17

Advanced Undecidability Proofs

In this chapter, we will discuss Rice's Theorem in Section 17.1, and the computational history method in Section 17.3. As discussed in Chapter 16, these are two additional important methods to approach the question of decidability.

17.1 Rice's Theorem

Rice's Theorem asserts:

Theorem 17.1. Every non-trivial partitioning of the space of Turing machine codes based on the languages recognized by these Turing machines is undecidable.

Rice's Theorem is, basically, a *general* statement about the undecidability of non-trivial partitions one can erect on Turing machine codes based on the language that the corresponding Turing machines accept. Stated another way, Rice's Theorem asserts the impossibility of building an *algorithmic* classifier for Turing machine codes ("programs") based on the *language* recognized by these Turing machines, if the classification attempted is anything but trivial (a trivial partition puts all the Turing machines into one or the other bin).

Relating these notions more to real-world programs, consider the language L consisting of all ASCII character sequences $s_1, s_2, \ldots$ such that each s_i is a C program c_i. Now suppose that each c_i, when run on inputs from $\{0,1\},^*$ accepts only those sequences that describe a regular set. *Rice's Theorem says that languages such as L are not decidable.* In other words, it is impossible to classify all C programs (or equivalently TMs) into those *whose* languages are regular and those whose languages are non-regular. Mathematically, given a property $\mathcal{P}$, consider the set

$$L = \{\langle M \rangle \mid M \text{ is a Turing machine and } \mathcal{P}(Lang(M))\}.$$

Furthermore, let $\mathcal{P}$ be non-trivial — meaning, it is neither $\emptyset$ nor Σ^*. For example, $\mathcal{P}$ could be "*Regular*"; since there *are* TMs that encode regular sets and there are TMs that do not encode regular sets, $\mathcal{P}$ represents a non-trivial partition over TR languages. Rice's Theorem asserts that sets such as L above are undecidable.

17.2 Failing proof attempt

```
M'(x)
  Run M on w ;
  IF this run ends at state reject_M, THEN loop ;
  Manifest N;
  RUN N on x ;
  IF this run accepts, THEN goto accept_M' ;
  IF this run rejects, THEN goto reject_M' ;
```

Fig. 17.1. Machine $M^{'}$ in the proof of Rice's Theorem

For the ease of exposition, we present, as a special case of Rice's Theorem, the proof of Rice's Theorem for $\mathcal{P} = Regular$. Our first proof attempt will fail because of a small technical glitch. The glitch is caused by $\emptyset \in \mathcal{P}$, or in other words, allowing $\mathcal{P}(\emptyset)$. In our special case proof, this glitch manifests in the form of $\emptyset \in Regular$, as we are proving for the special case of $\mathcal{P} = Regular$. We fix this glitch in Section 17.2.1.

Proof, with a small glitch, of a special case of Rice's Theorem:

By contradiction, assume that L has a decider, namely D_L. The Turing machine D_L is capable of classifying Turing machine codes into those whose languages pass the predicate test *Regular*. As noted earlier, *Regular* is a non-trivial property. Therefore, given D_L, it should be possible to find at least *one* Turing machine— say N — whose language is regular (it does not matter which Turing machine this is). In particular, algorithm *Manifest* N is:

> *Find the first (in numeric order) string from Σ^* accepted by D_L.*

Now, we use *Manifest* N in defining a machine $M^{'}$ shown in Figure 17.1. Then, using $M^{'}$ (built with respect to arbitrary M and x), we try to derive a contradiction:

- If M accepts w, $L(M^{'}) = Lang(N)$, which is regular. Hence, $M^{'}$ is in $Regular_{TM}$.
- If M does not accept w, $L(M^{'}) = \emptyset$. Unfortunately, $\emptyset$ is also regular! Hence, *in this case too*, $M^{'}$ is in $Regular_{TM}$. *Alas, this is the place where we ought to have obtained a contradiction.* Somehow, we "blew our proof."
- In other words, if we feed $M^{'}$ to D_L, we will get *true* regardless of whether or not M accepts w. Therefore, no contradiction results.

17.2.1 Corrected proof

Surprisingly, the proof goes through if we take $\mathcal{P} = \textit{non-regular}$; in this case, we *will* obtain a contradiction. We redo our proof steps as follows:

- Now define L and D_L for the case of $\mathcal{P} = \textit{non-regular}$.
- Define *Manifest* N using D_L, which manifests a Turing machine N whose language is not regular.
- Define $M^{'}$ in terms of N. In other words, in $M^{'}$'s body, we will manifest N and also use it.

Let us analyze the language of $M^{'}$:

- If M accepts w, $L(M^{'}) = Lang(N)$, which is *non-regular*. Hence, $L(M^{'})$ is in *non-regular*.
- If M does not accept w, $L(M^{'}) = \emptyset$. $\emptyset$ is regular, and in this case, $L(M^{'})$ is NOT in *non-regular*.
- Therefore, if the decider D_L exists, we can generate $M^{'}$ and feed it to D_L. Then we will see that if M accepts w, $D_L(M^{'}) = true$, and if M does not accept w, $D_L(M^{'}) = false$. Hence, if D_L exists, we can decide A_{TM}. Since this is a contradiction, D_L cannot exist; and hence, we obtain a contradiction!

Summary of the proof of Rice's Theorem: We observe that if we go with $\emptyset \notin \mathcal{P}$, then the proof succeeds. The general proof of Rice's Theorem *assumes* that $\mathcal{P}$ does not contain $\emptyset$. This approach is justified; *there is no loss of generality.* This is because if $\mathcal{P}$ indeed contains $\emptyset$, then we can always proceed with $\neg\mathcal{P}$ as our "new $\mathcal{P}$" and finish the proof. When we fail to obtain a contradiction with respect to $\mathcal{P}$ but obtain a contradiction with respect to $\neg\mathcal{P}$, we would have shown that D_L — the decider of Turing machine codes whose language is $\neg\mathcal{P}$ — is

undecidable. But, this is tantamount to showing that D_L — the decider of Turing machine codes whose language is $\mathcal{P}$ — is undecidable.

Proof for a general non-trivial property $\mathcal{P}$

The reader can easily verify that substituting $\mathcal{P}$ for *Regular* in the previous proof makes everything work out. In particular, $M^{'}$'s language either would be $\emptyset$ or would be N. If $\emptyset \notin \mathcal{P}$, this would result in a full contradiction when D_L is fed $M^{'}$. If $\emptyset \in \mathcal{P}$, then $M^{'}$ would be constructed with respect to $M_{has\neg\mathcal{P}}$'s language, and even here full contradiction will result. Therefore, we would end up showing that either L defined with respect to an arbitrary non-trivial property $\mathcal{P}$ has no decider, or that L defined with respect to $\neg\mathcal{P}$ has no decider.

17.2.2 Greibach's Theorem

There is a theorem analogous to Rice's Theorem for PDAs. Known as *Greibach's Theorem*, the high-level statement of the theorem (in terms of its practical usage) is as follows:

> It is impossible to algorithmically classify (using, of course, a Turing machine) context-free grammars on the basis of whether their languages are regular or not.

For details, please see [61, page 205].

17.3 The Computation History Method

As we mentioned in Chapter 1, it is possible to "teach" LBAs (and NPDAs) to answer certain difficult questions about Turing machines. This idea is detailed in Section 17.3.2 through 17.3.4.

We first recap basic facts about linear bounded automata (LBA) and present a decidability result about them (Section 17.3.1). Thereafter, we present three undecidability results based on the computation history method: (i) emptiness of LBA languages (Section 17.3.2), (ii) universality of the language of a CFG (Section 17.3.3), and (iii) Post's correspondence problem (Section 17.3.4). In Chapter 18, Section 18.2.3, we emphasize the importance of the undecidability of Post's correspondence problem (PCP) by presenting a classic proof due to Robert Floyd: we reduce PCP to the validity problem for first-order logic. This proof then establishes that the validity problem for first-order logic is undecidable.

17.3.1 Decidability of LBA acceptance

LBAs are Turing machines that are allowed to access (read or write) only that region of the input tape where the input was originally presented. To enforce such a restriction, one may place two distinguished symbols, say ¢ and \$, around the original input string.[1] With these conventions, it can easily be seen that instantaneous descriptions of an LBA that begin as q_0w will change to the general form lqr, where $|lr| = |w|$. Therefore, there are a finite number, say N, of these IDs (see Exercise 17.1). A decider can simulate an LBA starting from ID q_0w, and see if it accepts within this many IDs; if not, the LBA will not accept its input. Hence, LBA acceptance is decidable.

17.3.2 Undecidability of LBA language emptiness

Suppose a Turing machine M accepts input w; it will then have an accepting computational history of IDs starting with q_0w, going through intermediate IDs, and ending with an ID of the form aq_fb where $q_f \in F$. With respect to a given $\langle M, w\rangle$ pair, it is possible to generate an LBA $LB_{\langle M,w\rangle}$ that accepts a string s exactly when s is a sequence of IDs representing an accepting computational history of M running on w.[2] All $LB_{\langle M,w\rangle}$ need to do is this: check that the first ID is q_0w; check that the $i+1$st ID follows from the ith ID through a legal transition rule of the Turing machine M; and check that the final ID is of the form aq_fb. Hence, if the emptiness of an LBA's language were decidable through a decider D_{E_LBA}, one could apply it to $LB_{\langle M,w\rangle}$.

> By virtue of its design, $LB_{\langle M,w\rangle}$ has an empty language exactly when M does not accept w.

Therefore, the decision of D_{E_LBA} would be tantamount to whether or not M accepts w — a known undecidable problem; and hence, D_{E_LBA} cannot exist.

17.3.3 Undecidability of PDA language universality

The question of whether an NPDA over alphabet Γ has a universal language (language equal to Γ^*) is undecidable. The proof proceeds almost exactly like the proof that D_{E_LBA} cannot exist.

• We will define an NPDA $P_{\langle M,w\rangle}$ (created with respect to Turing machine M and input string w), such that

[1] If the input to an LBA is ε, ¢ and \$ will lie in adjacent cells.

[2] It is possible to separate these IDs using some fixed separator character that is not in the original Γ.

- The language of $P_{\langle M,w\rangle}$ is Γ^* if M does not accept w.
- The language of $P_{\langle M,w\rangle}$ is $\Gamma^* \setminus \{s\}$ if M accepts w through an accepting computational history s.

If we manage to define such an NPDA, *then* simply feeding it to a claimed decider for universality will allow us to solve the acceptance problem of Turing machines, which is known to be undecidable (Section 16.2.3).

• Here is how $P_{\langle M,w\rangle}$ is designed:

- $P_{\langle M,w\rangle}$ is designed to examine the computation history of M running on w. In other words, what is fed to $P_{\langle M,w\rangle}$ is a sequence of instantaneous descriptions (ID) of some Turing machine M (for reasons to be made clear soon, we require odd-numbered IDs to be reversed in the input).
- If what is fed to $P_{\langle M,w\rangle}$ is an *accepting* computational history of M on w, then $P_{\langle M,w\rangle}$ rejects the input.
- If there is some i such that ID $i{+}1$ does *not* follow from ID i through a rule of M, then the given sequence of IDs is *not* an accepting computational history of M on w; in this case, $P_{\langle M,w\rangle}$ accepts.

From this design, it is clear that if M does not accept w, then $P_{\langle M,w\rangle}$ has a universal language. This is because *no* string will be an accepting computational history in this situation! On the other hand, if M accepts w, $P_{\langle M,w\rangle}$'s language will precisely miss the accepting computational history of M on w.

• Now, all we need to present is *how* $P_{\langle M,w\rangle}$ can do the said checks. This is easy:

- An NPDA can be made to nondeterministically pick the ith ID on the tape; it nondeterministically decides to either move over an ID or actually read and stack that ID.
- Once the ith ID has been picked and stacked, the $i+1$st ID can be compared against it by popping the ith ID from the stack each time one character of the $i+1$st ID is read.
- The only twist is that the NPDA will have to detect how ID i changed over to ID $i+1$. Fortunately, this comparison can be done around the head of the TM in the ith ID and the head of the TM in the $i+1$st ID. This much (finite) information can be recorded within the finite-state control of the NPDA.

17.3.4 Post's correspondence problem (PCP)

At first glance, a PCP *instance* is a simple puzzle about finite sequences[3] of pairs of strings of the form

$$\langle 01, 1\rangle\langle 01, \varepsilon\rangle\langle 01, 0\rangle\langle 1, 101\rangle.$$

It is customary to think of the above as "tiles" (or "dominoes"), with each tile at the respective index portrayed thus:

```
Index  0    1    2    3

Tile  [01] [01] [01] [  1]
      [ 1] [  ] [ 0] [101]
```

The question is: is there an arrangement of one or more of the above tiles, with repetitions allowed, so that the top and bottom rows read the same? Here is such an arrangement:

```
[  1] [01] [01] [1  ] [01] --> This row reads 10101101
[101] [ 0] [ 1] [101] [  ] --> This row reads 10101101
```

In obtaining this solution, the tiles were picked, *with repetition*, according to the sequence given by the indices `3,2,0,3,1`:

```
Index  3     2    0    3     1    --> Solution is
                                       3,2,0,3,1
       [  1] [01] [01] [1  ] [01] --> reads 10101101
       [101] [ 0] [ 1] [101] [  ] --> reads 10101101
```

Given a PCP instance S of length N ($N = 4$ in our example), a *solution* is a sequence of numbers $i_1, i_2, \ldots, i_k$ where $k \geq 1$ and each $i_j \in \{0 \ldots N-1\}$ for $j \in \{1 \ldots k\}$ such that $S[i_1]S[i_2]\ldots S[i_k]$ has the property of the top and bottom rows reading the same. By the term "solution" we will mean either the above sequence of integers or a sequential arrangement of the corresponding tiles.

Note that `3,2,0` is another solution, as is `3,1`. The solution `3,1` is:

```
Index  3     1    --> Solution is 3,1
       [  1] [01] --> reads 101
       [101] [  ] --> reads 101
```

[3] Recall from Chapter 8 that sequences and strings are synonymous terms.

17.3.5 PCP is undecidable

The PCP is a fascinating undecidable problem! Since its presentation by Emil L. Post in 1946, scores of theoretical and practical problems have been shown to be undecidable by reduction *from* the PCP.[4] A partial list includes the following problems:

- Undecidability of the ambiguity of a given CFG (see Exercise 17.2).
- Undecidability of aliasing in programming languages [101].
- Undecidability of the validity of an arbitrary sentence of first-order logic (see Section 18.2.3 for a proof).

The impressive diversity of these problems indicates the commonality possessed by this variety of problems that Post's correspondence problems embody.
The main undecidability result pertaining to PCPs can be phrased as follows. Given any alphabet Σ such that $|\Sigma| > 1$, consider the *tile alphabet* $\mathcal{T} \subseteq \Sigma^* \times \Sigma^*$. Now consider the language

$$PCP = \{S \mid S \text{ is a finite sequence over } \mathcal{T} \text{ that has a solution}\}.$$

Theorem 17.2. PCP is undecidable (a proof-sketch is provided in Section 17.3.6).

In [125], PCP is studied at a detailed level, and a software tool `PCPSolver` is made available to experiment with the PCP. The following terminology is first defined:

- A PCP instance is a member of the language PCP.
- If any member of $\mathcal{T}$ is of the form $\langle w, w \rangle$, then that PCP instance is *trivial* (has a trivial solution).
- The number of pairs in a PCP instance is its *size*. The length of the longest string in either position of a pair ("upper or lower string of a tile") in a PCP instance is the *width* of the instance.
- An *optimal* solution is the shortest solution sequence. The example given on page 315 has an optimal solution of length 2.

With respect to the above definitions, here are some *fascinating* results cited in [125], that reveal the depth of this simple problem:

- Bounded PCP is NP-complete (finding solutions of length less than an a priori given constant $K \in Nat$). Basically, checking whether solutions below a given length is decidable, but has, in all likelihood, exponential running time (see Chapter 19 for an in-depth discussion of NP-completeness).

[4] PCP is taken as the existing undecidable problem, and a mapping reduction to a new problem P is found, thus showing P to be undecidable.

- PCP instances of size 2 are decidable, while PCP instances of size 7 are undecidable (note: no restriction on the width is being imposed). Currently, decidability of sizes 3 through 6 are unknown.

Here are more unusual results that pertain to the shortest solutions that exist for two innocent looking PCP instances:

- The PCP instance below has an optimal solution of length 206:

```
[1000] [01] [1  ] [00 ]
[   0] [ 0] [101] [001]
```

- The PCP instance below has two optimal solutions of length 75:

```
[100] [0  ] [1]
[  1] [100] [0]
```

These discussions help build our intuitions towards the proof we are about to sketch:

> the undecidability of PCP indicates the *inherent inability to bound search while solving a general PCP instance.*

17.3.6 Proof sketch of the undecidability of PCP

The basic idea behind the proof of undecidability of PCP is to use A_{TM} and the computation history method. In particular,

- Given a Turing machine M, we systematically go through the transition function δ of M as well as the elements of its tape alphabet, Γ, and generate a finite set of tiles, $Tiles_M$.
- Now we turn around and ask for an input string w that is in the initial tape of M. Then, with respect to M and w, we generate one additional tile, $tile_{Mw}$. We define $Tiles = Tiles_M \cup \{tile_{Mw}\}$.
- We arrange it so that any solution to $Tiles$ must begin with $tile_{Mw}$. This is achieved by putting special additional characters around the top and bottom rows of each tile, as will soon be detailed.
- We then prove that $Tiles$ has one solution, namely $Soln_{Tiles}$, exactly when M accepts w. If M does not accept w, $Tiles$ will have no solutions, by construction. Furthermore, $Soln_{Tiles}$ would end up being a sequence $t_1, t_2, \ldots, t_k$ such that when the tiles are lined up according to this solution, the top and bottom rows of the tiles would, essentially,[5] be the accepting computation history of M on w.

[5] We say "essentially" because there would be extraneous characters introduced to "align" various tiles. In addition, at the very end of the solution sequence, the

- Hence, given a solver for PCP, we can obtain a solver for A_{TM}.

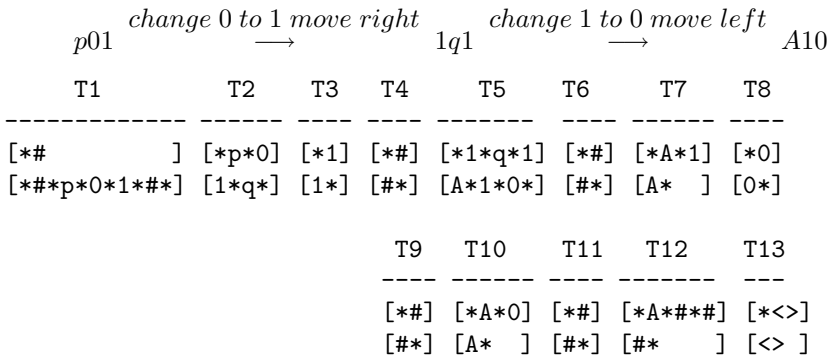

Fig. 17.2. An accepting computation history and its encoding as per the PCP encoding rules

The crux of achieving the above reduction is to generate each set of tiles carefully; here is how we proceed to generate the members of $Tiles$.[6] Here, the following notations are used: if u is the string of characters $u_1u_2 \ldots u_n$,

- $*u = *u_1 * u_2 \ldots * u_n$,
- $u* = u_1 * u_2 * \ldots u_n*$, and
- $*u* = *u_1 * u_2 * \ldots u_n*$.

In Figure 17.2, we illustrate the encoding ideas behind the PCP undecidability proof through the example of a Turing machine starting from state p with its tape containing string 0100 — *i.e.*, ID $p01$. This Turing machine first moves one step right to ID $1q1$, and in the process changes the 0 it was initially facing to a 1, as shown in Figure 17.2. Then the Turing machine moves one step left, and in the process changes the 1 it was facing to a 0, as also shown in Figure 17.2. At this point, it enters the accepting state A, as shown by the ID of $A10$ attained by this Turing machine, and hence halts. The general rules below are illustrated on specific tiles mentioned by the annotation "**Tn**" below:

top and bottom rows will shrink from being an accepting ID to an ID that simply contains q_a. Ignoring these characters, we would have the computational history of M on w on the top and bottom.

[6] We base our explanations quite heavily on those provided in [111].

$[*\#]$

T1: $tile_{Mw} = [*\#q_0w\#*]$. As said earlier, any solution to the PCP will start with this tile. This is ensured, as we shall soon see, by having a $*$ begin the top and bottom row of this tile.

$[*\diamond]$

T13: $Tiles_M$ must include $[\diamond]$. This tile will end any solution to the PCP. The extra $*$ at the top will, as we shall soon see, supply the last needed star in a run of tiles. The $\diamond$ at the top and bottom forces this to be the last tile.

$[*a]$

T3,T4,T6,T8,T9,T11: For every $a \in \Gamma$, $[a*]$ is a tile in $Tiles_M$. In $[*\#]$
addition, include $[\#*]$ as a tile in $Tiles_M$.

$[*qa]$

T2: For every move $\delta(q, a) = (r, b, R)$, $[br*]$ is a tile in $Tiles_M$. Notice that this pattern in a 2×2 window captures the Turing machine head, changing the 'a' character of the ID qa into a b, and moving right, attaining state r in the process.

$[*cqa]$

T5: For every move $\delta(q, a) = (r, b, L)$ and for every $c \in \Gamma$, $[rcb*]$ is a tile in $Tiles_M$. This pattern in a 3×2 window captures the tape head moving left.

$[*aq_a]$ $[*q_aa]$

T10: For every $a \in \Gamma$, $[q_a*]$ and $[q_a*]$ are tiles in $Tiles_M$. These tiles help shrink the top and bottom rows of the solution sequence from being the accepting ID to being the ID q_a.

$[*q_a\#\#]$

T12: For $q_a \in F$, $[\#*]$ is in $Tiles_M$. This tile helps finish off the accepting computation history that will form on the top row of a solution sequence.

The reader may verify that the top and bottom rows in Figure 17.2 essentially have the accepting computation history for M on w. Ignoring `[`, `]`, and `*`, what we have is

p01 # 1q1 # A10 # A0 # A ## $\diamond$.

The crux of the PCP proof was that A_{TM} can be decided if and only if the PCP instance generated by these tile generation rules *can be decided to possess a solution.*

Chapter Summary

This chapter discussed Rice's Theorem, the computation history method, and Post's correspondence problem. We took a semi-formal approach, but highlighting many details and intuitions often lost in highly theoretical presentations of these ideas. The basic techniques are all quite simple, and boil down to the undecidability of the acceptance problem, A_{TM}.

Exercises

17.1. Calculate N, referred to in Section 17.3.1, in terms of $|Q|$ and $|\Gamma|$.

17.2. Show that it is undecidable whether an arbitrary CFG is ambiguous. *Hint:* Let

$$A = w_1, w_2, \ldots, w_n$$

and

$$B = x_1, x_2, \ldots, x_n$$

be two lists of words over a finite alphabet Σ. Let $a_1, a_2, \ldots, a_n$ be symbols that do not appear in any of the w_i or x_i. Let G be a CFG

$$(\{S, S_A, S_B\},\ \Sigma \cup \{a_1, \ldots, a_n\},\ P,\ S),$$

where P contains the productions

$S \to S_A$,
$S \to S_B$,
For $1 \leq i \leq n$, $S_A \to w_i S_A a_i$,
For $1 \leq i \leq n$, $S_A \to w_i a_i$,
For $1 \leq i \leq n$, $S_B \to x_i S_B a_i$, and
For $1 \leq i \leq n$, $S_B \to x_i a_i$.

Now, argue that G is ambiguous if and only if the PCP instance (A, B) has a solution (thus, we may view the process of going from (A, B) to G as a mapping reduction).

17.3. Show that the unary PCP problem — PCP over a singleton alphabet ($|\Sigma| = 1$) — is decidable.

17.4. Modify the Turing machine for which Figure 17.2 is drawn, as follows: in state q, when faced with a 0, it changes 0 to a 1 and moves right, and rejects (gets stuck) in state R. Now go through the entire PCP "tile construction" exercise and show that the PCP instance that emerges out of starting this Turing machine on input string 00 in state p has no solution.

17.5. Someone provides this "proof:" There are PDAs that recognize the language $L = \{0^n 1^n \mid n \geq 0\}$. This PDA erects a non-trivial partitioning of $\{0,1\}^*$. Both the partitions are recursively enumerable. Hence, L is the language of infinitely many Turing machines M_L^i, for i ranging over some infinite index set. The same is true for $\overline{L}$ and $M_{\overline{L}}^j$. However this is a non-trivial partitioning of the space of Turing machines. Hence, L is undecidable.

Describe the main flaw in such a "proof."