Reconstructing the Halting Problem

Eric C.R. Hehner

Department of Computer Science, University of Toronto
hehner@cs.utoronto.ca

Abstract The halting problem has not been proven to be incomputable.

Introduction

To say "the halting problem is incomputable" means that no program can determine whether the execution of an arbitrary program terminates. The first proof that halting is incomputable was by Alan Turing [2]. The present paper is a drastically shortened version of [0], which includes many alternative proofs, and a thorough discussion of the problem.

I begin by presenting Turing's proof, but modernized in two respects. Instead of using Turing Machine operations, I use two modern programming language constructs: the conditional **if** ... **then** ... **else** ... **fi** , and the function/procedure call. To apply a function to the domain of programs, Turing encoded programs as numbers. Today, when one program is presented as input data to another program (for example, to a compiler or interpreter), it is represented as a text (character string), and that's the encoding I will use. These changes are standard in modern textbooks; they change nothing essential in the proof of incomputability. I refer to any program fragment (statement or expression) as a "program".

Halting Problem

Assume that halting is computable.

Define H to be a program with two input text parameters p and q such that $H(p, q)$ computes whether execution of the program represented by text p halts when provided with input text q . Eliding a few words:

If p halts on input q , then $H(p, q)$ halts with result $\top$ (true).
If p does not halt on input q , then $H(p, q)$ halts with result $\bot$ (false).

Define programs T and N , and text D , as follows:

$T(q)$ = (a program whose execution terminates on every input q)
$N(q)$ = (a program whose execution does not terminate on any input q)
D = " **if** $H(D, D)$ **then** $N(D)$ **else** $T(D)$ **fi** "

We place the definitions of H , T , N , and D in a dictionary (or library) of definitions. When identifiers (like H , T , N , and D) occur within a program, they are interpreted (or understood, or given meaning) by looking them up in the dictionary.

Argue: If $H(D, D)$ is $\top$, then D represents a program that is equivalent to $N(D)$, whose execution does not terminate, and so $H(D, D)$ is $\bot$. And if $H(D, D)$ is $\bot$, then D represents a program that is equivalent to $T(D)$, whose execution does terminate, and so $H(D, D)$ is $\top$. We have an inconsistency.

Conclude: the assumption was wrong; halting is not computable.

The two parameters (p, q) are a two-dimensional space, and point (D, D) is on its diagonal; for that reason, this proof is called a "diagonal argument".

Reconstruction

I am now going to reconstruct this proof slowly, with commentary. I begin with the following informal "definition".

> Let D be a program with the property that if its execution calumates, then its execution does not calumate; and if its execution does not calumate, then its execution calumates.

The word "definition" was placed in quotation marks because someone might very reasonably object that I have not defined anything; there cannot be such a program. Someone else might say that it is a definition because it has the form of a definition, even though it is an inconsistent definition. It is not important to me to decide between these two viewpoints; what is important to me is that we all agree that there is an inconsistency. The inconsistency does not depend on the meaning of "calumate". Let it mean any property of program executions; you may suppose it means "terminate", "take longer than 10 seconds", "print "hello" ", or any other property you care to imagine.

At this stage we have only an informal inconsistent definition, not yet any program purported to determine calumation. I now start to formalize, being careful to maintain the inconsistency.

> Define T, N, and D, as follows:
>
> T = (a program whose execution calumates)
>
> N = (a program whose execution does not calumate)
>
> D = **if** execution of D calumates **then** N **else** T **fi**
>
> Argue: If execution of D calumates, then D is equivalent to N, whose execution does not calumate. And if execution of D does not calumate, then D is equivalent to T, whose execution does calumate. We have an inconsistency.

If you have decided what property "calumate" is, you can easily fill in the definitions of T and N. If we can just formalize "execution of D calumates" then we will have a normal recursive definition in any programming language (except possibly for syntactic differences).

> Define H to be a binary expression that expresses whether execution of D calumates. Eliding a few words:
>
> If D calumates, then H has value $\top$.
>
> If D does not calumate, then H has value $\bot$.
>
> Define T, N, and D, as follows:
>
> T = (a program whose execution calumates)
>
> N = (a program whose execution does not calumate)
>
> D = **if** H **then** N **else** T **fi**
>
> Argue: If H has value $\top$, then D is equivalent to N, whose execution does not calumate, and so H has value $\bot$. And if H has value $\bot$, then D is equivalent to T, whose execution does calumate, and so H has value $\top$. We have an inconsistency.

As in Turing's proof, the argument here neglects the time to evaluate H, and that can make an important difference. For a discussion of that point, see [0].

Even if we allow the definition of H to employ all of mathematics, including quantifiers and higher-order functions, not limiting it to programming notations, there is still an inconsistency. The inconsistency was there before we introduced H; it was not introduced by H. But now we can start to shift the blame from D to H. We can say that D would be a perfectly good definition in any programming language, except that there cannot be such an expression as H. That's because we ask H to perform an impossible task: if it says $\top$ then it should say $\bot$; if it says $\bot$ then it should say $\top$. The problem is not that H must compute a well-defined but incomputable function. The problem is that H has an inconsistent specification.

Next I give H a parameter so that it can say, not just whether execution of D calumates, but whether execution of any program calumates. To do that, I have to encode programs as data. So I redefine D as text.

Define H to be a function with text parameter p that expresses whether execution of the program represented by p calumates. Eliding a few words:

If p calumates, then $H(p)$ is $\top$.

If p does not calumate, then $H(p)$ is $\bot$.

Define T, N, and D, as follows:

T = (a program whose execution calumates)

N = (a program whose execution does not calumate)

D = "**if** $H(D)$ **then** N **else** T **fi**"

Argue: If $H(D)$ is $\top$, then D represents behavior equivalent to N, whose execution does not calumate, and so $H(D)$ is $\bot$. And if $H(D)$ is $\bot$, then D represents behavior equivalent to T, whose execution does calumate, and so $H(D)$ is $\top$. We have an inconsistency.

Now D is completely innocent: it is just a text (character string). But function H has an inconsistent specification.

A parameter is often useful; it allows a function to be applied to many values, producing many results. But in our proof, we have just one value that we want to apply H to, and that's D. If a function is applied to only one value, there is no point in having the parameter; we may as well define H to tell us just about D, as we did previously. But maybe there's a psychological reason for the parameter. With it, we can say $H(\text{"}T\text{"}) = \top$ and $H(\text{"}N\text{"}) = \bot$, giving the reader the feeling that the specification of H is perfectly reasonable, before we consider $H(D)$.

Next I add a second parameter to H, and a parameter to each of T and N, to provide them with input. These parameters will be used just once each, so again there is no point in having them. Furthermore, input can always be replaced by some assignment statements at the start of a program, so it is not theoretically necessary. In our case, it doesn't actually matter what input we provide, because the calumation of T and N doesn't depend on their inputs, and therefore neither does the behavior represented by D. Since we have text D lying around, let's use it instead of defining a new value. Besides, calling $H(D, D)$ allows us to say it's a "diagonal argument"; calling $H(D, q)$ for any other value of q would not be on the diagonal, although it would work just as well for the proof.

Define H to be a function with two input text parameters p and q such that $H(p, q)$ expresses whether execution of the program represented by text p calumates when provided with input text q. Eliding a few words:

If p calumates on input q, then $H(p, q)$ is $\top$.
If p does not calumate on input q, then $H(p, q)$ is $\bot$.

Define T, N, and D, as follows:

$T(q)$ = (a program whose execution calumates on every input q)
$N(q)$ = (a program whose execution does not calumate on any input q)
D = " **if** $H(D, D)$ **then** $N(D)$ **else** $T(D)$ **fi** "

Argue: If $H(D, D)$ is $\top$, then D represents behavior equivalent to $N(D)$, whose execution does not calumate, and so $H(D, D)$ is $\bot$. And if $H(D, D)$ is $\bot$, then D represents behavior equivalent to $T(D)$, whose execution does calumate, and so $H(D, D)$ is $\top$. We have an inconsistency.

Even if we don't require H to be a program, even if we don't require H to be a computable function, there is still an inconsistency in its definition. But maybe the definitions are now sufficiently dressed up to hide the source of the inconsistency. So, before the argument showing inconsistency, we slip in an assumption, and after the argument showing inconsistency, we conclude its opposite, making a "proof by contradiction".

Assume there is life on other planets.

Define H to be a function with two input text parameters p and q such that $H(p, q)$ expresses whether execution of the program represented by text p calumates when provided with input text q. Eliding a few words:

If p calumates on input q, then $H(p, q)$ is $\top$.
If p does not calumate on input q, then $H(p, q)$ is $\bot$.

Define T, N, and D, as follows:

$T(q)$ = (a program whose execution calumates on every input q)
$N(q)$ = (a program whose execution does not calumate on any input q)
D = " **if** $H(D, D)$ **then** $N(D)$ **else** $T(D)$ **fi** "

Argue: If $H(D, D)$ is $\top$, then D represents behavior equivalent to $N(D)$, whose execution does not calumate, and so $H(D, D)$ is $\bot$. And if $H(D, D)$ is $\bot$, then D represents behavior equivalent to $T(D)$, whose execution does calumate, and so $H(D, D)$ is $\top$. We have an inconsistency.

Conclude: the assumption was wrong; there is no life on other planets.

Perhaps that assumption was too obviously irrelevant. Let's assume that calumation is computable, because that looks like it might be relevant, but it isn't relevant because it isn't used anywhere in the definitions and argument.

Assume that calumation is computable.

Define H to be a program with two input text parameters p and q such that $H(p, q)$ computes whether execution of the program represented by text p calumates when provided with input text q. Eliding a few words:

If p calumates on input q, then $H(p, q)$ halts with result $\top$.
If p does not calumate on input q, then $H(p, q)$ halts with result $\bot$.

Define programs T and N, and text D, as follows:

$T(q)$ = (a program whose execution calumates on every input q)
$N(q)$ = (a program whose execution does not calumate on any input q)
D = " **if** $H(D, D)$ **then** $N(D)$ **else** $T(D)$ **fi** "

Argue: If $H(D, D)$ is $\top$, then D represents a program that is equivalent to $N(D)$, whose execution does not calumate, and so $H(D, D)$ is $\bot$. And if $H(D, D)$ is $\bot$, then D represents a program that is equivalent to $T(D)$, whose execution does calumate, and so $H(D, D)$ is $\top$. We have an inconsistency.

Conclude: the assumption was wrong; calumation is not computable.

Calumation has not been defined; it could be any property of program executions (see [1]).

Finally, we replace calumation by halting, and we have reconstructed the proof we started with. But this "proof" that halting is incomputable has nothing to do with halting, and nothing to do with computability.

Conclusion

Unsurprisingly, if we apply a halting program to a program text that includes a call to that same halting program with that same text as argument, then we have an infinite loop. A mathematical version of it cannot escape the corresponding problem: either we leave the definition of the halting function incomplete, not saying its result when applied to its own program, or we suffer inconsistency. If we choose an incomplete definition of the halting function (we don't require it to apply to its own program), then we cannot require its program to apply to the halting program either, and we cannot conclude that it is incomputable. If we choose an inconsistent definition of the halting function, then it makes no sense to program it. Either way, the incomputability argument is lost.

References

[0] E.C.R.Hehner: Problems with the Halting Problem, *Advances in Computer Science and Engineering* v.10 n.1 p.31-60, 2013 March www.cs.utoronto.ca/~hehner/PHP.pdf

[1] H.G.Rice: Classes of Recursively Enumerable Sets and their Decision Problems, *Transactions of the American Mathematical Society* v.74 p.358-366, 1953

[2] A.M.Turing: on Computable Numbers with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society* s.2 v.42 p.230-265, 1936; correction s.2 v.43 p.544-546, 1937