Undecidability

Notes on Automata and Theory of Computation

Chia-Ping Chen

Department of Computer Science and Engineering
National Sun Yat-Sen University
Kaohsiung, Taiwan ROC

Beyond RE Set

- The RE set appears to be very broad, including all languages that can be computed mechanically.
- Is there any language not recognizable by any TM?
- This is a question about the *limit* of computation.

Theorem

- The powerset of a (infinite) countable set S is uncountable.
- Suppose $S = \{s_1, s_2, \ldots\}$. An element L of 2^S can be represented by a sequence of 0 and 1 where the ith bit is 0 if $s_i \notin L$ and 1 otherwise.
- We can prove the theorem by contradiction. Suppose 2^S is countable, then we can write $L_1, L_2, \ldots$ for the elements in 2^S. Representing the L_i's as bit sequences, we can create a table T. Complementing each bit in the main diagonal of T, we get a bit sequence that is different from any of the sequences in T, contradicting that T includes all elements in 2^S.

Existence of Non-RE Languages

- For a finite alphabet Σ, Σ^* is countable. The set of all languages defined on Σ, 2^{Σ^*}, is *uncountable*.
- The set of TM's is countable as a TM can be encoded by a string in $\{0, 1\}$ (to be shown shortly).
- Each TM defines a RE language, so the set of RE languages is *countable*.
- There are more languages than there are regular languages. So there are (infinitely many) languages that are not RE.

Within RE Set

- A TM M may enter an infinite loop for an input not in its language. $L(M)$ is RE, but it is not good that we may not decide for some inputs whether they are in $L(M)$.
- For the above reason, we distinguish between those languages that can be accepted by a TM that always halts and those that cannot. This leads to the concept of **decidability**.
- Within the set of recursive languages, we further distinguish those halt in practical time of computation by a deterministic TM and those do not. This leads to the concept of **tractability**.

Encoding TM

- To ask (and answer) problems about TMs or RE languages, one should know it is possible to encode TMs by strings. One such encoding is described here.
- We first enumerate the states, tape symbols and directions, respectively.
- With 0 as separator, a transition $\delta(p, a) = (q, b, D)$ can be represented by 5 strings of 1's (unary representation for the enumerations of p, a, q, b, D).
- With 00 as separator, entries in the transition function can be concatenated.
- If an input w is specified, then w can be appended with 000 as separator.

Universal TM

- A universal TM M_u can simulate the computation of any M on any w.
- M_u models a general-purpose computer.
- The universal language L_u is defined by

$$L_u = L(M_u) = \{(w_i, w) : w \in L(M_i)\}.$$

- That is, L_u includes (w_i, w) if the TM M_i encoded by w_i accepts w.
- L_u plays a fundamental role in computation theory.

The Diagonalization Language

- As we have just shown, it is possible to encode a TM with a binary string.
- We can say that the ith TM M_i is the TM whose code is w_i, the ith binary string ($i = 1w_i$).
- $L(M_i) = \emptyset$ if w_i is not a valid code for TM.
- The diagonalization language L_d is defined by

$$\{w_i : w_i \notin L(M_i)\}.$$

L_d is the set of strings such that the TM whose code is w does not accept w.

Representing L_d

- Consider a (infinite) matrix where element i, j indicates whether M_i accepts w_j (1: accept, 0: not accept).
- Each row represents an RE language. For example, row i is called the characteristic vector for $L(M_i)$.
- The ith diagonal value indicates whether M_i accept w_i. The diagonal vector is a characteristic vector.
- L_d is represented by the complement of this diagonal vector.

L_d Is Not RE

- L_d is not recognized by any TM. It is not RE.
- To prove, suppose the contrary is true, so $L_d = L(M_i)$ for some i. Is $w_i \in L_d$?
  - If $w_i \in L_d$, then $w_i \in L(M_i)$. But then w_i is not in L_d by definition of L_d.
  - If $w_i \notin L_d$, then $w_i \notin L(M_i)$. But then w_i is in L_d by definition of L_d.

  Both lead to contradiction. So the assumption that M_i exists cannot be true.

Recursive

- A language L is said to be **recursive** if it is accepted by a TM, say M, that always halts. Such an M is also called an *algorithm*.
- Note that a RE language, say $L = L(M')$, can be non-recursive since M' may enter an infinite loop for some input $w \notin L(M')$.
- An interesting question is whether there exists any RE language that is not recursive.
- We have shown L_d to be non-RE. We will show that the complement of L_d to be RE but not recursive.

Theorems about Complements

- If L is recursive, so is its complement $\overline{L}$.
- If L and $\overline{L}$ are RE, then L is recursive.
- Only one of the four possibilities is true for L and $\overline{L}$.
  - L and $\overline{L}$ are both recursive.
  - Neither L nor $\overline{L}$ is RE.
  - L is RE but not recursive, $\overline{L}$ is not RE.
  - $\overline{L}$ is RE but not recursive, L is not RE.

$\overline{L_d}$ Is RE But Not Recursive

- From L_d being non-RE it follows that $\overline{L_d}$ is either non-RE or RE but not recursive.
- Note that $\overline{L_d}$ is the set of strings w_i such that M_i accepts w_i.
- We can use the universal TM to simulate running M_i on w_i for each i. To make sure that all such w_i's are enumerated, the simulation is carried out in a round-robin fashion.

Decidability

- Languages and problems are really the same thing. A problem becomes a language if we can represent instances of the problem by strings.
- A problem is said to be **decidable** if the language is recursive. It is said to be **undecidable** if the language is not recursive.
- Dividing problems or languages between decidable and undecidable is often more important then division between RE and non-RE. This is because a TM not guaranteed to halt does not solve a problem for all instances.

L_u Is Not Recursive

- We have shown that L_u is RE by constructing a TM, the universal TM, for it.
- Suppose L_u were recursive. Then $\overline{L_u}$ would be recursive as well. Let $\overline{L_u} = L(M)$. From M, we could construct a TM M' for L_d (and that's a contradiction).
  - Given w as input, M' copies w to be $w000w$. It then uses M to run on $w000w$. M' accepts w if M accepts $w000w$, and rejects otherwise.
  - Note w_i is accepted by M' iff w_i is not accepted by M_i. In other words $L(M') = L_d$.
- Since there cannot be TM for L_d, we conclude that L_u cannot be recursive.

Reduction

- In the previous proof, we use the method of reduction. The basic ideas are as follows.
  - We reduce a problem P_1 to another problem P_2: if we could solve P_2 then we could solve P_1.
  - But P_1 is known to be not solvable (in some sense), therefore P_2 cannot be solved (in the same sense).
    - If P_1 is not RE, then P_2 is not RE.
    - If P_1 is not recursive, then P_2 is not recursive.
- In the above proof, P_1 is L_d while P_2 is $\overline{L_u}$.

L_e and L_{ne}

- Every string is a TM. Define the languages

$$L_e = \{w_i \in \{0,1\}^* : L(M_i) = \emptyset\},$$
$$L_{ne} = \{w_i \in \{0,1\}^* : L(M_i) \neq \emptyset\}.$$

  L_e is the set of TMs (encodings) that accept the empty language. L_{ne} is its complement.

- We will show that L_{ne} is RE but not recursive, and L_e is not RE.

L_{ne} Is RE

- L_{ne} is RE as we can construct a TM M for it.
- M takes an input string w and interprets it as the code of a TM, say M_i.
- M simulates the running of M_i on strings in a proper order. If any string is accepted by M_i, then w is accepted by M.
- The code of any TM that accepts something will be recognized by M.

L_e Is Not RE

- L_{ne} is not recursive since we can reduce L_u to L_{ne}. That is, if we have an algorithm for L_{ne}, then we have an algorithm for L_u, which cannot be true!
- From a pair (M, w) we construct a TM M'. M' simulates the running of M on w. For any input v, M' accepts v if M accepts w, and rejects if M does not accept w.
- If we can decide whether $L(M')$ is empty, we can decide whether M accept w for any (M, w).
- It follows that L_e is not RE.

Property of RE Languages

- A property of the RE languages is a set of RE languages.
- Since an RE language is defined by at least one TM, a property is also equivalent to a set of TMs.
- If P is a property, then L_P is the set of codes for TMs M_i such that $L(M_i)$ is in P.
- For example, not-accepting any string is a property, say P, and $L_P = L_e$.
- A property P is said to be decidable if L_P is decidable.
- A property is trivial if it is the empty set or it is the set of all RE languages. Otherwise it is nontrivial.

Rice Theorem

- Every nontrivial property, say P, of the RE languages is undecidable. This is proved by reducing L_u to L_P.
- Suppose $\emptyset \notin L_P$. Suppose L is in L_P and $L = L(M_L)$.
- From a pair (M, w) we construct a TM M'. A part of M' simulates the running of M on w. Only if M accepts w does M_L run on input x.
  - If M does not accept w, then M_L never runs, and $L(M') = \emptyset$, so the code of $M' \notin L_P$.
  - If M accepts w, then M' accepts any string in L, $L(M') = L$, so the code of $M' \in L_P$.
- Being able to decide L_P would make L_u decidable.

Post Correspondence Problem

- Suppose we are given two lists, say A, B of strings over the same alphabet. A and B must be of the same length, say k. Let

$$A = w_1, w_2, \ldots, w_k, \quad B = x_1, x_2, \ldots, x_k.$$

- A instance of PCP (defined by A, B) has a solution $i_1, \ldots, i_m$ if

$$w_{i_1} \ldots w_{i_m} = x_{i_1} \ldots x_{i_m}.$$

- Is it possible to decide whether there exists a solution for any A, B?
- Unlike other problems we have been discussing, PCP appears to be very concrete and realistic.

PCP Is Undecidable

- PCP is a prime example of undecidable problem. That is, there is no algorithm to conclude whether a solution exists, for any given instance.
- We prove that by reducing L_u to PCP. We will do this through another problem called MPCP, the modified PCP.

MPCP

- Given two lists of strings A, B

  $$A = w_1, w_2, \ldots, w_k, \quad B = x_1, x_2, \ldots, x_k,$$

  MPCP asks whether there is a solution, which is a list of 0 or more integers $i_1, \ldots, i_m$ such that

  $$w_1 w_{i_1} \ldots w_{i_m} = x_1 x_{i_1} \ldots x_{i_m}.$$

- Note that (w_1, x_1) is required to start the strings. This is the main difference from PCP.

Reduce MPCP to PCP (1)

- From an MPCP instance, we can construct an instance of PCP such that a solution to the PCP instance implies a solution to the MPCP instance.
- From $A = \{w_{i=1}^k\}, B = \{x_{i=1}^k\}$, let

$$C = y_0, y_1, \ldots, y_k, y_{k+1},\ D = z_0, z_1, \ldots, z_k, z_{k+1},$$

  where y_i is based on w_i with a $*$ after each symbol of w_i and z_i is based on x_i with a $*$ before each symbol of x_i, and

$$y_0 = *y_1, z_0 = z_1$$

$$y_{k+1} = \$, z_{k+1} = *\$$$

Reduce MPCP to PCP (2)

- If there is a solution to the PCP instance with (C, D), then it must begin with pair (y_0, z_0) to match the first $*$ in z, and end with $k+1$ for a similar reason. That is,

$$*y_1 y_{i_1} \dots y_{i_m} * \$ = z_1 z_{i_1} \dots z_{i_m} * \$$$

- Stripping the $*$ and \$, $i_1 \dots i_m$ is a solution for MPCP instance with (A, B). That is

$$w_1 w_{i_1} \dots w_{i_m} = x_1 x_{i_1} \dots x_{i_m}$$

- So if we had an algorithm for PCP, we would have an algorithm for MPCP.

Reduce L_u to MPCP: Basic Idea

- Given a pair (M, w), we construct an instance of MPCP, say (A, B), such that M accepts w if and only if MPCP instance (A, B) has a solution.
- The basic idea is that (A, B) simulates the computation of M on w. A partial solution assumes the form

$$\#\alpha_1\#\alpha_3\#\alpha_3\# \ldots,$$

  where α_1 is the initial ID and $\alpha_i \vdash \alpha_{i+1}$.
- If a final state is entered, then A can catch up with B. Otherwise, B is always one ID ahead of A and no solution is possible.

Five Kinds of String Pairs

1. first pair: $(\#, \#q_0w\#)$
2. pairs for copy: $(\#, \#),\ (X, X)\ \forall X \in \Gamma$
3. pairs for transition function:

$$\delta(q, X) = (p, Y, R) : (qX, Yp)$$

$$\delta(q, X) = (p, Y, L) : (ZqX, pZY)\ \forall Z \in \Gamma$$

$$\delta(q, B) = (p, Y, R) : (q\#, Yp\#)$$

$$\delta(q, B) = (p, Y, L) : (Zq\#, pZY\#)\ \forall Z \in \Gamma$$

4. pairs for final state:
   $(XqY, q), (Xq, q), (qY, q)\ \forall q \in F,\ X, Y \in \Gamma$
5. final pair: $(q\#\#, \#)$

An Example

- Let $M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_3\})$, with

	$\delta(q_i, 0)$	$\delta(q_i, 1)$	$\delta(q_i, B)$
q_1	$(q_2, 1, R)$	$(q_2, 0, L)$	$(q_2, 1, L)$
q_2	$(q_3, 0, L)$	$(q_1, 0, R)$	$(q_2, 0, R)$

- Consider input 01. It is accepted by M as

$$q_1 01 \vdash 1 q_2 1 \vdash 10 q_1 \vdash 1 q_2 01 \vdash q_3 101$$

- Using the pairs we also have a MPCP solution with the following final common string

$$\# q_1 01 \# 1 q_2 1 \# 10 q_1 \# 1 q_2 01 \# q_3 101 \# q_3 01 \# q_3 1 \# q_3 \#\#$$

Proof

- A solution for the constructed instance of MPCP exists iff M accepts w.
- If M accepts w, then there is a sequence of IDs leading to a final state. The MPCP instance (A, B) has a solution as the string from A catches up with the string from B by construction.
- Suppose there is a solution for MPCP instance (A, B). It must start with $\#q_0w\#$. The next string from A is decided by the unmatched part of B string. Either a pair for copy is used, or a pair for transition is used if a state symbol is involved. This ensures that $\alpha_i \vdash \alpha_{i+1}$. Since A only catches up with B with a final state, w must be accepted by M.