COMS W3261
Computer Science Theory
Chapter 9 Notes

Alexander Roth

2014–11–02

Undecidability

A Language That Is Not Recursively Enumerable

Recall that a language L is *recursively enumerable* if $L = L(M)$ for some TM M.

Our long-range goal is to prove undecidable the language consisting of pairs (M, w) such that:

1. M is a Turing machine (suitably coded, in binary) with input alphabet $\{0, 1\}$,
2. w is a string of 0's and 1's, and
3. M accepts input w.

If this problem with inputs restricted to the binary alphabet is undecidable, then surely the more general problem, where TM's may have any alphabet, is undecidable.

We shall show that the language L_d, the "diagonalization language," which consists of all those strings w such that the TM represented by w does not accept the input w, has no Turing machine that accepts it. Shwoing there is no Turing machine at all for a language is showing that it has no algorithm or TM that will always halt.

Enumerating the Binary Strings

If w is a binary string, treat $1w$ as a binary integer i. Then we shall call w the ith string. That is, ϵ, is the first string, 0 is the second, 1 the third, 00 the fourth, 01 the fifth, and so on. Equivalent.y, strings are ordered by length and strings of equal length are ordered lexicographically. Hereafter, we shall refer to the ith string as w_i.

Codes for Turing Machines

To represent a TM $M(Q, \{0,1\}, \Gamma, \delta, q_1, B, F)$ as a binary string, we must first assign integers to the states, tape symbols, and directions L and R.

- We shall assume the states are $q_1, q_2, \ldots, q_r$ for some r. The start state will always be q_1, and q_2 will be the only accepting state.
- We shall assume the tape symbols are $X_1, X_2, \ldots, X_s$ for some s. X_1 always will be the symbol 0, X_2 will be 1, and X_3 will be B, the blank.
- We shall refer to direction L as D_1 and direction R as D_2.

Since each TM M can have integers assigned to its states and tape symbols in many different orders, there will be more than one encoding of the typical TM.

Suppose one transition rule is $\delta(q_i, X_j) = (q_k, X_l, D_m)$, for some integers i, j, k, l, and m. We shall code this rule by the string $0^i10^j10^k10^l10^m$. Notice that, since all of i, j, k, l, and m are at least one, there are no occurrences of two or more consecutive 1's within the code for a single transition.

A code for the entire TM M consists of all the codes for the transitions, in some order, separated by pairs of 1's:

$$C_1 11 C_2 11 \cdots C_{n-1} 11 C_n$$

where each of the C's is the code for one transition of M.

The Diagonalization Language

If w_i is not a valid TM code, we shall take M_i to be the TM with one state and no transitions. That is, for these values of i, M_i is a Turing machine that immediately halts on any input. Thus, $L(M_i)$ is $\emptyset$ if w_i fails to be a valid TM code.

That is,

- The language L_d, the *diagonalization language*, is the set of strings w_i such that w_i is not in $L(M_i)$.

That is, L_d consists of all strings w such that the TM M whose code is w does not accept when given w as input.

L_d can be represented in tabular form. The ith row can be thought of as the *characteristic vector* for the language $L(M_i)$; that is, the 1's in this row indicate the strings that are members of this language.

The diagonal values tell whether M_i accepts w_i. To construct L_d, we complement the diagonal.

The trick of complementing the diagonal to construct the characteristic vector of a language that cannot be the language that appears in any row is called *diagonalization*. It works because the complement of the diagonal is itself a characteristic vector describing membership in some language, namely L_d.

Proof that L_d is Not Recursively Enumerable

There is no Turing machine that accepts language L_d.

Theorem 1. L_d is not a recursively enumerable language. That is, there is no Turing machine that accepts L_d.

An Undecidable Problem That Is RE

Recursive Languages

WE call a language L *recursive* if $L = L(M)$ for some Turing machine M such that:

1. If w is in L, then M accepts (and therefore halts).
2. If w is not in L, then M eventually halts, although it never enters an accepting state.

A TM of this type corresponds to our notion of an "algorithm," a well-defined sequence of steps that always finishes and produces an answer. If we think of the language L as a "problem," then problem L is called *decidable* if it is a recursive language, and it is called *undecidable* if it is not a recursive language.

Dividing problems or languages between the decidable and those that are undecidable is often more important than the division between the recursively enumerable languages and the non-recursively-enumberal languages. The relationship among the three classes of languages is:

1. The recursive languages.
2. The languages that are recursively enumerable but not recursive.
3. The non-recursively-enumberable (*non-RE*) languages.

Complements of Recursive and RE languages

If a language L is RE, but $\overline{L}$, the complement of L, is not RE, then we know L cannot be recursive. For if L were recursive, then $\overline{L}$ would also be recursive and thus surely RE.

Theorem 2. If L is a recursive language, so is $\overline{L}$.

Theorem 3. If both a language L and its complement are RE, then L is recursive. Note that then by Theorem 2, $\overline{L}$ is recursive as well.

Of the nine possible ways to place a language L and its complement $\overline{L}$, only the following four are possible:

1. Both L and $\overline{L}$ are recursive.
2. Neither L nor $\overline{L}$ is RE.

3. L is RE but no recursive, and $\overline{L}$ is not RE.

4. $\overline{L}$ is RE but not recursive, and L is not RE.

The Universal Language

We define L_u, the *universal language*, to be the set of binary strings that encode a pair (M, w), where M is a TM with the binary input alphabet, and w is a string in $(\mathbf{0} + \mathbf{1})^*$, such that w is in $L(M)$. That is, L_u is the set of strings representing a TM and an input accepted by that TM. There is a TM U, often called the *universal Turing machine*, such that $L_u = L(U)$. Since the input to U is a binary string, U is in fact some M_j in the list of binary-input Turing machines.

U is a multitape Turing machine. In the case of U, the transitions of M are stored initially on the first tape, along with the string w. A second tape is used to hold the simulated tape of M, using the same format as for the code of M. That is, tape symbols X_i of M will be represented by 0^i, and tape symbols will be separated by single 1's. The third tape of U holds the state of M, with state q_i represented by i 0's.

The operation of U can be summarized as follows:

1. Examine the input to make sure that the code for M is a legitimate code for some TM. If not, U halts without accepting.

2. Initialize the second tape to contain the input w, in its encoded form. That is, for each 0 of w, place 10 on the second tape, and for each 1 of w, place 100 there. Note that the blanks on the simulated tape of M, which are represented by 1000, will not actually appear on that tape; all cells beyond those used for w will hold the blank of U.

3. Place 0, the start state of M, on the third tape, and move the head of U's second tape to the first simulated cell.

4. To simulate a move of M, U searches on its first tape for a transition $0^i10^j10^k10^l10^m$, such that 0^i is the state on tape 3, and O^j is the tape symbol of M that begins at the position on tape 2 scanned by U.

5. If M has no transition that matches the simulated state and tape symbol, no transition is found. Thus, M halts in the simulated configuration, and U must do likewise.

6. If M enters its accepting state, then U accepts.

In this manner, U simulates M on w. U accepts the coded pair (M, w) if and only if M accepts w.

Undecidability of the Universal Language

Theorem 4. L_u is RE but not recursive.

Knowing that L_u is undecidable (i.e., not a recursive language) is in many ways more value that knowing that L_d is not RE. The reduction of L_u to another problem P can be used to show there is no algorithm to solve P, regardless of whether or not P is not RE. However, reduction of L_d to P is only possible if P is not RE, so L_d cannot be used to show undecidability for those problems that are RE but not recursive. If we want to show a problem not to be RE, then only L_d can be used; L_u is useless since it *is* RE.

THe Halting Problem

We could define $H(M)$ for TM M to be the set of inputs w such that M halts given input w, regardless of whether or not M accepts w. Then, the *halting problem* is the set of pairs (M, w) such that w is in $H(M)$. This problem/language is another example of one that is RE but not recursive.

Post's Correspondence Problem

Definition of Post's Correspondence Problem

An instance of *Post's Correspondence Problem* (PCP) consists of two lists of strings over some alphabet Σ; the two lists must be of equal length: A and B, where $A = w_1, w_2, \ldots, w_k$ and $B = x_1, x_2, \ldots, x_k$, for some integer k. For each i, the pair (w_i, x_i) is said to be a *corresponding* pair.

The instance of PCP *has a solution*, if there is a sequence of one or more integers $i_1, i_2, \ldots, i_m$ that, when interpreted as indexes for strings in the A and B lists, yield the same string. That is, $w_{i_1}, w_{i_2}, \ldots, w_{i_m} = x_{i_1}, x_{i_2}, \ldots, x_{i_m}$. Thus, $i_1, i_2, \ldots, i_m$ is a *solution* to this instance of PCP.

Partial Solutions

The possible *partial soultions* of PCP instances are sequences of indexes $i_1, i_2, \ldots, i_r$ such that one of $w_{i_1}, w_{i_2}, \cdots, w_{i_r}$ and $x_{i_1}, x_{i_2}, \cdots, x_{i_r}$ is a prefix of the other, although the two strings are not equal. Notice that if a sequence of integers is a solution, then every prefix of that sequence must be a partial solution. Thus, understanding what the partial solutions are allows us to argue about what solutions there might be.

However, because PCP is undecidable, there is no algorithm to compute all the partial solutions. Thus, there can be an infinite number of them.

The "Modified" PCP

In the modified PCP, there is the additional requirement on a solution that the first pair on the A and B lists must be the first pair in the solution. An

instance of MCPCP is two lists $A = w_1, w_2, \ldots, w_k$ and $B = x_1, x_2, \ldots, x_k$, and a solution is a list of 0 or more integers $i_1, i_2, \ldots, i_m$ such that

$$w_1 w_{i_1} w_{i_2} \cdots w_{i_m} = x_1 x_{i_1} x_{i_2} \cdots x_{i_m}$$

Notice that the pair (w_1, x_1) is forced to be at the beginning of the two strings, even though the index 1 is not mentioned at the front of the list that is the solution.

AN important step in showing PCP is undecidable is reducing MPCP to PCP.

We are given an instance of MPCP with lists $A = w_1, w_2, \ldots, w_k$ and $B = x_1, x_2, \ldots, x_k$. We assume $*$ and \$ are symbols not present in the alphabet Σ of this MPCP instance. We construct a PCP instance $C = y_0, y_1, \ldots, y_{k+1}$ and $D = z_0, z_1, \ldots, z_{k+1}$ as follows:

1. For $i = 1, 2, \ldots, k$, let y_i be w_i with a $*$ after each symbol of w_i, and let z_i be x_i with a $*$ before each symbol of x_i.
2. $y_0 = *y_1$ and $z_0 = z_1$.
3. $y_{k+1} =$ \$ and $z_{k+1} = *$\$.

Theorem 5. MPCP reduces to PCP.

Completion of the Proof of PCP Undecidability

Given a pair (M, w), we construct an instance (A, B) of MPCP such that TM M accepts input w if and only if (A, B) has a solution.

The essential idea is that MPCP instance (A, B) simulates, in its partial solutions, the computation of M on input w. Partial solutions will consist of strings that are prefixes of the sequence of ID's of M: $\#\alpha_1\#\alpha_2\cdots$, where α_1 is the initial ID of M with input w, and $\alpha_i \vdash \alpha_{i+1}$ for all i. The string from B list will always be one ID ahead of the string from the A list, unless M enters an accepting state. In that case, there will be pairs to use that will allow the A list to "catch up" to the B list and eventually produce a solution.

We may assume our TM never prints a blank, and never moves left from its initial head position. In that case, an ID of the Turing machine will always be a string of the form $\alpha q \beta$, where α and β are strings of nonblank tape symbols, and q is a state. The symbols of α and β will correspond exactly to the contents of the cells that held the input, plus any cells to the right that the head has previously visited.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a TM described above and let w in Σ^* be an input string.

1. The first pair is $(\#, \#q_0 w\#)$. This pair, which must start any solution according to the rules of MPCP, begins the simulation of M on input w.

2. Tape symbols and the separator # can be appended to both lists. In effect, choice of these pairs lets us extend the A string to match the B string, and at the same time copy parts of the previous ID to the end of the B string.

3. To simulate a move of M, we have certain pairs that reflect those moves. These pairs help extend the B string to add the next ID, by extending the A string to match the B string. However, these pairs use the state to determine the change in the current ID that is needed to produce the next ID. These changes are reflected in the ID being constructed at the end of the B string.

4. If the ID at the end of the B string has an accepting state, then we need to allow the partial solution to become a complete solution. We do so by extending with "ID's" that are not really ID's of M, but represent what would happen if the accepting state were allowed to consume all the tape symbols to either side of it.

5. Finally, once the accepting state has consumed all tape symbols, it stands alone as the last ID on the B string. That is, the *remaineder* of the two strings is $q\#$.

Theorem 6. Post's Correspondence Problem is undecidable.

Other Undecidable Problems

The principal technique is reducing PCP to the problem we wish to prove undecidable.

Problems About Programs

We can write a program, in any conventional language, that takes as input an instance of PCP and searches for solutions some systematic manner, e.g., in order of the *length* of potential solutions. Since PCP allows arbitrary alphabets, we should encode the symbols of its alphabet in binary or some other fixed alphabet.

We can have our program do any particular thing we want when and if it finds a solution. Otherwise, the program will never perform that particular action. Thus, it is undecidable whether a program makes a nontrivial action. Any nontrivial property that involves what the program does must be undecidable.

Undecidability of Ambiguity for CFG's

Let the PCP instance consist of lists $A = w_1, w_2, \ldots, w_k$ and $B = x_1, x_2, \ldots, x_k$. For list A we shall construct a CFG with A as the only variable. The terminals are all the symbols of the alphabet Σ used for this PCP instance, plus a distinct set of *index symbols* $a_1, a_2, \ldots, a_k$ that represents the choices of pairs of strings

in a solution to the PCP instance. That is, the index symbol a_i represents the choice of w_i from the A list or x_i from the B list. The productions for the CFG for the A list are:

$$\begin{aligned} A \quad \rightarrow \quad & w_1 A a_1 \,|\, w_2 A a_2 \,|\, \cdots \,|\, w_k A a_k \,| \\ & w_1 a_1 \,|\, w_2 a_2 \,|\, \cdots \,|\, w_k a_k \end{aligned}$$

We shall call this grammar G_A and its language L_A, or *the language for the list* A.

Notice that the terminal strings derived by G_A are all those of the form $w_{i_1} w_{i_2} \cdots w_{i_m} a_{i_m} \cdots a_{i_2} a_{i_1}$ for some $m \geq 1$ and list of integers $i_1, i_2, \ldots, i_m$; each integer is in the range 1 to k. The sentential forms of G_A all have a single A between the strings and the index symbols, until we use one of the last group of k productions, none of which has an A in the body.

Observer also that any terminal string derivable from A in G_A has a unique derivation.

We can develop another grammar G_B from the list $B = x_1, x_2, \ldots, x_k$ as follows:

$$\begin{aligned} B \quad \rightarrow \quad & x_1 B a_1 \,|\, x_2 B a_2 \,|\, \cdots \,|\, x_k B a_k \,| \\ & x_1 a_1 \,|\, x_2 a_2 \,|\, \cdots \,|\, x_k a_k \end{aligned}$$

The language of this grammar will be referred to as L_B. The same observations of L_A hold on L_B.

Finally, we combine the languages and grammars of the two lists to form a grammar G_{AB} for the entire PCP instance. G_{AB} consists of:

1. Variable A, B, and S; the latter is the start symbol.
2. Productions $S \rightarrow A \,|\, B$.
3. All the productions of G_A.
4. All the productions of G_B.

G_{AB} is ambiguous if and only if the instance (A, B) of PCP has a solution.

The Complement of a List Language

Theorem 7. If L_A is the language of list A, then $\overline{L_A}$ is a context-free language.

Theorem 8. Let G_1 and G_2 be context-free grammars, and let R be a regular expression. Then the following are undecidable:

1. Is $L(G_1) \cap L(G_2) = \emptyset$?
2. Is $L(G_1) = L(G_2)$?
3. Is $L(G_1) = L(R)$?

4. Is $L(G_1) = T^*$ for some alphabet T?
5. Is $L(G_1) \subseteq L(G_2)$?
6. Is $L(R) \subseteq L(G_1)$?