# the Halting Collection

Eric Hehner

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

## Halting Problem

When Alan Turing laid the foundation for computation in 1936, he wanted to show what computation can do, and what it cannot do. For the latter, he invented a problem that we now call the “Halting Problem”. In modern terms, it is as follows.

> In a general-purpose programming language L, write a program that reads a text (character string) *p* representing a program in L, and reads another text *i* representing its input, and outputs *true* if execution of *p* with input *i* terminates, and outputs *false* if execution of *p* with input *i* does not terminate.

The problem cannot be solved; there is no such program.

This paper is a collection of ten interestingly different versions of the Halting Problem that I have encountered or invented.

## Turing's Proof

Here is the key paragraph from Turing's paper. To help the modern reader, I have added the square bracketed words. And when Turing says “machine”, he means “program”.

[start of Turing's proof] Let us suppose that there is a such a process; that is to say, that we can invent a machine D which, when supplied with the S.D [standard description] of any computing machine M will test this S.D and if M is circular [nonterminating] will mark the S.D with the symbol "u" [unsatisfactory] and if it is circle-free [terminating] will mark it with "s" [satisfactory]. By combining the machines D and U [universal machine, or interpreter] we could construct a machine H to compute the sequence beta' [a sequence that differs from the diagonal with U]. ... Now let K be the D.N [description number, or code] of H. What does H do in the Kth section of its motion? [What happens when H works on the representation of H?] It must test whether K is satisfactory, giving a verdict "s" or "u". Since K is the D.N of H and since H is circle- free, the verdict cannot be "u". On the other hand, the verdict cannot be "s". For if it were, then in the Kth section of its motion H would be bound to compute the first R(K-1)+1 = R(K) figures [R(n) is the number of terminating programs among the first n programs] of the sequence computed by the machine with K as its D.N and to write down the R(K)th as a figure of the sequence computed by H. The computation of the first R(K)-1 figures would be carried out all right, but the instructions for calculating the R(K)th would amount to "calculate the first R(K) figures computed by H and write down the R(K)th". This R(K)th figure would never be found. I.e., H is circular, contrary both to what we have found in the last paragraph and to the verdict "s". Thus both verdicts are impossible and we conclude that there can be no machine D. [end of Turing's proof]

The proof doesn't seem to mention any programming language, but implicitly it does. It talks about the standard description of a computing machine, which is a number that encodes a program. And Turing Machine programs can be numbered because they are in a language, the Turing Machine language, that has syntactic rules that enable us to enumerate programs. And then a diagonal program D is assumed to be in that same enumeration, so it's in the same language, and the halting program H is constructed from D, so it is also in the same programming language. Turing's proof proves that there cannot be a program in the Turing Machine programming language, running on a Turing Machine, that determines halting for all programs in that same language running on that same machine.

## Pascal Version

I cannot write a Pascal function to say whether the execution of any Pascal procedure halts, so instead I write the function header, and a comment to specify what the result of the function is supposed to be. Following that is a procedure to which the function can be applied.

```
function halts (p, i: string): boolean;
{ returns true if string p represents a Pascal procedure with one string input }
{ whose execution halts when given input i ; returns false otherwise }

procedure twist (s: string);
begin
    if halts (s, s) then twist (s)
end
```

We assume there is a dictionary of function and procedure definitions that is accessible to *halts* , so that the call *halts* ('*twist*', '*twist*') allows *halts* to look up '*twist*' , and subsequently '*halts*' , in the dictionary, and retrieve their texts for analysis.

Assume function *halts* has been programmed according to its specification. Does execution of *twist* ('*twist*') terminate? If it terminates, then *halts* ('*twist*', '*twist*') returns *true* according to its specification, and so we see from the body of *twist* that execution of *twist* ('*twist*') does not terminate. If it does not terminate, then *halts* ('*twist*', '*twist*') returns *false* , and so execution of *twist* ('*twist*') terminates. This is a contradiction (inconsistency). Therefore function *halts* cannot have been programmed according to its specification.

## No Input Version

This version is a simplification of the previous version. The input parameter is unnecessary to the proof, so it has been eliminated.

> **function** *halts* (*p*: **string**): **boolean**;
> { returns *true* if string *p* represents a parameterless Pascal procedure }
> { whose execution halts; returns *false* otherwise }
>
> **procedure** *twist*;
> **begin**
>     **if** *halts* ('*twist*') **then** *twist*
> **end**

We assume there is a dictionary of function and procedure definitions that is accessible to *halts* , so that the call *halts* ('*twist*') allows *halts* to look up '*twist*' , and subsequently '*halts*' , in the dictionary, and retrieve their texts for analysis.

Assume function *halts* has been programmed according to its specification. Does execution of *twist* terminate? If it terminates, then *halts* ('*twist*') returns *true* according to its specification, and so we see from the body of *twist* that execution of *twist* does not terminate. If it does not terminate, then *halts* ('*twist*') returns *false* , and so execution of *twist* terminates. This is a contradiction (inconsistency). Therefore function *halts* cannot have been programmed according to its specification.

## Specialized Version

This version is a further simplification. Although *halts* was defined to apply to any procedure, the proof applies it to only one procedure. So we can write a specialized version of *halts* that applies to only that one procedure, and the proof remains intact.

> **function** *halts*: **boolean**;
> { returns *true* if execution of *twist* terminates; returns *false* otherwise }
>
> **procedure** *twist*;
> **begin**
>     **if** *halts* **then** *twist*
> **end**

Assume function *halts* has been programmed according to its specification. Does execution of *twist* terminate? If it terminates, then *halts* returns *true* according to its specification, and so we see from the body of *twist* that execution of *twist* does not terminate. If it does not terminate, then *halts* returns *false* , and so execution of *twist* terminates. This is a contradiction (inconsistency). Therefore function *halts* cannot have been programmed according to its specification.

Since *halts* has no parameters, it is a constant function. Its body must be either *halts*:= *true* or *halts*:= *false* (that's Pascal for **return** *true* and **return** *false* ).

## Formal Proof Version

This version is just like the "No Input Version", but instead of using Pascal, it uses a notation that is more amenable to formal proof. Choose a programming language L. Define

*halts* = (a function that maps texts representing parts of programs in L to their halting status)
*loop* = (a statement or procedure in L whose execution does not terminate)
*stop* = (a statement or procedure in L whose execution terminates)
*twist* = " **if** *halts* (*twist*) **then** *loop* **else** *stop* **fi** "

Now calculate.

| | | |
|---|---|---|
| | *halts* (*twist*) | definition of *twist* |
| = | *halts* (" **if** *halts* (*twist*) **then** *loop* **else** *stop* **fi** ") | semantic transparency |
| = | **if** *halts* (*twist*) **then** *halts* ("*loop*") **else** *halts* ("*stop*") **fi** | definitions of *halts* , *loop* , and *stop* |
| = | **if** *halts* (*twist*) **then** *false* **else** *true* **fi** | binary algebra |
| = | ¬ *halts* (*twist*) | |

The inconsistency we arrive at does not depend on whether *halts* is a programmed function or mathematical function. The semantic transparency step implicitly assumes that if *halts* is a programmed function, then execution of *halts* (*twist*) is terminating. That assumption could be made explicit by adding the axiom

*halts* ("*halts* (" *s* ")")

where *s* is a text representing a part of a program, and text catenation is represented by juxtaposition. Then the calculation is as follows.

| | | |
|---|---|---|
| | *halts* (*twist*) | definition of *twist* |
| = | *halts* (" **if** *halts* (*twist*) **then** *loop* **else** *stop* **fi** ") | apply *halts* to "**if**..." |
| = | *halts* ("*halts* (*twist*)") ∧ **if** *halts* (*twist*) **then** *halts* ("*loop*") **else** *halts* ("*stop*") **fi** | use the *halts* axiom and the definitions of *halts* , *loop* , and *stop* |
| = | *true* ∧ **if** *halts* (*twist*) **then** *false* **else** *true* **fi** | binary algebra |
| = | ¬ *halts* (*twist*) | |

## Direct Version

A proof-by-contradiction makes the assumption that halting is computable, then finds an inconsistency, and concludes that the assumption was wrong. This version does not begin that way.

Choose a programming language L that includes the text (character string) data type and the binary data type. Define

$h$ = (the mathematical function that maps texts representing programs in L to their halting status)
$T$ = (a program in L whose execution terminates)
$N$ = (a program in L whose execution does not terminate)
$P$ = (a program in L that applies to texts with binary result)
$D$ = " **if** $P$ $(D)$ **then** $N$ **else** $T$ **fi** "

Now calculate

| | | |
|---|---|---|
| | *true* | definition of $D$ |
| $=$ | $D$ = " **if** $P$ $(D)$ **then** $N$ **else** $T$ **fi** " | function transparency |
| $\Longrightarrow$ | $h\ (D)$ = $h$ (" **if** $P$ $(D)$ **then** $N$ **else** $T$ **fi** ") | interpretation |
| $=$ | $h\ (D)$ = ($h$ ("$P\ (D)$") $\wedge$ **if** $P$ $(D)$ **then** $h$ ("$N$") **else** $h$ ("$T$") **fi**) | definitions of $N$ and $T$ |
| $=$ | $h\ (D)$ = ($h$ ("$P\ (D)$") $\wedge$ **if** $P$ $(D)$ **then** *false* **else** *true* **fi**) | binary algebra |
| $=$ | $h\ (D)$ = ($h$ ("$P\ (D)$") $\wedge$ $\neg$ $P\ (D)$) | |

For program $P$ to be an implementation of function $h$ , $P$ has to apply to at least the domain of $h$ (it does), and on the domain of $h$ it has to give the same results as $h$ . Suppose $h\ (D)$ is *true* . Then in the last line of the calculation, $h$ ("$P\ (D)$") $\wedge \neg P\ (D)$ means that execution of $P\ (D)$ terminates with result *false* , and so $P$ is not an implementation of $h$ . Suppose $h\ (D)$ is *false* . Then, according to the last line of the calculation, either execution of $P\ (D)$ does not terminate, or $P\ (D)$ is *true* , and so again $P$ is not an implementation of $h$ . In all cases, $P$ is not an implementation of $h$ . Since $P$ was any program in L from texts to binaries, there is no program in L to determine halting for all programs in L.

## LISP Version

The proof by Robert Boyer and J Moore is distinguished by their claim that it is completely formalized and verified using an automated prover, ACL [2]. They use a constructive logic in which all recursions must be well-founded to ensure termination. They define the bounded halting program $B(p, n)$ saying whether execution of $p$ terminates within $n$ steps, but they cannot define the halting program

$$H(p) \;=\; \exists n: nat\cdot B(p, n)$$

because they lack quantification over an infinite domain. In place of Turing Machine operations, they use LISP programs, which are defined by writing a bounded EVAL function to interpret LISP. When execution of $p$ runs past $n$ steps, EVAL $(p, n)$ returns the result BTM . So "execution of $p$ fails to halt" becomes

$$\forall n: nat\cdot \text{EVAL}(p, n) \;=\; \text{BTM}$$

which cannot be expressed due to the unbounded quantification, but which can be proven by induction for any choice of nonterminating $p$ . The gap between a constructive prover and an essentially classical theorem is filled with convincing but informal reasoning, so the proof is not fully formal.

In place of a numeric encoding of programs, they use a textual encoding, as do other proofs in this paper. And they define function CIRC exactly the same as the definition of *twist* in this paper, but in LISP rather than Pascal.

```
CIRC (A)
   ( IF ( HALTS   (QUOTE (CIRC A))
                  ( LIST ( CONS (QUOTE A)
                                 A ) )
                  A )
        (LOOP)
        T )
```

The theorem they prove, paraphrased roughly, says: If a program named HALTS behaves like the halting function (returning T for programs that halt and F for those that don't), then HALTS applied to CIRC returns BTM . This is inconsistent, therefore there is no LISP function to compute halting for all LISP functions. The same conclusion applies to any programming language. But again, the possibility of computing halting for all programs in a set by using a program outside the set was not considered.

## Self-Duplicating Version

The trick used in this version is due to W.V.O.Quine, and it was brought to my attention by Lambert Meertens.

Let L be your favorite programming language. Every programming language provides some way of writing the text that consists of a quotation mark (perhaps by writing it twice, or perhaps by preceding it with a backslash), and I ask you to fill in the definitions of $q$ and $Q$ with whatever language L uses.

$q$ = (the opening quotation mark)

$Q$ = (the closing quotation mark)

Next, in L, define $D$ to have three text parameters, and produces a text result by joining texts together as follows.

$D(x, y, z) = x\ q\ x\ Q\ y\ q\ y\ Q\ y\ q\ z\ Q\ z$

Next, for $T$ choose any program in L whose execution terminates, and for $N$ choose any program in L whose execution does not terminate.

$T$ = (a program in L whose execution terminates)

$N$ = (a program in L whose execution does not terminate)

Finally, assume (for contradiction) that $H$ is a program in L to determine halting for all programs in L.

$H(p)$ = (a program in L that determines halting for all programs in L)

After these definitions, using L syntax, write the following program:

**if** *H* (*D* (“ **if** *H* (*D* ( ”, “,”, “ )) **then** *N* **else** *T* **fi** ”)) **then** *N* **else** *T* **fi**

This program is very interesting because $H$ is being applied to an argument

*D* (“ **if** *H* (*D* ( ”, “,”, “ )) **then** *N* **else** *T* **fi** ”)

that evaluates to the text representing the program itself. (You should try evaluating this expression to see for yourself.) In other words, the program has the form

**if** *H* ( $\mathcal{P}$ ) **then** *N* **else** *T* **fi**

where $\mathcal{P}$ is a text expression whose value represents the program itself. So if execution of this program terminates, then it is equivalent to $N$ whose execution does not terminate. And if execution of this program does not terminate, then it is equivalent to $T$ whose execution does terminate. We have an inconsistency. There is no way to write a program in language L to determine halting for all programs in L.

## Semantics Version

In any programming language, all programs are finite sequences of characters, although not all finite sequences of characters are programs. Suppose we have a programming language L such that the execution of each program begins by reading a finite sequence of characters as input, and after some computing, does one of three actions:

- writes a 0 as output and then terminates.
- writes a 1 as output and then terminates.
- does not write and does not terminate.

Let $C$ be a finite character set, and let $C^*$ be the set of all finite sequences of characters. Define the mathematical function $[\![\ ]\!]$ (not a program) called "semantics" as follows.

$$[\![\ ]\!]\colon C^* \to C^* \to \{0, 1, 2, 3\}$$

$[\![p]\!](i) = 0$ if $p$ is a program in L and execution of $p$ on input $i$ writes $0$ then terminates
$1$ if $p$ is a program in L and execution of $p$ on input $i$ writes $1$ then terminates
$2$ if $p$ is a program in L and execution of $p$ on input $i$ does not write and does not terminate
$3$ if $p$ is not a program in L

$[\![p]\!](i) = 0$ and $[\![p]\!](i) = 1$ both include the possibility that execution does not read the entire input, and $[\![p]\!](i) = 2$ includes the possibility that execution waits forever for more input.

Define the mathematical function $D$ (not a program) called "diagonal" as follows.

$$D\colon C^* \to \{0, 1\}$$

$D(p) = 0$ if $[\![p]\!](p) = 1$ or $[\![p]\!](p) = 2$ or $[\![p]\!](p) = 3$
$1$ if $[\![p]\!](p) = 0$

For all $p$ in $C^*$ , $D(p)$ differs from $[\![p]\!](p)$ , so $D$ is not the semantics of any program in language L. Therefore $D$ cannot be computed by a program in language L.

Define the mathematical function $H$ (not a program) called "halting" as follows.

$$H\colon C^* \to \{0, 1\}$$

$H(p) = 0$ if $[\![p]\!](p) = 2$ or $[\![p]\!](p) = 3$
$1$ if $[\![p]\!](p) = 0$ or $[\![p]\!](p) = 1$

This halting function reports the halting status for each program in L on only a single input.

Assume (for contradiction) that $H$ is computable by a program in L. Then $H$ is the semantics $[\![h]\!]$ of some program $h$ in L. If L is sufficiently expressive (Turing-Machine equivalent), as every general-purpose programming language is, we can write a program in L to compute $D(p)$ as follows.

> Read the input and save it as $p$ . Execute $h$ on input $p$ , but don't output. If the output from executing $h$ on input $p$ would be $0$ , output $0$ . If the output from executing $h$ on input $p$ would be $1$ , execute $p$ on input $p$ , but don't output. If the output from executing $p$ on $p$ would be $0$ , output $1$ . If the output from executing $p$ on $p$ would be $1$ , output $0$ .

We thus compute $D$ by a program in L. But $D$ cannot be computed by a program in L. Therefore $H$ cannot be computed by a program in L.

## Diagonalize-Then-Reduce Version

This is the same as the Semantics Version, but without the semantics function.

In any programming language, all programs are finite sequences of characters, although not all finite sequences of characters are programs. Suppose we have a programming language L such that the execution of each program in L begins by reading a finite sequence of characters as input, and after some computing, does one of three actions:

- writes a 0 as output and then terminates.
- writes a 1 as output and then terminates.
- does not write and does not terminate.

Let $C$ be a finite character set, and let $C^*$ be the set of all finite sequences of characters. Define the mathematical function $D$ (not a program) called "diagonal" as follows.

$D$: $C^* \rightarrow \{0, 1\}$

$D(p)$ = 1 if $p$ is a program in language L and execution of $p$ on input $p$ writes 0 and then terminates
0 otherwise

If $p$ is a program in L and execution of $p$ on input $p$ writes 0 and terminates, then $D(p) = 1$ whether or not the entire input was read. If $p$ is a program in L and execution of $p$ on input $p$ reads its entire input and waits forever for more input, then $D(p) = 0$. If $p$ is not a program in L, then $D(p) = 0$.

Let $d$ be a program in L. Does $d$ implement $D$? Implementation means:

- For all $p$ in $C^*$, if $D(p) = 0$ then execution of $d$ on input $p$ writes 0 and terminates.
- For all $p$ in $C^*$, if $D(p) = 1$ then execution of $d$ on input $p$ writes 1 and terminates.

However, if execution of $d$ on input $d$ writes 0 and terminates, then $D(d) = 1$, not 0. And if execution of $d$ on input $d$ writes 1 and terminates, then $D(d) = 0$, not 1. So $d$ does not implement $D$. Since $d$ was an arbitrary program in L, $D$ cannot be computed by a program in language L.

Define the mathematical function $H$ (not a program) called "halting" as follows.

$H$: $C^* \rightarrow \{0, 1\}$

$H(p)$ = 1 if $p$ is a program and execution of $p$ on input $p$ writes 0 or 1 and then terminates
0 otherwise

This halting function reports the halting status for each program in L on only a single input.

Assume (for contradiction) that $H$ is computable by a program $h$ in language L. If L is sufficiently expressive (Turing-Machine equivalent), as every general-purpose programming language is, we can write a program in L to compute $D(p)$ as follows.

> Read the input and save it as $p$. Execute $h$ on input $p$, but don't output. If the output from executing $h$ on $p$ would be 0, output 0. If the output from executing $h$ on $p$ would be 1, execute program $p$ on input $p$, but don't output. If the output from executing $p$ on $p$ would be 0, output 1. If the output from executing $p$ on $p$ would be 1, output 0.

We thus compute $D$ by a program in L. But $D$ cannot be computed by a program in L. Therefore $H$ cannot be computed by a program in L.

## General Diagonalize-Then-Reduce Version

This is the same as Diagonalize-Then-Reduce but with a more realistic range of programs.

Choose a programming language L. All programs in L are finite sequences of characters, although not all finite sequences of characters are programs. Execution of a program in L may read a sequence of characters as input, and may write a sequence of characters as output. Reading does not have to precede writing; they can be mixed. The input sequence may be empty, or a nonempty finite number of characters, or an infinite number of characters. Likewise the output sequence. Execution may terminate, or it may run forever.

Let $C$ be a finite character set, and let $C^*$ be the set of all finite sequences of characters. Define the mathematical function $D$ (not a program) called “diagonal” as follows.

$D$: $C^* \rightarrow$ {“red”, “blue”}
$D(p)$ = “red” if $p$ is a program in L and execution of $p$ on input $p$ writes “blue” and then terminates
“blue” otherwise

$D(p)$ = “red” when
- $p$ is a program in L, and execution of $p$ on input $p$ writes “blue” and terminates

$D(p)$ = “blue” when
- $p$ is a program in L, and execution of $p$ on input $p$ writes nothing and terminates
- $p$ is a program in L, and execution of $p$ on input $p$ writes anything other than “blue” and terminates
- $p$ is a program in L, and execution of $p$ on input $p$ reads its entire input and waits forever for more input, regardless of what is written
- $p$ is a program in L, and execution of $p$ on input $p$ does not terminate, regardless of what is read or written
- $p$ is a not a program in L

Let $d$ be a program in L. Does $d$ implement $D$ ? Implementation means:
- For all $p$ in $C^*$ , if $D(p)$ = “red” then execution of $d$ on input $p$ writes “red” and terminates.
- For all $p$ in $C^*$ , if $D(p)$ = “blue” then execution of $d$ on input $p$ writes “blue” and terminates.

However, if execution of $d$ on input $d$ writes “red” and terminates, then $D(d)$ = “blue” , not “red” . And if execution of $d$ on input $d$ writes “blue” and terminates, then $D(d)$ = “red” , not “blue” . So $d$ does not implement $D$ . Since $d$ was an arbitrary program in L, $D$ cannot be computed by a program in L.

Define the mathematical function $H$ (not a program) called “halting” as follows.

$H$: $C^* \rightarrow$ {“yes”, “no”}
$H(p)$ = “yes” if $p$ is a program in L and execution of $p$ on input $p$ terminates
“no” otherwise

This halting function reports the halting status for each program in L on only a single input. If $p$ is a program in L and execution of $p$ does not read the entire input $p$ , then $H(p)$ = “yes” . If $p$ is a program in L and execution of $p$ reads the entire input $p$ and waits forever for more input, then $H(p)$ = “no” . If $p$ is not a program in L, then $H(p)$ = “no” .

Assume (for contradiction) that $H$ is computed by a program $h$ in language L. If L is sufficiently expressive (Turing-Machine equivalent), as every general-purpose programming language is, we can write a program in L to compute $D(p)$ as follows.

> Read the input and save it as $p$ . Execute $h$ on input $p$ , but don't output. If the output from executing $h$ on $p$ would be “no” , output “blue” . If the output from executing $h$ on $p$ would be “yes” , execute program $p$ on input $p$ , but don't output. If the output from executing $p$ on $p$ would be “blue” , output “red” . If the output from executing $p$ on $p$ would be anything other than “blue” , output “blue” .

We thus compute $D$ by a program in L. But $D$ cannot be computed by a program in L. Therefore $H$ cannot be computed by a program in L.

## Simpler Diagonalize-Then-Reduce Version

In any programming language, all programs are finite sequences of characters, although not all finite sequences of characters are programs. Execution of a program may read characters as input, may write characters as output, and either terminate or compute forever.

Let $C$ be a finite character set, and let $C^*$ be the set of all finite sequences of characters in $C$. Let L be a programming language. Define the mathematical function $H$ (not a program) called "the halting function" as follows.

$H$: $C^* \rightarrow$ {"yes", "no"}

$H(p)$ = "yes" if $p$ is a program in L and execution of $p$ on input $p$ terminates
"no" otherwise

The halting function $H$ reports the halting status for each program in L on only a single input. If $p$ is a program in L and execution of $p$ on input $p$ terminates, then $H(p)$ = "yes", whether or not the entire input $p$ is read. If $p$ is a program in L and execution of $p$ reads the entire input $p$ and waits forever for more input, then $H(p)$ = "no". If $p$ is not a program in L, then $H(p)$ = "no".

Is there a program $d$ in language L with the following behavior? For all $p$ in $C^*$,

- if $H(p)$ = "no" then execution of $d$ on input $p$ terminates;
- if $H(p)$ = "yes" then execution of $d$ on input $p$ does not terminate.

If execution of $d$ on input $d$ terminates, then $H(d)$ = "yes", not "no". And if execution of $d$ on input $d$ does not terminate, then $H(d)$ = "no", not "yes". So there is no such program in L.

Assume (for contradiction) that $H$ is computed by a program $h$ in language L. If L is sufficiently expressive (Turing-Machine equivalent), as every general-purpose programming language is, we can write program $d$ in language L as follows.

> Read the input and save it as $p$. Execute $h$ on input $p$, but don't output. If the output from executing $h$ on $p$ would be "no", terminate execution. If the output from executing $h$ on $p$ would be "yes", loop forever.

But there is no such program. Therefore $H$ cannot be computed by a program in language L.