From Entscheidungsproblem to Turing Machine

Hilbert decidability problem

Since old times one of the major maths problems had always been the solution of algebraic equations, to which many mathematical problems refer. The solutions to the equations of the first and second degree in terms of radicals did not present any difficulties since Greeks maths. Problems that can be worked out by finding solutions of algebraic equations in the domain of integer numbers are known since the very beginning of mathematics. In general, the algebraic equations in the domain of integers can be defined this way: P (x, y) = 0, where P is a polynomial with integer coefficients and one, two or more unknowns.

The problem to be solved is: how could we determine, whether it presents solutions in the domain of integer numbers, and, if it does, how to find all of them efficiently? Such equations are called Diophantine equations. Even today We have no general method of solving an arbitrary degree Diophantine equation, furthermore Matiyasevich demonstrates such an algorithm does not exists. All the sophisticated methods invented apply only to very specific types of equations. In 1900 He proposed a list of 23 problems that should become relevant in the next century. The 10th of these 23 Hilbert's problems has been the following: *Determination of the solvability of a Diophantine equation. Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined by a finite number of operations whether the equation is solvable in rational integers.*

Hilbert's tenth problem relates to the also known as "decision problems", i.e. the problem consisting of an infinite number of individual problems, each requiring a definite answer: yes or no. The essence of a decision problem is requirement to find a single method that will give an answer to any individual subset of problems.

After that, in 1928 he generalized the question posing the *Entscheidungsproblem*. The problem asks for an algorithm that takes input from a statement of a first-order

logic, possibly with a finite number of axioms beyond the usual axioms of first-order logic, and answers "Yes" or "No" according to whether the statement is universally valid, i.e., valid in every structure satisfying the axioms.

The first answer: Gödel incompleteness theorems

In order to understand Gödel's work, we must first explain the key concepts essential to it: they are formal system, consistency, and completeness. A formal system is a system of axioms provided with rules of inference, that enables one to generate new theorems. The pattern of axioms is requested to be finite or at least decidable, so must exist a procedure which enables one to mechanically decide whether a given assertion is an axiom or not. If this condition is satisfied, the theory is called “recursively axiomatizable”, or, simply, “axiomatizable”. The laws of inference (of a formal system) are also effective operations, such that it can always be mechanically decided whether one presented a legitimate application of a rule of inference at hand. A formal system is complete if for every statement of the language of the system, either the statement or its negation can be derived (i.e., proved) in the system. A formal system is consistent if there is no proposition such that the proposition itself and its negation are both derivable in the system. Only consistent systems are of any interest in this context, for it is an elementary fact of logic that in an inconsistent formal system every statement is derivable, and consequently, such a system is trivially complete. Gödel established two different though related incompleteness theorems, usually called the first incompleteness theorem and the second incompleteness theorem. Here we analyse just the first and the most relevant.

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.

Gödel's theorem does not merely claim that such statements exist: the method of Gödel's proof explicitly produces a sentence that is neither provable nor refutable in F; the “undecidable” statement can be found mechanically from a specification of F.

The sentence in question is a relatively simple statement of number theory, a purely universal arithmetical sentence.

A common misunderstanding is to interpret Gödel's first theorem was showing that there are truths that cannot be proved. This is, however, incorrect, for the incompleteness theorem does not deal with provability in any absolute sense, while only concerns derivability in some formal system or another.

Gödel -Church–Turing conjecture

Gödel originally only established the incompleteness of a though very comprehensive formalized theory P, a variant of Russell's type-theoretical system PM (for Principia Mathematica), and all extension of P with the same language, whose set of axioms is primitive recursive. Though it turns out that Gödel in fact already had a very general result, it was, at the time, unclear just how general this really was

What was still missing was an analysis of the intuitive notion of decidability, needed in the characterization of the notion of an arbitrary formal system. Recall that the set of axioms and the proof relation of a formalized system are required to be decidable. Mathematicians and logicians have implicitly used the intuitive notion of a decision method since antiquity, and if one asked for a positive solution, it was enough that one presented a concrete method that intuitively stroked everyone for a mechanical method. For the general limitative results, such as the general incompleteness theorems, or the undecidability results, however, a precise mathematical reason of this concept would be needed. Instead of decidable patterns or properties, one often considers effective or computable functions or operations, while in fact these are just two aspect of the same phenomenon—properties of one can be easily ascribed other ones.

Gödel (1934), Alonzo Church (1936) and Alan Turing (1936) came up independently with different proposals for an exact formal definition of computable functions, and consequently, of decidable sets (of numbers). These proposals, though, all turned out to be equivalent. Turing's careful conceptual analysis which used fictional and abstract computing machines was particularly important, as Gödel himself

emphasized (e.g., Gödel 1963). The equation of the intuitive notion and some of these mathematical explications is often called “The Church-Turing-thesis”. The label “recursive function” has, for historical reasons, been dominant in the logical literature. Consequently, decidable sets are often called “recursive sets”. For a proper understanding of the incompleteness and undecidability results, it is vital in order to understand the difference between the two key notions regarding sets. First, there may be a mechanical method which decides whether any given number belongs to the set at issue or not (in which case the set is called “decidable” or “recursive”), and, second, there may be a mechanical method which generates or lists the elements of the set, number by number. In the latter case, the set is called “recursively enumerable” (r.e.), that is to say, it can be effectively generated, or it is “semi-decidable.” It is a fundamental result of the theory of computability (or “the theory of recursive functions”) that there are semi-decidable sets, sets which can be effectively generated (i.e., are recursively enumerable), while are not decidable (i.e., not recursive). In fact, this is, in the very abstract level, the essence of the first incompleteness theorem. Moreover, if both a set and its complement are recursively enumerable, the set is recursive, i.e., decidable.

Turing machine: definition and the halting problem

The read/write head is programmable. It is be helpful to think of the operation of programming as consisting of altering the head's internal wiring by means of a plugboard arrangement. In order to compute with the device, you program it, inscribe the input on the tape (in binary or decimal code, say), place the head over the square containing the leftmost input symbol, and set the machine in motion. Once the computation is completed, the machine will come to a halt with the head positioned over the square containing the leftmost symbol of the output (or elsewhere if so programmed).

The head contains a subdevice that I call the *indicator*. This is a second form of working memory. The indicator can be set to any one of several 'positions'. In Turing machine jargon, the position of the indicator at any time is called the *state* of the machine at that time. To give a simple example of the indicator's function, it may be

used in order to keep track of whether the symbol last encountered was '0' or '1'. If '0', the indicator is set to its first position, and if '1', to its second position. There are just six types of fundamental operation that a Turing machine performs in the course of a computation. It can:

- read (i.e. identify) the symbol currently under the head
- write a symbol on the square currently under the head (after first deleting the symbol already written there, if any)
- move the tape left one square
- move the tape right one square
- change state
- halt.

These are called the *primitive* or *atomic* operations of the machine. A complicated computation may consist of hundreds of thousands, or even millions, of occurrences of these atoms

A program or 'instruction table' for a Turing machine is a finite collection of instructions, each calling for certain atomic operations to be performed if certain conditions are met.

A Turing machine program is said to *terminate* just in case any Turing machine running the program will halt no matter what the input. An easy way for writing a *non-terminating* program is simply to omit an instruction to halt. An example of a non-terminating Turing machine program is a program that calculates sequentially each digit of the decimal representation of pi (say by using one of the standard power series expressions for π. A Turing machine running this program will spend all eternity writing out the decimal representation of pi digit by digit, 3.14159.etc. Turing called the numbers that can be written out by a Turing machine the *computable* numbers. That is, a number is computable, in Turing's sense, if and only if there is a Turing machine that calculates in sequence each digit of the number's decimal representation. Straight off, one might expect it to be the case that *every* number that *has* a decimal representation, either finite or infinite--that is to say, every *real* number-

-is computable. For what could prevent there being, for any real number, a Turing machine that 'churns out' that number's decimal representation digit by digit? However, Turing was able to prove that not every real number is computable. The decimal representations of some real numbers are so completely lacking in pattern that there simply is no finite table of instructions of the sort that can be followed by a Turing machine for calculating the nth digit of the representation, for arbitrary n. In fact, computable numbers are relatively scarce among the real numbers.

Following Turing, to say that a mathematical *function*, for example addition over the integers, is computable is to say that there is a Turing machine which is such that if, for any pair of integers x and y, the machine is given x and y as input, it will print out the value of x+y and halt.

Addition over the integers is a computable function, whereas addition over the real numbers is not a computable function. This is because there is no way even of inscribing the inputs x and y on a Turing machine's tape if x and y are incomputable numbers[1]. Is addition over the *computable numbers* a computable function? That is, is x+y computable whenever x and y are both computable numbers?

The answer is *yes*, instead off the top of one's head one might think otherwise, for if x (or y) is a computable number having an infinite decimal representation, for example [1], how can x be input, given the restriction that the input inscribed on the tape must consist of a finite number of symbols? The solution is to input x in the form of a *program*. The computable number x may be represented by means of a program which, if inscribed on the otherwise blank tape of some universal Turing machine, would cause the universal machine to calculate the decimal representation of x digit by digit.

Thus, Turing has given us a new method for representing numbers. In this way, there are *finite* representations of some numbers which, in the decimal system, can be represented only by means of an infinite sequence of symbols.

[1] Input must consist of a finite number of symbols

Universal Turing machine

There are two ways of arranging for a Turing machine to act in accordance with a machine table or program. One is to appropriately modify the 'wiring' in the head of a Turing machine. The other is to translate the machine table into, say, binary code and inscribe the result on the tape of a special type of Turing machine known as a *universal* Turing machine.

Turing was able to demonstrate that there is a table U which is such that if the head of a Turing machine is programmed in accordance with U, and if any table whatsoever, P, is translated and written out on the machine's tape, then the machine will behave *as if* its head had been programmed in accordance with P. A universal Turing machine is any Turing machine whose head has been programmed in accordance with U.

There are many different tables that will do the work of U and thus many distinct universal Turing machines, all equivalent in computational power.

Bibliography

Yuri V. Matiyasevich, Hilbert's Tenth Problem, MIT Press, Cambridge, Massachusetts, 1993.

Kozen, D.C., Automata and Computability, Springer,2012

Sipser M., Introduction to the Theory of Computation, PWS Publishing Co,1996

Torkel Franzén, Gödel's Theorem: An Incomplete Guide to its Use and Abuse. A.K. Peters, 2005.

Per Lindström. Aspects of Incompleteness, Lecture Notes in Logic v. 10,1997.

Boolos, George; John Burgess; Richard Jeffrey. Computability and Logic (4th ed.). Cambridge UK: Cambridge University Press, 2002.

Hartley Rogers, Jr., Theory of Recursive Functions and Effective Computability, The MIT Press, Cambridge MA, 1987

.