CS20: Rice's Theorem and Post's Correspondence Problem

These notes describe a powerful theorem, known as Rice's theorem, for proving the undecidability of a large class of languages. They also introduce Post's correspondence problem, which leads to a very simple-looking yet provably undecidable language. For further reading on these topics, see Hopcroft and Ullman [1] and Lewis and Papadimitriou [2].

1 Classes of Undecidable Languages

Let **R** denote the class of *recursive* (or *decidable*) languages, and let **RE** denote the class of *recursively enumerable* (or *Turing-acceptable*) languages. Clearly $\mathbf{R} \subset \mathbf{RE}$, since all Turing-decidable languages are automatically Turing-acceptable. However, $\mathbf{R} \neq \mathbf{RE}$, since there are undecidable languages that are r.e. Another distinct class of languages is denoted **co-RE**, and defined to consist of those languages whose *complements* are r.e., whether or not they are themselves r.e. These three classes are related as shown in the figure below.

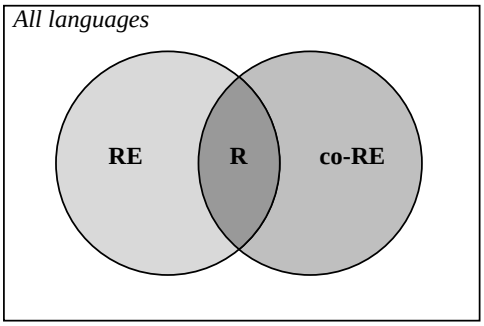

Figure 1: *All languages outside of* **R** *are undecidable. Those in* **RE** *are recursively enumerable, and the complements of those in* **co-RE** *are recursively enumerable. All three classes are distinct, and together they are a proper subset of all languages.*

An example of a language in $\mathbf{RE} - \mathbf{R}$ is L_{H}. An example of a language in $\mathbf{co\text{-}RE} - \mathbf{R}$ is $\overline{L_{\mathrm{H}}}$. Of course, examples of languages in **R** abound since every regular language and every context-free language is in this set. But is every language either in **RE** or **co-RE**? A straightforward cardinality argument shows that this is not so; in fact, almost all languages fail to have either property.

2 Rice's Theorem

In this section we introduce a theorem that demonstrates in one broad stroke that a very large class of languages is undecidable. In essence, it says that we can decide almost nothing about languages based purely on the Turing machines that accept them.

Theorem 1 (Informal Statement of Rice's Theorem) *Any language consisting of (encodings of) Turing machines whose languages possess some non-trivial property is undecidable.*

To make this theorem precise, we need to define what is meant by a "non-trivial property" of a language. We may give this an extremely general interpretation: a "non-trivial" property of a language will simply mean membership in some subset of r.e. languages, provided that the subset is neither empty nor the entire class **RE**. Note that we can restrict our attention to only r.e. languages, since the theorem is stated in terms of the languages defined by (i.e. accepted by) Turing machines; hence, every language the theorem applies to is already in **RE**.

Theorem 2 (Formal Statement of Rice's Theorem) *Let S be any non-trivial subset of the class* **RE** *of r.e. languages. That is, $S \subset \mathbf{RE}$ with $S \neq \emptyset$ and $S \neq \mathbf{RE}$. Then the language*

$$L_S = \{\langle M \rangle \mid L(M) \in S\}$$

is undecidable.

Proof: We will show that for any non-trivial subset S of **RE**, $L_\varepsilon \leq_T L_S$. That is, we will show that there exists an algorithm for computing a function f such that M halts on ε if and only if $L(f(M)) \in S$. First, let L_0 be any language in S. Now define the function f in such a way that $f(\langle M \rangle) = \langle M' \rangle$, where M' is a Turing machine with the property that

$$L(M') = \begin{cases} L_0 & \text{if } M \text{ halts on } \varepsilon \\ \emptyset & \text{if } M \text{ does not halt on } \varepsilon \end{cases} \tag{1}$$

The machine M' is easy to construct algorithmically from M and M_0, where M_0 is a Turing machine that accepts L_0. In particular, we construct M' so that it initially ignores its own input string w' and runs M on ε (in such a way that it does not disturb the string w'). If M eventually halts on ε, M' then runs M_0 on its own input string w' and accepts it if and only if M_0 accepts it. As a consequence of this construction, if M loops forever on ε, then the machine M' loops forever on *all* input strings and therefore $L(M') = \emptyset$. On the other hand, if M halts on ε, then M' accepts the input string w' if and only if it is in L_0, which is to say $L(M') = L_0$.

The function f *almost* has the desired properties; the only problem occurs when $\emptyset \in S$; that is, when S contains the empty language. When $\emptyset \in S$, the function f does not help us to distinguish between strings in L_ε and those outside L_ε, since both cases in equation (1) are then languages in S. To fix this problem observe that if $\emptyset \in S$, then the construction above shows that the complement of S (with respect to the universe **RE**) is undecidable, since it does not contain the empty language $\emptyset$, and the above proof works in that case. But a language in **RE** is decidable if and only if its complement with respect to **RE** is decidable. Therefore, L_S is undecidable whether or not S contains the empty language. □□

Note that Rice's theorem applies to properties of the *languages accepted by* Turing machines, not to the Turing machines themselves, or specifics of their operation. For example, the language

$$L_1 = \{\langle M \rangle \mid M \text{ has an even number of states}\}$$

is obviously decidable, since we need only decode and examine the machine's definition to determine how many states it has. But now consider the language

$$L_2 = \{\langle M \rangle \mid M \text{ never writes three consecutive blanks on its tape given input } \varepsilon\}.$$

The language L_2 is undecidable (in fact, it is not even r.e.), which can be easily demonstrated either by showing $\overline{L_\varepsilon} \leq_T L_2$, or $\overline{L_{\text{H}}} \leq_T L_2$. However, the undecidability of L_2 does not follow from Rice's theorem since it is not defined in terms of a property of the languages accepted by Turing machines.

3 Post's Correspondence Problem

Thus far, all of the undecidable languages that we have encountered have involved encodings of Turing machines, which makes them seem rather artificial. This section describes an unsolvable problem (which corresponds to an undecidable language) known as *Post's correspondence problem*, or PCP, that can be stated in terms of a simple "card game." In this "game" we are given a finite

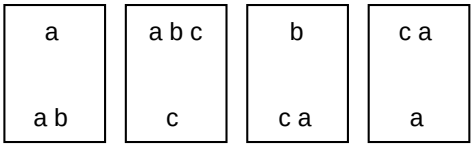

Figure 2: *This is an instance of PCP. Here the first card is to be the starting card.*

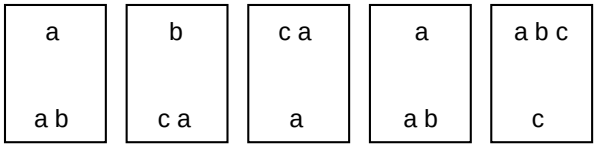

Figure 3: *This is a solution to the previous instance of PCP. Note that it starts with the designated starting card, and that one of the cards is repeated. In this example, all of the cards are used, although this needn't be the case in general.*

set of "cards," each with two non-empty strings of symbols on them; one across the top of the card and one across the bottom of the card. One of the cards is designated as the "starting" card. The problem is to determine whether there is a finite sequence of these cards, beginning with the starting card and using any number of repetitions of each card, such that the string formed by concatenating the symbols across the tops of all the cards exactly matches the string formed by concatenating the symbols across the bottoms of the cards. For example, consider the collection of cards shown in Figure 2, where the first card is designated as the "starting" card. In this case, there is a solution to the Post correspondence problem, as shown in Figure 3. In the solution, both the top and bottom rows read "abcaaabc", and this sequence begins with the designated starting card. Notice that one of the cards has been used twice; not all cards need appear in a solution.

More formally, we will define an instance of Post's correspondence problem to be a 3-tuple, (Σ, C, S), where Σ is an alphabet, C is a finite subset of $(\Sigma^* - \{\varepsilon\}) \times (\Sigma^* - \{\varepsilon\})$ representing the "cards", and $S \in C$ is the starting card. Since PCP is a decision problem, we may express it as a language recognition problem. Accordingly, we define L_{pcp} to be the language consisting of (encodings of) all PCP problem instances that have solutions. In other words, testing a string for membership in L_{pcp} is equivalent to determining whether the encoded PCP has a solution. The following theorem states a surprising fact about Post's correspondence problem.

Theorem 3 *The language L_{pcp} is undecidable.*

Proof: We will show that $L_a \leq_T L_{\text{pcp}}$, where $L_a = \{\langle M, w\rangle \mid M \text{ accepts } w\}$, which is undecidable. We exhibit an easily computed mapping f from (M, w) to an instance of PCP that is solvable if and only if $w \in L(M)$. The PCP instance is constructed in such a way that the only feasible sequences of cards are those that mimic the sequence of configurations that result when M is run on input w; the sequence terminates, resulting in a solution, if and only if M ultimately accepts w.

Figure 4 shows the nine different patters of cards that are defined for a specific Turing machine M and input w; we assume here that the tape alphabet Γ of M does not contain the symbol "#", and that $\Gamma \cap Q = \emptyset$. Card number 1 is the starting card, and its bottom string depicts the initial configuration of M with input w, flanked by the special character "#". A card with pattern 3 is added for each symbol a in Γ. Pattern 4 is created for each transition of the form (p, a, q, b, R) in the transition relation Δ of M. Pattern 5 is created for each transition $(p, a, q, b, L) \in \Delta$ and each symbol $c \in \Gamma$. Pattern 6 is created for each transition $(p, \$, q, a, R) \in \Delta$, where "$" is the blank

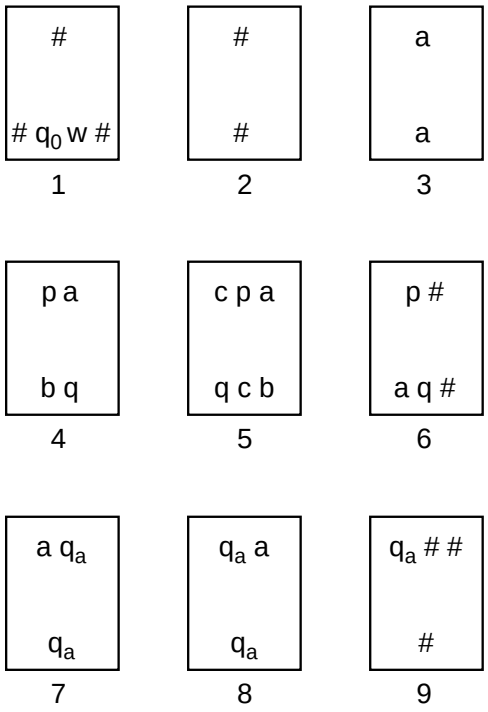

Figure 4: *Nine types of cards are required to mimic the operation of a given Turing machine on a given string w with an instance of PCP. Card 1 is the starting card, and every type of card except 1 and 2 have multiple versions corresponding to all the transitions, tape symbols, and accepting states of the Turing machine.*

symbol. Finally, patters 7, 8, and 9 are created for each accepting state $q_a \in A$ and each symbol $a \in \Gamma$.

To generate a solution for this PCP instance, it is necessary to generate the string appearing at the bottom of the starting card (minus the leading "#") across the tops of the subsequent cards. The cards are designed so that this will always be possible, and that such a sequence will automatically generate the *next* configuration of the Turing machine along the *bottom* row. By again adding a sequence of cards to match this new string along the top, we thereby generate the next configuration along the bottom, which now needs to be matched, and so on. Cards 3, 4, 5, and 6 are the key to this behavior. Card 3 allows us to copy the contents of the tape, but not the state symbol. Cards 4 and 5 allow us to copy the state symbol along the top, while also creating a string along the bottom that depicts the new configuration, after the head moves right or left, respectively. Card 6 allows more symbols to be added as the head moves beyond the last non-blank symbol currently on the tape.

This process continues until an accepting state is entered. Cards 7 and 8 then allow the top row to finally catch up by gradually "erasing" each symbol from an accepting configuration. Finally, card 9 finishes the sequence of cards by "erasing" the accepting state. This scenario is demonstrated in Figure 6, which shows the sequence of cards that mimic the sequence of configurations

$$\begin{aligned}(\varepsilon, q_0, \$a\$\$) &\vdash_{\text{M}} (\$, q_1, a\$\$) \\ &\vdash_{\text{M}} (\$b, q_2, \$\$)\end{aligned}$$

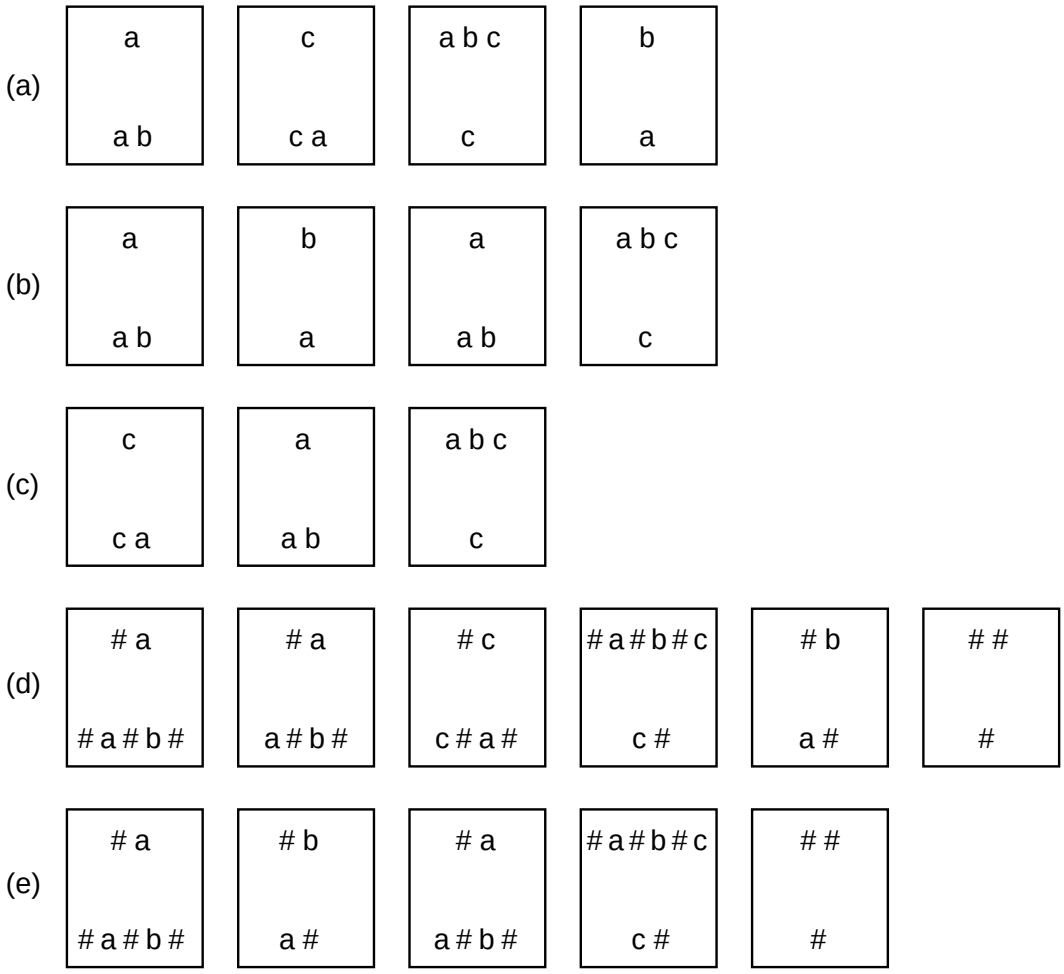

Figure 5: *Line (a) shows an instance of PCP, where the first card is to be the starting card. This instance has a solution, shown in line (b). If we allow any card to be the starting card, there is also another solution, shown in line (c). Line (d) shows the modified cards, with one new card, where "#" is a symbol not in the original alphabet. The card corresponding to the original starting card is now the* only *card that can begin a solution. Line (e) shows a solution with the modified cards.*

$$\vdash_{\mathrm{M}} \quad (\$b\$, q_3, \$),$$

where "$" is the blank symbol, q_0 is the start state, and q_3 is an accept state.

Post's correspondence problem is frequently stated in a slightly different form, where a solution can begin with any of the cards. We'll call this version the *Free* PCP, or FPCP, since we are free to start with any card. We shall denote the corresponding language by L_{fpcp}. While L_{fpcp} is also undecidable, this fact is not an immediate consequence of the undecidability of L_{pcp}; it requires a proof.

Theorem 4 *The language L_{fpcp}, in which a PCP solution is allowed to start with any of the cards, is undecidable.*

Proof: We shall show that $L_{\mathrm{pcp}} \leq_T L_{\mathrm{fpcp}}$, which implies that L_{fpcp} is undecidable. To do this, we

must show how to transform an instance of PCP into an instance of FPCP in such a way that the former has a solution if and only if the latter has a solution. One way to accomplish this is to alter the cards slightly so that any existing solution (beginning with the designated card) is preserved, yet it is provably impossible to form a solution beginning with any card except the one that is designated to be the starting card. We can do this with the construction shown in Figure 5, where "#" is assumed to be a letter not in the original alphabet. If the original problem instance has n cards, the modified problem instance has $n+2$ cards. The new instance includes two versions of the starting card; one that is easily seen to be the only candidate for beginning a solution, and one that can occur anywhere else in a solution. The new instance also includes the card $(\#\#, \#)$, which is the only candidate for ending a solution; however, this card cannot begin a solution, since none of the other cards can possibly follow it. □□

4 Subsets and Supersets of Undecidable Languages

In general, subsets of undecidable languages needn't be undecidable themselves, since every finite subset is decidable, and there may well be infinite subsets of "easy" instances that can be decided. For example, L_{pcp} has an infinite subset that is trivially decidable: the set of all instances consisting of a single card. These trivial instances have a solution if and only if the top and bottom strings on the single card match. As another example, the subset of L_{pcp} consisting of instances with $|\Sigma| = 1$ is also decidable.

Similarly, supersets of undecidable languages needn't be undecidable themselves. As a trivial example, observe that $L_{\text{H}} \subset \Sigma^*$, where Σ is the alphabet of L_{H}. Clearly Σ^* is decidable, since it is regular. However, there is an important special case in which supersets of undecidable languages *are* undecidable. Suppose that $L_1 = S \cap L_2$, where S is a decidable language. Then $L_1 \subset L_2$, and L_2 is undecidable if L_1 is. This follows from the fact that decidable languages are closed under intersection. We think of L_1 as being a *subproblem* of L_2; that is, L_1 consists of those elements in L_2 that also have some additional property that is easy to discern. For example, let L^*_{pcp} denote the language consisting of generalized PCP instances that have solutions. Here "generalized" means that the strings can contain "wild symbols" that match any symbol, by definition. We can represent such an instance using the same (Σ, C, S) convention by allowing the use of symbols that are not in Σ and declaring any such symbol to be a "wild symbol". Then

$$L_{\text{pcp}} = S \cap L^*_{\text{pcp}},$$

where S consists of all PCP instances in which *no* wild symbols appear; that is, all the symbols appearing in the strings are in Σ. Since S is easy to decide, and L_{pcp} is undecidable, it follows that the generalized problem L^*_{pcp} is also undecidable. The rule is this: If a language L is undecidable, then so is any superset of L provided that there is an algorithm to distinguish the new elements from the original elements of L.

References

[1] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[2] Harry R. Lewis and Christos H. Papadimitiou. *Elements of the Theory of Computation.* Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1998.

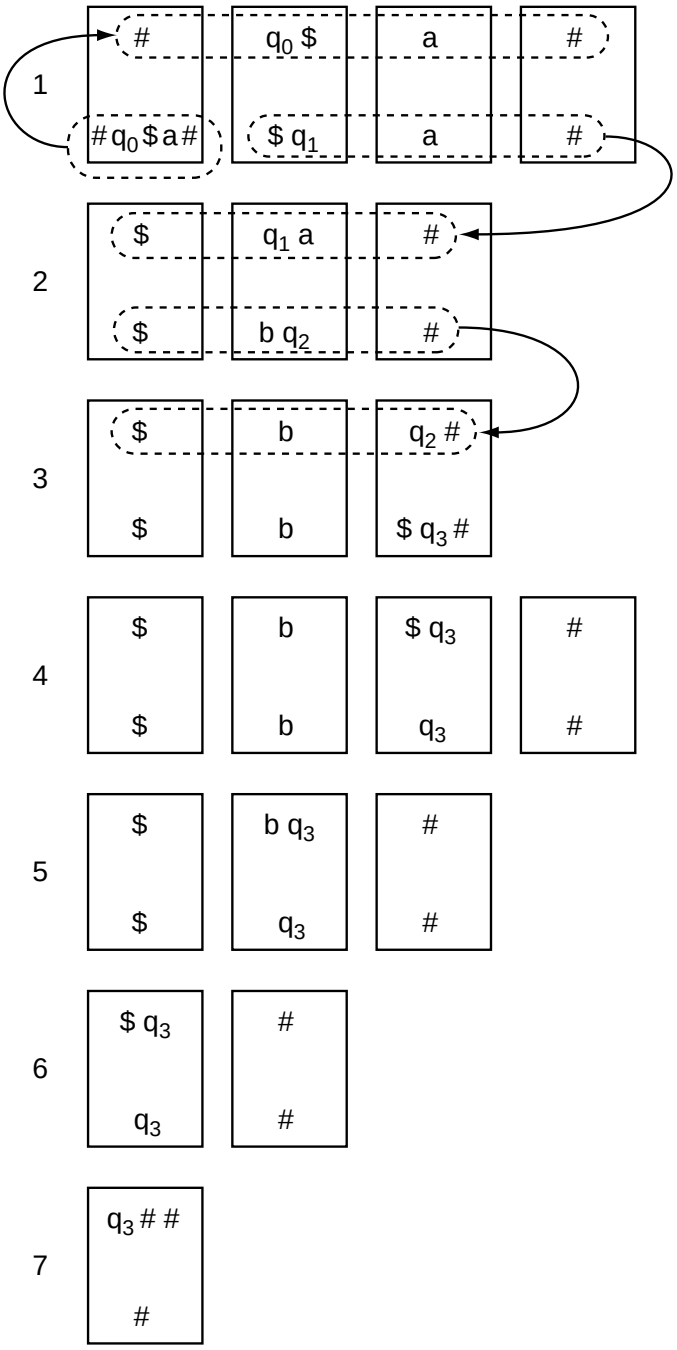

Figure 6: *In this example, the cards of a PCP instance simulate the operation of a Turing machine that overwrites the string "a" with the string "b" and then accepts. The seven rows of cards are to be placed in sequence from top to bottom; they are depicted in rows to emphasize the sequence of configurations. The pairs of dashed regions connected by arrows indicate strings that match. Following the starting card the only feasible strategy is to add cards that copy the hitherto unmatched portion of the bottom string to the top. Doing so creates a new (unmatched) string along the bottom that encodes the next configuration of the Turing machine. Rows 5, 6, and 7 show how the accepting configuration is matched and the sequence terminated.*