Theory of Formal Languages (Decidability and Computability)

Lecture 13: April 06, 2021

Prof. K.R. Chowdhary *: Professor of CS*

13.1 Introduction

A language acceptable by Turing machine is called *Recursively Enumerable* (RE), which means that the set of strings in the language accepted by the Turing Machine can be enumerated. Recursively enumerable language is also called *partially decidable* or *Turing-recognizable*. It is *type-0* language in the Chomsky hierarchy. The RE languages are always countably infinite. The class of RE languages has a broad coverage of languages, and they include some languages, which cannot be defined by a mechanical algorithm. TMs will fail to halt on some input not in these languages. If w is a string in RE language then the TM will eventually halt on w.

There exists three equivalent definitions for the concept of a RE language .

Definition 13.1 RE Language. *An RE language is a formal language for which there exists a Turing machine (or other computable function) that will halt and accept when presented with any string in the language as input. But may either halt and reject or loop forever when presented with a string not in the language.*

Contrast this to **recursive** languages, which require that the Turing machine halts in all cases.

All regular, context-free, context-sensitive and recursive languages are recursively enumerable. The RE languages together with their complement *co-RE*, form the basis for the *arithmetical hierarchy*.

A Turing machine M that is still running on some input, we can never tell whether M will ultimately accept if it is allowed to run long enough or M will run forever. Therefore, it is appropriate to separate the subclass of RE languages accepted by at least one TM that halts on all inputs. However, halting may or may not be preceded by acceptance.

We start by asking a grand question: "What problems are solvable?". However, we find it a very general and vague question, which is required to be precise. Few examples of problems are:

1. Given any natural number $n \in \mathbb{N}$, determine if n is prime.

2. Given a formula θ in the language of propositional logic, find out whether θ is a satisfiable?

3. Given a formula θ in the language of propositional logic, find out whether θ is a tautology?

4. Given a formula ψ in predicate logic, find out whether ψ is provable using the natural deduction. To be precise, given the collection of predicate formulas Γ can we deduce $\Gamma \vdash \psi$?

5. Given a polynomial $P(x_1, ..., x_n)$ of n variables with integer coefficients, determine whether $P(x_1, ..., x_n) = 0$ has integer solutions?

6. Given a C program P, determine whether this program will ultimately halt for some definite given input, or it will loop forever?

7. Given a C program Q, determine whether Q will ultimately halt on every input?

8. Given a natural number $n > 0$ and $n \in \mathbb{N}$, determine whether there exists n consecutive 3's in the decimal expansion of π?

All these questions further raise interesting questions, which we will clarify in our future discussions. However, one thing is clear that they are all *decision based* problems, which requires some decision is to be made in the form of Yes/No.

For each of the above problems, there are instances that we can solve easily. For the problem 5 it is easy to find the root. For the predicate logic based problem 4 we can use deduction to find out the solution. For the problem 8, we can generate the decimal value of π, and when n consecutive 3's come, the process can be terminated. All these are the given instances of the problems, which can be easily solved. However, we are interested for a general solution to these problems, so that it works for all the instances of the problem.

In the problem 1, a program can received input, and a fixed algorithm can check if it is prime, the prints Yes or No depending on the result of that program. This is a completely mechanical procedure. In the problem 2, one can construct truth-table, and if there is True in any row of the expression θ, it is satisfied else not. Similar is the case in the problem 3, where all the values in the column of θ must be True, for θ to be tautology. Hence, both these problems are decidable, as we can write a general programs to solve these problems, which can work for all the instances of the problems.

Hence, we can say that a problem is *decidable* or *solvable*, if there exists an effective procedure which gives the correct outcome for each of the possible instances of the problem.

Undecidable Problems. By contrast, like above cases, no recipes are available for predicate logic. We can try for a natural deduction proof for the formula ψ, but after trying a lot if we do not succeed, we are not sure whether the lack of success is because ψ is not provable or because natural deduction just is not the strong point. Note that it may not be the case that we do not have good algorithm in hand, but there may not be existing any algorithm. In the latter case, the problem's solution is undecidable.

Consider again the decimal expansion of π. The best way we can do it is to run the concerning algorithm, till we get n number of 3's, for a given value of n. Up to this point

the algorithm seems analogous to that of predicate logic problem, where we try to generate or deduce ψ. However, there is an important difference between two problems. In case of ψ the decision procedure may be impossible, as a matter of fact, while in case of π, we simply do not have a procedure. Such problems are called *undecidable problems*.

Semi-decidable Problems. There is another interesting observation about some some problems. Even though some of these problems are unsolvable, it may still be easy to decide whether a given answer to the problem is correct. For example, finding factors of a polynomial problem may be difficult, but checking whether the given sequence $n_1, ..., n_k$ is a solution, is straight forward calculation. We can simply multiply the given factors, and if we get original expression, the given factors make the correct solution. Similarly, we might not been able to come up with a natural deductive proof for a given formula ψ, but if we are given the solution, then it is not hard to verify whether the solution is indeed correct.

This suggests following strategy for the problem 4: start generating proofs in a systematic way, and make sure that we generate all the proofs eventually, and for each we check that it is a valid proof for the formula ψ. Clearly, if a proof exists for ψ, we will be able to generate the proof. But, if really there does not exist a proof, we will be trying to generate more and more proofs, and we would never know if at all there is a proof ahead. Such problems are some times called *semi-decidable* problems. The example is,

$$z^n = x^n + y^n, \tag{13.1}$$

where x, y, z, n are all integers. It is well known that for $n = 2$ there are many solutions, but, it is not known whether there exists any x, y, z for which this equation holds true for $n \geq 3$!

A C language program P, given an input (problem 6), whether ultimately halts or not is semi-decidable: one can simply run the program P and wait for its response. If it halts, then we have found the solution, but if it does not halt, we cannot say, whether it is going to halt ultimately or may run forever !

The problem 7 is not for one input, but infinitely may inputs, for which we do not have time to test. May be it halted for one billion different inputs, we are not sure for the next input. Alternatively, the program may not require any input, in that case, say, the program has run continuously for one hundred year and not halted, who knows it may halt in the 22nd century if continuously run !

Hence, the problems 6 and 7 are unsolvable problems. However, for the problem 8 it is yet unknown as to which problem it belongs.

13.2 Decision Problems

About the answer of "What problems are solvable", we will proceed in a systematic manner. However, we take a sudden jump, and consider the problem(s) of this form: Given a function,

$$f : \mathbb{N}^k \to \mathbb{N}, \tag{13.2}$$

compute $f(x)$ for the input argument x. The reason for straight way choosing this problem are many: the functions on natural numbers or equivalently, on real numbers, are well-defined, and are the natural objects of study of mathematics. There is also historical reasons for study of these functions, one of them is *Peano Arithmetic*. Apart from these, the *recursion theory* is study of functions over natural numbers. Once we have understood the functions on natural numbers to be computable, we can use this understanding to address other problems that are not based on recursion theory. The concepts and study that we develop for study of computable functions can be applied in other settings, through analogy. Other approach, which is more important: many other problems can often be encoded into natural numbers in such a way that we can use our knowledge of latter to reason about the new problems.

13.2.1 Effective Procedure

We have chosen to study the functions over natural numbers is due to their level of difficulty, i.e., simplicity. But, we still need to define what we mean by "solution"? In fact, what we are after is some "effective procedure", which takes as input possible instances of the problem, and produces the correct answer. This procedure should be *complete*, *deterministic*, and should consist of simple steps, which can be done by hand.

The kind of steps may be, e.g., adding 1 to to a given number, given two numbers discard the first and select the second, testing whether a given number is zero, etc. In other words, we are looking, for the explanation of effective procedure in terms of elementary mechanical computation steps, each of that involves the manipulation of finite amount of data.

Deterministic means each step should have at most one unambiguous outcome, and *complete* means, it should work for the entire domain of input and not selective one.

A function that can be computed using such a procedure will be called *effectively computable* functions. There are two approaches to handle this effective procedure. First approach is mathematical in nature, which starts with few simple and well-understood objects. These objects are are some very simple functions which are intuitively computable. Using these functions as basic building blocks, we gradually build more complicated objects.

For the above, we build new functions is using the simpler version of the same functions, called *recursion*. The newly produced functions are called *recursive functions*. Thus, the answer to the questions: "Which functions are effectively computable", is recursive functions.

A simple example of decision function is an arbitrary polynomial $P(x_1, x_2, \ldots, x_n)$, with non-negative integral coefficients, which are not identically zero. If the x's are assigned arbitrary positive integral values, for example, in the Arabic notation, the algorithms comprising of addition and multiplication in that notation enable us to calculate the corresponding positive integral value of the polynomials. That is, $P(x_1, x_2, \ldots, x_n)$ is an *effectively calculable* function of positive integers. The importance of the technical concept – *recursive function* derives from the overwhelming evidence that it is coextensive with the intuitive concept effectively calculable function.

The second approach is practical and may be regarded as belonging to the field of computer science. This approach is concerned with theoretical framework for computation, and study of what can be computed in the framework, and models the actions of an idealized human computing agent, so that we can systematically analyze the possibilities and limitations of

the same. Among such several frameworks, the Turing machine is the best-known framework. The purpose of this framework is to study computations in an idealized setting, i.e., there is no limitation of time and memory size, which is sufficiently general but technically simple, so that the study is not overly complicated.

13.2.2 Models for computation

Apart from the Turing machine, there is another machine – the *register machine* that can model the computations, using natural numbers. The recursive functions are exactly those which are computable by a register machine. However, the same computation can be modeled using Turing machine, and can also be defined using Lambda-calculus. But, the outcome remains the same. The theory of recursive functions derives its significance from the fact that recursive functions can be *effectively enumerated*. One can encode each of that as natural number such that the code can effectively retrieve the function. This means, in Recursive theory, the natural numbers appears in two guises: once as code of recursive functions, and other as arguments for these functions.

There are however, are functions and sets that are not recursive. The sets whose membership can be effectively decided, are the *recursively enumerable* (RE) sets. A set is RE if there is an effective procedure for generating (enumerating) its elements, like in the problems 4, 5, 6. Thus if a given element belong to an RE set we can run this procedure and it will eventually confirm that the element was in the set. But if the element is not in the set, the procedure will search forever and will never know the answer. For this reason, the RE sets are sometimes called *semi-decidable*, or *semi-recursive* sets.

A set of positive integers is said to be RE if there is a recursive function $f(x)$ of one positive integral variable whose values, for positive integral values of x, constitute the given set of positive integers. The sequence $f(1), f(2), f(3), \ldots$ is then said to be a *recursive enumeration* of the set. The corresponding intuitive concept is that of an effectively enumerable set of positive integers of positive integers. To prepare us in part for our intuitive approach, consider the following three examples of *recursively enumerable* sets of positive integers.

1. $1^2, 2^2, 3^2, \ldots,$
2. $1, 2, 2^{1+2}, 2^{1+2+2^{1+2}}, \ldots$
   (a) $1^2, 2^2, 3^2, \ldots$
   (b) $1^3, 2^3, 3^3, \ldots$
   (c) $1^4, 2^4, 3^4, \ldots$

In the first example, the set is produced by a recursive enumeration using the recursive function $f(x) = x^2$. In the second example, the set is generated in a linear sequence, each new element being effectively obtained from the elements previously generated, in this case by raising 2 to the power the sum of the preceding elements, e.g., in the third element, 2^{1+2} is obtained by raising the sum of previous elements $1 + 2$ over 2. This set is effectively enumerable, since given n, the nth element of the sequence can be found by regenerating the sequence through its first n elements.

In the third example with three series, we rather imagine the positive integers 1, 2, 3, ... generated in their natural order, and, as each positive integer n is generated, a corresponding process is set up which generates $n^2, n^3, n^4, \ldots,$ all these to be in the set. Actually,

the standard method for proving that an enumerable set of enumerable sets is enumerable yields an effective enumeration of the set. The series a, b, c here are enumeration sets and their union is also enumerated set [post44].

Several more examples can be given to convey the concept of a generated set, in the present instance of positive integers. Suffice it to say that each element of the set is at some time written down, and earmarked as belonging to the set, as a result of predetermined effective processes. It is understood that once an element is placed in the set, it stays there. Some times this is referred to as a generalization, which may be restated *every generated set of positive integers is recursively enumerable*. For comparison purposes this may be resolved into the two statements: (1). every generated set is effectively enumerable, and (2). every effectively enumerable set of positive integers is recursively enumerable. The first of these statements is applicable to generated sets of arbitrary symbolic expressions; their converses are immediately seen to be true. We shall find the above concept and generalization useful in our intuitive development.

But while we shall frequently say, explicitly or implicitly, "set so and so of positive integers is a generated, and hence recursively enumerable set," that is merely to mean "the set has intuitively been shown to be a generated set; it can indeed be proved to be recursively enumerable." Likewise, for other identifications of informal concepts with corresponding mathematically defined formal concepts.

13.2.3 Formulating Decision Problems

In the previous chapters we designed machines to detect patterns in strings, (for example palindromes, $a^n b^n c^n$, etc) and recognize the languages based on these patterns. Many higher level problems can be posed based on these pattern recognitions of strings and string manipulations. For example, we may be interested in following questions:

"Is a given natural number perfect square?"

"Is a given natural number prime?"

"Does a given graph has cycle?"

"Does the computation of TM halt before 25th transition?"

Each of these general questions describe a *decision problem*. A decision problem P is a set of *related questions* p_i, each of which has Yes/No answer. For example, to determine, if a natural number is a perfect square, requires answering following questions, and we carry on the process until we reach to that natural number[1].

p_0: Is 0 a perfect square?

p_1: Is 1 a perfect square?

[1]For example, to answer if 20 is perfect square, we generate the sequence of 1, 2, 3, .., and compare each number's square with 20, and carry on as long as the squared number is less than 20. After getting square of 4, when next square is 25, we note that it is greater than 20, hence we terminate the process and declare 20 as not perfect square. However, if any generated square matched with the given number, we would say it is success.

p_2: Is 2 a perfect square?

...

Each of the p_i is an instance of the problem P. The solution of a decision problem P is an algorithm that determines the answer of every question "Is $p_i \in P$"? A *decision problem* is *decidable*, if it has a solution.

Definition 13.2 Decision Problem. *By the decision problem of a given set of positive integers we mean the problem of* effectively *determining for an arbitrarily given positive integer whether it is, or is not, in the set.*□

While, in a certain sense, the theory of recursively enumerable sets of positive integers is potentially as wide as the theory of general recursive functions. The decision problems for such sets constitute a very special class of decision problems. Since decision problem's solution requires an algorithm, an intuitive notion of algorithm is essential. Following are the properties of such algorithms.

- *Complete*: Correct answer should be given for every problem instance,
- *Mechanistic*: Consists of finite sequence of instructions, each can be carried out without requirement of insight, ingenuity, or guesswork, and
- *Deterministic*: With identical input, the identical computation is carried out.

Definition 13.3 Effective procedure. *A procedure having the properties of complete, Mechanistic, and deterministic, is called effective procedure.*□

A standard TM is an effective algorithm if it is, mechanistic, deterministic, and complete. However, it is complete only if, it halts on every input. Thus, TM can be used to solve decision problems. To solve a problem using TM, we need to transform the problem instances into strings, called, representation of the decision problem. A problem instance is answered by a TM, if TM accepts and halts, else rejects and halts.

The *Church-Turing* thesis for decision problem states that TM can be designed to solve the decision problem if it can be solvable by an *effective procedure*.

The Table 13.1 shows the general representation of problem P, in the form of its instances $p_1, ..., p_l$, with inputs to TM as $w_1, ..., w_l$, and for each instance the TM gives output Yes/No, when it completes the processing.

The Table 13.2 shows the representation of a problem: "acceptance of unary number strings" that represent even numbers.

In general, 1^i, such that i is odd, is an even number. The TMs for acceptance of odd numbers can be constructed for unary and binary numbers, as shown in Fig. 13.1(a) and (b). In case of input as a unary number, when TM reaches to blank symbol at the end of input, it will halt at state q_2 if the number is even. In case of binary number as input, which should be terminated by 0 for input to be an even number, it halts in q_2 (see Figure 13.1(b)).

Table 13.1: General format for problem instances

Problem instance	TM Input	Answer after the Turing machine computes
p_1	w_1	Yes/No
p_2	w_2	Yes/No
...	...	...
p_i	w_i	Yes/No

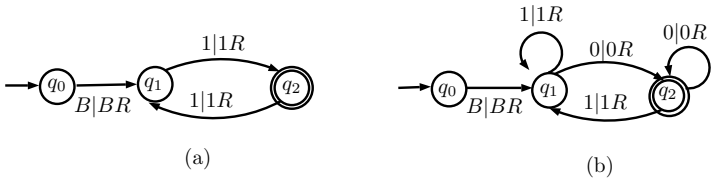

Figure 13.1: TM recognizes, (a) even unary numbers, (b) even binary numbers

Table 13.2: Problem instances with solutions

Problem instance	TM Input	Is number even?
0	1	Yes
1	11	No
2	111	Yes
3	1111	No
...	...	...