The Unsolvability of the Halting Problem

The following result is probably the best-known undecidability result.

Theorem 8 (Unsolvability of the Halting Problem) *There is no program that meets the following specification. When given two inputs, a program x and a string y, print "Yes" if x, when run on y, terminates, and print "No" otherwise.*

Proof Assume HALTING is decidable. I.e. assume there is a program `halting` which takes two file names, `x` and `y`, as arguments and prints `Yes` if `x`, when run on `y`, halts, and prints `No` otherwise. Then SELFLOOPING can be solved by running `halting` with the two arguments being the same, and then reversing the answer, i.e., changing `Yes` to `No` and vice versa.

Gödel's Incompleteness Theorem

Once more digressing a bit from our focus on programs and their power, it is worth pointing out that our proof techniques easily let us prove what is certainly the most famous theorem in mathematical logic and one of the most famous in

— all programs have that property. The other is the property of not being a program — no program has that property. In terms of sets, these are the set of all programs and the empty set.

all of mathematics. The result was published in 1931 by Kurt Gödel and put an end to the notion that if only we could find the right axioms and proof rules, we could prove in principle all true conjectures. Put another way, it put an end to the notion that mathematics could somehow be perfectly "automated."

Theorem 9 (Gödel's Incompleteness Theorem, 1931)[10] *For any system of formal proofs that includes at least the usual axioms about natural numbers, there are theorems that are true but do not have a proof in that system.*

Proof If every true statement had a proof, SELFLOOPING would be decidable, by the following algorithm.

For any given program x the algorithm generates all strings in some systematic order until it finds either a proof that x has property SELFLOOPING or a proof that x does not have property SELFLOOPING.

It is crucial for the above argument that verification of formal proofs can be automated, i.e., done by a program. It is not the verification of formal proofs that is impossible. What is impossible is the design of an axiom system that is strong enough to provide proofs for all true conjectures.

Does this mean that once we choose a set of axioms and proof rules, our mathematical results will forever be limited to what can be proven from those axioms and rules? Not at all. The following lemma is an example.

Lemma 2 *For any system of formal proofs, the program in Figure 1.14 does have property* SELFLOOPING *but there is no formal proof of that fact within the system.*

Proof The program in Figure 1.14 is a modification of the program we used to prove SELFLOOPING undecidable. Now we prove SELFLOOPING "unprovable." When given its own source as input, the program searches systematically for a proof that it does loop and if it finds one it terminates. Thus the existence of such a proof leads to a contradiction. Since no such proof exists the program keeps looking forever and thus does in fact have property SELFLOOPING. □

Note that there is no contradiction between the fact that there is no proof of this program having property SELFLOOPING *within the given axiom system* and the fact that we just proved that the program has property SELFLOOPING. It is just that our (informal) proof cannot be formalized within the given system.

What if we enlarged the system, for example by adding one new axiom that says that this program has property SELFLOOPING? Then we could prove that fact (with a one-line proof no less). This is correct but note that the program had the axiom system "hard coded" in it. So if we change the system we get a new program and the lemma is again true — for the new program.

[10]Kurt Gödel, Über formal unentscheidbare Sätze der *Principia Mathematica* und verwandter Systeme I, *Monatshefte für Mathematik und Physik,* 38, 1931, 173–98.

```
void main( void )
{
              // systematically generates all proofs
      ...     // in the proof system under consideration
              // and stops if and when it finds a proof
              // that its input has property SELFLOOPING
}
```

Figure 1.14: Illustrating the proof of Lemma 2

Exercises

Ex. 1.1. The rational numbers are countable (as argued in class). Why does the diagonalization proof that shows that the real numbers are not countable not work for the rational numbers? **Diagonalization**

Ex. 1.2. Is it decidable if a program has a "memory leak?"

Ex. 1.3. Is it decidable if a program will try to "de-reference a null pointer?"

Ex. 1.4. Let SELFRECURSIVE be the property of programs that they make a recursive function call when running on their own source code as input. Describe a diagonalization proof that SELFRECURSIVE is not decidable.

Ex. 1.5. Somebody who has not taken this course cannot believe that there is no way to write a program that finds out, given a program p and input x, whether or not p when run on input x goes into an infinite loop. That person hands you a floppy with the source code of a program that they claim does in fact carry out that analysis.

Can you produce a program p and input x on which their program fails?

Ex. 1.6. Describe a transformation from SELFLOOPING to the property of not reading any input. **Transformations**

Ex. 1.7. The function `edit` that is used in the proof of Theorem 5 transforms SELFLOOPING not only to LOOPING but also to other properties. Give three such properties.

Ex. 1.8. Consider the following problem. Given a graph, is there a set of edges so that each node is an endpoint of exactly one of the edges in that set. (Such a set of edges is called a "complete matching".) Describe a transformation of this problem to the satisfiability problem for Boolean expressions. You may do this by showing (and explaining, if it's complicated) the expression for one of the above graphs.

Ex. 1.9. If G is a graph that you are trying to find a 3-coloring for, if $t(G)$ is the Boolean expression the graph gets translated into by our transformation of 3-COLOR into SAT, and, finally, if someone handed you an assignment of truth values to the variables of $t(G)$ that satisfied the expression, describe how you can translate that assignment of truth values back to a 3-coloring of the graph.

Ex. 1.10. Consider trying to transform 3-COLOR into SAT but using, in place of

$$(a_{\text{GREEN}} \wedge \overline{a_{\text{RED}}} \wedge \overline{a_{\text{BLUE}}}) \vee (\overline{a_{\text{GREEN}}} \wedge a_{\text{RED}} \wedge \overline{a_{\text{BLUE}}}) \vee (\overline{a_{\text{GREEN}}} \wedge \overline{a_{\text{RED}}} \wedge a_{\text{BLUE}})$$

the expression

$$x_{\text{GREEN}} \vee x_{\text{RED}} \vee x_{\text{BLUE}} .$$

(a) Would this still be a transformation of 3-COLOR into SAT? Explain why, or why not.

(b) Could we still translate a satisfying assignment of truth values back into a 3-coloring of the graph? Explain how, or why not.

Ex. 1.11. *(a)* Transform SELFLOOPING into the property of halting on those inputs that contain the word `Halt`, and looping on all others. *(b)* What does this prove?

Ex. 1.12. *(a)* Transform ODDLENGTH into SELFLOOPING. *(b)* What does this prove?

I/0-Properties

Ex. 1.13. Give three undecidable properties of programs that are not input-output properties and that were not mentioned in the notes. Give three others that are nontrivial I/O properties, hence undecidable, but were not mentioned in class.

Ex. 1.14. We needed to suppress the output in the `x_on_x_no_input_no_output` function used in the proof of Rice's Theorem. Which line of the proof could otherwise be wrong?

Ex. 1.15. Is SELFLOOPING an input-output property?

More Undecidability Results

Ex. 1.16. Formulate and prove a theorem about the impossibility of program optimization.

Ex. 1.17. "Autograders" are programs that check other programs for correctness. For example, an instructor in a programming course might give a programming assignment and announce that your programs will be graded by a program she wrote.

[10] If having a choice is not acceptable in a real "algorithm," we can always program the algorithm to prefer RED over GREEN over BLUE.

What does theory of computing have to say about this? Choose one of the three answers below and elaborate in no more than two or three short sentences.

1. "Autograding" is not possible.
2. "Autograding" is possible.
3. It depends on the programming assignment.

Ex. 1.18. Which of the following five properties of programs are decidable? Which are IO-properties?

1. Containing comments.
2. Containing only correct comments.
3. Being a correct parser for C (i.e., a program which outputs "Yes" if the input is a syntactically correct C program, and "No" otherwise).
4. Being a correct compiler for C (i.e., a program which, when the input is a C program, outputs an equivalent machine-language version of the input. "Equivalent" means having the same IO table.).
5. The output not depending on the input.