CSE547T Class 14

Jeremy Buhler

March 8, 2017

1 Intuition for Reduction

We've proven exactly two languages undecidable (i.e. non-recursive). Can we extend this observation to more languages?

- Direct proofs are sometimes possible (following same scheme as SA)
- However, there is another way!
- Let A and B be two computational problems.
- If any algorithm for B can easily be modified to solve A, we would say colloquially that we can "reduce" solving A to solving B.
- *Example*: finding directions from the airport to the Arch with Google Maps can be reduced to a shortest path problem on a graph.
- Two interesting observations about reduction...
- If B is known to be easy, and A reduces to B, *and* reduction is easy, then A is easy.
- If A is known to be hard, and A reduces to B, *and* reduction is easy, then B must be hard.
- Note that in both cases, the reduction must be easy – otherwise, we could simply move all the complexity there.
- *Example*: dividing one number by another number is easy.
- To factor a number n, divide n by every number between 2 and $n-1$.
- When you find one that divides evenly, report the factors!
- (We don't believe that factoring is easy – we have to make a *lot* of calls to division.)
- Reduction is a very useful proof technique, both in complexity theory (NP-completeness, coming soon) and in computability.

2 Formalizing Reduction

- Let L_1 and L_2 be two languages.
- Suppose we have
  1. a TM M_2 that decides L_2.
  2. a mapping $f : \Sigma_1^* \to \Sigma_2^*$ such that $x \in L_1$ iff $f(x) \in L_2$.
- Then, to decide whether x is in L_1, we construct a TM M_1 that, given input x,
  1. computes $f(x)$,
  2. simulates operation of M_2 on $f(x)$ and returns the result that M_2 obtains.
- Observe that M_1 decides L_1.
- The above construction is called a *reduction* from L_1 to L_2.
- We say that L_1 *reduces to* L_2, also written $L_1 \leq L_2$. (Read "L_1 is *no harder than* L_2.")
- We ensure that f does not conceal uncomputable operations by making the TM M_1 compute it.
- Now suppose we know L_1 to be undecidable (non-recursive).
- Then there exists *no* TM M_1 deciding L_1. Contradiction!
- Conclude that M_2 cannot exist, and L_2 is not recursive. QED
- To summarize:

  If L_1 is not recursive, and $L_1 \leq L_2$, then L_2 is not recursive.

3 A Few Famous Examples

- Consider language

$$ACC = \{e(M)e(w) \mid M \text{ is a TM}, w \in L(M)\}$$

- **Want to prove**: ACC is not recursive.
- Suppose to the contrary that we have a TM M_{ACC} deciding ACC.
- We will use M_{ACC} to construct a machine M_{SA} that decides the language SA from last time:

$$SA = \{e(M) \mid M \text{ is a TM}, e(M) \in L(M)\}.$$

- On input $x \in \Sigma^*$, M_{SA} proceeds as follows.
- First, M_{SA} checks that x has form $e(M)$ for a TM M. If not, M_{SA} sets $f(x)$ to some invalid input for M_{ACC} (i.e. not an encoding of a TM and a string).

- Otherwise, M_{SA} constructs string $f(x) = e(M)e(x) = e(M)e(e(M))$.
- Finally, M_{SA} runs M_{ACC} on $f(x)$ and returns the result of M_{ACC}.
- **Claim**: $x \in$ SA iff $f(x) \in$ ACC.
- **Pf**: $x \in$ SA iff x encodes a TM M that accepts $e(M)$.
- But this is true iff the pair $(M, e(M))$ is in ACC.
- Since $f(x)$ encodes $(M, e(M))$, the claim is proven.
- Because M_{ACC} decides ACC, we conclude that M_{SA} decides SA, which is impossible because SA is not recursive.
- Hence ACC is not recursive either. QED
- (BTW, the complement of ACC is not RE, and neither is NOT-ACC, the complement of ACC under valid input encodings.)

Apparently, there's no algorithm to decide if an arbitrary TM accepts an arbitrary string. (Why not? TM might run forever!)

- Here's another rather natural question.
- Consider the language

$$EMPTY = \{e(M) \mid M \text{ is a TM}, L(M) = \emptyset\}$$

- *Claim*: EMPTY is not recursive.
- Suppose to the contrary that we have a TM $M_\emptyset$ deciding EMPTY.
- We will construct a TM M_{NACC} to decide NOT-ACC.
- On input $x \in \Sigma^*$, M_{NACC} proceeds as follows.
- First, M_{NACC} checks that x has form $e(M)e(w)$ for a TM M and string w. If not, M_{NACC} sets some invalid input for $M_\emptyset$ (i.e. not an encoding of a TM).
- Otherwise, M_{NACC} constructs $f(x) = e(M^*)$, where M^* is a TM that behaves as follows.
  1. On input y, M^* first compares y to w. If $y \neq w$, M^* rejects y.
  2. Otherwise, if $y = w$, then M^* simulates M on y and accepts iff M accepts.
- Finally, M_{NACC} runs $M_\emptyset$ on $f(x)$ and returns the result of $M_\emptyset$.
- **Claim**: $x \in$ NOT-ACC iff $f(x) \in$ EMPTY.
- **Pf**: $x \in$ NOT-ACC iff x encodes a TM M and string w such that M fails to accept w.
- Notice that M^* rejects every string except perhaps w.

- Hence, M^*'s language is empty iff M fails to accept w.
- Since $f(x)$ encodes M^*, the claim is proven.
- Conclude that M_{NACC} decides NOT-ACC, which is impossible because NOT-ACC is not recursive.
- Hence EMPTY is not recursive either. QED

Maybe simply testing for halting (acceptance *or* rejection) is easier than testing for acceptance?

- Consider the language

$$HALT = \{e(M)e(w) \mid M \text{ is a TM}, M \text{ halts on input } w\}.$$

- *Claim*: $HALT$ is not recursive.
- Suppose to the contrary that we have a TM M_{HALT} deciding HALT.
- We will construct a TM M_{ACC} to decide ACC.
- On input $x \in \Sigma^*$, M_{ACC} proceeds as follows.
- First, M_{ACC} checks that x has form $e(M)e(w)$ for a TM M and string w. If not, M_{ACC} sets $f(x)$ to an invalid input for M_{HALT} (i.e. not an encoding of a TM and a string).
- Otherwise, M_{ACC} constructs $f(x) = e(M^*)e(w)$, where M^* is a new TM built as follows.
  1. M^* is mostly identical to M, but it contains a new state p.
  2. However, whenever M would enter h_r, M^* enters p and runs forever, i.e. $\delta(p, c) = (p, c, S)$.
  3. Moreover, we mark leftmost cell of M's tape with an "end-of-tape" mark.
  4. If M would move left on a given symbol, and that symbol is marked "end-of-tape", M^* instead enters an infinite loop on state p.
  5. Similarly, if M would crash because no move is defined for some state q and cell content a, then M^* instead enters an infinite loop on state p.
- Finally, M_{ACC} runs M_{HALT} on $f(x)$ and returns the result of M_{HALT}.
- **Claim**: $x \in$ ACC iff $f(x) \in$ HALT.
- **Pf**: $x \in$ ACC iff x encodes a TM M and string w such that M accepts w.
- Now M may or may not reject some inputs, but M^* has been modified so that, if M would reject, then M^* runs forever.
- Hence, M^* halts on input w iff M accepts w.
- Since $f(x)$ encodes (M^*, w), the claim is proven.
- Conclude that M_{ACC} decides ACC, which is impossible because ACC is not recursive.
- Hence HALT is not recursive either. QED

The "Halting Problem" is apparently undecidable as well.