Effective Calculability and Unsolvability

No matter where it is or when it is, at some point soon after we begin learning about computer science, someone introduces the concept of an algorithm to us and defines it in some manner that we accept and come to take for granted. When we begin to write programs and take classes in programming, the word becomes ever more important and obvious. Nonetheless, if one is asked to write down a rigorous definition of an algorithm, could it be done?

Let us say that an algorithm to solve a class of problems is a precise set of instructions that can be carried out by mechanical means, without creative thought, that provides a solution to any problem in that class in a finite amount of time. This definition is still pretty ambiguous, but we know what we mean by it. Suppose that we could write down a truly rigorous definition of an algorithm. Furthermore, suppose that, not only could we define precisely what an algorithm is, but that if we were reading a description of some alleged algorithm, we could, by using equally precise means, determine whether or not the thing being defined actually is an algorithm. In other words, let us assume the truth of two statements.

The first is that there exists some precise and universal notion of an algorithm.

The second is that there is an algorithm that we can use to determine whether something purported to be an algorithm is actually an algorithm.

At this point, I will reproduce an argument made by Martin Davis in the introduction to his book, *Computability and Unsolvability* [Davis1].

Consider all possible functions $f(x)$ defined on the positive integers whose values are positive integers. Functions such as x^2, x^3, $x!$, and a^x, fall into this category. There are clearly infinitely many such functions. Let us say a function $f(x)$ is effectively calculable if there is an algorithm that, given a value for x, lets us compute the value of $f(x)$ by carrying it out step by step.

Such an algorithm can be described using the English language. Think of the description of the algorithm in the English language as a string of letters, punctuation marks, and spaces. We can order all possible algorithms like this, first by the length of the description, and then, for all descriptions of the same length, by their dictionary order. In other words, all descriptions of length one come first, then of length two, then of length three, and so on. Within any one length, the descriptions would be in dictionary order. We can make rules as to whether uppercase precedes lowercase and so on, but this is not important.

Because the algorithms can be written in a precise sequence, we can number them, say by A_1, A_2, A_3, and so on. In general, A_k is the algorithm whose description is k^{th} in the list, and the function that Ak computes is then called $f_k(x)$. We now have a sequence of functions $f_k(x)$, for $k = 1, 2, 3, 4, \ldots$ that are all effectively calculable.

Define the function $g(x)$ by

$$g(x) = f_x(x) + 1. \qquad (1)$$

The function $g(x)$ is also a function that has a positive integer argument with a positive integer value. To find its value for a given x, we find the x^{th} algorithm in the list of algorithms A_1, A_2, A_3, ..., apply the algorithm to x and add 1 to the result. However,

Theorem 1. *There is no value k such that* $g(x) = f_k(x)$.

Proof.

Suppose, to the contrary, that such a k exists. Then for all x, $f_k(x) = g(x)$. Since $g(x) = f_x(x) + 1$ by definition,

$$f_k(x) = g(x) = f_x(x) + 1$$

for all values of x. In particular, it must be true when we choose $x = k$:

$$f_k(k) = f_k(k) + 1$$

This, of course, is impossible, so we have reached a contradiction, implying that no such value k exists. ■

But if there is no value k for which $g(x) = f_k(x)$, then $g(x)$ is not a function computed by any algorithm, which implies that

Theorem 2. *g(x) is not effectively calculable.*

Even though $g(x)$ is not effectively calculable, the description of how to compute $g(x)$ above seems very algorithmic. To make it more precise, we could do the following:

1. Given a number x, start generating the descriptions A_1, A_2, A_3, ..., until the x^{th} description is obtained.
2. Apply the description A_x to x.
3. Add 1 to the result.

Since Theorem 2 is true, there must be a flaw in the above description. How can we generate the list A_1, A_2, A_3, ...? It is certainly possible to generate all possible strings of length 1, then 2, then 3, and so on. Each time that we generate a string, we check whether that string represents an algorithm that computes a function on the positive integers with a positive integer result. If that string does represent such an algorithm, we increment our counter, and we do this until we have counted off x such algorithms.

It seems perfectly reasonable then to generate the list of such algorithms. Or does it? In the procedure described above, we are making the assumption that it is possible to mechanically determine whether a piece of English text describes an algorithm for a function of a single, positive integer argument with a positive integer value. Everything else is provably true. Therefore, if Theorem 2 is true, it implies that this is not possible. In other words:

Theorem 3. *There is no algorithm that can be used to determine whether a purported algorithm to compute the values of a function over the positive integers whose range is the positive integers, is actually such an algorithm.*

Another way to look at this is to replace the English text by syntactically correct programs in a fixed language such as C. The statement can be interpreted to mean that there is no algorithm to determine whether a given program computes a total function from positive integers to positive integers.

This is an example of an unsolvable problem. We reached the result by fuzzy means only because we did not have a precise definition of an algorithm. If our definition were precise, we could reach the result with mathematical precision.

References

Davis1: Davis, Martin, Computability and Unsolvability, 1982. Dover Publications. New York.