth (i.e., theorem) based on previous theorems. A rule of inference typically used in mathematical systems is *modus ponens*, written schematically as

$$\frac{A,\ A \to B}{B}$$

The rule states that if A is a theorem and $A \to B$ is a theorem, then B is also a theorem. We say that B follows from A and $A \to B$ by modus ponens. Here, the letters "A" and "B" stand for any statements in the system. The usually intended meaning of "$A \to B$" is "A implies B," or "if A then B," but the key point here is that the correctness of applying modus ponens can be checked based entirely on the syntactical structure of the statements involved, independent of meaning. We'll assume that modus ponens is a rule of inference in all formal systems.

A new theorem T in the system can thus be established by a finite sequence of statements of the system, where T is the last statement in the sequence, and each statement in the sequence is either an axiom of the system or else follows from previous statements in the sequence via some rule of inference of the system (e.g., modus ponens). Such a sequence is a (formal) *proof* of T in the system. Theorems are just those statements that have proofs.

A statement that has a proof is *provable* in the system. A statement whose negation has a proof is *refutable* in the system (i.e., an established falsehood). A system is *consistent* if no statement of the system can be both proven and refuted by the system. A system is *complete* if every statement of the system can either be proven or refuted. The First Incompleteness result of Gödel, which we'll see in Section 4, states that

> Any reasonable [see below] and consistent mathematical system is incomplete, that is, there are statements expressible in the system that can be neither proved nor refuted in the system.

Such statements are called *undecidable* or *formally undecidable* by the system. Gödel actually constructs an undecidable statement for a particular reasonable mathematical system, assumed to be consistent.[2] Actually, this is not quite what Gödel proved. As we will see, he had to assume a property of the system slightly stronger than consistency (but still evidently true for most systems in common use). Subsequent work by Rosser established the statement above as it stands (see Section 7).

The Second Incompleteness result of Gödel (see Section 5) states that

[2]Gödel used a formal system P based on Russell and Whitehead's *Principia Mathematica*. Other more commonly used systems include first-order Peano arithmetic (PA) and Zermelo-Fraenkel set theory (ZFC).

No reasonable, consistent mathematical system can prove its own consistency.

Neither of these results would be possible without the concept of a mathematical system with completely formal, syntactic rules for statement manipulation. Such systems are specified to such exactitude that they themselves can become objects of fruitful mathematical study. This is exactly what these two results do: treat mathematical systems as mathematical objects. In the next subsection, we'll see how computer programs can serve as intermediaries for this treatment.

2.2 Computer Programs

When run, computer programs process data. They can read data from, and write data to, various media, such as files on a disk or other computers via a network. They are good at manipulating text. Computer programs are themselves written as text in some kind of programming language, such as Fortran, Cobol, Basic, Lisp, Pascal, Ada, C, C++, Perl, Java, ML, Haskell, Prolog, SmallTalk, Python, Logo, Foxpro, *ad infinitum*.

Computer programs frequently process their own text (this is how compilers and interpreters work). For example, one program can, when run, read another program in the same language as input from a text file, and do any number of things with it: check its syntax, translate it into machine language (compiling), or simulate its execution (interpreting), to name a few.

2.3 Computer Programs and Mathematics

Computer programs also have highly predictable behavior. What a program does in any finite interval of time is completely determined, in a well-understood fashion, by the program itself and its input. A program's behavior can be expressed mathematically, and many facts about it and its behavior can be proven in commonly used mathematical systems.

On the other hand, we have seen that a formal mathematical system has statements given as strings of text, and purely syntactic rules for manipulating them. So one would expect that various computer programs can be written to process formal mathematics. For instance, a computer program should be able to read a string of text as input and check whether it forms a syntactically correct statement in the formal system, and if so, whether or not it is an axiom of the system. A computer program could read a sequence of statements as input and check whether or not it constitutes a (syntactically) correct proof in the system, relieving humans of this mind-numbingly tedious job. A computer program could also generate statements of the system as output.

Figure 1 depicts this reciprocity between mathematical systems and computer programs. We will exploit it to prove the two incompleteness results, but first we will describe it more carefully.

We fix a programming language (one of those listed earlier, say) and assume that all computer programs we talk about are written in this language. Let F be some mathematical

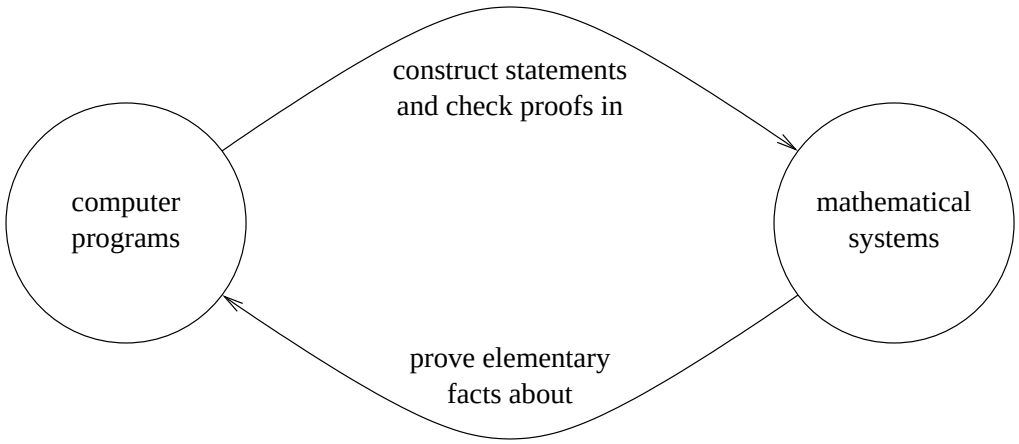

Figure 1: How computer programs and mathematical systems relate to each other

system. Two basic properties we will assume about F, and any other mathematical system we discuss, are these:

Computational Completeness. F is strong enough to express and to prove some elementary facts about the behavior of any computer program. In particular, if some computer program halts on some particular input in some given finite number of time steps, then one can express and prove this fact within F. Also, if the program does not halt on the input in the specified time, then this fact is also provable in F. Finally, F can prove whether or not a program outputs any particular given value in any given finite period of time. More specifically, if a program outputs some value and then halts without outputting anything else, then F proves this fact.

Computational Formalizability. The rules of F can be rendered in a completely formal, rigorous, concrete, "mechanical" way, so that statements and lists of statements in F can be read, manipulated, and generated by computer programs as described above, and also so that proofs in F can be checked for correctness by a computer program.

These are not outlandish claims, at least not Computational Formalizability, since formulas and proofs can be specified purely syntactically, and statements and proofs can be stored in, say, a text file that can be read and written by software. As for Computational Completeness, mathematics wouldn't be very useful if it couldn't prove such elementary facts as these. These two properties are exactly what was meant by a "reasonable" system in the statement of the first Incompleteness result, above. (An additional property is needed for the second result; see below.) They restate the reciprocal relationship in Figure 1. If we compose the two arrows together, we see that computer programs can use mathematics to analyze themselves, as shown in Figure 2. The proofs of the two incompleteness results will involve such programs.

We will fix once and for all some arbitrary reasonable (i.e., computationally complete and

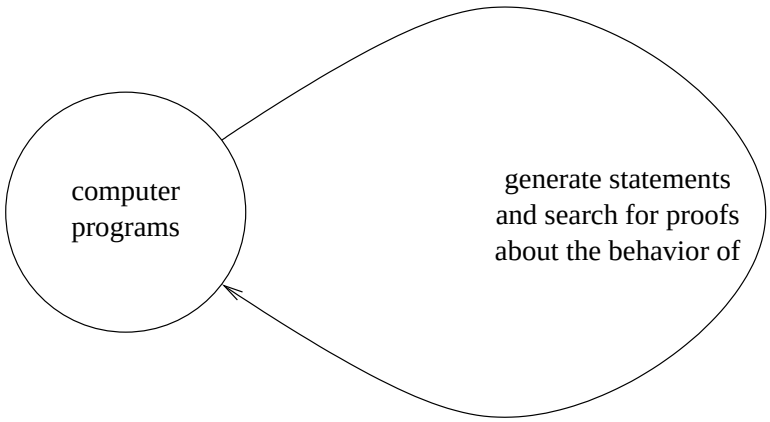

Figure 2: Computer programs can refer to themselves through mathematics.

computationally formalizable) mathematical system F. The Incompleteness theorem holds for any such F. These systems include first-order Peano Arithmetic (PA) and Zermelo-Fraenkel Set Theory (ZFC), among others.

Since F is computationally formalizable, we can assume that there are two computer programs W (short for "well-formed formula") and B (short for "Beweisung," the German word meaning "proof"). W reads a text file s and checks its syntax to determine whether or not s is a statement of the system F. It outputs "yes" or "no" accordingly, and halts. B reads two text files p and s. It checks the syntax of both files, and if s is a statement of F and p is a correct proof of s, then B outputs "yes" and halts; otherwise, B outputs "no" and halts. B is our proof-checking program, and uses W as a subroutine.[3]

Also by computational formalizability, we can assume that a program can "cycle through" all possible statements and proofs of F. After all, statements and proofs are just finite strings of computer text. So for instance we can exaustively generate strings of text one by one, then run W on each, discarding the ones where W outputs "no.". Running through all possible text strings can be done as follows: first run through all strings of length zero, then all strings of length one, then all strings of length 2, and so on. There is exactly one string of length zero, namely, the empty string. There are 256 strings of length 1, one for each of the 256 characters in the standard computer (ASCII) character set. There are $256^2 = 65{,}536$ strings of length 2, i.e., all possible pairs of ASCII characters, and so on. The point is, for any given length, there are only a finite number of strings of that length, so we will eventually see them all, then move on to the next higher length. Text strings can be of any length, so there are infinitely many of them, and so we cannot run through them all in a finite amount of time. But for any given string s, we will eventually encounter s if we cycle long enough. That is what we mean by "exhaustively."[4]

[3]How hard is it to write B? Not very. Any student who did well in a class on compilers or automated proof checking could probably write it in a few weeks. B can also be implemented so that it runs fairly efficiently, given the sizes of its inputs.

[4]Again, such a cycling program is not hard to write, but it will run a *very* long time before it gets to any

3 Consistency and Computational Soundness

There is another property that we desire for F:

Consistency. One cannot prove both a statement and its negation in F.

In other words, one cannot prove a contradiction. Inconsistent systems are not useful, since one can prove *any* statement given a single contradiction. Thus if F is inconsistent, then all statements in F are provable, whether or not they are true, so F is completely trivial and unsuitable for use by mathematicians. The systems mentioned above, PA and ZFC, are (almost) universally believed to be consistent.

Actually, for Gödel's original Incompleteness theorem, we will need a slightly stronger property for F than mere consistency:

Computational Soundness. If a given program runs forever on a given input, then there is no proof in F that that program halts on that input.

This is sort of a converse to computational completeness, which says that if a program halts, then F proves that it halts. Computational soundness means that if F proves that a program halts, then it really does halt. We would certainly not want F to be computationally unsound, since it would then prove false statements about the behavior of programs. Computational soundness implies consistency.[5] In Section 7, we will show how to get an undecidable statement based on consistency alone, without this additional assumption.

Finally, for completeness, we recall the following property that F will almost certainly *not* have, and give it a bold header:

Completeness. Every statement in F is either provable or refutable in F.

4 The First Incompleteness Theorem

We are now ready to prove Gödel's first Incompleteness theorem, which we can now state fairly precisely.

Theorem 1 (Gödel) *Let F be a computationally complete, computationally formalizable system. If F is computationally sound, then F is incomplete.*

Proof. Let P_0 be a computer program[6] that does the following:

1. P_0 reads as input a program P from a text file (it reads from a file and checks that its contents form a syntactically legitimate program).

nontrivial proof.

[5] For those who are technically knowledgeable, computational soundness is equivalent to a restricted form of ω-consistency, dealing only with elementary statements about computer programs.

[6] P_0 would take that student about a month to write.

2. P_0 constructs a formal statement φ in the system F that asserts the following:

   "P does not halt when run on input P."

3. P_0 cycles through all possible text strings looking for a proof of φ in the system F (using B as a subroutine).

4. If P_0 ever finds a proof of φ in F, then P_0 outputs "P runs forever" and halts. Otherwise, P_0 will never find such a proof, and so it will run forever.

Now suppose we run P_0 on *itself*, i.e., with input P_0. What can happen? Either P_0 halts on input P_0 or it doesn't. If P_0 halts on P_0, then we know that F proves the statement,

"P_0 halts when run on input P_0,"

because F is assumed to be computationally complete. But P_0 halting on input P_0 means that it discovered a proof in F of the statement,

"P_0 does not halt when run on input P_0."

Thus F proves two contradictory statements, and hence F is inconsistent, let alone computationally unsound.

So since we assume that F is computationally sound, and hence consistent, it must be that P_0 does not halt on input P_0. But this means that P_0 never finds the proof it is looking for, namely, a proof in F of

"P_0 does not halt when run on input P_0."

Since P_0 checks all possible proofs exhaustively, it must be that no such proof exists. Let G be this last statement above (G is known as the *Gödel sentence*). We must conclude that G is true but unprovable in F. Note that this conclusion follows from the consistency of F alone; the computational soundness assumption is not needed here.

Now since F is in addition computationally sound, there likewise can be no proof in F of the statement,

"P_0 halts when run on input P_0,"

which is the negation of G. Thus G can neither be proven nor refuted in the system F. □

5 The Second Incompleteness Theorem

The argument above might appear paradoxical if taken too informally: we have just proved that G is true but not provable. But didn't we just *prove* that G is true, so G is provable after all? The way out of this paradox leads to Gödel's second Incompleteness theorem.

We actually didn't prove that G is true unconditionally. What we proved, at least in the first part, was the statement,

"If F is consistent, then G is true."

Call this statment H_F. This H_F can be expressed formally in F (provided F is computationally complete and computationally formalizable), as we will now see. First we express the statement

"F is consistent"

formally in F. This statement is denoted "$\texttt{Con}(F)$." Consider a computer program `FindContradiction` that halts if and only if F is inconsistent: `FindContradiction` searches exhaustively for a proof in F of some contradiction, i.e., a statement and its negation. If `FindContradiction` ever finds such a proof, it halts; otherwise, it runs forever. Such a program exists because of the computational formalizability of F. But then $\texttt{Con}(F)$ is just equivalent to the statement,

"`FindContradiction` does not halt,"

which is formally expressible in F by computational completeness. So we can just *define* $\texttt{Con}(F)$ to be the formal rendering in F of this statement, "`FindContradiction` does not halt." But now H_F is of the form, "$\texttt{Con}(F) \rightarrow G$," so H_F is formally expressible in F.

Now let's make another reasonable assumption about F:

Formal Reflection (I). H_F is provable in F.

That is, F is strong enough to capture the informal proof of H_F we gave in the last section. (PA, ZFC, and many other systems all share this additional property.) So in F there is a proof of H_F. But we already know that there is no proof of G in F, provided F is consistent. The only way to reconcile these two facts is for there to be no proof in F of $\texttt{Con}(F)$. For if there were a proof of $\texttt{Con}(F)$ in F, then combining this proof with the proof for H_F and then using modus ponens would immediately yield a proof of G in F, which we already know cannot exist if F is consistent. Thus we've shown the following.

Theorem 2 (Gödel) *If F is computationally complete, computationally formalizable, consistent, and formally reflective (I), then $\texttt{Con}(F)$ is not provable in F.*

Of course, if F is inconsistent, then $\texttt{Con}(F)$ *is* provable in F, since all statements are provable in F.

If F is also computationally sound, then $\texttt{Con}(F)$ is not refutable in F, either. To see this, consider our computer program `FindContradiction` above. Since F is computationally sound, it is also consistent, so `FindContradiction` will run forever. Thus by computational soundness, F does not prove the statement,

"`FindContradiction` halts,"

which is just the negation of $\texttt{Con}(F)$. So $\texttt{Con}(F)$ is formally undecidable.

6 Stronger Systems

We can neither prove nor refute G in F, assuming F is computationally sound. Can we prove G in a stronger system? Perhaps the problem with F is that it is too weak. Can we strengthen F to a new consistent system F' which can decide the statements that cannot be decided in F? We can always add G as a new axiom to F, for example. Then G is trivially true in the stronger system, which is still consistent. Perhaps more systematically, we can add $\texttt{Con}(F)$ as a new axiom to F, to get a new consistent system F' that proves G and perhaps many other statements undecidable in F. The problem is that F' will itself be subject to the Incompleteness theorem, so F' will have its own share of undecidable statements. We could then add $\texttt{Con}(F')$ as a new axiom, obtaining a new system F'', and so on. This process will never end up with a complete system, since whenever we stop, the system we've built is still susceptible to the Incompleteness theorem.

Incidentally, if we add the negation of G as a new axiom to F, we also get a consistent system. This new system is not computationally sound, however, because it proves (axiomatically) the statement,

"P_0 halts when run on input P_0,"

which is false.

7 Rosser's Extension

In Section 4 we constructed a statement G that is undecidable in the system F, but we had to assume the hypothesis that F is computationally sound, which (as we just saw) is strictly stronger than just assuming that F is consistent. Can we get an undecidable statement just on the assumption that F is consistent? Yes we can, via a trick due to Rosser.

We describe a computer program P_1 that is similar to P_0, but that differs from P_0 in a few crucial details. The new trick is to look simultaneously for a proof of a statement and a proof of the statement's negation. P_1 behaves as follows:

1. P_1 reads as input a program P from a text file (it reads from a file and checks that it constitutes a legitimate program).

2. P_1 constructs two formal statements in the system F:

   φ_1 = "P outputs 'yes' when run on input P,"

   and

   φ_2 = "P does not output 'yes' when run on input P."

   Note that φ_2 is just the negation of φ_1.

3. P_1 cycles through all possible text strings looking for either a proof of φ_1 or a proof of φ_2 in the system F (using B as a subroutine).

4. If P_1 finds a proof of φ_1 in F, then P_1 immediately outputs "no" and halts, without looking any further. If P_1 finds a proof of φ_2 in F, then P_1 immediately outputs "yes" and halts, without looking any further. Otherwise, P_1 will never find a proof of either statement, and so it will run forever without outputting anything.

Assume that F is consistent. As we did in Section 4, we now consider what can happen when P_1 runs on input P_1. Suppose first that P_1 outputs "yes." Then it must have found a proof in F of the statement,

"P_1 does not output 'yes' when run on input P_1."

But by the computational completeness of F, there is also a proof in F of this statement's negation, namely,

"P_1 outputs 'yes' when run on input P_1."

Thus F is inconsistent, contrary to our assumption.

So suppose then that P_1 outputs "no." Then P_1 found a proof in F of

"P_1 outputs 'yes' when run on input P_1."

But in this case P_1 halts without outputting "yes," and so by computational completeness, F also proves the statement,

"P_1 does not output 'yes' when run on input P_1,"

which just negates the statement above it. Thus F is inconsistent in this case as well.

So we must have that P_1 runs forever. This means that there is no proof in F of either of the previous two statements, since P_1 did not find one. That is, the statement,

"P_1 outputs 'yes' when run on input P_1"

is neither provable nor refutable in F. This statement is called the *Rosser sentence* and is denoted by R. We have shown

Theorem 3 (Rosser) *If formal system F is computationally complete, computationally formalizable, and consistent, then R is neither provable nor refutable in F.*

Note that we only used the assumption that F is consistent to show that R is undecidable; we did not need the further assumption that F is computationally sound.

8 A Fixed Point Theorem

All the results above and more are special cases of a simple yet powerful theorem about reasonable systems F, together with some assumptions about F being strong enough to capture certain forms of elementary reasoning. The theorem, known as the Fixed Point Theorem, asserts that for any F-expressible property φ, there is a statement in F which is essentially says, "I have property φ." We can, for instance, apply the Fixed Point Theorem to properties such as "is not provable" to immediately get the first Incompleteness theorem. We give some background first before stating and proving the Fixed Point Theorem.

Until now, we've been assuming that statements in F are simple assertions of fact, and have a single truth value, i.e., either true or false. More generally, a *formula* or *predicate* in F is a statement that may have one or more placeholders that can refer to objects that are supplied later. For example, the formula,

"x is a prime integer"

has a single placeholder x that potentially stands for any object. We sometimes call x a *free variable*. The formula above does not have a truth value as it stands; it only acquires a truth value when some object is substituted for x, and the truth value depends on the object. So,

"17 is a prime integer"

is a true statement, whereas

"8 is a prime integer"

and

"Arnold Schwarzenegger is a prime integer"

are false statements. A formula with no free variables is called a *closed formula* or *sentence*, so the above three statements are sentences. Sentences have definite truth values; formulas with free variables only have "potential" truth values. We say that an object *satisfies* a formula if substituting the object for the free variable x makes the formula true. Thus 17 satisfies the formula, "x is a prime integer," but 8 and Arnold Schwarzenegger do not.

A formula may contain more than one variable, or the same variable in more than one place. The same object must be substituted for each occurrence of the same variable. For example, consider the two formulas,

"x is a program that halts when run on input x"

and

"x is a program that halts when run on input y."

If we substitute the program `FindProof` for x in the first formula, we must make the same substitution at both occurrences of x, yielding the statement,

"`FindProof` is a program that halts when run on input `FindProof`."

This statement is false, since `FindProof` only halts when given a provable *statement*, but `FindProof` is a program, not a statement.[7] We could rephrase the first formula as,

"x is a program that halts when run on itself as input."

We can make different substitutions for x and for y in the second formula, however:

"`FindProof` is a program that halts when run on input '$0 = 0$'."

This is a true statement.

As we just saw, formulas of F can also be treated as objects, since they are just strings of text, and F can say things about them. For example, the formula,

"x is a (syntactically correct) formula"

is formally expressible in F. This fact follows from computational completeness: Let W_f be a program that reads a text string as input, checks its syntax, and outputs "yes" if the string conforms to the syntax of a formula of F, and "no" otherwise. Then the formula above can be rendered into the equivalent form,

"W_f outputs 'yes' on input x,"

which is more clearly seen to be expressible in F because F is computationally complete.

Here's one final example. By computational completeness, the formula,

"x is a formula with no free variables"

is expressible in F, because we can imagine a program that first checks that its input is a formula, then checks that it has no free variables. The formula becomes a true statement when "$1 + 1 = 3$" is substituted for x, and it becomes false when "$x + 1 = 3$" is substituted for x. If we write out a formula where another formula is substituted for a free variable, we keep our substituted formula in quotes to be clear that it is an object and not part of the larger formula. For example,

"'$x + 1 = 3$' is a formula with no free variables"

is how we would write the false statement above. This use of quotes is significant because it can make subtle distinctions. For example,

"x is variable name"

is a formula with a free variable x, whose truth value depends on what object is substituted for x. Whereas,

[7] We assume that the allowed syntax for computer programs and the allowed syntax for statements of F are mutually exclusive, so that a program can never be mistaken for a formula or vice versa.

"'x' is variable name"

is a true sentence that refers to the string object "x," which is indeed a variable name.

We are now ready to state the Fixed Point Theorem.

Theorem 4 (Fixed Point Theorem) *Suppose that F is computationally complete and computationally formalizable. Let φ be any formula of F with only one free variable x (which nevertheless may occur any number of times in φ). There exists a sentence G_φ of F such that F proves the statement,*

"G_φ is true if and only if 'G_φ' satisfies φ."

We can restate this more succinctly. Given φ as above, for any object o, we'll let "$\varphi(o)$" denote the sentence obtained by substituting o for the variable x in φ. Theorem 4 says that there is a sentence G_φ that is provably (in F) equivalent to $\varphi(\text{"}G_\varphi\text{"})$. In other words, there is a sentence G_φ that "says," "φ holds for me," or, "I have property φ," or "I satisfy φ," or any number of equivalent rephrasings, and this fact is provable in F. We say that G_φ is a *fixed point* of the formula φ.

Proof. Let S be a program that does the following:

1. S reads a string of text σ as input.
2. S checks whether σ is a formula of F with at most one free variable x, and if not, S halts with no output.
3. Otherwise, S finds every occurrence of the free variable x and replaces it with the string "σ" (including the quotes), producing the sentence $\sigma(\text{"}\sigma\text{"})$, which it outputs and then halts.

The program S exists by computational formalizability. Suppose φ is the formula given in the statement of the Theorem. Let ξ_φ be the formula,

"S outputs something when run on input x, and this output satisfies φ."

By computational completeness, ξ_φ is formally expressible in the system F. It is a formula with one free variable x. Now we define G_φ to be the output of S on input ξ_φ. Letting "$\iff$" denote "if and only if," the following equivalences are provable in F:

$$G_\varphi \iff \xi_\varphi(\text{"}\xi_\varphi\text{"}) \iff \varphi(\text{"}\xi_\varphi(\text{'}\xi_\varphi\text{'})\text{"}) \iff \varphi(\text{"}G_\varphi\text{"}).$$

The first and third equivalences hold because G_φ and $\xi_\varphi(\text{"}\xi_\varphi\text{"})$ *are the same formula*, and thus the left- and right-hand sides are the same string, making the equivalence a logical tautology in each case. The second equivalence comes from substituting "ξ_φ" for x in the formula ξ_φ, and noting that in this case F proves that S outputs "$\xi_\varphi(\text{'}\xi_\varphi\text{'})$" and nothing else. □

How are Gödel's and Rosser's theorems special cases of Theorem 4? Let `FindProof` be the program that

1. reads as input a formal statement φ in F (given as a text string), then
2. looks exhaustively for a proof in F of φ, halts if it ever finds one, or else runs forever.

Then let π be the formula,

> "`FindProof` does not halt on input x."

Then G_π, which says in effect, "I am not provable," is essentially the Gödel sentence G of Section 4, and the same reasoning we used there also applies here. Likewise, let `Rosser` be the program that

1. reads as input a formal statement φ in F (given as a text string), then
2. looks exhaustively for a proof in F of either φ or its negation, outputting "yes" if it ever finds a proof of φ, or "no" if it ever finds a proof of φ's negation, or else runs forever, not outputting anything.

Then let ρ be the formula,

> "`Rosser` does not output 'yes' on input x."

Then G_ρ is essentially the Rosser sentence R given in Section 7, which says, "A proof of my negation will be found before any proof of me will be found."

We can show more. There was nothing really mysterious or tricky about how we constructed G_φ from φ. After a little reflection, it becomes clear that this process can be automated. That is, there is a "fixed point-producing program" `FixedPoint` which outputs G_φ given input φ.

We can get the Second Incompleteness Theorem as a corollary to Theorem 4 for a reasonable, formally reflective (I) system F. The argument goes like this: Suppose that F proves `Con`(F). We'll conclude that F is inconsistent. By formal reflection (I) and modus ponens, F also proves the Gödel sentence G_π above. So the `FindProof` program will eventually halt on input "G_π." By computational completeness, F must then prove

> "`FindProof` halts on input 'G_π'."

But by Theorem 4 and elementary logic, F also proves

> "If G_π, then 'G_π' is not provable,"

so again by modus ponens, F proves

> "'G_π' is not provable,"

which, stated more formally, is

> "`FindProof` does not halt on input 'G_π'."

So we see that F proves contradictory statements, and hence it is inconsistent.

8.1 Löb's Theorem

In the 1950s, Henkin asked about the sentence $G_{\neg\pi}$, which proudly proclaims, "I *am* provable."[8] Is this statement provable? There is no easy heuristic to decide this: if $G_{\neg\pi}$ is provable, then it is true, so there is no violation of soundness. If $G_{\neg\pi}$ is not provable, then it is false. No problem there, either.

By a subtle and nonintuitive proof, Löb showed that the self-esteem of $G_{\neg\pi}$ is justified; it is indeed provable.

Theorem 5 (Löb) *If φ is any statement such that F proves*

> *"If φ is provable, then φ is true,"*

then F proves φ.

The statement,

> "If φ is provable, then φ is true"

is as close as one can get to formally stating in F that F is sound—at least with respect to φ. So one implication of Löb's theorem is that F cannot prove such a soundness assertion, except in the trivial case where F already proves φ.

The proof uses the Fixed Point theorem together with some additional reasonable properties of F and the program `FindProof`. Let Prov be the formula, "x is provable (in F)," or more precisely, "`FindProof` halts on input x." The basis of all our assumed properties is that the `FindProof` program is written in a reasonably transparent way, and that F can thus prove some basic facts about the behavior of `FindProof` (which, incidentally, are true by virtue of the rules for proving theorems in F). The first property follows immediately from computational completeness alone and does not depend on how `FindProof` is written:

Adequacy. For any sentence φ, if F proves φ, then F also proves $\text{Prov}(\varphi)$.

This is because if F proves φ, then `FindProof` halts on input φ, and hence F proves this fact, i.e., $\text{Prov}(\varphi)$.

The other two properties do not follow from computational completeness alone, but they are still reasonable, and they hold for all the formal systems mentioned before, provided `FindProof` is written in a perspicuous fashion. The next one says that F is strong enough to formalize the Adequacy property above, and to capture the reasoning used to show it.

Formal Adequacy. For any sentence φ, the sentence,

> "$\text{Prov}(\varphi) \rightarrow \text{Prov}(\text{Prov}(\varphi))$"

is provable in F.

In other words, F proves,

[8] $\neg\pi$ is the negation of π, i.e., the formula, "`FindProof` halts on input x."

"If φ is provable, then it is provable that φ is provable."

The final assumption we make is that F can capture and prove the fact that new theorems can be derived from old through modus ponens. Recall that modus ponens (m.p.) is the rule of inference by which a sentence ψ is proved from some pair of already-proven statements φ and $\varphi \to \psi$.

Formal Modus Ponens (FMP). For any sentences φ and ψ, the sentence,

"$\mathrm{Prov}(\varphi \to \psi) \to (\mathrm{Prov}(\varphi) \to \mathrm{Prov}(\psi))$"

is provable in F.

In other words, F proves,

"If $\varphi \to \psi$ is provable, then it is the case that if φ is provable then ψ is provable."

Proof of Theorem 5. We assume that F is computationally complete, computationally formalizable, formally adequate, and formally modus ponens. Suppose φ is as in the hypothesis of Theorem 5, i.e., F proves "$\mathrm{Prov}(\varphi) \to \varphi$." Let τ be the formula, "If x is provable, then φ is true," and let G_τ be its fixed point. That is, F proves

"$G_\tau \leftrightarrow (\mathrm{Prov}(G_\tau) \to \varphi)$."

So F now proves the following statements:

	Statement	**Reason**
(1)	$G_\tau \to (\mathrm{Prov}(G_\tau) \to \varphi)$	("$\to$" part of "$\leftrightarrow$")
(2)	$\mathrm{Prov}(G_\tau \to (\mathrm{Prov}(G_\tau) \to \varphi))$	((1), Adequacy)
(3)	$\mathrm{Prov}(G_\tau) \to \mathrm{Prov}(\mathrm{Prov}(G_\tau) \to \varphi)$	((2), FMP & m.p.)
(4)	$\mathrm{Prov}(\mathrm{Prov}(G_\tau) \to \varphi) \to (\mathrm{Prov}(\mathrm{Prov}(G_\tau)) \to \mathrm{Prov}(\varphi))$	(FMP)
(5)	$\mathrm{Prov}(G_\tau) \to (\mathrm{Prov}(\mathrm{Prov}(G_\tau)) \to \mathrm{Prov}(\varphi))$	((3,4), logic & m.p.)
(6)	$\mathrm{Prov}(G_\tau) \to \mathrm{Prov}(\mathrm{Prov}(G_\tau))$	(Formal Adequacy)
(7)	$\mathrm{Prov}(G_\tau) \to \mathrm{Prov}(\varphi)$	((5,6), logic & m.p.)
(8)	$\mathrm{Prov}(\varphi) \to \varphi$	(hypothesis)
(9)	$\mathrm{Prov}(G_\tau) \to \varphi$	((7,8), logic & m.p.)
(10)	$(\mathrm{Prov}(G_\tau) \to \varphi) \to G_\tau$	("$\leftarrow$" part of "$\leftrightarrow$")
(11)	G_τ	((9,10), m.p.)
(12)	$\mathrm{Prov}(G_\tau)$	((11), Adequacy)
(13)	φ	((9,12), m.p.)

□