Halting misconceived?

Bill Stoddart

August 25, 2017

Abstract

The halting problem is considered to be an essential part of the theoretical background to computing. That halting is not in general computable has been "proved" in many text books and taught on many computer science courses, and is supposed to illustrate the limits of computation. However, there is a dissenting view that these proofs are misconceived. In this paper we look at what is perhaps the simplest such proof, based on a program that interrogates its own halting behaviour and then decides to thwart it. This leads to a contradiction that is generally held to show that a halting function cannot be implemented. The dissenting view agrees with the conclusion that the halting function, as described, cannot be implemented, but suggests that this is because its specification is inconsistent. Our paper uses Forth to illustrate complex abstract arguments.

Keywords: Forth, halting problem, proof

1 Introduction

In his invited paper [2] at The First International Conference on Unifying Theories of Programming, Eric Hehner dedicates a section to the proof of the halting problem, claiming that it entails an unstated assumption. He agrees that the halt test program cannot exist, but concludes that this is due to an inconsistency in its specification. Hehner has republished his arguments using less formal notation in [3].

The halting problem is considered to be an essential part of the theoretical background to computing. That halting is not in general computable has

been "proved" in many text books and taught on many computer science courses to illustrate the limits of computation. Hehner's claim is therefore extraordinary. Nevertheless he is an eminent computer scientist[1] whose opinions reward careful attention. In judging Hehner's thesis we will take the view that to illustrate the limits of computation we need a program which can be consistently specified, but not implemented.

In this paper our aims are to examine Hehner's arguments by expressing them in Forth. A secondary aim is to show the suitability of Forth for performing such an analysis.

The halting problem is typically stated as follows. Given a Turing machine equivalent (TME) language there is no halt test program $H(P, X)$ which will tell us, for arbitrary program P and data X, whether P will halt when applied to X.

Hehner simplifies this, saying there is no need to consider a program applied to data, as data passed to a program could always be incorporated within the program. So his version is that there is no halt test $H(P)$ which tells us, for an arbitrary program P, whether execution of P will halt.

To express the halting proof in Forth, we assume we have implemented a program H with stack effect (xt $--$ f) which, for any token xt, reports whether execution of xt will halt.[2] We write $'P$ to represent the token for program P.

Were H to exist, we could use it as follows:

```
: Skip ;   : Loop BEGIN AGAIN ;
'Skip H . ↲ -1 ok
'Loop H . ↲ 0 ok
```

[1] In the area of programming semantics, Hehner was the first to propose "programs as predicates", an approach later adopted in Hoare and He's work on unifying theories. He was the first person to express the semantics of selection and iteration in terms of two simple semantic primitives, choice and guard, an approach now generally adopted and used, for example, in Abrial's B-Method. He has proposed a reformulation of set theory that supports unpacked collections and gives semantic meaning to the contents of a set, which is referred to as a bunch, and has properties perfect for representing non-determinism. His other contributions have been in areas as diverse as quantum computing and the semantics of OO languages. When the book "Beauty is our business" was conceived as a tribute to the work of E W Dijkstra, Hehner contributed a chapter discussing Gödel's incompleteness theorem. His book *A Practical Theory of Programming* [1] is available online in updated form.

[2] Our tokens are abstract analogies of Forth execution tokens, freed from any finiteness constraints.

When a Forth program is executed from the keyboard it either comes back with an "ok" response, or exhibits some pathological behaviour such as reporting an error, not responding because it is in an infinite loop, or crashing the whole system. We classify the "ok" response as what we mean by "halting".

The proof that H cannot be implemented goes as follows. Under the assumption that we have implemented H, we ask whether the following program will halt:[3]

```
: S 'S H IF Loop THEN ;
```

Now within S, H must halt and leave either a true or false judgement for the halting of S. If it leaves a true flag (judging that S will halt) then S will enter a non-terminating Loop. If it leaves a false flag (judging that S will not halt), then S will immediately halt.

Since H cannot pass a correct judgement for S, we must withdraw our assumption that there is an implementation of H. Thus halting behaviour cannot, in general, be computed. □

Hehner asks us to look in more detail at the specification of H. Since it must report on the halting behaviour of any program, it must assume the objective existence of such a behaviour. But S contains a "twisted self reference" and the halting behaviour of S is altered by passing judgement on it. H does not have a consistent specification. It cannot be implemented, but this has little significance, as it is due to the inconsistency of its specification. When someone claims a universal halt test is uncomputable, and you reply, "What do you mean by a universal halt test?" you won't receive a mathematically consistent answer.

From a programming perspective, we can add that S looks as if it will NOT terminate, because when `'S H` is executed, it will be faced with again commencing execution of `'S H`, and with no additional information to help it. S will not terminate, but this is because the halt test invoked within it cannot terminate.

The paper is structured as follows. In section 2 we verify Hehner's simplification of the halting problem. In section 3 we make some general remarks on halting, finite memory computations, and connections between halt tests and mathematical proofs. In section 4 we present a tiny language consisting of

[3]The program accesses its own token. Later, in some code experiments, we show how this is achieved.

three Forth programs with access to a halt test. We find we can still use the same proof, that a halt test cannot be implemented. We examine the specification of the halt test for this minimal scenario in detail, and we produce a Forth implementation of an amended halt test that is allowed to report non-halting either by the return of a stack argument or by an error message. In section 5 we perform a semantic analysis of S, taking its definition as a recursive equation, and conclude that its defining equation has no solution. S does not exist as a conceptual object, and neither does H.

The halting problem is generally attributed to Turing's paper on Computable Numbers [6], but this attribution is misleading. In an appendix we briefly describe Turing's paper and how the halting problem emerged from it. We also give an example of uncomputability which has a consistent specification.

The original aspects of this paper are: a careful examination of Hehner's arguments by re-expressing them in Forth; a translation of the halting problem proof to a minimal language where exactly the same argument can be made; an examination of the consistency of the halt test specification for our minimal language with an extension of this argument to the general case; an implementation of a less strict halt test for the minimal language which allows a result to be computed except in the self referential case, where non-halting is reported as an error; a semantic analysis of S and H as a conceptual objects; and a critique of the response to Hehner's 2006 paper [2] presented in *Halting still standing* [5].

2 Hehner's simplification of the halting problem

Normally the halting problem is discussed in terms of a halt test taking data D and program P and reporting whether P halts when applied to D.

Hehner's simplified halt test takes a program P and reports whether it halts.

We refer to the first of these halt tests as H_2, since it takes two arguments, and the second as H.

To verify Hehner's simplification of the halting problem we show that any test that can be performed by H_2 can also be performed by H, and any test that can be performed by H can also be performed by H_2.

Proof Given P_0 ($--$?), P_1 (x $--$?) and D ($--$ x), where P_0, P_1 are

arbitrary Forth definitions with the given signatures and D is arbitrary Forth code that returns a single stack value, and assuming tests H (xt $--$ f) and H_2 (x xt $--$ f) where $'P_0$ H reports whether P_0 halts, and D $'P_1$ H_2 reports whether D P_1 halts, then:

The test D $'P_1$ H_2 can be performed by H with the aid of the definition
: T D P_1 ; as $'T$ H.

The test $'P_0$ H can be performed by H_2 with the aid of the definition
: U $DROP$ P_0 ; as D $'U$ H_2. □

3 Some notes on halting analysis

Fermat's last theorem states that for any integer $n > 2$ there are no integers a, b, c such that:

$$a^n + b^n = c^n$$

Fermat died leaving a note in a copy of Diophantus's *Arithmetica* saying he had found a truly marvellous proof of his theorem, but it was too long to write in the margin. No proof was never found. All subsequent attempts failed until 1995, when Andrew Wiles produced a proof 150 pages long.

However, given a program *FERMAT* which searches exhaustively for a counter example and halts when it finds one, and a halt test H we could have proved the theorem by the execution:

$'FERMAT$ H . ↲ 0 *ok*

This would tell us the program *FERMAT* does not halt, implying that the search for a counter example will continue forever, in other words that no counter example exists and the theorem is therefore true.

In the same way we could explore many mathematical conjectures by writing a program to search exhaustively over the variables of the conjecture for a counter example. Then use H to determine whether the program fails to halt, in which case there is no counter example, and the conjecture is proved.

3.1 Known, bounded, and unbounded memory requirements

If a program has a known memory requirement of n bits its state transitions can take it to at most 2^n different states. We can solve the halting problem by running it in a memory space of $2n$ bits and using the additional n bits as a counter. When we have counted 2^n state transitions and the program has not halted, we know it will never halt because it must have, at some point, revisited a previous state.

The postulated *FERMAT* program above has an *unbounded* memory requirement, since as it performs its exhaustive search for a counter example it will need to work with larger and larger integers.

Now consider the Goldbach conjecture, which states that every even integer can be expressed as the sum of two primes (we include 1 in the prime numbers). This is an unproved conjecture, but so far no counter example has been found, although it has been checked for all numbers up to and somewhat beyond 10^{18}.

Now suppose we have a program *GOLDBACH* which performs an exhaustive search for a counter example to Goldbach's conjecture and halts when it finds one. If Goldbach's conjecture is true, this program has unbounded memory requirements. If the conjecture is false, it has bounded memory requirements, but the bound is unknown.

When Turing formulated his Turing machines he gave them an unbounded memory resource in the form of infinite tapes. This allows a Turing machine to be formulated which will perform an unbounded calculation, such as calculating the value of π. Although we cannot ever complete the calculation, we can complete it to any required degree of accuracy, and the existence of an effective procedure for calculating π gives us a finite representation of its value.

A Turing machine consists of a finite state machine (FSM) plus an infinite tape. To be TME a language needs to be powerful enough to program a FSM, and needs to be idealised to the extent of having an infinite memory resource corresponding to the tape of the Turing machine. Providing Forth with a pair of infinite stacks is sufficient to simulate an infinite tape.

Unbounded memory resources are important for the discussion of halting in this section, but they do not play a part in our discussion of the halting proof.

4 Halting in a trivial language

The conventional view of the halting problem proof is that it shows a universal halt test is impossible in a TME language. We have also seen that failure to halt can be detected in programs with known memory requirements, because after a known number of transitions such programs are bound to have revisited a previous state, which tells us they will never terminate.

It is rather strange, therefore, that we can apply the halting program proof to a minimal language whose only programs, in semantic terms, are one that terminates and one that does not.

Consider a language $\mathcal{L}_0$ consisting of two words, *Skip* and *Loop*.

This is a stateless language for which we can specify and implement a halt test H_0. The specification is consistent because it has a model:

$$\{\ 'Skip \ \mapsto \ true\ ,\ \ 'Loop \ \mapsto \ false\ \}$$

Now we become ambitious and wish to consider a more complex language $\mathcal{L}_1$ which consists of three words, *Skip*, *Loop*, and S, with a halt test H.

Our definition of S is still:

$$:\ S\ \ 'S\ \ H\ IF\ \ Loop\ \ THEN\ \ ;$$

and note that, were S to exist it will either behave like *Skip* or *Loop*,

and our specification for H is:

H (xt $--$ f) Where xt is the execution token of *Skip*, *Loop*, *or* S, return a flag that is true if and only if execution of xt halts.

But what is the model for H ?

$$\{\ 'Skip \ \mapsto \ true\ ,\ \ 'Loop \ \mapsto \ false,\ \ 'S \ \mapsto ?\ \ \}$$

Our model must map $'S$ to either true or false, but whichever is chosen will be wrong. We have no model for H, so it cannot have a consistent specification.

We have reduced the halting scenario to a minimal language so we can write out the model of halting, but exactly the same argument applies to halting in a TME language.

In this minimal scenario of a state free language we can make the same "proof" that halting is uncomputable that we used for TME languages in the introduction. Yet we have seen that for programs with known memory requirements halting can be verified by monitoring execution of the program until it terminates or has performed enough steps for us to know that it will not halt. Of course the question being answered by the proof, on the one hand, and the monitoring of execution, on the other, are not the same. Monitoring execution does not require a "twisted self reference". There is a separation between the monitor, as observer, and the executing program, as the thing observed.

4.1 Experiments with code

We have already noted in the introduction that S looks as if it will NOT terminate, because when $'S\ H$ is executed, it will be faced with again commencing execution of $'S\ H$ with no additional information to help it. S will not terminate, but this is because the halt test invoked within it cannot terminate.

There is no reason, however, why a halt test cannot terminate in other situations, or why failure to halt cannot be reported via an error message when the halt test itself cannot halt.

Here is a specification of a slightly different halting test.

$H_1\ (\ xt\ --\ f\ )$, Return a true flag if execution of xt halts. If execution of xt does not halt return a false flag, unless that failure to halt is due to non-termination within H_1, in which case report an error.

We define : $S_1\ 'S_1\ H_1\ IF\ Loop\ THEN\ ;$

Here is the error report when S_1 is invoked.

$S_1\ \hookleftarrow$
Error at S_1
Cannot terminate
reported at H_1 *in file ...*

And here is the interaction when halt tests are invoked directly from the keyboard.

$'Loop\ H_1\ .\ \hookleftarrow 0\ ok$
$'Skip\ H_1\ .\ \hookleftarrow -1\ ok$
$'S_1\ H_1\ .\ \hookleftarrow 0\ ok$

Implementation requires H_1 to know when it is being invoked within S_1. This information is present in the run time system, and we obtain it from the word S_1X "S_1 executing" which, when used in H_1, will return true if and only if H_1 has been invoked by S_1.

```
0 VALUE 'Skip  0 VALUE 'Loop  0 VALUE 'S1  0 VALUE 'H1

: S1X ( -- f, true if S1 is executing, implementation specific code )
  R> R@ SWAP >R 'S1 - 16 = ;

: Skip ;   : Loop BEGIN AGAIN ;

: H1 ( xt -- f, , If H1 has been invoked within xt and cannot terminate
without compromising the termination behaviour of xt, report a
Cannot terminate error. Otherwise return the halting behaviour of xt)
  CASE
   'Skip OF TRUE ENDOF
   'Loop OF FALSE ENDOF
   'S1 OF S1X ABORT" Cannot terminate" FALSE ENDOF
    DROP
  ENDCASE ;

: S1 ( -- ) 'S1 H1 IF Loop THEN ;

' Skip to 'Skip  ' Loop to 'Loop  ' S1 to 'S1  ' H1 to 'H1
```

This illustrates that the problem is not that halting of S_1 cannot be computed, but that the result cannot always be communicated in the specified way. Requiring H (or in this case H_1) to halt in all cases is too strong, as it may be the halt test itself that cannot halt. We may, however, require that the halt test should always halt when not invoked recursively within S_1.

5 Proof and paradox

In [2] the halting problem is compared to the Barber's paradox. "The barber, who is a man, shaves all and only the men in the village who do not shave themselves. Who shaves the barber?" If we assume he shaves himself, we see we must be wrong, because the barber shaves only men who do not shave themselves. If we assume he does not shave himself, we again see we must be wrong, because the barber shaves all men who do not shave themselves. The statement of the paradox seems to tell us something about the village, but it does not, since conceptually no such village can exist.

In a similar way, the program S which we have used in the halting problem proof, does not exist as a conceptual object[4] so what we say about it can be paradoxical.

To prove this we need a rule for the termination of the form $g\ IF\ T\ THEN$ under the assumption that computation of g terminates. To formulate the rule we need a mixture of Forth notation and formal logic notation: where the Forth program P has stack effect $--\ x$ we use $\lceil P \rceil$ to represent the value of x in our formal logic. We use $trm(T)$ for the predicate which is true if and only if T will terminate.

Now we can state our rule as:[5]

$$trm(g)\ \Rightarrow\ (\ trm(g\ IF\ T\ THEN)\ \Leftrightarrow\ \ (\neg \lceil g \rceil\ \vee\ (\lceil g \rceil\ \Rightarrow\ trm(T))\ )\ (1)$$

And we can state the specification of $'P\ H$ as:

$$\lceil 'P\ H \rceil\ \Leftrightarrow\ trm(P)$$

Bearing in mind that $trm(H)$ is true by the specification of H, we argue:

$trm(S)\ \Leftrightarrow$ by definition of S
$trm(\ 'S\ H\ IF\ Loop\ THEN\ )\ \Leftrightarrow$ by rule (1) above
$\neg \lceil 'S\ H \rceil\ \vee\ (\ \lceil 'S\ H \rceil\ \Rightarrow\ trm(Loop)\ \Leftrightarrow$ property of $Loop$
$\neg \lceil 'S\ H \rceil\ \vee\ (\ \lceil 'S\ H \rceil\ \Rightarrow\ false\ )\ \Leftrightarrow$ logic
$\neg \lceil 'S\ H \rceil\ \vee\ \neg \lceil 'S\ H \rceil\ \Leftrightarrow$ logic
$\neg \lceil 'S\ H \rceil\ \Leftrightarrow$ specification of H
$\neg\ trm(S)$

So we have proved that $trm(S)\ \Leftrightarrow\ \neg\ trm(S)$. This tells us that S does not exist as a conceptual object, let alone as a program. We have seen in the previous section that by relaxing the specification of H we can implement the same textual definition of S, so the non existence of S proved here can only be due to the specification of H being inconsistent.

The proof of the halting problem assumes a universal halt test exists and then provides S as an example of a program that the test cannot handle. But S is not a program at all. It is not even a conceptual object, and this

[4]Examples of conceptual objects include numbers, sets, predicates, and programs. Suppose we have P, which is supposed to be a predicate, but is claimed to have the property $P\ \Leftrightarrow\ \neg\ P$. No such predicate exists: P is not a conceptual object. $P\ \Leftrightarrow\ \neg\ P$ reduces to false, from which we can prove anything, including paradoxical properties.

[5]A referee queried whether we need to refer to a formal semantics of recursion. Such a semantics would suggest that S might not exist as a conceptual object [5]. However, since the only semantic question concerns termination, we can show it does not exist with a simple direct approach.

is due to inconsistencies in the specification of the halting function. H also doesn't exist as a conceptual object, and we have already seen this from a previous argument where we show it has no model.

A response to Hehner's Unifying Theories paper was given by by Verhoeff *et al* [5]. This paper, like [2], frames its arguments in the specialist notation of [4]. They note that Hehner's proof that the specification of S does not define a conceptual object is based on an analysis of the definition $S = \neg\ ok' \lhd\ H(S)\ \rhd\ ok'$. This just says S halts if H says it doesn't and *vice versa*. But Hehner defines $S =$ "$\neg\ ok' \lhd H(S) \rhd ok'$ ", i.e. S is a string, and this is what is passed to H. This can be fixed with a notational adjustment. Their second point is that whilst specifications, which are just mathematics, may not define conceptual objects (not all equations have solutions) the same is not true of programs. Code always defines a semantic object. However we find that, under the assumption that H has been implemented, we don't have the right mathematical conditions for the implementation of S to have a solution, and this is enough to establish the contradiction. This point caused Hehner to change his rhetoric slightly – his point is not that the proof does not confirm the non-existence of a universal halt test, but rather that a universal halt test does not exist conceptually, so we can't expect to implement it.

Our notion of an uncomputable specification requires the specification to have a model, but no implementation. An example is provided by Turing's uncomputable sequence β, discussed briefly in the appendix.

6 Conclusions

The halting problem is universally used in university courses on Computer Science to illustrate the limits of computation. Hehner claims the halting problem is misconceived. Presented with a claim that a universal halt test cannot be implemented we might ask – what is the specification of this test that cannot be implemented? The informal answer, that there is no program H which can be used to test the halting behaviour of an arbitrary program, cannot be formalised as a consistent specification.

The program S, used as example of a program whose halting cannot be analysed, observes its own halting behaviour and does the opposite. Hehner calls this a "twisted self reference". It violates the key scientific principle of, where possible, keeping what we are observing free from the effects of the

observation.

To better understand Hehner's thesis we have re-expressed his argument using Forth as our programming language. We have verified Hehner's simplification of the problem, and proposed a minimal language of three programs and a halt test, to which exactly the same proof can be applied.

Our programming intuition tells us that S will not terminate because when $'S\ \ H$ is invoked within S, H will not terminate. However, we cannot require H to return a value to report this, because that would require it to terminate! We provide a programming example based on a minimal language where we resolve this by allowing the option for a halt test to report via an error message when it finds itself in this situation. However, we can require that the halt test should always halt other situations. The problem is not the uncomputability of halting!

We have also performed semantic analysis using Forth. This analysis confirms that the halt test and S *do not exist as conceptual objects.*

We have found nothing to make us disagree with Hehner's analysis. Defenders of the status quo might say – so the halt test can't even be conceived, so it doesn't exist. What's the difference? Hehner says that uncomputability requires a *consistent* specification that cannot be implemented. Turing's uncomputable sequence β can provide such an example. A computation that inputs n and outputs $\beta(n)$ has a model, since β is mathematically well defined, but if we could compute it for arbitrary n, then β would be a computable sequence. The uncomputability of β is proved in the appendix.

Forth has been invaluable in this work in providing a concise notation, and in helping us combine programming intuition with abstract arguments. We have used it to transfer the argument to the scenario of a minimal language, where the proof still holds, and to play with a variation of the halt test that demonstrates that he problem in the scenarios we examine is not the uncomputabilty of halting.

Acknowledgements.

Thanks to Ric Hehner for extensive electronic conversations; Steve Dunne for extended discussions; participants at EuroForth 2016 for the stimulating comments and questions in response to my talk "The halting problem in Forth"; to the referees and Ric Hehner for their corrections of, and interesting comments on, a draft paper; also to Campbell Ritchie for proof reading the final version.

References

[1] E C R Hehner. *A Practical Theory of Programming.* Springer Verlag, 1993. Latest version available on-line.

[2] E C R Hehner. Retrospective and Prospective for Unifying Theories of Programming. In S E Dunne and W Stoddart, editors, *UTP2006 The First International Symposium on Unifying Theories of Programming*, number 4010 in Lecture Notes in Computer Science, 2006.

[3] E C R Hehner. Problems with the halting problem. *Advances in Computer Science and Engineering*, 10(1):31–60, 2013.

[4] C A R Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice Hall, 1998.

[5] Cornelis Huizing, Ruurd Kuiper, and Tom Verhoeff. *Halting Still Standing – Programs versus Specifications*, pages 226–233. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[6] Alan M Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

Appendix: Turing's 1936 paper and the halting problem

On computable numbers, with a contribution to the Entscheidungsproblem, Turing's paper from 1936 [6] is cited as the source of the halting problem, but it does not mention halting. The paper captures Hilbert's notion of an "effective procedure" by defining "computing machines", consisting of finite state machines with an infinite tape, which are similar to what we now call Turing machines but with significant differences. He uses such machines to define all numbers with a finite representation as "computable numbers", with the fractional part of such a number being represented by a machine that computes an infinite binary sequence. The description of these machines is finite, so numbers such as π, which are computable to any desired accuracy, can have a finite representation in terms of the machines that compute them.

Turing's idea of a computer calculating π would perhaps have been of a human being at a desk, performing the calculation, and now and then writing down another significant figure. His "computing machines" are supposed to continue generating the bits of their computable sequence indefinitely,

but faulty machines may fail to do so, and these are not associated with computable sequences.

The computing machines that generate the computable sequences can be arranged in order. Turing orders them by an encoding method which yields a different number for each computing machine, but we can just as well think of them being lexigraphically ordered by their textual descriptions.

The computable sequences define binary fractions that can be computed. Turing's contribution to the Entscheidungsproblem is in *defining* a binary sequence β that *cannot be computed*. Let $M(n)$ be the *nth* computable sequence, and define the sequence:

$\beta(n) = \quad \textbf{if } M(n)(n) = 1 \textbf{ then } 0 \textbf{ else } 1 \textbf{ end}$.

By a diagonalisation argument β is not one of the computable sequences: it is definable but not computable. The link with halting comes from asking why it cannot be computed, the reason being that although we can talk about the sequence of computing machines that generate infinite binary sequences of 0's and 1's we cannot distinguish these from machines which have the correct syntactic properties but which do not generate infinite sequences. So we cannot compute which of the computable sequences is the nth computable sequence because we cannot distinguish good and bad computing machines.

The first reference to the "halting problem" I have been able to find comes in Martin Davis's book *Computability and Unsolvability*, from 1958. By then Turing machines had taken their current form and were required to halt before the output was read from their tape. He credits Turing's 1936 paper as the source of the problem's formulation.

A proof using a computing mechanism which enquires about its own halting behaviour and then does the opposite appears in Marvin Minsky, *Computation. Finite and infinite machines*, from 1967.