Creativistic Philosophy: Exploring the Limits of Formalization, #11[1]—The "Halting Problem"

Andreas Keller

(Lally, 2013)[2]

1. Introduction

More and more computer programming is done by AI "coding agents." This leads to the question whether programming will be or can be completely automated, a question at the hart of AI and thus also of AI-based cognitive science, since "AI" may be redefined as "automatic programming."

We have seen part of the answer already in previous installments of this series, and we are going to look into it in more detail again in a later installment: every single algorithm encodes only a limited pattern or set of patterns, and since the total computable functions are not algorithmically enumerable, there is no universal algorithm that covers all possible patterns. Another part of the answer will emerge in the present installment, which focuses on the so-called "Halting Problem."

---

[2] The armless clock here symbolizes an infinite waiting time when a program does not halt for a given input.

2. The "Halting Problem"

One could—jokingly—say that in mathematics, something is called a "problem" only once it is solved, i.e., only once a proof, or a refutation, is known. Up to that point, it is called a "conjecture." The "halting problem" is of this type. This is a question of central importance to the theory of computability: namely, whether there can be an algorithm that, given any program as input, determines whether this program halts for all its inputs or not. So, is the property of programs to halt for all their inputs algorithmically "decidable"?[3]

The answer is known (so this is a "problem", not a conjecture), and it is "no." And after the preceding installments, we are now in a position to see quite easily why that is so. So, one possible proof is sketched in the following.[4]

In a former installment, I have introduced the idea of Gödel numbers, i.e., the idea that it is possible to code arbitrary data as natural numbers. This possibility means that any algorithm that maps some input data to some output data can be replaced by an algorithm that maps corresponding (natural) input numbers to (natural) output numbers.

So in that sense, if we would restrict a programming language to natural numbers as the only data type, we would not really loose anything, at least as far as computability theory is concerned. While this would not be useful for practical purposes, it makes proofs about what is or is not possible with computers much easier.

For this reason, some of the formalizations of the concept of an algorithm, i.e., of computability, consist of simple "programming languages" that are only dealing with natural numbers. An example is the formalism known as "recursive functions."

Programs are written in a programming language. They are strings of symbols. The (finite) set of symbols that can occur in a program written in a particular programming language is called an "alphabet" in computer science, even if it contains not only letters, but typically also digits, some special characters and "whitespace," for example, blanks and linefeeds.

The set of all strings that can be formed from the characters in a finite alphabet can be algorithmically enumerated. You may start with strings that only contain one

[3] In computability theory, and also some other branches of mathematics, the words "decide", "decidable", "decidability" are technical terms whose meaning is slightly different from everyday use. The word "decide" in this context does not mean to make a willful decision, but rather to determine whether a proposition is true or not.

[4] More formal, exact treatments can be found in textbooks on computability theory, such as (Rogers, 1987).

character, like “a”, “b” etc., then you continue with all strings of length 2 (“aa”, “ab”, “ac” … “ba”, “bb”, etc., then the strings of length 3, and so on. This way, all possible strings that can be formed from the symbols in the given alphabet can be systematically generated.

For any programming language there is a syntax, i.e., a set of rules describing which strings are programs in that programming language. For such a syntax, it is possible to write a program known as a “parser.” This is a program that, given an arbitrary string, can determine (“decide”) if that string is a program according to the rules of the programming language.

Since all strings that can be formed from the characters in a programming language’s alphabet can be enumerated and the parser can then decide for each of these strings if they are programs inside that programming language, the programs in the language can be algorithmically enumerated. We can think of this as a “pipeline” in which two programs are connected in series, so that the output of the first program serves as the input for the second. The second program, the parser, filters out all strings that are not programs. The series of its outputs then forms an algorithmic enumeration of all programs in that programming language.[5]

In installment #10, we have seen that the computable total functions of natural numbers are not algorithmically enumerable. “Total” here means that there is an output for every natural number you put in as an input value. Every algorithm that produces an enumeration of such functions, or the programs computing them, is incomplete with respect to the set of all computable total functions. This means that it is always possible to construct additional computable total functions not covered by a given enumeration.

The proof used the trick of “diagonalization.” As I noted, the proof relies on the enumerated functions to be total. Thinking of an enumeration of functions as a table in which each column represents a function and each row represents a natural number as an input, we could see that we get a contradiction if we assume the “diagonal + 1”-function is contained in the table as one of its columns.

But if some of the functions are not total, i.e. do not assign a value to every input, the proof no longer works if the field of the table where the contradiction occurs in the case of total functions could simply be empty.

Some programs do not halt after some (possibly very long) time for some inputs. For example, for some inputs they might go into a loop of computational steps

[5] There are much more efficient ways to enumerate all programs of a given programming language. It is possible to transform the rules of the syntax into a generator that does so systematically, without having to enumerate all possible strings first, but I am not concerned with efficiency here, just with theoretical possibility.

that is repeated forever without coming to an end. Such programs would not assign any value to the inputs where this happens. This means they compute functions that are not total.

A program computing a total function will run for a while for each input but will halt with the result after some time. If, however, the program does not halt for some inputs, for example because it goes into a never ending loop of computational steps, it will not assign a value to those inputs for which it does not halt.

So the total computable functions are exactly those for which an algorithm exists that halts for all its inputs.

Now let us return to the pipeline of programs we have considered above, consisting of a generator that produces all strings from a given alphabet and then a parser that filters out those that are programs of a given programming language. We consider here a language in which natural numbers are the only data type.

In turn, let us assume that we had a third program that takes programs as inputs and can decide for any program in its input if it halts for all its inputs or not. We assume that this third program would discard its input if it does not halt for all its inputs and lets it through otherwise. So the first program in the pipeline enumerates all strings that can be formed from an alphabet, the second filters out all those that are not programs and the third filters out all those programs that do not halt for some of their inputs.

The output of this third program in the pipeline would then be an enumeration of all programs that halt for all of their inputs. If the programming language is one that is restricted to processing only natural numbers (e.g., the formalism of recursive functions), these would exactly be all the total functions. So this pipeline of programs would produce an algorithmic enumeration of all computable total functions.

But we have seen already that this is not possible. So the third program in the pipeline that decides if the string in its input is a program that halts for all of its inputs must be impossible.[6]

We can conclude from this that it is not possible to write a program that can check, for arbitrary programs, whether they halt for all inputs. In other words, every program that can decide for other programs if they halt for all their inputs must be incomplete or partially incorrect, i.e., either it will not yield a result at all or it will not yield a correct result for some inputs.

[6] The enumeration would contain instances of several programs computing the same function, i.e. some columns in the table of output values would be the same, but this does not matter here.

3. Consequences

A direct consequence of this is that it is also impossible to write a program that can decide, for arbitrary pairs of programs, whether they are equivalent. Two programs are equivalent if they produce the same outputs for the same inputs, and halt or do not halt for the same inputs. If we had a program that could check the equivalence of arbitrary programs, we could use it to construct a program that decides for arbitrary programs if they halt for all of their inputs: to check if a given program p halts for all inputs, that program would put it inside another program that runs it, discards its output when it halts and then instead produces the output 1. We can then check if this program is equivalent to one that just always directly produces the output 1 (i.e. a program calculating the constant function with the output value 1). Both programs would be equivalent if and only if the program p halts for all inputs. Since we know that deciding by algorithm whether an arbitrary program halts for all its inputs is impossible, we can conclude that deciding the equivalence of arbitrary programs is also impossible.

Another consequence that I can only briefly sketch here and which I will look at in more detail in a later installment, as I indicated in the Introduction to this essay, is that it is not possible to write an algorithm that decides if a given program is correct with respect to some specification. Increasingly, AI systems are used for programming. These are algorithms that produce programs according to some specification. The specification could be given in the form of some formal language or, as is increasingly the case, in the form of natural language.

If these algorithms are guaranteed to produce only correct programs, they must be incomplete, i.e., they will not produce a result for some specifications. This follows from the impossibility of enumerating all computable total functions. On the other hand, if a programming algorithm is guaranteed to produce a result in every case, this result must sometimes be wrong, i.e., either not halt in some cases or produce a wrong result.

Such systems would be extendable, so every single mistake or gap in their capabilities could be corrected. Some such systems have shown impressive capabilities. But all of them are necessarily incomplete and/or error-prone. Correctness of software is not formalizable in the sense that it is not possible to produce an algorithmic system for automatic programming that does not either have gaps (i.e. is incapable of yielding a result for some tasks) or produces wrong results in some cases. And unlike creative humans, no such system, as long as it is an algorithm, would be capable of correcting all such mistakes or closing all such gaps on its own.

These limitations of AI, and specifically of automatic programming and automatic program checking are fundamental. If the system is an algorithm, it is

limited in this way. It seems that not all people in the AI industry are aware of the fundamental nature of these limitations.

REFERENCES

(Lally, 2013) Lally, D. "Handless Clock." *Wikimedia Commons*. Available online at URL = <https://commons.wikimedia.org/wiki/File:Handless_clock_-_geograph.org.uk_-_3465720.jpg>.

(Keller, 2024). Keller, A. "Intelligence and Paraintelligence." *Borderless Philosophy* 7: 62-108. Available online at URL = <https://www.cckp.space/single-post/bp-7-2024-andreas-keller-intelligence-and-paraintelligence-62-108>.

(Rogers, 1987). Rogers, H. *Theory of Recursive Functions and Effective Computability*. Cambridge MA: MIT Press.