Simulating Halt Deciders Defeat the Halting Theorem

The novel concept of a simulating halt decider enables halt decider H to to correctly determine the halt status of the conventional “impossible” input D that does the opposite of whatever H decides. This works equally well for Turing machines and “C” functions. The algorithm is demonstrated using “C” functions because all of the details can be shown at this high level of abstraction.

Simulating halt decider H correctly determines that D correctly simulated by H would remain stuck in recursive simulation never reaching its own final state. D cannot do the opposite of the return value from H because this return value is unreachable by every correctly simulated D. This same result is shown to be derived in the Peter Linz Turing machine based proof.

> 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. Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.
>
> For any program H that might determine if 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

```c
int D(int (*x)())
{
  int Halt_Status = H(x, x);
  if (Halt_Status)
    HERE: goto HERE;
  return Halt_Status;
}

int main()
{
  Output("Input_Halts = ", H(D,D));
  Output("Input_Halts = ", D(D));
}
```

(a) If simulating halt decider H correctly simulates its input D until H correctly predicts that its simulated D would never reach its own "return" statement in any finite number of simulated steps THEN

(b) H can abort its simulation of D and correctly report that D specifies a non-halting sequence of configurations.

The above words are a tautology in that the meaning of the words proves that:
(b) is a necessary consequence of (a).

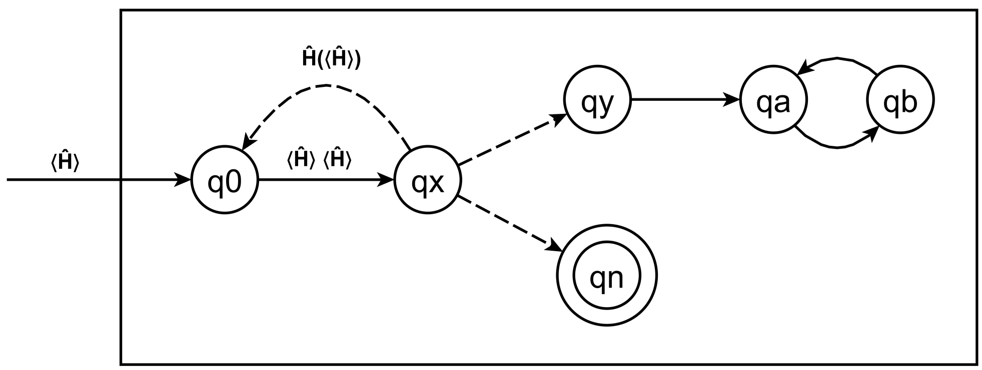
It is a verified fact that: H(D,D) does correctly compute the mapping from its input to its reject state on the basis that H correctly predicts that D correctly simulated by H would never halt (reach its own "return" statement and terminate normally).

Complete halt deciding system (Visual Studio Project)
(a) x86utm operating system
(b) x86 emulator adapted from libx86emu to compile under Windows
(c) Several halt deciders and their sample inputs contained within Halt7.c
(d) The execution trace of H applied to D is shown in Halt7out.txt
https://liarparadox.org/2023_02_07.zip

Peter Linz Halting Problem Proof adapted to use a simulating halt decider

When we see the notion of a simulating halt decider applied to the embedded copy of Linz H at state (qx) then we can see that the ⟨Ĥ⟩ ⟨Ĥ⟩ input to embedded_H specifies recursive simulation that never reaches its final state of ⟨Ĥ.qn⟩ forming a cycle from (qx) to (q0).

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



Linz describes state (qx) as a copy of his original H that has been embedded within his Ĥ

Ĥ.q_0 ⟨Ĥ⟩ ⊢* embedded_H ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qy ∞
If ⟨Ĥ⟩ ⟨Ĥ⟩ correctly simulated by embedded_H would reach its own final state of ⟨Ĥ.qn⟩.

Ĥ.q_0 ⟨Ĥ⟩ ⊢* embedded_H ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qn
If ⟨Ĥ⟩ ⟨Ĥ⟩ correctly simulated by embedded_H would never reach its own final state of ⟨Ĥ.qn⟩.

When Ĥ is applied to ⟨Ĥ⟩ // subscripts indicate unique finite strings
Ĥ copies its input ⟨$\hat{H}_0$⟩ to ⟨$\hat{H}_1$⟩ then H simulates ⟨$\hat{H}_0$⟩ ⟨$\hat{H}_1$⟩

Then these steps would keep repeating: (unless their simulation is aborted)
$\hat{H}_0$ copies its input ⟨$\hat{H}_1$⟩ to ⟨$\hat{H}_2$⟩ then embedded_H_0 simulates ⟨$\hat{H}_1$⟩ ⟨$\hat{H}_2$⟩
$\hat{H}_1$ copies its input ⟨$\hat{H}_2$⟩ to ⟨$\hat{H}_3$⟩ then embedded_H_1 simulates ⟨$\hat{H}_2$⟩ ⟨$\hat{H}_3$⟩
$\hat{H}_2$ copies its input ⟨$\hat{H}_3$⟩ to ⟨$\hat{H}_4$⟩ then embedded_H_2 simulates ⟨$\hat{H}_3$⟩ ⟨$\hat{H}_4$⟩...

Since we can see that the input: ⟨$\hat{H}_0$⟩ ⟨$\hat{H}_1$⟩ correctly simulated by embedded_H would never reach its own final state of ⟨$\hat{H}_0$.qn⟩ we know that ⟨$\hat{H}_0$⟩ specifies a non-halting sequence of configurations.

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

Appendix

When H correctly simulates D it finds that D remains stuck in recursive simulation

```
int D(int (*x)())
{
  int Halt_Status = H(x, x);
  if (Halt_Status)
    HERE: goto HERE;
  return Halt_Status;
}

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

_D()
[00001d12] 55             push ebp
[00001d13] 8bec           mov ebp,esp
[00001d15] 51             push ecx
[00001d16] 8b4508         mov eax,[ebp+08]
[00001d19] 50             push eax
[00001d1a] 8b4d08         mov ecx,[ebp+08]
[00001d1d] 51             push ecx
[00001d1e] e83ff8ffff     call 00001562
[00001d23] 83c408         add esp,+08
[00001d26] 8945fc         mov [ebp-04],eax
[00001d29] 837dfc00       cmp dword [ebp-04],+00
[00001d2d] 7402           jz 00001d31
[00001d2f] ebfe           jmp 00001d2f
[00001d31] 8b45fc         mov eax,[ebp-04]
[00001d34] 8be5           mov esp,ebp
[00001d36] 5d             pop ebp
[00001d37] c3             ret
Size in bytes:(0038) [00001d37]

_main()
[00001d72] 55             push ebp
[00001d73] 8bec           mov ebp,esp
[00001d75] 68121d0000     push 00001d12
[00001d7a] 68121d0000     push 00001d12
[00001d7f] e8def7ffff     call 00001562
[00001d84] 83c408         add esp,+08
[00001d87] 50             push eax
[00001d88] 6883070000     push 00000783
[00001d8d] e810eaffff     call 000007a2
[00001d92] 83c408         add esp,+08
[00001d95] 33c0           xor eax,eax
[00001d97] 5d             pop ebp
[00001d98] c3             ret
Size in bytes:(0039) [00001d98]

 machine   stack     stack     machine    assembly
 address   address   data      code       language
 ========  ========  ========  =========  =============
[00001d72][0010305d][00000000] 55         push ebp
[00001d73][0010305d][00000000] 8bec       mov ebp,esp
[00001d75][00103059][00001d12] 68121d0000 push 00001d12
[00001d7a][00103055][00001d12] 68121d0000 push 00001d12
[00001d7f][00103051][00001d84] e8def7ffff call 00001562
```

```
H: Begin Simulation   Execution Trace Stored at:113109
Address_of_H:1562
[00001d12][001130f5][001130f9] 55         push ebp      ; begin D
[00001d13][001130f5][001130f9] 8bec       mov ebp,esp
[00001d15][001130f1][001030c5] 51         push ecx
[00001d16][001130f1][001030c5] 8b4508     mov eax,[ebp+08]
[00001d19][001130ed][00001d12] 50         push eax      ; push address of D
[00001d1a][001130ed][00001d12] 8b4d08     mov ecx,[ebp+08]
[00001d1d][001130e9][00001d12] 51         push ecx      ; push address of D
[00001d1e][001130e5][00001d23] e83ff8ffff call 00001562  ; call H
H: Infinitely Recursive Simulation Detected Simulation Stopped
```

We can see that the first seven instructions of D simulated by H precisely match the first seven instructions of the x86 source-code of D. This conclusively proves that these instructions were simulated correctly.

Anyone sufficiently technically competent in the x86 programming language will agree that the above execution trace of D correctly simulated by H shows that D could never reach its own final state at machine address [00001d98] and terminate normally.

H detects that D is calling itself with the exact same arguments that H was called with and there are no conditional branch instructions from the beginning of D to its call to H that can possibly escape the repetition of this recursive simulation.

```
[00001d84][0010305d][00000000] 83c408         add esp,+08
[00001d87][00103059][00000000] 50             push eax
[00001d88][00103055][00000783] 6883070000     push 00000783
[00001d8d][00103055][00000783] e810eaffff     call 000007a2
Input_Halts = 0
[00001d92][0010305d][00000000] 83c408         add esp,+08
[00001d95][0010305d][00000000] 33c0           xor eax,eax
[00001d97][00103061][00000018] 5d             pop ebp
[00001d98][00103065][00000000] c3             ret
Number of Instructions Executed(975) == 15 Pages
```