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, write a program that reads a text (character string) $p$ representing a program, 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 nine interestingly different versions of the Halting Problem that I have encountered or invented. I present the proofs without mingling any criticisms or comments in them; for criticisms and comments, see http://www.cs.utoronto.ca/~hehner/halting.html. Warning: these proofs may not all be valid; but I leave you to decide their validity for yourself.

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.

```pascal
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 diag (s: string);
begin
  if halts (s, s) then diag (s)
end

Assume function halts has been programmed according to its specification. Does execution of diag ('diag') terminate? If it terminates, then halts ('diag', 'diag') returns true according to its specification, and so we see from the body of diag that execution of diag ('diag') does not terminate. If it does not terminate, then halts ('diag', 'diag') returns false, and so execution of diag ('diag') 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.

```pascal
function halts (p: string): boolean;
{ returns true if string p represents a parameterless Pascal procedure }
{ whose execution halts; returns false otherwise }

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

Assume function `halts` has been programmed according to its specification. Does execution of `diag` terminate? If it terminates, then `halts ('diag')` returns `true` according to its specification, and so we see from the body of `diag` that execution of `diag` does not terminate. If it does not terminate, then `halts ('diag')` returns `false`, and so execution of `diag` 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.

```pascal
function halts: boolean;
{ returns true if execution of diag terminates; returns false otherwise }

procedure diag;
begin
  if halts then diag
end
```

Assume function `halts` has been programmed according to its specification. Does execution of `diag` terminate? If it terminates, then `halts` returns `true` according to its specification, and so we see from the body of `diag` that execution of `diag` does not terminate. If it does not terminate, then `halts` returns `false`, and so execution of `diag` 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. Define

\[
\begin{align*}
\text{halts} & = \text{ (a function that maps texts representing parts of programs to their halting status) } \\
\text{loop} & = \text{ (a statement or procedure whose execution does not terminate) } \\
\text{stop} & = \text{ (a statement or procedure whose execution terminates) } \\
\text{diag} & = \text{ “if halts (diag) then loop else stop ” }
\end{align*}
\]

Now calculate.

\[
\begin{align*}
\text{halts (diag)} & \quad \text{definition of diag} \\
= & \quad \text{halts (“ if halts (diag) then loop else stop ”)} \\
= & \quad \text{if halts (diag) then halts (“loop”) else halts (“stop”)} \quad \text{definitions of halts, loop, and stop} \\
= & \quad \text{if halts (diag) then false else true} \quad \text{boolean algebra} \\
= & \quad \neg \text{halts (diag)}
\end{align*}
\]

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

\[
\text{halts (“halts (“ s “””)}
\]

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

\[
\begin{align*}
\text{halts (diag)} & \quad \text{definition of diag} \\
= & \quad \text{halts (“ if halts (diag) then loop else stop ”)} \quad \text{apply halts to “if...”} \\
= & \quad \text{halts (“halts (diag)” \& if halts (diag) then halts (“loop”) else halts (“stop”) } \quad \text{use the halts axiom and the definitions of halts, loop, and stop} \\
= & \quad \text{true \& if halts (diag) then false else true} \quad \text{boolean algebra} \\
= & \quad \neg \text{halts (diag)}
\end{align*}
\]
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 that includes the text (character string) data type and the boolean data type. Define

\[ h = \text{(the mathematical function that maps texts representing programs to their halting status)} \]

\[ T = \text{(a program whose execution terminates)} \]

\[ N = \text{(a program whose execution does not terminate)} \]

\[ P = \text{(a program that applies to texts with boolean result)} \]

\[ D = \text{“if } P (D) \text{ then } N \text{ else } T \text{”} \]

Now calculate

\[ true \]

\[ \Rightarrow D = \text{“if } P (D) \text{ then } N \text{ else } T \text{”} \] \hspace{1cm} \text{definition of } D

\[ \Rightarrow h (D) = h \text{ (“if } P (D) \text{ then } N \text{ else } T \text{”)} \] \hspace{1cm} \text{function transparency}

\[ \Rightarrow h (D) = (h \text{ (“} P (D) \text{”)} \land \text{if } P (D) \text{ then } h \text{ (“} N \text{”)} \text{ else } h \text{ (“} T \text{”)}) \] \hspace{1cm} \text{interpretation}

\[ \Rightarrow h (D) = (h \text{ (“} P (D) \text{”)} \land \text{if } P (D) \text{ then } \bot \text{ else } T) \] \hspace{1cm} \text{definitions of } N \text{ and } T

\[ \Rightarrow h (D) = (h \text{ (“} P (D) \text{”)} \land \neg P (D)) \] \hspace{1cm} \text{boolean algebra}

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, \( h \text{ (“} P (D) \text{”)} \land \neg P (D) \) 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 from texts to booleans, the halting function is incomputable.
Self-Duplicating Version

The trick used in this version is due to W.V.O. Quine.

Define

\[ q = \text{(the opening quotation mark)} \]
\[ Q = \text{(the closing quotation mark)} \]
\[ D(x,y,z) = x q x Q y q y Q y q z Q z \]
\[ T = \text{(a program whose execution terminates)} \]
\[ N = \text{(a program whose execution does not terminate)} \]
\[ H(p) = \text{(a program that implements the halting function)} \]

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 your favorite programming language uses. Program \( D \) has three text parameters and produces a text result by catenation. For \( T \) choose any program whose execution terminates, and for \( N \) choose any program whose execution does not terminate. And finally, the ability to define \( H \) as a program is the computability assumption.

After these definitions, the program

\[
\text{if } H(D(\text{"if } H(D(\text{"", ",", "}) \text{ then } N \text{ else } T\text{"})) \text{ then } N \text{ else } T}
\]

is very interesting because \( H \) is being applied to an argument

\[
D(\text{"if } H(D(\text{"", ",", "}) \text{ then } N \text{ else } T\text{"}))
\]

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

\[
\text{if } H(\mathcal{P}) \text{ then } N \text{ else } T
\]

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 cannot be any way to define program \( H \).
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 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.

$$\Semantics(C^*) \rightarrow \{0, 1, 2, 3\}$$

\[
\Semantics(p)(i) = 
\begin{cases} 
0 & \text{if } p \text{ is a program and execution of } p \text{ on input } i \text{ writes 0 then terminates} \\
1 & \text{if } p \text{ is a program and execution of } p \text{ on input } i \text{ writes 1 then terminates} \\
2 & \text{if } p \text{ is a program and execution of } p \text{ on input } i \text{ does not write and does not terminate} \\
3 & \text{if } p \text{ is not a program}
\end{cases}
\]

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

Define the mathematical function (not a program) called “diagonal” as follows.

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

$$D(p) = 
\begin{cases} 
0 & \text{if } \Semantics(p)(p) = 1 \text{ or } \Semantics(p)(p) = 2 \text{ or } \Semantics(p)(p) = 3 \\
1 & \text{if } \Semantics(p)(p) = 0
\end{cases}
\]

For all $p$ in $C^*$, $D(p)$ differs from $\Semantics(p)(p)$, so $D$ is not the semantics of any program. Therefore $D$ is incomputable.

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

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

$$H(p) = 
\begin{cases} 
0 & \text{if } \Semantics(p)(p) = 2 \text{ or } \Semantics(p)(p) = 3 \\
1 & \text{if } \Semantics(p)(p) = 0 \text{ or } \Semantics(p)(p) = 1
\end{cases}
\]

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

Assume (for contradiction) that $H$ is computable. Then $H$ is the semantics $\Semantics(h)$ of some program $h$. If the programming language is sufficiently expressive (Turing-Machine equivalent), as every general-purpose programming language is, we can 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 $D$. But $D$ is incomputable. Therefore $H$ is incomputable.
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 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 “diagonal” as follows.

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

\[
D(p) = 1 \quad \text{if } p \text{ is a program and execution of } p \text{ on input } p \text{ writes 0 and then terminates}
\]

\[
D(p) = 0 \quad \text{otherwise}
\]

\( D(p) = 1 \) includes the possibility that \( p \) is a program and execution of \( p \) does not read the entire input \( p \). \( D(p) = 0 \) includes the possibility that \( p \) is a program and execution of \( p \) reads the entire input \( p \) and waits forever for more input.

Can \( D \) be implemented by a program \( d \)? 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, \( D \) is incomputable.

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

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

\[
H(p) = 1 \quad \text{if } p \text{ is a program and execution of } p \text{ on input } p \text{ writes 0 or 1 and then terminates}
\]

\[
H(p) = 0 \quad \text{otherwise}
\]

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

Assume (for contradiction) that \( H \) is computable. Then \( H \) is implemented by some program \( h \). If the programming language is sufficiently expressive (Turing-Machine equivalent), as every general-purpose programming language is, we can 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 \( D \). But \( D \) is incomputable. Therefore \( H \) is incomputable.
General Program Version same as previous version but with a more realistic range of programs

Choose a programming language. All programs in that language are finite sequences of characters, although not all finite sequences of characters are programs. Execution of a program 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 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 (not a program) called “diagonal” as follows.

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

\( D(p) = \text{“red”} \) when

- \( p \) is a program, and execution of \( p \) writes “blue” and terminates, with or without reading the entire input \( p \)

\( D(p) = \text{“blue”} \) when

- \( p \) is a program, and execution of \( p \) writes nothing or writes anything other than “blue” and terminates, with or without reading the entire input \( p \)
- \( p \) is a program, and execution of \( p \) reads the entire input \( p \) and waits forever for more input, regardless of what is written
- \( p \) is a program, and execution of \( p \) on input \( p \) does not terminate, regardless of what is read/written
- \( p \) is a not a program

Can \( D \) be implemented by a program \( d \)? Implementation means:

- For all \( p \) in \( C^* \), if \( D(p) = \text{“red”} \) then execution of \( d \) on input \( p \) writes “red” and terminates.
- For all \( p \) in \( C^* \), if \( D(p) = \text{“blue”} \) then execution of \( d \) on input \( p \) writes “blue” and terminates.

However, if execution of \( d \) on input \( d \) writes “red” and terminates, then \( D(d) = \text{“blue”} \), not “red”.

And if execution of \( d \) on input \( d \) writes “blue” and terminates, then \( D(d) = \text{“red”} \), not “blue”. So \( d \) does not implement \( D \). Since \( d \) was an arbitrary program, \( D \) is incomputable.

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

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

This halting function reports the halting status for each program on only a single input. \( H(p) = \text{“yes”} \) includes the possibility that \( p \) is a program and execution of \( p \) does not read the entire input \( p \). \( H(p) = \text{“no”} \) includes the possibility that \( p \) is a program and execution of \( p \) reads the entire input \( p \) and waits forever for more input.

Assume (for contradiction) that \( H \) is computable. Then \( H \) is implemented by some program \( h \). If the programming language is sufficiently expressive (Turing-Machine equivalent), as every general-purpose programming language is, we can 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 \( D \). But \( D \) is incomputable. Therefore \( H \) is incomputable.