CHAPTER

8

UNDECIDABILITY

We now consider the classes of recursive and recursively enumerable languages. The most interesting aspect of this study concerns languages whose strings are interpreted as codings of instances of problems. Consider the problem of determining if an arbitrary Turing machine accepts the empty string. This problem may be formulated as a language problem by encoding TM's as strings of 0's and 1's. The set of all strings encoding TM's that accept the empty string is a language that is recursively enumerable but not recursive. From this we conclude that there can be no algorithm to decide which TM's accept the empty string and which do not.

In this chapter we shall show that many questions about TM's, as well as some questions about context-free languages and other formalisms, have no algorithms for their solution. In addition we introduce some fundamental concepts from the theory of recursive functions, including the hierarchy of problems induced by the consideration of Turing machines with "oracles."

8.1 PROBLEMS

Informally we use the word *problem* to refer to a question such as: "Is a given CFG ambiguous?" In the case of the *ambiguity problem*, above, an instance of the problem is a particular CFG. In general, an *instance* of a problem is a list of arguments, one argument for each parameter of the problem. By restricting our attention to problems with yes-no answers and encoding instances of the problem by strings over some finite alphabet, we can transform the question of whether there exists an algorithm for solving a problem to whether or not a particular language is recursive. While it may seem that we are throwing out a lot of impor-

tant problems by looking only at yes-no problems, in fact such is not the case. Many general problems have yes-no versions that are provably just as difficult as the general problem.

Consider the ambiguity problem for CFG's. Call the yes-no version AMB. A more general version of the problem, called FIND, requires producing a word with two or more parses if one exists and answering "no" otherwise. An algorithm for FIND can be used to solve AMB. If FIND produces a word w, then answer "yes"; if FIND answers "no," then answer "no." Conversely, given an algorithm for AMB we can produce an algorithm for FIND. The algorithm first applies AMB to the grammar G. If AMB answers "no" our algorithm answers "no." If AMB answers "yes," the algorithm systematically begins to generate all words over the terminal alphabet of G. As soon as a word w is generated, it is tested to see if it has two or more parse trees. Note that the algorithm does not begin generating words unless G is ambiguous, so some w eventually will be found and printed. Thus we indeed have an algorithm. The portion of the algorithm that tests w for two or more parses is left as an exercise.

The process whereby we construct an algorithm for one problem (such as FIND), using a supposed algorithm for another (AMB), is called a *reduction* (of FIND to AMB). In general, when we reduce problem A to problem B we are showing that B is at least as hard as A. Thus in this case, as in many others, the yes-no problem AMB is no easier than the more general version of the problem. Later we shall show that there is no algorithm for AMB. By the reduction of AMB to FIND we conclude there is no algorithm for FIND either, since the existence of an algorithm for FIND implies the existence of an algorithm for AMB, a contradiction.

One further instructive point concerns the coding of the grammar G. As all Turing machines have a fixed alphabet, we cannot treat the 4-tuple notation $G = (V, T, P, S)$ as the encoding of G without modification. We can encode 4-tuples as binary strings as follows. Let the metasymbols in 4-tuples, that is, the left and right parentheses, brackets, comma and $\rightarrow$, be encoded by 1, 10, 100, ..., 10^5, respectively. Let the ith grammar symbol (in any chosen order) be encoded by 10^{i+5}. In this encoding, we cannot tell the exact symbols used for either terminals or nonterminals. Of course renaming nonterminals does not affect the language generated, so their symbols are not important. Although we ordinarily view the identities of the terminals as important, for this problem the actual symbols used for the terminals is irrelevant, since renaming the terminals does not affect the ambiguity or unambiguity of a grammar.

Decidable and undecidable problems

A problem whose language is recursive is said to be *decidable*. Otherwise, the problem is *undecidable*. That is, a problem is undecidable if there is no algorithm that takes as input an instance of the problem and determines whether the answer to that instance is "yes" or "no."

An unintuitive consequence of the definition of "undecidable" is that problems with only a single instance are trivially decidable. Consider the following problem based on Fermat's conjecture. Is there no solution in positive integers to the equation $x^i + y^i = z^i$ if $i \geq 3$? Note that x, y, z, and i are not parameters but bound variables in the statement of the problem. There is one Turing machine that accepts any input and one that rejects any input. One of these answers Fermat's conjecture correctly, even though we do not know which one. In fact there may not even be a resolution to the conjecture using the axioms of arithmetic. That is, Fermat's conjecture may be true, yet there may be no arithmetic proof of that fact. The possibility (though not the certainty) that this is the case follows from Godel's incompleteness theorem, which states that any consistent formal system powerful enough to encompass number theory must have statements that are true but not provable within the system.

It should not disturb the reader that a conundrum like Fermat's conjecture is "decidable." The theory of undecidability is concerned with the existence or nonexistence of algorithms for solving problems with an infinity of instances.

8.2 PROPERTIES OF RECURSIVE AND RECURSIVELY ENUMERABLE LANGUAGES

A number of theorems in this chapter are proved by reducing one problem to another. These reductions involve combining several Turing machines to form a composite machine. The state of the composite TM has a component for each individual component machine. Similarly the composite machine has separate tapes for each individual machine. The details of the composite machine are usually tedious and provide no insight. Thus we choose to informally describe the constructions.

Given an algorithm (TM that always halts), we can allow the composite TM to perform one action if the algorithm accepts and another if it does not accept. We could not do this if we were given an arbitrary TM rather than an algorithm, since if the TM did not accept, it might run forever, and the composite machine would never initiate the next task. In pictures, an arrow into a box labeled "start" indicates a start signal. Boxes with no "start" signal are assumed to begin operating when the composite machine does. Algorithms have two outputs, "yes" and "no," which can be used as start signals or as a response by the composite machine. Arbitrary TM's have only a "yes" output, which can be used for the same purposes.

We now turn to some basic closure properties of the classes of recursive and r.e. sets.

Theorem 8.1 The complement of a recursive language is recursive.

Proof Let L be a recursive language and M a Turing machine that halts on all inputs and accepts L. Construct M' from M so that if M enters a final state on input w, then M' halts without accepting. If M halts without accepting, M' enters a

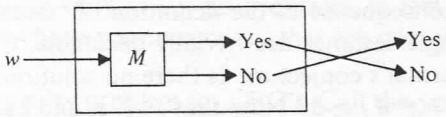

Fig. 8.1 Construction showing that recursive languages are closed under complementation.

final state. Since one of these two events occurs, M' is an algorithm. Clearly $L(M')$ is the complement of L and thus the complement of L is a recursive language. Figure 8.1 pictures the construction of M' from M. □

Theorem 8.2 The union of two recursive languages is recursive. The union of two recursively enumerable languages is recursively enumerable.

Proof Let L_1 and L_2 be recursive languages accepted by algorithms M_1 and M_2. We construct M, which first simulates M_1. If M_1 accepts, then M accepts. If M_1 rejects, then M simulates M_2 and accepts if and only if M_2 accepts. Since both M_1 and M_2 are algorithms, M is guaranteed to halt. Clearly M accepts $L_1 \cup L_2$.

For recursively enumerable languages the above construction does not work, since M_1 may not halt. Instead M can simultaneously simulate M_1 and M_2 on separate tapes. If either accepts, then M accepts. Figure 8.2 shows the two constructions of this theorem. □

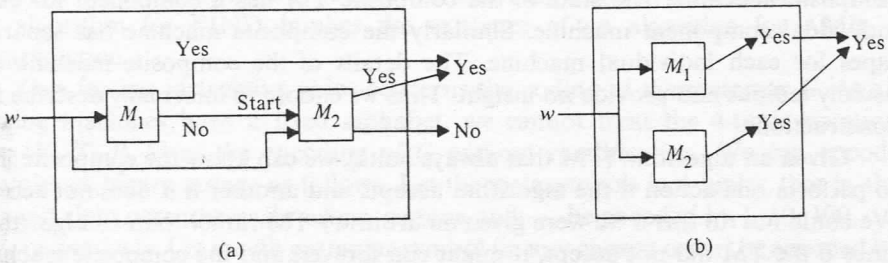

Fig. 8.2 Construction for union.

Theorem 8.3 If a language L and its complement $\bar{L}$ are both recursively enumerable, then L (and hence $\bar{L}$) is recursive.

Proof Let M_1 and M_2 accept L and $\bar{L}$ respectively. Construct M as in Fig. 8.3 to simulate simultaneously M_1 and M_2. M accepts w if M_1 accepts w and rejects w if M_2 accepts w. Since w is in either L or $\bar{L}$, we know that exactly one of M_1 or M_2 will accept. Thus M will always say either "yes" or "no," but will never say both. Note that there is no *a priori* limit on how long it may take before M_1 or M_2 accepts, but it is certain that one or the other will do so. Since M is an algorithm that accepts L, it follows that L is recursive. □

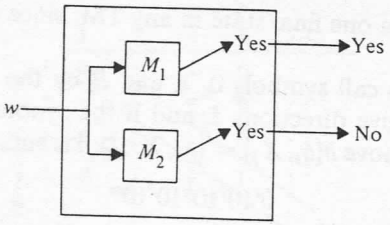

Fig. 8.3 Construction for Theorem 8.3.

Theorems 8.1 and 8.3 have an important consequence. Let L and $\bar{L}$ be a pair of complementary languages. Then either

1) both L and $\bar{L}$ are recursive,
2) neither L nor $\bar{L}$ is r.e., or
3) one of L and $\bar{L}$ is r.e. but not recursive; the other is not r.e.

An important technique for showing a problem undecidable is to show by diagonalization that the complement of the language for that problem is not r.e. Thus case (2) or (3) above must apply. This technique is essential in proving our first problem undecidable. After that, various forms of reductions may be employed to show other problems undecidable.

8.3 UNIVERSAL TURING MACHINES AND AN UNDECIDABLE PROBLEM

We shall now use diagonalization to show a particular problem to be undecidable. The problem is: "Does Turing machine M accept input w?" Here, both M and w are parameters of the problem. In formalizing the problem as a language we shall restrict w to be over alphabet $\{0, 1\}$ and M to have tape alphabet $\{0, 1, B\}$. As the restricted problem is undecidable, the more general problem is surely undecidable as well. We choose to work with the more restricted version to simplify the encoding of problem instances as strings.

Turing machine codes

To begin, we encode Turing machines with restricted alphabets as strings over $\{0, 1\}$. Let

$$M = (Q, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\})$$

be a Turing machine with input alphabet $\{0, 1\}$ and the blank as the only additional tape symbol. We further assume that $Q = \{q_1, q_2, \ldots, q_n\}$ is the set of states, and that q_2 is the only final state. Theorem 7.10 assures us that if $L \subseteq (\mathbf{0} + \mathbf{1})^*$ is accepted by any TM, then it is accepted by one with alphabet $\{0, 1, B\}$. Also, there

is no need for more than one final state in any TM, since once it accepts it may as well halt.

It is convenient to call symbols 0, 1, and B by the synonyms X_1, X_2, X_3, respectively. We also give directions L and R the synonyms D_1 and D_2, respectively. Then a generic move $\delta(q_i, X_j) = (q_k, X_\ell, D_m)$ is encoded by the binary string

$$0^i10^j10^k10^\ell10^m. \tag{8.1}$$

A binary code for Turing machine M is

$$111\ \text{code}_1\ 11\ \text{code}_2\ 11 \cdots 11\ \text{code}_r\ 111, \tag{8.2}$$

where each code_i is a string of the form (8.1), and each move of M is encoded by one of the code_i's. The moves need not be in any particular order, so each TM actually has many codes. Any such code for M will be denoted $\langle M \rangle$.

Every binary string can be interpreted as the code for at most one TM; many binary strings are not the code of any TM. To see that decoding is unique, note that no string of the form (8.1) has two 1's in a row, so the code_i's can be found directly. If a string fails to begin and end with exactly three 1's, has three 1's other than at the end, or has two pair of 1's with other than five blocks of 0's in between, then the string represents no TM.

The pair M and w is represented by a string of the form (8.2) followed by w. Any such string will be denoted $\langle M, w \rangle$.

Example 8.1 Let $M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\})$ have moves:

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

Thus one string denoted by $\langle M, 1011 \rangle$ is

$$11101001000101001100010101001 0011$$

$$000100100101001100010001000100101111011$$

Note that many different strings are also codes for the pair $\langle M, 1011 \rangle$, and any of these may be referred to by the notation $\langle M, 1011 \rangle$.

A non-r.e. language

Suppose we have a list of $(\mathbf{0} + \mathbf{1})^*$ in canonical order (see Section 7.7), where w_i is the ith word, and M_j is the TM whose code, as in (8.2) is the integer j written in

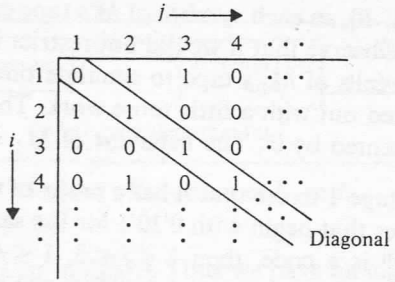

Fig. 8.4 Hypothetical table indicating acceptance of words by TM's.

binary. Imagine an infinite table that tells for all i and j whether w_i is in $L(M_j)$. Figure 8.4 suggests such a table;† 0 means w_i is not in $L(M_j)$ and 1 means it is.

We construct a language L_d by using the diagonal entries of the table to determine membership in L_d. To guarantee that no TM accepts L_d, we insist that w_i is in L_d if and only if the (i, i) entry is 0, that is, if M_i does not accept w_i. Suppose that some TM M_j accepted L_d. Then we are faced with the following contradiction. If w_j is in L_d, then the (j, j) entry is 0, implying that w_j is not in $L(M_j)$ and contradicting $L_d = L(M_j)$. On the other hand, if w_j is not in L_d, then the (j, j) entry is 1, implying that w_j is in $L(M_j)$, which again contradicts $L_d = L(M_j)$. As w_j is either in or not in L_d, we conclude that our assumption, $L_d = L(M_j)$, is false. Thus, no TM in the list accepts L_d, and by Theorem 7.10, no TM whatsoever accepts L_d.

We have thus proved

Lemma 8.1 L_d is not r.e.

The universal language

Define L_u, the "universal language," to be $\{\langle M, w\rangle \mid M \text{ accepts } w\}$. We call L_u "universal" since the question of whether any particular string w in $(\mathbf{0} + \mathbf{1})^*$ is accepted by any particular Turing machine M is equivalent to the question of whether $\langle M', w\rangle$ is in L_u, where M' is the TM with tape alphabet $\{0, 1, B\}$ equivalent to M constructed as in Theorem 7.10.

Theorem 8.4 L_u is recursively enumerable.

Proof We shall exhibit a three-tape TM M_1 accepting L_u. The first tape of M_1 is the input tape, and the input head on that tape is used to look up moves of the TM M when given code $\langle M, w\rangle$ as input. Note that the moves of M are found between the first two blocks of three 1's. The second tape of M_1 will simulate the tape of M.

† Actually as all low-numbered Turing machines accept the empty set, the correct portion of the table shown has all 0's.

The alphabet of M is $\{0, 1, B\}$, so each symbol of M's tape can be held in one tape cell of M_1's second tape. Observe that if we did not restrict the alphabet of M, we would have to use many cells of M_1's tape to simulate one of M's cells, but the simulation could be carried out with a little more work. The third tape holds the state of M, with q_i represented by 0^i. The behavior of M_1 is as follows:

1) Check the format of tape 1 to see that it has a prefix of the form (8.2) and that there are no two codes that begin with 0^i10^j1 for the same i and j. Also check that if $0^i10^j10^k10^\ell10^m$ is a code, then $1 \leq j \leq 3$, $1 \leq \ell \leq 3$, and $1 \leq m \leq 2$. Tape 3 can be used as a scratch tape to facilitate the comparison of codes.
2) Initialize tape 2 to contain w, the portion of the input beyond the second block of three 1's. Initialize tape 3 to hold a single 0, representing q_1. All three tape heads are positioned on the leftmost symbols. These symbols may be marked so the heads can find their way back.
3) If tape 3 holds 00, the code for the final state, halt and accept.
4) Let X_j be the symbol currently scanned by tape head 2 and let 0^i be the current contents of tape 3. Scan tape 1 from the left end to the second 111, looking for a substring beginning 110^i10^j1. If no such string is found, halt and reject; M has no next move and has not accepted. If such a code is found, let it be $0^i10^j10^k10^\ell10^m$. Then put 0^k on tape 3, print X_ℓ on the tape cell scanned by head 2 and move that head in direction D_m. Note that we have checked in (1) that $1 \leq \ell \leq 3$ and $1 \leq m \leq 2$. Go to step (3).

It is straightforward to check that M_1 accepts $\langle M, w\rangle$ if and only if M accepts w. It is also true that if M runs forever on w, M_1 will run forever on $\langle M, w\rangle$, and if M halts on w without accepting, M_1 does the same on $\langle M, w\rangle$. □

The existence of M_1 is sufficient to prove Theorem 8.4. However, by Theorems 7.2 and 7.10, we can find a TM with one semi-infinite tape and alphabet $\{0, 1, B\}$ accepting L_u. We call this particular TM M_u, the *universal Turing machine*, since it does the work of any TM with input alphabet $\{0, 1\}$.

By Lemma 8.1, the diagonal language L_d is not r.e., and hence not recursive. Thus by Theorem 8.1, $\bar{L}_d$ is not recursive. Note that $\bar{L}_d = \{w_i \mid M_i \text{ accepts } w_i\}$. We can prove the universal language $L_u = \{\langle M, w\rangle \mid M \text{ accepts } w\}$ not to be recursive by reducing $\bar{L}_d$ to L_u. Thus L_u is an example of a language that is r.e. but not recursive. In fact, $\bar{L}_d$ is another example of such a language.

Theorem 8.5 L_u is not recursive.

Proof Suppose A were an algorithm recognizing L_u. Then we could recognize $\bar{L}_d$ as follows. Given string w in $(\mathbf{0} + \mathbf{1})^*$, determine by an easy calculation the value of i such that $w = w_i$. Integer i in binary is the code for some TM M_i. Feed $\langle M_i, w_i\rangle$ to algorithm A and accept w if and only if M_i accepts w_i. The construction is shown in Fig. 8.5. It is easy to check that the constructed algorithm accepts w if

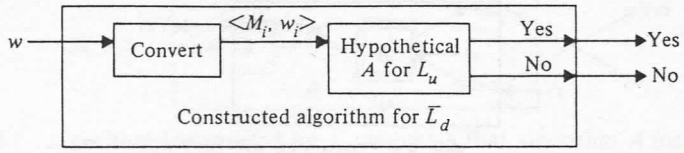

Fig. 8.5 Reduction of $\bar{L}_d$ to L_u.

and only if $w = w_i$ and w_i is in $L(M_i)$. Thus we have an algorithm for $\bar{L}_d$ Since no such algorithm exists, we know our assumption, that algorithm A for L_u exists, is false. Hence L_u is r.e. but not recursive. □

8.4 RICE'S THEOREM AND SOME MORE UNDECIDABLE PROBLEMS

We now have an example of an r.e. language that is not recursive. The associated problem "Does M accept w?" is undecidable, and we can use this fact to show that other problems are undecidable. In this section we shall give several examples of undecidable problems concerning r.e. sets. In the next three sections we shall discuss some undecidable problems taken from outside the realm of TM's.

Example 8.2 Consider the problem: "Is $L(M) \neq \varnothing$?" Let $\langle M \rangle$ denote a code for M as in (8.2). Then define

$$L_{ne} = \{\langle M \rangle \mid L(M) \neq \varnothing\} \quad \text{and} \quad L_e = \{\langle M \rangle \mid L(M) = \varnothing\}.$$

Note that L_e and L_{ne} are complements of one another, since every binary string denotes some TM; those with a bad format denote the TM with no moves. All these strings are in L_e. We claim that L_{ne} is r.e. but not recursive and that L_e is not r.e.

We show that L_{ne} is r.e. by constructing a TM M to recognize codes of TM's that accept nonempty sets. Given input $\langle M_i \rangle$, M nondeterministically guesses a string x accepted by M_i and verifies that M_i does indeed accept x by simulating M_i on input x. This step can also be carried out deterministically if we use the pair generator described in Section 7.7. For pair (j, k) simulate M_i on the jth binary string (in canonical order) for k steps. If M_i accepts, then M accepts $\langle M_i \rangle$.

Now we must show that L_e is not recursive. Suppose it were. Then we could construct an algorithm for L_u, violating Theorem 8.5. Let A be a hypothetical algorithm accepting L_e. There is an algorithm B that, given $\langle M, w \rangle$, constructs a TM M' that accepts $\varnothing$ if M does not accept w and accepts $(\mathbf{0} + \mathbf{1})^*$ if M accepts w. The plan of M' is shown in Fig. 8.6. M' ignores its input x and instead simulates M on input w, accepting if M accepts.

Note that M' is not B. Rather, B is like a compiler that takes $\langle M, w \rangle$ as "source program" and produces M' as "object program." We have described what B must do, but not how it does it. The construction of B is simple. It takes $\langle M, w \rangle$

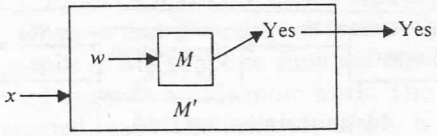

Fig. 8.6 The TM M'.

and isolates w. Say $w = a_1 a_2 \cdots a_n$ is of length n. B creates $n + 3$ states $q_1, q_2, \ldots, q_{n+3}$ with moves

$$\delta(q_1, X) = (q_2, \$, R) \text{ for any } X \text{ (print marker)},$$

$$\delta(q_i, X) = (q_{i+1}, a_{i-1}, R) \text{ for any } X \text{ and } 2 \leq i \leq n + 1 \text{ (print } w),$$

$$\delta(q_{n+2}, X) = (q_{n+2}, B, R) \text{ for } X \neq B \text{ (erase tape)},$$

$$\delta(q_{n+2}, B) = (q_{n+3}, B, L),$$

$$\delta(q_{n+3}, X) = (q_{n+3}, X, L) \text{ for } X \neq \$ \text{ (find marker)}.$$

Having produced the code for these moves, B then adds $n + 3$ to the indices of the states of M and includes the move

$$\delta(q_{n+3}, \$) = (q_{n+4}, \$, R) \text{ /* start up } M \text{ */}$$

and all the moves of M in its generated TM. The resulting TM has an extra tape symbol \$, but by Theorem 7.10 we may construct M' with tape alphabet $\{0, 1, B\}$, and we may surely make q_2 the accepting state. This step completes the algorithm B, and its output is the desired M' of Fig. 8.6.

Now suppose algorithm A accepting L_e exists. Then we construct an algorithm C for L_u as in Fig. 8.7. If M accepts w, then $L(M') \neq \varnothing$; so A says "no" and C says "yes." If M does not accept w, then $L(M') = \varnothing$, A says "yes," and C says "no." As C does not exist by Theorem 8.5, A cannot exist. Thus, L_e is not recursive. If L_{ne} were recursive, L_e would be also by Theorem 8.1. Thus L_{ne} is r.e. but not recursive. If L_e were r.e., then L_e and L_{ne} would be recursive by Theorem 8.3. Thus L_e is not r.e.

Example 8.3 Consider the language

$$L_r = \{\langle M \rangle \mid L(M) \text{ is recursive}\}$$

and

$$L_{nr} = \{\langle M \rangle \mid L(M) \text{ is not recursive}\}.$$

Note that L_r is not $\{\langle M \rangle \mid M$ halts on all inputs$\}$, although it includes the latter language. A TM M could accept a recursive language even though M itself might loop forever on some words not in $L(M)$; some other TM equivalent to M must always halt, however. We claim neither L_r nor L_{nr} is r.e.

Suppose L_r were r.e. Then we could construct a TM for $\bar{L}_u$ which we know

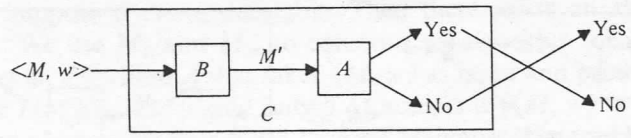

Fig. 8.7 Algorithm constructed for L_u assuming that algorithm A for L_e exists.

does not exist. Let M_r be a TM accepting L_r. We may construct an algorithm A that takes $\langle M, w \rangle$ as input and produces as output a TM M' such that

$$L(M') = \begin{cases} \varnothing & \text{if } M \text{ does not accept } w, \\ L_u & \text{if } M \text{ accepts } w. \end{cases}$$

Note that L_u is not recursive, so M' accepts a recursive language if and only if M does not accept w. The plan of M' is shown in Fig. 8.8. As in the previous example, we have described the output of A. We leave the construction of A to the reader.

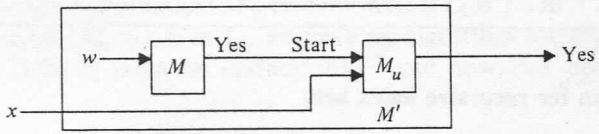

Fig. 8.8 The TM M'.

Given A and M_r we could construct a TM accepting $\bar{L}_u$, shown in Fig. 8.9, which behaves as follows. On input $\langle M, w \rangle$ the TM uses A to produce M', uses M_r to determine if the set accepted by M' is recursive, and accepts if and only if $L(M')$ is recursive. But $L(M')$ is recursive if and only if $L(M') = \varnothing$, which means M does not accept w. Thus the TM of Fig. 8.9 accepts $\langle M, w \rangle$ if and only if $\langle M, w \rangle$ is in $\bar{L}_u$.

Now let us turn to L_{nr}. Suppose we have a TM M_{nr} accepting L_{nr}. Then we may use M_{nr} and an algorithm B, to be constructed by the reader, to accept $\bar{L}_u$. B takes $\langle M, w \rangle$ as input and produces as output a TM M' such that

$$L(M') = \begin{cases} \Sigma^* & \text{if } M \text{ accepts } w, \\ L_u & \text{if } M \text{ does not accept } w. \end{cases}$$

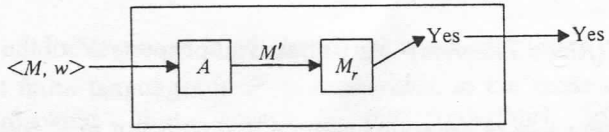

Fig. 8.9 Hypothetical TM for $\bar{L}_u$.

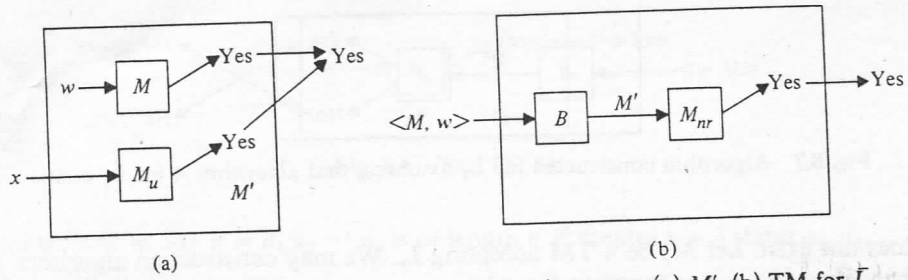

Fig. 8.10 Constructions used in proof that L_{nr} is not r.e. (a) M'. (b) TM for $\bar{L}_u$.

Thus M' accepts a recursive language if and only if M accepts w. M', which B must produce, is shown in Fig. 8.10(a), and a TM to accept $\bar{L}_u$ given B and M_{nr}, is shown in Fig. 8.10(b). The TM of Fig. 8.10(b) accepts $\langle M, w \rangle$ if and only if $L(M')$ is not recursive, or equivalently, if and only if M does not accept w. That is, the TM accepts $\langle M, w \rangle$ if and only if $\langle M, w \rangle$ is in $\bar{L}_u$. Since we have already shown that no such TM exists, the assumption that M_{nr} exists is false. We conclude that L_{nr} is not r.e.

Rice's Theorem for recursive index sets

The above examples show that we cannot decide if the set accepted by a Turing machine is empty or recursive. The technique of proof can also be used to show that we cannot decide if the set accepted is finite, infinite, regular, context free, has an even number of strings, or satisfies many other predicates. What then can we decide about the set accepted by a TM? Only the trivial predicates, such as "Does the TM accept an r.e. set?," which are either true for all TM's or false for all TM's.

In what follows we shall discuss languages that represent properties of r.e. languages. That is, the languages are sets of TM codes such that membership of $\langle M \rangle$ in the language depends only on $L(M)$, not on M itself. Later we shall consider languages of TM codes that depend on the TM itself, such as "M has 27 states," which may be satisfied for some but not all of the TM's accepting a given language.

Let $\mathscr{S}$ be a set of r.e. languages, each a subset of $(\mathbf{0} + \mathbf{1})^*$. $\mathscr{S}$ is said to be a *property of the r.e. languages*. A set L has property $\mathscr{S}$ if L is an element of $\mathscr{S}$. For example, the property of being infinite is $\{L \mid L \text{ is infinite}\}$. $\mathscr{S}$ is a *trivial* property if $\mathscr{S}$ is empty or $\mathscr{S}$ consists of all r.e. languages. Let $L_{\mathscr{S}}$ be the set $\{\langle M \rangle \mid L(M) \text{ is in } \mathscr{S}\}$.

Theorem 8.6 (*Rice's Theorem*) Any nontrivial property $\mathscr{S}$ of the r.e. languages is undecidable.

Proof Without loss of generality assume that $\varnothing$ is not in $\mathscr{S}$ (otherwise consider $\bar{\mathscr{S}}$). Since $\mathscr{S}$ is nontrivial, there exists L with property $\mathscr{S}$. Let M_L be a TM

accepting L. Suppose $\mathscr{S}$ were decidable. Then there exists an algorithm $M_{\mathscr{S}}$ accepting $L_{\mathscr{S}}$. We use M_L and $M_{\mathscr{S}}$ to construct an algorithm for L_u as follows. First construct an algorithm A that takes $\langle M, w\rangle$ as input and produces $\langle M'\rangle$ as output, where $L(M')$ is in $\mathscr{S}$ if and only if M accepts w ($\langle M, w\rangle$ is in L_u).

The design of M' is shown in Fig. 8.11. First M' ignores its input and simulates M on w. If M does not accept w, then M' does not accept x. If M accepts w, then M' simulates M_L on x, accepting x if and only if M_L accepts x. Thus M' either accepts $\varnothing$ or L depending on whether M accepts w.

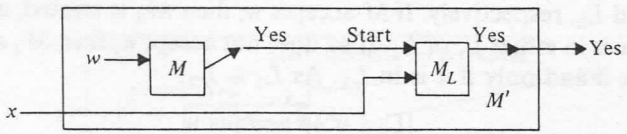

Fig. 8.11 M' used in Rice's theorem.

We may use the hypothetical $M_{\mathscr{S}}$ to determine if $L(M')$ is in $\mathscr{S}$. Since $L(M')$ is in $\mathscr{S}$ if and only if $\langle M, w\rangle$ is in L_u, we have an algorithm for recognizing L_u, a contradiction. Thus $\mathscr{S}$ must be undecidable. Note how this proof generalizes Example 8.2. □

Theorem 8.6 has a great variety of consequences, some of which are summarized in the following corollary.

Corollary The following properties of r.e. sets are not decidable:

a) emptiness,
b) finiteness,
c) regularity,
d) context-freedom.

Rice's Theorem for recursively enumerable index sets

The condition under which a set $L_{\mathscr{S}}$ is r.e. is far more complicated. We shall show that $L_{\mathscr{S}}$ is r.e. if and only if $\mathscr{S}$ satisfies the following three conditions.

1) If L is in $\mathscr{S}$ and $L \subseteq L'$, for some r.e. L', then L' is in $\mathscr{S}$ (the *containment property*).
2) If L is an infinite language in $\mathscr{S}$, then there is a finite subset of L in $\mathscr{S}$.
3) The set of finite languages in $\mathscr{S}$ is *enumerable*, in the sense that there is a Turing machine that generates the (possibly) infinite string $\text{code}_1\#\text{code}_2\#\ \ldots$, where code_i is a code for the ith finite language in $\mathscr{S}$ (in

any order). The code for the finite language $\{w_1, w_2, \ldots, w_n\}$ is just w_1, $w_2, \ldots, w_n$.

We prove this characterization with a series of lemmas.

Lemma 8.2 If $\mathcal{S}$ does not have the containment property, then $L_{\mathcal{S}}$ is not r.e.

Proof We generalize the proof that L_{nr} is not r.e. Let L_1 be in $\mathcal{S}$, $L_1 \subseteq L_2$, and let L_2 not be in $\mathcal{S}$. [For the case where $\mathcal{S}$ was the nonrecursive sets, we chose $L_1 = L_u$ and $L_2 = (0 + 1)^*$.] Construct algorithm A that takes as input $\langle M, w \rangle$ and produces as output TM M' with the behavior shown in Fig. 8.12, where M_1 and M_2 accept L_1 and L_2, respectively. If M accepts w, then M_2 is started, and M' accepts x whenever x is in either L_1 or L_2. If M does not accept w, then M_2 never starts, so M' accepts x if and only if x is in L_1. As $L_1 \subseteq L_2$,

$$L(M') = \begin{cases} L_2 & \text{if } M \text{ accepts } w, \\ L_1 & \text{if } M \text{ does not accept } w. \end{cases}$$

Thus $L(M')$ is in $\mathcal{S}$ if and only if M does not accept w.

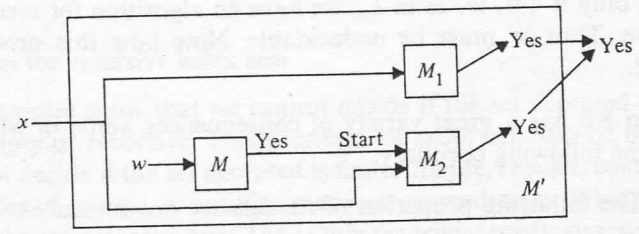

Fig. 8.12 The TM M'.

We again leave it to the reader to design the "compiler" A that takes $\langle M, w \rangle$ as input and connects them with the fixed Turing machines M_1 and M_2 to construct the M' shown in Fig. 8.12. Having constructed A, we can use a TM $M_{\mathcal{S}}$ for $L_{\mathcal{S}}$ to accept $\bar{L}_u$, as shown in Fig. 8.13. This TM accepts $\langle M, w \rangle$ if and only if M' accepts a language in $\mathcal{S}$, or equivalently, if and only if M does not accept w. As such a TM does not exist, we know $M_{\mathcal{S}}$ cannot exist, so $L_{\mathcal{S}}$ is not r.e. □

We now turn to the second property of recursively enumerable index sets.

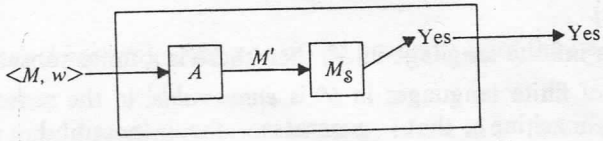

Fig. 8.13 Hypothetical TM to accept $\bar{L}_u$.

Lemma 8.3 If $\mathcal{S}$ has an infinite language L such that no finite subset of L is in $\mathcal{S}$, then $L_{\mathcal{S}}$ is not r.e.

Proof Suppose $L_{\mathcal{S}}$ were r.e. We shall show that $\bar{L}_u$ would be r.e. as follows. Let M_1 be a TM accepting L. Construct algorithm A to take a pair $\langle M, w\rangle$ as input and produce as output a TM M' that accepts L if w is not in $L(M)$ and accepts some finite subset of L otherwise. As shown in Fig. 8.14, M' simulates M_1 on its input x. If M_1 accepts x, then M' simulates M on w for $|x|$ moves. If M fails to accept w after $|x|$ moves, then M' accepts x. We leave the design of algorithm A as an exercise.

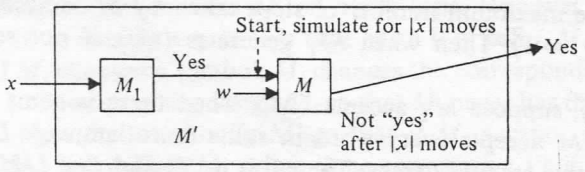

Fig. 8.14 Construction of M'.

If w is in $L(M)$, then M accepts w after some number of moves, say j. Then $L(M') = \{x \mid x \text{ is in } L \text{ and } |x| < j\}$, which is a finite subset of L. If w is not in $L(M)$, then $L(M') = L$. Hence, if M does not accept w, $L(M')$ is in $\mathcal{S}$, and if M accepts w, $L(M')$, being a finite subset of L, is not in $\mathcal{S}$ by the hypothesis of the lemma. An argument that is by now standard proves that if $L_{\mathcal{S}}$ is r.e., so is $\bar{L}_u$. Since the latter is not r.e., we conclude the former is not either. □

Finally, consider the third property of r.e. index sets.

Lemma 8.4 If $L_{\mathcal{S}}$ is r.e., then the list of binary codes for the finite sets in $\mathcal{S}$ is enumerable.

Proof We use the pair generator described in Section 7.7. When (i, j) is generated, we treat i as the binary code of a finite set, assuming 0 is the code for comma, 10 the code for zero, and 11 the code for one. We may in a straightforward manner construct a TM $M^{(i)}$ (essentially a finite automaton) that accepts exactly the words in the finite language represented by i. We then simulate the enumerator for $L_{\mathcal{S}}$ for j steps. If it has printed $M^{(i)}$, we print the code for the finite set represented by i, that is, the binary representation of i itself, followed by a delimiter symbol #. In any event, after the simulation we return control to the pair generator, which generates the pair following (i, j). □

Theorem 8.7 $L_{\mathcal{S}}$ is r.e. if and only if

1) If L is in $\mathcal{S}$ and $L \subseteq L'$, for some r.e. L', then L' is in $\mathcal{S}$.
2) If L is an infinite set in $\mathcal{S}$, then there is some finite subset L' of L that is in $\mathcal{S}$.
3) The set of finite languages in $\mathcal{S}$ is enumerable.

Proof The "only if" part is Lemmas 8.2, 8.3, and 8.4. For the "if" part, suppose (1), (2), and (3) hold. We construct a TM M_1 that recognizes $\langle M \rangle$ if and only if $L(M)$ is in $\mathcal{S}$ as follows. M_1 generates pairs (i, j) using the pair generator. In response to (i, j), M_1 simulates M_2, which is an enumerator of the finite sets in $\mathcal{S}$, for i steps. We know M_2 exists by condition (3). Let L_1 be the last set completely printed out by M_2. [If there is no set completely printed, generate the next (i, j) pair.] Then simulate M for j steps on each word in L_1. If M accepts all words in L_1, then M_1 accepts $\langle M \rangle$. If not, M_1 generates the next (i, j)-pair.

We use conditions (1) and (2) to show that $L(M_1) = L_{\mathcal{S}}$. Suppose L is in $L_{\mathcal{S}}$, and let M be any TM with $L(M) = L$. By condition (2), there is a finite $L' \subseteq L$ in $\mathcal{S}$ (take $L' = L$ if L is finite). Let L' be generated after i steps of M_2, and let j be the maximum number of steps taken by M to accept a word in L' (if $L' = \varnothing$, let $j = 1$). Then when M_1 generates (i, j), if not sooner, M_1 will accept $\langle M \rangle$.

Conversely, suppose M_1 accepts $\langle M \rangle$. Then there is some (i, j) such that within j steps M accepts every word in some finite language L' such that M_2 generates L' within its first i steps. Then L' is in $\mathcal{S}$, and $L' \subseteq L(M)$. By condition (1), $L(M)$ is in $\mathcal{S}$, so $\langle M \rangle$ is in $L_{\mathcal{S}}$. We conclude that $L(M_1) = L_{\mathcal{S}}$. □

Theorem 8.7 has a great variety of consequences. We summarize some of them as corollaries and leave others as exercises.

Corollary 1 The following properties of r.e. sets are not r.e.

a) $L = \varnothing$.

b) $L = \Sigma^*$.

c) L is recursive.

d) L is not recursive.

e) L is a *singleton* (has exactly one member).

f) L is a regular set.

g) $L - L_u \neq \varnothing$.

Proof In each case condition (1) is violated, except for (b), where (2) is violated, and (g), where (3) is violated. □

Corollary 2 The following properties of r.e. sets are r.e.

a) $L \neq \varnothing$.

b) L contains at least 10 members.

c) w is in L for some fixed word w.

d) $L \cap L_u \neq \varnothing$.

Problems about Turing machines

Does Theorem 8.6 say that everything about Turing machines is undecidable? The answer is no. That theorem has to do only with properties of the language accepted, not properties of the Turing machine itself. For example, the question "Does a given Turing machine have an even number of states?" is clearly decidable. When dealing with properties of Turing machines themselves, we must use our ingenuity. We give two examples.

Example 8.4 It is undecidable if a Turing machine with alphabet $\{0, 1, B\}$ ever prints three consecutive 1's on its tape. For each Turing machine M_i we construct $\hat{M}_i$, which on blank tape simulates M_i on blank tape. However, $\hat{M}_i$ uses 01 to encode a 0 and 10 to encode a 1. If M_i's tape has a 0 in cell j, $\hat{M}_i$ has 01 in cells $2j - 1$ and $2j$. If M_i changes a symbol, $\hat{M}_i$ changes the corresponding 1 to 0, then the paired 0 to 1. One can easily design $\hat{M}_i$ so that $\hat{M}_i$ never has three consecutive 1's on its tape. Now further modify $\hat{M}_i$ so that if M_i accepts, $\hat{M}_i$ prints three consecutive 1's and halts. Thus $\hat{M}_i$ prints three consecutive 1's if and only if M_i accepts ϵ. By Theorem 8.6, it is undecidable whether a TM accepts ϵ, since the predicate "ϵ is in L" is not trivial. Thus the question of whether an arbitrary Turing machine ever prints three consecutive 1's is undecidable.

Example 8.5 It is decidable whether a single-tape Turing machine started on blank tape scans any cell four or more times. If the Turing machine never scans any cell four or more times, than every *crossing sequence* (sequence of states in which the boundary between cells is crossed, assuming states change before the head moves) is of length at most three. But there is a finite number of distinct crossing sequences of length three or less. Thus either the Turing machine stays within a fixed bounded number of tape cells, in which case finite automaton techniques answer the question, or some crossing sequence repeats. But if some crossing sequence repeats, then the TM moves right with some easily detectable pattern, and the question is again decidable.

8.5 UNDECIDABILITY OF POST'S CORRESPONDENCE PROBLEM

Undecidable problems arise in a variety of areas. In the next three sections we explore some of the more interesting problems in language theory and develop techniques for proving particular problems undecidable. We begin with Post's Correspondence Problem, it being a valuable tool in establishing other problems to be undecidable.

An instance of *Post's Correspondence Problem* (*PCP*) consists of two lists, $A = w_1, \ldots, w_k$ and $B = x_1, \ldots, x_k$, of strings over some alphabet Σ. This instance