Creativistic Philosophy: Exploring the Limits of Formalization, #5[1]—Enumeration and Incompleteness

Andreas Keller

(Novák, 2005)[2]

In this installment of my creativistic philosophy series, we will slowly begin to venture into philosophically interesting territory. But before we can do so, we have to continue our more technical considerations for a few more paragraphs.

---

[2] A vacuum cleaner can be used to clean everything inside an apartment—except for its own inside. This can be compared to the inability of algorithms and formal theories to access themselves, which we will begin to explore in this installment.

In the last two installments[3] we have been looking at simple properties of natural numbers, like "odd," "even," "smaller than or equal to 3," and so on. Each such property can be used to generate a sequence of statements on all the natural numbers, like "1 is odd," "2 is not odd," "3 is odd," etc. A statement like "2 is not odd" can be written as "not (2 is odd)." In place of the numbers, mathematicians will put a variable, so these statements have the general form "n is odd." Mathematicians would commonly write such a statement as "odd(n)," i.e., the name of the predicate, followed by the numbers (or variables standing for them). A statement like "6 is divisible by 3" would, in such a notation, look something like "divisible(6, 3)." Such a predicate has two "arguments." It can be used to define one-argument predicates like "divisible_by_3(6)." In the previous installments we have looked mainly at one-argument predicates, and we will continue to do so.[4]

In the previous installments, I have been looking at algorithms producing enumerations of algorithms for one-argument predicates. I have visualized these by tables where the columns represent predicates and where a black table field represents the output "true" while a white one represents the output "false". We have seen in the previous installment that such algorithms (or the corresponding tables) can be extended and merged.

If we put a new column into such a table, it might be equal to another column that is already there, i.e., the two columns will agree in all places. However, there is a way we can generate, for a given table, a new predicate that is guaranteed not to be contained in it yet.

Let us call the algorithm for the table "A" and use the index-notation introduced in a previous installment to denote the column-predicates. Then we have column predicates A_1, A_2, A_3, etc. or in general, i.e., for an arbitrary natural number n, A_n. Now we proceed in two steps:

We define a new predicate D as $D(n) := A_n(n)$. This means that in order to calculate the value of the predicate for the number n, we go to the nth column of the table and take the value in the nth row, i.e., the value assigned to n by the predicate A_n. This predicate D practically is the diagonal of the table: It assigns the value $A_1(1)$ to the number 1, $A_2(2)$ to the number 2, and so on. The diagonal will cross all the columns and agrees with each column A_n in the nth row. Since A is computable, i.e.,

[3] (Keller, 2025a) and (Keller, 2025b).

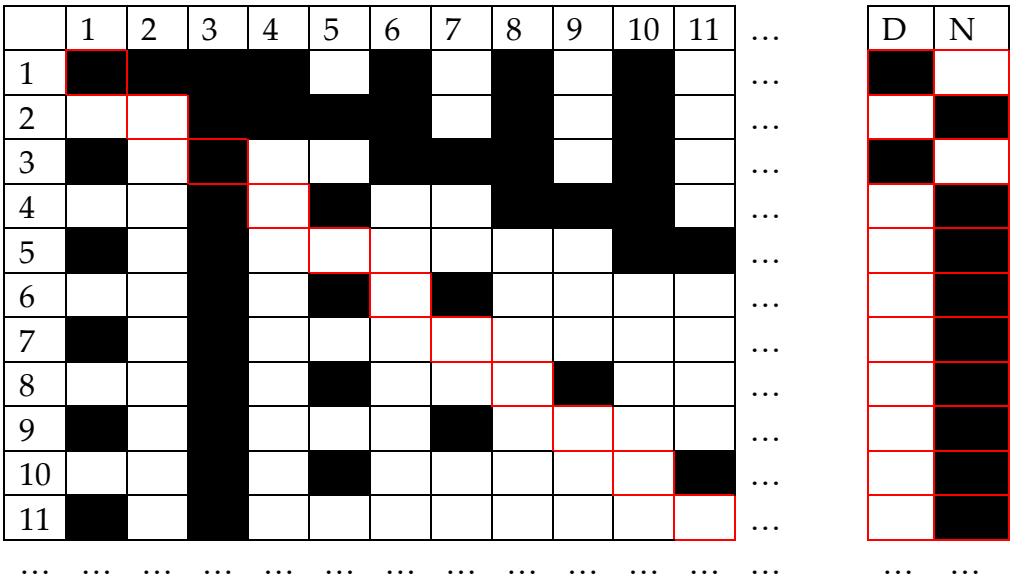
[4] In the tradition of mathematics, there is a tendency to use one-letter-names or simple symbols for functions, predicates and other operators and to exploit other alphabets, like the Greek alphabet, to have enough names. In the computer science tradition, on the other hand, names are normally strings of letters, digits and some special characters, like the underscore, where the first symbol in such an "identifier" is usually a letter. This is a convention widespread in programming languages. Coming out of the computer science tradition, I will rather use such letter sequences than introduce different alphabets or fonts.

an algorithm, we can easily construct an algorithm for D from it, i.e., D is also a computable predicate.

We now define another predicate N by negating D. So, N(n) := not(D(n)), which means N(n) equals not(A_n(n)). The not-operator just flips the truth values, replacing true by false and false by true. This is also a computable operation, so N is also a computable predicate.

Since D(n) is equal to A_n(n), N(n) is guaranteed to be different from A_n(n), so N differs from all predicates A_n in at least one place.

The following graphic illustrates the operation: it takes the last example from the previous installment as the algorithm A. The diagonal is marked in red. The predicate D, corresponding to that diagonal, is shown besides the table, followed by its negation N. N is guaranteed not to be contained in the original table A.



This way, if we have an algorithm enumerating computable predicates on the natural numbers (i.e., producing such a table), we can always produce another predicate not contained in that enumeration.

As a consequence, no enumeration of computable predicates can ever be complete, i.e., yield all computable predicates. If we assume we had a complete enumeration C, we can produce its "negated diagonal predicate" N. If C is complete, this predicate should already appear somewhere in the enumeration, i.e., there must be some natural number m so that N equals C_m. Then if C_m(m) is "true," C_m(m) must be "false" and vice versa, because N is defined as the negation of the diagonal, i.e., it flips the value of C_m(m). In other words, the assumption that a complete algorithmic enumeration of all computable predicates is possible leads to a contradiction.

Therefore, we conclude that the computable one-argument predicates on natural numbers are not algorithmically enumerable.

We can also formulate this result this way: the natural numbers have more computable true properties than can be computed by any single algorithm. No matter how complex and convoluted that algorithm will be, we can always construct a new predicate that is not covered by it yet.

Interestingly, the new predicate that the algorithm cannot produce is constructed from that algorithm itself!

Since algorithms and formal theories are equivalent, we could also say: the natural numbers have more true, computable properties than can be derived in any single formal theory.

Of course, as we have seen in the previous installment, we can always extend the algorithm or formal theory we are looking at. We can take the new predicate and merge it into the algorithm, e.g., as a new first column. This gives us a new, usually more complex algorithm that is more comprehensive, i.e., it computes all the statements the old algorithm could produce, plus some new ones.

But of course, this extended algorithm will be incomplete as well because we can apply the negated-diagonal operation to it in turn.

You might think: can't we somehow build that negated-diagonal operation into the algorithm? Well, yes, we can. It can be formulated as an algorithm and built into our algorithm. We can even formulate the algorithm in such a way that it would apply the operation of negating the diagonal and merging it into the original table repeatedly. This would lead us to some more comprehensive algorithm that produces more statements than the original one. However, no matter how tricky and sophisticated the way we do this, we could then apply the negated-diagonal operation to that sophisticated algorithm in turn and produce from itself a new predicate it does not produce. So, it is impossible to reach completeness that way. Somehow, the algorithm cannot grasp itself. This situation has been compared with the inability of a vacuum cleaner to clean its own inside.[5]

Actually, this result should not be so surprising. Any algorithm can be represented by a finite string of symbols, i.e., a finite text in some programming

[5] There appears to be something strange here. The table is produced by an algorithm. The new predicate is determined by it and it looks like it is produced by an operation ("take the diagonal and flip the truth values) we could also formulate by an algorithm." Yet we still cannot use it to produce a complete algorithm. I will look into this strange point in a later installment. We are also going to see that we can produce further new patterns that are not completely determined by the given algorithm.

language. A finite text contains only a finite amount of information. Every predicate can be viewed as a pattern and the table produced by an algorithm enumerating such column patterns can be viewed as a "super-pattern."

It should not surprise us too much that the finite information contained in a finite program text can only describe a limited set of patterns and that some other patterns always can be constructed that are not generated by it. The set of patterns generated by an algorithm is not necessarily finite because the algorithm might contain repetitions (programmable by programming constructs like loops, backward jumps or recursion), but such program structures can only, in each instance, generate repetitive patterns of certain types and are limited in this sense. There are always other patterns possible which would require a different program structure of an algorithm generating them.

"But wait," somebody might say, "can we not replace the algorithm by an AI which would reach completeness by means of its intelligence?"

The answer is: yes, we can put an AI model into our algorithm or use one as our algorithm, but every AI model is just an algorithm, although perhaps a complex, convoluted one. The line of thought presented above applies to all algorithms, not just to simple ones. It does not depend on the complexity of the algorithms at all. So, the negated-diagonal operation can be applied to that complex algorithm containing an AI model in turn, producing something that AI model was not able to produce. I am going to have a deeper look into this matter in later installments.

Let us summarize what we have learnt:

Every algorithm contains a finite amount of information. This finite information can only describe a limited set of patterns. There are always other patterns it cannot produce.

Specifically, for each algorithm enumerating a set of computable one-argument predicates on the natural numbers, we can construct an algorithm for a new predicate not contained in that enumeration.

We do so by first producing the "diagonal" of the enumeration and then negating it, i.e., flipping the truth values.

This way, we produce, from the enumerating algorithm itself, a new computable predicate that is not contained in that given algorithm.

This means that every algorithmic enumeration of the computable predicates must be incomplete. The set of all computable predicates on natural numbers cannot be algorithmically enumerated.

Since algorithms and formal theories are equivalent, this means that no single formal theory can produce all computable (provable), true statements about the natural numbers. The natural numbers have more true properties than can be derived (proven) in any single formal theory.

We can always merge the new algorithm for the new predicate into the existing algorithm (i.e., extend the formal theory). But the resulting algorithm is going to be incomplete in turn because we can produce the negated diagonal for it in turn, and so on.

If we attempt to build this operation into the algorithm itself, we might get a more comprehensive (and more complex algorithm), but this algorithm will be incomplete in turn.

Putting an artificial intelligence into our algorithm does not help to make it complete since the AI is also just an algorithm, although a very complex one

If an AI cannot help us to get completeness, despite all of its alleged intelligence, is the label "intelligence," let alone "general intelligence," appropriate for such systems? Personally, I think it is not and the term "AI" for such systems is an exaggeration. I would expect from a "general intelligence" that it can find and apprehend arbitrary patterns in arbitrary data. But we have seen already that each algorithm can only express a limited set of patterns and that there are always other patterns outside of this limited set. Here we get a glimpse already of the fact that all algorithmic AIs are incomplete, i.e., that they have systematic blind spots. This points to the unattainability of general or true intelligence by means of algorithms, a topic of future installments.

REFERENCES

(Keller, 2025a). Keller, A. "Creativistic Philosophy: Exploring the Limits of Formalization, #3 —Patterns and Algorithms" *Against Professional Philosophy*. 7 September. Available online at URL = <https://againstprofphil.org/2025/09/07/creativistic-philosophy-exploring-the-limits-of-formalization-3-patterns-and-algorithms/>.

(Keller, 2025b). Keller, A. "Creativistic Philosophy: Exploring the Limits of Formalization, #4—Extending Algorithms" *Against Professional Philosophy*. 28 September. Available online at URL = <https://https://againstprofphil.org/2025/09/28/creativistic-philosophy-exploring-the-limits-of-formalization-4-extending-algorithms/>.

(Novák, 2005). Novák, P. "Vacuum Cleaner." *Wikimedia.* (Petr Novák, Wikipedia). Available online at URL = <https://commons.wikimedia.org/wiki/File:Vacuum_cleaner.jpg>.