

## Does not Compute

Gerard J. Holzmann

Most of us who work with software know all too well how easy it is to make small mistakes that escape detection in tests, and that come back to haunt us later. Unfortunately, when you study formal software verification techniques, one of the first things you learn is that a foolproof method for analyzing your code to reliably prevent these types of unpleasanties does not exist. Worse, you learn that it cannot exist. Although you may not remember a proof, you've certainly heard of the halting problem. Alan Turing already showed in 1936 that there cannot be an algorithm that can decisively show if an arbitrary program will terminate on a given input or not.



*Figure 1 -- sample illustration (this one is likely copyrighted though, from the internet)*

There is a simple way to show that the problem, even if it wasn't completely unsolvable, is at least very hard. Take a look, for instance, at the following small C fragment.

```
assert(n>0);           // for positive n
while (n != 1)
{
    if (n%2 == 0)      // n is even
    {
        n = n/2;
    } else             // n is odd
    {
        n = 3*n+1;
    }
}
```

Does the while loop terminate for any positive value of  $n$ ? If you know the answer, you can be assured fame, because even though this question was first posed in 1937, the problem (known as the Collatz Conjecture) remains unsolved. For all values that anyone has tried, the computation does eventually reach the value 1, and the program stops, but that doesn't prove that this will always be true.

That is, if you take the problem as it was intended: as a problem in mathematics where  $n$  is a positive natural number. Things change if you take the problem as expressed in the C program shown here. Computers of course don't use natural numbers. Unless you take special precautions, they use bounded integers. The integer variable  $n$ , depending on the machine you work on, is either a 32-bit or a 64-bit signed value. That means that it is a bounded quantity, and we can examine the entire space of possible values it could represent. That also means that if the value of  $n$  gets too close to the value of `INTMAX`, it will overflow on the multiplication, and you can suddenly end up with a negative value for  $n$  inside the

loop. It's not hard to see that you can then end up with a non-terminating run that cycles through the negative number sequence: -20, -10, -5, -14, -7, -20 forever. Welcome to the world of software, where stranger things do tend to happen. And it is a nice reminder that there is a difference between computer science and mathematics, and if we want to build reliable systems we should be keenly aware of that.



Figure 2 -- sample illustration (this one is probably copyrighted though)

### Don't Stop me Now...

In many textbooks that cover the unsolvability of the halting problem you can find a wonderfully simple proof. The proof was first published in 1965, in a letter to the editor of the *Computer Journal* by Christopher Strachey titled "An Impossible Program." It consists of an even smaller example program than we used for the Collatz conjecture above [1]. We begin with the assumption that we have developed a function that when presented with the source text of an algorithm can tell if that algorithm will terminate after executing a finite number of steps, or not. That function, let's call it `halts(P)`, returns *true* if algorithm `P` terminates and *false* if it does not. For simplicity we'll also assume that `P` takes no further external inputs and is completely deterministic, so that its execution is fixed: it either always terminates or it always halts. If such a procedure `halts(P)` could exist, reasoned Strachey, then we could also write the following program:

```
L: if (halts(P)) { goto L; }
```

which says that if our function determines that `P` halts, this new program will loop, and vice versa. All is well so far, but now we ask the question; What happens if `P` is itself the program we apply this new program to? Now if our procedure says that `P` will halt, it will loop, and if it says that `P` will loop, it will halt instead. So it can never give a correct answer, and therefore, concluded Strachey, it cannot exist. We could of course also conclude more modestly that this argument shows only that the hypothetical function `halts(P)` cannot be applied to itself. Below I will argue that that is in fact the right conclusion.

### Bounds

Let's go back for a second to our earlier observation that computers are in fact not mathematical objects, but only a fairly crude abstraction of these. We're interested in the halting problem for practical reasons: can there be a way to formally verify software, or is it pretty much hopeless because we will always be up against unsurmountable barriers like the halting problem? Let's take a closer look. All computers that you've ever used, or that you are likely to use in the future, are finite objects with only a finite amount of memory. Yes, programs can use external memory, but even though there can be massive amounts of both internal and external memory, the amount of storage is never truly infinite.

So, any program that we may want to analyze is, for very practical reasons, always a finite object, performing computations that can be in finitely many configurations or states. Even though that number can be astronomically large: it is finite. An infinite computation, then, on that same down-to-earth computer system, can only exist if we visit at least some of the finitely many different states infinitely often: the computation must eventually repeat itself. That means that in principle, though perhaps not always in practice, one could monitor that system and record the finitely many states that are reached. In this way, we can indeed distinguish terminating computations from non-terminating ones, in a finite amount of time. Within a finite number of steps, the program either terminates or it revisits a state that was visited before. There is of course a third possibility and that is that the program attempts to use more memory than our finite computer can provide: it can overflow its stack or exhaust heap memory, with the likely result that the computation is aborted without producing a result. Again, the result is a finite computation.

There's not too much wiggle room here, except that I can still write Strachey's program that tries to apply this hypothetical monitor program to itself. And yes, that attempt must again fail, and the most likely outcome is that it will crash after having exhausted the available finite resources. After all, the monitoring program will likely need quite a bit more memory than the object being monitored. It is important to note though that this failure doesn't mean that the working of the monitor program would also be impossible for any *other* code (that is everything except itself). That's clearly not the case. Logic model checkers, for instance, exploit the fact that systems can be represented by finite models, and they have no trouble distinguishing finite from infinite computations in those models. So, what's up with Strachey's construction?

### The Liar's Paradox

To see what's going on we can fall back on a fairly substantial literature about logical paradoxes. Many of these paradoxes exist because they are self-referential. The famous "Liar's Paradox," for instance, was already discussed in the 4<sup>th</sup> Century B.C. by Eubulides, who was a student of Euclid. The statement "I am a liar" cannot be true, because then it would be a lie, and it cannot be a lie, because then it would be true. There are plenty of variants of this, that throughout the centuries have been pinned on the vagueness and imprecision of human language.

Things got a little more interesting when in 1902 Bertrand Russell came up with a variant of the same paradox that could be made quite precise, and expressed in a basic form of set theory. Russell famously wrote about the paradox in a letter to Gottlob Frege on 16 June 1902, after studying the 1<sup>st</sup> Volume of Frege's life's work "The Basic Laws of Arithmetic" (*Grundgesetze der Arithmetik*). Russell described the problem in a somewhat casual way as just "one point where I have encountered a difficulty" in understanding Frege's definitions [2]. The problem is today known as Russell's "Class Paradox."

It goes like this. Within Frege's framework we can define sets that have any type of element, including other sets. The set of toys, for instance, can contain a set of dolls or cars, etc. This means that we can also define the set of all sets. More interestingly, we can define the set of all sets that do not contain themselves as an element. Let's call that set S. And, you probably feel this coming now, the key question is now if set S contains itself or not. If it does, it should not, and if it does not, it should. We're back at that point again with a self-referential construct that contradicts itself. The solution in set theory comes down to forbidding that we can create self-referential constructs like this.



*Figure 3 -- sample illustration (this one is probably copyrighted, from amazon.com)*

### The Self-Shaving Barber

There are a couple of things worth noting here. One is that the paradox itself is quite compelling. There are many variations that have been thought of that all recreate the same conundrum. One that I like the best was also used by Russell. It goes like this. There is a village in Italy that has one barber. This barber is asked to shave every male inhabitant of the village who doesn't shave himself. The barber is of course also an inhabitant of the village, and unfortunately in this case, he is also a male. Does the barber have to shave himself? Again, there's no answer.

Now importantly, the conclusion from the barber's paradox is not that it is impossible to tell if a man shaves himself or not. And similarly, the conclusion from Russell's paradox is not that it is impossible to tell if a set contains another set or not. The problem is not the shaving and not set membership itself: it is the use of a self-reference that relies on its own answer.

So, is Strachey's construction any different? My vote is that it isn't, and that it therefore isn't an acceptable proof for the unsolvability of the halting problem. For full disclosure let me add here that I first tried to make this argument a couple of decades ago when I was a graduate student, and I didn't know any better. But, sometimes the passage of time doesn't make us any wiser, so I'm still stuck thinking about this one. What do you think?

### References

[1] C. Strachey, "An impossible program," *The Computer Journal*, Vol. 7, Issue 4, Jan. 1965, p. 313.

[2] J. van Heijenoort (Ed.), "From Frege to Godel: a source book in mathematical logic, 1879-1931," Publ. iUniverse, 1999. Lincoln, NE. Russell's letter to Frege and Frege's response appear on pp 124-128.