Alan M. Turing

# The Essential Turing

Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life *plus* The Secrets of Enigma

*Edited by* **B. Jack Copeland**

CLARENDON PRESS · OXFORD

A document written by Woodger in 1947 used the single 'm' spelling: 'A Program for Version H'.[55] Woodger recalls: 'We used both spellings carelessly for some years until Goodwin (Superintendent of Mathematics Division from 1951) laid down the rule that the "American" spelling should be used.'[56] It is possible that the single 'm' spelling first came to the NPL via the American engineer Huskey, who spent 1947 with the ACE group. Huskey was responsible for 'Version H', a scaled-down form of Turing's design for the ACE (see Chapter 10).

Like Turing, Eckert and Mauchly, the chief architects of ENIAC, probably inherited the terms 'programming' and 'program' from the plug-board calculator. In 1942, while setting out the idea of a high-speed electronic calculator, Mauchly used the term 'programming device' (which he sometimes shortened to 'program device') to refer to a mechanism whose function was to determine how and when the various component units of a calculator shall perform.[57] In the summer of 1946 the Moore School organized a series of influential lectures entitled 'Theory and Techniques for Design of Electronic Digital Computers'. In the course of these, Eckert used the term 'programming' in a similar sense when describing the new idea of storing instructions in high-speed memory: 'We . . . feed those pieces of information which relate to programming from the memory.'[58] Also in 1946, Burks, Goldstine, and von Neumann (all ex-members of the Moore School group) were using the verb-form 'program the machine', and were speaking of 'program orders' being stored in memory.[59] The modern nominalized form appears not to have been adopted in the USA until a little later. Huskey says, 'I am pretty certain that no one had written a "program" by the time I left Philadelphia in June 1946.'[60]

## Part II Computability and Uncomputability

### 8. Circular and Circle-Free Machines

Turing calls the binary digits '0' and '1' symbols 'of the first kind'. Any symbols that a computing machine is able to print apart from the binary digits—such as

[55] M. Woodger, 'A Program for Version H', handwritten MS, 1947 (in the Woodger Papers, National Museum of Science and Industry, Kensington, London (catalogue reference N30/37) ).

[56] Letter from Woodger to Copeland (6 Oct. 2000).

[57] J. W. Mauchly, 'The Use of High Speed Vacuum Tube Devices for Calculating' (1942), in Randell, *The Origins of Digital Computers: Selected Papers*.

[58] J. P. Eckert, 'A Preview of a Digital Computing Machine' (15 July 1946), in M. Campbell-Kelly and M. R. Williams (eds.), *The Moore School Lectures* (Cambridge, Mass.: MIT Press, 1985), 114.

[59] Sections 1.2, 5.3 of Burks, Goldstine, and von Neumann, 'Preliminary Discussion of the Logical Design of an Electronic Computing Instrument' (von Neumann, *Collected Works*, vol. v, 15, 43).

[60] Letter from Huskey to Copeland (3 Feb. 2002).

'2', '*', '*x*', and blank—Turing calls 'symbols of the second kind' (p. 60). He also uses the term 'figures' for symbols of the first kind.

A computing machine is said by Turing to be *circular* if it never prints more than a finite number of symbols of the first kind. A computing machine that will print an infinite number of symbols of the first kind is said to be *circle-free* (p. 60). For example, a machine operating in accordance with Table 1 is circle-free. (The terms 'circular' and 'circle-free' were perhaps poor choices in this connection, and the terminology has not been followed by others.)

A simple example of a circular machine is one set up to perform a single calculation whose result is an integer. Once the machine has printed the result (in binary notation), it prints nothing more.

A circular machine's scanner need not come to a halt. The scanner may continue moving along the tape, printing nothing further. Or, after printing a finite number of binary digits, a circular machine may work on forever, printing only symbols of the second kind.

Many real-life computing systems are circle-free, for example automated teller machine networks, air traffic control systems, and nuclear reactor control systems. Such systems never terminate by design and, barring hardware failures, power outages, and the like, would continue producing binary digits forever.

In Section 8 of 'On Computable Numbers' Turing makes use of the circular/circle-free distinction in order to formulate a mathematical problem that cannot be solved by computing machines.

### 9. Computable and Uncomputable Sequences

The sequence of binary digits printed by a given computing machine on the F-squares of its tape, starting with a blank tape, is called the *sequence computed by the machine*. Where the given machine is circular, the sequence computed by the machine is finite. The sequence computed by a circle-free machine is infinite.

A sequence of binary digits is said to be a *computable* sequence if it is the sequence computed by some circle-free computing machine. For example, the infinite sequence 010101 . . . is a computable sequence.

Notice that although the finite sequence 010, for example, is the sequence computed by some machine, this sequence is *not* a *computable* sequence, according to Turing's definition. This is because, being finite, 010 is not the sequence computed by any circle-free machine. According to Turing's definition, no finite sequence is a computable sequence. Modern writers usually define 'computable' in such a way that every finite sequence is a computable sequence, since each of them can be computed (e.g. by means of an instruction table that simply prints the desired sequence). Turing, however, was not much interested in finite sequences.

The focus of Turing's discussion is his discovery that not every infinite sequence of binary digits is a computable sequence. That this is so is shown by what mathematicians call a *diagonal* argument.

## *The diagonal argument*

Imagine that all the computable sequences are listed one under another. (The order in which they are listed does not matter.) The list stretches away to infinity both downwards and to the right. The top left-hand corner might look like this:

```
01100101011000100101101001000111101 . . .
0101110100111000111111111111110111 . . .
110100000110110101000000110010000011 . . .
                    .
                    .
                    .
```

Let's say that this list was drawn up in the following way (by an infinite deity, perhaps). The first sequence on the list is the sequence computed by the machine with a description number that is smaller than any description number of any other circle-free machine. The second sequence on the list is the one computed by the circle-free machine with the *next smallest* description number, and so on. Every computable sequence appears somewhere on this list. (Some will in fact be listed twice, since sometimes different description numbers correspond to the same sequence.)

To prove that not all infinite binary sequences are computable, it is enough to describe one that does not appear on this list. To this end, consider the infinite binary sequence formed by moving diagonally down and across the list, starting at the top left:

```
01100 . . .
01011 . . .
11010 . . .
   .
   .
   .
```

The twist is to transform this sequence into a different one by switching each '0' lying on the diagonal to '1' and each '1' to '0'. So the first digit of this new sequence is formed by switching the first digit of the first sequence on the list (producing 1); the second digit of the sequence is formed by switching the second digit of the second sequence on the list (producing 0); the third digit is formed by switching the third digit of the third sequence on the list (producing 1); and so on. Turing calls this sequence 'β' (p. 72).

A moment's reflection shows that β cannot itself be one of the listed sequences, since it has been constructed in such a way that it differs from each of these. It differs from the first sequence on the list at the first digit. It differs from the second sequence on the list at the second digit. And so on. Therefore, since every computable sequence appears somewhere on this list, β is not among the computable sequences.

## *Why the computable sequences are listable*

A sceptic might challenge this reasoning, saying: 'Perhaps the computable sequences *cannot* be listed. In assuming that the computable sequences can be listed, one, two, three, and so on, you are assuming in effect that each computable sequence can be paired off with an integer (no two sequences being paired with the same integer). But what if the computable sequences cannot be paired off like this with the integers? Suppose that there are just *too many* computable sequences for this to be possible.' If this challenge were successful, it would pull the rug out from under the diagonal argument.

The response to the challenge is this. Each circle-free Turing machine produces just one computable sequence. So there cannot be more computable sequences than there are circle-free Turing machines. But there certainly cannot be more circle-free Turing machines than there are integers. This is because every Turing machine has a description number, which *is* an integer, and this number is not shared by any other Turing machine.

This reasoning shows that each computable sequence can be paired off with an integer, one sequence per integer. As Turing puts this, the computable sequences are 'enumerable' (p. 68).

The totality of infinite binary sequences, however, is *non-enumerable*. Not all the sequences can be paired off with integers in such a way that no integer is allocated more than one sequence. This is because, once *every* integer has had an infinite binary sequence allocated to it, one can 'diagonalize' in the above way and produce an extra sequence.

## *Starting with a blank tape*

Incidentally, notice the significance, in Turing's definition of *sequence computed by the machine*, of the qualification 'starting with a blank tape'. If the computing machine were allowed to make use of a tape that had already had an infinite sequence of figures printed on it by some means, then the concept of a computable sequence would be trivialized. Every infinite binary sequence would become computable, simply because any sequence of digits whatever—e.g. β—could be present on the tape before the computing machine starts printing.

The following trivial programme causes a machine to run along the tape printing the figures that are already there!

| | | | |
|---|---|---|---|
| a | 1 | P[1], R | a |
| a | 0 | P[0], R | a |
| a | - | P[-], R | a |

(The third line is required to deal with blank E-squares, if any.)

## 10. Computable and Uncomputable Numbers

Prefacing a binary sequence by '0' produces a real number expressed in the form of a binary decimal. For example, prefacing the binary sequence 010101 . . . produces 0.010101 . . . (the binary form of the ordinary decimal 0.363636 . . . ). If *B* is the sequence of binary digits printed by a given computing machine, then 0.*B* is called the *number computed by the machine.*

Where the given machine is circular, the number computed by the machine is always a rational number. A circle-free machine may compute an irrational number ($\pi$, for example).

A number computed by a circle-free machine is said to be a *computable number.* Turing also allows that any number 'that differs by an integer' from a number computed by a circle-free machine is a computable number (p. 61). So if *B* is the infinite sequence of binary digits printed by some circle-free machine, then the number computed by the machine, 0.*B*, is a computable number, as are all the numbers that differ from 0.*B* by an integer: 1.*B*, 10.*B*, etc.

In Section 10 of 'On Computable Numbers', Turing gives examples of large classes of numbers that are computable. In particular, he proves that the important numbers $\pi$ and *e* are computable.

Not all real numbers are computable, however. This follows immediately from the above proof that not all infinite binary sequences are computable. If *S* is an infinite binary sequence that is uncomputable, then 0.*S* is an uncomputable number.

## 11. The Satisfactoriness Problem

In Section 8 of 'On Computable Numbers' Turing describes two mathematical problems that cannot be solved by computing machines. The first will be referred to as the *satisfactoriness problem.*

### Satisfactory descriptions and numbers

A standard description is said to be *satisfactory* if the machine it describes is circle-free. (Turing's choice of terminology might be considered awkward, since there need be nothing at all unsatisfactory, in the usual sense of the word, about a circular machine.)

A number is satisfactory if it is a description number of a circle-free machine. A number is unsatisfactory if either it is a description number of a circular machine, or it is not a description number at all.

The satisfactoriness problem is this: decide, of any arbitrarily selected standard description—or, equivalently, any arbitrarily selected description number—whether or not it is satisfactory. The decision must be arrived at in a finite number of steps.

### The diagonal argument revisited

Turing approaches the satisfactoriness problem by reconsidering the above proof that not every binary sequence is computable.

Imagine someone objecting to the diagonal argument: 'Look, there must be something wrong with your argument, because $\beta$ evidently *is* computable. In the course of the argument, you have in effect given instructions for computing each digit of $\beta$, in terms of counting out digits and switching the relevant ones. Let me try to describe how a Turing machine could compute $\beta$. I'll call this Turing machine BETA. BETA is similar to the universal machine in that it is able to simulate the activity of any Turing machine that one wishes. First, BETA simulates the circle-free machine with the smallest description number. BETA keeps up the simulation just as far as is necessary in order to discover the first digit of the sequence computed by this machine. BETA then switches this digit, producing the first digit of $\beta$. Next, BETA simulates the circle-free machine with the next smallest description number, keeping up the simulation until it finds the second digit of the sequence computed by this machine. And so on.'

The objector continues: 'I can make my description of BETA specific. BETA uses only the E-squares of its tape to do its simulations, erasing its rough work each time it begins a new simulation. It prints out the digits of $\beta$ on successive F-squares. I need to take account of the restriction that, in order for it to be said that $\beta$ is the sequence computed by BETA, BETA must produce the digits of $\beta$*starting from a blank tape.* What BETA will do first of all, starting from a blank tape, is find the smallest description number that corresponds to a circle-free machine. It does this by checking through the integers, one by one, starting at 1. As BETA generates the integers one by one, it checks each to see whether it is a description number. If the integer is not a description number, then BETA moves on to the next. If the integer is a description number, then BETA checks whether the number is satisfactory. Once BETA finds the first integer to describe a circle-free machine, it uses the instructions contained in the description number in order to simulate the machine. This is how BETA finds the first digit of $\beta$. Then BETA continues its search through the integers, until it finds the next smallest description number that is satisfactory. This enables BETA to calculate the second digit of $\beta$. And so on.'

Turing tackles this objection head on, proving that no computing machine can possibly do what BETA is supposed to do. It suffices for this proof to consider a slightly simplified version of BETA, which Turing calls $\mathfrak{H}$. $\mathfrak{H}$ is just like BETA except that $\mathfrak{H}$ does not switch the digits of the list's 'diagonal' sequence. $\mathfrak{H}$ is supposed to write out (on the F-squares) the successive digits not of β but of the 'diagonal' sequence itself: the sequence whose first digit is the first digit of the first sequence on the list, whose second digit is the second digit of the second sequence on the list, and so on. Turing calls this sequence $\beta'$. If no computing machine can compute $\beta'$, then there is no such computing machine as BETA—because if there were, a machine that computes $\beta'$ could be obtained from it, simply by deleting the instructions to switch each of the digits of the diagonal sequence.

### *What happens when $\mathfrak{H}$ meets itself?*

Turing asks: *what happens when, as $\mathfrak{H}$ searches through the integers one by one, it encounters a number describing $\mathfrak{H}$ itself?* Call this description number K.

$\mathfrak{H}$ must first check whether K is a description number. Having ascertained that it is, $\mathfrak{H}$ must test whether K is satisfactory. Since $\mathfrak{H}$ is supposed to be computing the endless binary sequence $\beta'$, $\mathfrak{H}$ itself must be circle-free. So $\mathfrak{H}$ must pronounce K to be satisfactory.

In order to find the next digit of $\beta'$, $\mathfrak{H}$ must now simulate the behaviour of the machine described by K. Since $\mathfrak{H}$ is that machine, $\mathfrak{H}$ must simulate its *own* behaviour, starting with its very first action. There is nothing wrong with the idea of a machine starting to simulate its own previous behaviour (just as a person might act out some episode from their own past). $\mathfrak{H}$ first simulates (on its E-squares) the series of actions that it performed up to and including writing down the first digit of $\beta'$, then its actions up to and including writing down the second digit of $\beta'$, and so on.

Eventually, however, $\mathfrak{H}$'s simulation of its own past reaches the point where $\mathfrak{H}$ began to simulate the behaviour of the machine described by K. What must $\mathfrak{H}$ do now? $\mathfrak{H}$ must simulate the series of actions that it performed when simulating the series of actions that culminated in its writing down the first digit of $\beta'$, and then simulate the series of actions that it performed when simulating the series of actions that culminated in its writing down the second digit of $\beta'$, and so on! $\mathfrak{H}$ is doomed to relive its past forever.

From the point when it began simulating itself, $\mathfrak{H}$ writes only on the E-squares of its tape and never adds another digit to the sequence on its F-squares. Therefore, $\mathfrak{H}$ does *not* compute $\beta'$. $\mathfrak{H}$ computes some finite number of digits of $\beta'$ and then sticks.

The problem lies with the glib assumption that $\mathfrak{H}$ and BETA are able to determine whether each description number is satisfactory.

### *No computing machine can solve the satisfactoriness problem*

Since, as has just been shown, no computing machine can possibly do what $\mathfrak{H}$ was introduced to do, one of the various tasks that $\mathfrak{H}$ is supposed to carry out must be impossible for a computing machine. But all the things that $\mathfrak{H}$ is supposed to do *apart* from checking for satisfactoriness—decide whether a number is a description number, extract instructions from a description number, simulate a machine that follows those instructions, and so on—are demonstrably things that can be done by the universal machine.

By a process of elimination, then, the task that it is impossible for a computing machine to carry out must be that of determining whether each description number is satisfactory or not.

## 12. The Printing and Halting Problems

### *The printing problem*

Some Turing-machine programmes print '0' at some stage in their computation; all the remaining programmes never print '0'. Consider the problem of deciding, given any arbitrarily selected programme, into which of these two categories it falls. This is an example of the printing problem.

The printing problem (p. 73) is the problem of determining whether or not the machine described by any arbitrarily selected standard description (or, equivalently, any arbitrarily selected description number) ever prints a certain symbol ('0', for example). Turing proves that if the printing problem were solvable by some computing machine, then the satisfactoriness problem would be too. Therefore neither is.

### *The halting problem*

Another example of a problem that cannot be solved by computing machines, and a close relative of the printing problem, is the *halting problem*. This is the problem of determining whether or not the machine described by any arbitrarily selected standard description eventually halts—i.e. ceases moving altogether—when started on a given tape (e.g. a blank tape).

The machine shown in Table 1 is rather obviously one of those that never halt—but in other cases it is not at all obvious from a machine's table whether or not it halts. Simply watching the machine run (or a simulation of the machine) is of little help, for what can be concluded if after a week or a year the machine has not halted? If the machine does eventually halt, a watching human—or Turing machine—will sooner or later find this out; but in the case of a machine that has not yet halted, there is no systematic method for deciding whether or not it is going to halt.

The halting problem was so named (and, it appears, first stated) by Martin Davis.[61] The proposition that the halting problem cannot be solved by computing machine is known as the 'halting theorem'.[62] (It is often said that Turing stated and proved the halting theorem in 'On Computable Numbers', but strictly this is not true.)

## 13. The Church–Turing Thesis

### *Human computers*

When Turing wrote 'On Computable Numbers', a computer was not a machine at all, but a human being. A computer—sometimes also spelt 'computor'—was a mathematical assistant who calculated by rote, in accordance with a systematic method. The method was supplied by an overseer prior to the calculation. Many thousands of human computers were employed in business, government, and research establishments, doing some of the sorts of calculating work that nowadays is performed by electronic computers. Like filing clerks, computers might have little detailed knowledge of the end to which their work was directed.

The term 'computing machine' was used to refer to small calculating machines that mechanized elements of the human computer's work. These were somewhat like today's non-programmable hand-calculators: they were not automatic, and each step—each addition, division, and so on—was initiated manually by the human operator. A computing machine was in effect a homunculus, calculating more quickly than an unassisted human computer, but doing nothing that could not in principle be done by a human clerk working by rote. For a complex calculation, several dozen human computers might be required, each equipped with a desk-top computing machine.

In the late 1940s and early 1950s, with the advent of electronic computing machines, the phrase 'computing machine' gave way gradually to 'computer'. During the brief period in which the old and new meanings of 'computer' coexisted, the prefix 'electronic' or 'digital' would usually be used in order to distinguish machine from human. As Turing stated, the new electronic machines were 'intended to carry out any definite rule of thumb process which could have been done by a human operator working in a disciplined but unintelligent manner'.[63] Main-frames, laptops, pocket calculators, palm-pilots—all carry out

[61] See M. Davis, *Computability and Unsolvability* (New York: McGraw-Hill, 1958), 70. Davis thinks it likely that he first used the term 'halting problem' in a series of lectures that he gave at the Control Systems Laboratory at the University of Illinois in 1952 (letter from Davis to Copeland, 12 Dec. 2001).

[62] It is interesting that if one lifts the restriction that the determination must be carried out in a *finite* number of steps, then Turing machines *are* able to solve the halting and printing problems, and moreover in a finite time. See B. J. Copeland, 'Super Turing-Machines', *Complexity*, 4 (1998), 30–2, and 'Accelerating Turing Machines', *Minds and Machines*, 12 (2002), 281–301.

[63] Turing's *Programmers' Handbook for Manchester Electronic Computer*, 1.

work that a human rote-worker could do, if he or she worked long enough, and had a plentiful enough supply of paper and pencils.

It must be borne in mind when reading 'On Computable Numbers' that Turing there used the word 'computer' in this now archaic sense. Thus he says, for example, 'Computing is normally done by writing certain symbols on paper' (p. 75) and 'The behaviour of the computer at any moment is determined by the symbols which he is observing, and his "state of mind"' (p. 75).

The Turing machine is an idealization of the human computer (p. 59): 'We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions ... called "*m*-configurations". The machine is supplied with a "tape" ...' Wittgenstein put the point in a striking way: 'Turing's "Machines". These machines are *humans* who calculate.'[64]

In the primary sense, a computable number is a real number that can be calculated by a human computer—or in other words, a real number that a human being can calculate by means of a systematic method. When Turing asserts that 'the "computable" numbers include all numbers which would naturally be regarded as computable' (p. 74), he means that each number that is computable in this primary sense is also computable in the technical sense defined in Section 2 of 'On Computable Numbers' (see Section 10 of this introduction).

### *The thesis*

*Turing's thesis*, that

> the 'computable' numbers include all numbers which would naturally be regarded as computable,

is now known as the *Church–Turing thesis.*

Some other ways of expressing the thesis are:

1. The universal Turing machine can perform any calculation that any human computer can carry out.
2. Any systematic method can be carried out by the universal Turing machine.

The Church–Turing thesis is sometimes heard in the strengthened form:

> Anything that can be made completely precise can be programmed for a universal digital computer.

However, this strengthened form of the thesis is false.[65] The printing, halting, and satisfactoriness problems are completely precise, but of course cannot be programmed for a universal computing machine.

[64] L. Wittgenstein, *Remarks on the Philosophy of Psychology*, vol. i (Oxford: Blackwell, 1980), § 1096.

[65] As Martin Davis emphasized long ago in his *Computability and Unsolvability*, p. vii.

## Systematic methods

A systematic method—sometimes also called an *effective* method and a *mechanical* method—is any mathematical method of which all the following are true:

- the method can, in practice or in principle, be carried out by a human computer working with paper and pencil;
- the method can be given to the human computer in the form of a *finite* number of instructions;
- the method demands neither insight nor ingenuity on the part of the human being carrying it out;
- the method will definitely work if carried out without error;
- the method produces the desired result in a finite number of steps; or, if the desired result is some *infinite* sequence of symbols (e.g. the decimal expansion of $\pi$), then the method produces each individual symbol in the sequence in some finite number of steps.

The term 'systematic' and its synonyms 'effective' and 'mechanical' are terms of art in mathematics and logic. They do not carry their everyday meanings. For example: if some type of machine were able to solve the satisfactoriness problem, the method it used would not be systematic or mechanical in *this* sense. (Turing is sometimes said to have proved that *no machine* can solve the satisfactoriness problem. This is not so. He demonstrates only that his idealized human computers—Turing machines—cannot solve the satisfactoriness problem. This does not in itself rule out the possibility that some other type of machine might be able to solve the problem.[66])

Turing sometimes used the expression *rule of thumb* in place of 'systematic'. If this expression is employed, the Church–Turing thesis becomes (Chapter 10, p. 414):

> LCMs can do anything that could be described as 'rule of thumb' or 'purely mechanical'.

'LCM' stands for 'logical computing machine', a term that Turing seems to have preferred to the (then current) 'Turing machine'.

Section 9 of 'On Computable Numbers' contains a bouquet of arguments for Turing's thesis. The arguments are persuasive, but do not offer the certainty of mathematical proof. As Turing says wryly of a related thesis in Chapter 17 (p. 588): 'The statement is . . . one which one does not attempt to prove. Propaganda is more appropriate to it than proof.'

Additional arguments and other forms of evidence for the thesis amassed. These, too, left matters short of absolute certainty. Nevertheless, before long it was, as Turing put it, 'agreed amongst logicians' that his proposal gives the

[66] See R. Gandy, 'Church's Thesis and Principles for Mechanisms', in J. Barwise, H. J. Keisler, and K. Kunen (eds.), *The Kleene Symposium* (Amsterdam: North-Holland, 1980).

'correct accurate rendering' of talk about systematic methods (Chapter 10, p. 414).[67] There have, however, been occasional dissenting voices over the years (for example, Kalmár and Péter).[68]

## The converse of the thesis

The *converse* of the Church–Turing thesis is:

> Any number, or binary sequence, that can be computed by the universal Turing machine can be calculated by means of a systematic method.

This is self-evidently true—the instruction table on the universal machine's tape is itself a specification of a systematic method for calculating the number or sequence in question. In principle, a human being equipped with paper and pencil could work through the instructions in the table and write out the digits of the number, or sequence, without at any time exercising ingenuity or insight ('in principle' because we have to assume that the human does not throw in the towel from boredom, die of old age, or use up every sheet of paper in the universe).

## Application of the thesis

The concept of a systematic method is an informal one. Attempts—such as the above—to explain what counts as a systematic method are not rigorous, since the requirement that the method demand neither insight nor ingenuity is left unexplicated.

One of the most significant achievements of 'On Computable Numbers'—and this was a large step in the development of the mathematical theory of computation—was to propose a rigorously defined expression with which the informal expression 'by means of a systematic method' might be replaced. The rigorously defined expression is, of course, 'by means of a Turing machine'.

The importance of Turing's proposal is this. If the proposal is correct—i.e. if the Church–Turing thesis is true—then talk about the existence or non-existence of systematic methods can be replaced throughout mathematics and logic by talk about the existence or non-existence of Turing-machine programmes. For instance, one can establish that there is no systematic method at all for doing such-and-such a thing by proving that no Turing machine can do the thing in question. This is precisely Turing's strategy with the *Entscheidungsproblem*, as explained in the next section.

[67] There is a survey of the evidence in chapters 12 and 13 of S. C. Kleene, *Introduction to Metamathematics* (Amsterdam: North-Holland, 1952).

[68] L. Kalmár, 'An Argument against the Plausibility of Church's Thesis', R. Péter, 'Rekursivität und Konstruktivität'; both in A. Heyting (ed.), *Constructivity in Mathematics* (Amsterdam: North-Holland, 1959).

### *Church's contribution*

In 1935, on the other side of the Atlantic, Church had independently proposed a different way of replacing talk about systematic methods with formally precise language (in a lecture given in April of that year and published in 1936).[69] Turing learned of Church's work in the spring of 1936, just as 'On Computable Numbers' was nearing completion (see the introduction to Chapter 4).

Where Turing spoke of numbers and sequences, Church spoke of mathematical *functions*. ($x^2$ and $x + y$ are examples of mathematical functions. 4 is said to be the *value of the function* $x^2$ for $x = 2$.) Corresponding to each computable sequence $S$ is a computable function $fx$ (and vice versa). The value of $fx$ for $x = 1$ is the first digit of $S$, for $x = 2$, the second digit of $S$, and so on. In 'On Computable Numbers' Turing said (p. 58): 'Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions . . . I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique.'

Church's analysis was in terms of his and Stephen Kleene's concept of a *lambda-definable* function. A function of positive integers is said to be lambda-definable if the values of the function can be calculated by a process of repeated substitution.

Thus we have alongside Turing's thesis

*Church's thesis*: every function of positive integers whose values can be calculated by a systematic method is lambda-definable.

Although Turing's and Church's approaches are different, they are nevertheless equivalent, in the sense that every lambda-definable function is computable by the universal machine and every function (or sequence) computable by the universal machine is lambda-definable.[70] Turing proved this in the Appendix to 'On Computable Numbers' (added in August 1936).

The name 'Church–Turing thesis', now standard, seems to have been introduced by Kleene, with a flourish of bias in favour of his mentor Church: 'So Turing's and Church's theses are equivalent. We shall usually refer to them both as *Church's thesis*, or in connection with that one of its . . . versions which deals with "Turing machines" as *the Church-Turing thesis*.'[71]

Although Turing's and Church's theses are equivalent in the logical sense, there is nevertheless good reason to prefer Turing's formulation. As Turing wrote in 1937: 'The identification of "effectively calculable" functions with computable

[69] Church, 'An Unsolvable Problem of Elementary Number Theory'.

[70] Equivalent, that is, if the computable functions are restricted to functions of positive integers. Turing's concerns were rather more general than Church's, in that whereas Church considered only functions of positive integers, Turing described his work as encompassing 'computable functions of an integral variable or a real or computable variable, computable predicates, and so forth' (p. 58, below). Turing intended to pursue the theory of computable functions of a real variable in a subsequent paper, but in fact did not do so.

[71] S. C. Kleene, *Mathematical Logic* (New York: Wiley, 1967), 232.

functions is possibly more convincing than an identification with the λ-definable [lambda-definable] or general recursive functions.'[72] Church acknowledged the point:

As a matter of fact, there is . . . equivalence of three different notions: computability by a Turing machine, general recursiveness in the sense of Herbrand–Gödel–Kleene, and λ-definability in the sense of Kleene and [myself]. Of these, the first has the advantage of making the identification with effectiveness in the ordinary (not explicitly defined) sense evident immediately . . . The second and third have the advantage of suitability for embodiment in a system of symbolic logic.[73]

The great Kurt Gödel, it seems, was unpersuaded by Church's thesis until he saw Turing's formulation. Kleene wrote:

According to a November 29, 1935, letter from Church to me, Gödel 'regarded as thoroughly unsatisfactory' Church's proposal to use λ-definability as a definition of effective calculability . . . It seems that only after Turing's formulation appeared did Gödel accept Church's thesis.[74]

Hao Wang reports Gödel as saying: 'We had not perceived the sharp concept of mechanical procedures sharply before Turing, who brought us to the right perspective.'[75]

Gödel described Turing's analysis of computability as 'most satisfactory' and 'correct . . . beyond any doubt'.[76] He also said: 'the great importance of . . . Turing's computability . . . seems to me . . . largely due to the fact that with this concept one has for the first time succeeded in giving an absolute definition of an interesting epistemological notion.'[77]

## 14. The *Entscheidungsproblem*

In Section 11 of 'On Computable Numbers', Turing turns to the *Entscheidungsproblem*, or *decision problem*. Church gave the following definition of the *Entscheidungsproblem*:

By the Entscheidungsproblem of a system of symbolic logic is here understood the problem to find an effective method by which, given any expression Q in the notation of the system, it can be determined whether or not Q is provable in the system.[78]

[72] Turing, 'Computability and λ-Definability', *Journal of Symbolic Logic*, 2 (1937), 153–63 (153).

[73] Church's review of 'On Computable Numbers' in *Journal of Symbolic Logic*, 43.

[74] S. C. Kleene, 'Origins of Recursive Function Theory', *Annals of the History of Computing*, 3 (1981), 52–67 (59, 61).

[75] H. Wang, *From Mathematics to Philosophy* (New York: Humanities Press, 1974), 85.

[76] K. Gödel, *Collected Works*, ed. S. Feferman et al., vol. iii (Oxford: Oxford University Press, 1995), 304, 168.

[77] Ibid., vol. ii. (Oxford: Oxford University Press, 1990), 150.

[78] Church, 'A Note on the Entscheidungsproblem', 41.

The decision problem was brought to the fore of mathematics by the German mathematician David Hilbert (who in a lecture given in Paris in 1900 set the agenda for much of twentieth-century mathematics). In 1928 Hilbert described the decision problem as 'the main problem of mathematical logic', saying that 'the discovery of a general decision procedure is a very difficult problem which is as yet unsolved', and that the 'solution of the decision problem is of fundamental importance'.[79]

## The Hilbert programme

Hilbert and his followers held that mathematicians should seek to express mathematics in the form of a complete, consistent, decidable formal system—a system expressing 'the whole thought content of mathematics in a uniform way'.[80] Hilbert drew an analogy between such a system and 'a court of arbitration, a supreme tribunal to decide fundamental questions—on a concrete basis on which everyone can agree and where every statement can be controlled'.[81] Such a system would banish ignorance from mathematics: given any mathematical statement, one would be able to tell whether the statement is true or false by determining whether or not it is provable in the system. As Hilbert famously declared in his Paris lecture: 'in mathematics there is no *ignorabimus*' (there is no *we shall not know*).[82]

It is important that the system expressing the 'whole thought content of mathematics' be *consistent*. An inconsistent system—a system containing contradictions—is worthless, since *any* statement whatsoever, true or false, can be derived from a contradiction by simple logical steps.[83] So in an inconsistent

[79] D. Hilbert and W. Ackermann, *Grundzüge der Theoretischen Logik* [Principles of Mathematical Logic] (Berlin: Julius Springer, 1928), 73, 77.

[80] D. Hilbert, 'The Foundations of Mathematics' (English translation of a lecture given in Hamburg in 1927, entitled 'Die Grundlagen der Mathematik'), in J. van Heijenoort (ed.), *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931* (Cambridge, Mass.: Harvard University Press, 1967), 475.

[81] D. Hilbert, 'Über das Unendliche' [On the Infinite], *Mathematische Annalen*, 95 (1926), 161–90 (180); English translation by E. Putnam and G. Massey in R. L. Epstein and W. A. Carnielli, *Computability: Computable Functions, Logic, and the Foundations of Mathematics* (2nd edn. Belmont, Calif.: Wadsworth, 2000).

[82] D. Hilbert, 'Mathematical Problems: Lecture Delivered before the International Congress of Mathematicians at Paris in 1900', *Bulletin of the American Mathematical Society*, 8 (1902), 437–79 (445).

[83] To prove an arbitrary statement from a contradiction *P* & *not P*, one may use the following rules of inference (see further pp. 49–52, below):

(*a*) $not\ P \vdash not\ (P \ \&\ X)$
(*b*) $P \ \&\ not\ (P \ \&\ X) \vdash not\ X.$

Rule (*a*) says: from the statement that it is not the case that *P*, it can be inferred that not *both P* and *X* are the case—i.e. inferred that one at least of *P* and *X* is not the case—where *X* is any statement that you please. Rule (*b*) says: given that *P* is the case and that not *both P* and *X* are the case, it can be inferred that *X* is not the case. Via (*a*), the contradiction '*P* & *not P*' leads to '*not* (*P* & *X*)'; and since the contradiction also offers us *P*, we may then move, via (*b*), to '*not X*'. So we have deduced an arbitrary statement, '*not X*', from the contradiction. (To deduce simply *X*, replace *X* in (*a*) and (*b*) by '*not X*', and at the last step use the rule saying that two negations 'cancel out': *not not* $X \vdash X$.)

system, absurdities such as $0 = 1$ and $6 \neq 6$ are provable. An inconsistent system would indeed contain all true mathematical statements—would be *complete*, in other words—but would in addition also contain all false mathematical statements!

Hilbert's requirement that the system expressing the whole content of mathematics be *decidable* amounts to this: there must be a systematic method for telling, of each mathematical statement, whether or not the statement is provable in the system. If the system is to banish ignorance totally from mathematics then it must be decidable. Only then could we be confident of always being able to tell whether or not any given statement is provable. An undecidable system might sometimes leave us in ignorance.

The project of expressing mathematics in the form of a complete, consistent, decidable formal system became known as 'proof theory' and as the 'Hilbert programme'. In 1928, in a lecture delivered in the Italian city of Bologna, Hilbert said:

In a series of presentations in the course of the last years I have . . . embarked upon a new way of dealing with fundamental questions. With this new foundation of mathematics, which one can conveniently call proof theory, I believe the fundamental questions in mathematics are finally eliminated, by making every mathematical statement a concretely demonstrable and strictly derivable formula . . .

[I]n mathematics there is no *ignorabimus*, rather we are always able to answer meaningful questions; and it is established, as Aristotle perhaps anticipated, that our reason involves no mysterious arts of any kind: rather it proceeds according to formulable rules that are completely definite—and are as well the guarantee of the absolute objectivity of its judgement.[84]

Unfortunately for the Hilbert programme, however, it was soon to become clear that most interesting mathematical systems are, if consistent, *incomplete* and *undecidable*.

In 1931, Gödel showed that Hilbert's ideal is impossible to satisfy, even in the case of simple arithmetic.[85] He proved that the formal system of arithmetic set out by Whitehead and Russell in their seminal *Principia Mathematica*[86] is, if consistent, incomplete. That is to say: if the system is consistent, there are true

[84] D. Hilbert, 'Probleme der Grundlegung der Mathematik' [Problems Concerning the Foundation of Mathematics], *Mathematische Annalen*, 102 (1930), 1–9 (3, 9). Translation by Elisabeth Norcliffe.

[85] K. Gödel, 'Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I.' [On Formally Undecidable Propositions of Principia Mathematica and Related Systems I], *Monatshefte für Mathematik und Physik*, 38 (1931), 173–98. English translation in M. Davis (ed.), *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions* (New York: Raven, 1965), 5–38.

[86] A. N. Whitehead and B. Russell, *Principia Mathematica*, vols. i–iii (Cambridge: Cambridge University Press, 1910–13).

statements of arithmetic that are not provable in the system—the formal system fails to capture the 'whole thought content' of arithmetic. This is known as Gödel's *first incompleteness theorem.*

Gödel later generalized this result, pointing out that 'due to A. M. Turing's work, a precise and unquestionably adequate definition of the general concept of formal system can now be given', with the consequence that incompleteness can 'be proved rigorously for *every* consistent formal system containing a certain amount of finitary number theory'.[87] The definition made possible by Turing's work is this (in Gödel's words): 'A formal system can simply be defined to be any mechanical procedure for producing formulas, called provable formulas.'[88]

In his incompleteness theorem, Gödel had shown that no matter how hard mathematicians might try to construct the all-encompassing formal system envisaged by Hilbert, the product of their labours would, if consistent, inevitably be incomplete. As Hermann Weyl—one of Hilbert's greatest pupils—observed, this was nothing less than 'a catastrophe' for the Hilbert programme.[89]

## *Decidability*

Gödel's theorem left the question of decidability open. As Newman summarized matters:

The Hilbert decision-programme of the 1920's and 30's had for its objective the discovery of a general process . . . for deciding . . . truth or falsehood . . . A first blow was dealt at the prospects of finding this new philosopher's stone by Gödel's incompleteness theorem (1931), which made it clear that truth or falsehood of *A* could not be equated to provability of *A* or not-*A* in any finitely based logic, chosen once for all; but there still remained in principle the possibility of finding a mechanical process for deciding whether *A*, or not-*A*, or neither, was formally provable in a given system.[90]

The question of decidability was tackled head on by Turing and, independently, by Church.

On p. 84 of 'On Computable Numbers' Turing pointed out—by way of a preliminary—a fact that Hilbertians appear to have overlooked: if a system is complete then it follows that it is also decidable. Bernays, Hilbert's close collaborator, had said: 'One observes that [the] requirement of deductive completeness

[87] Gödel, 'Postscriptum', in Davis, *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*, 71–3 (71); the Postscriptum, dated 1964, is to Gödel's 1934 paper 'On Undecidable Propositions of Formal Mathematical Systems' (ibid. 41–71).

[88] Ibid. 72.

[89] H. Weyl, 'David Hilbert and his Mathematical Work', *Bulletin of the American Mathematical Society*, 50 (1944), 612–54 (644).

[90] M. H. A. Newman, 'Alan Mathison Turing, 1912–1954', *Biographical Memoirs of Fellows of the Royal Society*, 1 (1955), 253–63 (256).

does not go as far as the requirement of decidability.'[91] Turing's simple argument on p. 84 shows that there is no conceptual room for the distinction that Bernays is claiming.

Nevertheless, the crucial question was still open: given that in fact simple arithmetic is (if consistent) *in*complete, is it or is it not decidable? Turing and Church both showed that no consistent formal system of arithmetic is decidable. They showed this by proving that not even the *functional calculus*—the weaker, purely logical system presupposed by any formal system of arithmetic—is decidable. The Hilbertian dream of a completely mechanized mathematics now lay in total ruin.

## *A tutorial on first-order predicate calculus*

What Turing called the functional calculus (and Church, following Hilbert, the *engere Funktionenkalkül*) is today known as *first-order predicate calculus* (FOPC). FOPC is a formalization of deductive logical reasoning.

There are various different but equivalent ways of formulating FOPC. One formulation presents FOPC as consisting of about a dozen formal rules of inference. (This formulation, which is more accessible than the Hilbert–Ackermann formulation mentioned by Turing on p. 84, is due to Gerhard Gentzen.[92])

The following are examples of formal rules of inference. The symbol '⊢' indicates that the statement following it can be concluded from the statements (or statement) displayed to its left, the premisses.

(i) $X$, if $X$ then $Y \vdash Y$
(ii) $X$ and $Y \vdash X$
(iii) $X,\ Y \vdash X$ and $Y$

So if, for example, '*X*' represents 'It is sunny' and '*Y*' represents 'We will go for a picnic', (i) says:

'We will go for a picnic' can be concluded from the premisses 'It is sunny' and 'If it is sunny then we will go for a picnic'.

(ii) says:

'It is sunny' can be concluded from the conjunctive premiss 'It is sunny and we will go for a picnic'.

Turing uses the symbol '→' to abbreviate 'if then' and the symbol '&' to abbreviate 'and'. Using this notation, (i)–(iii) are written:

[91] P. Bernays, 'Die Philosophie der Mathematik und die Hilbertsche Beweistheorie' [The Philosophy of Mathematics and Hilbert's Proof Theory], *Blätter für Deutsche Philosophie*, 4 (1930/1931), 326–67. See also H. Wang, *Reflections on Kurt Gödel* (Cambridge, Mass.: MIT Press, 1987), 87–8.

[92] G. Gentzen, 'Investigations into Logical Deduction' (1934), in *The Collected Papers of Gerhard Gentzen*, ed. M. E. Szabo (Amsterdam: North-Holland, 1969).

(i) $X,\ X \rightarrow Y \vdash Y$
(ii) $X \,\&\, Y \vdash X$
(iii) $X,\ Y \vdash X \,\&\, Y$

Some more rules of the formal calculus are as follows. *a* represents any object, *F* represents any property:

(iv) *a* has property $F \vdash$ there is an object that has property *F*
(v) each object has property $F \vdash a$ has property *F*

In Turing's notation, in which '*a* has property *F*' is abbreviated '$F(a)$', these are written:

(iv) $F(a) \vdash (\exists x)F(x)$
(v) $(x)F(x) \vdash F(a)$

'$(\exists x)$' is read: 'there is an object (call it *x*) which . . .'. So '$(\exists x)F(x)$' says 'there is an object, call it *x*, which has property *F*'. '$(x)$' is read: 'each object, *x*, is such that . . .'. So '$(x)F(x)$' says 'each object, *x*, is such that *x* has property *F*'.

Set out in full, FOPC contains not only rules like (i)–(v) but also several rules leading from statements containing '$\vdash$' to other statements containing '$\vdash$'. One such rule is the so-called 'cut rule', used in moving from lines (2) and (3) to (4) in the proof below.

Turing calls '$(\exists x)$' and '$(x)$' *quantors*; the modern term is *quantifiers*. A symbol, such as '*F*', that denotes a property is called a *predicate*. Symbols denoting relationships, for example '<' (less than) and '=' (identity), are also classed as predicates. The symbol '*x*' is called a *variable*.

(FOPC is *first-order* in the sense that the quantifiers of the calculus always involve variables that refer to individual objects. In *second-order* predicate calculus, on the other hand, the quantifiers can contain predicates, as in '$(\exists F)$'. The following are examples of second-order quantification: 'Jules and Jim have some properties in common,' 'Each relationship that holds between *a* and *b* also holds between *c* and *d*.')

Using the dozen or so basic rules of FOPC, more complicated rules of inference can be proved as *theorems* ('provable formulas') of FOPC. For example:

*Theorem* $(x)(G(x) \rightarrow H(x)),\ G(a) \vdash (\exists x)H(x)$

This theorem says: 'There is an object that has property *H*' can be concluded from the premisses 'Each object that has property *G* also has property *H*' and '*a* has property G'.

The proof of the theorem is as follows:

(1) $(x)(G(x) \rightarrow H(x)) \vdash G(a) \rightarrow H(a)$ (rule (v))
(2) $G(a),\ (G(a) \rightarrow H(a)) \vdash H(a)$ (rule (i))
(3) $H(a) \vdash (\exists x)H(x)$ (rule (iv))
(4) $G(a),\ (G(a) \rightarrow H(a)) \vdash (\exists x)H(x)$ (from (2) and (3) by the cut rule)

(5) $(x)(G(x) \rightarrow H(x)),\ G(a) \vdash (\exists x)H(x)$
(from (1) and (4) by the cut rule)

The cut rule (or rule of transitivity) says in effect that whatever can be concluded from a statement *Y* (possibly in conjunction with additional premisses *P*) can be concluded from any premiss(es) from which *Y* can be concluded (together with the additional premisses *P*, if any). For example, if $Y \vdash Z$ and $X \vdash Y$, then $X \vdash Z$. In the transition from (1) and (4) to (5), the additional premiss $G(a)$ in (4) is gathered up and placed among the premisses of (5).

So far we have seen how to prove further inference rules in FOPC. Often logicians are interested in proving not inference rules but single statements unbroken by commas and '$\vdash$'. An example is the complex statement

$$not\,(F(a)\ \&\ not\,(\exists x)F(x)),$$

which says 'It is not the case that *both* $F(a)$ and the denial of $(\exists x)F(x)$ are true'; or in other words, you are not going to find $F(a)$ true without finding $(\exists x)F(x)$ true.

To say that a single statement, as opposed to an inference rule, is provable in FOPC is simply to say that the result of *prefixing* that statement by '$\vdash$' can be derived by using the rules of the calculus. Think of a '$\vdash$' with no statements on its left as indicating that the statement on its right is to be concluded as a matter of 'pure logic'—no premisses are required.

For example, the theorem

$$\vdash not\ (F(a)\ \&\ not\ (\exists x)F(x))$$

can be derived using rule (iv) and the following new rule.[93]

$$\frac{X \vdash Y}{\vdash not\ (X\ \&\ not\ Y)}$$

This rule is read:

If *Y* can be concluded from *X*, then it can be concluded that not *both X* and the denial of *Y* are true.

Much of mathematics and science can be formulated within the framework of FOPC. For example, a formal system of arithmetic can be constructed by adding a number of arithmetical axioms to FOPC. The axioms consist of very basic arithmetical statements, such as:

$$(x)(x + 0 = x)$$

and

$$(x)(y)(Sx = Sy \rightarrow x = y),$$

[93] In Gentzen's system this rule can itself be derived from the basic rules. It should be mentioned that in the full system it is permissible to write any finite number of statements (including zero) on the right hand side of '$\vdash$'.

where 'S' means 'the successor of'—the successor of 1 is 2, and so on. (In these axioms the range of the variables '$x$' and '$y$' is restricted to numbers.) Other arithmetical statements can be derived from these axioms by means of the rules of FOPC. For example, rule (v) tells us that the statement

$$1 + 0 = 1$$

can be concluded from the first of the above axioms.

If FOPC is undecidable then it follows that arithmetic is undecidable. Indeed, if FOPC is undecidable, then so are very many important mathematical systems. To find decidable logics one must search among systems that are in a certain sense *weaker* than FOPC. One example of a decidable logic is the system that results if all the quantifier rules—rules such as (iv) and (v)—are elided from FOPC. This system is known as the *propositional calculus.*

## *The proof of the undecidability of FOPC*

Turing and Church showed that there is no systematic method by which, given any formula Q in the notation of FOPC, it can be determined whether or not Q is provable in the system (i.e. whether or not $\vdash$ Q). To put this another way, Church and Turing showed that the *Entscheidungsproblem* is *unsolvable* in the case of FOPC.

Both published this result in 1936.[94] Church's demonstration of undecidability proceeded via his lambda calculus and his thesis that to each effective method there corresponds a lambda-definable function. There is general agreement that Turing was correct in his view, mentioned above (p. 45), that his own way of showing undecidability is 'more convincing' than Church's.

Turing's method makes use of his proof that no computing machine can solve the printing problem. He showed that if a Turing machine could tell, of any given statement, whether or not the statement is provable in FOPC, then a Turing machine could tell, of any given Turing machine, whether or not it ever prints '0'. Since, as he had already established, no Turing machine can do the latter, it follows that no Turing machine can do the former. The final step of the argument is to apply Turing's thesis: if no Turing machine can perform the task in question, then there is no systematic method for performing it.

[94] In a lecture given in April 1935—the text of which was printed the following year as 'An Unsolvable Problem of Elementary Number Theory' (a short 'Preliminary report' dated 22 Mar. 1935 having appeared in the *Bulletin of the American Mathematical Society* (41 (1935), 332–3) )—Church proved the undecidability of a system that includes FOPC as a part. This system is known as *Principia Mathematica*, or PM, after the treatise in which it was first set out (see n. 86). PM is obtained by adding mathematical axioms to FOPC. Church established the conditional result that if PM is *omega-consistent*, then PM is undecidable. Omega-consistency (first defined by Gödel) is a stronger property than consistency, in the sense that a consistent system is not necessarily omega-consistent. As explained above, a system is consistent when there is no statement $S$ such that both $S$ and *not*-$S$ are provable in the system. A system is omega-consistent when there is no predicate $F$ of integers such that all the following are provable in the system: $(\exists x)F(x)$, *not*-$F(1)$, *not*-$F(2)$, *not*-$F(3)$, and so on, for every integer. In his later paper 'A Note on the Entscheidungsproblem' (completed in April 1936) Church improved on this earlier result, showing unconditionally that FOPC is undecidable.

In detail, Turing's demonstration contains the following steps.

1. Turing shows how to construct, for any computing machine **m**, a complicated statement of FOPC that says 'at some point, machine **m** prints 0'. He calls this formula 'Un(**m**)'. (The letters 'Un' probably come from 'undecidable' or the German equivalent 'unentscheidbare'.)
2. Turing proves the following:
   (*a*) If Un(**m**) is provable in FOPC, then at some point **m** prints 0.
   (*b*) If at some point **m** prints 0, then Un(**m**) is provable in FOPC.
3. Imagine a computing machine which, when given any statement Q in the notation of FOPC, is able to determine (in some finite number of steps) whether or not Q is provable in FOPC. Let's call this machine HILBERT'S DREAM. 2(*a*) and 2(*b*) tell us that HILBERT'S DREAM would solve the printing problem. Because if the machine were to indicate that Un(**m**) is provable then, in view of 2(*a*), it would in effect be indicating that **m** does print 0; and if the machine were to indicate that the statement Un(**m**) is not provable then, in view of 2(*b*), it would in effect be indicating that **m** does not print 0. Since no computing machine can solve the printing problem, it follows that HILBERT'S DREAM is a figment. *No computing machine is able to determine in some finite number of steps, of each statement Q, whether or not Q is provable in FOPC.*
4. If there were a systematic method by which, given any statement Q, it can be determined whether or not Q is provable in FOPC, then it would follow, by Turing's thesis, that there is such a computing machine as HILBERT'S DREAM. Therefore there is no such systematic method.

## *The significance of undecidability*

Poor news though the unsolvability of the *Entscheidungsproblem* was for the Hilbert school, it was very welcome news in other quarters, for a reason that Hilbert's illustrious pupil von Neumann had given in 1927:

If undecidability were to fail then mathematics, in today's sense, would cease to exist; its place would be taken by a completely mechanical rule, with the aid of which any man would be able to decide, of any given statement, whether the statement can be proven or not.[95]

As the Cambridge mathematician G. H. Hardy said in a lecture in 1928: 'if there were . . . a mechanical set of rules for the solution of all mathematical problems . . . our activities as mathematicians would come to an end.'[96]

[95] J. von Neumann, 'Zur Hilbertschen Beweistheorie' [On Hilbert's Proof Theory], *Mathematische Zeitschrift*, 26 (1927), 1–46 (12); reprinted in vol. i of von Neumann's *Collected Works*, ed. A. H. Taub (Oxford: Pergamon Press, 1961).

[96] G. H. Hardy, 'Mathematical Proof', *Mind*, 38 (1929), 1–25 (16) (the text of Hardy's 1928 Rouse Ball Lecture).

## Further reading

Barwise, J., and Etchemendy, J., *Turing's World: An Introduction to Computability Theory* (Stanford, Calif.: CSLI, 1993). (Includes software for building and displaying Turing machines.)

Boolos, G. S., and Jeffrey, R. C., *Computability and Logic* (Cambridge: Cambridge University Press, 2nd edn. 1980).

Copeland, B. J., 'Colossus and the Dawning of the Computer Age', in R. Erskine and M. Smith (eds.), *Action This Day* (London: Bantam, 2001).

Epstein, R. L., and Carnielli, W. A., *Computability: Computable Functions, Logic, and the Foundations of Mathematics* (Belmont, Calif.: Wadsworth, 2nd edn. 2000).

Hopcroft, J. E., and Ullman, J. D., *Introduction to Automata Theory, Languages, and Computation* (Reading, Mass.: Addison-Wesley, 1979).

Minsky, M. L., *Computation: Finite and Infinite Machines* (Englewood Cliffs, NJ: Prentice-Hall, 1967).

Sieg, W., 'Hilbert's Programs: 1917–1922', *Bulletin of Symbolic Logic*, 5 (1999), 1–44.

Sipser, M., *Introduction to the Theory of Computation* (Boston: PWS, 1997).

## Appendix

### *Subroutines and M-Functions*[97]

Section 3 of this guide gave a brief introduction to the concept of a *skeleton table*, where names of subroutines are employed in place of letters referring to states of the machine. This appendix explains the associated idea of an *m-function*, introduced by Turing on p. 63. *m*-functions are subroutines with *parameters*—values that are plugged into the subroutine before it is used.

The example of the 'find' subroutine f makes this idea clear. The subroutine f$(A, B, x)$ is defined in Section 3 (Tables 2 and 3). Recall that f$(A, B, x)$ finds the leftmost $x$ on the tape and places the machine in $A$, leaving the scanner resting on the $x$; or if no $x$ is found, places the machine in $B$ and leaves the scanner resting on a blank square to the right of the used portion of the tape. '$A$', '$B$', and '$x$' are the parameters of the subroutine. Parameter '$x$' may be replaced by any symbol (of the Turing machine in question). Parameters '$A$' and '$B$' may be replaced by names of states of the machine. Alternatively, Turing permits '$A$' and '$B$' (one or both) to be replaced by a name of a *subroutine*. For example, replacing '$A$' by the subroutine name 'e$_1(C)$' produces:

$$\mathrm{f}(\mathrm{e}_1(C),\ B,\ x)$$

This says: find the leftmost $x$, let the scanner rest on it, and go into subroutine e$_1(C)$; or, if there is no $x$, go into $B$ (leaving the scanner resting on a blank square to the right of the used portion of the tape).

The subroutine e$_1(C)$ simply erases the scanned square and places the machine in $C$, leaving the scanner resting on the square that has just been erased. ('$C$' is another parameter of the same type as '$A$' and '$B$'.) Thus the subroutine f(e$_1(C)$, $B$, $x$) finds

[97] By Andrés Sicard and Jack Copeland.

the leftmost occurrence of the symbol $x$ and erases it, placing the machine in $C$ and leaving the scanner resting on the square that has just been erased (or if no $x$ is found, leaves the scanner resting on a blank square to the right of the used portion of the tape and places the machine in $B$). Since in this case nothing turns on the choice of letter, the name of the subroutine may also be written 'f(e$_1(A)$, $B$, $x$)'.

The subroutine f(e$_1(A)$, $B$, $x$) is one and the same as the subroutine e$(A, B, x)$ (Section 3). The new notation exhibits the structure of the subroutine.

More examples of *m*-functions are given below. While the use of *m*-functions is not strictly necessary for the description of any Turing machine, *m*-functions are very useful in describing large or complex Turing machines. This is because of the possibilities they offer for generalization, reusability, simplification, and modularization. Generalization is achieved because tasks of a similar nature can be done by a single *m*-function, and modularization because a complex task can be divided into several simpler *m*-functions. Simplification is obtained because the language of *m*-functions submerges some of the detail of the language of instruction-words—i.e. words of the form $q_iS_jS_kMq_l$—so producing transparent descriptions of Turing machines. Reusability arises simply because we can employ the same *m*-function in different Turing machines.

Although it is difficult (if not impossible) to indicate the exact role that Turing's concept of an *m*-function played in the development of today's programming languages, it is worth emphasizing that some characteristics of *m*-functions are present in the subroutines of almost all modern languages. Full use was made of the idea of parametrized subroutines by Turing and his group at the National Physical Laboratory as they pioneered the science of computer programming during 1946. A contemporary report (by Huskey) outlining Turing's approach to programming said the following:

> The fact that repetition of subroutines require[s] large numbers of orders has led to the abbreviated code methods whereby not only standard orders are used but special words containing parameters are converted into orders by an interpretation table. The general idea is that these describe the entries to subroutines, the values of certain parameters in the subroutine, how many times the subroutine is to be used, and where to go after the subroutine is finished.[98]

Rather than give a formal definition of an *m*-function we present a series of illustrative examples.

First, some preliminaries. An *alphabet* A is some set of symbols, for example {-, 0, 1, 2}, and a *word* of alphabet A is a finite sequence of non-blank symbols of A. The blank symbol, represented '-', is used to separate different words on the tape and is part of the alphabet, but never occurs within words. The following examples all assume that, at the start of operation, there is a single word $w$ of the alphabet on an otherwise blank tape, with the scanner positioned over any symbol of $w$. The symbols of $w$ are written on adjacent squares, using both E-squares and F-squares, and $w$ is surrounded by blanks (some of the examples require there to be at least one blank in front of $w$ and at least three following $w$).

[98] H. D. Huskey, untitled typescript, National Physical Laboratory, n.d. but *c.* Mar. 1947 (in the Woodger Papers, National Museum of Science and Industry, Kensington, London (catalogue reference M12/105); a digital facsimile is in The Turing Archive for the History of Computing <www.AlanTuring.net/huskey_1947>).

Let M be a Turing machine with alphabet A = {-, 0, 1, 2}. The following instructions result in M printing the symbol '1' at the end of *w*, replacing the first blank to the right of *w*:

$$q_1 00Rq_1,\ q_1 11Rq_1,\ q_1 22Rq_1,\ q_1\text{-}1Nq_2$$

The first three instructions move the scanner past the symbols '0', '1', and '2', and once the scanner arrives at the first blank square to the right of *w*, the fourth instruction prints '1' (leaving M in state $q_2$).

If the symbols '3', '4', . . . , '9' are added to the alphabet, so A = {-, 0, 1, ..., 9}, then the necessary instructions for printing '1' at the end of *w* are lengthier:

$$q_1 00Rq_1,\ q_1 11Rq_1,\ \ldots,\ q_1 99Rq_1,\ q_1\text{-}1Nq_2$$

The *m*-function **add**(S, α) defined by Table 4 carries out the task of printing one symbol 'α' at the end of any word *w* of *any* alphabet (assuming as before that the machine starts operating with the scanner positioned over one or another symbol of *w* and that *w* is surrounded by blanks).

Table 4 is the skeleton table for the *m*-function **add**(S, α). (Skeleton tables are like tables of instructions but with some parameters to be replaced by concrete values.) Table 4 has two parameters, 'α' and 'S'. The second parameter 'S' is to be replaced by the state or *m*-function into which the machine is to go once **add**(S, α) completes its operation, and the first parameter 'α' is to be replaced by whatever symbol it is that we wish to be printed at the end of the word.

Both sets of instruction-words shown above can now be replaced by a simple call to the *m*-function **add**(S, α), where S = $q_2$ and α = 1.

If instead of adding '1' at the end of a word from alphabet A = {-, 0, 1, . . . , 9}, we wanted to add a pair of symbols '5' and '4', then the instruction-words would be:

$$q_1 00Rq_1,\ q_1 11Rq_1,\ \ldots,\ q_1 99Rq_1,\ q_1\text{-}5Rq_2,\ q_2\text{-}4Nq_3$$

These instruction-words can be replaced by the *m*-function **add**(**add**($q_3$, 4), 5). This *m*-function finds the end of the word and writes '5', going into *m*-function **add**($q_3$, 4), which writes '4' and ends in state $q_3$.

Another example: suppose that '5' and '4' are to be printed as just described, and then each occurrence of the symbol '3' is to be replaced by '4'. The *m*-function **add**(**add**(**change**($q_n$, 3, 4), 4), 5) carries out the required task, where the *m*-function **change**(S, α, β) is defined by Table 5. The *m*-function **change**$_1$(S, α, β) is a subroutine inside the *m*-function **change**(S, α, β).

*m*-functions can employ *internal variables.* Although internal variables are not strictly necessary, they simplify an *m*-function's description. Internal variables are not parameters of the *m*-function—we do not need to replace them with concrete values before the *m*-function is used. In the following example, the internal variable 'δ' refers to whatever symbol is present on the scanned square when the machine enters the *m*-function **repeat**$_1$(S).

Suppose we wish to print a repetition of the first symbol of *w* at the end of *w*. This can be achieved by the *m*-function **repeat**(S) defined by Table 5. (The *m*-function **add**(S, δ) is as given by Table 4.)

Every *m*-function has the form: **name**($S_1$, $S_2$, . . . , $S_n$, $\alpha_1$, $\alpha_2$, . . . , $\alpha_m$), where $S_1$, $S_2$, . . . , $S_n$ refer either to states or to *m*-functions, and $\alpha_1$, $\alpha_2$, . . . , $\alpha_m$ denote symbols. Each *m*-function is a Turing machine with parameters. To convert an *m*-function's

**Table 4**

| State | Scanned Square | Operations | Next State |
|---|---|---|---|
| **add**(S, α) | not - | R | **add**(S, α) |
| **add**(S, α) | - | P[α] | S |

**Table 5**

| State | Scanned Square | Operations | Next State |
|---|---|---|---|
| **change**(S, α, β) | not - | L | **change**(S, α, β) |
| **change**(S, α, β) | - | R | **change**$_1$(S, α, β) |
| **change**$_1$(S, α, β) | α | P[β], R | **change**$_1$(S, α, β) |
| **change**$_1$(S, α, β) | not α | R | **change**$_1$(S, α, β) |
| **change**$_1$(S, α, β) | - | L | S |

**Table 6**

| State | Scanned Square | Operations | Next State |
|---|---|---|---|
| **repeat**(S) | not - | L | **repeat**(S) |
| **repeat**(S) | - | R | **repeat**$_1$(S) |
| **repeat**$_1$(S) | δ | | **add**(S, δ) |

skeleton table to a Turing-machine instruction table, where each row is an instruction-word of the form $q_iS_jS_kMq_l$, it is necessary to know the context in which the *m*-function is to be used, namely, the underlying Turing machine's alphabet and states. It is necessary to know the alphabet because of the use in skeleton tables of expressions such as 'does not contain !', 'not α', 'neither α nor -', 'any'. Knowledge of the underlying machine's states is necessary to ensure that the *m*-function begins and ends in the correct state.

The economy effected by *m*-functions is illustrated by the fact that if the *m*-functions are eliminated from Turing's description of his universal machine, nearly 4,000 instruction-words are required in their place.[99]

[99] A. Sicard, 'Máquinas de Turing dinámicas: historia y desarrollo de una idea' [Dynamic Turing Machines: Story and Development of an Idea], appendix 3 (Master's thesis, Universidad EAFIT, 1998); 'Máquina universal de Turing: algunas indicaciones para su construcción' [The Universal Turing Machine: Some Directions for its Construction], *Revista Universidad EAFIT*, vol. 108 (1998), pp. 61–106.