Halting According to aPToP

Eric C.R. Hehner

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

Introduction

In 1976, [0] suggested that computer programs could directly express their semantics, without the need for a semantic function. The suggestion was made at the end of the paper, and not acted upon in the paper. (Due to referee request, it was removed from the published version [1] of the paper.) In 1981, [2] did act upon that suggestion (and also in the published version [3]), and it became the basis for what was then called "Predicative Programming" [4][5]. More recently, the same basis was used in the 1993 book *a Practical Theory of Programming* [6], and in the current free online edition. We are using the initials of that book (aPToP) to designate this kind of specification and program semantics. The same basis was used in the 1994 paper "the Temporal Logic of Actions" [11], and in the 1998 book *Unifying Theories of Programming* [9].

We specify computation (computer behavior) using a binary (boolean) expression whose nonlocal (free) variables represent whatever is of interest about the computation. In this paper, the initial value x and final value x' of a single natural variable, and the initial (start) time t and final (end) time t' of the computation are of interest. If the computation is unending, the final time is ∞ .

A program is a specification that has been implemented so that a computer can execute it. For example, $x:= x+2$ is an assignment program (statement). If its execution takes time 1 , then

$$x:= x+2 \;\;=\;\; x'=x+2 \land t'=t+1$$

A specification s is called "implementable" if, for all initial states and times, there are final states and times to satisfy the specification, with nondecreasing time.

$$\forall x, t\cdot \exists x', t'\cdot s \land t'\ge t$$

Let s be a specification and let p be a program. Then

$$\forall x, t, x', t'\cdot s \Leftarrow p$$

says " s is implied by p ", " s is refined by p ", " s is implemented by p ". It means that execution of program p will satisfy specification s . If you implement an implementable specification, then the specification becomes a program. Recursion is allowed. For example,

$$x'=0 \land t'=t+x \;\;\Longleftarrow\;\; \textbf{if } x=0 \textbf{ then } ok \textbf{ else } x:= x-1.\; t:= t+1.\; x'=0 \land t'=t+x \textbf{ fi}$$

The specification $x'=0 \land t'=t+x$ is being implemented. Program ok is defined as

(0) $ok \;=\; x'=x \land t'=t$

which directs a computer to do nothing (the "empty" program). The assignment $t:= t+1$ is not an instruction to a computer; it is just accounting for the passage of time each loop iteration. The loop is formed by the recursive use of the specification $x'=0 \land t'=t+x$; refinement of the specification $x'=0 \land t'=t+x$ by a program makes $x'=0 \land t'=t+x$ a program, so it can be used in programs (even in the program refining it). (Proof of the refinement is easy, and is omitted.)

The above specifications and programs are "imperative", meaning they describe a change in state. We also need "functional" specifications and programs. Any expression of a mathematical function can serve as a functional specification. A functional program is a functional specification that has been implemented so that a computer can execute it.

Halting Proof

Let p be an imperative program. Execution of p, starting in state x at time t, terminates if and only if

(1) $\exists f\cdot \forall x', t'\cdot\ p \Rightarrow t' \le t + f\,x$

where f is a mathematical function that applies to the start state x, which is a natural. The result type of f could be the natural numbers or the real numbers; it must be both nonnegative and finite. Expression (1) is the mathematical halting function. It has nonlocal variables p (program), x (start state), and t (start time). Let's call it *halt*, and turn its nonlocal variables into explicit parameters.

(2) $halt\ p\ x\ t\ \ =\ \ \exists f\cdot \forall x', t'\cdot\ p \Rightarrow t' \le t + f\,x$

$halt\ p\ x\ t$ is $\top$ (binary "true") if execution of p, starting in state x at time t, terminates, and $\bot$ (binary "false") otherwise.

> Technical point: For programs to be parameters, they must be encoded as data. Turing encoded them as numbers [13]. He called the decoder that turns numbers into programs the Universal Machine. The modern practice is to encode programs as text (character string); that is how programs are passed to compilers. The decoder that turns text into program is called an interpreter. For example,
>
> $\mathbb{I}\ \text{“}x{:=}\,0\text{”}\ \ =\ \ x{:=}\,0$
>
> where $\mathbb{I}$ is the interpreter. A text encoding of a program is so transparent that it is easy to forget the difference between a program and its encoding as text. In the aPToP world, a program is a specification, and the type of an imperative specification is binary: it evaluates to $\top$ when a computation satisfies it, and to $\bot$ when a computation does not satisfy it. But in $halt\ p\ x\ t$, we are not passing a binary value p to *halt*; we are passing text. Thus we are abusing notation, relying on the context to make it clear, rather than cluttering our expressions with quotation marks and uses of $\mathbb{I}$.

Assume (in order to show a contradiction) that functional specification *halt* has been implemented, and is therefore a functional program. We now create an imperative program, let's call it *twist*, defined as

(3) $twist\ \ =\ \ \textbf{if}\ halt\ twist\ x\ t\ \textbf{then}\ loop\ \textbf{else}\ ok\ \textbf{fi}$

where *loop* is defined as

(4) $loop\ \ =\ \ t'{=}\infty$

and easily implemented as

$loop\ \ \Leftarrow\ \ t{:=}\,t{+}1.\ loop$

> Technical point: We assume there is a dictionary of definitions, including (0), (2), (3), and (4), that is accessible to *halt*, so that *halt* can look up *ok*, *halt*, *twist*, and *loop* in the dictionary, and retrieve their texts for analysis.

For any values of x and t, *halt twist x t* is either $\top$ or $\bot$. Suppose *halt twist* $0\ 0 = \top$. We calculate:

$$
\begin{array}{lll}
 & \top & \text{supposition} \\
= & \mathit{halt\ twist}\ 0\ 0 & (2) \\
= & \exists f\cdot\ \forall x', t'\cdot\ \mathit{twist} \Rightarrow t' \le 0{+}f\,0 & (3) \\
= & \exists f\cdot\ \forall x', t'\cdot\ \textbf{if}\ \mathit{halt\ twist}\ 0\ 0\ \textbf{then}\ \mathit{loop}\ \textbf{else}\ \mathit{ok}\ \textbf{fi} \Rightarrow t' \le f\,0 & \text{supposition} \\
= & \exists f\cdot\ \forall x', t'\cdot\ \textbf{if}\ \top\ \textbf{then}\ \mathit{loop}\ \textbf{else}\ \mathit{ok}\ \textbf{fi} \Rightarrow t' \le f\,0 & \text{simplify } \textbf{if} \\
= & \exists f\cdot\ \forall x', t'\cdot\ \mathit{loop} \Rightarrow t' \le f\,0 & (4) \\
= & \exists f\cdot\ \forall x', t'\cdot\ t'{=}\infty \Rightarrow t' \le f\,0 & \text{one-point law} \\
= & \exists f\cdot\ \infty \le f\,0 & \text{the result of } f \text{ is finite} \\
= & \bot &
\end{array}
$$

Now suppose *halt twist* $0\ 0 = \bot$. We calculate:

$$
\begin{array}{lll}
 & \bot & \text{supposition} \\
= & \mathit{halt\ twist}\ 0\ 0 & (2) \\
= & \exists f\cdot\ \forall x', t'\cdot\ \mathit{twist} \Rightarrow t' \le 0{+}f\,0 & (3) \\
= & \exists f\cdot\ \forall x', t'\cdot\ \textbf{if}\ \mathit{halt\ twist}\ 0\ 0\ \textbf{then}\ \mathit{loop}\ \textbf{else}\ \mathit{ok}\ \textbf{fi} \Rightarrow t' \le f\,0 & \text{supposition} \\
= & \exists f\cdot\ \forall x', t'\cdot\ \textbf{if}\ \bot\ \textbf{then}\ \mathit{loop}\ \textbf{else}\ \mathit{ok}\ \textbf{fi} \Rightarrow t' \le f\,0 & \text{simplify } \textbf{if} \\
= & \exists f\cdot\ \forall x', t'\cdot\ \mathit{ok} \Rightarrow t' \le f\,0 & (0) \\
= & \exists f\cdot\ \forall x', t'\cdot\ x'{=}0 \wedge t'{=}0 \Rightarrow t' \le f\,0 & \text{one-point law} \\
= & \exists f\cdot\ 0 \le f\,0 & \text{for } f \text{ use the constant } 0 \text{ function} \\
= & \top &
\end{array}
$$

We have a contradiction. Therefore the assumption that there is a program to implement the halting specification was wrong. This is the aPToP proof of the classic result by Turing [13].

Specification Version

In the proof of the previous section, we consider that there is a fixed set of programs for *halt* to apply to. If this set includes *halt* and *twist*, we have a contradiction.

The aPToP (and UTP [9] and TLA [11]) view of programs is different. aPToP was designed to aid programmers in writing programs, and it takes their view. If you write a program, there is now one more program than there was before you wrote it. Any implementable specification becomes a program when it is refined by a program. To be compatible with this view, we now generalize functional specification *halt* to apply to all imperative specifications.

(5) $\quad \mathit{halt}\ s\ x\ t \ = \ \exists f\cdot\ \forall x', t'\cdot\ s \Rightarrow t' \le t + f\,x$

where s is any imperative specification. Definition (5) is identical to definition (2), except that in (2) *halt* applied only to programs, and in (5) *halt* applies to all specifications (including programs). *halt s x t* is $\top$ if s specifies a computation that, starting in state x at time t, terminates, and $\bot$ otherwise.

If we assume that this generalized *halt* specification is implemented, then both *halt* and *twist* are programs, and the exact same calculations (previous section) arrive at the same contradiction.

If *halt* is not implemented, then both *halt* and *twist* are not programs, but they are nonetheless specifications. And the exact same calculations (previous section, except that hint (2) becomes hint (5)) arrive at the same contradiction. We arrive at this contradiction without having assumed that there is a program to implement the halting specification. So we have no assumption to blame for the contradiction. The conclusion is that definitions (5) and (3) (*halt* and *twist*) together are inconsistent. That is the conclusion whether or not *halt* and *twist* are programs. The inconsistency neither appears nor disappears by changing the domain of *halt* from programs to all specifications. For a similar conclusion, see [12]. For a contrary conclusion, see [10].

Discussion

What Turing proved, and what many textbooks prove, and what we proved in this paper in aPToP style, is

A. The definition of *halt* as a program applying to program *twist* is inconsistent.

From that, we might conclude

B. The definition of *halt* as a program applying to all programs is inconsistent.

If a consistent specification cannot be implemented in a Turing-Machine-Equivalent (TME) programming language, we say it is “undecidable”. The specification must be consistent so that it does specify something, and it must be something a TME programming language is not expressive enough to implement. An inconsistent specification cannot be implemented in any consistent language, no matter how powerful or expressive, but that doesn't merit calling it “undecidable”. If it did, we could just use $\perp$ as an example of an undecidable specification. To say that halting is “undecidable”, we also need

C. The definition of *halt* as an unimplemented specification applying to all programs is consistent.

C is not proven, but it is commonly believed for the weak reason that no inconsistency is apparent. B and C are necessary to call halting “undecidable”, but they are not sufficient. The problem lies in the words “all programs” in B and C.

The aPToP approach begins with specifications. We prove

D. The definition of *halt* as a specification applying to all specifications is inconsistent.

by exactly the same proof that we used to prove A, except that *halt* and *twist* are specifications (they may or may not be programs), and *halt* applies to all specifications (including but not limited to programs).

In the aPToP view, programs are a growing subset of specifications; when an implementable specification is refined by a program, that specification becomes a program. So we might limit our attention to implementable specifications, which are the potential programs. In definition (3), if $halt\ twist\ x\ t = \top$, then $twist = loop$, so $twist$ is implementable. And if $halt\ twist\ x\ t = \bot$, then $twist = ok$, so again $twist$ is implementable. Therefore $twist$ is classically implementable. But since we cannot determine whether $halt\ twist\ x\ t$ is $\top$ or $\bot$, $twist$ is not constructively implementable. Classically, we can conclude

E. The definition of *halt* as an unimplemented specification applying to implementable specifications is inconsistent.

Contrast C with E: changing "programs" to "implementable specifications" changes "consistent" to "inconsistent".

If we reduce the domain of *halt* to just the two programs *loop* and *ok* , *halt* is easily implemented. So

F. The definition of *halt* as a program applying to the programs *loop* and *ok* is consistent.

To talk about "all programs", we need to have a syntactically defined programming language, and then we can talk about all programs in that language. We shall say "L-program" for a program written in TME programming language L. Conclusion B becomes

G. The definition of *halt* as an L-program applying to all L-programs is inconsistent.

That's because, if *halt* is an L-program, then there is an L-program just like *twist* . So we dare to suggest

H. The definition of *halt* as an L-program is consistent if we restrict its domain to those L-programs that do not refer to *halt* , neither directly nor indirectly through other L-programs.

This is not the largest class for which a *halt* program exists; there are L-programs that refer to *halt* in a benign fashion, and could be included without inconsistency. Consider

(6) $\quad straight\ =\ \textbf{if}\ halt\ straight\ x\ t\ \textbf{then}\ ok\ \textbf{else}\ loop\ \textbf{fi}$

For any values of x and t , $halt\ straight\ x\ t$ is either $\top$ or $\bot$. Suppose $halt\ straight\ 0\ 0 = \top$. We calculate:

	$\top$	supposition
$=$	$halt\ straight\ 0\ 0$	(2) or (5)
$=$	$\exists f\cdot\ \forall x', t'\cdot\ straight\ \Rightarrow\ t' \le 0+f\,0$	(6)
$=$	$\exists f\cdot\ \forall x', t'\cdot\ \textbf{if}\ halt\ straight\ 0\ 0\ \textbf{then}\ ok\ \textbf{else}\ loop\ \textbf{fi}\ \Rightarrow\ t' \le f\,0$	supposition
$=$	$\exists f\cdot\ \forall x', t'\cdot\ \textbf{if}\ \top\ \textbf{then}\ ok\ \textbf{else}\ loop\ \textbf{fi}\ \Rightarrow\ t' \le f\,0$	simplify **if**
$=$	$\exists f\cdot\ \forall x', t'\cdot\ ok\ \Rightarrow\ t' \le f\,0$	(4)
$=$	$\exists f\cdot\ \forall x', t'\cdot\ x'=0 \land t'=0\ \Rightarrow\ t' \le f\,0$	one-point law
$=$	$\exists f\cdot\ 0 \le f\,0$	for f use the constant 0 function
$=$	$\top$	

Now suppose $halt\ straight\ 0\ 0\ =\ \bot$. We calculate:

	$\bot$	supposition
$=$	$halt\ straight\ 0\ 0$	(2) or (5)
$=$	$\exists f\cdot\ \forall x', t'\cdot\ straight\ \Rightarrow\ t' \le 0{+}f\,0$	(6)
$=$	$\exists f\cdot\ \forall x', t'\cdot\ \textbf{if}\ halt\ straight\ 0\ 0\ \textbf{then}\ ok\ \textbf{else}\ loop\ \textbf{fi}\ \Rightarrow\ t' \le f\,0$	supposition
$=$	$\exists f\cdot\ \forall x', t'\cdot\ \textbf{if}\ \bot\ \textbf{then}\ ok\ \textbf{else}\ loop\ \textbf{fi}\ \Rightarrow\ t' \le f\,0$	simplify **if**
$=$	$\exists f\cdot\ \forall x', t'\cdot\ loop\ \Rightarrow\ t' \le f\,0$	(0)
$=$	$\exists f\cdot\ \forall x', t'\cdot\ t'{=}\infty\ \Rightarrow\ t' \le f\,0$	one-point law
$=$	$\exists f\cdot\ \infty \le f\,0$	the result of f is finite
$=$	$\bot$	

We can suppose *halt straight* 0 0 is $\top$ or $\bot$ without any inconsistency. Therefore a *halt* program, when applied to program *straight* , can return either $\top$ or $\bot$.

In [7] and [8] the suggestion is made to define *halt* to work on all L-programs, and to write an implementation of *halt* in TME language M, where L-programs cannot call M-programs. This disallows the inconsistency proof, while at the same time defining and implementing *halt* to apply to all programs in a TME language. If we can do that, then we cannot call halting "undecidable".

The obvious objection to this suggestion is that if we could program *halt* in language M to apply to all L-programs, we could translate it into language L; but there is no L-program to implement *halt* for all L-programs, so there cannot be an M-program to do so either. The objection is wrong: even though *halt* written in language M does compute halting for all L-programs, its translation to an L-program does not compute halting for all L-programs. This surprising result is the subject of [8].

Conclusion

For Turing, and for most of the programming world today, even for most of the formal methods community, programs and specifications are different things. They have separate languages in which they are expressed. Programs are instructions to a computer, not mathematical expressions. Formal specifications are mathematical expressions, not instructions to a computer. Specifications are used to reason about programs.

To the majority of people, the Halting Problem is about programs, not about specifications. The computability assumption is essential; we have to assume the existence of a program to compute halting so that we can reason about it. An inconsistency is created by the assumption that such a program exists. The reasoning that there is an inconsistency uses specifications, but, to the majority of people, the specifications are talking about programs.

In the aPToP world [6], programs are a growing subset of specifications. We do not specify programs; we specify computation, or computer behavior. Programs are those specifications that have been implemented so that they can be executed by a computer. If we prove something about all specifications, that includes all programs. If we find an inconsistency in a collection of specifications, that finding is unaffected by an assumption that one or more of the specifications are programs.

Maybe a revision of the Halting Problem will have to wait (possibly forever) until the aPToP view of programming is accepted.

References

[0] E.C.R.Hehner: "do considered od: a Contribution to the Programming Calculus", CSRG-75, U. of Toronto, 1976 November

[1] E.C.R.Hehner: "do considered od: a Contribution to the Programming Calculus", *Acta Informatica*, v.11 p.287-304, 1979

[2] E.C.R.Hehner, C.A.R.Hoare: "Another Look at Communicating Processes", CSRG-134, U. of Toronto, 1981 September

[3] E.C.R.Hehner, C.A.R.Hoare: "a More Complete Model of Communicating Processes", *Theoretical Computer Science*, v.26 p.105-120, 1983 September

[4] E.C.R.Hehner: "Predicative Programming, Part I", *Communications of the ACM*, v.27 n.2 p.134-143, 1984 February

[5] E.C.R.Hehner: "Predicative Programming, Part II", *Communications of the ACM*, v.27 n.2 p.144-151, 1984 February

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

[7] E.C.R.Hehner: "How to Compute Halting", 2014 January 2

[8] E.C.R.Hehner: "Objective and Subjective Specifications", 2017 July 10, WST Workshop on Termination, Oxford, 2018 July 18

[9] C.A.R.Hoare, J.He: *Unifying Theories of Programming*, Prentice-Hall, 1998

[10] C.Huizing, R.Kuiper, T.Verhoeff: "No Holes in Halting – Programs versus Specifications", International Symposium on Unifying Theories of Programming, Springer LNCS v.6445 p.226-233, Shanghai, 2010 November 15-16

[11] L.Lamport: "the Temporal Logic of Actions", *ACM Transactions on Programming Languages and Systems*, v.16 n.3 p.872-923, 1994 May

[12] W.Stoddart: "the Halting Paradox", *FACS FACTS: the Newsletter of the Formal Aspects of Computing Science Specialist Group*, 2018 January

[13] 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

other papers on halting

Appendix

The aPToP specification of termination is $t' \le t + f\, x$ where f is a nonnegative and finite function. It may seem that $t'<\infty$ expresses termination more simply. But $t'<\infty$ is an unimplementable specification. That's because programs, and more generally specifications, can be composed sequentially. If the first program in a sequential composition is an infinite loop, then its execution ends at time ∞ (in other words, it never ends), and so execution of the second program in the sequential composition starts at time ∞ (in other words, it never starts). Since any program can be placed following an infinite loop, a specification cannot promise that execution will end before time ∞ (at a finite time). But it can promise that execution will take a finite amount of time.

An implementable candidate to express termination is $t{<}\infty \Rightarrow t'{<}\infty$; if the computation starts at a finite time, then it ends at a finite time. But, surprisingly, this specification can be implemented by an infinite loop.

$$t{<}\infty \Rightarrow t'{<}\infty \quad \Leftarrow \quad t{:=}\ t{+}1.\ \ t{<}\infty \Rightarrow t'{<}\infty$$

In order for a user of this program to complain that its execution violates the specification, the user must observe computer behavior contrary to the specification. The user must observe that the computation starts at a finite time (easy), and ends at time infinity (it never ends). All observations are made at finite times; the user can never complain that the computation has taken forever. So this candidate fails to express termination.

The specification $t' \le t + f\,x$ for any given nonnegative finite function f does express termination, and furthermore, gives a time bound for termination. If execution takes longer than $f\,x$, the user can complain. The specification $t'{=}\infty$ does express nontermination; if execution ends at a finite time, the user can complain. For further details, see [6 p.50-51].