Diagonalize Then Reduce

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

Twisted Self-Reference

There is a standard argument, appearing in many textbooks, in a variety of different notations, that is supposed to prove that the Halting Problem is incomputable. It considers a procedure, let's call it diag, whose only action is

if halts (“diag”) then infiniteloop else terminate fi

where halts is a function that determines whether execution of a program terminates, infiniteloop is an infinite loop, and terminate terminates. If halts says that execution of diag is terminating, then it's nonterminating; and if halts says that execution of diag is nonterminating, then it's terminating. Whatever halts reports for diag, it is wrong; there cannot be a halting program. I will call this argument the “twisted self-reference” proof. In the paper Epimenides, Gödel, Turing: an Eternal Gölden Tangle, I argue that the twisted self-reference proof does not prove that halting is incomputable; rather it proves that the specification “Write a program in language L that determines whether execution of any program in language L terminates.” is inconsistent, or self-contradictory.

Diagonalize Then Reduce

There is another argument, which I will call “diagonalize-then-reduce”, that is supposed to prove that the Halting Problem is incomputable without using any self-reference. Here is a version of it.

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 D (not a program) called “diagonal” as follows.

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

- D(p) = “red” when
  - p is a program, and execution of p writes “blue” and terminates, with or without reading the entire input p
- D(p) = “blue” when
  - p is a program, and execution of p writes nothing and terminates, with or without reading the entire input p
- p is a program, and execution of p 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 or written
- p is a not a program

Let prog be a program. Does prog implement D? Implementation means:

- For all p in C*, if D(p) = “red” then execution of prog on input p writes “red” and terminates.
- For all p in C*, if D(p) = “blue” then execution of prog on input p writes “blue” and terminates.

However, if execution of prog on input prog writes “red” and terminates, then D(prog) = “blue”, not “red”. And if execution of prog on input prog writes “blue” and terminates, then D(prog) = “red”, not “blue”. So prog does not implement D. Since prog was an arbitrary program, D is incomputable.
Define the mathematical function $H$ (not a program) called “halting” as follows.

$$H: \mathcal{C}^* \to \{\text{"yes"}, \text{"no"}\}$$

$H(p) = \text{"yes"}$ if $p$ is a program and execution of $p$ on input $p$ terminates

$\text{"no"}$ otherwise

This halting function reports the halting status for each program $p$ on only a single input $p$. $H(p) = \text{"yes"}$ 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 $\text{halts}$. 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 $\text{halts}$ on input $p$, but don't output. If the output from executing $\text{halts}$ on $p$ would be “$\text{no}$”, output “blue”. If the output from executing $\text{halts}$ on $p$ would be “$\text{yes}$”, 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 “$\text{blue}$”, output “blue”.

We thus compute $D$. But $D$ is incomputable. Therefore $H$ is incomputable.

**Discussion**

We began by choosing a programming language; call it $L$. Mathematical function $D$ is defined by diagonalizing over the programs of language $L$. The definition of mathematical function $D$ is not self-referential, and it is consistent. We then ask whether $D$ is implemented by a program in $L$; let's call it $\text{prog}$. Program $\text{prog}$ must implement $D$, which is defined over programs in $L$, including $\text{prog}$, with a twist so that $D$ differs from $\text{prog}$. Program $\text{prog}$ is defined with a twisted self-reference; its specification is inconsistent; there is no such program. But we cannot conclude that $D$ is incomputable, because we have not asked whether $D$ can be implemented in a programming language other than the one over which $D$ is defined.

Consider the question “Can an $L$ program correctly answer “no” to this question?”. It is easy to write an $L$ program whose execution prints “yes”, but that answer says that “no” is the correct answer. There is another $L$ program that prints “no”, but that answer says that no $L$ program can do what it is doing (printing “no” in answer to the question). There is no program in language $L$ that answers the question correctly. But there is a program in language $M$ that answers that same question correctly: it prints “no”, saying that no $L$ program can correctly answer the question. Due to the twisted self-reference, the task is impossible for an $L$ program. But it is not incomputable; it can be answered by an $M$ program. Symmetrically, the question “Can an $M$ program correctly answer “no” to this question?” cannot be correctly answered by an $M$ program, but it can be correctly answered by an $L$ program.

Likewise function $D$ cannot be computed by an $L$ program due to the twisted self-reference. But that does not prevent $D$ from being computed by an $M$ program. The conclusion that $D$ is incomputable is unwarranted.

We have done the diagonalization; now comes the reduction. Mathematical function $H$ is defined as the halting function for programs in language $L$. Its definition is not self-referential, and it is consistent. The final paragraph says: if we could compute halting, then we could compute $D$. But we can't compute $D$. So we can't compute halting; halting is incomputable. To be more precise, the final paragraph means: if we could write an $L$ program to compute halting for all $L$ programs, then we could write an $L$ program to compute $D$. But we can't write an $L$ program to compute $D$. So we can't write an $L$ program to compute halting for all $L$ programs. We cannot conclude that halting is incomputable. We can conclude only that the specification “Write an $L$ program to compute halting for all $L$ programs.” is inconsistent. That conclusion does not prevent halting for language $L$ from being computed by a program in a language other than $L$.

other papers on halting
Appendix in reply to a challenge, added 2016-11-13

My "Discussion" section contains the statement "But we cannot conclude that $D$ is incomputable, because we have not asked whether $D$ can be implemented in a programming language other than the one over which $D$ is defined.". A friend suggested the following argument, concluding that $D$ cannot be implemented in any programming language.

Define mathematical function $D$ as follows: for all programs $p$ in language $L$, $D(p) \neq p(p)$ . Function $D$ differs from all programs in $L$ on at least one input. Therefore $D$ is not computed by any program in $L$. Let $C$ be a program in language $M$ that computes $D$: for all programs $p$ in $L$, $C(p) = D(p)$ . Then there is an equivalent program $B$ in $L$: for all programs $p$ in $L$, $B(p) = C(p)$ . Now calculate:

\[
\begin{align*}
C(B) & \quad \text{use definition of } C \\
= & \quad D(B) \quad \text{use definition of } D \\
\neq & \quad B(B) \quad \text{use definition of } B \\
= & \quad C(B)
\end{align*}
\]

Hence $C(B) \neq C(B)$ , which is a self-contradiction. Conclusion: there is no program in $M$ that computes $D$ .

There are some minor problems with this argument. To pass a program as data to a function or to another program, you need to encode it (as a number or character string). That problem is trivial to fix, and I'll ignore it. Another problem is that if execution of program $p$ does not terminate on input $p$ , then $p(p)$ is undefined. That problem may seem to be fixed by saying that $D(p)$ can be any result for that case, although there are problems with that fix; but I'll ignore that problem too. Another problem is that $D(p) \neq p(p)$ does not say what the value of $D(p)$ is; only what it isn't. That problem is fixed by choosing a specific result for $D(p)$ except when $p(p)$ is also that result, and for that case choosing one other result. Equivalently, we restrict programs to those with a binary result, and define $D$ to have a binary result. So I'll ignore that problem too.

When we arrive at the contradiction $C(B) \neq C(B)$ , we are compelled to withdraw some assumption we made leading to the contradiction. The assumption chosen is: " $C$ is a program in $M$ that computes $D$ ". But there is another candidate. The statement "there is an equivalent program $B$ in $L$" contains a hidden assumption that I think is wrong. I'll explain in a moment.

Here's the same argument as above, but I simplify by getting rid of the function's parameter, making it a constant.

Define mathematical constant $D$ as the correct answer to the question "Can an $L$ program correctly answer "no" to this question?". If an $L$ program can correctly answer "no", then $D=$"yes" . If an $L$ program cannot correctly answer "no", leaving "yes" as the correct answer, then $D=$"no" . Constant $D$ is defined such that if an $L$ program says $B$ , then $B$ is not the correct answer: $D \neq B$ . Assume there is a program in $M$ that gives the correct answer $C$ ; then $C=D$ . Then there is an equivalent program $B$ in $L$ that gives the same answer: $B=C$ . Now calculate:

\[
\begin{align*}
C & \quad \text{use definition of } C \\
= & \quad D \quad \text{use definition of } D \\
\neq & \quad B \quad \text{use definition of } B \\
= & \quad C
\end{align*}
\]

Hence $C \neq C$ , which is a self-contradiction. Conclusion: there is no program in $M$ that correctly answers $D$ .

The conclusion is wrong: there is a program in $M$ that answers correctly: it prints "no" . Where does the argument go wrong? The argument says "there is an equivalent program $B$ in $L$ that gives the same answer: $B=C$ ". Indeed there is a program in $L$ that prints the same answer "no", but when a program in $L$ prints "no", it's incorrect.

Likewise in the previous argument where $D$ is a function with a parameter. If there is a program $C$ in $M$ that computes $D$ , then yes, there is an "equivalent" program in $L$ which, for each input, gives the same output. But that $L$ program doesn't compute $D$ .

I put the word "equivalent" in quotation marks because I think it is ambiguous. It might mean "for each input gives the same output"; let's call that extensional equivalence. Or it might mean "satisfies the same specification"; let's...
call that “intensional equivalence”. Most of the time, intensional and extensional equivalence are the same thing. They may differ when there’s a self-reference. The above proofs pivot on the word “equivalence”.

In the simplified version where $D$ is a constant, the calculation $C = D \neq B = C$ uses an intensional step: $D \neq B$. $D$ is defined to differ from $B$. A reasonable person might say: first show me $B$, then we can define $D$ to be the other answer. That would be an extensional definition. But we cannot show $B$ because both answers are incorrect when said by an L program. So $D$ is not defined extensionally. It is defined intensionally as differing from $B$, whatever $B$ is.

Likewise in the version where $D$ is a function with a parameter. The calculation $C(B) = D(B) \neq B(B) = C(B)$ uses an intensional step: $D(B) \neq B(B)$. $D(p)$ is defined to differ from $p(p)$, and so $D(B) \neq B(B)$. A reasonable person might say: first show me $B(B)$, then we can define $D(B)$ to be the other answer. That would be an extensional definition. But we cannot show $B(B)$. So $D(B)$ is not defined extensionally. It is defined intensionally as differing from $B(B)$, whatever $B(B)$ is.

When we come to the self-contradiction, the assumption that I would flag as being wrong is the hidden assumption that intensional definitions are equivalent to extensional definitions. Normally they are equivalent, but in the presence of a self-reference, they may not be equivalent, and in this case, they are not equivalent.