International Mathematical Forum, Vol. 13, 2018, no. 1, 9 - 13
HIKARI Ltd, www.m-hikari.com
https://doi.org/10.12988/imf.2018.71299

How to Make a Proof of Halting Problem More Convincing: A Pedagogical Remark

Benjamin W. Robertson[1], Vladik Kreinovich[1], and Olga Kosheleva[2]

Departments of [1]Computer Science and [2]Teacher Education
University of Texas at El Paso
500 W. University
El Paso, TX 79968, USA

Abstract

As an example of an algorithmically undecidable problem, most textbooks list the impossibility to check whether a given program halts on given data. A usual proof of this result is based on the assumption that the hypothetical halt-checker works for *all* programs. To show that a halt-checker is impossible, we design an auxiliary program for which the existence of such a halt-checker leads to a contradiction. However, this auxiliary program is usually very artificial. So, a natural question arises: what if we only require that the halt-checker work for *reasonable* programs? In this paper, we show that even with such a restriction, halt-checkers are not possible – and thus, we make a proof of halting problem more convincing for students.

Mathematics Subject Classification: 03D35 68Q01 97Q80

Keywords: halting problem, 10th Hilbert problem, Russell's paradox, set theory, pedagogical notes

1 Formulation of the Problem

Halting problem: reminder. A computer science degree means acquiring both the practical skills needed to design and program software and the theoretical knowledge describing which computational tasks are possible and which are not. Different programs include different examples of problems for which no computational solution is possible, but all of them include – with proof – the very first example of such a problem: the halting problem, according to which no algorithm is possible that, given a program p and data d, always checks whether p halts on d; see, e.g., [2].

Some textbooks describe this result in terms of Turing machines but, in our opinion, this result is much clearer to students when it is described in terms of programs – i.e., something with which are very familiar – rather than in terms of Turing machines, a new concept that they have just learned in the corresponding theoretical course and with which they are not yet very familiar.

Let us therefore concentrate on the formulation of this result in terms of programs.

How this result is usually proved. To come up with a proof, we first need to describe both the program and the data in terms of natural numbers. This description is almost straightforward. Indeed, in the computer, no matter what symbols we type, any program or data is represented as a sequence of 0s and 1s.

In principle, we can just take the corresponding sequence of 0s and 1s, and interpret it as a binary number: e.g., 100 would be interpreted as 4, 101 as 5, etc. However, this does not provide us with a perfect representation, since in this case, two different sequences of 0s and 1s, e.g., 1 and 001, are represented by the same natural number – and so, based on this number, it is impossible to uniquely reconstruct the corresponding program. To avoid this non-uniqueness, we can append 1 in front the sequence of 0s and 1s. This way, 1 becomes 11 which is the number 3, while 0011 becomes 1001 which is a different natural number 9.

Once this representation is agreed upon, we can then prove, by contradiction, that the desired halt-checker $h(p, d)$ is not possible. Indeed, if it was possible, then we could write the following auxiliary program:

```
procedure auxiliary(i):
    if h(i,i) then
       loop forever
    else
       return 0
```

where `loop forever` means invoking a non-halting while-loop, such as

```
while(true}) i = i;
```

The proof is straightforward. Indeed, our auxiliary program corresponds to some integer i_0.

- If this program halts on the number i_0, then $h(i_0, i_0)$ is true and hence, our auxiliary program loops forever – which contradicts to our assumption that it halts.
- Similarly, if the auxiliary program does not halt, this means that $h(i_0, i_0)$ is false and thus, our auxiliary program returns 0 – which contradicts to our assumption that it does not halt.

In both possible cases, we get a contradiction, which means that our assumption – that a halting program $h(p, d)$ is possible – is wrong. Thus, no such halting program is possible.

A natural question. What this proof shows that if we require that a program $h(p, d)$ correctly checks halting for *all* possible programs p and data d. To prove this, we consider a weird example of an auxiliary program – a program which was invented for the sole purpose of proving this result. What if we limit ourselves to program which are more reasonable (in some natural sense)? Will this result still hold?

This question makes perfect sense in view of the analogy with barber's paradox and Russell's paradox in set theory. Most textbooks emphasize that the main idea behind the above proof comes from the origins of set theory. The corresponding construction can be informally described by the known barber's paradox, when a military barber attached to a detachment is commanded to shave those and only those who do not shave themselves. This is clearly a paradox:

- if he shaves himself, then he is not allowed to do it, and
- vice versa, if he does not shave himself, then he is commanded to shave himself.

In set theory context, this paradox was first described by the famous philosopher Bertrand Russell who proposed to consider the set $S \stackrel{\text{def}}{=} \{x : x \notin x\}$ of all the sets that are not elements of themselves (being an element of oneself is not an impossibility: e.g., the set of all possible sets is clearly its own element). Here:

- If $S \in S$, then, since S is an element of the class of all sets that do not belong to themselves, we should have $S \notin S$.
- Vice versa, if S is not an element of S, then it should not have the property $S \notin S$ and thus, we would have $S \in S$.

In both cases, we have a contradiction.

This situation is indeed similar to the halting problem. However, in contrast to the halting problem, for sets, we do not make a radical conclusion that sets do not exist: it turns out that if we limit ourselves to reasonable sets, paradoxes disappear, and we have a very reasonable theory – actually, set theory is, at present, the foundation for all mathematics.

So, this analogy emphasizes the above natural question: what if we limit ourselves to reasonable programs – like in set theory, when we limit ourselves to reasonable sets — will we get a different result?

What we do in this paper. In this paper, we explain that – as is rather easy to explain in class – the halt-checking program $h(p, d)$ is not possible even if we require that it only work for reasonable programs.

2 Halt-Checking Is Impossible Even If We Limit Ourselves to Reasonable Programs

Main idea. The main idea behind our explanation is based on another algorithmically unsolvable problem which is often presented in theoretical computing classes – the problem of checking whether a given Diophantine equation is solvable.

A Diophantine equation is a equation of the type $P(x_1, \ldots, x_n) = 0$, where $P(x_1, \ldots, x_n)$ is a polynomial with integer coefficients, and we are looking for solutions in which all x_i are natural numbers. It is known that no algorithm is possible that would check whether a given Diophantine equation has a solution. This result is a solution to one of the 23 challenging mathematical problems that David Hilbert, on behalf of the world's mathematical community, presented to the 20 century mathematicians – this problem (No. 10 on Hilbert's list) was eventually solved by Yuri Matiyasevich in 1970; see, e.g., [1].

How to transform this result into a more convincing proof of the halting problem. For each polynomial P, we can use exhaustive search to see if the corresponding polynomial equation has a solution in natural numbers:

- we start with Stage 0, on which we check whether $P(x_1, \ldots, x_n) = 0$ for any tuple for which $\sum\limits_{i=1}^{n} x_i = 0$ – there is actually only one such tuples $x_1 = \ldots = x_n = 0$, so this checking is easy;
- then, we perform Stage 1, i.e., we check whether $P(x_1, \ldots, x_n) = 0$ for any tuple for which $\sum\limits_{i=1}^{n} x_i = 1$ – there are n such tuples, with $x_i = 1$ for some i and $x_j = 0$ for all $j \neq i$;

- after that, we perform Stage 2, i.e., we check whether $P(x_1, \ldots, x_n) = 0$ for any tuple for which $\sum\limits_{i=1}^{n} x_i = 2$;
- ...
- at Stage k of this algorithm, we check whether $P(x_1, \ldots, x_n) = 0$ for any tuple for which $\sum\limits_{i=1}^{n} x_i = k$;
- ...

At each stage, we check finitely many tuples.

If the polynomial equation has a solution, this algorithm will find it. Thus, this program is reasonable (definitely more reasonable than the program used in the usual proof of halting problem).

On the other hand, if the original equation does not have a solution, then this program will never halt. So, if it was possible to check whether any such program halts or not, we would then be able to tell whether a given Diophantine equation has a solution – and we know, from Matiyasevich's result, that this is not possible.

Thus, even if we limit ourselves to reasonable programs p, it is still not possible to have a program that would check, for each such program p and data d, whether p halts on d.

Comment. Of course, the impossibility of such a general result does not preclude us from sometimes being able to check whether a program halts: such checks are indeed possible for many specific classes of programs.

Acknowledgments. This work was supported in part by the National Science Foundation grant HRD-1242122 (Cyber-ShARE Center of Excellence).

References

[1] Yu. Matiyasevich, *Hilbert's 10th Problem (Foundations of Computing)*, MIT Press, Cambridge, Massachusetts, 1993.

[2] M. Sipser, *Introduction to the Theory of Computation*, Cengage Learning, Boston, Massachusetts, 2012.

Received: December 22, 2017; Published: January 5, 2018