Creativistic Philosophy: Exploring the Limits of Formalization, #10[1]—Total Functions, Innovation, and Cultural Bandwidth

Andreas Keller

(Collins, 2010)[2]

1. Total Functions

Up to this point, we have been looking at computable predicates of natural numbers. Such predicates assign a truth value (true or false) to each natural number. We are now instead going to look at computable functions that assign a natural number to natural numbers.

---

[2] The image shows a Sumerian cuneiform tablet as an example of early writing. Writing expanded the information storage capacity of civilization, making new kinds of cultural phenomena possible (e.g., in this case, bureaucracy).

In a previous installment, I introduced the notion of a Gödel number. We learnt that it is possible to code arbitrary data as natural numbers.[3] So if we construct a programming language in which the only data type is natural numbers, we are, in a way, not losing anything, because any program that has different types of data as inputs and outputs can be mapped to such programs restricted to natural numbers by means of Gödel numberings. In fact, some of the formalisms that have been introduced to formalize the concept of algorithm, e.g. the so-called "recursive functions", are only defined for natural numbers.

A computable function that has natural numbers as inputs and outputs is called a "computable total function" if it assigns a value to every natural number. So there are no gaps in it, i.e. there is no input value to which a function value is not assigned.

In previous installments, I have shown that the computable predicates on natural numbers cannot be algorithmically enumerated: for each such enumeration, we can construct another predicate that is not contained in the enumeration, so every algorithm that generates algorithms for such a set of such predicates is incomplete. So, the set of all computable predicates on natural numbers cannot be described in terms of a single algorithm or formal theory. It is not formalizable in this sense.

The trick to construct a new predicate not contained in a given enumeration was to think of the enumeration of predicates as a table and to construct the diagonal of that table and then modify the "output". To modify the output of an algorithm producing truth values, i.e. of a computable predicate, we can simply flip the truth value, i.e. replace true by false and vice versa.

We can easily see that we can apply the same principle (construct the diagonal and then modify the output) to computable total functions as well and prove that way that it is impossible to enumerate all computable total functions with a single algorithm. Let us look as a very simple example.

A constant function maps every natural number to the same number. For example, we can construct a simple algorithm that takes any natural number as input, discards this input and always produces 1 as its output. Let us call this function (and the program calculating it) "C1". It is not difficult to write a program that produces algorithms for all constant functions, by replacing the number used as output with each subsequent number, so we get a Program "C2" that always outputs 2, "C3" that always outputs 3, and so on. If we think of the result as a table, it looks like this (where each row represents the inputs and each column represents one of the functions):

[3] Note that this idea can also be used to represent truth values. For example., we could code "true" as 1 and "false" as 2 and thus do all the things we did in previous installments with predicates with total functions on natural numbers instead.

	C1	C2	C3	C4	…
1	1	2	3	4	…
2	1	2	3	4	…
3	1	2	3	4	…
4	1	2	3	4	…
5	1	2	3	4	…
6	1	2	3	4	…
7	1	2	3	4	…
8	1	2	3	4	…
9	1	2	3	4	…
10	1	2	3	4	…
…	…	…	…	…	…

In the graphics, the diagonal has been marked in green. The set of constant functions is clearly enumerable by algorithm. The diagonal can also be viewed as a function. It assigns 1 to the input 1, 2 to the input 2 and so on. This function is known as the identical function, the function that assigns each natural number to itself. Clearly, this function is also computable: to calculate the value for a given number n, we can produce the algorithm for Cn and then apply it to n.[4]

In the current case, it is clear that the identical function, i.e., the diagonal function of our table, is not contained in the table since the table contains just all the constant functions and the identical function is not constant. However, we could construct another table (i.e. enumeration of functions) that does contain its own diagonal. So to make sure we get a new function, we add a second step that corresponds to the flipping of truth values in the case of predicates. We cannot just "flip" the numbers, of course, but we can apply an operation that guarantees that the output is not equal to the input. The simplest way to do so is to just add 1.[5]

In the example, this results in a function that produces the output 2 for the input 1, the output 3 for the input 2, and so on. We can easily see that this function cannot be part of the given enumerations of functions. If it where, there would have to be a number m so that this "diagonal + 1" function is equal to Cm. If we apply this function to m (i.e. we look at its value on the diagonal), we see that the value of Cm(m) would have to be equal to Cm(m) + 1. This is not possible.

[4] Of course, we can come up with a much simpler algorithm for the identical function: simply take the input and put it into the output. But in the example we can see that we can always produce an algorithm for calculating the diagonal of an enumeration of total functions from that enumeration, even if that algorithm might not be the simplest one.

[5] There are other operations that guarantee that the output is unequal to the input, i.e. always adding 2. Adding 1 is perhaps just the simples one.

So the "diagonal + 1" function of any algorithmic enumeration of total functions cannot be part of that enumeration. As a result, the set of all computable total functions cannot be algorithmically enumerable. Assuming you had an algorithm that enumerates this set, i.e., produces a table that contains every computable total function in some column would yield a contradiction. Its "diagonal + 1" function would have to be in the table in some column with number m and the value of that function at the diagonal, i.e. in the m-th line of the m-th columns, would than have to be equal to itself plus 1. So assuming the existence of such an enumeration leads to a contradiction, which means that such an enumeration cannot exist.[6]

The proof relies on the enumerated functions to be total. If we include functions that are not total, i.e., that do not assign a value to some inputs, the proof no longer works because the field of the table where the contradiction occurs in the case of total functions could simply be empty. I will come back to this topic in a future installment when I deal with what is known as the "halting problem."

2. Total Functions, Storage Space, and Innovation

Before I continue with additional mathematical ideas like the "halting problem," let me make a detour into the philosophy of civilization, shedding some light on a general trend of human cultural development through the ages: the trend towards larger and larger information storage capacity, or to an increasing "cultural bandwidth."

By means of Gödel numberings, computable total functions on natural numbers can be thought of as representing programs processing arbitrary data. Each such function maps natural numbers to other numbers, generating a sequence of output numbers or output data. We can think of such sequences as patterns that represent some knowledge.

As we have seen, it is not possible to algorithmically enumerate all programs calculating total functions, i.e. to algorithmically enumerate all possible patterns. We can always construct another pattern (in fact, many of them), by means of operations like producing the diagonal and adding 1.[7]

The algorithm that produces an enumeration of total functions, i.e. an enumeration (table) of a certain type of patterns, would be a program written in some

[6] A more exact step by step version of this proof can be found in (Keller 2024). A textbook version of the proof can be found in (Rogers, 1987: p. 12).

[7] Such operations are called "productive functions" by mathematicians. Computable total functions form what is known as a "productive set." If a set is productive, then for any algorithmic enumeration of a true subset of it (in our example: the constant functions), you can always produce a new element of the productive set not contained in that enumeration by applying the productive function to the enumeration.

programming language. A program is a finite "text", i.e. a string of characters from an "alphabet", i.e. a set of symbols that can occur in programs of that programming language. The program would have a certain length. So it is possible to describe all the patterns of that type by a program of that length. We have seen the constant functions as a type of simple patterns.

Applying the productive function "diagonal + 1" to this enumeration algorithm produces a new function that is not of that type. It is new with respect to the patterns we have seen before. We can add it to the enumeration (remember the operation of shifting the table and inserting the new function in the first column that we saw in a previous installment dealing with predicates). The program enumerating the new table would in most cases have to be more complex. It will occupy a larger space.

That the set of all computable total functions is not algorithmically enumerable then means that it would require an infinite amount of program text to describe that set, i.e., to produce all the programs and only those programs calculating computable total functions. So there is always another different pattern, and another one, and another one that follows a different "law," i.e., is calculated by a different program that works in a different way. You can always add more patterns, but there is no universal computable pattern that covers them all.

So, extending the available storage space enables us to describe more complex patterns that we could not describe before. Moreover, it enables us to do things that are qualitatively new with respect to the algorithm (knowledge in the wider sense of the word) that we had before.

Every algorithm we arrive at will be incomplete in turn, i.e., it will not be able to produce all possible computable patterns, but we can always extend it. Extending it will, in general, require additional storage space.[8] The set of all possible computable patterns cannot be formalized, i.e., described in terms of a single algorithm or formal theory.

If we think of these algorithms as part of our culture, this means that human culture cannot be completely formalized.[9] We can always add something qualitatively

---

[8] Occasionally, there may be cases when several distinct sub-algorithms calculating different patterns can be unified into one algorithm, leading to reductions of complexity. But the unification of all such sub-algorithms into one universal one is impossible.

[9] You can think of the culture as the totality of all information that is passed, non-genetically, from one generation to the next. This knowledge, as far as it can be made exact and explicit, could be encoded in terms of formal theories or algorithms, so the totality of a culture's knowledge can be described as a set of patterns or algorithms. Of course, in reality, human culture has a "creative edge," where it is not possible to make the knowledge exact and explicit, and where we are in direct interaction with social and physical reality. Creative processes at this edge lead to change and new information and it is this process that cannot be described in terms of algorithms.

new, for example, we can add another program to the internet that describes a pattern not covered by all the algorithms that were there before. As long as we can extend the storage space, we can produce phenomena that are qualitatively new with respect to anything that was there before. We can innovate.

It is therefore not astonishing that there is a trend towards increasing information storage capacity in the history of human cultures. Key inventions, like writing, printing, libraries, computers etc. lead to larger and larger information storage capacity, enabling innovation, i.e. the introduction of new cultural or technological phenomena that where qualitatively new with respect to what was there before.

So, creativity, which was defined as the ability to move out of the scope of any previous formal theory/algorithm describing human cognition or culture, seems to be linked to increasing the "bandwidth" of culture, i.e., the total amount of information that can be passed on from generation to generation.

I suspect that this relationship holds true in general, even if our thought processes do not proceed in terms of computable total functions, which here only serve as a very simple model of the process of increasing the complexity and as a way to see that innovation is really possible (by adding new programs to the set of all known algorithms).[10]

As our technology gives us access to larger and larger information storage capacity, we can expect to see more innovations, i.e., phenomena that are new with respect to what was there before. We could also say: we should expect more disruptions. If our civilization can survive these repeated disruptions in the long term remains to be seen.

[10] You can think of the website where this article is published as a system of algorithms that outputs text and images in response to user actions. These algorithms encode a specific body of knowledge or cultural information and can be expanded at any time, for example by the administrator publishing this article there. Sometimes it is necessary to add additional storage space.

REFERENCES

(Collins, 2010). Collins, G. "Issue of Barley Rations." *Wikimedia Commons*. Available online at URL = <https://commons.wikimedia.org/wiki/File:Issue_of_barley_rations.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.