CS 70 Discrete Mathematics and Probability Theory
Summer 2022 Jingjia Chen, Michael Psenka and Tarang Srivastava Note 12

Self-Reference and Computability

In this lecture we will explore the deep connection between proofs and computation. At the heart of this connection is the notion of self-reference, and it has far-reaching consequences for the limits of computation (the Halting Problem) and the foundations of logic in mathematics (Gödel's incompleteness theorem).

1 The Liar's Paradox

Recall that propositions are statements that are either true or false. We saw in an earlier lecture that some statements are not well defined or too imprecise to be called propositions. But here is a statement that is problematic for more subtle reasons:

"All Cretans are liars."

So said a Cretan in antiquity, thus giving rise to the so-called liar's paradox which has amused and confounded people over the centuries. Why? Because if the statement above is true, then the Cretan was lying, which implies the statement is false. But actually the above statement isn't really a paradox; it simply yields a contradiction if we assume it is true, but if it is false then there is no problem.

A true formulation of this paradox is the following statement:

"This statement is false."

Is the statement above true? If the statement is true, then what it asserts must be true; namely that it is false. But if it is false, then it must be true. So it really is a paradox, and we see that it arises because of the self-referential nature of the statement. Around a century ago, this paradox found itself at the center of foundational questions about mathematics and computation.

We will now study how this paradox relates to computation. Before doing so, let us consider another manifestation of the paradox, created by the great logician Bertrand Russell. In a village with just one barber, every man keeps himself clean-shaven. Some of the men shave themselves, while others go to the barber. The barber proclaims:

"I shave all and only those men who do not shave themselves."

It seems reasonable then to ask the question: Does the barber shave himself? Thinking more carefully about the question though, we see that, assuming that the barber's statement is true, we are presented with the same self-referential paradox: a logically impossible scenario. If the barber does not shave himself, then according to what he announced, he shaves himself. If the barber does shave himself, then according to his statement he does not shave himself!

2 Self-Replicating Programs

Can we use self-reference to design a program that outputs itself? To illustrate the idea, let us consider how we can do this if we could write the program in English. Consider the following instruction:

```
Print out the following sentence:
    "Print out the following sentence:"
```

If we execute the instruction above (interpreting it as a program), then we will get the following output:

> Print out the following sentence:

Clearly this is not the same as the original instruction above, which consists of two lines. We can try to modify the instruction as follows:

```
Print out the following sentence twice:
    "Print out the following sentence twice:"
```

Executing this modified instruction yields the output which now consists of two lines:

> Print out the following sentence twice:
>
> Print out the following sentence twice:

This almost works, except that we are missing the quotes in the second line. We can fix it by modifying the instruction as follows:

```
Print out the following sentence twice, the second time in quotes:
    "Print out the following sentence twice, the second time in quotes:"
```

Then we see that when we execute this instruction, we get exactly the same output as the instruction itself:

> Print out the following sentence twice, the second time in quotes:
>
> "Print out the following sentence twice, the second time in quotes:"

Quines and the Recursion Theorem

In the above section we have seen how to write a self-replicating program in English. But can we do this in a real programming language? In general, a program that prints itself is called a *quine*,[1] and it turns out we can always write quines in any programming language.

As another example, consider the following pseudocode:

```
(Quine "s")
    (s "s")
```

The pseudocode above defines a program `Quine` that takes a string `s` as input, and outputs `(s "s")`, which means we run the string `s` (now interpreted as a program) on itself. Now consider executing the program `Quine` with input `"Quine"`:

```
(Quine "Quine")
```

By definition, this will output

> (Quine "Quine")

which is the same as the instruction that we executed!

This is a simple example, but how do we construct quines in general? The answer is given by the *recursion theorem*. The recursion theorem states that given any program $P(x,y)$, we can always convert it to another

[1] Quine is named after the philosopher and logician Willard Van Orman Quine, as popularized in the book *"Gödel, Escher, Bach: An Eternal Golden Braid"* by Douglas Hofstadter.

program $Q(x)$ such that $Q(x) = P(x, Q)$, i.e., Q behaves exactly as P would if its second input is the description of the program Q. In this sense we can think of Q as a "self-aware" version of P, since Q essentially has access to its own description.

3 The Halting Problem

Are there tasks that a computer cannot perform? For example, we would like to ask the following basic question when compiling a program: does it go into an infinite loop? In 1936, Alan Turing showed that there is no program that can perform this test. The proof of this remarkable fact is very elegant and combines two ingredients: self-reference (as in the liar's paradox), and the fact that we cannot separate programs from data. In computers, a program is represented by a string of bits just as integers, characters, and other data are. The only difference is in how the string of bits is interpreted.

We will now examine the Halting Problem. Given the description of a program and its input, we would like to know if the program ever halts when it is executed on the given input. In other words, we would like to write a program `TestHalt` that behaves as follows:

$$\texttt{TestHalt(P,x)} = \begin{cases} \text{"yes"}, & \text{if program } P \text{ halts on input } x \\ \text{"no"}, & \text{if program } P \text{ loops on input } x \end{cases}$$

Why can't such a program exist? First, let us use the fact that a program is just a bit string, so it can be input as data. This means that it is perfectly valid to consider the behavior of `TestHalt(P,P)`, which will output "yes" if P halts on P, and "no" if P loops forever on P. We now prove that such a program cannot exist.

Theorem: *The Halting Problem is uncomputable; i.e., there does not exist a computer program TestHalt with the behavior specified above on all inputs* (P,x).

Proof: Assuming for contradiction the existence of the program `TestHalt`, use it to construct the following program:

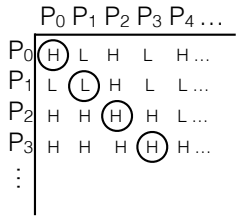
```
Turing(P)
  if TestHalt(P,P) = "yes" then loop forever
  else halt
```

So if the program P when given P as input halts, then `Turing(P)` loops forever; otherwise, `Turing(P)` halts. Assuming we have the program `TestHalt`, we can easily use it as a subroutine in the above program `Turing`.

Now let us look at the behavior of `Turing(Turing)`. There are two cases: either it halts, or it does not. If `Turing(Turing)` halts, then it must be the case that `TestHalt(Turing,Turing)` returned "no." But by definition of `TestHalt`, that would mean that `Turing(Turing)` should not have halted. In the second case, if `Turing(Turing)` does not halt, then it must be the case that `TestHalt(Turing, Turing)` returned "yes," which would mean that `Turing(Turing)` should have halted. In both cases, we arrive at a contradiction which must mean that our initial assumption, namely that the program `TestHalt` exists, was wrong. Thus, `TestHalt` cannot exist, so it is impossible for a program to check if any general program halts! □

What proof technique did we use? This was actually a proof by diagonalization, the same technique that we used in the previous lecture to show that the real numbers are uncountable. Why? Since the set of all

computer programs is countable (they are, after all, just finite-length strings over some alphabet, and the set of all finite-length strings is countable), we can enumerate all programs as follows (where P_i represents the i^{th} program):



The $(i, j)^{\text{th}}$ entry in the table above is H if program P_i halts on input P_j, and L (for "Loops") if it does not halt. Now if the program `Turing` exists it must occur somewhere on our list of programs, say as P_n. But this cannot be, since if the n^{th} entry in the diagonal is H, meaning that P_n halts on P_n, then by its definition `Turing` loops on P_n; and if the entry is L, then by definition `Turing` halts on P_n. Thus the behavior of `Turing` is different from that of P_n, and hence `Turing` does not appear on our list. Since the list contains all possible programs, we must conclude that the program `Turing` does not exist. And since `Turing` is constructed by a simple modification of `TestHalt`, we can conclude that `TestHalt` does not exist either. Hence the Halting Problem cannot be solved.

In fact, there are many more questions we would like to answer about programs but cannot. For example, we cannot know if a program ever outputs anything or if it ever executes a specific line. We also cannot check if two programs produce the same output. And we cannot check to see if a given program is a virus. These issues are explored in greater detail in the advanced course CS172 (Computability and Complexity).

The Easy Halting Problem

As noted above, the key idea in establishing the uncomputability of the Halting Problem is self-reference: Given a program P, we ran into trouble when deciding whether $P(P)$ halts. But in practice, how often do we want to execute a program with its own description as input? Is it possible that if we disallow this kind of self-reference, we can solve the Halting Problem?

For example, given a program P, what if we ask instead the question: "Does P halt on input 0?" This looks easier than the Halting Problem (hence the name Easy Halting Problem), since we only need to check whether P halts on a specific input 0, instead of an arbitrary given input (such as P itself). However, it turns out this seemingly easier problem is still uncomputable! We prove this claim by showing that if we could solve the Easy Halting Problem, then we could also solve the Halting Problem itself; since we know the Halting Problem is uncomputable, this implies the Easy Halting Problem must also be uncomputable.

Specifically, suppose we have a program `TestEasyHalt` that solves the Easy Halting Problem:

$$\texttt{TestEasyHalt(P)} = \begin{cases} \text{"yes"}, & \text{if program } P \text{ halts on input 0} \\ \text{"no"}, & \text{if program } P \text{ loops on input 0} \end{cases}$$

Then we can use `TestEasyHalt` as a subroutine in the following algorithm that solves the Halting Problem:

```
Halt(P,x)
  construct a program P' that, on input 0, returns P(x)
  return TestEasyHalt(P')
```

The algorithm `Halt` constructs another program P', which depends on both the original program P and the original input x, such that when we call $P'(0)$ we return $P(x)$. Such a program P' can be constructed very simply as follows:

```
P'(y)
   return P(x)
```

That is, the new program P' ignores its input y and always returns $P(x)$. (Note that the descriptions of P and of x are "hard-wired" into P'.) Then we see that $P'(0)$ halts if and only if $P(x)$ halts. Therefore, if we have such a program `TestEasyHalt`, then `Halt` will correctly solve the Halting Problem. Since we know there cannot be such a program `Halt`, we conclude `TestEasyHalt` does not exist either.

The technique that we use here is called a *reduction*. Here we are reducing one problem "Does P halt on x?" to another problem "Does P' halt on 0?", in the sense that if we know how to solve the second problem, then we can use that knowledge to construct an answer for the first problem. This implies that the second problem is actually as difficult as the first, despite the apparently simpler description of the second problem.

4 Godel's Incompleteness Theorem

In 1900, the great mathematician David Hilbert posed the following two questions about the foundation of logic in mathematics:

1. Is arithmetic consistent?
2. Is arithmetic complete?

To understand the questions above, we recall that mathematics is a formal system based on a list of axioms (for example, Peano's axioms of the natural numbers, axiom of choice, etc.) together with rules of inference. The axioms provide the initial list of true statements in our system, and we can apply the rules of inference to prove other true statements, which we can again use to prove other statements, and so on.

The first question above asks whether it is possible to prove both a proposition P and its negation $\neg P$. If this is the case, then we say that arithmetic is *inconsistent*; otherwise, we say arithmetic is *consistent*. If arithmetic is inconsistent, meaning there are false statements that can be proved, then the entire arithmetic system will collapse because from a false statement we can deduce anything, so every statement in our system will be vacuously true.

The second question above asks whether every true statement in arithmetic can be proved. If this is the case, then we say that arithmetic is *complete*. We note that given a statement, which is either true or false, it can be very difficult to prove which one it is. As a real-world example, consider the following statement, which is known as Fermat's Last Theorem:

$$(\forall n \geq 3)\ \neg(\exists x, y, z \in \mathbb{Z}^+)(x^n + y^n = z^n).$$

This theorem was first stated by Pierre de Fermat in 1637,[2] but it has eluded proofs for centuries until it was finally proved by Andrew Wiles in 1994.

In 1928, Hilbert formally posed the questions above as the Entscheidungsproblem. Most people believed that the answer would be "yes," since ideally arithmetic should be both consistent and complete. However, in 1930 Kurt Gödel proved that the answer is in fact "no": Any formal system that is sufficiently rich

[2] Along with the famous note: "I have discovered a truly marvelous proof of this, which this margin is too narrow to contain."

to formalize arithmetic is either inconsistent (there are false statements that can be proved) or incomplete (there are true statements that cannot be proved). Gödel proved his result by exploiting the deep connection between proofs and arithmetic. Actually Gödel's theorem also embodies a deep connection between proofs and computation, which was illuminated after Turing formalized the definition of computation in 1936 via the notion of Turing machines and computability.

In the rest of this note, we will first sketch the essence of Gödel's proof, and then we will outline an easier proof of the theorem using what we know about the Halting Problem.

Sketch of Gödel's Proof

Suppose we have a formal system F, which consists of a list of axioms and rules of inference, and assume F is sufficiently expressive that we can use it to express all of arithmetic.

Now suppose we can write the following statement:

$$S(F) = \text{“This statement is not provable in } F\text{.”}$$

Once we have this statement, there are two possibilities:

1. Case 1: $S(F)$ is provable. Then the statement $S(F)$ is true, but by inspecting the content of the statement itself, we see that this implies $S(F)$ should not be provable. Thus, F is inconsistent in this case.

2. Case 2: $S(F)$ is not provable. By construction, this means the statement $S(F)$ is true. Thus, F is incomplete in this case, since there is a true statement (namely, $S(F)$) that is not provable.

To complete the proof, it now suffices to construct such a statement $S(F)$. This is the difficult part of Gödel's proof, which requires a clever encoding (a so-called “Gödel numbering”) of symbols and propositions as natural numbers.

Proof via the Halting Problem

Let us now see how we can prove Gödel's result by reduction to the Halting Problem. Here we proceed by contradiction: Suppose arithmetic is both consistent and complete; we will use this assumption to solve the Halting Problem, which we have seen is impossible.

Recall that in the Halting Problem we want to decide whether a given program P halts on a given input x. For fixed P and x, let $S_{P,x}$ denote the proposition “P halts on input x.” The key observation is that this proposition can be phrased as a statement in arithmetic. The form of the statement $S_{P,x}$ will be

$$\exists z(z \text{ encodes a valid halting execution sequence of } P \text{ on input } x).$$

Although the details require some work, your programming intuition should hopefully convince you that such a statement can be written, in a fairly mechanical way, using only the language of standard arithmetic, with the usual operators, connectives and quantifiers: basically the statement just has to check, step by step, that the string z (encoded as a very long integer in binary) lists out the sequence of states that a computer would go through when running program P on input x.

Now let us assume, for contradiction, that arithmetic is both consistent and complete. This means that, for any (P,x), the statement $S_{P,x}$ is either true or false, and that there must exist a proof in arithmetic of either $S_{P,x}$

or its negation, $\neg S_{P,x}$ (and not both). But now recall that a proof is simply a finite binary string. Therefore, there are only countably many possible proofs, so we can enumerate them one by one and search for a proof of $S_{P,x}$ or $\neg S_{P,x}$. The following program performs this task:

```
Search(P,x)
  for every proof q:
    if q is a proof of S_{P,x} then output "yes"
    if q is a proof of ¬S_{P,x} then output "no"
```

The program Search takes as input the program P, and proceeds to check every possible proof until it finds either one that proves $S_{P,x}$, or one that proves $\neg S_{P,x}$. By assumption, we know that one of these proofs always exists, so the program Search will terminate in finite time, and it will correctly solve the Halting Problem. On the other hand, since we have already established that the Halting Problem is uncomputable, such a program Search cannot exist. Therefore, our initial assumption must be wrong, so it is not true that arithmetic is both consistent and complete.

Note that in the argument above we rely on the fact that, given a proof, we can construct a program that mechanistically checks whether it is a valid proof of a given proposition. This is a manifestation of the intimate connection between proofs and computation.