Termination Analyzer H is Not Fooled by Pathological Input D

Peter Olcott

21 January 2024

Abstract
A pair of C functions are defined such that D has the halting problem proof's pathological relationship to simulating termination analyzer H. When it is understood that D correctly simulated by H (a) Is the behavior that D specifies to H and (b) Cannot possibly halt then it is understood that H is correct to abort its simulation and reject D as non-halting. This exact same reasoning is shown to equally apply to the Linz Turing machine based halting problem proof.

Overview

> For any program H that might determine whether programs halt, a "pathological" program D, called with some input, can pass its own source and its input to H and then specifically do the opposite of what H predicts D will do. **No H can exist that handles this case.** https://en.wikipedia.org/wiki/Halting_problem

The x86utm operating system: https://github.com/plolcott/x86utm enables one C function to execute another C function in debug step mode. Termination analyzer H simulates the x86 machine code of its input (using libx86emu) in debug step mode until it correctly determines that its input never halts.

Can D correctly simulated by H terminate normally?

```
01 int D(ptr x)  // ptr is pointer to int function
02 {
03   int Halt_Status = H(x, x);
04   if (Halt_Status)
05     HERE: goto HERE;
06   return Halt_Status;
07 }
08
09 void main()
10 {
11   H(D,D);
12 }
```

Execution Trace
Line 11: main() invokes H(D,D);

keeps repeating (unless aborted)
Line 03: simulated D(D) invokes simulated H(D,D) that simulates D(D)

Simulation invariant:
D correctly simulated by H cannot possibly reach past its own line 03.

D correctly simulated by H cannot possibly reach its simulated final state in 1 to ∞ steps of correct simulation.

Because the directly executed D(D) specifies that it calls H(D,D) in recursive simulation and D(D) itself would never stop running unless H aborts this recursive simulation we know that D(D) never actually terminates normally and halts even though it really looks like it does.

Although I independently derived the notion of a simulating halt decider myself March 14, 2017 at 9:05 AM (CDT) USENET Message-ID:
<e18ff0a9-7f9d-4799-9d13-55d021afaa82@googlegroups.com>
along with two different versions of correct halt status criteria:

Professor Stoddart referenced one element of this criteria and Professor Hehner referenced two of the halt status criteria that H currently uses. H also verifies that the call from D to itself was unconditional.

> Implementation of H1 requires it to determine whether it is being invoked from within S1. In a typical compiled sequential language this information can be deduced from the return address for the call to H, and the symbol table, which contains information that will tell us whether this return address is within the code body of S1. However, this information is not usually directly accessible in the language, so **we will suppose we have written at assembly code level a test In S1 which will report whether the operation that invokes it has been invoked from within S1.** (Stoddart:2017)

References one element of the criteria that H uses to determine that D is calling H in recursive simulation. When D calls H(D,D), H sees that D is calling its own machine address.

> **From a programmer's point of view, if we apply an interpreter to a program text that includes a call to that same interpreter with that same text as argument, then we have an infinite loop.** A halting program has some of the same character as an interpreter: it applies to texts through abstract interpretation. Unsurprisingly, **if we apply a halting program to a program text that includes a call to that same halting program with that same text as argument, then we have an infinite loop.** A mathematical version of it cannot escape the corresponding problem: either we leave the definition of the halting function incomplete, not saying its result when applied to its own program, or we suffer inconsistency. If we choose incompleteness, we cannot require the program version to apply to texts that invoke the halting program, and we cannot conclude that it is incomputable. If we choose inconsistency, then it makes no sense to propose a program version. Either way, the incomputability argument is lost. (Hehner:2011)

Alternatively when simulating halt decider H correctly determines that D correctly simulated by H never halts this provides the basis for a correct halt status decision of the Halting Problem's otherwise impossible input.

Every computation that only stops running because some steps of this same computation had to be aborted to prevent the infinite execution of this computation is a computation that does not halt.

Because D specifies that it calls H(D,D) in recursive simulation and H must abort this recursive simulation to prevent the infinite execution of D(D) we know that D(D) does not actually halt even though it really looks like it does.

Within the Turing Machine model of computation only UTM simulations can be aborted, direct executions cannot be aborted.

Simulating (partial) halt decider applied to Peter Linz Halting Problem Proof
A simulating (partial) halt decider correctly predicts whether or not its correctly simulated input can possibly reach its own final state and halt. It does this by correctly recognizing several non-halting behavior patterns in a finite number of steps of correct simulation. Inputs that do terminate are simply simulated until they complete.

When a simulating (partial) halt decider correctly simulates N steps of its input it derives the exact same N steps that a pure UTM would derive because it is itself a UTM with extra features.

My reviewers cannot show that any of the extra features added to the UTM change the behavior of the simulated input for the first N steps of simulation:
(a) Watching the behavior doesn't change it.
(b) Matching non-halting behavior patterns doesn't change it
(c) Even aborting the simulation after N steps doesn't change the first N steps.

Because of all this we can know that the first N steps of input D simulated by simulating (partial) halt decider H are the actual behavior that D specifies to H for these same N steps.

computation that halts… “the Turing machine will halt whenever it enters a final state” (Linz:1990:234)

When we see (after N steps) that D correctly simulated by H cannot possibly reach its simulated final state in any finite number of steps of correct simulation then we have conclusive proof that D presents non-halting behavior to H.

A simulating (partial) halt decider must always stop its simulation and report non-halting when-so-ever it correctly detects that its correct simulation would never otherwise stop running. All halt deciders compute the mapping from their inputs to an accept or reject state on the basis of the actual behavior specified by this input.

When an input is defined to have a pathological relationship to its simulator this changes the behavior of this input. A simulating (partial) halt decider (with a pathological relationship) must report on this changed behavior to prevent its own infinite execution by aborting its simulation.

Summary of Linz Halting Problem Proof
The Linz halting problem proof constructs its counter-example input ⟨Ĥ⟩ on the basis of prepending and appending states to the original Linz H, (assumed halt decider) thus is named embedded_H.

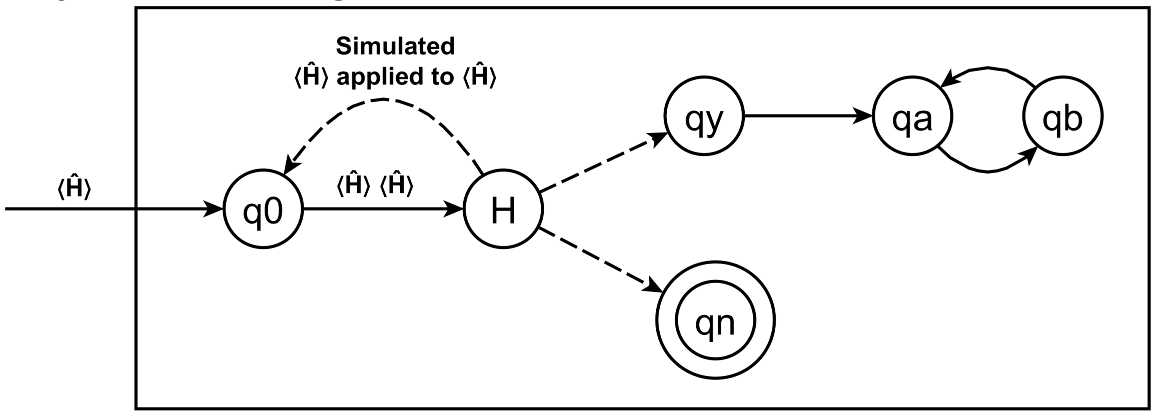
Original Linz Turing Machine H
H.q0 ⟨M⟩ w ⊢* H.qy // TM description M and finite string w, accept state
H.q0 ⟨M⟩ w ⊢* Hqn // TM description M and finite string w, reject state

The Linz term “move” means a state transition and its corresponding tape head action {move_left, move_right, read, write}.

(q0) is prepended to H to copy the ⟨M⟩ input of Ĥ. The transition from (qa) to (qb) is the conventional infinite loop appended to the (qy) accept state of embedded_H.
⊢* indicates an arbitrary number of moves.

Ĥ.q0 ⟨M⟩ ⊢* embedded_H ⟨M⟩ ⟨M⟩ ⊢* Ĥ.qy ∞ // see diagram below
Ĥ.q0 ⟨M⟩ ⊢* embedded_H ⟨M⟩ ⟨M⟩ ⊢* Ĥ.qn

Analysis of Linz Halting Problem Proof



When Ĥ is applied to ⟨Ĥ⟩
Ĥ.q0 ⟨Ĥ⟩ ⊢* embedded_H ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qy ∞
Ĥ.q0 ⟨Ĥ⟩ ⊢* embedded_H ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qn

Simulating Partial Halt Decider Applied to Linz Proof
Non-halting behavior patterns can be matched in N steps. The simulated ⟨Ĥ⟩ halts only it when reaches its simulated final state of ⟨Ĥ.qn⟩ in a finite number of steps.

Execution trace of Ĥ applied to ⟨Ĥ⟩
(a) Ĥ.q0 The input ⟨Ĥ⟩ is copied then transitions to embedded_H
(b) embedded_H applied ⟨Ĥ⟩ ⟨Ĥ⟩ (input and copy) simulates ⟨Ĥ⟩ applied to ⟨Ĥ⟩
(c) which begins at its own simulated ⟨Ĥ.q0⟩ to repeat the process

Simulation invariant: ⟨Ĥ⟩ correctly simulated by embedded_H never reaches its own simulated final state of ⟨Ĥ.qn⟩.

Therefore when embedded_H aborts the simulation of its input and transitions to its own final state of Ĥ.qn it is merely reporting this verified fact.

Conclusion

We have shown a 100% fully operational concrete example of a simulating termination analyzer applied to a pair of C functions that have the Halting Problem's pathological relationship to each other.

When it is understood that D correctly simulated by H cannot possibly halt and that H is reporting on the behavior of this correctly simulated input then H is correct to abort its simulation of D and report that this input does not halt.

The exact same reasoning applies to the Peter Linz Halting Problem proof. When embedded_H is applied to ⟨Ĥ⟩ ⟨Ĥ⟩ it transitions to Ĥ.qn indicating that its correctly simulated input cannot possibly reach its own simulated final state of ⟨Ĥ.qn⟩.

References

[1] Steffen Winterfeldt and others **libx86emu** (x86 emulation library)
1996-2017 https://github.com/wfeldt/libx86emu

[2] Olcott, Peter 2023. **The x86utm operating system:**
https://github.com/plolcott/x86utm
Several fully operational simulating termination analyzers with sample inputs.

[3] Bill Stoddart. **The Halting Paradox**
20 December 2017
https://arxiv.org/abs/1906.05340
arXiv:1906.05340 [cs.LO]

[4] E C R Hehner. **Problems with the Halting Problem**, COMPUTING2011 Symposium on 75 years of Turing Machine and Lambda-Calculus, Karlsruhe Germany, invited, 2011 October 20-21; Advances in Computer Science and Engineering v.10 n.1 p.31-60, 2013
https://www.cs.toronto.edu/~hehner/PHP.pdf

[5] Linz, Peter 1990. **An Introduction to Formal Languages and Automata.** Lexington/Toronto: D. C. Heath and Company. (317-320)