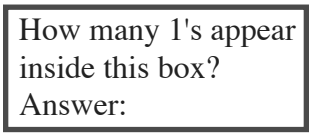
Boxes

Eric Hehner

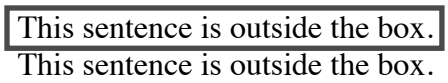
Introduction

Here is a simple question.



The answer is 1. You are invited to write the answer inside the box. But if you write the answer inside the box, you make the answer wrong. If you write the answer outside the box, it's right. (To prevent you from answering “one”, or “3–2” inside the box, I insist that the answer is an evaluated number, not an unevaluated expression.) It's a silly little trick, but it's also the heart of some famous mathematics.

Here is a simpler example of the same phenomenon. The exact same sentence appears twice; but one of them is true, and one of them is false.



This sentence is outside the box.

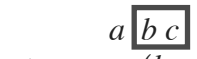
There's a town called Boxville, with some people in it. A person might admire some people, and not admire other people. A person might admire themself, or not. A person who admires themself is conceited; a person who does not admire themself is modest. The position of character judge for Boxville has one requirement: to admire the modest people in Boxville, and not to admire the conceited people in Boxville. No-one in Boxville can fulfill the requirement. That is because they would have to admire themself if and only if they do not admire themself. But someone outside Boxville can be the character judge for Boxville with no difficulty.

The Boxville story can be simplified. We just need one person; let's call him Boxy. If Boxy admires himself, he's conceited; if he doesn't, he's modest. He cannot be his own character judge because he would have to admire himself if and only if he does not admire himself. Anyone else can be a character judge of Boxy with no difficulty.

Expressibility

Imagine a large plane surface. On that surface you can draw two-dimensional boxes (rectangles). They can be any size you like. You can draw boxes anywhere you like. You can draw boxes inside boxes. But you cannot draw boxes whose boundaries cross. Now sprinkle variables (like x and y) anywhere you like. You can slide boxes and variables around, as long as you don't move anything into or out of a box (do not cross a box boundary). If a variable appears two or more times in the same space (you can draw a line from one to the other without crossing a box boundary) then you can delete all but one of them. If two boxes with the same contents appear two or more times in the same space then you can delete all but one of them. This is a formalism called Box Algebra, and those are the rules of box expression formation.

There are several ways to interpret box expressions. One interpretation is Binary (Boolean) Algebra. A box negates the expression inside it. An empty space represents true. An empty box represents false. Variables and boxes in the same space are conjoined. For example,

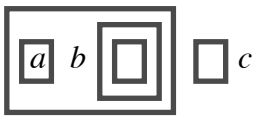


represents $a \wedge \neg(b \wedge c)$. Boxes and variables can be moved around in their space (not crossing a box boundary) because the order of conjuncts is not significant (conjunction is symmetric (commutative)): $a \wedge b$ is equivalent to $b \wedge a$. You can delete all but one of a variable or box and contents in the same space because conjunction is idempotent: $a \wedge a$ is equivalent to a.

Another interpretation of Box Algebra is Set Theory. The variables are the elements, and the boxes make the sets. The empty set $\{\}$ is an empty box. For example,

$\{\{a\}, b, \{\{\}\}\}, \{\}, c$

becomes



Boxes and variables can be moved around in their space because the order of elements in a set is not significant: $\{a, b\}$ is equivalent to $\{b, a\}$. Two identical boxes or variables in the same space can be reduced to one of them because the number of occurrences of an element is not significant: $\{a, a\}$ is equivalent to $\{a\}$.

Natural numbers can be represented with boxes. The obvious way, just lining them up, doesn't work. For example, the number 3 would be

but since the number of times an empty box occurs is not significant, this picture is equivalent to a picture with one empty box, or ten empty boxes. Here is one way that works. The number 0 is represented as nothing (empty space). To add 1, just enclose the previous number in a box. We get

0 is

1 is 

2 is 

3 is 

Provability

Kurt Gödel was a logician famous for his "incompleteness" theorems. He was a platonist mathematician (as are most mathematicians); as a consequence, he believed some mathematical sentences are true, for example, 1+1=2, and some are false, for example, 1+1=3. He was interested in mathematical formalisms. A formalism has rules to say what are the sentences in the formalism, and rules to say how to prove that a sentence is true. Using the rules, Gödel found a way to say whether a sentence in the formalism is provable. Then he constructed a sentence that denies its own provability. I'll call the formalism F and the sentence G (for Gödel), and write an informal version of it.

G: G is not provable in F.

(Gödel did not name sentences; he numbered them. But that doesn't matter. What matters is that Gödel created a formal mathematical version of this sentence.) Now, we need two definitions (in platonic terms):

A formalism is complete if all true sentences in the formalism are provable,
and incomplete if some true sentence in the formalism is not provable.
A formalism is consistent if no false sentence in the formalism is provable,
and inconsistent if some false sentence in the formalism is provable.

Let F be a consistent formalism. Consider F to be a box: sentences inside the box are expressed in F, and proofs inside the box are conducted in F. Outside the box, sentences and proofs can be informal, or in other formalisms.

G: G is not provable in F.
H: G is not provable in F.

Suppose G is false. That means G is provable in F. Therefore F is inconsistent. But F is consistent. So G must be a true sentence. That means G is not provable in F. Therefore F is incomplete.

Outside the box we have sentence H. Although H is not expressed in formalism F, its meaning is identical to G. Since G is true, H is also true. That is an informal proof of H. We have a sentence that is provable outside the box of formalism F, but cannot be proven inside the box of formalism F.

We expressed H informally, and proved it informally, for the sake of easy readability. But we could choose a formalism other than F in which to express and prove H.

Computability

A programming language is a formalism. Pascal and Python are programming languages. In this section, one formalism box contains all Pascal programs, and another contains all Python programs. We want a program in each box to answer the following question:

Pascal Can a Pascal program correctly answer “no” to this question?

Python Can a Pascal program correctly answer “no” to this question?

In the Pascal box, we can write a Pascal program that answers “yes”, but that answer says that “no” is the correct answer, so that's wrong. We can write a Pascal program that answers “no”, but that answer says that “no” is an incorrect answer, so that's wrong too. We cannot write a Pascal program to correctly answer the question. But in the Python box, we can write a Python program that answers “no”, and that answer is correct, because no Pascal program can correctly answer “no”. If we translate the Python program into Pascal, a program that was correct before translation becomes incorrect after translation. This is not because Python is a more “powerful” programming language than Pascal, able to answer questions that Pascal cannot answer. It is because Python programs are outside the Pascal box, and Pascal programs are inside the Pascal box. We could just as well tell the story switching the two languages.

Here is a more interesting example of the same phenomenon. Write a program to say whether the program you are writing is in Pascal. Let's write it in Python. There are two ways to write this program. The easy way is to write a print statement in Python that prints “no” because that is the correct answer. The hard way is to give the program access to its own text, perform the lexical analysis and parsing and type checking and so on, just as a compiler would do, and then

print the answer, which will be “no”. Next, we want to translate our Python program into Pascal. If we programmed the easy way, the translated program prints “no”, which is now incorrect. If we programmed the hard way, the translated program accesses its own text, does the analysis, and prints the correct answer, which is “yes”. The translation either prints the same answer, but now it's incorrect, or it prints a different answer, which is now correct.

Our last example is the famous halting problem. Write a function named *halts* with one parameter that names a Pascal procedure. The function result should be **true** if execution of the named procedure terminates (in finite time), and **false** if execution of the named procedure goes on forever. We provide a dictionary of Pascal function and procedure definitions so that *halts* can look up a name in the dictionary and find the corresponding function or procedure definition. Now consider Pascal procedure *twist* :

```
procedure twist;
begin
    if halts ('twist') then twist
end
```

Its execution calls *halts* ('*twist*') to determine if execution of *twist* halts. If *halts* ('*twist*') says **true** , meaning that it does halt, then *twist* calls itself, creating an infinite loop without termination. If *halts* ('*twist*') says **false** , meaning that it does not halt, then execution of *twist* halts. Either way, *halts* is wrong. We cannot write a Pascal program to determine halting for all Pascal programs.

What happens when we move out of the Pascal box? If we then move into the Python box, we find, by the same argument, that there is no Python program to compute halting for all Python programs. The argument was first made by Alan Turing [5] in 1936 for the Turing-Machine language. There is no program in any programming language to determine halting for all programs in that same language.

Could there be a Python program to compute halting for all Pascal programs? According to orthodox computer science, the answer is no. The reason given is that if we could write a Python program to compute halting for all Pascal programs, we could translate it into Pascal, creating a Pascal program to compute halting for all Pascal programs. And we know there can't be such a Pascal program. So there can't be such a Python program. This argument is wrong. As we have already seen, translating a program that gives correct answers can change it to one that gives incorrect answers. Maybe we can write a Python program *halts* that correctly determines halting for all Pascal programs. Python programs cannot be called from Pascal programs, so there is no *twist* program in Pascal that calls Python *halts* . Suppose Python *halts* says that execution of *twist* terminates. When Python *halts* is translated into Pascal *halts* , and is called by *twist* , it incorrectly says that *twist* does not terminate. And that is why execution of *twist* terminates, as reported correctly by Python *halts* .

We have no proof that a Python program to determine halting for all Pascal programs cannot be written. And vice versa.

Conclusion

A box makes one simple distinction: it distinguishes what is inside it from what is outside it. It doesn't have to be rectangular; any shape will do. It doesn't have to be two-dimensional; one-dimensional or three-dimensional will do. It just has to distinguish inside from outside. Gödel's incompleteness theorems and Turing's incomputability proof can be explained as the difference between the inside and outside of a box. All of mathematics can be represented as sets, so all of mathematics can be represented in Box Algebra.

Acknowledgements

I saw the problem of how many 5's in a box in a posting by Daryl McCullough in a Google discussion group [3], and I simplified it to 1's. I thank Peter Olcott for drawing my attention to it. Gödel's incompleteness theorems are found in [0] and [1]. Box Algebra [2] is a special case of Boundary Algebra [4], which is the work of Philip Meguire of the University of Canterbury in Christchurch, New Zealand.

References

[0] Gödel, Kurt: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I, *Monatshefte für Mathematik und Physik* v.38 p.173-198, Leipzig, 1931.

[1] Gödel, Kurt: *On Formally Undecidable Propositions Of Principia Mathematica And Related Systems*, translated into English by B. Meltzer, with Introduction by R.B. Braithwaite, Oliver&Boyd, Edinburgh, 1962.

[2] Hehner, Eric: Boundary Algebra, hehner.ca/boundaryalgebra.pdf

[3] McCullough, Daryl: https://groups.google.com/g/sci.logic/c/4kIXI1kxmsI/m/hRroMoQZx2IJ

[4] Meguire, Philip: *Boundary Algebra: a Simpler Approach to Boolean Algebra and Sentential Logic*, 2023

[5] Turing, Alan: on Computable Numbers with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society* series 2 volume 42 pages 230-265, 1936; correction series 2 volume 43 pages 544-546, 1937