CS 3813: Introduction to Formal Languages and Automata

Undecidability, problem reduction, and Rice's Theorem (12.2)

Other undecidable problems

- Once we have shown that the halting problem is undecidable, we can show that a large class of other problems about the input/output behavior of programs are undecidable.
- In fact, we can show that any nontrivial property of the input/output behavior of programs is undecidable.

Examples of undecidable problems

- About Turing machines:
  - Is the language accepted by a TM empty, finite, regular, or context-free?
  - Does a TM meet its "specification," that is, does it have any "bugs."
- About context-free languages:
  - Are two context-free grammars equivalent?
  - Is a context-free grammar ambiguous?

Not so surprising

- Although this result is sweeping in scope, maybe it is not too surprising.
- If a simple question such as whether a program halts or not is undecidable, why should we expect that any other property of the input/output behavior of programs is decidable?
- Rice's theorem makes it clear that failure to decide halting implies failure to decide any other interesting question about the input/output behavior of programs.

Problem reduction

- Before we consider Rice's theorem, we need to understand the concept of problem reduction on which its proof is based.
- Reducing problem B to problem A means finding a way to convert problem B to problem A, so that a solution to problem A can be used to solve problem B.
- Why is this important? A reduction of problem B to problem A shows that problem A is at least as difficult to solve as problem B.

Using problem reduction to prove undecidability

- To show that a problem A is undecidable, we show that another problem B that we already know is undecidable can be reduced to A.
- Having proved that the halting problem is undecidable, we use problem reduction to show that other problems are undecidable.

Two examples

- *Totality problem*: Decide whether an arbitrary TM halts on all inputs. (If it does, it computes a "total function.") This is equivalent to the problem of whether a program can ever enter an infinite loop, for any input. It differs from the halting problem, which asks whether it enters an infinite loop for a particular input.
- *Equivalence problem*: Decide whether two TMs accept the same language. This is equivalent to the problem of whether two programs compute the same output for every input.

Proof that the totality problem is undecidable

- We prove that the halting problem is reducible to the totality problem. That is, if an algorithm can solve the totality problem, it can be used to solve the halting problem. Since no algorithm can solve the halting problem, the totality problem must also be unsolvable.
- The reduction is as follows. For any TM M and input w, we create another TM M' that takes an arbitrary input, ignores it, and runs M on w. Note that M' halts on all inputs if and only if M halts on input w. Therefore, an algorithm that tells us whether M' halts on all inputs also tells us whether M halts on input w, which would be a solution to the halting problem.

Proof that the equivalence problem is undecidable

- We prove that the totality problem is reducible to the equivalence problem. That is, if an algorithm can solve the equivalence problem, it can be used to solve the totality problem. Since no algorithm can solve the totality problem, the equivalence problem must also be unsolvable.
- The reduction is as follows. For any TM M, we can construct a TM M' that takes any input w, runs M on that input, and outputs "yes" if M halts on w. We can also construct a TM M'' that takes any input and simply outputs "yes." If an algorithm can tell us whether M' and M'' are equivalent, it can also tell us whether M' halts on all inputs, which would be a solution to the totality problem.

Practical implications

- The fact that the totality problem is undecidable means that we cannot write a program that can find any infinite loop in any program.
- The fact that the equivalence problem is undecidable means that the code optimization phase of a compiler may improve a program, but can never guarantee finding the optimally efficient version of the program. There may be potentially improved versions of the program that it cannot even be sure are equivalent.

Properties of programs

- We now describe a more general way of showing that a problem is undecidable, a result called Rice's theorem.
- First we introduce some definitions.
- A *property* of a program (TM) can be viewed as the set of programs that have that property.
- A *functional (or non-trivial) property* of a program (TM) is one that some programs have and some don't.

Rice's theorem

- "Any functional property of programs is undecidable."
- A functional property is:
  - a property of the input/output behavior of the program, that is, it describes the mathematical function the program computes
  - nontrivial, in the sense that it is a property of some programs but not all programs

Examples of functional properties

- The language accepted by a TM contains at least two strings.
- The language accepted by a TM is empty (contains no strings)
- The language accepted by a TM contains two different strings of the same length.

Rice's theorem continued

- The proof generalizes the reasoning involved in reducing the halting problem to other problems.
- Rice's theorem can be used to show that whether the language accepted by a Turing machine is context-free, regular, or even finite, are undecidable problems.

Not all properties of programs are functional

- Some properties of programs are decidable because they are not about the function the program computes, but instead, are about some details of the program itself
- Examples:
  - the program contains the transition ((q,0),(p,1))
  - starting on the empty tape, the program P reaches state q in at most five steps