Self-Reference and Computability

The Liar's Paradox

Propositions are statements that are either true or false. We saw before 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. 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 stronger formulation of this paradox is the following statement: "This statement is false." Is the statement 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. Around a century ago, this paradox found itself at the center of foundational questions about mathematics and computation.

In this lecture, we will study how this paradox relates to computation. Before doing so, let us consider another manifestation of this 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 we are presented with 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!

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 algorithm 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:

$$\text{TestHalt}(P,I) = \begin{cases} \text{“yes”}, & \text{if program } P \text{ halts on input } I; \\ \text{“no”}, & \text{if program } P \text{ loops on input } I. \end{cases}$$

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 input P, and "no" if P loops forever on input P. We now prove that such a program cannot exist.

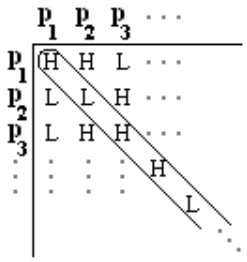
Proof: Define the program Turing, as follows:

Turing(P):
1. If TestHalt(P,P)="yes", then loop forever.
2. Otherwise, halt.

So if the program P halts when given P as input, then Turing(P) loops forever; otherwise, Turing(P) halts. Assuming we have the program TestHalt, we can easily use it as a subroutine in line 1 of the above program.

Now let us look at the behavior of Turing(Turing), i.e., look at the result of running the program Turing on the input 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 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. 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^{th} entry is H if program P_i halts on input P_j and L 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 TestHalt(P_n, P_n) claims to halt, then Turing loops. If the entry asserts that it loops, then Turing should have halted. Thus the existence of a TestHalt program contradicts the fact that we listed all possible programs P, and therefore the halting problem cannot be solved.

In fact, there are many more cases of questions we would like to answer about a program, but cannot. For example, we cannot know if a program ever outputs anything. We cannot know if it ever executes a specific line. We cannot even check to see if the program is a virus. These issues are explored in greater detail in the advanced course CS172.