Theory of Formal Languages (Decidability Theory & Decidable Languages)

Lecture 12: Feb. 15, 2019

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

12.1 *Gödel*'s Incompleteness Theorem

Gödel's incompleteness theorems are two theorems of mathematical logic that demonstrate the inherent limitations of every formal axiomatic system containing basic arithmetic.These results, published by Kurt Godel in 1931, are important both in mathematical logic and in the philosophy of mathematics. The theorems are widely, but not universally, interpreted as showing that Hilbert's program to find a complete and consistent set of axioms for all mathematics is impossible [soloman06].

The *first* incompleteness theorem states that, "no consistent system of axioms whose theorems can be listed by an effective procedure (i.e., an algorithm) is capable of proving all truths about the arithmetic of the natural numbers. For any such formal system, there will always be statements about the natural numbers that are true, but that are unprovable within the system."

The *second* incompleteness theorem, an extension of the first, shows that the system cannot demonstrate its own consistency

Employing a diagonal argument, Godel's incompleteness theorems were the first of several closely related theorems on the limitations of formal systems. They were followed by Tarski's undefinability theorem on the formal undefinability of truth, Church's proof that Hilbert's Entscheidungsproblem (The Decision problem) is unsolvable, and Turing's theorem that there is no algorithm to solve the halting problem.

First incompleteness Theorem Godel's first incompleteness theorem first appeared as "Theorem VI" in Godel's 1931 paper "On Formally Undecidable Propositions of Principia Mathematica and Related Systems I". The hypotheses of the theorem were improved shortly thereafter by J. Barkley Rosser (1936) using Rosser's trick. The resulting theorem (incorporating Rosser's improvement) may be paraphrased in English as follows, where "formal system" includes the assumption that the system is effectively generated [].

As per the First Incompleteness Theorem, "any consistent formal system F within which a certain amount of elementary arithmetic can be carried out is incomplete; i.e., there are statements of the language of F which can neither be proved nor disproved in F."

The unprovable statement GF referred to by the theorem is often referred to as "the Godel sentence" for the system F. The proof constructs a particular Godel sentence for the system

F, but there are infinitely many statements in the language of the system that share the same properties, such as the conjunction of the Godel sentence and any logically valid sentence.

Each effectively generated system has its own Godel sentence. It is possible to define a larger system F' that contains the whole of F plus GF as an additional axiom. This will not result in a complete system, because Godel's theorem will also apply to F', and thus F' also cannot be complete. In this case, GF is indeed a theorem in F', because it is an axiom. Because GF states only that it is not provable in F, no contradiction is presented by its provability within F'. However, because the incompleteness theorem applies to F', there will be a new Godel statement GF' for F', showing that F' is also incomplete. GF' will differ from GF in that GF' will refer to F', rather than F.

12.1.1 Formal System and Completeness

The incompleteness theorems apply to formal systems that are of sufficient complexity to express the basic arithmetic of the natural numbers and which are complete, consistent, and effectively axiomatized, these concepts being detailed below. Particularly in the context of first-order logic, formal systems are also called formal theories. In general, a formal system is a deductive apparatus that consists of a particular set of axioms along with rules of symbolic manipulation (or rules of inference) that allow for the derivation of new theorems from the axioms. One example of such a system is first-order Peano arithmetic, a system in which all variables are intended to denote natural numbers. In other systems, such as set theory, only some sentences of the formal system express statements about the natural numbers. The incompleteness theorems are about formal provability within these systems, rather than about "provability" in an informal sense.

There are several properties that a formal system may have, including completeness, consistency, and the existence of an effective axiomatization. The incompleteness theorems show that systems which contain a sufficient amount of arithmetic cannot possess all three of these properties.

A set of axioms is (syntactically, or negation-) complete if, for any statement in the axioms' language, that statement or its negation is provable from the axioms. This is the notion relevant for Godel's first Incompleteness theorem. It is not to be confused with semantic completeness, which means that the set of axiom proves all the semantic tautologies of the given language. In his completeness theorem, Godel proved that first order logic is semantically complete. But it is not syntactically complete, since there are sentences expressible in the language of first order logic that can be neither proved nor disproved from the axioms of logic alone: for example, "the flower is pretty", can neither be proved nor unproved.

In a mere system of logic it would be absurd to expect syntactic completeness. But in a system of mathematics, thinkers such as Hilbert had believed that it is just a matter of time to find such an axiomatization that would allow to either prove or disprove (by proving its negation) each and every mathematical formula.

A formal system might be syntactically incomplete by design: such as logics generally are. Or it may be incomplete simply because not all the necessary axioms have been discovered or included. For example, Euclidean geometry without the parallel postulate is incomplete, because it is not possible to prove or disprove the parallel postulate from the remaining axioms. Similarly, the *theory of dense linear orders* is not complete, but becomes complete with an extra axiom stating that there are no endpoints in the order. The *continuum hypothesis* is a statement in the language of ZFC(Zermelo Fraenkel set theory) that is not provable within ZFC (Zermelo Fraenkel set theory), so ZFC is not complete. In this case, there is no obvious candidate for a new axiom that resolves the issue [].

The theory of first-order Peano arithmetic is consistent, has an infinite but recursively enumerable set of axioms, and can encode enough arithmetic for the hypotheses of the incompleteness theorem. Thus, by the first incompleteness theorem, Peano Arithmetic is not complete. The theorem gives an explicit example of a statement of arithmetic that is neither provable nor disprovable in Peano's arithmetics. Moreover, this statement is true in the usual model. Moreover, no effectively axiomatized, consistent extension of Peano arithmetic can be complete.

12.1.2 Effective Axiomatization and Consistency

A formal system is said to be effectively axiomatized (also called effectively generated) if its set of theorems is a recursively enumerable set. This means that there is a computer program that, in principle, could enumerate all the theorems of the system without listing any statements that are not theorems. Examples of effectively generated theories include *Peano arithmetic* and *ZermeloFraenkel set theory* (ZFC).

The theory known as true arithmetic consists of all true statements about the standard integers in the language of Peano arithmetic. This theory is consistent, and complete, and contains a sufficient amount of arithmetic. However it does not have a recursively enumerable set of axioms, and thus does not satisfy the hypotheses of the incompleteness theorems.

A set of axioms is (simply) consistent if there is no statement such that both the statement and its negation are provable from the axioms, and inconsistent otherwise. If one takes all statements in the language of Peano arithmetic as axioms, then this theory is complete, has a recursively enumerable set of axioms, and can describe addition and multiplication. However, it is not consistent.

Additional examples of inconsistent theories arise from the paradoxes that result when the axiom schema of unrestricted comprehension is assumed in set theory.

12.1.3 Syntactic form of the Godel sentence

The Godel sentence is designed to refer, indirectly, to itself. The sentence states that, when a particular sequence of steps is used to construct another sentence, that constructed sentence will not be provable in F. However, the sequence of steps is such that the constructed sentence turns out to be GF itself. In this way, the Godel sentence GF indirectly states its own unprovability within F.

To prove the first incompleteness theorem, Godel demonstrated that the notion of provability within a system could be expressed purely in terms of arithmetical functions that operate on Godel numbers of sentences of the system. Therefore, the system, which can prove certain facts about numbers, can also indirectly prove facts about its own statements, provided that it is effectively generated. Questions about the provability of statements within the system are represented as questions about the arithmetical properties of numbers themselves, which would be decidable by the system if it were complete.

Thus, although the Godel sentence refers indirectly to sentences of the system F, when read as an arithmetical statement the Godel sentence directly refers only to natural numbers. It asserts that no natural number has a particular property, where that property is given by a primitive recursive relation. As such, the Godel sentence can be written in the language of arithmetic with a simple syntactic form. In particular, it can be expressed as a formula in the language of arithmetic consisting of a number of leading universal quantifiers followed by a quantifier-free body.

12.2 Unsolvable problem of elementary number theory

There is a class of problems in elementary number theory which can be stated as: it is required to find an effectively calculable function $f(x_1, x_2, ..., x_n) = 2$, where arguments are positive variables, is necessary and sufficient condition for the truth of certain proposition of elementary number theory having $x_1, x_2, ..., x_n$ as free variables [alonzoch].

An example of such as problem is to find a means of determining of any given positive integer n, whether or not there exists positive integers x, y, z, such that $x^n + y^n = z^n$. This problem may be interpreted as, it is required to find an effectively calculable function f, such that $f(n)$ is equal to 2 if an only if there exists positive integers x, y, z, such that $x^n + y^n = z^n$. Clearly, the condition that the function f be effectively calculable is an essential part of the problem, since without it the problem becomes trivial [alonzoch].

12.3 Unsolvability

There are infinitely more problems than algorithms, and there are in fact infinitely more problems than machines. So building a machine that is more powerful than computer, would not help a bit. There would still be an infinite number of problems it would not be able to solve. In the next we list the steps to define such a problem.

Defining Problem-I We start by defining a problem: A problem is described by its *inputs* and *outputs*. The *primality* testing problem, for example, is defined as:

Inputs: A natural number x.

Output: True, if x is prime, False otherwise.

Following are steps to define it:

1. We can define a problem as a function from its inputs to output, $y = f(x)$, where x is input, y is output, and f is function (an algorithm). To make matters more simple, we limit ourselves to an input that is a natural number and to an output that is a digit $(0, ..., 9)$.

2. The next step is defining, what is an algorithm? We define the algorithm as a text of finite length. It is clear that all algorithms are texts but there are many texts that are not algorithms.

3. The next step is to show one-to-one function from the algorithms onto the natural numbers ($\mathbb{N}$). In simple words, we show how to transform every algorithm to a natural number and vice-versa. The transformation is simple: for example, assume that our alphabet has 1000 letters that are encoded by the numbers 000 to 999. By concatenating the code of all the letters in our text, we get an integer. For example: Our text is: "pizza", and the codes for p, i, z, a are: 632, 625, 642, 617, respectively. consequently, the text "pizza" can be encoded into the an integer: 632625642642617.

4. We may decode an integer to a text, by dividing the digits into triplets and writing the letter that each triplet represents. To avoid the problem of leading zeros (the number 000123 is equal to 123) we take the *space* character to be the one coded as 000, and ban texts that begin with a string of spaces.

   Therefore, one algorithm can be mapped to one and only one integer. In case it is found that two algorithms map to a single integer number $n \in \mathbb{N}$, then the algorithms are same. By above explanation it is clear that two different integers cannot map to a single algorithm.

   Defining the problem II The next problem is to show a one-to-one function from the set of problems on to the set of real numbers $[0, 1]$. We do this by concatenating the output digits of all possible inputs (one to infinity) of the problem of primality test mentioned above. The problem can be stated as,

   Input: Natural Number
   Output: 1 if input is prime and 0 otherwise.

   For input 1, output is 0, for input 2 the output is 1, for input 3, output is 1, and so on. The real that corresponds to this problem is 0.0110101000101..., which is infinite in length. We have given it only for the first 13 integers. We may consider that for infinitely large number of problems, every problem will have some kind of representation. Like this, every problem has a unique representation.

   The next step is to show how we compare cardinality of infinite sets. Obviously, the problem, represented by a sequence like this real number, are *uncountably infinite.* Whereas, the number of algorithms (step 3 above), are *countably infinite.* Using *Cantor's diagonalization* method (chapter 1), we conclude that there are more real numbers between 0 and 1, than (the number of algorithms).

After the described steps, we conclude that there are infinite numbers of problems that cannot be solved by a computer.

12.4 Unsolvability of Diagonal Language

In the following discussions, our objective will be those problems which cannot be solved by computer. In other words, we will be able to show that no Turing Machine exists to solve such problems.

Consider that M is a TM with alphabet $\Sigma = \{a, b\}$ and $w \in \Sigma^*$, and M accepts w. We had discussed that a TM M can be encoded as $en(M)$, and its input w as $en(w)$, accordingly, for the alphabet Σ, the encoding is:

$$\begin{aligned} &\delta(q_i, a) = (q_j, b, L) \\ &= 001^{i+1}01^2 01^{j+1}01^3 01^2 00 \end{aligned} \tag{12.1}$$

Similarly, all the transitions can be encoded. Hence $\langle M, w\rangle$ is encoded as $00m_i$ $00m_j00\ldots$ $00m_k$, where m_i is a transition, like shown in equation 12.1. We note that all the transitions of a TM, separated by 00 is a big integer number. If this process is followed, we can encode all the possible TMs. If a string is not a well-formed representation of any TM, we take this as encoding of a TM with null moves, hence every binary string, whether long or short, can be taken as representation of some TM. Therefore, all Turing machines are represented by integers $\{1, 2, 3, \ldots, i, \ldots\}$, which is enumeration of Turing machines M_i, and are the countable infinite numbers.

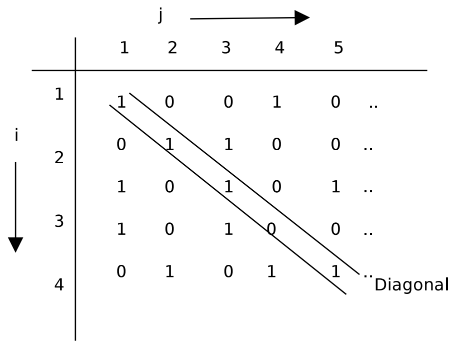

Figure 12.1: Diagonalization

Consider that language for a TM M_i is $L(M_i)$. For example, in typical case the language is $M_i = \{1, 10, 110, 111, 1001, \ldots, \}$. This is subset of lexicographic order binary set: $M_u = \{1, 10, 11, 100, 101, 110, 111, 1000, 1001, \dot{\}}$. In a encoded representation of language M_i, we indicate the presence of string w in M_u as 1 and absence of that as 0. Hence, the encoded representation of above M_i is $\{1, 1, 0, 0, 0, 1, 1, 0, 1, \ldots\}$. Hence, if we construct a matrix of $M_i \times w_j$, then $(i, j) = 1$ if w_j is accepted by M_i, else 0. So, ith row is a characteristic vector for language $L(M_i)$ (Fig. 12.1).

We think of the diagonal vector for the above figure, which is $[1, 1, 1, 0, 1 ...]$, and take this as a characteristic vector, the corresponding language is $\{1, 10, 11, 101, \dot{\}}$, which is obvious by the definition of language we defined in the beginning of this discussion. As next step take the *complement* of this diagonal vector, which is $\{0, 0, 0, 1, 0, \ldots\}$.

We note, that the the complement disagrees with every row vector, because the first element of the complement disagrees with the first element of first row, and similarly it disagrees with all the rows we have listed. Hence, it represents a language which is not accepted by any TM. The complement of the diagonal vector cannot be characteristic vector of any TM, because the TMs are enumerated. Let the language of this complemented characteristic vector be L_d. This language is undecidable, as there does not exists any TM which can recognize it.

Let us assume that L_d is some language. Therefore, it should appear as some string w_j. But, we note that $L_d \notin \{w_k\}$, where w_k is some language. This is a contradiction. So, there does not exist any TM which recognizes the L_d, hence L_d is undecidable.

The Church-Turing Thesis (CTT) also has consequence for undecidability. If a problem cannot be solved by any TM, then it cannot be solved by any algorithm. A decision language having no algorithmic solution, is called *undecidable*.

There are countable TMs (i.e., Algorithms), but the number of problems are uncountable. It follows that there are languages whose membership problems are undecidable.

Example 12.1 *Halting Problem version for C Program.*

The standard "Halting problem" exists for Turing machines. However, we consider its version for C program. The halting problem is like a C program, named as "cp" whose input is a file, called "infile", and, output is "Yes" if cp halts when run with *infile* as input. And, the output is "No" if the cp program does not halt with this infile.

Having given above program and input file, we are interested to write another program *HALT*, whose inputs are *cp* and $infile$, and *HALT* decides whether *cp* halts or not, based on its input the *cp* and $infile$. That is, *HALT* will print yes, if *cp* prints Yes, an *HALT* prints No if *cp* prints No. The halting problem states that that: $\langle cp, infile\rangle$ is *undecidable*, as a general case, that is, *HALT* cannot decide about the outcome of cp with input as infile.

However, If the above is decidable, then there cannot ever can be an infinite loop.

12.5 Barber's Paradox

A barber in a small town declares that he shaves every one in this town who does not shave himself, and that he does not shave any one who shaves himself. We will see that barber's declaration leads to contradiction. This contradiction is heart of the argument to prove that the halting problem is unsolvable.

Let us try this Barber's paradox. Let X denote how the barber behaves, $X = 1$ means "to shave a villager", and $X = 0$ means "not to shave a villager". Similarly, Y denotes how the villager behaves, $Y = 1$ means "to shave himself, and $Y = 0$ means "not to shave himself."

Clearly, what Barber declared is for any villager including himself, so (X, Y) must be $(1, 0)$ or $(0, 1)$.

On the other hand, the barber shaves not only as barber, but also as "one of the villager", i.e., himself, which is : shaves to the barber, and also shaves to villager. Therefore, if barber shaves himself, (X, Y) becomes $(1, 1)$. And, when barber does not shave himself, (X, Y) becomes $(0, 0)$. In either cases, this contradicts what barber has declared.

If the barber is not treated as villager, the contradiction would not occur. The contradiction in above results to *self reference* in the declaration.

12.6 Undecidability of Halting Problem

In general there exists no Turing machine that eventually halts and decides YES/NO for a given problem, then the problem is called as *undecidable*. The problem called the *halting problem*, seeks the answer to the question, whether any Turing machine will eventually halt or run forever. We will show in the remaining discussion in this section that halting problem is undecidable.

At the first instance the halting problem seems decidable. A Turing machine M with input w yielding YES, when w is accepted, and NO, otherwise. This machine with input w is simulated on the Universal Turing machine M_u, so M_u should yield YES and halt when simulated M accepts w. But, when M with input w reject, then the Universal Turing machine M_u runs forever, and hence cannot output NO ever. So we cannot solve the halting problem with the universal Turing machine running M.

Theorem 12.2 *Show that Halting problem is undecidable.*

Proof: Let us assume that there is a TM M that accepts input w with $w \in L(M)$, and rejects $w \notin L(M)$. To model the halting problem, let us assume that there is a TM H that decides this. That is, when H is simulated to run M with input w, so that when M prints "accept", H will also print "accept", and when M print "reject", the H will also print "reject". For this we feed representation $R(M)$ and w as input to H (see Fig. 12.2(a)).

Next, we do minor modification in H, as shown in Fig. 12.2(b), where accept is replaced by halts, and reject is replaced by loop. The modified H is shown as H'. The first is simple, as only the message content has changed. For the reject case, when M rejects w, the H' calls a loop.

In the Fig. 12.2(c), we only input the representation $R(M)$, and by a "copy" TM, another similar string is obtained, so in place of w we have input to H' as $R(M)$. In addition, the outputs: loop and halt are exchanged. Let this modified machine is D (for Diagonalization), as we did complement of the diagonal language L_d (see page no. 12-6). And therefore, the complement is not arbitrary.

Since $R(M)$ was input to TM M simulated by H' earlier, we can also feed $R(D)$ as input to D machine, just like $R(M)$. With this final modification, we have shown the machine in Fig. 12.2(d).

Now we analyze the output of D, and appropriately restate as follows:

1. *If D halts with input $R(D)$ then D loops,*

2. *If D does not halt with input $R(D)$ then D halts.*

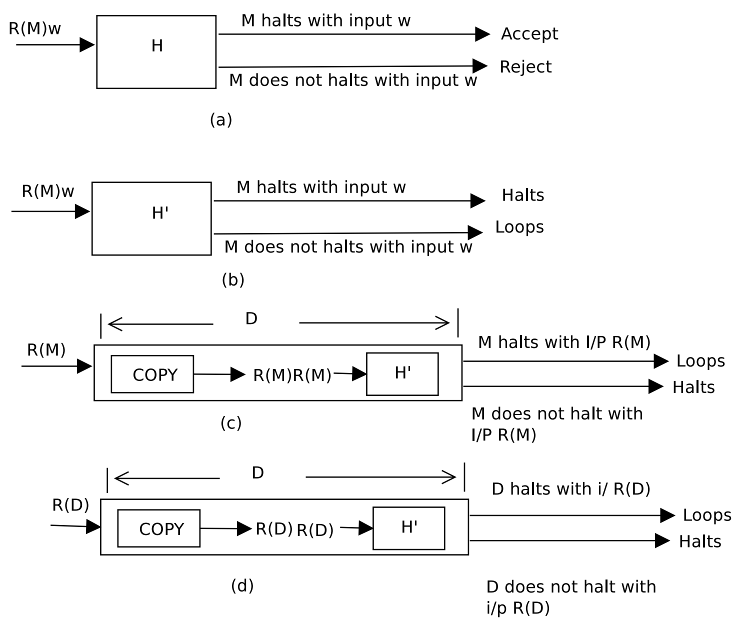

Figure 12.2: Halting Problem

In other words, we note that "D loop if D halts with input $R(D)$", and "D halts with input $R(D)$ if, D does not halt with input $R(D)$". This is a self contradicting statement. Therefore, the halting problem is "undecidable", i.e., when a machines receives its own program (encoding) as input, we cannot decide whether it will halt or not. ■

Why contradiction exists? The contradiction in the above proof uses self reference and Diagonalization. To obtain the standard relation table for a Diagonalization argument, we consider every string $v \in \{0,1\}^*$ to represent TM. If v does not have the form $R(M)$, the one state TM with no transition is assigned to it v. Thus, Turing machines can be listed as M_0, M_1, …, corresponding to strings ε, 0, 1, 00, 10, …, as vertical column in Fig. 12.3, and their representation represented as $\langle M_0\rangle, \langle M_1\rangle, ..., \langle D\rangle$... as horizontal rows.

	$\langle M_0\rangle$	$\langle M_1\rangle$	$\langle M_2\rangle$	•••	$\langle M_i\rangle$	•••	$\langle D\rangle$
M_0	1	0	0		1		0
M_1	0	1	1		0		0
M_2	1	0	1		0		1
⋮							
M_i	1	0	1		0		0
⋮							
D	0	1	0		1		?

Figure 12.3: Does M_i halt when run on itself?

The entries in the cell of rows and column intersection is 1 if a TM accepts its representation, else 0. Now consider the task that lists the table along the horizontal and vertical lines.

The (i, j)th entry in Fig. 12.3 is:

$$\begin{cases} 1, & \text{if } M_i \text{ halts when run with input } R(M_j), \text{i.e., } \langle M_j \rangle \\ 0, & \text{if } M_i \text{ does not halts when run with } R(M_j), \text{ i.e., } \langle M_j \rangle \end{cases} \tag{12.2}$$

Corollary 12.3 *A language*

$$L_H = \{R(M)w \mid R(M) \textit{ is representation of TM } M \textit{ and } M \textit{ halts over } w\}$$

is not Recursively Enumerable (RE). Since it is not RE, it is also not R (recursive).

The Diagonal of the table in Fig. 12.3 represents the answers to the self reference question: "Does M_i halt when run on itself?" The machine D was constructed to produce contradiction in response to this question.

In general if there exists a TM that eventually halts and decides YES/NO for a given problem, the problem is called *decidable*. The problem called "halting problem" to tell whether TM will eventually halt or run for ever is such a problem, i.e., undecidable. In the above we have concluded that halting problem is undecidable.

Following are some fundamental problems which are inherently important or they have played historical role in development of computation theory. Consider that P is a program, and x is a string over input alphabet $\Sigma\{0, 1\}$.

1. *Universal Simulation*: Given a program P and input x to P, determine the output (if any) that P would produce on input x.
2. Type-0 grammar membership: Given a type-0 grammar G and a string x, determine whether x can be derived from start symbol of G.

Theorem 12.4 *The diagonal language L_d is NOT recursively enumerable.*

Proof: Suppose that for the sake of contradiction that L_d is in RE. Then there is a TM M' that accepts L_d. Now what does M' do on input $w(= R(M'))$? If M' accepts w, i.e., $R(M')$, then $R(M') \notin L_d$. But this contradicts $L(M') = L_d$. Hence, the TM M', such that $L(M') = L_d$ cannot exist. So, L_d is not RE. ■

The definition of L_d is motivated by Russell's paradox, where in the definition of some specific set: $S = \{X \mid X \notin X\}$. Where as in Russell's paradox we had to conclude that S is not a set, here we conclude that L_d is not a Turing-acceptable set.