Advances in Computer Science and Engineering
Volume 10, Number 1, 2013, Pages 31-60
Published Online: May 2013
Available online at http://pphmj.com/journals/acse.htm
Published by Pushpa Publishing House, Allahabad, INDIA

PROBLEMS WITH THE HALTING PROBLEM

Eric C. R. Hehner

Department of Computer Science
University of Toronto
Canada
e-mail: hehner@cs.utoronto.ca

Abstract

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 incompleteness, then we cannot require a halting program to apply to programs that invoke the halting program, and we cannot conclude that it is incomputable. If we choose inconsistency, then it makes no sense to propose a halting program. Either way, the incomputability argument is lost.

Introduction

In 1936, Turing wrote a paper [7] introducing a computing machine that we now call a "*Turing Machine*". He showed how it can be programmed, and how different programs give the machine different behaviors, or in his terminology, create different machines. He showed how one such machine, a "Universal Turing Machine", can be given a description of any Turing Machine (a program), and then simulate the operation of that Turing Machine (execute the program). That work remains an important part of the foundation of computer science. Page 247 of that paper is a proof that a

© 2013 Pushpa Publishing House
Keywords and phrases: halting problem, computability.
Communicated by K. K. Azad
Received January 22, 2013

certain problem that we now call the "*Halting Problem*" cannot be solved by computation. I want to re-examine that proof.

The halting function is defined to say whether a program's execution terminates. In this paper, I will use the word "function" to mean a mathematical total function, mapping every element of its domain to an element of its range.

To apply a function to the domain of programs, Turing encoded programs as numbers. His programs were sequences of Turing Machine operations; each operation was encoded as a sequence of decimal digits, and each program was encoded by joining together the digit sequences representing the operations. In this paper, I use programming notations that are typical of modern programming languages, rather than Turing Machine operations. Today, when programs are presented as input data to a compiler or interpreter, they are represented as texts (character strings), and that is the encoding I will use. These changes modernize and simplify the presentation without changing anything essential in the proof of incomputability.

In some formulations of the Halting Problem, the halting function is applied to two operands: a representation of a program, and the initial state of the program variables (the input). The function tells whether the program's execution terminates when started in the initial state. The presentation of the Halting Problem can be simplified by eliminating the initial state (input) operand. One way to do that is to start execution of every program in the same initial state; if you wanted some other initial state, just begin the program with some initializing assignments to create the state you wanted. In other words, creation of the initial state you wanted is the initial part of the program. Turing did not distinguish the initial state of program variables from program, and I will follow Turing on that point.

The proof is a "diagonal argument", famously used by Cantor [2] in 1890, and by Gödel [3] in 1931. In Turing's proof, the diagonalization is implicit in the self-referential definition of a program code to which he applies the halting function.

Notations and Terminology

For the purpose of this paper, we need a programming language that includes the following three kinds of programming statement:

$x := E$

if B **then** P **else** Q

while B **do** P

These are standard assignment, conditional, and loop statements found in most modern programming languages (except perhaps for minor syntactic differences). In the assignment statement, x is a program variable, and E is an expression of the same type as x. In the next two, B is a boolean expression, and P and Q are programming statements. The two boolean values are $\top$ (true) and $\bot$ (false). By "boolean expression" I mean an expression whose overall type is boolean; subexpressions within it are not restricted to type boolean. Any programming language will include only a restricted choice of types and operators; for our purpose, all we need is call, and some way to return a result. I refer to any program fragment (statement or expression) as a "program".

Halting Problem

Define function $h : \mathbb{P} \to \mathbb{B}$, where $\mathbb{P}$ is the data type of texts that represent programs, and $\mathbb{B}$ is the boolean data type, so that when h is applied to a text representing a program whose execution terminates, the result is $\top$, and when applied to a text representing a program whose execution does not terminate, the result is $\bot$. Here are two examples:

$h(\text{“}x := 0\text{”}) = \top$

$h(\text{“}\textbf{while } \top \textbf{ do } x := 1\text{”}) = \bot$

Assume that h is computable, and that a program H for it has been written.

Then

H (“$x := 0$”) = $\top$

H (“**while** $\top$ **do** $x := 1$”) = $\bot$

Define program text D (for diagonal) as follows:

D = “**if** $H(D)$ **then while** $\top$ **do** $x := 1$ **else** $x := 0$”

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

Argue: If $H(D)$ is $\top$, then D represents a program that is equivalent to **while** $\top$ **do** $x := 1$, whose execution does not terminate, and so $H(D)$ is $\bot$. And if $H(D)$ is $\bot$, then D represents a program that is equivalent to $x := 0$, whose execution does terminate, and so $H(D)$ is $\top$. We have an inconsistency.

Conclude: Program H does not exist; function h is not computable.

There is a concern about the preceding proof: D is defined as a text, but is it a text that represents a program? We have supposed that **if then else** and **while do** and $\top$ and assignment and function call are all in the programming language, but to know that

if $H(D)$ **then while** $\top$ **do** $x := 1$ **else** $x := 0$

is a program, we need to know that H is being applied to an element of its domain. In other words, we need to know whether D represents a program, which was the question. The proof quietly makes the assumption that D does represent a program. Then, when the argument exposes an inconsistency, the cause could be this assumption, rather than the assumption of computability. I will repair this problem later in the section titled “Direct Proof”.

The argument that there is an inconsistency was made informally, both in

Turing's paper (see the Appendix) and in the box above. Informality can hide problems, so I now write the argument as a formal proof.

$H(D)$ use definition of D

$= H$ ("**if** $H(D)$ **then while** $\top$ **do** $x \coloneqq 1$ **else** $x \coloneqq 0$") semantic transparency

$=$ **if** $H(D)$ **then** H ("**while** $\top$ **do** $x \coloneqq 1$") **else** H ("$x \coloneqq 0$") given examples

$=$ **if** $H(D)$ **then** $\bot$ **else** $\top$ boolean algebra

$= \neg H(D)$

The property "semantic transparency" means that if two programs are semantically equivalent, then function *h* gives the same result when applied to their representations. And therefore program *H* must also. The specific instance used here formalizes the argument that if $H(D)$ is $\top$, then the program

if $H(D)$ **then while** $\top$ **do** $x \coloneqq 1$ **else** $x \coloneqq 0$

is equivalent to its **then**-part, and so *H* applied to the representation of that program must give the same result as when applied to the representation of its **then**-part. Likewise when $H(D)$ is $\bot$, *H* applied to the representation of that program must give the same result as when applied to the representation of its **else**-part.

Semantic transparency misses part of the computation. In the usual (eager) interpretation of **if** ... **then** ... **else** ... , its execution terminates exactly when the following are true:

- execution of the **if**-part terminates
- if the **if**-part terminates with result $\top$, execution of the **then**-part terminates
- if the **if**-part terminates with result $\bot$, execution of the **else**-part terminates

Semantic transparency missed the first bullet-point. Here is a proof that includes the missing piece. In it, I have used the words "function transparency" to say that when a function is applied to equal operands, the results are equal; it is a general property in mathematics, not specific to programs. The three bullet-points are called "*interpretation*".

$\top$ definition of D

$= D =$ "**if** $H(D)$ **then while** $\top$ **do** $x := 1$ **else** $x := 0$" function transparency

$\Rightarrow H(D) = H($"**if** $H(D)$ **then while** $\top$ **do** $x := 1$ **else** $x := 0$"$)$ interpretation

$= H(D) = (H($"$H(D)$"$) \wedge$ **if** $H(D)$ **then** $H($"**while** $\top$ **do** x $:= 1$"$)$

else $H($"$x := 0$"$))$ given examples

$= H(D) = (H($"$H(D)$"$) \wedge$ **if** $H(D)$ **then** $\bot$ **else** $\top)$ boolean algebra

$= H(D) = (H($"$H(D)$"$) \wedge \neg H(D))$ boolean algebra

$= \neg H($"$H(D)$"$) \wedge \neg H(D)$

The last line is the uncomfortable conclusion that program H tells us both that execution of $H(D)$ does not terminate and that execution of the program represented by D does not terminate. There is no formal inconsistency, and we cannot formally conclude that h is incomputable. I will repair this problem later in the section titled "Direct Proof".

Calumation Problem

An argument identical to the Halting Problem proves that any property of program executions is incomputable.

Define function $h : \mathbb{P} \to \mathbb{B}$, where $\mathbb{P}$ is the data type of texts that represent programs, and $\mathbb{B}$ is the boolean data type, so that when h is applied to a text representing a program whose execution calumates, the result is $\top$, and when applied to a text representing a program whose

execution does not calumate, the result is ⊥. I do not define calumation; I just give two examples:

$h(\text{“}x := 0\text{”}) = \top$

$h(\text{“}x := 1\text{”}) = \bot$

Assume that *h* is computable, and that a program *H* for it has been written. Then

$H(\text{“}x := 0\text{”}) = \top$

$H(\text{“}x := 1\text{”}) = \bot$

Define program text *D* as follows:

$D =$ “**if** $H(D)$ **then** $x := 1$ **else** $x := 0$”

Argue: If $H(D)$ is ⊤, then *D* represents a program that is equivalent to $x := 1$, whose execution does not calumate, and so $H(D)$ is ⊥. And if $H(D)$ is ⊥, then *D* represents a program that is equivalent to $x := 0$, whose execution does calumate, and so $H(D)$ is ⊤. We have an inconsistency.

Conclude: Program *H* does not exist; function *h* is not computable.

All we need to know about *h* is that it has a positive and a negative example, which I chose arbitrarily. We do not even need to know what the positive and negative examples are; if we just know that there is at least one of each, we can make the same proof. And the range of the function need not be boolean; it can be any set with at least two values. According to this generalization of Turing’s proof, every function applying to representations of programs, and having semantic transparency, is incomputable, with the possible exception of constant functions (functions that do not actually depend on their operands) [6].

Russell's Barber

Before I discuss the Halting Problem further, I present a well-known "paradox" attributed to Bertrand Russell, in the hope that it will illuminate the Halting Problem.

In a very small town, there are exactly 3 men, named *A*, *B* and *C*. The man named *B* is the barber. The barber shaves the men in the town who do not shave themselves. (Perhaps today it would make more sense to talk about cutting hair, but in Russell's day barbers also shaved men who did not shave themselves.) Formalizing,

$$\forall m \in \{A, B, C\} \cdot (B \textit{ shaves } m) = \neg(m \textit{ shaves } m)$$

An immediate consequence is obtained by specialization:

$$(B \textit{ shaves } B) = \neg(B \textit{ shaves } B)$$

and we have an inconsistency.

This inconsistency surprises some people, and that is why it was called a "paradox". Here, I think, is the source of the surprise. The sentence "The barber shaves the men in the town who do not shave themselves." seems perfectly reasonable; that is what barbers do (or did in Russell's day). Anyone saying the sentence in earnest means "The barber shaves the other men in the town who do not shave themselves." Formally,

$$\forall m \in \{A, C\} \cdot (B \textit{ shaves } m) = \neg(m \textit{ shaves } m)$$

The question whether *B shaves B* is left open, and can be answered either way without inconsistency. Similarly, someone might say "There's no-one here.", and reasonable people understand that to mean "no-one but me". Someone might say "She's smarter than anyone.", or "I never eat fish, but on that one occasion I did.", and reasonable people understand. But Bertrand Russell was a philosopher and logician; he took the sentence literally, and exposed the inconsistency, to the surprise of reasonable people.

Is *shaves* computable? I am not seriously asking, but I give the question the same sort of proof that we have already seen twice.

Define function *shaves*: $\mathbb{M} \times \mathbb{M} \to \mathbb{B}$, where $\mathbb{M}$ is the men of the town, and $\mathbb{B}$ is the boolean data type, such that the barber shaves the men in the town who do not shave themselves.

Assume that *shaves* is computable, and that a program *SHAVES* for it has been written.

Argue:

$$\forall m \in \{A, B, C\} \cdot (B \ SHAVES \ m) = \neg(m \ SHAVES \ m)$$

and by specialization

$$(B \ SHAVES \ B) = \neg(B \ SHAVES \ B)$$

We have an inconsistency.

Conclude: Program *SHAVES* does not exist; function *shaves* is not computable.

Initially, *shaves* is defined informally; a reasonable person understands it, and does not doubt its consistency. After the computability assumption, we use Russell's literal interpretation of the definition, including the question whether *B SHAVES B*. I do not intend this proof to be taken seriously; I use it to illustrate what happens when inconsistency is not suspected before a computability assumption, and then is discovered after. If we intend the characteristic property of *shaves* to apply to all of *A*, *B*, and *C*, then *shaves* is inconsistently defined, and its computability is not a sensible question. If we intend the characteristic property of *shaves* to apply to only *A* and *C*, then we should not require the characteristic property of its program counterpart *SHAVES* to apply to *B*. Either way, the conclusion is void.

Could there be a hidden inconsistency in the definition of the halting function, without assuming computability?

Oracle Halting Problem

Let us look again at the Halting Problem, but this time without the assumption that *h* is computable. To do that, we augment the programming

language with an oracle $\mathcal{H}$ that magically tells whether a program in this augmented programming language has terminating behavior. The oracle's behavior is always terminating. The concept of an oracle within a program was invented by Turing [8] in his 1938 Ph.D. thesis to study relative computability. It exactly serves our purpose: to provide the result of the halting function without concern for how, or even whether, it can be computed.

Define function $h : \mathbb{P} \to \mathbb{B}$, where $\mathbb{P}$ is the data type of texts that represent programs in the augmented programming language, and $\mathbb{B}$ is the boolean data type, so that when h is applied to a text representing a program whose execution terminates, the result is $\top$, and when applied to a text representing a program whose execution does not terminate, the result is $\bot$. Here are two examples:

$h(\text{“}x := 0\text{”}) = \top$

$h(\text{“}\textbf{while } \top \textbf{ do } x := 1\text{”}) = \bot$

Let $\mathcal{H}$ be an oracle for h. Then

$\mathcal{H}(\text{“}x := 0\text{”}) = \top$

$\mathcal{H}(\text{“}\textbf{while } \top \textbf{ do } x := 1\text{”}) = \bot$

Define program text $\mathcal{D}$ as follows:

$\mathcal{D} =$ “**if** $\mathcal{H}(\mathcal{D})$ **then while** $\top$ **do** $x := 1$ **else** $x := 0$”

Argue: If $\mathcal{H}(\mathcal{D})$ is $\top$, then $\mathcal{D}$ represents a program that is equivalent to **while** $\top$ **do** $x := 1$, whose execution does not terminate, and so $\mathcal{H}(\mathcal{D})$ is $\bot$. And if $\mathcal{H}(\mathcal{D})$ is $\bot$, then $\mathcal{D}$ represents a program that is equivalent to $x := 0$, whose execution terminates, and so $\mathcal{H}(\mathcal{D})$ is $\top$. We have an inconsistency.

Conclude: ?

The Oracle Halting Problem shows us that there is an inconsistency even without the assumption of computability. As before, we could substitute any property (e.g., calumation) in place of termination. But this time, without the assumption of computability, we cannot conclude its opposite. What should we conclude? that there cannot be an oracle for the halting function?

The use of a magic oracle may cause the reader concern about the legitimacy of the result: if we assume magic, we might come to unreal conclusions. Indeed, if we had come to a positive conclusion of the form "Therefore we can compute *X*", it would be invalid. But we came to a negative conclusion: "even if we are given the result of the halting function, with no need to compute it, there is still an inconsistency", and that conclusion is valid. To remove all concern, I now replace the oracle with its modern counterpart: specification.

Specification Halting Problem

In Turing's time, programs were commands to a computer. Today, they are also mathematical expressions in their own right [4, 5]. Programs are specifications of computer behavior written in a restricted formal notation (a programming language) that a computer can execute. Not all specifications of computer behavior are programs. We might start a programming task with an informal, natural language specification of desired computer behavior, then formalize the specification using any mathematical notations that we find useful, then refine the specification until we have a program. During refinement, the specification is a mixture of programming and other notations. Instead of imagining a magic oracle to use in a program, we can write a specification using the mathematical halting function. Here is the Specification Halting Problem.

Define function $h : \mathbb{S} \to \mathbb{B}$, where $\mathbb{S}$ is the data type of texts that represent specifications of computer behavior, and $\mathbb{B}$ is the boolean data type, so that when h is applied to a text representing a specification of terminating behavior, the result is $\top$, and when applied to a text

representing a specification of nonterminating behavior, the result is $\perp$. Here are two examples:

$h(\text{“}x := 0\text{”}) = \top$

$h(\text{“}\textbf{while } \top \textbf{ do } x := 1\text{”}) = \perp$

Define specification text d as follows:

d = “**if** $h(d)$ **then while** $\top$ **do** $x := 1$ **else** $x := 0$”

Argue: If $h(d)$ is $\top$, then d represents a specification that is equivalent to **while** $\top$ **do** $x := 1$, which specifies nonterminating behavior, and so $h(d)$ is $\perp$. And if $h(d)$ is $\perp$, then d represents a specification that is equivalent to $x := 0$, which specifies terminating behavior, and so $h(d)$ is $\top$. We have an inconsistency.

Conclude: ?

Moving from the original Halting Problem to the Specification Halting Problem, we augmented the domain of h from program texts to specification texts. But to arrive at inconsistency, all we really need is a domain consisting of three texts: the two program texts “$x := 0$” and “**while** $\top$ **do** $x := 1$”, and specification text d. Applying h to the two program texts is not problematic; the results are given as examples. The problem arises when we apply h to d. If we intend h to apply to all three of “$x := 0$”, “**while** $\top$ **do** $x := 1$”, and d (in the Specification Halting Problem), then h is inconsistently defined, and its computability is not a sensible question. If we intend h to apply to only “$x := 0$” and “**while** $\top$ **do** $x := 1$”, then we should not require its program counterpart H to apply to D (in the original Halting Problem).

Mimic Halting Problem

In the original Halting Problem, the role of the computability assumption has nothing to do with computability. To illustrate, I invent a new kind

of function, which I will call "*mimic functions*". Some functions have a corresponding mimic function; some do not. If a function has a corresponding mimic function, the function and its mimic have the same results. (If you cannot see any distinction between a function and its mimic, you are in good company. But bear with me.) When we ask whether the halting function has a mimic, here is what happens.

Define function $h : \mathbb{S} \to \mathbb{B}$, where $\mathbb{S}$ is the data type of texts that represent specifications of computer behavior, and $\mathbb{B}$ is the boolean data type, so that when *h* is applied to a text representing a specification of terminating behavior, its result is $\top$, and when applied to a text representing a specification of nonterminating behavior, its result is $\bot$. Here are two examples:

$h(\text{“}x := 0\text{”}) = \top$

$h(\text{“}\textbf{while } \top \textbf{ do } x := 1\text{”}) = \bot$

Assume that function *h* has a mimic function *H*. Then

$H(\text{“}x := 0\text{”}) = \top$

$H(\text{“}\textbf{while } \top \textbf{ do } x := 1\text{”}) = \bot$

Define specification text *D* as follows:

$D =$ “**if** $H(D)$ **then while** $\top$ **do** $x := 1$ **else** $x := 0$”

Argue: If $H(D)$ is $\top$, then *D* represents a specification that is equivalent to **while** $\top$ **do** $x := 1$, which specifies nonterminating behavior, and so $H(D)$ is $\bot$. And if $H(D)$ is $\bot$, then *D* represents a specification that is equivalent to $x := 0$, which specifies terminating behavior, and so $H(D)$ is $\top$. We have an inconsistency.

Conclude: Mimic function *H* does not exist; function *h* is not mimicable.

The Mimic Halting Problem talks about specifications, as does the Specification Halting Problem, but its structure is now identical to the original Halting Problem. The role of the mimic function *H* in this proof is to take the blame, protecting the original function *h* from blame.

As hinted in parentheses before the proof, there is no distinction between a function and its mimic, so the Mimic Halting Problem is bogus. Likewise, in the original Halting Problem, function *h* and program *H* are both treated as mappings from the same domain elements to the same range elements, giving the same results. There is no difference between them that is used in the proof. In the original Halting Problem, the only role of program *H* is to take the blame, protecting the mathematical function *h* from blame.

Nature of Computation

There is an important difference between a mathematical function and a program. A program specifies a physical process that progresses through steps that take resources, in particular, time, and in the extreme, may take infinite time (not producing a result). A mathematical function is unconcerned with any process by which the result is produced; it is just a mapping from a domain to a range. In general, programs cannot be characterized as total mathematical functions from initial state to final state, with no mention of time. They have been characterized as total functions to a range augmented with an extra element that stands for nontermination. They have been characterized as partial functions. And they have been characterized as functions or relations with a time component. The time component may be as simple as a boolean variable that distinguishes finite from infinite time [5]. Or it can be a numeric variable (integer or rational or real) extended with an infinite value to account for nontermination [4].

Turing distinguished infinite and finite execution time, using the words “circular” and “circle-free”. After arguing that $H(D)$ can produce neither $\top$ nor $\bot$ due to inconsistency, Turing concluded that $H(D)$ is circular, producing no result, contrary to the computability assumption. The computability assumption is that function *h* can be computed by program *H*.

But the argument that follows that assumption makes absolutely no use of the computational nature of *H*. It treats *H* as a total function, just as if it were *h*. It applies *H* to the same domain elements, producing the same results that *h* would produce. Program *H* mimics mathematical function *h*, and its only role is to take the blame.

We have now seen function *h*, oracle $\mathcal{H}$, and program *H*, all applying to texts that represent programs, all telling us whether execution terminates. Oracle $\mathcal{H}$ seems to be intermediate between *h* and *H* in the sense that, like *h* it may not be computable, like *H* we can use it in a program. The proof makes no use of any distinction among them. Both $\mathcal{H}$ and *H* mimic *h*. Since *h*, $\mathcal{H}$, and *H* are the same, the three diagonal definitions

d = "**if** $h(d)$ **then while** $\top$ **do** $x := 1$ **else** $x := 0$"

$\mathcal{D}$ = "**if** $\mathcal{H}(\mathcal{D})$ **then while** $\top$ **do** $x := 1$ **else** $x := 0$"

D = "**if** $H(D)$ **then while** $\top$ **do** $x := 1$ **else** $x := 0$"

are the same. If, under the computability assumption, *D* is well-defined, then, without any computability assumption, $\mathcal{D}$ and *d* must be just as well-defined. And the same argument can be made using any of these diagonal definitions: $h(d)$, $\mathcal{H}(\mathcal{D})$, and $H(D)$ can be neither $\top$ nor $\bot$ due to inconsistency. The inconsistency is not due to a computability assumption.

The combination of *h* and *d* is inconsistent; to restore consistency, we withdraw the definition of *d*. Likewise the combination of $\mathcal{H}$ and $\mathcal{D}$ is inconsistent; to restore consistency, we withdraw the definition of $\mathcal{D}$. Likewise the combination of *H* and *D* is inconsistent; to restore consistency, we withdraw the definition of *D*.

Simple Problem

Define function $h : \mathbb{N} \to \mathbb{B}$ as a function from naturals to booleans. Here are two examples:

$h(0) = \top$

$h(1) = \bot$

Assume that h is computable, and that a program H for it has been written. Then

$H(0) = \top$

$H(1) = \bot$

Define natural D as follows:

$D =$ **if** $H(D)$ **then** 1 **else** 0

Argue: If $H(D)$ is $\top$, then D is 1, and so $H(D)$ is $\bot$. And if $H(D)$ is $\bot$, then D is 0, and so $H(D)$ is $\top$. We have an inconsistency.

Conclude: Program H does not exist; function h is not computable.

Function h could be the halting function applied to programs encoded as numbers, as in Turing's paper. Program number 0 could be one whose execution terminates, and program number 1 one whose execution does not terminate. Thus this proof could be talking about the halting function. But this time I omitted all the words describing what function h is supposed to be, so h could be any function that maps 0 to $\top$ and 1 to $\bot$. Perhaps h is the *even* function. And this proof comes to the absurd conclusion that such a function is incomputable. (The difference between the Simple Problem and the original Halting Problem is a change in the coding levels in the definition of D.)

This proof shows that the combination of definitions of H and D is inconsistent. If we also define

$d =$ **if** $h(d)$ **then** 1 **else** 0

we similarly find that the combination of definitions of h and d is inconsistent. There is no difference between h and H that is used in the

argument. So there is also no difference between the definitions of *d* and *D*. If we intend *h* to apply to *d*, we have an inconsistency before we get to the assumption of computability. If we withdraw the definition of *d* to restore consistency, then we should not ask *H* to apply to *D*.

Direct Proof

The form of proof we have been looking at so far is "proof by contradiction". We make a computability assumption, then we discover an inconsistency or contradiction, and we conclude that the computability assumption was wrong. But it is possible that the inconsistency was already there before the computability assumption, or that the inconsistency was due to a later unstated assumption. Here is a Direct Proof, with no computability assumption.

Let $\mathcal{L}$ be a programming language that includes the text (character string) data type and the boolean data type. Let *h* (the halting function) be a function that applies to texts that represent programs in $\mathcal{L}$. When *h* is applied to a text representing a program whose execution terminates, the result is $\top$, and when applied to a text representing a program whose execution does not terminate, the result is $\bot$.

In $\mathcal{L}$, define

T = (a program whose execution terminates)

N = (a program whose execution does not terminate)

P = (a program that applies to texts with boolean result)

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

As part of program compilation/interpretation, definitions are placed in a dictionary. Also as part of compilation/interpretation, these definitions are used to supply meaning for the identifiers *T*, *N*, *P*, and *D* whenever they are used in a program.

Program P applies to all texts, regardless of whether they represent programs. Therefore

if $P(D)$ **then** N **else** T

is a program, and so D is a text representing a program. Calculate

$\top$ definition of D

$= D =$ "**if** $P(D)$ **then** N **else** T" function transparency

$\Rightarrow h(D) = h(\text{“}$**if** $P(D)$ **then** N **else** $T\text{”})$ interpretation

$= h(D) = (h(\text{“}P(D)\text{”}) \wedge$ **if** $P(D)$ **then** $h(\text{“}N\text{”})$ **else** $h(\text{“}T\text{”}))$ given examples

$= h(D) = (h(\text{“}P(D)\text{”}) \wedge$ **if** $P(D)$ **then** $\bot$ **else** $\top)$ boolean algebra

$= h(D) = (h(\text{“}P(D)\text{”}) \wedge \neg P(D))$

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. Suppose $h(D)$ is $\top$. Then the last line of the calculation $h(\text{“}P(D)\text{”}) \wedge \neg P(D)$ means that execution of $P(D)$ terminates with result $\bot$, and so P is not an implementation of h. Suppose $h(D)$ is $\bot$. Then, according to the last line of the calculation, either execution of $P(D)$ does not terminate, or $P(D)$ is $\top$, 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.

Proofs of incomputablility are essentially classical (nonconstructive). That is clear in this Direct Proof: we do not know whether $h(D)$ is $\top$ or $\bot$, but either way, it differs from $P(D)$.

The first criticism of the Turing proof, or rather, of our modern formalization of it, was that we cannot say whether the diagonal definition defines a representation of a program. Strangely, by insisting in the Direct Proof that P applies to all texts, not just to texts that represent programs, we

can be certain that *D* represents a program; that solves the problem, even though *D* is the only argument we want to apply *P* to.

The second criticism of the Turing proof was that semantic transparency did not take full account of the interpretation of **if**-**then**-**else**; specifically, execution of the **if**-part must terminate. The Direct Proof does not have that flaw.

The Turing proof made a computability assumption, and concluded that this particular assumption was the cause of inconsistency. The Direct Proof makes no computability assumption. It introduces *P* as any program, and concludes that *P* cannot be an implementation of the halting function.

The Direct Proof has escaped all previous criticisms except the main one. Suppose language $\mathcal{L}$ includes the halting oracle. Since *P* is any program from texts to booleans, it could be the halting oracle (extended to give result $\perp$ (or anything) when applied to a text that does not represent a program). The Direct Proof concludes that the halting oracle differs from the halting function on program text *D*. But the halting oracle is defined to agree with the halting function on all program texts. So we have an inconsistency between the halting oracle and the diagonal definition. Since the halting oracle is identical to the halting function, not even differing by consuming resources, the inconsistency, as always, is between the halting function and the diagonal definition.

To avoid the word “oracle”, let *P* be any specification that applies to texts with a boolean result, require the halting function to apply to one diagonally defined specification, and conclude that *P* differs from *h*. But *P* could be *h*. Therefore there is an inconsistency between the halting function and the diagonal definition.

Reversed Halting Problem

The source of the inconsistency can be seen most clearly by reversing the process: this time I start with the inconsistency, and build toward the halting problem. Suppose we are interested in calumation, which is said to be a

property of program executions. 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. I will say it is a definition only because it makes my English sentences more direct.

At this stage, we have only an inconsistent definition, not yet any program purported to determine calumation. The inconsistency is far too obvious, so I now begin a process of obfuscation, being careful to maintain the inconsistency.

> Let *C* be a program whose execution calumates.
>
> Let *N* be a program whose execution does not calumate.

Now I can make the definition of *D* a little more formal.

> $D =$ **if** execution of *D* calumates **then** *N* **else** *C*

This is almost a normal recursive definition in any programming language. To finish making it so,

> Let *H* be a boolean expression (or boolean function of no arguments) that tells specifically whether execution of *D* calumates.

Now we define

> $D =$ **if** *H* **then** *N* **else** *C*

I will obfuscate a little more in a moment, but I want to take stock now. The inconsistency is still there. 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 is because we ask *H* to perform an impossible task. If it says $\top$ then $D = N$ and so it should say $\bot$; if it says $\bot$ then $D = C$ and so 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.

My next obfuscation is to make *H* more general. I want to give it a program parameter so that I can show it working on some programs, like *C* and *N*, and so make it seem more reasonable. But to do that, I first have to encode programs as data. So I redefine

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

Now *D* is completely innocent: it is just a text (character string). And I redefine *H* to have one text parameter *p* so that

$H(p) = \top$ if text *p* represents a program whose execution calumates

$H(p) = \bot$ if text *p* represents a program whose execution does not calumate

$H(p) = \bot$ (or $\top$, it does not matter which) if text *p* does not represent a program

Now I can proudly display $H(\text{“}C\text{”}) = \top$ and $H(\text{“}N\text{”}) = \bot$, showing how reasonable the specification of *H* is, before we consider the more difficult question of $H(D)$.

I have been using a simplified kind of program that has no input, or dependence on initial state. I have argued that input, or initial state creation, can be moved to the initial part of the program without loss of generality. Now I unsimplify. I redefine *H* to have two parameters:

$H(p, q)$ is a boolean expression that tells whether execution of the program represented by text *p* calumates when provided with input text *q*.

I must also redefine D:

$$D = \text{“}\textbf{if } H(D, D) \textbf{ then } N \textbf{ else } C\text{”}$$

For each parameter p and q, there is only one text D that we supply as argument, which is why parameterization was unnecessary. But the occurrence of “$H(D, D)$” motivates the name “diagonal”: the parameters (p, q) give us a two dimensional space, and point (D, D) is on its diagonal.

Finally, we make the well-known argument: if $H(D, D)$ is $\top$ then D represents a program equivalent to N and so $H(D, D)$ should be $\bot$; if $H(D, D)$ is $\bot$, then D represents a program equivalent to C and so $H(D, D)$ should be $\top$.

The definitions are now sufficiently complicated 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. Any assumption will do; we could assume there is life on other planets, but that would be too obviously irrelevant. So we assume that calumation is computable because that looks like it might be relevant, and then conclude that calumation must be incomputable.

We have been talking about calumation, which has not been defined. To be more definite, I now replace calumation by halting, and conclude that halting is incomputable. This proof is the same as Turing’s, except that we are using a modern programming language rather than Turing Machine operations, and we are using a text encoding rather than a numeric encoding. But this “proof” that halting is incomputable actually has nothing to do with halting, and nothing to do with computability.

Self-duplicating Program

Diagonalization was Turing’s method of creating a self-reference. We have been using the modern programming construct of recursive call for that purpose. Let me characterize a call by this little conversation:

"What computation are you calling, please?"

"The one named "*P*"."

If *P* is the name of the program (or specification) in which this answer occurs, then it is a recursive call. Another way to answer the question is directly. For example,

"What computation are you calling, please?"

"This one: "$x := 0$"."

To make a direct answer recursive (self-referential) requires a trick usually attributed to W. V. O. Quine: we make the answer be a text expression that evaluates to a representation of the program (or specification) in which it occurs.

Define

$oQ =$ (the opening quotation mark)

$cQ =$ (the closing quotation mark)

$T =$ (a program whose execution terminates)

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

$H(p) =$ (a program that implements the halting function)

$D(x,\ y,\ z) = x\ oQ\ x\ cQ\ y\ oQ\ y\ cQ\ y\ oQ\ z\ cQ\ z$

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 the reader to fill in the definitions of *oQ* and *cQ* with whatever their favorite programming language uses. For *T* choose any program whose execution terminates, and for *N* choose any program whose execution does not terminate. The ability to define *H* as a program is the computability assumption. And finally, program *D* has three text parameters and produces a text result by catenation.

After these definitions, the program

if $H(D(\text{“}\textbf{if}\ H(D(\text{”},\text{“},\text{”},\text{“}))\ \textbf{then}\ N\ \textbf{else}\ T\text{”}))\ \textbf{then}\ N\ \textbf{else}\ T$

is very interesting because H is being applied to an argument

$D(\text{“}\textbf{if}\ H(D(\text{”},\text{“},\text{”},\text{“}))\ \textbf{then}\ N\ \textbf{else}\ T\text{”})$

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

if $H(\mathcal{P})$ **then** N **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. The usual conclusion is that there cannot be any way to define program H, and the halting function is incomputable.

Suppose the language includes the halting oracle, and we use it as the definition of H. This time, we are not supposing that the halting function is computable. But we come to the exact same inconsistency.

Alternatively, drop the definition of H, and notice that the specification

if $h(D(\text{“}\textbf{if}\ h(D(\text{”},\text{“},\text{”},\text{“}))\ \textbf{then}\ N\ \textbf{else}\ T\text{”}))\ \textbf{then}\ N\ \textbf{else}\ T$

has the form

if $h(\mathcal{S})$ **then** N **else** T

where $\mathcal{S}$ is a text expression whose value represents the specification itself. Again, we are not supposing that the halting function is computable, but we come to the exact same inconsistency. Evidently, the inconsistency is not due to the computability assumption.

Bounded Halting Problem

One way to define the halting function h is by means of the bounded

halting function *b*. Define $b(p, n)$ to say whether execution of the program represented by text *p* terminates within *n* steps, or time units. Function *b* can be defined without inconsistency, and it is computable. Then we define *h* as

$$h(p) = \exists n : \mathbb{N} \cdot b(p, n)$$

where $\mathbb{N}$ is the natural numbers.

Program text can refer to any program name that has been defined (and placed in the dictionary). If a program *B* has been written for function *b*, and we define *D* as

$$D = \text{“}\textbf{if } B(D, n) \textbf{ then while } \top \textbf{ do } x := 1 \textbf{ else } x := 0\text{”}$$

there is still no problem because calls add to the step count, and *B* terminates its execution when the step count exceeds *n*. The step count, or execution time, accounts for the computational nature of programs. But when *h* is defined as above, the existential quantification hides the step count, reporting only whether it is finite or infinite, and so we lose most of the computational nature of programs. Now, when *d* is defined as

$$d = \text{“}\textbf{if } h(d) \textbf{ then while } \top \textbf{ do } x := 1 \textbf{ else } x := 0\text{”}$$

we have the same inconsistency as before. The pair of definitions (h, d) is inconsistent.

Automated Proof

Boyer and Moore [1] completely formalized and verified a proof of the incomputability of the halting function using an automated prover. They use a constructive logic in which all recursions must be well-founded to ensure termination. They define the bounded halting program $B(p, n)$, but they cannot define the halting program

$$H(p) = \exists n : \mathbb{N} \cdot B(p, n)$$

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

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

which cannot be expressed due to the unbounded quantification, but which can be proven by induction for any choice of nonterminating p.

In place of a numeric encoding of programs, they use a textual encoding, as does this paper. And they define a diagonal element CIRC using recursion, very similar to the diagonal definition of D in this paper. 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 do not), then HALTS applied to CIRC returns BTM. This is inconsistent, and they conclude that halting is incomputable.

The Boyer-Moore proof can be challenged as follows. Let x and y be natural variables, and let $x := ?$ mean that x is nondeterministically assigned an arbitrary natural number. There are many ways to implement nondeterministic assignment, so we can include it in a programming language. We have to decide whether EVAL is an interpreter, making a particular choice of value, or a semantics, making all choices of value. Either way, the problem is that natural induction fails to classify executions of the program

$$x := ?;\ \textbf{while}\ x > 0\ \textbf{do}\ (x := x - 1;\ y := ?;\ \textbf{while}\ y > 0\ \textbf{do}\ y := y - 1)$$

as always terminating.

If we exclude nondeterminism from programs, then Boyer and Moore proved what Turing proved, and what is proved in the Halting Problem box at the beginning of this paper. Like Turing, they did not question whether the mathematical halting function is inconsistent if it is required to apply to a diagonally constructed element. They show that a program with exactly the same properties is inconsistent, and conclude incomputability of an unchallenged mathematical function.

Incomputable by Reduction

Many functions have been proven to be incomputable by reduction (directly or indirectly) to the halting function. The reduction argument goes like this: if *f* were computable, then here is how we could use *f* to compute the halting function; but since the halting function is incomputable, *f* must be too. I make no challenge to the validity of the reductions, but I suggest they may be proving something different. They may be proving: if the definition of *f* were consistent, then here is how we could use *f* to compute the halting function; but since the definition of the halting function is inconsistent, the definition of *f* must be too. The inconsistency of *f* may be due to the application of *f* to a diagonally defined element that was unintended in the informal definition of *f*, just like the halting function.

Conclusion

In Turing's day, mathematics was a well established subject, and computation was very new. So it seemed reasonable to ask: which mathematical functions are computable? A computer scientist might ask the reverse question: which programs can be characterized as total functions from initial state to final state? In other words, which programs have terminating computations? Without interaction, termination is essential, otherwise there is no output. Total functions from initial state to final state are a reasonable characterization of noninteracting terminating computation. But the question of termination has lost most of its interest in recent years. Today, computation is usually interactive and potentially infinite; termination occurs when you click on "quit". Today, total functions from initial state to final state are inadequate to characterize computation.

Even for noninteracting computation, it is worthless to specify or guarantee or require termination. If a specification says that the result is 2 and we observe that the result is 3, we have grounds for complaint. If a specification says that execution is nonterminating and we observe termination, again we have grounds for complaint. If a specification says that execution is terminating (but gives no time bound), there is never a time

when we can observe the computation violating the specification. Specifying or guaranteeing or requiring termination (with no time bound) is worthless because it does not rule out any observation. It seems to me that theoretical computer science began badly by its emphasis on termination.

Turing said “what I shall prove is quite different from the well-known results of Gödel” [7, p. 259 referring to 3], but I think it is the same result. Gödel showed that if a logical formalism is expressive enough to encode its own interpreter, then the formalism is either inconsistent or incomplete. From a programmer’s point of view, if we apply an interpreter to a program text that includes a call to that same interpreter with that same text as argument, then we have an infinite loop. A halting program has some of the same character as an interpreter: it applies to texts through abstract interpretation. 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 incompleteness, we cannot require the program version to apply to texts that invoke the halting program, and we cannot conclude that it is incomputable. If we choose inconsistency, then it makes no sense to propose a program version. Either way, the incomputability argument is lost.

Acknowledgements

I thank IFIP Working Groups 2.1 and 2.3 for hearing this controversial idea. I thank Manfred Broy, Ernie Cohen, John Harrison, Kees Huizing, Ruurd Kuiper, Carroll Morgan, Natarajan Shankar, and Tom Verhoeff for good criticisms and discussion. Jules Desharnais suggested the direct version, and Lambert Meertens the self-duplicating version. I especially thank Olga Tveretina for inviting me to present this paper at COMPUTING 2011 (Symposium on 75 years of Turing Machine and Lambda-Calculus, no published proceedings) in Karlsruhe Germany on 2011 October 21. Some of

the aforementioned people tried to explain to me why this paper is all wrong, and they may not be pleased to have their names associated with it.

Appendix: Turing's Words

Here is the key paragraph from Turing's paper. To help the modern reader, I have added the square bracketed words.

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] of *H*. What does *H* do in the *K*th 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 *K*th 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*.

References

[1] R. S. Boyer and J. S. Moore, A mechanical proof of the unsolvability of the halting problem, J. ACM 31(3) (1984), 441-458.

[2] G. Cantor, Über ein Elementare Frage der Mannigfaltigkeitslehre, Deutsche Mathematiker-Vereinigung 1 (1890), 75-78.

[3] K. Gödel, Über Formal Unentscheidbare Sätze de Principia Mathematica und Verwandter Systeme I, Monatshefte für Mathematik und Physik 38 (1931), 173-198.

[4] E. C. R. Hehner, A Practical Theory of Programming, 1st ed., Springer, 1993. Current edition: www.cs.utoronto.ca/~hehner/aPToP

[5] C. A. R. Hoare and J. He, Unifying Theories of Programming, Prentice-Hall, London, 1998.

[6] H. G. Rice, Classes of recursively enumerable sets and their decision problems, Trans. Amer. Math. Soc. 74 (1953), 358-366.

[7] A. M. Turing, On computable numbers with an application to the Entscheidungsproblem, Proceedings of the London Mathematical Society s.2 42 (1936), 230-265; Correction: s.2 43 (1937), 544-546.

[8] A. M. Turing, Systems of logic based on ordinals, Proceedings of the London Mathematical Society s.2 45 (1939), 161-228.