Simulating (partial) Halt Deciders Defeat the Halting Problem Proofs

A simulating 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 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:

- Watching the behavior doesn't change it.
- Matching non-halting behavior patterns doesn't change it
- 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 halt decider H are the actual behavior that D presents 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.

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.

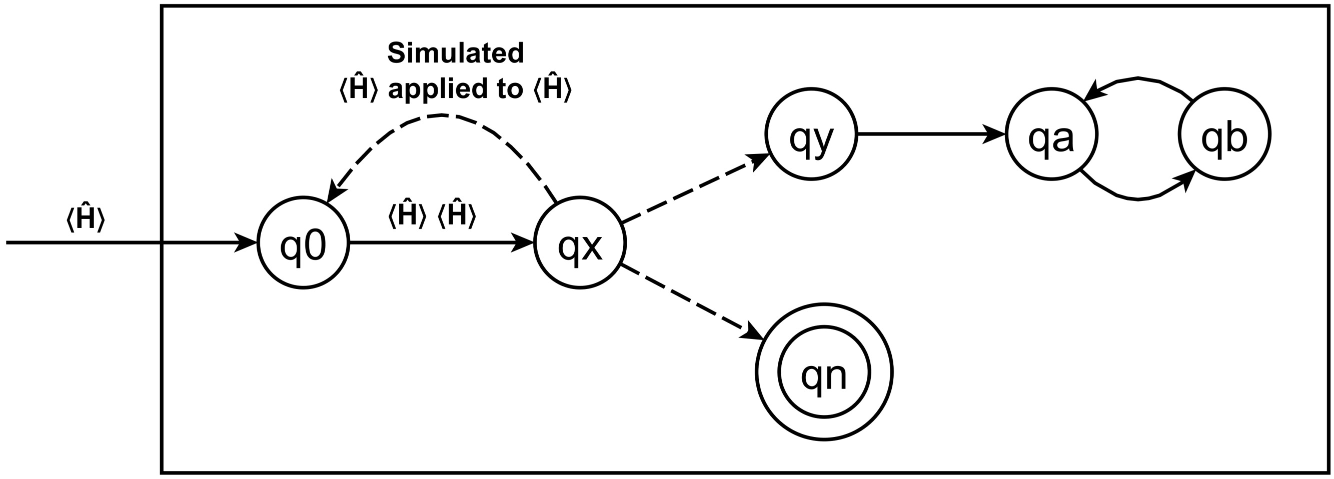
Original Linz Turing Machine H
H.q0 ⟨M⟩ w ⊢* H.qy // Turing Machine description M and finite string w, reject state
H.q0 ⟨M⟩ w ⊢* Hqn // Turing Machine description M and finite string w, accept state

(q0) is prepended to H to copy the ⟨Ĥ⟩ 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.

The Linz term “move” means a state transition and its corresponding tape head action {move_left, move_right, read, write}. Ĥ is applied to its own machine description ⟨Ĥ⟩.

Ĥ.q0 ⟨M⟩ ⊢* embedded_H ⟨M⟩ ⟨M⟩ ⊢* Ĥ.qy ∞
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

(q0) The input ⟨Ĥ⟩ is copied then transitions to (qx)
(qx) embedded_H is applied to ⟨Ĥ⟩ ⟨Ĥ⟩ (input and copy)
which simulates ⟨Ĥ⟩ applied to ⟨Ĥ⟩ **which begins at its own (q0) to repeat the process.**

This process continues to repeat until embedded_H recognizes the repeating pattern and aborts its simulation of ⟨Ĥ⟩ ⟨Ĥ⟩. embedded_H can see the same repeating pattern that we see.

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

Every "rebuttal" simply ignores this key fact
⟨Ĥ⟩ ⟨Ĥ⟩ correctly simulated by embedded_H cannot possibly reach its own simulated final state of ⟨Ĥ.qn⟩ and halt in any finite number of steps of correct simulation.

Linz, Peter 1990. An Introduction to Formal Languages and Automata. Lexington/Toronto: D. C. Heath and Company. (3**17-320)**

Concrete Rebuttal of Halting Theorem with fully operational software

Simulating halt decider H correctly predicts whether or not D correctly simulated by H can possibly reach its own final state in any finite number of correctly simulated steps. 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.

```
01 int D(int (*x)())
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 }
```

We form an isomorphism to the Linz Turing Machine analysis in the C programming language. It has the key required element that D attempts to do the opposite of whatever value H returns.

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

As was proved above in the Linz example the first N steps of input D correctly simulated by simulating halt decider H are the actual behavior that D presents to H for these same N steps.

When H correctly determines (after N simulated steps) that D correctly simulated by H cannot possibly reach its own final state on line 6 and halt then H has conclusive proof that D presents non-halting behavior to H.

Here is the sequence if H would never abort it simulation
main() calls H(D,D) that simulates D(D) at line 11
keeps repeating: simulated D(D) calls simulated H(D,D) that simulates D(D) at line 03 ...

This proves that D correctly simulated by H cannot possibly reach its own simulated final state at line 6 and halt in any finite number of steps of correct simulation.

When H aborts its simulation and returns 0 it is only affirming this verified fact.

H(D,D) fully operational in x86utm operating system: https://github.com/plolcott/x86utm

Source-code of several different partial halt deciders and their sample input.
https://github.com/plolcott/x86utm/blob/master/Halt7.c