

Chapter 1

The Computability Concept

1 The informal concept

Decidable sets

Computability theory, also known as recursion theory, is the area of mathematics dealing with the concept of an *effective procedure*—a procedure that can be carried out by following specific rules. For example, we might ask whether there is some effective procedure—some algorithm—that, given a sentence about the positive integers, will decide whether that sentence is true or false. In other words, is the set of true sentences about the positive integers *decidable*? (We will see later that the answer is negative.) Or for a simpler example, the set of prime numbers is certainly a decidable set. That is, there are quite mechanical procedures, which are taught in the schools, for deciding of any given positive integer whether or not it is a prime number. (For a very large number, the procedure taught in the schools might take a long time.) If we want, we can write a computer program to execute the procedure. Simpler still, the set of even positive integers is decidable. We can write a computer program that, given a positive integer, will very quickly decide whether or not it is even. Our goal is to study what decision problems can be solved (in principle) by a computer program, and what decision problems (if any) cannot.

More generally, consider a set S of natural numbers. (The natural numbers are $0, 1, 2, \dots$. In particular, 0 is natural.) We say that S is a *decidable* set if there exists an effective procedure that, given any natural number, will eventually end by supplying us with the answer: “Yes” if the given number is a member of S and “No” if it is not a member of S .

(Initially, we are going to examine computability in the context of the natural numbers. Later, we will see that computability concepts can be readily transferred to the context of strings of letters from a finite alphabet. In that context, we can consider a set S of strings, such as the set of equations, like $x(y+z) = xy+xz$, that hold in the algebra of real numbers. But to start with, we will consider sets of natural numbers.)

And by an *effective procedure* here is meant a procedure for which we can give exact instructions—a program—for carrying out the procedure. Following these instructions should not demand brilliant insights on the part of the agent (human or machine) following them. It must be possible, at least in principle, to make the instructions so explicit that they can be executed by a diligent clerk (who is very good at following directions but is not too clever) or even a machine (which does not think at all). That is, it must be possible for our instructions to be *mechanically implemented*. (One might imagine a mathematician so brilliant that he or she can look at any sentence of arithmetic and say whether it is true or false. But you cannot ask the clerk to do this. And there is no computer

program to do this. It is not merely that we have not succeeded in writing such a program. We can actually prove that such a program cannot possibly exist!)

Although these instructions must of course be finite in length, we impose no upper bound on their possible length. We do not rule out the possibility that the instructions might even be absurdly long. (If the number of lines in the instructions exceeds the number of electrons in the universe, we merely shrug and say, “That’s a pretty long program.”) We insist only that the instructions—the program—be finitely long, so that we can *communicate* them to the person or machine doing the calculations. (There is no way to give someone all of an infinite object.) Similarly, in order to obtain the most comprehensive concepts, we impose no bounds on the time that the procedure might consume before it supplies us with the answer. Nor do we impose a bound on the amount of storage space (scratch paper) that the procedure might need to use. (The procedure might, for example, need to utilize very large numbers requiring a substantial amount of space simply to write down.) We merely insist that the procedure give us the answer eventually, in some finite length of time. What is definitely ruled out is doing infinitely many steps and *then* giving the answer.

In Chapter 7, we will consider more restrictive concepts, where the amount of time is limited in some way, so as to exclude the possibility of ridiculously long execution times. But initially we want to avoid such restrictions, to obtain the limiting case where practical limitations on execution time or memory space are removed. It is well known that in the real world, the speed and capability of computers has been steadily growing. We want to ignore actual speed and actual capability, and instead to ask what the purely theoretical limits are.

The foregoing description of effective procedures is admittedly vague and imprecise. In the following section, we will look at how this vague description can be made precise—how the concept can be made into a *mathematical* concept. Nonetheless, the informal idea of what can be done by effective procedure, that is, what is *calculable*, can be very useful. Rigor and precision can wait until the *next* chapter. First we need a sense of where we are going.

For example, any finite set of natural numbers must be decidable. The program for the decision procedure can simply include a list of all the numbers in the set. Then given a number, the program can check it against the list. Thus the concept of decidability is interesting only for infinite sets.

Our description of effective procedures, vague as it is, already shows how limiting the concept of decidability is. One can, for example, utilize the concepts of countable and uncountable sets (see the appendix for a summary of these concepts). It is not hard to see that there are only countably many possible instructions of finite length that one can write out (using a standard keyboard, say). But there are uncountably many sets of natural numbers (by Cantor’s diagonal argument). It follows that almost all sets, in a sense, are *undecidable*.

The fact that not every set is decidable is relevant to theoretical computer science. The fact that there is a limit to what can be carried out by effective procedures means there is a limit to what can—even in principle—be done by computer programs. And this raises the questions: What can be done? What cannot?

Historically, computability theory arose before the development of computers, from considerations in mathematical logic. At the heart of mathematical activity is the proving of theorems. Consider what is required for a string of symbols to constitute an “acceptable mathematical proof.” Before we accept a proof, and add the result being proved to our storehouse of mathematical knowledge, we insist that the proof be *verifiable*. That is, it should be possible for another mathematician, such as the referee of the paper containing the proof, to check, step by step, the correctness of the proof. Eventually the referee either concludes that the proof is indeed correct, or concludes that the proof contains a gap or an error and is not yet acceptable. That is, the set of acceptable mathematical proofs—regarded as strings of symbols—should be *decidable*. This fact will be seen (in a later chapter) to have significant consequences for what can and cannot be proved. We conclude that computability theory is relevant to the foundations of mathematics. But if logicians had not invented the computability concept, then computer scientists would later have done so.

Calculable functions

Before going on, we should broaden the canvas from considering decidable and undecidable sets to considering the more general situation of *partial functions*. Let $\mathbb{N} = \{0, 1, 2, \dots\}$ be the set of natural numbers. Then an example of a 2-place function on \mathbb{N} is the subtraction function

$$g(m, n) = \begin{cases} m - n & \text{if } m \geq n \\ 0 & \text{otherwise} \end{cases}$$

(where we have avoided negative numbers). A different subtraction function is the “partial” function

$$f(m, n) = \begin{cases} m - n & \text{if } m \geq n \\ \uparrow & \text{otherwise} \end{cases}$$

where “ \uparrow ” indicates that the function is undefined. Thus $f(5, 2) = 3$, but $f(2, 5)$ is undefined; the pair $\langle 2, 5 \rangle$ is not in the domain of f .

In general, say that a k -place *partial function* on \mathbb{N} is a function whose domain is some set of k -tuples of natural numbers and whose values are natural numbers. In other words, for a k -place partial function f and a k -tuple $\langle x_1, \dots, x_k \rangle$, possibly $f(x_1, \dots, x_k)$ is defined (i.e., $\langle x_1, \dots, x_k \rangle$ is in the domain of f) in which case the function value $f(x_1, \dots, x_k)$ is in \mathbb{N} , and possibly $f(x_1, \dots, x_k)$ is undefined (i.e., $\langle x_1, \dots, x_k \rangle$ is not in the domain of f).

At one extreme, there are partial functions whose domains are the set \mathbb{N}^k of *all* k -tuples; such functions are said to be *total*. (The adjective “partial” covers both the total and the non-total functions.) At the other extreme, there is the empty function, that is, the function that is defined nowhere. The empty function might not seem particularly useful, but it does count as one of the k -place partial functions.

For a k -place partial function f , we say that f is an *effectively calculable partial function* if there exists an effective procedure with the following property:

- Given a k -tuple \vec{x} in the domain of f , the procedure eventually halts and returns the correct value for $f(\vec{x})$.
- Given a k -tuple \vec{x} *not* in the domain of f , the procedure does not halt and return a value.

(There is one issue here: How can a number be *given*? To communicate a number x to the procedure, we send it the *numeral* for x . Numerals are bits of language, which can be communicated. Numbers are not. Communication requires language. Nonetheless, we will continue to speak of being “given numbers m and n ” and so forth. But at a few points, we will need to be more accurate and to take account of the fact that what the procedure is given are numerals. There was a time in the 1960’s when, as part of the “new math,” schoolteachers were encouraged to distinguish carefully between numbers and numerals. This was a good idea that turned out not to work.)

For example, the partial function for subtraction

$$f(m, n) = \begin{cases} m - n & \text{if } m \geq n \\ \uparrow & \text{otherwise} \end{cases}$$

is effectively calculable, and procedures for calculating it, using base-10 numerals, are taught in the elementary schools.

The empty function is effectively calculable. The effective procedure for it, given a k -tuple, does not need to do anything in particular. But it must never halt and return a value.

The concept of decidability can then be described in terms of functions: For a subset S of \mathbb{N}^k , we can say that S is *decidable* iff its characteristic function

$$C_S(\vec{x}) = \begin{cases} \text{Yes} & \text{if } \vec{x} \in S \\ \text{No} & \text{if } \vec{x} \notin S \end{cases}$$

(which is always total) is effectively calculable. Here “Yes” and “No” are some fixed members of \mathbb{N} , such as 1 and 0.

(That word “iff” in the preceding paragraph means “if and only if.” This is a bit of mathematical jargon that has proved to be so useful that it has become a standard part of mathspeak.)

Here if $k = 1$ then S is a set of numbers. If $k = 2$ then we have the concept of a decidable binary relation on numbers, and so forth. Take for example the divisibility relation, that is, the set of pairs $\langle m, n \rangle$ such that m divides n evenly. (For definiteness, assume that 0 divides only itself.) The divisibility relation is decidable, because given m and n , we can carry out the division algorithm we all learned in the fourth grade, and see whether the remainder is 0 or not.

Example: Any total constant function on \mathbb{N} is effectively computable. Suppose, for example, $f(x) = 36$ for all x in \mathbb{N} . There is an obvious procedure for

calculating f ; it ignores its input and writes “36” as the output. This may seem a triviality, but compare it with the next example.

Example: Define the function F as follows.

$$F(x) = \begin{cases} 1 & \text{if Goldbach's conjecture is true} \\ 0 & \text{if Goldbach's conjecture is false.} \end{cases}$$

Goldbach’s conjecture states that every even integer greater than 2 is the sum of two primes; for example $22 = 5 + 17$. This conjecture is still an open problem in mathematics. Is this function F effectively computable? (Choose your answer before reading the next paragraph.)

Observe that F is a total constant function. (Classical logic enters here: Either there is an even number that serves as a counterexample or there isn’t.) So as noted in the preceding example, F is effectively computable. What, then, is a procedure for computing F ? I don’t know, but I can give you two procedures, and be confident that one of them computes F .

The point of this example is that effective computability is a property of the function itself, not a property of some linguistic description we might give the function. (One says that the effective computability property is *extensional*.) There are many English phrases that would serve to define F . For a function to be effectively computable, there must *exist* (in the mathematical sense) an effective procedure for computing it. That is not the same as saying that you hold such procedure in your hand. If, in the year 2083, some creature in the universe proves (or refutes) Goldbach’s conjecture, that does *not* mean that F will suddenly change from non-computable to computable. It was computable all along.

There will be, however, situations later in which we will want more than the mere existence of an effective procedure P ; we will want some way of actually finding P , given some suitable clues. That is for later.

It is very natural to extend these concepts to the the situation where we have half of decidability: Say that S is *semi-decidable* if its “semi-characteristic function”

$$c_S(\vec{x}) = \begin{cases} \text{Yes} & \text{if } \vec{x} \in S \\ \uparrow & \text{if } \vec{x} \notin S \end{cases}$$

is an effectively calculable partial function. Thus a set S of numbers is semi-decidable if there is an effective procedure for *recognizing* members of S . We can think of S as the set that the procedure *accepts*. And the effective procedure, while it may not be a decision procedure, is at least an *acceptance* procedure.

Any decidable set is also semi-decidable. If we have an effective procedure that calculates the characteristic function C_S , then we can convert it to an effective procedure that calculates the semi-characteristic function c_S . We simply replace each “output No” command by some endless loop. Or more informally, we simply unscrew the No bulb.

What about the converse? Are there semi-decidable sets that are not decidable? We will see that there are indeed. The trouble with the semi-characteristic

function is that it never produces a No answer. Suppose that we have been calculating $c_S(\vec{x})$ for 37 years, and the procedure has not yet terminated. Should we give up and conclude that \vec{x} is not in S ? Or maybe working just another ten minutes would yield the information that \vec{x} does belong to S . There is in general no way to know.

Here is another example of a calculable partial function:

$$F(n) = \text{the smallest } p > n \text{ such that both } p \text{ and } p + 2 \text{ are prime}$$

Here it is to be understood that $F(n)$ is undefined if there is no number p as described; thus F might not be total. For example, $F(9) = 11$, because both 11 and 13 are prime. It is not known whether or not F is total. The “twin prime conjecture,” which says that there are infinitely many pairs of primes that differ by 2, is equivalent to the statement that F is total. The twin prime conjecture is still an open problem. Nonetheless, we can be certain that F is effectively calculable. One procedure for calculating $F(n)$ proceeds as follows. “Given n , first put $p = n + 1$. Then check whether or not p and $p + 2$ are both prime. If they are, then stop and give output p . If not, increment p and continue.” What if n is huge, say $n = 10^{10^{10}}$? On the one hand, if there is a larger prime pair, then this procedure will find the first one, and halt with the correct output. On the other hand, if there is no larger prime pair, then the procedure never halts, so it never gives us an answer. That is all right, because $F(n)$ is undefined—the procedure *should not* give us an answer.

Now suppose we modify this example. Consider the total function:

$$G(n) = \begin{cases} F(n) & \text{if } F(n) \downarrow \\ 0 & \text{otherwise} \end{cases}$$

Here “ $F(n) \downarrow$ ” means that $F(n)$ is defined, so that n belongs to the domain of F . Then the function G is *also* effectively calculable. That is, there *exists* a program that calculates G correctly.

The twin prime conjecture is either true or false: Either the prime pairs go on forever, or there is a largest one. (At this point classical logic enters once again.) In the first case, $F = G$ and the effective procedure for F also computes G . In the second case, G is eventually constantly 0. And any eventually constant function is calculable (the procedure can utilize a table for the finite part of the function before it stabilizes).

So in either case, there *exists* an effective procedure for G . That is not the same as saying that we *know* that procedure. This example indicates once again the difference between knowing that a certain effective procedure exists and having the effective procedure in our hands.

One person’s program is another person’s data. This is the principle behind operating systems (and behind the idea of a stored-program computer). One’s favorite program is, to the operating system, another piece of data to be received as input and processed. The operating system is calculating the values of a two-place “universal” function. We next want to see if these concepts can be applied to our study of computable functions. (Historically, the flow of ideas

was in exactly the opposite direction! The following digression expands on this point.)

Digression. The concept of a general-purpose, stored-program computer is now very common, but the concept developed slowly over a period of time. The ENIAC machine, the most important computer of the 1940's, was programmed by setting switches and inserting cables into plugboards! This is a far cry from treating a program like data. It was von Neumann who in a 1945 technical support laid out the crucial ideas for a general-purpose stored-program computer, that is, for a universal computer. Turing's 1936 paper on what are now called Turing machines, had proved the existence of a "universal Turing machine" to compute the Φ function described below. When Turing went to Princeton in 1936–37, von Neumann was there and must have been aware of his work. Apparently von Neumann's thinking in 1945 was influenced by Turing's work of nearly a decade later.

Suppose we adopt a fixed method of encoding any set of instructions by a single natural number. (First we convert the instructions to a string of 0's and 1's—one always does this with computer programs—and then we regard that string as naming a natural number under a suitable base-2 notation.) Then the "universal function"

$\Phi(w, x)$ = the result of applying the instructions coded by w to the input x

is an effectively calculable partial function (where it is understood that $\Phi(w, x)$ is undefined whenever applying the instructions coded by w to the input x fails to halt and return an output). Here are the instructions for Φ : "Given w and x , decode w to see what it says to do with x , and then do it." Of course, the function Φ is not total. For one thing, when we try to decode w , we might get complete nonsense, so that the instruction "then do it" leads nowhere. And even if decoding w yields explicit and comprehensible instructions, applying those instructions to a particular x might never yield an output.

The two-place partial function Φ is "universal" in the sense that *any* one-place effectively calculable partial function f is given by the equation

$$f(x) = \Phi(e, x) \quad \text{for all } x$$

where e codes the instructions for f . It will be helpful to introduce a special notation here: Let $\llbracket e \rrbracket$ be the one-place partial function defined by the equation

$$\llbracket e \rrbracket(x) = \Phi(e, x).$$

That is, $\llbracket e \rrbracket$ is the partial function whose instructions are coded by e , with the understanding that, because some values of e might not code anything sensible, the function $\llbracket e \rrbracket$ might be the empty function. In any case, $\llbracket e \rrbracket$ is the partial function we get from Φ when we hold its first variable fixed at e . Thus

$$\llbracket 0 \rrbracket, \llbracket 1 \rrbracket, \llbracket 2 \rrbracket, \dots$$

is a complete list of all the 1-place effectively calculable partial functions. The values of $\llbracket e \rrbracket$ are given by the $(e + 1)$ st row in the following table:

$\llbracket 0 \rrbracket$	$\Phi(0, 0)$	$\Phi(0, 1)$	$\Phi(0, 2)$	$\Phi(0, 3)$	\dots
$\llbracket 1 \rrbracket$	$\Phi(1, 0)$	$\Phi(1, 1)$	$\Phi(1, 2)$	$\Phi(1, 3)$	\dots
$\llbracket 2 \rrbracket$	$\Phi(2, 0)$	$\Phi(2, 1)$	$\Phi(2, 2)$	$\Phi(2, 3)$	\dots
$\llbracket 3 \rrbracket$	$\Phi(3, 0)$	$\Phi(3, 1)$	$\Phi(3, 2)$	$\Phi(3, 3)$	\dots
\dots	\dots	\dots	\dots	\dots	\dots

Using the universal partial function Φ , we can construct an *undecidable* binary relation, the *halting* relation H :

$$\begin{aligned} \langle w, x \rangle \in H &\iff \Phi(w, x) \downarrow \\ &\iff \text{applying the instructions coded by } w \text{ to input } x \text{ halts} \end{aligned}$$

On the positive side, H is semi-decidable. To calculate the semi-characteristic function $c_H(w, x)$, given w and x , we first calculate $\Phi(w, x)$. If and when this halts and returns a value, we give output “Yes” and stop.

On the negative side, H is not decidable. To see this, first consider the following partial function:

$$f(x) = \begin{cases} \text{Yes} & \text{if } \Phi(x, x) \uparrow \\ \uparrow & \text{if } \Phi(x, x) \downarrow \end{cases}$$

(Notice that we are employing the classical diagonal construction. Looking at the earlier table of the values of Φ arranged in a two-dimensional array, one sees that f has been made by going along the diagonal of that table, taking the entry $\Phi(x, x)$ found there, and making sure that $f(x)$ differs from it.)

There are two things to be said about f . First, f cannot possibly be effectively calculable. Consider any set of instructions that *might* compute f . Those instructions have some code number k and hence compute the partial function $\llbracket k \rrbracket$. Could that be the same as f ? No, f and $\llbracket k \rrbracket$ differ at the input k . That is, f has been constructed in such a way that $f(k)$ differs from $\llbracket k \rrbracket(k)$; they differ because one is defined and the other is not. So these instructions cannot correctly compute f ; they produce the wrong result at the input k . And because k was arbitrary, we are forced to conclude that *no* set of instructions can correctly compute f . (This is our first example of a partial function that is not effectively calculable. There are a great many more, as will be seen.)

Secondly, we can argue that *if* we had a decision procedure for H , then we *could* calculate f . To compute $f(x)$, we first use that decision procedure for H to decide if $(x, x) \in H$ or not. If not, then $f(x) = \text{Yes}$. But if $(x, x) \in H$, then the procedure for finding $f(x)$ should throw itself into an infinite loop, because $f(x)$ is undefined.

Putting these two observations about f together, we conclude that there can be no decision procedure for H . The fact that H is undecidable is usually expressed by saying that “the halting problem is unsolvable”; i.e., we cannot effectively determine, given w and x , whether applying the instructions coded by w to the input x will eventually terminate or will go on forever:

Unsolvability of the halting problem: The relation

$$\{\langle w, x \rangle \mid \text{applying instructions coded by } w \text{ to input } x \text{ halts}\}$$

is semi-decidable but not decidable.

The function f in the preceding argument

$$f(x) = \begin{cases} \text{Yes} & \text{if } \Phi(x, x) \uparrow \\ \uparrow & \text{if } \Phi(x, x) \downarrow \end{cases}$$

is the semi-characteristic function of the set $\{x \mid \Phi(x, x) \uparrow\}$. Because its semi-characteristic function is not effectively calculable, we can conclude that this set is *not* semi-decidable.

Let K be the complement of this set:

$$K = \{x \mid \Phi(x, x) \downarrow\} = \{x \mid \llbracket x \rrbracket(x) \downarrow\}.$$

This set *is* semi-decidable. How might we compute $c_K(x)$, given x ? We try to compute $\Phi(x, x)$ (which is possible, because Φ is an effectively calculable partial function. If and when the calculation halts and returns an output, we give the output “Yes” and stop. Until such time, we keep trying. (This argument is the same one we saw for the semidecidability of H . And $x \in K \Leftrightarrow \langle x, x \rangle \in H$.)

Kleene’s theorem: A set (or a relation) is decidable if and only if both it and its complement are semi-decidable.

Here if we are working with sets of numbers, then the complement is with respect to \mathbb{N} ; if we are working with a k -ary relation then the complement is with respect to \mathbb{N}^k .

Proof: On the one hand, if a set S is decidable, then its complement \bar{S} is also decidable—we simply switch the “Yes” and the “No.” So both S and its complement \bar{S} are semi-decidable, because decidable sets are also semi-decidable.

On the other hand, suppose that S is a set for which both c_S and $c_{\bar{S}}$ are effectively calculable. The idea is to glue together these two halves of a decision procedure to make a whole one. Say we want to find $C_S(x)$, given x . We need to organize our time. During odd-numbered minutes, we run our program for $c_S(x)$. During even-numbered minutes, we run our program for $c_{\bar{S}}(x)$. Of course, at the end of each minute we store away what we have done, so that we can later pick up where we left off.

Eventually we must receive a “Yes.” If during an odd-numbered minute, we find that $c_S(x) = \text{Yes}$ (this must happen eventually if $x \in S$) then we give output “Yes” and stop. And if during an even-numbered minute, we find that $c_{\bar{S}}(x) = \text{Yes}$ (this must happen eventually if $x \notin S$) then we give output “No” and stop.

(Alternatively, we can imagine working ambidextrously. With the left hand, we work on calculating $c_S(x)$; with the right hand we work on $c_{\bar{S}}(x)$. Eventually one hand discovers the answer.) \dashv

The set K is an example of a semi-decidable set that is not decidable. Its complement \bar{K} is not semi-decidable; we have seen that its semi-characteristic function f is not effectively calculable.

The connection between effectively calculable partial functions and semi-decidable sets can be further described as follows:

Theorem: (i) A relation is semi-decidable if and only if it is the domain of some effectively calculable partial function.

(ii) A partial function f is an effectively calculable partial function if and only if its graph G (i.e., the set of pairs $\langle \vec{x}, y \rangle$ such that $f(\vec{x}) = y$) is a semi-decidable relation.

For statement (i), one direction is true by definition: Any relation is the domain of its semi-characteristic function, and for a semi-decidable relation, that function is an effectively calculable partial function.

Conversely, for an effectively calculable partial function f we have the natural semi-decision procedure for its domain: Given \vec{x} , we try to compute $f(\vec{x})$. If and when we succeed in finding $f(\vec{x})$, we ignore the value and simply say Yes and halt.

To prove (ii) in one direction, suppose that f is an effectively calculable partial function. Here is a semi-decision procedure for its graph G : Given $\langle \vec{x}, y \rangle$, we proceed to compute $f(\vec{x})$. If and when we obtain the result, we check to see whether it is y or not. If the result is indeed y , then we say Yes and halt.

Of course this procedure fails to give an answer if $f(\vec{x}) \uparrow$, which is exactly as it should be, because $\langle \vec{x}, y \rangle$ is not in the graph.

To prove the other direction of (ii), suppose that we have a semi-decision procedure for the graph G . We seek to compute, given \vec{x} , the value $f(\vec{x})$, if this is defined. Our plan is to check $\langle \vec{x}, 0 \rangle$, $\langle \vec{x}, 1 \rangle$, \dots , for membership in G . But to budget our time sensibly, we use a procedure called “dovetailing.” Here is what we do:

- (1) Spend one minute testing whether $\langle \vec{x}, 0 \rangle \in G$.
- (2) Spend two minutes testing whether $\langle \vec{x}, 0 \rangle \in G$ and two minutes testing whether $\langle \vec{x}, 1 \rangle \in G$.
- (3) Similarly, spend three minutes on each of $\langle \vec{x}, 0 \rangle$, $\langle \vec{x}, 1 \rangle$, $\langle \vec{x}, 2 \rangle$.

And so forth. If and when we discover that, in fact, $\langle \vec{x}, k \rangle \in G$, then we return the value k and halt. Observe that whenever $f(\vec{x}) \downarrow$, then sooner or later the foregoing procedure will correctly determine $f(\vec{x})$ and halt. Of course, if $f(\vec{x}) \uparrow$, then the procedure runs forever. \dashv

Church’s thesis

While the concept of effective calculability has here been described in somewhat vague terms, the following section will describe a precise (mathematical) concept of a “computable partial function.” In fact, it will describe several equivalent ways of formulating the concept in precise terms. And it will be argued that

the mathematical concept of a computable partial function is the *correct* formalization of the informal concept of an effectively calculable partial function. This claim is known as *Church's thesis* or the *Church–Turing thesis*.

Church's thesis, which relates an informal idea to a formal idea, is not itself a mathematical statement, capable of being given a proof. But one can look for evidence for or against Church's thesis; it all turns out to be evidence in favor.

One piece of evidence is the absence of counterexamples. That is, any function examined thus far that mathematicians have felt was effectively calculable, has been found to be computable.

Stronger evidence stems from the various attempts that different people made independently, trying to formalize the idea of effective calculability. Alonzo Church used λ -calculus; Alan Turing used an idealized computing agent (later called a Turing machine); Emil Post developed a similar approach. Remarkably, all these attempts turned out to be equivalent, in that they all defined exactly the same class of functions, namely the computable partial functions!

The study of effective calculability originated in the 1930's with work in mathematical logic. As noted previously, the subject is related to the concept of an *acceptable proof*. More recently, the study of effective calculability has formed an essential part of theoretical computer science. A prudent computer scientist would surely want to know that, apart from the difficulties the real world presents, there is a purely theoretical limit to calculability.

Exercises

1. Assume that S is a set of natural numbers containing all but finitely many natural numbers. (That is, S is a *cofinite* subset of \mathbb{N} .) Explain why S must be decidable.

2. Assume that A and B are decidable sets of natural numbers. Explain why their intersection $A \cap B$ is also decidable. (Describe an effective procedure for determining whether or not a given number is in $A \cap B$.)

3. Assume that A and B are decidable sets of natural numbers. Explain why their union $A \cup B$ is also decidable.

4. Assume that A and B are semi-decidable sets of natural numbers. Explain why their intersection $A \cap B$ is also semi-decidable.

5. Assume that A and B are semi-decidable sets of natural numbers. Explain why their union $A \cup B$ is also semi-decidable.

6. (a) Assume that R is a decidable binary relation on the natural numbers. That is, it is a decidable 2-ary relation. Explain why its domain, $\{x \mid \langle x, y \rangle \in R \text{ for some } y\}$ is a semi-decidable set.

(b) Now suppose that instead of assuming that R is decidable, we assume only that it is semi-decidable. Is it still true that its domain must be semi-decidable?

7. (a) Assume that f is a one-place total calculable function. Explain why its graph is a decidable binary relation.

(b) Conversely, show that if the graph of a one-place total function f is decidable, then f must be calculable.

(c) Suppose that in part (a), we drop the assumption of totality. That is, assume that f is a one-place calculable partial function. Explain why its graph must be a semi-decidable binary relation.

8. Assume that S is a decidable set of natural numbers, and that f is a *total* effectively calculable function on \mathbb{N} . Explain why $\{x \mid f(x) \in S\}$ is decidable. (This set is called the *inverse image* of S under f .)

9. Assume that S is a semi-decidable set of natural numbers, and that f is an effectively calculable partial function on \mathbb{N} . Explain why

$$\{x \mid f(x) \downarrow \text{ and } f(x) \in S\}$$

is semi-decidable.

10. In the decimal expansion of π , there might be a string of many consecutive 7's. Define the function f so that $f(x) = 1$ if there is a string of x or more consecutive 7's and $f(x) = 0$ otherwise:

$$f(x) = \begin{cases} 1 & \text{if } \pi \text{ has a run of } x \text{ or more 7's} \\ 0 & \text{otherwise.} \end{cases}$$

Explain, without using any facts about π or any number theory, why f is effectively computable.

11. Assume that g is a total non-increasing function on \mathbb{N} (that is, $g(x) \geq g(x+1)$ for all x). Explain why g must be effectively computable.

12. Assume that f is a total function on the natural numbers and that f is eventually periodic. That is, there exist positive numbers m and p such that for all x greater than m , we have $f(x+p) = f(x)$. Explain why f is effectively calculable.

13. (a) Assume that f is a total effectively calculable function on the natural numbers. Explain why the range of f (that is, the set $\{f(x) \mid x \in \mathbb{N}\}$) is semi-decidable.

(b) Now suppose f is an effectively calculable partial function (not necessarily total). Is it still true that the range must be semi-decidable?

2. Formalizations—an overview

In the preceding section, the concept of effective calculability was described only very informally. Now we want to make those ideas precise (i.e., make them part of mathematics). In fact, several approaches to doing this will be described: idealized computing devices, generative definitions (i.e., the least class containing certain initial functions and closed under certain constructions), programming languages, and definability in formal languages. It is a significant fact that these very different approaches all yield exactly equivalent concepts.

This section gives a general overview of a number of different (but equivalent) ways of formalizing the concept of effective calculability. Later chapters will develop a few of these ways in full detail.

Digression: The 1967 book by Rogers cited in the References demonstrates that the subject of computability can be developed *without* adopting any of these formalizations. And that book was preceded by a 1956 mimeographed preliminary version, which is where I first saw this subject. A few treasured copies of the mimeographed edition still exist.

Turing machines

In early 1935, Alan Turing was a 22-year-old graduate student at King's College in Cambridge. Under the guidance of Max Newman, he was working on the problem of formalizing the concept of effective calculability. In 1936, he learned of the work of Alonzo Church, at Princeton. Church had also been working on this problem, and in his 1936 paper *An unsolvable problem of elementary number theory* he presented a definite conclusion: that the class of effectively calculable functions should be identified with the class of functions definable in the *lambda calculus*, a formal language for specifying the construction of functions. Church moreover showed that exactly the same class of functions could be characterized in terms of formal derivability from equations.

Turing then promptly completed writing his paper, in which he presented a very different approach to characterizing the effectively calculable functions, but one that—as he proved—yielded once again the same class of functions as Church had proposed. With Newman's encouragement, Turing went to Princeton for two years, where he wrote a Ph.D. dissertation under Alonzo Church.

Turing's paper remains a very readable introduction to his ideas. How might a diligent clerk carry out a calculation, following instructions? He (or she) might organize the work in a notebook. At any given moment his attention is focused on a particular page. Following his instructions, he might alter that page, and then he might turn to another page. And the notebook is large enough (or the supply of fresh paper is ample enough) that he never comes to the last page.

The alphabet of symbols available to the clerk must be finite; if there were infinitely many symbols, then there would be two that were arbitrarily similar and so might be confused. We can then without loss of generality regard what can be written on one page of notebook as a single symbol. And we can envision the notebook pages as being placed side by side, forming a paper tape, consist-

ing of squares, each square being either blank or printed with a symbol. (For uniformity, we can think of a blank square as containing the “blank” symbol B .) At each stage of his work, the clerk—or the mechanical machine—can alter the square under examination, can turn attention to the next square or the previous one, and can look to the instructions to see what part of them to follow next. Turing described the latter part as a “change of state of mind.”

Turing wrote, “We may now construct a machine to do the work.” Such a machine is of course now called a *Turing machine*, a phrase first used by Church in his review of Turing’s paper in *The Journal of Symbolic Logic*. The machine has a potentially infinite tape, marked into squares. Initially the given input numeral or word is written on the tape, but it is otherwise blank. The machine is capable of being in any one of finitely many “states” (the phrase “of mind” being inappropriate for a machine).

At each step of calculation, depending on its state at the time, the machine can change the symbol in the square under examination at that time, and can turn its attention to the square to the left or to the right, and can then change its state to another state. (The tape stretches endlessly in both directions.)

The program for this Turing machine can be given by a table. Where the possible states of the machine are q_1, \dots, q_r , each line of the table is a quintuple $\langle q_i, S_j, S_k, D, q_m \rangle$ which is to be interpreted as directing that whenever the machine is in state q_i and the square under examination contains the symbol S_j , then that symbol should be altered to S_k and the machine should shift its attention to the square on the left (if $D = L$) or on the right (if $D = R$), and should change its state to q_m . Possibly S_j is the “blank” symbol B , meaning the square under examination is blank; possibly S_k is B , meaning that whatever is in the square is to be erased. For the program to be unambiguous, it should have no two different quintuples with the same first two components. (By relaxing this requirement regarding absence of ambiguity, we obtain the concept of a *non-deterministic* Turing machine, which will be useful later, in the discussion of feasible computability.) One of the states, say q_1 , is designated as the initial state—the state in which the machine begins its calculation. If we start the machine running in this state, and examining the first square of its input, it might (or might not), after some number of steps, reach a state and a symbol for which its table lacks a quintuple having that state and symbol for its first two components. At that point the machine *halts*, and we can look at the tape (starting with the square then under examination) to see what the output numeral or word is.

Now suppose that Σ is a finite alphabet (the blank B does not count as a member of Σ). Let Σ^* be the set of all words over this alphabet (that is, Σ^* is the set of all strings, including the empty string, consisting of members of Σ). Suppose that f is a k -place partial function from Σ^* into Σ^* . We will say that f is *Turing computable* if there exists a Turing machine \mathcal{M} that, when started in its initial state scanning the first symbol of a k -tuple \vec{w} of words (written on the tape, with a blank square between words, and with the rest of the tape blank), behaves as follows:

- If $f(\vec{w}) \downarrow$ (i.e., if $\vec{w} \in \text{dom } f$) then \mathcal{M} eventually halts, and at that time it is scanning the leftmost symbol of the word $f(\vec{w})$ (which is followed by a blank square).
- If $f(\vec{w}) \uparrow$ (i.e., if $\vec{w} \notin \text{dom } f$) then \mathcal{M} never halts.

Example: Take a two-letter alphabet $\Sigma = \{a, b\}$. Let \mathcal{M} be the Turing machine given by the following set of six quintuples, where q_1 is designated as the initial state:

$$\begin{aligned} &\langle q_1, a, a, R, q_1 \rangle \\ &\langle q_1, b, b, R, q_1 \rangle \\ &\langle q_1, B, a, L, q_2 \rangle \\ &\langle q_2, a, a, L, q_2 \rangle \\ &\langle q_2, b, b, R, q_2 \rangle \\ &\langle q_2, B, B, R, q_3 \rangle \end{aligned}$$

Suppose we start this machine in state q_1 , scanning the first letter of a word w . The machines move (in state q_1) to the right end of w , where it appends the letter a . Then it moves (in state q_2) back to the left end of the word, where it halts (in state q_3). Thus \mathcal{M} computes the total function $f(w) = wa$.

We need to adopt special conventions for handling the empty word λ , which occupies zero squares. This can be done in different ways; the following is the way chosen. If the machine halts scanning a blank square, then the output word is λ . For a 1-place function f , to compute $f(\lambda)$, we simply start with a blank tape. For a 2-place function g , to compute $g(w, \lambda)$ we start with only the word w , scanning the first symbol of w . And to compute $g(\lambda, w)$, we also start with only the word w on the tape, but scanning the blank square just to the left of w . And in general, to give a k -place function the input $\vec{w} = \langle u_1, \dots, u_k \rangle$ consisting of k words of lengths n_1, \dots, n_k , we start the machine scanning the first square of the input configuration of length $n_1 + \dots + n_k + k - 1$

$$(n_1 \text{ symbols from } u_1)B(n_2 \text{ symbols from } u_2)B \cdots B(n_k \text{ symbols from } u_k)$$

with the rest of the tape blank. Here any n_i can be zero; in the extreme case, they can all be zero.

An obvious drawback of these conventions is that there is no difference between the pair $\langle u, v \rangle$ and the triple $\langle u, v, \lambda \rangle$. Other conventions avoid this drawback, at the cost of introducing their own idiosyncracies.

The definition of Turing computability can be readily adapted to apply to k -place partial functions on \mathbb{N} . The simplest way to do this is to use base-1 numerals. We take a one-letter alphabet $\Sigma = \{\mid\}$ whose one letter is the tally mark \mid . Or to be more conventional, let $\Sigma = \{1\}$, using the symbol 1 in place of the tally mark. Then the input configuration for the triple $\langle 3, 0, 4 \rangle$ is

$$111BB1111.$$

Then *Church's thesis*, also called—particularly in the context of Turing machines—the *Church–Turing thesis*, is the claim that this concept of Turing

computability is the correct formalization of the informal concept of effective calculability. Certainly the definition reflects the ideas of following predetermined instructions, without limitation of the amount of time that might be required. (The name “Church–Turing thesis” obscures the fact that Church and Turing followed very different paths in reaching equivalent conclusions.)

Church’s thesis has by now achieved universal acceptance. Kurt Gödel, writing in 1964 about the concept of a “formal system” in logic, involving the idea that the set of correct deductions must be a decidable set, said 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.” And others agree.

The robustness of the concept of Turing computability is evidenced by the fact that it is insensitive to certain modifications to the definition of a Turing machine. For example, we can impose limitations on the size of the alphabet, or we can insist that the machine never move to the left of its initial starting point. None of this will affect that class of Turing computable partial functions.

Turing developed these ideas prior to the introduction of modern digital computers. After World War II, Turing played an active rôle in the development of early computers, and in the emerging field of artificial intelligence. (During the war, he worked on deciphering the German battlefield code Enigma, militarily important work which remained classified until after Turing’s death.) One can speculate as to whether Turing might have formulated his ideas somewhat differently, if his work had come after the introduction of digital computers.

Digression: There is an interesting example here, that goes by the name¹ of “the busy beaver problem.”

Suppose we want a Turing machine, starting on a blank tape, to write as many 1’s as it can, and then stop. With a limited number of states, how many 1’s can we get?

To make matters more precise, take Turing machines with the alphabet $\{1\}$ (so the only symbols are B and 1). We will allow such machines to have n states, plus a halting state (that can occur as the last member of a quintuple, but not as the first member). For each n , there are only finitely many essentially different such Turing machines. Some of them, started on a blank tape, might not halt. For example the 1-state machine

$$\langle q_1, B, 1, R, q_1 \rangle$$

keeps writing forever without halting. But among those that do halt, we seek the ones that write a lot of 1’s.

Define $\sigma(n)$ to be the largest number of 1’s that can be written by an n -state Turing machine as described above before it halts. For example, $\sigma(1) = 1$, because the 1-state machine

$$\langle q_1, B, 1, R, q_H \rangle$$

(the halting state q_H doesn’t count) writes one 1, and none of the other 1-state machines do any better. (There are not so very many 1-state machines, and

¹This name has given translators much difficulty.

one can examine all of them in a reasonable length of time). Let's agree that $\sigma(0) = 0$. Then σ is a total function. It is also nondecreasing, since having an extra state to work with is never a handicap. Despite the fact that $\sigma(n)$ is merely the largest member of a certain finite set, there is no algorithm that lets us, in general, evaluate it.

Example: Here is a two-state candidate:

$$\begin{aligned} &\langle q_1, B, 1, R, q_2 \rangle \\ &\langle q_1, 1, 1, L, q_2 \rangle \\ &\langle q_2, B, 1, L, q_1 \rangle \\ &\langle q_2, 1, 1, R, q_H \rangle \end{aligned}$$

Started on a blank tape, this machine write four consecutive 1's, and then halts (after six steps), scanning the third 1. You are invited to verify this by running the machine. We conclude the $\sigma(2) \geq 4$.

Rado's Theorem (1962): The function σ is not Turing computable. Moreover, for any Turing computable total function f , we have $f(x) < \sigma(x)$ for all sufficiently large x . That is, σ eventually dominates any Turing computable total function.

Proof outline: Assume we are given some Turing computable total f . We must show that σ eventually dominates it. Define (for reasons that may initially appear mysterious) the function g :

$$g(x) = \max(f(2x), f(2x + 1)) + 1$$

Then g is total and one can show that it is Turing computable. So there is some Turing machine \mathcal{M} with, say, k states that computes it, using the alphabet $\{1\}$ and base-1 notation. For each x , let \mathcal{N}_x be the $(x + k)$ -state Turing machine that first writes x 1's on the tape, and then imitates \mathcal{M} . (The x states let us write x 1's on the tape in a straightforward way, and then there are the k states in \mathcal{M} .)

Then \mathcal{N}_x , when started on a blank tape, writes $g(x)$ 1's on the tape and halts. So $g(x) \leq \sigma(x + k)$, by the definition of σ . Thus we have

$$f(2x), f(2x + 1) < g(x) \leq \sigma(x + k)$$

and if $x \geq k$ then

$$\sigma(x + k) \leq \sigma(2x) \leq \sigma(2x + 1).$$

Putting these two lines together, we see that $f < \sigma$ from $2k$ on. \dashv

So σ grows faster—eventually—than any Turing computable total function. How fast does it grow? Among the smaller numbers, $\sigma(2) = 4$. (The preceding example shows that $\sigma(2) \geq 4$. The other inequality is not entirely trivial, because there are thousands of 2-state machines.) It has also been shown that $\sigma(3) = 6$ and $\sigma(4) = 13$. From here on, only lower bounds are known. In

1984 it was found that $\sigma(5)$ is at least 1,915. In 1990 this was raised to 4,098. And $\sigma(6) > 4.6 \times 10^{1439}$. And $\sigma(7)$ must be astronomical. These lower bounds are established by using ingeniously convoluted coding to make small Turing machines that write that many 1's and then halt.

Proving further *upper* bounds would be difficult. In fact, one can show, under some reasonable assumptions, that upper bounds on $\sigma(n)$ are provable for only finitely many n 's. (We will return to this point in Chapter 6.)

If we could solve the halting problem, we would then have the following method for computing $\sigma(n)$:

- List all the n -state machines.
- Discard those that never halt.
- Run those that do halt.
- Select the highest score.

It is the second step in this method that gives us trouble. (New information on Rado's σ function continues to be discovered. Recent news can be obtained from the webpage maintained by Heiner Marxen, <http://www.drb.insel.de/~heiner/BB>.)

Primitive recursiveness and search

For a second formalization of the calculability concept, we will define a certain class of partial functions on \mathbb{N} as the smallest class that contains certain initial function and is closed under certain constructions.

For the initial functions, we take the following very simple total functions:

- The *zero* functions, that is, the constant functions f defined by the equation:

$$f(x_1, \dots, x_k) = 0$$

There is one such function for each k .

- The *successor* function S , defined by the equation:

$$S(x) = x + 1$$

- The *projection* functions I_n^k from k -dimensions onto the n th coordinate,

$$I_n^k(x_1, \dots, x_k) = x_n$$

where $1 \leq n \leq k$.

We want to form the closure of the class of initial functions under three constructions: composition, primitive recursion, and search.

A k -place function h is said to be obtained by *composition* from the n -place function f and the k -place functions g_1, \dots, g_n if the equation

$$h(\vec{x}) = f(g_1(\vec{x}), \dots, g_n(\vec{x}))$$

holds for all \vec{x} . In the case of partial functions, it is to be understood here that $h(\vec{x})$ is undefined unless $g_1(\vec{x}), \dots, g_n(\vec{x})$ are all defined and $\langle g_1(\vec{x}), \dots, g_n(\vec{x}) \rangle$ belongs to the domain of f .

A $(k+1)$ -place function h is said to be obtained by *primitive recursion* from the k -place function f and the $(k+2)$ -place function g (where $k > 0$) if the pair of equations

$$\begin{aligned} h(\vec{x}, 0) &= f(\vec{x}) \\ h(\vec{x}, y+1) &= g(h(\vec{x}, y), \vec{x}, y) \end{aligned}$$

holds for all \vec{x} and y .

Again, in the case of partial functions, it is to be understood that $h(\vec{x}, y+1)$ is undefined unless $h(\vec{x}, y)$ is defined and $\langle h(\vec{x}, y), \vec{x}, y \rangle$ is in the domain of g .

Observe that in this situation, knowing the two functions f and g completely determines the function h . More formally, if h_1 and h_2 are both obtained by primitive recursion from f and g , then for each \vec{x} we can show by induction on y that $h_1(\vec{x}, y) = h_2(\vec{x}, y)$.

For the $k = 0$ case, the one-place function h is obtained by primitive recursion from the two-place function g by using the number m if the pair of equations

$$\begin{aligned} h(0) &= m \\ h(y+1) &= g(h(y), y) \end{aligned}$$

holds for all y .

Postponing the matter of search, we define a function to be *primitive recursive* if it can be built up from zero, successor, and projection functions by use of composition and primitive recursion. In other words, the class of primitive recursive functions is the smallest class that includes our initial functions and is closed under composition and primitive recursion. (Here saying that a class \mathcal{C} is “closed” under composition and primitive recursion means that whenever a function f is obtained by composition from functions in \mathcal{C} or is obtained by primitive recursion from functions in \mathcal{C} , then f itself also belongs to \mathcal{C} .)

Clearly all the primitive recursive functions are total. This is because the initial functions are all total, the composition of total functions is total, and a function obtained by primitive recursion from total functions will be total.

We say that a k -ary relation R on \mathbb{N} is primitive recursive if its characteristic function is primitive recursive.

One can then show that a great many of the common functions on \mathbb{N} are primitive recursive: addition, multiplication, \dots , the function whose value at m is the $(m+1)$ st prime, \dots . Chapter 2 will carry out the project of showing that many functions are primitive recursive.

On the one hand, it seems clear that every primitive recursive function should be regarded as being effectively calculable. (The initial functions are pretty

easy. Composition presents no big hurdles. Whenever h is obtained by primitive recursion from effectively calculable f and g , then we see how we could effectively find $h(\vec{x}, 99)$, by first finding $h(\vec{x}, 0)$ and then working our way up.) On the other hand, the class of primitive recursive functions cannot possibly comprehend all total calculable functions, because we can “diagonalize out” of the class. That is, by suitably indexing the “family tree” of the primitive recursive functions, we can make a list f_0, f_1, f_2, \dots of all the one-place primitive recursive functions. Then consider the diagonal function $d(x) = f_x(x) + 1$. Then d cannot be primitive recursive; it differs from each f_x at x . Nonetheless, if we made our list very tidily, the function d will be effectively calculable. The conclusion is the class of primitive recursive functions is an extensive but proper subset of the total calculable functions.

Next, we say that a k -place function h is obtained from the $k + 1$ -place function g by *search* and we write

$$h(\vec{x}) = \mu y[g(\vec{x}, y) = 0]$$

if for each \vec{x} , the value $h(\vec{x})$ either is the number y such that $g(\vec{x}, y) = 0$ and $g(\vec{x}, s)$ is defined and is non-zero for every $s < y$, if such a number t exists, or else is undefined, if no such number t exists. The idea behind this “ μ -operator” is the idea of searching for the least number y that is the solution to an equation, by testing successively $y = 0, 1, \dots$.

We obtain the *general recursive* functions by adding search to our closure methods. That is, a partial function is general recursive if it can be built up from the initial zero, successor, and projection functions, by use of composition, primitive recursion, and search (i.e., the μ -operator).

The class of general recursive partial functions on \mathbb{N} is (as Turing proved) exactly the same as the class of Turing computable partial functions. This is a rather striking result, in light of the very different ways in which the two definitions were formulated. Turing machines would seem, at first glance, to have little to do with primitive recursion and search. And yet we get exactly the same partial functions from the two approaches. And Church’s thesis therefore has the equivalent formulation that the concept of a general recursive function is the correct formalization of the informal concept of effective calculability.

What if we try to “diagonalize out” of the class of general recursive functions, as we did for the primitive recursive functions? As will be argued later, we can again make a tidy list $\varphi_0, \varphi_1, \varphi_2, \dots$ of all the one-place general recursive partial functions. And we can define the diagonal function $d(x) = \varphi_x(x) + 1$. But in this equation, $d(x)$ is undefined unless $\varphi_x(x)$ is defined. The diagonal function d is indeed among the general recursive partial functions, and hence is φ_k for some k , but $d(k)$ must be undefined. No contradiction results.

The class of primitive recursive functions was defined by Gödel, in his 1931 paper on the incompleteness theorems. Of course, the idea of defining functions on \mathbb{N} by recursion is much older, and reflects the idea that the natural numbers are built up from the number 0 by repeated application of the successor function. (Dedekind wrote about this topic.) The theory of the general recursive functions was worked out primarily by Stephen Kleene, a student of Church.

The use of the word “recursive” in the context of the primitive recursive functions is entirely reasonable. Gödel, writing in German, had used simply “rekursiv” for the primitive recursive functions. (It was Rózsa Péter who introduced the term “primitive recursive.”) But retaining the word “recursive” for the general recursive functions was a historical accident. The class of general recursive functions—as this section shows—has several characterizations in which *recursion* (i.e., defining a function in terms of its other values, or using routines that call themselves) plays no rôle at all.

Nonetheless, the terminology became standard. What are here called the computable partial functions were until recently (and often still are) standardly called the partial recursive functions. And for that matter, computability theory was called recursive function theory for many years, and then *recursion theory*. And relations on \mathbb{N} were said to be *recursive* if their characteristic functions were general recursive functions.

But now an effort is being made to change what had been the standard terminology. According, these notes on *Computability theory* speak of *computable* partial functions. And we will call a relation *computable* if its characteristic function is a computable function. Thus the concept of a computable relation corresponds to the informal notion of a decidable relation.

In any case, there is definitely a need to have separate adjectives for the informal concept (here “calculable” is used for functions, and “decidable” for relations) and the formally defined concept (here “computable”).

Loop and While programs

The idea behind the concept of effective calculable functions is that one should be able to give explicit instructions—a program—for calculating such a function. What programming language would be adequate here? Actually, any of the commonly used programming languages would suffice, if freed from certain practical limitations, such as the size of the number denoted by a variable. We give here a simple programming language with the property that the programmable functions are exactly the computable partial functions on \mathbb{N} .

The variables of the language are X_0, X_1, X_2, \dots . Although there are infinitely many variables in the language, any one program, being a finite string of commands, can have only finitely many of these variables. If we want the language to consist of words over a finite alphabet, we can replace X_3 , say, by X''' .

In running a program, each variable in the program gets assigned a natural number. There is no limit on how large this number can be. Initially, some of the variables will contain the input to the function; the language has no “input” commands. Similarly, the language has no “output” commands; when (and if) the program halts, the value of X_0 is to be the function value.

The commands of the language come in five kinds:

1. $X_n \leftarrow 0$. This is the *clear* command; its effect is to assign the value 0 to X_n .

2. $X_n \leftarrow X_n + 1$. This is the *increment* command; its effect is to increase the value assigned to X_n by one.
3. $X_n \leftarrow X_m$. This is the *copy* command; its effect is just what the name suggests; in particular it leaves the value of X_m unchanged.
4. `loop X_n` and `endloop X_n` . These are the *loop* commands, and they must be used in pairs. That is, if \mathcal{P} is a program—a syntactically correct string of commands—then so is the string:

$$\begin{array}{c} \text{loop } X_n \\ \mathcal{P} \\ \text{endloop } X_n \end{array}$$

What this program means is that \mathcal{P} is to be executed a certain number k of times. And that number k is the *initial* value of X_n , the value assigned to X_n before we start executing \mathcal{P} . Possibly \mathcal{P} will change the value of X_n ; this has no effect at all on k . If $k = 0$, then this string does nothing.

5. `while $X_n \neq 0$` and `endwhile $X_n \neq 0$` . These are the *while* commands; again, they must be used in pairs, like the the loop commands. But there is a difference. The program

$$\begin{array}{c} \text{while } X_n \neq 0 \\ \mathcal{P} \\ \text{endwhile } X_n \neq 0 \end{array}$$

also executes the program \mathcal{P} some number k of times. But now k is *not* determined in advance; it matters very much how \mathcal{P} changes the value of X_n . The number k is the least number (if any) such that executing \mathcal{P} that many times causes X_n to be assigned the value 0. The program will run forever if there is no such k .

And those are the only commands. A *while* program is a sequence of commands, subject only to the requirement that the loop and while commands are used in pairs, as illustrated. Clearly, this programming language is simple enough to be simulated by any of the common programming language, if we ignore overflow problems.

A *loop* program is a while program with no while commands; that is, it has only clear, increment, copy, and loop commands. Note the important property: A loop program *always halts*, no matter what. But it is easy to make a while program that never halts.

We say that a k -place partial function f on \mathbb{N} is *while-computable* if there exists a while program \mathcal{P} that, whenever started with a k -tuple \vec{x} assigned to the variables X_1, \dots, X_k and 0 assigned to the other variables, behaves as follows:

- If $f(\vec{x})$ is defined, then the program eventually halts, with X_0 assigned the value $f(\vec{x})$.
- If $f(\vec{x})$ is undefined, then the program never halts.

The *loop-computable* functions are defined in the analogous way. But there is the difference that any loop-computable function is total.

Theorem: (a) A function on \mathbb{N} is loop-computable if and only if it is primitive recursive.

(b) A partial function on \mathbb{N} is while-computable if and only if it is general recursive.

The proof in one direction, to show that every primitive recursive function is loop-computable, involves a series of programming exercises. The proof in the other direction involves coding the status of a program \mathcal{P} on input \vec{x} after t steps, and showing that there are primitive recursive functions enabling us to determine the status after $t + 1$ steps, and the terminal status.

Because the class of general recursive partial functions coincides with the class of Turing computable partial functions, we can conclude from the above theorem that while-computability coincides with Turing computability.

Register machines

Here is another programming language. On the one hand, it is extremely simple—even simpler than the language for loop-while programs. On the other hand, the language is “unstructured”; it incorporates (in effect) go-to commands. This formalization was presented by Shepherdson and Sturgis in a 1963 paper.

A *register machine* is to be thought of as a computing device with a finite number of “registers,” numbered $0, 1, 2, \dots, K$. Each register is capable of storing a natural number of any magnitude—there is no limit to the size of this number. The operation of the machine is determined by a *program*. A program is a finite sequence of *instructions*, drawn from the following list:

- I r (where $0 \leq r \leq K$). “Increment r .” The effect of this instruction is to increase the contents of register r by 1. The machine then proceeds to the next instruction in the program (if any).
- D r (where $0 \leq r \leq K$). “Decrement r .” The effect of this instruction depends on the contents of register r . If that number is non-zero, it is decreased by 1 and the machine proceeds *not* to the next instruction, but to the following one. But if the number in register r is zero, the machine simply proceeds to the next instruction. In summary, the machine tries to decrement register r and if it is successful then it skips one instruction.
- J q (where q is an integer—positive, negative, or zero). “Jump q .” All registers are left unchanged. The machine takes as its next instruction the q th instruction following this one in the program (if $q \geq 0$), or the $|q|$ th instruction preceding this one (if $q < 0$). The machine halts if there is no such instruction in the program. An instruction of J 0 results in a loop, with the machine executing this one instruction over and over again.

(Strictly speaking, in these instructions, r and q are numerals, not numbers. That is, an instruction should be a sequence of *symbols*. If we use base-10 numerals, then the alphabet is $\{I, D, J, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -\}$. An instruction is a correctly formed word over this alphabet.)

EXAMPLES

1. CLEAR 7: a program to clear register 7.

```

D 7      Try to decrement 7.
J 2
J -2     Go back and repeat.
        Halt.

```

2. MOVE from r to s : a program to move a number from register r to register s (where $r \neq s$).

```

CLEAR  $s$ .      Use the program of the first example.
D  $r$           Take 1 from  $r$ .
J 3          Halt when zero.
I  $s$           Add 1 to  $s$ .
J -3         Repeat.

```

This program has seven instructions altogether. It leaves a zero in register r .

3. ADD 1 to 2 and 3: a program to add register 1 to registers 2 and 3.

```

D 1
J 4
I 2
I 3
J -4

```

This program leaves a zero in register 1. It is clear how to adapt the program to add register 1 to more (or fewer) than two registers.

4. COPY from r to s (using t): a program to copy a number from register r to register s (leaving register r unchanged). We combine the previous examples.

```

        CLEAR  $s$ .      Use the first example.
MOVE from  $r$  to  $t$ .    Use the second example.
        ADD  $t$  to  $r$  and  $s$ .    Use the third example.

```

This program has fifteen instructions. It uses a third register, register t . At the end, the contents for register r are restored. But during execution, register r must be cleared; this is the only way of determining its contents. (It is assumed here that r , s , and t are distinct.)

5. (Addition) Say that x and y are in registers 1 and 2. We want $x + y$ in register 0, and we want to leave x and y still in registers 1 and 2 at the end.

	<i>Register contents</i>		
CLEAR 0.	0	x	y
MOVE from 1 to 3.	0	0	y x
ADD 3 to 1 and 0.	x	x	y 0
MOVE from 2 to 3.	x	x	0 y
ADD 3 to 2 and 0.	$x + y$	x	y 0

This program has twenty-seven instructions as it is written, but three of them are unnecessary. (In the fourth line we begin by clearing register 3, which is already clear.)

Now suppose f is an n -place partial function on \mathbb{N} . Possibly there will be a program \mathcal{P} such that if we start a register machine (having all the registers to which \mathcal{P} refers) with x_1, \dots, x_n in registers $1, \dots, n$ and 0 in the other registers, and we apply program \mathcal{P} , then the following conditions hold:

- If $f(x_1, \dots, x_n)$ is defined, then the computation eventually terminates with $f(x_1, \dots, x_n)$ in register 0. Furthermore, the computation terminates by seeking a $(p + 1)$ st instruction, where p is the length of \mathcal{P} .
- If $f(x_1, \dots, x_n)$ is undefined, then the computation never terminates.

If there is such a program \mathcal{P} , we say that \mathcal{P} *computes* f .

Which functions are computable by register machine programs? The language is so simple—it appears to be a toy language—that one’s first impression might be that only very simple functions are computable. This impression is misleading:

Theorem. Let f be a partial function. Then there is a register machine program that computes f iff f is a general recursive partial function.

Thus by using register machines we arrive at exactly the class of general recursive partial functions, a class we originally defined in terms of primitive recursion and search.

Definability in formal languages

In his 1936 paper in which he presented what is now known as Church’s thesis, Alonzo Church utilized a formal system, the λ -*calculus*. Church had developed this system as part of his study of the foundations of logic. In particular, for each natural number n there is a formula \bar{n} of the system denoting n , that is, a numeral for n . More importantly, formulas could be used to represent the construction of functions. He defined a two-place function F to be λ -*definable* if there existed a formula \mathbf{F} of the lambda calculus such whenever $F(m, n) = r$ then the formula $\{\mathbf{F}\}(\bar{m}, \bar{n})$ was convertible, following the rules of the system, to the formula \bar{r} , and only then. An analogous definition applied to k -place functions.

Church’s student Stephen Kleene showed that a function was λ -definable if and only if it was general recursive. (Church and his student J. B. Rosser also were involved in the development of this result.) Church wrote in his paper, “The fact . . . that two such widely different and (in the opinion of the author) equally natural definitions of effective calculability turn out to be equivalent adds to the strength of reasons . . . for believing that they constitute as general a characterization of this notion as is consistent with the usual intuitive understanding of it.”

Earlier, in 1934, Kurt Gödel, in lectures at Princeton, formulated a concept now referred to as Gödel–Herbrand computability. He did not, however, at the time propose the concept as a formalization of the concept of effective calculability. The concept involved a formal calculus of equations between terms built up from variables and function symbols. The calculus permitted the passage from an equation $A = B$ to another equation obtained by substituting for a part C of A or B another term D where the equation $C = D$ had been derived. If a set \mathcal{E} of equations allowed the derivation, in a suitable sense, of exactly the right values for a function f on \mathbb{N} , then \mathcal{E} was said to be a set of *recursion equations* for f . Once again, it turned out that a set of recursion equations existed for f if and only if f was a general recursive function.

A rather different approach to characterizing the effectively calculable functions involves definability by expressions in symbolic logic. A formal language for the arithmetic of natural numbers might have variables and a numeral for each natural number, and symbols for the equality relation and for the operations of addition and multiplication, at least. Moreover, the language should be able to handle the basic logical connectives such as “and,” “or,” and “not.” Finally, it should include the “quantifier” expressions $\forall v$ and $\exists v$ meaning “for all natural numbers v ” and “for some natural number v ,” respectively.

For example,

$$\exists s(u_1 + s = u_2)$$

might be an expression in the formal language, asserting a property of u_1 and u_2 . The expression is true (in \mathbb{N} with its usual operations) when u_1 is assigned 4 and u_2 is assigned 9 (take $s = 5$). But it is false when u_1 is assigned 9 and u_2 is assigned 4. More generally, we can say that the expression *defines* (in \mathbb{N} with its usual operations) the binary relation “ \leq ” on \mathbb{N} .

For another example,

$$v \neq 0 \text{ and } \forall x \forall y [\exists s(v + s = x) \text{ or } \exists t(v + t = y) \text{ or } v \neq x \cdot y]$$

might be an expression in the formal language, asserting a property of v . The expression is false (in \mathbb{N} with its usual operation) when v is assigned the number 6 (try $x = 2$ and $y = 3$). But the expression is true when v is assigned 7. More generally, the expression is true when v is assigned a prime number, and only then. We can say that this expression *defines* the set of prime numbers (in \mathbb{N} with its usual operations).

Say that a k -place partial function f on \mathbb{N} is Σ_1 -*definable* if the graph of f (that is, the $(k + 1)$ -ary relation $\{\langle \vec{x}, y \rangle \mid f(\vec{x}) = y\}$) can be defined in \mathbb{N} and with the operations of addition, multiplication, and exponentiation, by an expression of the following form:

$$\exists v_1 \exists v_2 \cdots \exists v_n (\text{expression without quantifiers})$$

Then the class of Σ_1 -definable partial functions coincides exactly with the class of partial functions given by the other formalizations of calculability described here. Moreover, Yuri Matijasevič showed in 1970 that the operation of exponentiation was not needed here.

Finally, say that a k -place partial function f on \mathbb{N} is *representable* if there exists some finitely axiomatizable theory T in a language having a suitable numeral \mathbf{n} for each natural number n , and there exists a formula φ of that language such that (for any natural numbers) $f(x_1, \dots, x_k) = y$ if and only if $\varphi(\mathbf{x}_1, \dots, \mathbf{x}_k, \mathbf{y})$ is a sentence deducible in the theory T . Then once again the class of representable partial functions coincides exactly with the class of partial functions given by the other formalizations of calculability described here.

Church's thesis revisited

In summary, for a k -place partial function f , the following conditions are equivalent:

- The function f is a Turing-computable partial function.
- The function f is a general recursive partial function.
- The partial function f is while-computable.
- The partial function f is computed by some register-machine program.
- The partial function f is λ -definable.
- The partial function f is Σ_1 -definable (over the natural numbers with addition, multiplication, and exponentiation).
- The partial function f is representable (in some finitely axiomatizable theory).

The equivalence of these conditions is surely a remarkable fact! Moreover, it is evidence that the conditions characterize some natural and significant property. Church's thesis is the claim that the conditions in fact capture the informal concept of an effectively calculable function.

Definition: A k -place partial function f on the natural numbers is said to be a *computable partial function* if the foregoing conditions hold.

Then Church's thesis is the claim that this definition is the one we want.

The situation is somewhat analogous to one in calculus. An intuitively continuous function (defined on an interval) is one whose graph can be drawn without lifting the pencil off the paper. But to prove theorems, some formalized counterpart of this concept is needed. And so one gives the usual definition of ε - δ -continuity. Then it is fair to ask whether the precise concept of ε - δ -continuity is an accurate formalization of intuitive continuity. If anything, the class of ε - δ -continuous functions is too *broad*. It includes nowhere differentiable functions, whose graphs cannot be drawn without lifting the pencil—there is no way to impart a velocity vector to the pencil. But accurate or not, the class of ε - δ -continuous functions has been found to be a natural and important class in mathematical analysis.

Very much the same situation occurs with computability. It is fair to ask whether the precise concept of a computable partial function is an accurate formalization of the informal concept of an effectively calculable function. Again, the precisely defined class appears to be, if anything, too broad, because it includes functions requiring, for large inputs, absurd amounts of computing time. Computability corresponds to effective calculability in an idealized world, where length of computation and amount of memory space are disregarded. But in any event, the class of computable partial functions has been found to be a natural and important class.

Exercises

14. Give a loop program to compute the following function:

$$f(x, y, z) = \begin{cases} y & \text{if } x = 0 \\ z & \text{if } x \neq 0 \end{cases}$$

15. Give a loop program that computes $x \dot{-} y = \max(x - y, 0)$.

16. Give a loop program that when started with all variables assigned 0, halts with X_0 assigned some number greater than 1,000.

Give register machine programs that compute the following functions.

17. Subtraction, $x \dot{-} y = \max(x - y, 0)$.
18. Multiplication, $x \cdot y$.
19. $\max(x, y)$.