1 Introduction

It is easy to argue that there must be non recursive, and even non recursively enumerable languages. A language is a subset of Σ^* and so, provided that Σ has at least two elements, there are uncountably many languages. But a TM has a finite description, so there are only countably many TMs. That is, in some sense, almost every language is not recursively enumerable!

However, coming up with an example, particularly one of practical significance or interest, is not so easy. The problem is this: how do you go about describing a language without implicitly specifying a method for determining whether or not strings belong to it?

The topic of this lecture is the famous result that the language HALT, which we know to be recursively enumerable (because it is the language accepted by a universal TM) is *not* recursive. From this it follows immediately that the complement of HALT cannot even be recursively enumerable, as we have shown that if a language and its complement are both recursively enumerable, then the language is recursive.

There are various 'practical' interpretations of this result, of which the most basic is that it cannot be possible to write a program that will analyze source code to test for the possibility of infinite loops (and which guarantees a correct result). But in fact as we will see that it actually means that it's impossible to do virtually anything of interest in terms of analysing the behaviour of programs.

2 The problem

We want to show that:

$$\text{HALT} = \{R(M)w \mid M \text{ halts on } w\}$$

is not recursive. How can we do this? It seems like we need to examine every TM that halts on all inputs (but how do we know?) and determine that its accepted language is not HALT.

Look at that last sentence – the problem we are talking about is recognising TMs and inputs that halt, and the mechanisms we are supposed to have available are TMs that

do halt. There's a hint of self-reference there, and where there is self-reference there is a possibility of contradiction using a form of diagonalisation.

The most famous argument of this type is known as *Russell's paradox* (which is really based on the *liar paradox*) and was used to show that the foundations of so called naive set theory laid down by Cantor were inconsistent. In Cantor's theory any collection of objects was a set – in particular we could think of Ω, the set of all sets. Also, any subset of a set defined by a *predicate* (that is, rule) is a set. So, Russell said, what about:

$$B = \{x \in \Omega \mid x \notin x\},$$

the set of all sets that are not elements of themselves. Is B an element of B or not? Suppose so. Then it must satisfy the defining condition for B that is, $B \notin B$, so not. On the other hand if $B \notin B$ then it does satisfy the defining condition, so $B \in B$. Either way we have a contradiction, and thus a paradox.

In our case we will begin by imagining that we had a TM, H, which halts on all input and accepts HALT. Using a similar argument (after building in some self-reference) we will see that this leads to a contradiction. So, rather than a paradox, we simply have a proof that no such TM can exist.

3 Some plumbing

We begin with a TM, H, assumed to solve the halting problem. To be perfectly clear this means that H halts on all inputs and accepts the language:

$$\text{HALT} = \{R(M)w \mid M \text{ halts on } w\}.$$

We think of H as a black box and modify it in various ways.

Our first modification is to produce an "anti-halting" machine, AH. The operation of AH on input $R(M)w$ is as follows: first H is run on $R(M)w$, if it halts and rejects (i.e., if H certifies that M would not halt on w), then AH simply halts; while if it halts and accepts (certifying that M would halt on w) then AH enters an infinite loop.

So, if M would halt on input w then AH enters an infinite loop on input $R(M)w$ while, if M would loop on input w, AH halts on input $R(M)w$.

Now we do the diagonalization step – we build a TM, D, that operates on input $R(M)$ as follows: first it duplicates the input producing $R(M)R(M)$, and then it runs AH on that.

So, if M would halt on input $R(M)$ then D loops on input $R(M)$, while if M would loop on input $R(M)$ then D halts on input $R(M)$.

4 The punch line

The machine D is a TM. So, it has a representation $R(D)$. What does D do on input $R(D)$? To find out, change M to D in the last sentence of the previous section.

If D would halt on input $R(D)$ then D loops on input $R(D)$, while if D would loop on input $R(D)$ then D halts on input $R(D)$.

That looks like the contradiction we need – and our only assumption in this whole argument (beyond the fact that we can do the 'plumbing' which is easily justified) is that H existed. So:

Theorem 4.1. *The language* HALT *is recursively enumerable, but not recursive.*

5 Turing reducibility

We now know one language, HALT which is recursively enumerable but not recursive, and hence another, the complement of HALT which is not even recursively enumerable. How can we produce other 'interesting' examples of recursively enumerable but not recursive languages without having to go through the bother of a diagonalization argument each time?

The key idea is to think of *decision procedures* as black boxes. By a decision procedure for a language, L, we mean some mechanism that is able to answer reliably the question: is $w \in L$?

Then to show that some language L is not recursive it would suffice to show that we could mechanically transform instances of HALT into instances of L of the same status (i.e. things in HALT get mapped to things in L, and things not in HALT to things not

in L). The reason this works is that such a transformation, together with a mechanical decision procedure for L would give us one for HALT and we know that can't happen.

Given two languages L and K (or their associated decision problems), we say that L *is Turing reducible to* K, and write $L \xrightarrow{TR} K$, if there is a TM, M which always halts, and on input w leaves some other word $r(w)$ on the tape in such a way that:

$$w \in L \quad \text{if and only if} \quad r(w) \in K.$$

If L is an undecidable (i.e., non recursive) language and $L \xrightarrow{TR} K$, then K must also be undecidable (otherwise we could run the reduction, and then apply the decision procedure for K). So, in particular a tool for showing that K is undecidable is to show that HALT $\xrightarrow{TR} K$.

6 Examples

Consider the language:

$$\text{BLANK-HALT} = \{R(M) \mid M \text{ halts on a blank input tape}\}.$$

Given an instance of HALT, how do we transform it into a corresponding instance of BLANK-HALT? Well, an instance of HALT is really just a pair (M, w) consisting of a TM M and in input word w (and the associated question "does M halt on w"?) Since w is just a finite sequence of letters it's easy to construct a new TM, M' which, on a blank input tape does the following: first it writes w on the tape, then it resets the read-write head to the leftmost square, then it runs M. Of course M' halts on a blank tape if and only if M halts on w, so we have succeeded in reducing HALT to BLANK-HALT, thus proving BLANK-HALT to be undecidable.

What about:

$$\text{ALWAYS-HALT} = \{R(M) \mid M \text{ halts on all inputs}\}.$$

We just use the same idea. Given an instance, (M, w) of HALT construct a new TM M' which: erases anything on the tape, writes w, resets the read-write head, and then runs

M. Since effectively M' ignores its input and just simulates M running on w, we see that $M' \in$ ALWAYS-HALT if and only if $(M, w) \in$ Halt, so ALWAYS-HALT is also undecidable.

These two examples can be easily generalized to show that all questions of the form "Does the language accepted by M have some non-trivial property" are undecidable (this is known as Rice's theorem).

7 The hierarchy of languages

Roughly speaking what we have accomplished to this point is to construct a hierarchy of languages in terms of the complexity of the machinery or decision procedures required to recognise them. In order from least to most complex:

Finite languages Self-explanatory. To prove that a language is finite, either list it explicitly, or provide a bound on the length of the strings it contains. To prove it is not finite, show that for any length it contains a string of at least that length.

Regular languages These are recognised by finite state automata. To prove that a language is regular either design such an automaton that accepts it, or provide a regular expression or regular grammar for it. To prove that a language is not regular show that it violates the Myhill-Nerode theorem or the pumping lemma for regular languages. Regular languages have very strong closure properties.

Context-free languages These are recognised by pushdown automata. To prove that a language is context-free either design such an automaton that accepts it, or provide a context-free grammar for it. To prove that a language is not context-free show that it violates the pumping lemma for context-free languages. Context-free languages have reasonably strong closure properties, but notably are not closed under intersection or complement.

Recursive languages These are recognised by Turing machines that halt on all inputs. To prove that a language is recursive, design such a TM (or any other mechanical procedure with guaranteed termination – taking advantage of the Church-Turing thesis). Or, design one TM that accepts it (but may loop on non-examples) and one

that accepts its complement (ditto). To prove that a language is not recursive give a Turing reduction of HALT or some other non-recursive language to it. Since we can compose TM's in lots of different ways, the family of recursive languages has very strong closure properties (so strong that we didn't even bother to mention them until now).

Recursively enumerable languages These are accepted by Turing machines (which need not halt on negative instances). To prove that a language is recursively enumerable, design such a TM (or any other mechanical procedure). To prove that a language is not recursively enumerable, show that its complement is recursively enumerable but not recursive[1]. The family of recursively enumerable languages has reasonable closure properties (but obviously not closure under complementation).

[1]This is by no means a necessary condition – lots of non-RE languages have non-RE complements, but it's the only mechanism we currently have.

8 Exercises

Determine which of the following decision problems are undecidable and which are decidable. Remember the key to proving that "Problem P is undecidable" is to provide an argument that "If we had a black box that solved P, then we could solve X" where X is some problem already known to be undecidable (most frequently the halting problem). That is: if problem X is reducible to problem P, and X is undecidable, then P must be undecidable.

(Note that a *transition* is any single "read-write-move" event, even if it does not involve a change of state or tape)

1. SOMETIMES HALT
   Instance: A Turing machine M.
   Problem: Does M halt on some input?

2. EVEN HALT
   Instance: A Turing machine M and an input word w.
   Problem: Does M halt on input w after an even number of transitions?

3. BOUNDED HALT
   Instance: A Turing machine M, an input word w and an integer n.
   Problem: Does M halt on input w without making more than n transitions?

4. (*) NO WRITE
   Instance: A Turing machine M and an input word w.
   Problem: Does M halt on input w without making any change to the tape (i.e. all transitions only move the tape head)?

5. (Review) Show that the language HALT consisting of all strings $R(M)w$ such that M halts on input w *is* recursively enumerable. What can be said about its complement?

6. Fill in the missing code indicated in the comment below (well, don't really)

```
import java.util.Scanner;

public class D{

   public static void main(String[] args) {

      StringBuilder s  = new StringBuilder();
      Scanner in = new Scanner(System.in);
      while(in.hasNextLine()) s.append(in.nextLine() + "\n");
      String source = s.toString();

      if (halt(source, source)) {
         while (true) {
            int x = 2 + 2;
         }
      } else {
         System.out.println("D exits normally");
         System.exit(0);
      }
   }

   public static boolean halt(String program, String input) {

      // TODO: Replace stub below with method
      // Shouldn't be too hard!

      return false;

   }

}
```

7. What happens if we run:

```
> java D < D.java
```