Halting problem undecidability and infinitely nested simulation (V2)

The halting theorem counter-examples present infinitely nested simulation (non-halting) behavior to every simulating halt decider. This paper has been rewritten to be more compelling and more concise.

The pathological self-reference of the conventional halting problem proof counter-examples is overcome. The halt status of these examples is correctly determined. A simulating halt decider remains in pure simulation mode until after it determines that its input will never reach its final state. This eliminates the conventional feedback loop where the behavior of the halt decider effects the behavior of its input.

The x86utm operating system was created so that the halting problem could be examined concretely in the high level language of C. H is a function written in C that analyzes the x86 machine language execution trace of other functions written in C. H recognizes simple cases of infinite recursion and infinite loops. The conventional halting problem proof counter-example template is shown to simply be an input that does not halt.

H simulates its input with an x86 emulator until it determines that its input would never halt. As soon as H recognizes that its input would never halt it stops simulating this input and returns 0. For inputs that do halt H acts exactly as if it was an x86 emulator and simply runs its input to completion and then returns 1.

> In theoretical computer science the random-access stored-program (RASP) machine model is an abstract machine used for the purposes of algorithm development and algorithm complexity theory. ...The RASP is closest of all the abstract models to the common notion of computer.
> https://en.wikipedia.org/wiki/Random-access_stored-program_machine

The C/x86 model of computation is known to be Turing equivalent on the basis that it maps to the RASP model for all computations having all of the memory that they need. As long as an C/x86 function is a pure function of its inputs the C/x86 model of computation can be relied upon as a much higher level of abstraction of the behavior of actual Turing machines.

This criteria merely relies on the fact that the UTM simulation of a machine description of a machine is computationally equivalent to the direct execution of this same machine:

halt decider (Olcott 2021)
A halt decider accepts or rejects inputs on the basis of the actual behavior specified by these inputs. Whenever the direct execution or pure simulation of an input would never reach its final state this input is correctly decided as not halting.

> In computability theory, the halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever. https://en.wikipedia.org/wiki/Halting_problem

Because H only acts as a pure simulator of its input until after its halt status decision has been made it has no behavior that can possibly effect the behavior of its input.

Pathological Input to a halt decider is stipulated to mean any input that was defined to do the opposite of whatever its corresponding halt decider decides as Sipser describes:

> Now we construct a new Turing machine D with H as a subroutine.
> This new TM calls H to determine what M does when the input to M
> is its own description ⟨M⟩. Once D has determined this information,
> it does the opposite. (Sipser:1997:165)

When D is invoked with input **⟨D⟩** we have pathological self-reference when D calls H with ⟨D⟩ and does the opposite of whatever H returns.

Does D halt on its own machine description ⟨D⟩ ?
This question can only be correctly answered after the pathology has been removed. When a halt decider only acts as a pure simulator of its input until after its halt status decision is made there is no feedback loop of back channel communication between the halt decider and its input that can prevent a correct halt status decision. In this case the halt decider is only examining the behavior of the input and has no behavior that can effect the behavior of this input thus can ignore it own behavior.

The standard pseudo-code halting problem template "proved" that the halting problem could never be solved on the basis that neither value of true (halting) nor false (not halting) could be correctly returned form the halt decider to the confounding input.

```
// Simplified Linz(1990) Ĥ and Strachey(1965) P
void P(u32 x)
{
  if (H(x, x))
    HERE: goto HERE;
}
```

This problem is overcome on the basis that a simulating halt decider would abort the simulation of its input before ever returning any value to this input. It aborts the simulation of its input on the basis that its input specifies what is essentially infinite recursion (infinitely nested simulation) to any simulating halt decider.

Every input to a simulating halt decider that only stops running when its simulation is aborted unequivocally specifies a computation that never halts. When input to a simulating halt decider cannot possibly reach its final state then we know that this input never halts.

A simulating halt decider H divides all of its input into:
(1) Those inputs that never halt unless H aborts their simulation (never halting).
H aborts its simulation of these inputs an returns 0 for never halting.

(2) Those inputs that halt while H remains a pure simulator (halting).
H waits for its simulation of this input to complete and then returns 1 halting.

Simulating partial halt decider H correctly decides that P(P) never halts (V1)

```
#include <stdint.h>
#include <stdio.h>
typedef int (*ptr)();

int H(ptr x, ptr y)
{
  x(y); // direct execution of P(P)
  return 1;
}

// Minimal essence of Linz(1990) Ĥ
// and Strachey(1965) P
int P(ptr x)
{
  H(x, x);
  return 1; // Give P a last instruction at the "c" level
}

int main(void)
{
  H(P, P);
}
```

The above program is obviously infinitely recursive. It is self evident that when 0 to ∞ steps of the input to H(P,P) are directly executed or correctly simulated that the input to H(P,P) never reaches its final instruction.

PSR set (pathological self-reference)
$H_1(P_1,P_1)$ Is the above code.
$H_2(P_2,P_2)$ Is the above code where H_2 simulates rather than directly executes its input.
$H_3(P_3,P_4)$ Is the execution of N steps of the input of $H_1(P_1,P_1)$.
$H_4(P_4,P_4)$ Is the simulation of N steps of the input of $H_2(P_2,P_2)$.
Neither H_1 nor H_2 return any value. H_3 or H_4 may return some integer value or not.

The sequence of 1 to N configurations specified by the input to H(X, Y) cannot be correctly construed as anything other than the sequence of 1 to N steps of the (direct execution, x86 emulation or UTM simulation of this input by H.

When H directly executes 1 to N steps of its actual input this conclusively proves that this is the correct direct execution basis for the halt decider's halt status decision. The simulation of this same input derives the exact same sequence of steps.

The point in the sequence of 1 to N steps where the execution trace of the simulation of P shows that P is about to call H(P,P) again with the same input that H was called with provides conclusive proof that P would be infinitely recursive unless H aborted its simulation.

When P(P) calls H(P,P) reaches the above point in its simulation it returns 0 to P.

H is a computable function that accepts or rejects inputs in its domain on the basis that these inputs specify a sequence of configurations that reach their final state.

Simulating partial halt decider H correctly decides that P(P) never halts (V2)

```
// Simplified Linz Ĥ (Linz:1990:319)
// Strachey(1965) CPL translated to C
void P(u32 x)
{
  if (H(x, x))
    HERE: goto HERE;
}

int main()
{
  Output("Input_Halts = ", H((u32)P, (u32)P));
}

_P()
[00000c36](01)  55          push ebp
[00000c37](02)  8bec        mov ebp,esp
[00000c39](03)  8b4508      mov eax,[ebp+08] // 2nd Param
[00000c3c](01)  50          push eax
[00000c3d](03)  8b4d08      mov ecx,[ebp+08] // 1st Param
[00000c40](01)  51          push ecx
[00000c41](05)  e820fdffff  call 00000966    // call H
[00000c46](03)  83c408      add esp,+08
[00000c49](02)  85c0        test eax,eax
[00000c4b](02)  7402        jz 00000c4f
[00000c4d](02)  ebfe        jmp 00000c4d
[00000c4f](01)  5d          pop ebp
[00000c50](01)  c3          ret
Size in bytes:(0027) [00000c50]

_main()
[00000c56](01)  55          push ebp
[00000c57](02)  8bec        mov ebp,esp
[00000c59](05)  68360c0000  push 00000c36    // push P
[00000c5e](05)  68360c0000  push 00000c36    // push P
[00000c63](05)  e8fefcffff  call 00000966    // call H(P,P)
[00000c68](03)  83c408      add esp,+08
[00000c6b](01)  50          push eax
[00000c6c](05)  6857030000  push 00000357
[00000c71](05)  e810f7ffff  call 00000386
[00000c76](03)  83c408      add esp,+08
[00000c79](02)  33c0        xor eax,eax
[00000c7b](01)  5d          pop ebp
[00000c7c](01)  c3          ret
Size in bytes:(0039) [00000c7c]

 machine   stack     stack     machine    assembly
 address   address   data      code       language
 ========  ========  ========  =========  =============
[00000c56][0010172a][00000000] 55          push ebp
[00000c57][0010172a][00000000] 8bec        mov ebp,esp
[00000c59][00101726][00000c36] 68360c0000  push 00000c36 // push P
[00000c5e][00101722][00000c36] 68360c0000  push 00000c36 // push P
[00000c63][0010171e][00000c68] e8fefcffff  call 00000966 // call H(P,P)
```

Begin Local Halt Decider Simulation at Machine Address:c36

```
[00000c36][002117ca][002117ce] 55          push ebp
[00000c37][002117ca][002117ce] 8bec        mov ebp,esp
[00000c39][002117ca][002117ce] 8b4508      mov eax,[ebp+08]
[00000c3c][002117c6][00000c36] 50          push eax         // push P
[00000c3d][002117c6][00000c36] 8b4d08      mov ecx,[ebp+08]
[00000c40][002117c2][00000c36] 51          push ecx         // push P
[00000c41][002117be][00000c46] e820fdffff  call 00000966    // call H(P,P)
```

Local Halt Decider: Infinite Recursion Detected Simulation Stopped

Same criteria as V1, H sees that it is called a second time with the same input.

```
[00000c68][0010172a][00000000] 83c408      add esp,+08
[00000c6b][00101726][00000000] 50          push eax
[00000c6c][00101722][00000357] 6857030000  push 00000357
[00000c71][00101722][00000357] e810f7ffff  call 00000386
Input_Halts = 0
[00000c76][0010172a][00000000] 83c408      add esp,+08
[00000c79][0010172a][00000000] 33c0        xor eax,eax
[00000c7b][0010172e][00100000] 5d          pop ebp
[00000c7c][00101732][00000068] c3          ret
```

The direct execution of P(P) halts (V3)

The execution trace of the x86 emulation of P(P) by simulating halt decider H conclusively proves that P cannot possibly ever reach its final state of 0xc3f. This provides complete proof that that the input to H never halts thus H(P,P)==0 is correct.

```
// Simplified Linz Ĥ (Linz:1990:319)
// Strachey(1965) CPL translated to C
void P(u32 x)
{
  if (H(x, x))
    HERE: goto HERE;
}


int main()
{
  P((u32)P);
}


_P()
[00000c25](01)  55          push ebp
[00000c26](02)  8bec        mov ebp,esp
[00000c28](03)  8b4508      mov eax,[ebp+08]
[00000c2b](01)  50          push eax        // 2nd Param
[00000c2c](03)  8b4d08      mov ecx,[ebp+08]
[00000c2f](01)  51          push ecx        // 1st Param
[00000c30](05)  e820fdffff  call 00000955   // call H
[00000c35](03)  83c408      add esp,+08
[00000c38](02)  85c0        test eax,eax
[00000c3a](02)  7402        jz 00000c3e
[00000c3c](02)  ebfe        jmp 00000c3c
[00000c3e](01)  5d          pop ebp
[00000c3f](01)  c3          ret
Size in bytes:(0027) [00000c3f]

_main()
[00000c45](01)  55          push ebp
[00000c46](02)  8bec        mov ebp,esp
[00000c48](05)  68250c0000  push 00000c25 // push P
[00000c4d](05)  e8d3ffffff  call 00000c25 // call P(P)
[00000c52](03)  83c404      add esp,+04
[00000c55](02)  33c0        xor eax,eax
[00000c57](01)  5d          pop ebp
[00000c58](01)  c3          ret
Size in bytes:(0020) [00000c58]

 machine   stack     stack     machine    assembly
 address   address   data      code       language
 ========  ========  ========  =========  =============
[00000c45][001016d6][00000000] 55         push ebp
[00000c46][001016d6][00000000] 8bec       mov ebp,esp
[00000c48][001016d2][00000c25] 68250c0000 push 00000c25 // push P
[00000c4d][001016ce][00000c52] e8d3ffffff call 00000c25 // call P(P)
[00000c25][001016ca][001016d6] 55         push ebp      // P begins
[00000c26][001016ca][001016d6] 8bec       mov ebp,esp
[00000c28][001016ca][001016d6] 8b4508     mov eax,[ebp+08]
[00000c2b][001016c6][00000c25] 50         push eax      // push P
[00000c2c][001016c6][00000c25] 8b4d08     mov ecx,[ebp+08]
[00000c2f][001016c2][00000c25] 51         push ecx      // push P
[00000c30][001016be][00000c35] e820fdffff call 00000955 // call H(P,P)
```

Begin Local Halt Decider Simulation at Machine Address:c25
```
[00000c25][00211776][0021177a] 55         push ebp      // P begins
[00000c26][00211776][0021177a] 8bec       mov ebp,esp
[00000c28][00211776][0021177a] 8b4508     mov eax,[ebp+08]
[00000c2b][00211772][00000c25] 50         push eax      // push P
[00000c2c][00211772][00000c25] 8b4d08     mov ecx,[ebp+08]
[00000c2f][0021176e][00000c25] 51         push ecx      // push P
[00000c30][0021176a][00000c35] e820fdffff call 00000955 // call H(P,P)
```
Local Halt Decider: Infinite Recursion Detected Simulation Stopped

Same criteria as V2, H sees that it is called a second time with the same input.

```
[00000c35][001016ca][001016d6] 83c408     add esp,+08
[00000c38][001016ca][001016d6] 85c0       test eax,eax
[00000c3a][001016ca][001016d6] 7402       jz 00000c3e
[00000c3e][001016ce][00000c52] 5d         pop ebp
[00000c3f][001016d2][00000c25] c3         ret
[00000c52][001016d6][00000000] 83c404     add esp,+04
[00000c55][001016d6][00000000] 33c0       xor eax,eax
[00000c57][001016da][00100000] 5d         pop ebp
[00000c58][001016de][00000084] c3         ret
Number_of_User_Instructions(34)
Number of Instructions Executed(23729)
```

P(P) is conditional only on whatever H(P,P) returns. H(P,P) is conditional only on whatever the simulation or execution of its input actually does. These are two entirely different conditions that result in entirely different behavior.

Here are the divergent execution sequences at the C level:

int main(){ H(P,P); }
(1) main()
(2) calls H(P,P) that simulates the input to H(P,P)
(3) that calls H(P,P) which aborts its simulation of P(P) and returns to
(4) main().

int main(){ P(P); }
(a) main() calls P(P) that
(b) calls H(P,P) that simulates the input to H(P,P)
(c) that calls H(P,P) which aborts its simulation of P(P) and returns to
(d) P(P) that returns to main()

Peter Linz Ĥ applied to the Turing machine description of itself: ⟨Ĥ⟩

The following simplifies the syntax for the definition of the Linz Turing machine Ĥ, it is now a single machine with a single start state. A simulating halt decider is embedded at Ĥ.qx. It has been annotated so that it only shows Ĥ applied to ⟨Ĥ⟩, converting the variables to constants.

Ĥ.q0 ⟨Ĥ⟩ ⊢* Ĥ.qx ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qy ∞
If the UTM simulation of the input to Ĥ.qx ⟨Ĥ⟩ applied to ⟨Ĥ⟩ reaches its own final state.

Ĥ.q0 ⟨Ĥ⟩ ⊢* Ĥ.qx ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qn
If the pure simulation of the input to Ĥqx ⟨Ĥ⟩ ⟨Ĥ⟩ would never reach its final state (whether or not this simulation is aborted) then it is necessarily true that Ĥqx transitions to Ĥ.qn correctly.

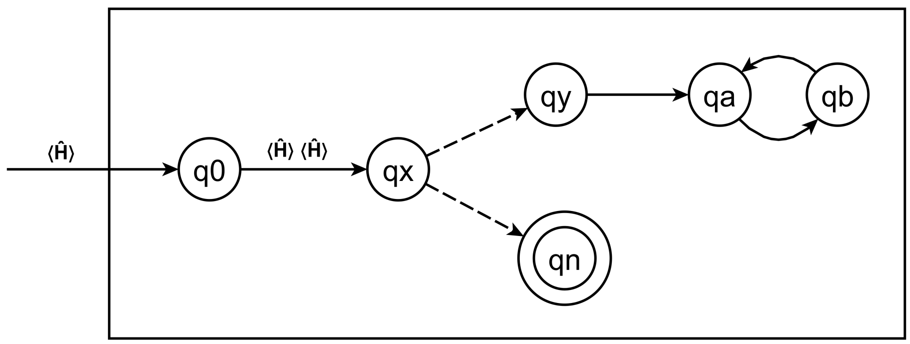

Figure 12.3 Turing Machine Ĥ applied to ⟨Ĥ⟩

Ĥ.q0 copies its input $\langle \hat{H}_0 \rangle$ to $\langle \hat{H}_1 \rangle$ then Ĥ.qx $\langle \hat{H}_0 \rangle$ $\langle \hat{H}_1 \rangle$ simulates its input
$\hat{H}_0$.q0 copies its input $\langle \hat{H}_1 \rangle$ to $\langle \hat{H}_2 \rangle$ then $\hat{H}_0$.qx $\langle \hat{H}_1 \rangle$ $\langle \hat{H}_2 \rangle$ simulates its input.
$\hat{H}_1$.q0 copies its input $\langle \hat{H}_2 \rangle$ to $\langle \hat{H}_3 \rangle$ then $\hat{H}_1$.qx $\langle \hat{H}_2 \rangle$ $\langle \hat{H}_3 \rangle$ simulates its input.
$\hat{H}_2$.q0 copies its input $\langle \hat{H}_3 \rangle$ to $\langle \hat{H}_4 \rangle$ then $\hat{H}_2$.qx $\langle \hat{H}_3 \rangle$ $\langle \hat{H}_4 \rangle$ simulates its input.

Ĥ.q0 copies its input $\langle \hat{H}_0 \rangle$ to $\langle \hat{H}_1 \rangle$ then Ĥ.qx $\langle \hat{H}_0 \rangle$ $\langle \hat{H}_1 \rangle$ simulates its input
$\hat{H}_0$.q0 copies its input $\langle \hat{H}_1 \rangle$ to $\langle \hat{H}_2 \rangle$ then $\hat{H}_0$.qx $\langle \hat{H}_1 \rangle$ $\langle \hat{H}_2 \rangle$
Ĥ.qx detects that a copy of itself is about to be simulated with a copy of its inputs.

If the simulating halt decider at Ĥ.qx never aborts its simulation of its input this input never halts. If Ĥ.qx aborts its simulation of its input this input never reaches its final state and thus never halts. In all cases for every simulating halt decider at Ĥ.qx its input never halts.

When the pure simulation of the actual input to Ĥ.qx ⟨Ĥ⟩ ⟨Ĥ⟩ never reaches the final state of this input then Ĥ.qx transitions to ⊢* Ĥ.qn is necessarily correct no matter what Ĥ ⟨Ĥ⟩ does. A halt decider is only accountable for correctly deciding the halt status of its actual input.

When the original Linz H is applied to ⟨Ĥ⟩ ⟨Ĥ⟩ it sees that its input transitions to Ĥ.qn. This provides the basis for H to transition to its final state of H.qy.

When Ĥ.qx is applied to ⟨Ĥ⟩ ⟨Ĥ⟩ it sees that none of the recursive simulations of its input ever halt it aborts the simulation of its input and correctly transitions to its final state of Ĥ.qn.

The Peter Linz conclusion (Linz:1990:320)

Now Ĥ is a Turing machine, so that it will have some description in Σ*, say ⟨Ĥ⟩. This string, in addition to being the description of Ĥ can also be used as input string. We can therefore legitimately ask what would happen if Ĥ is applied to ⟨Ĥ⟩.

Ĥ.q0 ⟨Ĥ⟩ ⊢* Ĥ.qx ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qy ∞

if Ĥ applied to ⟨Ĥ⟩ halts, and

Ĥ.q0 ⟨Ĥ⟩ ⊢* Ĥ.qx ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qn

if Ĥ applied to ⟨Ĥ⟩ does not halt. This is clearly nonsense. The contradiction tells us that our assumption of the existence of H, and hence the assumption of the decidability of the halting problem, must be false.

My rebuttal to the Peter Linz Conclusion
This explicitly ignores the possibility that the input to Ĥ.qx ⟨Ĥ⟩ ⟨Ĥ⟩ never halts and Ĥ transitions to Ĥ.qn causing Ĥ ⟨Ĥ⟩ to halt in exactly the same way that the input to H(P,P) never halts and H(P,P) returns 0 causing P(P) to halt.

Strachey, C 1965. An impossible program The Computer Journal, Volume 7, Issue 4, January 1965, Page 313, https://doi.org/10.1093/comjnl/7.4.313

Linz, Peter 1990. An Introduction to Formal Languages and Automata. Lexington/Toronto: D. C. Heath and Company. (318-320)

Sipser, Michael 1997. Introduction to the Theory of Computation. Boston: PWS Publishing Company (165-167)

Glossary of Terms

computation
The sequence of configurations leading to a halt state will be called a computation. (Linz:1990:238)

computation that halts
A Turing machine is said to halt whenever it reaches a configuration for which δ is not defined; … so the Turing machine will halt whenever it enters a final state. (Linz:1990:234)

computable function
Computable functions are the basic objects of study in computability theory. Computable functions are the formalized analogue of the intuitive notion of algorithms, in the sense that a function is computable if there exists an algorithm that can do the job of the function, i.e. given an input of the function domain it can return the corresponding output.
https://en.wikipedia.org/wiki/Computable_function

computable function (Olcott 2021)
An algorithm is applied to an input deriving an output.

computer science decider
A decider is a machine that accepts or rejects inputs.
https://cs.stackexchange.com/questions/84433/what-is-decider

halt decider (Olcott 2021)
Function H maps finite string pairs that specify a sequence of configurations to {0,1}

computer science decider
Intuitively, a decider should be a Turing machine that given an input, halts and either accepts or rejects, relaying its answer in one of many equivalent ways, such as halting at an ACCEPT or REJECT state, or leaving its answer on the output tape.
https://cs.stackexchange.com/questions/84433/what-is-decider

[Halting problem undecidability and infinitely nested simulation V2]
(https://www.researchgate.net/publication/356105750_Halting_problem_undecidability_and_infinitely_nested_simulation_V2)