Incomputable Indeed

Eric C.R. Hehner

Department of Computer Science, University of Toronto
Toronto ON, M5S 2E4, Canada
hehner@cs.utoronto.ca

Abstract There are no incomputable (or uncomputable) functions. The proof of incomputability of the halting (or any other) function is actually a proof of the inconsistency of its specification. Also, so-called "total correctness" is a poor choice of proof formalism.

0 Introduction

This paper argues against the concept of incomputability. It is not easy to make such an argument. That incomputable functions exist follows immediately from the accepted fact that there are more functions than there are programs (in any programming language). Obviously, some functions are incomputable. Furthermore, Alan Turing exhibited an incomputable function, the halting function, together with a proof that it is incomputable. A definition of the halting function, together with a proof of incomputability, appear in many textbooks and have been studied by many people, without finding flaws.

My arguments are on three levels. At a philosophical level, I present a view of what mathematics is, and what mathematical expressions express. I am obliged to present this view so that I do not have to argue the technical points at a disadvantage, using unfavorable terminology. After that preparation, I examine Cantor's diagonal argument leading to the claim that there are more functions than programs, and find it less than compelling. Finally, I argue that Turing's proof of the incomputability of the halting function has fatal flaws.

Arguments against the existence of incomputable functions have been made before by constructivist mathematicians. My arguments do not presuppose constructivism. They are made from a formalist viewpoint, with a contribution from formal methods of programming (theory of programming). I hope that my arguments make sense to any mathematician or computer scientist.

If the arguments in this paper are accepted, there are serious consequences for computer science: we must drop all mention of computability; and we must abandon the so-called "total correctness" proof formalisms in favor of formalisms that consider execution time, and can smoothly include programs whose executions do not terminate.

1 Philosophy

A map of the world would be the same if the shorelines had been discovered in a different order by different people. The world exists independent of our knowledge of it. That, at least, is the opinion of most people. Similarly, it is the opinion of most mathematicians that mathematical objects exist (in some abstract sense), and that truths about them are discovered. This opinion is called "platonism". According to a platonist, the order of discovery may be partly a historical accident, and our way of expressing truths may be a product of human design, but the truths themselves are independent of us, timeless and universal.

The opposing opinion is called “formalism”. According to a formalist, mathematics is not discovered but invented. It is a kind of language whose expressions are designed primarily to describe the world. By themselves the expressions are neither true nor false. Mathematical language is a human creation; the design of new mathematical expressions is influenced by their predecessors, by aesthetics, and by their usefulness in applications.

Galileo called mathematics the language of science and engineering; it is a language designed to describe and reason about the world we can observe in a precise and quantitative way. Platonists are amazed at the unreasonable effectiveness of mathematics in the natural sciences [11]; why should there be a strong correspondence between the world of mathematical objects and the world of physical objects? A formalist is less amazed that a language designed to describe scientific observations can be used effectively to do so [4]; it is neither more nor less amazing than describing the world with the words of a natural language, or with paint on canvas. Platonist mathematicians should be truly amazed, and therefore skeptical, about their unreasonable ability to discover anything at all about objects that cannot be sensed in any way.

Platonist mathematicians believe they discover truths about mathematical objects. Formalists believe that mathematics is a tool for organizing and summarizing some of our knowledge of the world, but would never say that they have discovered the “true” organization or summary. Platonism is the dominant philosophy, and it is therefore standard to talk about the existence of mathematical objects. Formalists may talk the same way in order to fit into the dominant mathematical culture, but they understand “existence” to mean consistency (satisfiability) of description, and “nonexistence” to mean inconsistency (unsatisfiability) of description. The arguments in this paper take the formalist position.

2 the Diagonal Argument

Cantor's diagonal argument [1] is supposed to show that there are more real numbers than integers. It is usually presented in platonic terms, informally, as follows. Without affecting the argument, we shall consider only naturals (nonnegative integers), and reals between 0 and 1 (inclusive) represented as a decimal point followed by an infinite sequence of decimal digits. The argument is often presented with the aid of a picture, which motivates the name “diagonal”.

0 → . **1** 4 1 5 9 2 6 5 3 5 8 ⋯
1 → . 1 **4** 2 8 5 7 1 4 2 8 5 ⋯
2 → . 1 0 **1** 0 0 1 0 0 0 1 0 ⋯
3 → . 6 5 3 **4** 9 7 6 4 8 4 2 ⋯
4 → . 7 6 4 3 **6** 9 0 6 3 4 5 ⋯
:

The picture depicts a mapping from naturals to reals. We now form a real number that is not on any row by taking its first digit different from the first digit in the first row (row 0), its second digit different from the second digit on the second row (row 1), and so on. From the picture, we might form the number .85853... . Since our mapping from naturals to reals was arbitrary, it is concluded that there is no “onto” mapping from the naturals to the reals. An onto mapping in the reverse direction is easy to construct, so it is concluded that there are more reals (between 0 and 1) than (nonnegative) integers.

One error in this diagonal argument is due to the fact that different digit sequences can be equal as real numbers; for example, .12399999...=.12400000... . Leaving that error aside, the formalist objection is that in the picture, on each row, we do not have an infinite sequence of digits at all; we have a finite sequence followed by three dots. How can we

remove the three dots and become formal? In what sense can we have an infinite sequence? It is an ancient question. A modern answer is that we have an infinite sequence when we have a way of generating as much as we want of it. This is a constructive answer, and there are some mathematicians who insist on it. I will not insist on it, but I want to explore it for a moment. In this formalization, we must replace each row of our picture by a program, which we can do completely without ellipsis. That takes care of the horizontal dots. Now, how do we get rid of the vertical dots? How can we have an infinite sequence of programs? Again, an answer is to have a program to print as many of the rows as we want. We can certainly write a program (in some language) to print out all programs (in that language), but we want a list of all and only those programs that produce an infinite sequence of digits. If we posit a program P to produce all and only these infinite-sequence programs S_i, we obtain a contradiction by the diagonal argument, as follows. We write a new program D to simulate P, producing the S_i, and in turn simulate each S_i up to i digits (without printing any), and then output a different digit. We thus create an infinite-sequence program that differs from all S_i. To argue as Cantor did, we would have to conclude that there are more infinite-sequence programs than integers. If the inability to list all the reals is cause for concluding that there are too many of them, then the inability to list all the infinite-sequence programs should equally be cause for concluding that there are too many of them. But of course that conclusion would be absurd: the infinite-sequence programs are a subset of all programs, and we can list all programs. The proper conclusion is simply that there cannot be a program to generate all and only the infinite-sequence programs. In formalist words, the specification "generate all and only the infinite-sequence programs" is inconsistent. Likewise, we should conclude from Cantor's argument that there is no sequence of all infinite sequences, or, more briefly, there is no sequence of all reals. This is not a conclusion about the sizes of sets.

The constructive formalization of Cantor's argument is not the only formalization; another comes from function theory. The details of the theory do not matter; I mention here only the key definition and theorem. Define relation © (a pre-order) between sets A and B as

$$A © B \quad = \quad \exists f: A \to B \cdot \ \forall b: B \cdot \ \exists a: A \cdot \ f(a) = b$$

The part to the right of the defining equals can be read "there is an onto function from A to B". We might choose to pronounce $A © B$ as "A covers B". Next, let *nat* be the set of natural numbers, let *dig* be a set of at least two digits, and let *nat*→*dig* be the set of functions from *nat* to *dig*, or in other words, the set of infinite sequences of digits. In this notation, with standard proof rules, we can prove

$$\neg(nat © nat \to dig)$$

It is a beautiful theorem saying that there is no onto function from the natural numbers to the infinite sequences of digits. But what does it say about the sizes of sets?

Two finite sets have the same size if their members can be put in one-to-one correspondence. Cantor asks us to extend this property of finite sets to infinite sets. Doing so, we are compelled to say that there are as many even integers as integers. (What is the probability that an unknown integer is even? Why?) In any finite interval of integers, only about half of them are even, but we are told that not all properties of finite sets can be correctly extended to infinite sets. We are also compelled to say that there are more reals than rationals, even though there is a rational between every two reals. Again, if we have a finite set of rationals and a finite set of reals such that there is a rational between every two reals, then that set of rationals is at least as large as that set of reals (minus one member); and again we are told that this property does not extend to infinite sets. Why is the one-to-one property chosen for extension at the expense of the other two properties? Why not extend the other two properties at the expense of the one-to-one property?

The platonist philosophy is that mathematical facts are discovered, and in particular, it has been discovered that there are more reals than rationals. The formalist philosophy is that we design mathematics for intended applications. If there are any applications that would benefit by assigning different sizes to infinite sets, we need to decide how best to do it. If there are any such applications, perhaps Cantor's relation is the best way for some, but not for others. Cantor's relation may be interesting, but we are not compelled to accept it as a comparison of sizes. Cantor could have chosen to pronounce his relation $A©B$ as “ A is more beautiful than B ”. (The suggestion is no more bizarre than calling a number “perfect” because it is the sum of its divisors.) If he had done so, would we now say that it is an established fact that the reals have more beauty than the rationals? Or would we realize that the words “more beautiful”, when used to compare sets, have nothing to do with beauty? After a century of established usage, it may now be very difficult for many of us to realize that the words “larger than”, when used to compare infinite sets, have nothing to do with size.

The claim that there are more functions than programs is based on the same sort of diagonal argument, together with the same decision to use Cantor's relation as a comparison of infinite set sizes. Here again, we do not have to make that decision. It is not a fact that there are more functions than programs, and we cannot rest the case for incomputability on that argument.

There is a much more direct argument for incomputability: exhibit an incomputable function. The first and most famous incomputable function is Turing's halting function. Before I present it, I would like to issue a warning about natural language, and then to present a small amount of programming theory.

3 Natural Language Can Lead Us Astray

The acceptance of 0 as a number has taken a long time, and is still incomplete. In English, no-one quite knows whether to treat 0 as singular or plural, does he? On your keypad or telephone it is placed after 9 , which is mathematically silly. In the 1991 Toronto phone book, there is a page that helpfully gives the time difference to various places in the world; to the U.K. it says “+5”, and to Costa Rica it says “–1”. But to Cuba it says “NA”, and the legenda explains “time difference not applicable”. By 1996 they tried to correct it; for Cuba it says “=”, with the same explanation. In 1997 they discovered the number 0 , but they felt the need then, and still do today, to explain that 0 means “no time difference”.

When we say “There are a number of issues to discuss.”, we do not mean there might be 0 of them. When 0 really is a possibility, people often add the phrase “if any”, as in “Please put all the leftovers in the fridge, if there are any.”. They create a case analysis, when none was needed. For example, the 1991 Canadian census asked the question “How many persons who have a usual home somewhere else in Canada stayed here overnight between 1991 June 3 and 4?”, then offers a place to tick if there were none, and a box to fill with the number of persons if there were some. The people designing the form probably know perfectly well that the box is sufficient, but without the place to tick “if none” they would be overwhelmed by people complaining that they cannot answer the question. Perhaps someday we will not feel the need to ask for “the number, if any”; we will simplify by just asking for “the number”, accepting 0 as an answer.

The Fortran language of 1955 had a loop construct, but its body had to be executed at least once; I suppose it seemed senseless to have a loop whose body might be executed 0 times. The design flaw was corrected in Algol in 1958, and in PL/I, and in Pascal, in part:

iteration might be 0 times, but the data structure over which one is iterating, the array, had to have at least one element. In Pascal that meant there was no null string. And that put the algebra of data structures back where the algebra of natural numbers was prior to 1930. We learn, but slowly; two steps forward, one step back.

The general public may not have fully accepted 0 as a number yet, but most mathematicians and computer scientists have. My purpose in reminding you of the long slow struggle is to foreshadow another similar struggle. In English, one sometimes hears the phrase "if ever", or "if and when", as in "I'll deal with that if and when it happens.". If we just say "I'll deal with that when it happens.", undoubtedly someone would immediately ask: "What if it never happens?". But it seems to me that case is already covered: if it never happens, I'll deal with it never. We simplify by eliminating the case analysis, and to do that we must learn to accept ∞ as an answer to the question "when?". We are not bothered by the different grammar in the two sentences "I don't have any bananas." and "I have 0 bananas."; one uses a negative verb and the other a positive verb, but we take them to mean the same thing. Likewise we should take "It never happens." and "It happens at time ∞." to mean the same thing. My reason is mathematical, not linguistic, as I shall explain in the next section. My point here is just to say that the best mathematics is not always a copy of natural language.

Some natural language sentences sound reasonable enough, but on close inspection, they are seen to be meaningless. For example, in a very small town named Russellville there are exactly 3 men, named a, b, and c. The one named b is the barber. The reasonable-sounding sentence is: "The barber shaves all and only the men in Russellville who do not shave themselves.". What's wrong with that? Formalizing,

$$\forall m \in \{a, b, c\}\cdot\ (b\ shaves\ m)\ =\ \neg(m\ shaves\ m)$$

An immediate consequence is obtained by specialization:

$$(b\ shaves\ b)\ =\ \neg(b\ shaves\ b)$$

We conclude that the reasonable-sounding sentence is inconsistent. The inconsistency may not be obvious from the English sentence, nor even from its formalization. But the contradictory consequence shows that the statement is inconsistent. To escape from inconsistency, we can weaken the statement to

$$\forall m \in \{a, c\}\cdot\ (b\ shaves\ m)\ =\ \neg(m\ shaves\ m)$$

Either we leave the defining characteristic of the Russellville barber incomplete, not applying it to himself, or we suffer inconsistency.

Russell's barber could, perhaps, be considered an example of the diagonal argument (the picture would list men on each axis, with a boolean value at each table entry telling whether someone shaves someone). An even simpler example is the liar's paradox: "This sentence is false.". Formalizing, using $\top$ for "true" and $\bot$ for "false", it is easy to say that some sentence S is false:

$$S = \bot$$

We are told that sentence S is that sentence. If it is, then it has the same truth value.

$$S = (S = \bot)$$

Since boolean equality is associative, this is equivalent to

$$(S = S) = \bot$$

and since equality is reflexive, this is equivalent to

$$\top = \bot$$

and the contradiction is apparent. (Someone might argue that $S = (S = \bot)$ doesn't quite say "This sentence is false.", but only that some sentence S is true if and only if it is false. In that case, the sentence "This sentence is false." is an instance of such an S.)

The liar's paradox is so simple that it doesn't fool anyone (except St.Paul [0]). Russell's barber is a little more complicated, and it fools some people; why shouldn't the

barber shave all and only those men who do not shave themselves? If it isn't complicated enough to fool you, I'm sure a more complicated example can be devised so that it looks reasonable to you until you analyze it. Most paradoxes turn out to be just a complicated way of asserting “false”.

There is another reason why a reasonable-sounding sentence may be meaningless. A father might say to his daughter “Please tidy your room.”. The daughter replies “Yes, daddy, I will.”. The request sounds reasonable, and the reply sounds positive, but every daughter knows that she has not committed herself to tidying her room. Later, father will say “Daughter, you have not yet tidied your room.”, and daughter will reply “I know, daddy, but I will.”. This meaningless couplet can be repeated many times until father realises that he must impose a deadline, and daughter then knows that she must tidy her room by the deadline or face the consequences.

Suppose someone claims that the execution of a program halts within 10 seconds. This claim might be part of the specification, or guarantee, that comes with some software. We execute the program, and if the execution continues for more than 10 seconds, we can complain, demand our money back, and sue for damage. The claim makes clear under what circumstances it is violated, and that gives it meaning. If the claim is that execution halts within 10^{100} seconds, we are prevented by practical resource constraints (human lifetime) from testing it. In principle the test is understood, and we could say the claim has a meaning. Or, we could say it is not very meaningful because of the impracticality. Now suppose someone claims that the execution of a program halts, with no time bound. How can we test this claim? After any amount of execution we still cannot say that the claim has been violated.

Perhaps we don't need to look at the execution to check the claim of termination. We can look at the program, and prove, according to some programming theory, that its execution will or won't terminate. Maybe we have a choice of theories, and according to one theory, execution will terminate, and according to another, it won't. For example, execution of the loop (i is an integer variable)

while $i \neq 0$ **do if** $i > 0$ **then** $i{:=}\ i{-}1$ **else** $i{:=}$ (an arbitrary nonnegative integer)

terminates according to the theory in [8] (the semantics of a loop is the least fixpoint of its generating function), but may not terminate according to the theory in [2] (the semantics of a loop is the limit of a sequence of approximations). How do we validate a theory? A theory is valid if all its claims correspond to observation, so we are thrown back to looking at execution. The claim was not about the program, but about its execution.

According to Shannon's Information Theory [9], a test that cannot fail conveys no information. Similarly, according to philosopher Karl Popper [7], a claim that cannot be challenged is empirically meaningless. According to Bayesian probability [12], we cannot confirm something (increase its probability) without tests that can potentially disconfirm it (decrease its probability). Whether we agree or disagree with Shannon, Popper, and the Bayesians, I think we all must agree that a guarantee is worthless if it offers no possibility for redress. The claim that execution will terminate, with no time bound, offers no possibility for redress, and is therefore worthless. Oddly enough, the opposite claim, that execution will not terminate, does offer the possibility of redress, and is worth making.

4 Programming Theory

A specification (of anything) has to distinguish those things that satisfy it from those that do not. In the scientific tradition, we use variables for quantities of interest, and observation of something provides us with values for the variables. When the variables of a

specification are instantiated by values from an observation, the specification must say "true" or "false". A formal specification is therefore a boolean expression whose nonlocal (free) variables represent whatever quantities are of interest. (We say "boolean expression" for any expression of type boolean, regardless of the types of variables and subexpressions within it. We say "predicate" for a function whose result type is boolean.)

When we specify computation, we may be interested in the initial and final values of the state variables, the execution time, the space occupied by the computation, and interactions with the computation. Suppose we are interested in the initial values x, y and final values x', y' of two integer state variables, and the execution time t whose domain is nonnegative numbers plus ∞ so that we can talk about nonterminating computation. Here is an example specification, measuring time in microseconds:

$$(x \ge 0 \Rightarrow x'=x+y \land y'=x-y \land 1 \le t < 2) \ \land \ (x<0 \Rightarrow t=\infty)$$

This particular example illustrates the specification of result values, and of lower and upper bounds on execution time. I might also say that it illustrates the specification of nontermination, but that is just a further example of a lower bound on execution time.

Here is a similar but subtly different example specification:

$$(x'=x+y \land y'=x-y) \ \land \ (x \ge 0 \Rightarrow 1 \le t < 2) \ \land \ (x<0 \Rightarrow t=\infty)$$

This specification says what the final values of variables x and y are, and then goes on to say that if x starts nonnegative the execution time is between 1 and 2, and if x starts negative the execution takes forever. Although these two specifications are not logically equivalent, they describe exactly the same computations. The second one is a little strange because if x starts negative, it says both what the results are, and that execution takes forever. A claim about results at time infinity may be worthless, but it is not self-contradictory. There is a practical reason for writing a specification this way: if we can prove that a computation satisfies each of the conjuncts separately, then we get their conjunction for free. It is enormously helpful to be able to break the proof obligations into parts, and especially to separate the results from the timing.

We are specifying computation, not programs. A program is also a specification of computation; it is a specification that a computer can execute, to produce the specified behavior. The purpose of a specification, in the first instance, is to be clear and easily understood. For that purpose, a specification language should be unrestricted, open to notations from the application area, and open to new notations invented to help make a particular specification clear. The purpose of a program, in the last instance, is to be executed; for that purpose, a programming language is restricted to notations that can be compiled or interpreted efficiently. The programming language should be part of the specification language for two reasons: one is so that programming notations can be used in the initial specification whenever they help to make it clear; the other is so that the initial specification can be refined, little by little, into a program, and at intermediate stages there is a meaningful mixture of programming and nonprogramming specification notations. If the initial specification is S, and the final specification (the program) is P, they must satisfy

$$P \Rightarrow S$$

so the behavior obtained by executing P will satisfy S.

A program is a specification, and a specification is a boolean expression; therefore a program is a boolean expression. Suppose the quantities of interest are x, y, x', y', t as before. Suppose also that execution of the assignment statement $x{:=}\ x+y$ takes time 1 (in some units). Then

$$(x{:=}\ x+y) \ = \ (x'=x+y \land y'=y \land t=1)$$

In a similar fashion, all programs are boolean expressions whose nonlocal (free) variables are the quantities of interest.

If the time variable is real-valued and its values really do account for the execution time,

then we have a calculus that allows us to prove real-time properties of computation. To be independent of the computing platform, we can use a more abstract measure of time in which the time variable is integer-valued, and we just count loop iterations; this measure is the most practical one most of the time. There is an even more abstract measure of time in which the time variable is boolean-valued; it distinguishes finite execution time from infinite execution time. From here on, we will use boolean variable f to mean "execution time is finite". A so-called "partial correctness" specification has the form $f \Rightarrow S$, and a so-called "total correctness" specification has the form $f \wedge S$, where S talks about the desired results. We can also specify nontermination as $\neg f$. (A so-called "total correctness" proof system provides the means for proving termination without a time bound (which is worthless), and does not provide any means for proving nontermination (which would be worth proving). I use the word "so-called" for its derisive value.)

If we have a numeric time variable, the semantics of loops is very simple, and strictly first-order. If we have a boolean time variable, or no time variable, the semantics of loops becomes considerably more complicated, requiring a least-fixpoint operator to compensate for the lack of a time variable that can "tick" and thereby make progress. A numeric time variable provides more information with less semantic cost and with easier proofs than a boolean time variable; for further discussion of this point see [5]. But in this paper, a boolean time variable is exactly what we need in order to discuss the halting problem.

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

ok
if B **then** P **else** Q
while B **do** P

The first one, *ok* , is the "empty statement", or do-nothing statement (some people call it *skip*). Its execution terminates immediately, leaving the state unchanged. In the next two, B is a boolean expression of some restricted form, and P and Q are any programming statements; they are the standard conditional and loop statements found in most programming languages (except perhaps for minor syntactic differences). In programming theory they are generalized by allowing B to be any boolean expression, and allowing P and Q to be arbitrary specifications, in which case the **if**- or **while**-statement is not a programming statement, but it is still a specification. For their formal definition, see [6].

5 the Halting Problem

The halting function (predicate) is defined to tell whether a program's execution terminates. I have raised the question whether termination without a time bound is meaningful; for the purpose of continuing, I will set aside that question. I will make two simplifications to the standard formulation, neither of which changes anything essential. We need to encode programs as data so we can apply the halting function to something that represents a program. In the standard formulation, programs are numbered, so we can apply the halting function to a number representing a program. Instead, I use a more transparent encoding: a program is represented by its text (character string). (That is how a program is presented to a compiler or interpreter.) The other simplification is to eliminate all mention of initial state (input). One way to do that is to define the halting function applied to program text p as saying whether " p halts from all initial states" or " p fails to halt on some initial state". Another way to do it is to pick some initial state as the one where execution of any program always starts; if you want some other initial state, just begin the program with some initializing assignments to create the state you want.

Define predicate $H: text \to bool$, where $text$ is the text data type and $bool$ is the boolean data type, so that when H is applied to a text representing a program whose execution terminates, its result is $\top$, and when applied to a text representing a program whose execution does not terminate, its result is $\bot$. For example,

H (" ok ") = $\top$

H (" **while** $\top$ **do** ok ") = $\bot$

Define text T (for Turing) as follows:

T = " **if** H (T) **then while** $\top$ **do** ok **else** ok "

Assume that H is computable, and that a program for it has been written; then T represents a program. When identifiers (like H and T) occur within a program, they are interpreted by looking them up in a dictionary of definitions; we place the definitions of H and T in the dictionary. Now we ask: what is the result of H (T) ? If T represents a program whose execution terminates, then H (T) is $\top$, and so T represents a program that is equivalent to **while** $\top$ **do** ok , whose execution does not terminate. And if T represents a program whose execution does not terminate, then H (T) is $\bot$, and so T represents a program that is equivalent to ok , whose execution does terminate. Conclusion: H cannot be computable. That's the orthodox argument, and the orthodox conclusion, first made by Turing [10], and now found in many textbooks.

Since programs are a special case of specification, let me generalize H to apply to all specification texts, not just to program texts. If a specification requires termination, H is true; if it allows nontermination, H is false. In particular,

H (" f ") = $\top$

H (" $\neg f$ ") = $\bot$

And this time, we don't make any assumption that H is computable. Define specification text S as follows:

S = " **if** H (S) **then** $\neg f$ **else** f "

or, equivalently and more simply, define

S = " $H(S) \neq f$ "

Now we ask: what is the result of H (S) ? If S specifies terminating behavior, then H (S) is $\top$, and so S specifies nonterminating behavior. And if S specifies nonterminating behavior, then H (S) is $\bot$, and so S specifies terminating behavior. What do you conclude from that?

This argument about specifications has exactly the same form as the orthodox argument about programs. Both arrive at a self contradiction. We look for a way out by looking for an assumption that can be withdrawn. In the argument about programs, the assumption was made that H is computable, so we withdrew that assumption. But from the argument about specifications, we see that the problem is still there, even without that assumption.

My conclusion is that we cannot consistently define H to tell us the termination status of all specification texts, including S . The inconsistency is not immediately apparent, but the above argument shows us that it is there. We can restore consistency by leaving the definition of H incomplete; specifically, it cannot tell us about S .

Let me try to make the inconsistency in the definition of H more apparent. Within S , H (S) and f have the same role. So S represents something equivalent to

if f **then** $\neg f$ **else** f

which says, as directly as possible, that if execution terminates, then it doesn't terminate, and if it doesn't terminate then it does. By simple boolean algebra,

$$
\begin{aligned}
& (\textbf{if } f \textbf{ then } \neg f \textbf{ else } f) \\
= \; & ((f \wedge \neg f) \vee (\neg f \wedge f)) \\
= \; & (\bot \vee \bot) \\
= \; & \bot
\end{aligned}
$$

(independent of the interpretation of f), so it is an inconsistent specification; it is not satisfied by any computation, terminating or nonterminating. It is not reasonable to ask H to tell us the termination status of an inconsistent specification.

Returning to the "program" example

T = " **if** H (T) **then while** ⊤ **do** *ok* **else** *ok* "

if H is a well-defined but incomputable function, then T does not represent a program, but it still represents a specification. Does T represent a consistent specification? If we assume it does, then it is reasonable to ask H about its termination status. As before, if T specifies terminating behavior, then $H(T)$ is ⊤ , and T represents a specification that is equivalent to **while** ⊤ **do** *ok* , which specifies nonterminating behavior. And if T specifies nonterminating behavior, then $H(T)$ is ⊥ , and T represents a specification that is equivalent to *ok* , which specifies terminating behavior. Without the assumption that H is computable, we still have a contradiction. What assumption do we withdraw now? the assumption that T represents a consistent specification? In that case, it is unreasonable to ask H about its termination status, and we lose the very specification we were using to demonstrate that H is incomputable.

The problem with H doesn't stop there. If we could define H consistently on just the consistent specifications, then we could consistently extend its definition to all specifications, for example, by saying $H(s) = \bot$ for all inconsistent specifications s . (Increasing the domain of a function by giving a simple value for the new elements cannot introduce an inconsistency.) Now if T is inconsistent, then $H(T) = \bot$, and so T represents a specification equivalent to *ok* , which is a consistent specification. It seems there is no way out.

6 the Interpreter Problem

The situation is exactly the same for an interpreter of boolean expressions (also known as a prover, or as Tarskian semantics, or as denotational semantics). Suppose we try to define I: *text*→*bool* so that, when we apply I to a text representing a boolean expression, we get the result of evaluating the boolean expression. For example, using juxtaposition for string catenation,

I("⊤") = ⊤
I("⊥") = ⊥
I("¬" s) = ¬ $I(s)$
I(s "∧" t) = $I(s)$ ∧ $I(t)$
I(s "∨" t) = $I(s)$ ∨ $I(t)$

And so on. It is a familiar fact to programmers that we can write an interpreter for a language in that same language, and that is just what we have started doing here. If we complete the definition of I , then I acts as the inverse of quotation marks; it "unquotes" its operand. (That is what a program interpreter does: it turns passive text into active execution.) Now define text G (for Gödel) as follows:

G = " **if** $I(G)$ **then** ⊥ **else** ⊤ "

or, equivalently and more simply, define

G = " ¬ $I(G)$ "

Applying I to G yields inconsistency, as follows.

	$I(G)$	replace G by its equal
=	I(" ¬ $I(G)$ ")	I unquotes its operand
=	¬ $I(G)$	

This is exactly Gödel's incompleteness theorem [3]: G is saying that G is the negation of

a theorem. Either we leave I incompletely defined (specifically, it does not interpret G), or we suffer inconsistency.

Wait a minute; there is a way out. We can write a program for interpreter I , and H is just a simplification of I . I tells us the result of evaluating, and H just tells us whether there is a result. So H really is a program. Applying H to T and to S results in an infinite loop, as does application of I to G . We could say that H does deliver a result for T and for S , and I does deliver a result for G , but only at time ∞ . The "incomputable function" H is just a program whose execution, for some input, is nonterminating. Such programs are common, and some of them are useful.

7 the Vacuum Cleaner Problem

Here's a "proof" that a vacuum cleaner is unbuildable. If you could build one, then you could use it to clean out its own bag. But that is a self contradiction (making the bag empty makes the bag full, and vice versa). So a vacuum cleaner is unbuildable.

The "proof" that a vacuum cleaner is unbuildable is like the "proof" that the halting function is incomputable in the following ways. It accepts without question that a vacuum cleaner is at least a meaningful, consistent concept, just as the standard incomputability proof accepts without question that a halting function is at least a meaningful, consistent concept. Then the vacuum cleaner is applied to itself (through a hole in the bag made for the purpose), just as the halting function is applied to itself (through a code made for the purpose). And, most importantly, time is not considered in the argument: in each case, there is no static solution, so we have inconsistency. To restore consistency, we seem to have three options.

The first option, à la Turing, is to remain steadfast in the belief that the vacuum cleaner and halting function are at least meaningful consistent concepts, but to label them as "unbuildable" and "incomputable" respectively. That withdraws an assumption made in the argument, but it was an irrelevant assumption. If you could just specify (never mind build) a vacuum cleaner, you arrive at the same contradiction. If you could just specify (never mind compute) the halting function, you arrive at the same contradiction. This option is not a way out. Neither "unbuildability" nor "incomputability" serve the purpose for which they were invented: to restore consistency.

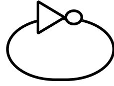
The second option, à la Gödel, is to say that the definition of a vacuum cleaner, and the definition of the halting function, are inconsistent unless we leave them incomplete, and we do not apply them to the example that gives rise to the contradiction. This option works, and it is the best we can do if we insist on a static, timeless solution.

The third option is to add a time variable. Then we can ask what really does happen (over time) if we apply them to the troublesome examples. What really happens if someone uses a vacuum cleaner to clean out its own bag is that they create an infinite loop, blowing dirt forever around a circular hose. That is not an inconsistency. Indeed, there are physical systems built intentionally as infinite loops; for example, pumping electrons around a circuit, doing useful work as they go. Likewise, applying the halting program to its troublesome example is an infinite computation, not a self contradiction.

8 the NOT Gate Problem

A simpler example is the problem of the NOT gate. If we could build one, then we could use it in a closed circuit that includes just one NOT gate, and nothing else (a physical

representation of the liar's paradox).



If we ignore time, we find an inconsistency: assuming either final state of the circuit leads to a contradiction. The inconsistency is not eliminated by labeling NOT gates "unbuildable" or "incomputable". The problem is eliminated if we outlaw this particular use (and all similar uses) of the NOT gate. But the best solution is to admit that a NOT gate takes time; we look at the circuit's behavior over time, we recognize it as an infinite loop, and we do not worry about what its final state might be. It is a useful circuit called an oscillator. (A practical oscillator is more complicated, but at its heart there is a NOT gate in a loop.)

9 Implementability

For simplicity of presentation, I have been considering programs without input. If we have input in the form of the initial value x of a state variable, and output in the form of the final value x' of the same state variable, and the execution time is t, then a specification S is implementable if and only if

$$\forall x\cdot \exists x', t\cdot\ S$$

Without the input, implementability reduces to the existential quantification only, which is to say the satisfiability of S, or as we have been saying, the consistency of S.

Also for simplicity of presentation, I used a single time variable, either f for boolean time or t for numeric time, but the laws are nicer and the calculations easier if we use a pair of variables. For numeric time, let t be the start time and let t' be the finish time (possibly ∞) of the computation. Then S is implementable if and only if

$$\forall x, t\cdot \exists x', t'\cdot\ S \wedge t\le t'$$

For boolean time, let f mean that the computation starts at a finite time and let f' mean that the computation finishes at a finite time. Then S is implementable if and only if

$$\forall x, f\cdot \exists x', f'\cdot\ S \wedge (f' \Rightarrow f)$$

To be implementable, I do not mean that the preceding formulas must be platonicly true; I mean that they must be formally provable.

Any implementable specification can be refined by a program unless the specified resource bounds are too small [6]. In other words, for any implementable specification S there is a program P such that $P\Rightarrow S$, unless the specified resource bounds are too small. Whether a time bound is too small depends on the computing operations allowed (parallelism? quantum computing operations?), and the chosen measure of time. If, like Turing, we are unconcerned with resource bounds, then any implementable specification (any consistent specification) can be refined by a program.

Suppose we have a specification without resource limitations. Then a constructive proof of implementability constitutes an implementation. If there is a classical proof but no constructive proof, then suppose we can generate the states in some order, and for each, test whether it is satisfactory as a final state; this test requires an interpreter (theorem prover). Do not wait for the result of the test, but continue in parallel to generate states and initiate tests. This exhaustive search again constitutes an implementation. I am not suggesting it is a practical one; I offer it only as a partial substantiation of the claim that any specification that is implementable as defined above (any consistent specification) is programmable.

10 Conclusion

When I began programming, I put my program, punched onto a deck of cards, in the “in” basket; hours later, the computer operator fed it into the computer, and put the output in the “out” basket, where I retrieved it. Computing involved an initial input and a final output, with no possibility of interaction. A “total correctness” theory is based on this out-of-date paradigm: without interaction, termination is essential. With the addition of interactive communication, nonterminating computations can be useful, so a semantics that does not insist on termination is useful. Furthermore, for some programs, for some inputs, we might well want to guarantee nontermination, which a “total correctness” formalism does not do. The operating system, even when I began programming, was an interacting, nonterminating computation. These days, every program I use terminates its execution when I click on “quit”. And each response to me must come within the time bound of my patience.

Characterizing programs as functions from input to output is out-of-date for the same reason. To accommodate nondeterminism in specifications we have generalized from functions to relations, but we are still stuck with the idea that termination (with no time bound) is important. To prove termination, you must do all the work of finding a time bound (variant, bound function, well-founded ordering), but without the reward. And, according to “total correctness” proof systems, you must prove termination before you can conclude anything about results or time bounds. And when you have proven termination, you have proven something worthless, because no observation of a computation can falsify it (nontermination is unobservable). It is time to retire the concept of “total correctness”, and to terminate our obsession with termination.

We cannot consistently define a function to tell us the termination status of all specifications, including inconsistent specifications. We cannot consistently define a function to tell us the termination status of just the consistent specifications. We cannot consistently define a function to tell us the termination status of just the programs. There is no halting function.

But there is a halting program. Self application results in an infinite loop, which is not surprising. This in no way implies the existence of incomputable functions.

11 References

[0] Christian Bible, Titus, chapter 1 verse 12: St. Paul takes Epimenides version of the liar's paradox (all Cretans are liars) at face value, and elaborates: “One of themselves, even a prophet of their own, said, The Cretians are alway liars, evil beasts, slow bellies.”

[1] G.Cantor: über ein Elementare Frage der Mannigfaltigkeitslehre, *Deutsche Mathematiker-Vereinigung* v.1 p.75-78, 1890

[2] E.W.Dijkstra: *a Discipline of Programming*, Prentice-Hall, 1976

[3] K.Gödel: über Formal Unentscheidbare Sätze de Principia Mathematica und Verwandter Systeme I, *Monatshefte für Mathematik und Physik* v.38 p.173-198, Leipzig, 1931

[4] R.W.Hamming: the Unreasonable Effectiveness of Mathematics, *the American Mathematical Monthly* v.87 n.2, 1980 www.dartmouth.edu/~matc/MathDrama/reading/Hamming.html

[5] E.C.R.Hehner: Specifications, Programs, and Total Correctness, *Science of Computer Programming* v.34 p.191-205, 1999, www.cs.utoronto.ca/~hehner/SPTC.pdf

[6] E.C.R.Hehner: *a Practical Theory of Programming*, first edition Springer 1993, current edition www.cs.utoronto.ca/~hehner/aPToP

[7] K.Popper: *the Logic of Scientific Discovery*, Basic Books, 1959

[8] D.S.Scott, C.Strachey: Outline of a Mathematical Theory of Computation, *Proceedings of the fourth annual Princeton Conference on Information Sciences and Systems* p.169-176, 1970

[9] C.E.Shannon, a Mathematical Theory of Communication, *Bell System Technical Journal* v.27 p.379–423 & 623–656, 1948

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

[11] E.P.Wigner, the Unreasonable Effectiveness of Mathematics in the Natural Sciences, *Communications of Pure and Applied Mathematics* v.13, 1960, www.dartmouth.edu/~matc/MathDrama/reading/Wigner.html

[12] E.S.Yudkowski: *an Intuitive Explanation of Bayesian Reasoning*, http://yudkowsky.net/bayes/bayes.html, 2003

originally written in 1985
thoroughly revised in 2007
latest change 2009