Halting problem undecidability and infinitely nested simulation

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 stop running unless its simulation is aborted. 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.

Simulating Halt Decider Theorem (Olcott 2020):
A simulating halt decider correctly decides that any input that never halts unless the simulating halt decider aborts its simulation of this input is an input that never halts.

> the Turing machine halting problem. Simply stated, the problem is: given the description of a Turing machine M and an input w, does M, when started in the initial configuration q0w, perform a computation that eventually halts? (Linz:1990:317).

> 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

In order to show that the above two definitions have been satisfied we only have to show that (an at least partial) halt decider H does correctly decide whether or not its input description of a Turing machine or computer program would halt on its input.

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. Because of this H screens out its own address range in every execution trace that it examines. This is why we never see any instructions of H in any execution trace after an input calls H.

Halting computation: is any computation that eventually reaches its own final state.

> A Turing machine is said to halt whenever it reaches a configuration for which δ is not defined; this is possible because δ is a partial function. In fact, we will assume that no transitions are defined for any final state, so the Turing machine will halt whenever it enters a final state. (Linz:1990:234)

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. It ignores 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.

```
procedure compute_g(i):   // (Wikipedia:Halting Problem)
  if f(i, i) == 0 then    // adapted from (Strachey, C 1965)
    return 0              // originally written in CPL
  else                    // ancestor of the BCPL, B and C
    loop forever          // programming languages
```

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 while H remains a pure simulator (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)

H analyzes the (currently updated) stored execution trace of its x86 emulation of P(P) after it simulates each instruction of input (P, P). As soon as a non-halting behavior pattern is matched H aborts the simulation of its input and decides that its input never reaches its final state.

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 0xc50. This provides complete proof that that the input to H never halts thus H(P,P)==0 is correct.

```
// Simplified Linz Ä (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((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)

[00000c36][0025c1f2][0025c1f6] 55          push ebp
[00000c37][0025c1f2][0025c1f6] 8bec        mov ebp,esp
[00000c39][0025c1f2][0025c1f6] 8b4508      mov eax,[ebp+08]
[00000c3c][0025c1ee][00000c36] 50          push eax       // push P
[00000c3d][0025c1ee][00000c36] 8b4d08      mov ecx,[ebp+08]
[00000c40][0025c1ea][00000c36] 51          push ecx       // push P
[00000c41][0025c1e6][00000c46] e820fdffff  call 00000966  // call H(P,P)
Local Halt Decider: Infinite Recursion Detected Simulation Stopped
```

In the above 14 instructions of the simulation of P(P) we can see that the first 7 instructions of P are repeated. The end of this sequence of 7 instructions P calls H with its own machine address as the parameters to H(P,P). Because H only examines the behavior of its inputs and ignores its own behavior when H(P,P) is called we only see the first instruction of P being simulated.

Anyone knowing the x86 language well enough can see that none of these 7 simulated instructions of P have any escape from their infinitely repeating behavior pattern that is specified directly in the x86 source-code of P. When H recognizes this infinitely repeating pattern it aborts its simulation of P(P) and reports that its input: (P,P) never reaches its final state of 0xc50.

```
[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
Number_of_User_Instructions(27)
Number of Instructions Executed(23721)
```

Simulating partial halt decider H correctly decides that P2(P2) halts (V2)

When we create an exact copy of H and change P to Call this exact copy: H1. H can see that H1 aborts its simulation of its input and returns 1 indicating that its input halts.

When H is the only halt decider it correctly reports that its input never halts unless H aborts the simulation of this input. When H is not the same halt decider as the one that P calls then H correctly reports that P halts because it can see that H1 aborts its simulation of P. In both cases H is correct.

```
void P2(u32 x)
{
  if (H1(x, x))
    HERE: goto HERE;
}


int main()
{

  Output("Input_Halts = ", H((u32)P2, (u32)P2));
}


_P2()
[00000e52](01)  55          push ebp
[00000e53](02)  8bec        mov ebp,esp
[00000e55](03)  8b4508      mov eax,[ebp+08]
[00000e58](01)  50          push eax
[00000e59](03)  8b4d08      mov ecx,[ebp+08]
[00000e5c](01)  51          push ecx
[00000e5d](05)  e8b0fcffff  call 00000b12  // call H1
[00000e62](03)  83c408      add esp,+08
[00000e65](02)  85c0        test eax,eax
[00000e67](02)  7402        jz 00000e6b
[00000e69](02)  ebfe        jmp 00000e69
[00000e6b](01)  5d          pop ebp
[00000e6c](01)  c3          ret
Size in bytes:(0027) [00000e6c]

_main()
[00000e72](01)  55          push ebp
[00000e73](02)  8bec        mov ebp,esp
[00000e75](05)  68520e0000  push 00000e52  // push P2
[00000e7a](05)  68520e0000  push 00000e52  // push P2
[00000e7f](05)  e84efeffff  call 00000cd2  // call H
[00000e84](03)  83c408      add esp,+08
[00000e87](01)  50          push eax
[00000e88](05)  6823030000  push 00000323
[00000e8d](05)  e8c0f4ffff  call 00000352
[00000e92](03)  83c408      add esp,+08
[00000e95](02)  33c0        xor eax,eax
[00000e97](01)  5d          pop ebp
[00000e98](01)  c3          ret
Size in bytes:(0039) [00000e98]
```

```
    machine   stack     stack     machine     assembly
    address   address   data      code        language
    ========  ========  ========  ==========  =============
...[00000e72][00101a94][00000000] 55          push ebp
...[00000e73][00101a94][00000000] 8bec        mov ebp,esp
...[00000e75][00101a90][00000e52] 68520e0000  push 00000e52 // push P2
...[00000e7a][00101a8c][00000e52] 68520e0000  push 00000e52 // push P2
...[00000e7f][00101a88][00000e84] e84efeffff  call 00000cd2 // call H

Begin Local Halt Decider Simulation at Machine Address:e52
...[00000e52][00211b34][00211b38] 55          push ebp
...[00000e53][00211b34][00211b38] 8bec        mov ebp,esp
...[00000e55][00211b34][00211b38] 8b4508      mov eax,[ebp+08]
...[00000e58][00211b30][00000e52] 50          push eax      // push P2
...[00000e59][00211b30][00000e52] 8b4d08      mov ecx,[ebp+08]
...[00000e5c][00211b2c][00000e52] 51          push ecx      // push P2
...[00000e5d][00211b28][00000e62] e8b0fcffff  call 00000b12 // call H1

Begin Local Halt Decider Simulation at Machine Address:e52
...[00000e52][0025c55c][0025c560] 55          push ebp
...[00000e53][0025c55c][0025c560] 8bec        mov ebp,esp
...[00000e55][0025c55c][0025c560] 8b4508      mov eax,[ebp+08]
...[00000e58][0025c558][00000e52] 50          push eax      // push P2
...[00000e59][0025c558][00000e52] 8b4d08      mov ecx,[ebp+08]
...[00000e5c][0025c554][00000e52] 51          push ecx      // push P2
...[00000e5d][0025c550][00000e62] e8b0fcffff  call 00000b12 // call H1
...[00000e52][002a6f84][002a6f88] 55          push ebp
...[00000e53][002a6f84][002a6f88] 8bec        mov ebp,esp
...[00000e55][002a6f84][002a6f88] 8b4508      mov eax,[ebp+08]
...[00000e58][002a6f80][00000e52] 50          push eax      // push P2
...[00000e59][002a6f80][00000e52] 8b4d08      mov ecx,[ebp+08]
...[00000e5c][002a6f7c][00000e52] 51          push ecx      // push P2
...[00000e5d][002a6f78][00000e62] e8b0fcffff  call 00000b12 // call H1
Local Halt Decider: Infinite Recursion Detected Simulation Stopped

...[00000e62][00211b34][00211b38] 83c408      add esp,+08
...[00000e65][00211b34][00211b38] 85c0        test eax,eax
...[00000e67][00211b34][00211b38] 7402        jz 00000e6b
...[00000e6b][00211b38][00000d8f] 5d          pop ebp
...[00000e6c][00211b3c][00000e52] c3          ret
...[00000e84][00101a94][00000000] 83c408      add esp,+08
...[00000e87][00101a90][00000001] 50          push eax
...[00000e88][00101a8c][00000323] 6823030000  push 00000323
---[00000e8d][00101a8c][00000323] e8c0f4ffff  call 00000352
Input_Halts = 1
...[00000e92][00101a94][00000000] 83c408      add esp,+08
...[00000e95][00101a94][00000000] 33c0        xor eax,eax
...[00000e97][00101a98][00100000] 5d          pop ebp
...[00000e98][00101a9c][00000004] c3          ret
Number_of_User_Instructions(1)
Number of Instructions Executed(617658)
```

The fact that H and H1 are at different machine addresses and that H is called first makes two functions with identical machine code behave differently on the exact same input.

The fact that H is called first causes H to monitor the results of the behavior of H1(P,P). This creates a dependency of H on behavior of H1(P,P). H1(P,P) has no such dependency on another halt decider.

The fact that H and H1 are at different machine address derives a key difference in their execution trace that derives a key difference in their halt status decision.

The halt deciders look for the same function to be called with the same data twice in sequence. H is never called twice. H1 sees a function (itself) called twice in sequence with the same data.

Because H is called first and H is at a different machine address than H1 the abort simulation criteria is not met for H. When the abort simulation criteria is met by H1(P,P) then H sees that its input halts.

The fact that H and H1 are at different machines addresses and that H is called first makes two functions with identical machine code behave differently on the exact same input.

Because H(P,P) and H1(P,P) are distinctly different computations they can have different behavior without contradiction.

The direct execution of P(P) shown in the next section is computationally equivalent to the pure simulation of P(P) invoked from main() by H shown above. Because P already has its own halt decider H never needs to abort its simulation of P(P) thus H stays in simulation mode.

When H is the only halt decider (as in the prior section) then it correctly determines that it must abort its simulation of P(P). H cannot simply wait for itself to abort its simulation of P(P) later on because it would never be aborted if H simply waited for itself to do this. This is computationally different than the direct invication of P(P) in the next section.

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 A (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 P0(P)
[00000c25][001016ca][001016d6] 55         push ebp      // P0 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 H0(P1,P1)
```

```
Begin Local Halt Decider Simulation at Machine Address:c25
[00000c25][00211776][0021177a] 55          push ebp      // P1 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 H1(P2,P2)

[00000c25][0025c19e][0025c1a2] 55          push ebp      // P2 begins
[00000c26][0025c19e][0025c1a2] 8bec        mov ebp,esp
[00000c28][0025c19e][0025c1a2] 8b4508      mov eax,[ebp+08]
[00000c2b][0025c19a][00000c25] 50          push eax      // push P
[00000c2c][0025c19a][00000c25] 8b4d08      mov ecx,[ebp+08]
[00000c2f][0025c196][00000c25] 51          push ecx      // push P
[00000c30][0025c192][00000c35] e820fdffff call 00000955 // call H2(P3,P3)
Local Halt Decider: Infinite Recursion Detected Simulation Stopped
```

In the above computation (zero based addressing) H_0 aborts the P_1 invocation chain. No P(P) ever stops running unless H_0 aborts its simulation of P_1

Subscripts indicate that a new process context (with its own RAM, stack and registers) has been created to simulate the virtual machine input to H. Every time H is called it creates a new process context to simulate its inputs.

P_0 and H_0 are executed rather than simulated in a process context.
P_1 and H_1 are simulated in the same process context and are slaves to H_0
P_2 and H_2 are simulated in the same process context and are slaves to H_1

```
[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)
```

(1) H does perform a pure simulation of its input until after it makes its halt status decision.

(2) It can be verified that this is a pure simulation on the basis that the execution trace does what the x86 source-code of P specifies.

(3) Because there are no control flow instructions in the execution trace that can possibly escape the infinite recursion the execution trace proves that a pure simulation of the above input cannot possibly ever reach its final state.

(4) Therefore H was correct when it decided that its input never halts.

The executed instances of H(P,P) are distinctly different computations than the simulated instances in that the executed instances are not under the dominion of a halt decider. It is this difference that enables them to have different behavior.

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

```
void Infinite_Loop()
{
  HERE: goto HERE;
}

int main()
{
  u32 Input_Would_Halt2 = H((u32)Infinite_Loop, (u32)Infinite_Loop);
  Output("Input_Would_Halt2 = ", Input_Would_Halt2);
}

_Infinite_Loop()
[00000ab0](01)  55              push ebp
[00000ab1](02)  8bec            mov ebp,esp
[00000ab3](02)  ebfe            jmp 00000ab3
[00000ab5](01)  5d              pop ebp
[00000ab6](01)  c3              ret
Size in bytes:(0007) [00000ab6]

_main()
[00000c00](01)  55              push ebp
[00000c01](02)  8bec            mov ebp,esp
[00000c03](01)  51              push ecx
[00000c04](05)  68b00a0000      push 00000ab0
[00000c09](05)  68b00a0000      push 00000ab0
[00000c0e](05)  e84dfdffff      call 00000960
[00000c13](03)  83c408          add esp,+08
[00000c16](03)  8945fc          mov [ebp-04],eax
[00000c19](03)  8b45fc          mov eax,[ebp-04]
[00000c1c](01)  50              push eax
[00000c1d](05)  684b030000      push 0000034b
[00000c22](05)  e859f7ffff      call 00000380
[00000c27](03)  83c408          add esp,+08
[00000c2a](02)  33c0            xor eax,eax
[00000c2c](02)  8be5            mov esp,ebp
[00000c2e](01)  5d              pop ebp
[00000c2f](01)  c3              ret
Size in bytes:(0048) [00000c2f]
```

Execution Trace of H(Infinite_Loop, Infinite_Loop)

```
 machine   stack     stack     machine    assembly
 address   address   data      code       language
 ========  ========  ========  =========  ==============
[00000c00][00101693][00000000] 55         push ebp
[00000c01][00101693][00000000] 8bec       mov ebp,esp
[00000c03][0010168f][00000000] 51         push ecx
[00000c04][0010168b][00000ab0] 68b00a0000 push 00000ab0
[00000c09][00101687][00000ab0] 68b00a0000 push 00000ab0
[00000c0e][00101683][00000c13] e84dfdffff call 00000960

Begin Local Halt Decider Simulation at Machine Address:ab0
[00000ab0][00211733][00211737] 55         push ebp
[00000ab1][00211733][00211737] 8bec       mov ebp,esp
[00000ab3][00211733][00211737] ebfe       jmp 00000ab3
[00000ab3][00211733][00211737] ebfe       jmp 00000ab3
Local Halt Decider: Infinite Loop Detected Simulation Stopped
```

```
[00000c13][0010168f][00000000] 83c408     add esp,+08
[00000c16][0010168f][00000000] 8945fc     mov [ebp-04],eax
[00000c19][0010168f][00000000] 8b45fc     mov eax,[ebp-04]
[00000c1c][0010168b][00000000] 50         push eax
[00000c1d][00101687][0000034b] 684b030000 push 0000034b
[00000c22][00101687][0000034b] e859f7ffff call 00000380
Input_Would_Halt2 = 0
[00000c27][0010168f][00000000] 83c408     add esp,+08
[00000c2a][0010168f][00000000] 33c0       xor eax,eax
[00000c2c][00101693][00000000] 8be5       mov esp,ebp
[00000c2e][00101697][00100000] 5d         pop ebp
[00000c2f][0010169b][00000050] c3         ret
Number_of_User_Instructions(21)
Number of Instructions Executed(640)
```

Simulating partial halt decider H decides that Infinite_Recursion() never halts

```
void Infinite_Recursion(u32 N)
{
  Infinite_Recursion(N);
}

int main()
{
   u32 Input_Halts = H((u32)Infinite_Recursion, 3);
   Output("Input_Halts = ", Input_Halts);
}

_Infinite_Recursion()
[00000ac6](01)  55              push ebp
[00000ac7](02)  8bec            mov ebp,esp
[00000ac9](03)  8b4508          mov eax,[ebp+08]
[00000acc](01)  50              push eax
[00000acd](05)  e8f4ffffff      call 00000ac6
[00000ad2](03)  83c404          add esp,+04
[00000ad5](01)  5d              pop ebp
[00000ad6](01)  c3              ret
Size in bytes:(0017) [00000ad6]


_main()
[00000c46](01)  55              push ebp
[00000c47](02)  8bec            mov ebp,esp
[00000c49](01)  51              push ecx
[00000c4a](02)  6a03            push +03
[00000c4c](05)  68c60a0000      push 00000ac6
[00000c51](05)  e810fdffff      call 00000966
[00000c56](03)  83c408          add esp,+08
[00000c59](03)  8945fc          mov [ebp-04],eax
[00000c5c](03)  8b45fc          mov eax,[ebp-04]
[00000c5f](01)  50              push eax
[00000c60](05)  6857030000      push 00000357
[00000c65](05)  e81cf7ffff      call 00000386
[00000c6a](03)  83c408          add esp,+08
[00000c6d](02)  33c0            xor eax,eax
[00000c6f](02)  8be5            mov esp,ebp
[00000c71](01)  5d              pop ebp
[00000c72](01)  c3              ret
Size in bytes:(0045) [00000c72]
```

Execution Trace of H(Infinite_Recursion, 3)

```
 machine   stack     stack     machine    assembly
 address   address   data      code       language
 ========  ========  ========  =========  ==============
[00000c46][001016fa][00000000] 55         push ebp
[00000c47][001016fa][00000000] 8bec       mov ebp,esp
[00000c49][001016f6][00000000] 51         push ecx
[00000c4a][001016f2][00000003] 6a03       push +03
[00000c4c][001016ee][00000ac6] 68c60a0000 push 00000ac6
[00000c51][001016ea][00000c56] e810fdffff call 00000966
```

```
Begin Local Halt Decider Simulation at Machine Address:ac6
[00000ac6][0021179a][0021179e] 55         push ebp
[00000ac7][0021179a][0021179e] 8bec       mov ebp,esp
[00000ac9][0021179a][0021179e] 8b4508     mov eax,[ebp+08]
[00000acc][00211796][00000003] 50         push eax
[00000acd][00211792][00000ad2] e8f4ffffff call 00000ac6
[00000ac6][0021178e][0021179a] 55         push ebp
[00000ac7][0021178e][0021179a] 8bec       mov ebp,esp
[00000ac9][0021178e][0021179a] 8b4508     mov eax,[ebp+08]
[00000acc][0021178a][00000003] 50         push eax
[00000acd][00211786][00000ad2] e8f4ffffff call 00000ac6
Local Halt Decider: Infinite Recursion Detected Simulation Stopped
```

_Infinite_Recursion() calls itself recursively with the same input. It has no escape from this infinite recursion. H recognizes this infinite behavior pattern, aborts its simulation of _Infinite_Recursion() and reports that this input never halts.

```
[00000c56][001016f6][00000000] 83c408     add esp,+08
[00000c59][001016f6][00000000] 8945fc     mov [ebp-04],eax
[00000c5c][001016f6][00000000] 8b45fc     mov eax,[ebp-04]
[00000c5f][001016f2][00000000] 50         push eax
[00000c60][001016ee][00000357] 6857030000 push 00000357
[00000c65][001016ee][00000357] e81cf7ffff call 00000386
Input_Halts = 0
[00000c6a][001016f6][00000000] 83c408     add esp,+08
[00000c6d][001016f6][00000000] 33c0       xor eax,eax
[00000c6f][001016fa][00000000] 8be5       mov esp,ebp
[00000c71][001016fe][00100000] 5d         pop ebp
[00000c72][00101702][00000068] c3         ret
Number_of_User_Instructions(27)
Number of Instructions Executed(1240)
```

Infinite recursion detection criteria:
If the execution trace of function X() called by function Y() shows:
(1) Function X() is called twice in sequence from the same machine address of Y().
(2) With the same parameters to X().
(3) With no conditional branch or indexed jump instructions in Y().
(4) With no function call returns from X().
then the function call from Y() to X() is infinitely recursive.

Simulating partial halt decider H decides that Factorial(3) halts

```
int Factorial(int n)
{
  Output("Factorial(n)",n);
  if (n > 1)
    return n * Factorial(n - 1);
  else
    return 1;
}


int main()
{
  Output("Input_Halts = ", H(Factorial, 3));
}


_Factorial()
[00000de2](01)  55          push ebp
[00000de3](02)  8bec        mov ebp,esp
[00000de5](03)  8b4508      mov eax,[ebp+08]
[00000de8](01)  50          push eax
[00000de9](05)  6813030000  push 00000313
[00000dee](05)  e85ff5ffff  call 00000352
[00000df3](03)  83c408      add esp,+08
[00000df6](04)  837d0801    cmp dword [ebp+08],+01
[00000dfa](02)  7e17        jng 00000e13
[00000dfc](03)  8b4d08      mov ecx,[ebp+08]
[00000dff](03)  83e901      sub ecx,+01
[00000e02](01)  51          push ecx
[00000e03](05)  e8daffffff  call 00000de2
[00000e08](03)  83c404      add esp,+04
[00000e0b](04)  0faf4508    imul eax,[ebp+08]
[00000e0f](02)  eb07        jmp 00000e18
[00000e11](02)  eb05        jmp 00000e18
[00000e13](05)  b801000000  mov eax,00000001
[00000e18](01)  5d          pop ebp
[00000e19](01)  c3          ret
Size in bytes:(0056) [00000e19]

_main()
[00000ea2](01)  55          push ebp
[00000ea3](02)  8bec        mov ebp,esp
[00000ea5](02)  6a03        push +03
[00000ea7](05)  68e20d0000  push 00000de2
[00000eac](05)  e821feffff  call 00000cd2
[00000eb1](03)  83c408      add esp,+08
[00000eb4](01)  50          push eax
[00000eb5](05)  6823030000  push 00000323
[00000eba](05)  e893f4ffff  call 00000352
[00000ebf](03)  83c408      add esp,+08
[00000ec2](02)  33c0        xor eax,eax
[00000ec4](01)  5d          pop ebp
[00000ec5](01)  c3          ret
Size in bytes:(0036) [00000ec5]
```

```
    machine   stack     stack     machine     assembly
    address   address   data      code        language
    ========  ========  ========  ==========  =============
...[00000ea2][00101ae7][00000000] 55          push ebp
...[00000ea3][00101ae7][00000000] 8bec        mov ebp,esp
...[00000ea5][00101ae3][00000003] 6a03        push +03
...[00000ea7][00101adf][00000de2] 68e20d0000  push 00000de2
...[00000eac][00101adb][00000eb1] e821feffff  call 00000cd2

Begin Local Halt Decider Simulation at Machine Address:de2
...[00000de2][00211b87][00211b8b] 55          push ebp
...[00000de3][00211b87][00211b8b] 8bec        mov ebp,esp
...[00000de5][00211b87][00211b8b] 8b4508      mov eax,[ebp+08]
...[00000de8][00211b83][00000003] 50          push eax
...[00000de9][00211b7f][00000313] 6813030000  push 00000313
---[00000dee][00211b7f][00000313] e85ff5ffff  call 00000352
Factorial(n)3
...[00000df3][00211b87][00211b8b] 83c408      add esp,+08
...[00000df6][00211b87][00211b8b] 837d0801    cmp dword [ebp+08],+01
...[00000dfa][00211b87][00211b8b] 7e17        jng 00000e13
...[00000dfc][00211b87][00211b8b] 8b4d08      mov ecx,[ebp+08]
...[00000dff][00211b87][00211b8b] 83e901      sub ecx,+01
...[00000e02][00211b83][00000002] 51          push ecx
...[00000e03][00211b7f][00000e08] e8daffffff  call 00000de2
...[00000de2][00211b7b][00211b87] 55          push ebp
...[00000de3][00211b7b][00211b87] 8bec        mov ebp,esp
...[00000de5][00211b7b][00211b87] 8b4508      mov eax,[ebp+08]
...[00000de8][00211b77][00000002] 50          push eax
...[00000de9][00211b73][00000313] 6813030000  push 00000313
---[00000dee][00211b73][00000313] e85ff5ffff  call 00000352
Factorial(n)2
...[00000df3][00211b7b][00211b87] 83c408      add esp,+08
...[00000df6][00211b7b][00211b87] 837d0801    cmp dword [ebp+08],+01
...[00000dfa][00211b7b][00211b87] 7e17        jng 00000e13
...[00000dfc][00211b7b][00211b87] 8b4d08      mov ecx,[ebp+08]
...[00000dff][00211b7b][00211b87] 83e901      sub ecx,+01
...[00000e02][00211b77][00000001] 51          push ecx
...[00000e03][00211b73][00000e08] e8daffffff  call 00000de2
...[00000de2][00211b6f][00211b7b] 55          push ebp
...[00000de3][00211b6f][00211b7b] 8bec        mov ebp,esp
...[00000de5][00211b6f][00211b7b] 8b4508      mov eax,[ebp+08]
...[00000de8][00211b6b][00000001] 50          push eax
...[00000de9][00211b67][00000313] 6813030000  push 00000313
---[00000dee][00211b67][00000313] e85ff5ffff  call 00000352
Factorial(n)1
...[00000df3][00211b6f][00211b7b] 83c408      add esp,+08
...[00000df6][00211b6f][00211b7b] 837d0801    cmp dword [ebp+08],+01
...[00000dfa][00211b6f][00211b7b] 7e17        jng 00000e13
...[00000e13][00211b6f][00211b7b] b801000000  mov eax,00000001
...[00000e18][00211b73][00000e08] 5d          pop ebp
...[00000e19][00211b77][00000001] c3          ret
...[00000eb1][00101ae7][00000000] 83c408      add esp,+08
...[00000eb4][00101ae3][00000001] 50          push eax
...[00000eb5][00101adf][00000323] 6823030000  push 00000323
---[00000eba][00101adf][00000323] e893f4ffff  call 00000352
Input_Halts = 1
...[00000ebf][00101ae7][00000000] 83c408      add esp,+08
...[00000ec2][00101ae7][00000000] 33c0        xor eax,eax
...[00000ec4][00101aeb][00100000] 5d          pop ebp
...[00000ec5][00101aef][000000c8] c3          ret
Number_of_User_Instructions(51)
Number of Instructions Executed(3714)
```

Strachey's Impossible Program

To the Editor,
The Computer Journal.

An impossible program

Sir,
A well-known piece of folk-lore among programmers holds that it is impossible to write a program which can examine any other program and tell, in every case, if it will terminate or get into a closed loop when it is run. I have never actually seen a proof of this in print, and though Alan Turing once gave me a verbal proof (in a railway carriage on the way to a Conference at the NPL in 1953), I unfortunately and promptly forgot the details. This left me with an uneasy feeling that the proof must be long or complicated, but in fact it is so short and simple that it may be of interest to casual readers. The version below uses CPL, but not in any essential way.

Suppose T[R] is a Boolean function taking a routine (or program) R with no formal or free variables as its argument and that for all R, T[R] — True if R terminates if run and that T[R] = False if R does not terminate. Consider the routine P defined as follows

```
rec routine P
  §L:if T[P] go to L
    Return §
```

If T[P] = True the routine P will loop, and it will only terminate if T[P] = False. In each case T[P] has exactly the wrong value, and this contradiction shows that the function T cannot exist.

Yours faithfully,
C. STRACHEY.

Churchill College,
Cambridge.

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

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.

Ĥ.q0 ⟨Ĥ₁⟩ ⊢* Ĥ.qx ⟨Ĥ₁⟩ ⟨Ĥ₂⟩ ⊢* Ĥ.qy ∞
if the simulated ⟨Ĥ₁⟩ applied to ⟨Ĥ₂⟩ halts, and

Ĥ.q0 ⟨Ĥ₁⟩ ⊢* Ĥ.qx ⟨Ĥ₁⟩ ⟨Ĥ₂⟩ ⊢* Ĥ.qn
if the simulated ⟨Ĥ₁⟩ applied to ⟨Ĥ₂⟩ does not halt

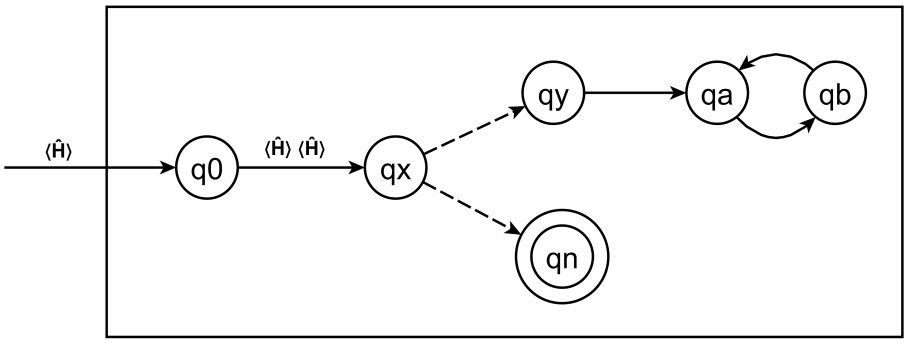


Figure 12.3 Turing Machine Ĥ applied to ⟨Ĥ⟩

When we define Ĵ to be exactly like Ĥ except that it has a UTM at Ĵ.qx instead of a simulating halt decider then we can see that Ĵ applied to ⟨Ĵ⟩ never halts.
There is an infinite cycle from Ĵ.qx to Ĵ.q0.

Ĵ.q0 ⟨Ĵ⟩ ⊢* Ĵ.qx ⟨Ĵ⟩ ⟨Ĵ⟩ ⊢* Ĵ.qn
Ĵ₀.q0 copies its input ⟨Ĵ₁⟩ to ⟨Ĵ₂⟩ then Ĵ₀.qx simulates Ĵ₁ with the ⟨Ĵ₂⟩ copy then
Ĵ₁.q0 copies its input ⟨Ĵ₂⟩ to ⟨Ĵ₃⟩ then Ĵ₁.qx simulates Ĵ₂ with the ⟨Ĵ₃⟩ copy then
Ĵ₂.q0 copies its input ⟨Ĵ₃⟩ to ⟨Ĵ₄⟩ then Ĵ₂.qx simulates Ĵ₃ with the ⟨Ĵ₄⟩ copy then ...

From this we can conclude that while the simulating halt decider at Ĥ.qx remains in pure simulation mode (thus not aborting the simulation of its input) ⟨Ĥ₁⟩ applied to ⟨Ĥ₂⟩ never halts.

This is expressed in figure 12.4 as a cycle from qx to q0 to qx.

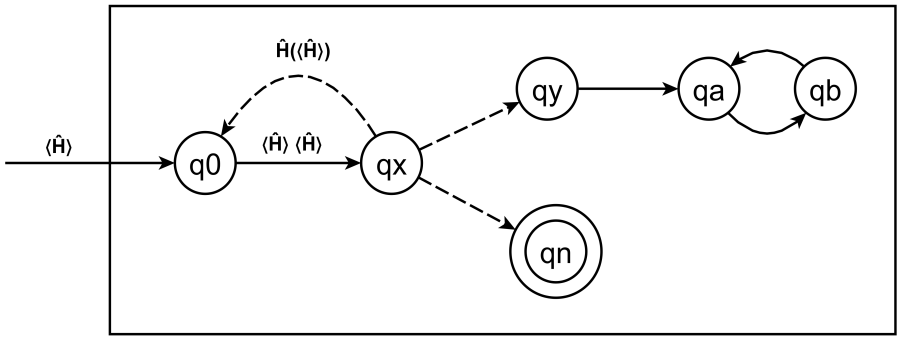


Figure 12.4 Turing Machine Ĥ applied to ⟨Ĥ⟩ input

> the Turing machine halting problem. Simply stated, the problem is: given the description of a Turing machine M and an input w, does M, when started in the initial configuration q0w, perform a computation that eventually halts? (Linz:1990:317).

In order to show that the above definition has been satisfied we only have to show that halt decider Ĥ.qx does correctly decide whether or not its input description ⟨Ĥ₁⟩ of a Turing machine would halt on its input ⟨Ĥ₂⟩.

Key halt deciding theorem that applies to every simulating halt decider (SHD):
(a) Every input to a SHD that never stops running unless its simulation is aborted, or
(b) Every input that never stops running while the SHD is in pure simulation mode.
Is an input that is correctly decided as never halting.

Next we examine the behavior of Ĥ applied to its own Turing machine description: ⟨Ĥ⟩ when the halt decider at Ĥ.qx bases its halt status decision on simulating its input.

Turing machine Ĥ is applied to its input ⟨Ĥ⟩. It copies this input such that this input and the copy of this input become the first and second parameters to the simulating halt decider at Ĥ.qx. When Ĥ.qx ⟨Ĥ⟩ ⟨Ĥ⟩ decides that the simulation of its first parameter on the input of its second parameter never halts it correctly transitions to its own final state of Ĥ.qn.

Within the hypothesis that the internal halt decider embedded within Ĥ simulates its input when Ĥ is applied to its own Turing machine description ⟨Ĥ⟩ then we can see that this derives infinitely nested simulation that must be aborted. Ĥ applied to ⟨Ĥ⟩ specifies an infinite cycle from Ĥ.qx to Ĥ.q0 all the time that Ĥ.qx remains a pure simulator of its input.

The fact that Ĥ applied to ⟨Ĥ⟩ transitions to its final state of Ĥ.qn and halts does not nullify the fact that Ĥ.qx ⟨Ĥ⟩ ⟨Ĥ⟩ correctly decides that its input never halts. Distinctly different computations can have different behavior without contradiction.

The execution of Ĥ.qx is the outer-most instance of what would otherwise be an infinite set of nested simulations. It is the only instance of Ĥ.qx that is not under the dominion of another instance of Ĥ.qx. This makes this outermost instance computationally distinct from the inner instances.

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)

Theorem 12.1

There does not exist any Turing machine H that behaves as required by Definition 12.1. The halting problem is therefore undecidable.

Proof: We assume the contrary, namely that there exists an algorithm, and consequently some Turing machine H, that solves the halting problem. The input to H will be the description (encoded in some form) of M, say w_M, as well as the input w. The requirement is then that, given any (w_M, w), the Turing machine H will halt with either a yes or no answer. We achieve this by asking that H halt in one of two corresponding final states, say, q_y or q_n. The situation can be visualized by a block diagram like Figure 12.1. The intent of this diagram is to indicate that, if M is started in state q_0 with input (w_M, w), it will eventually halt in state q_y or q_n. As required by Definition 12.1, we want H to operate according to the following rules:

$$q_0 w_M w \overset{*}{\vdash}_H x_1 q_y x_2,$$

if M applied to w halts, and

$$q_0 w_M w \overset{*}{\vdash}_H y_1 q_n y_2,$$

if M applied to w does not halt.

Figure 12.1

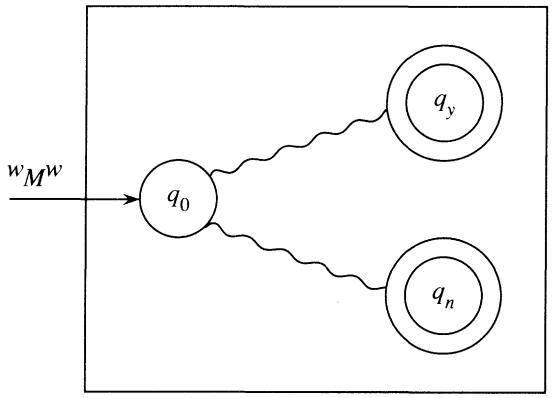
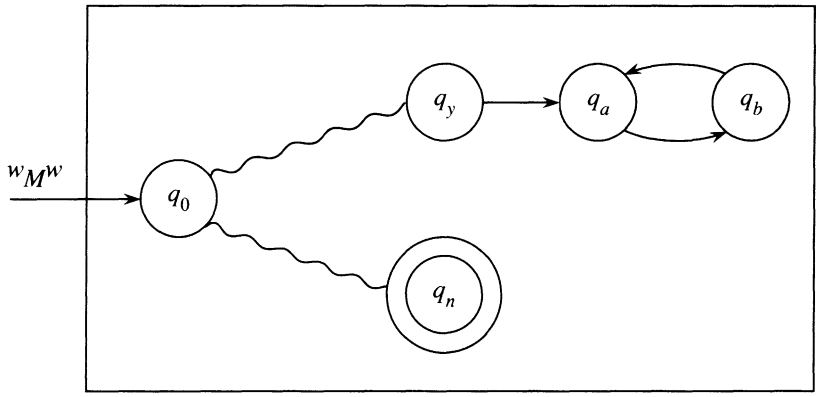


Figure 12.2



Next, we modify H to produce a Turing machine H' with the structure shown in Figure 12.2. With the added states in Figure 12.2 we want to convey that the transitions between state q_y and the new states q_a and q_b are to be made, regardless of the tape symbol, in such a way that the tape remains unchanged. The way this is done is straightforward. Comparing H and H' we see that, in situations where H reaches q_y and halts, the modified machine H' will enter an infinite loop. Formally, the action of H' is described by

$$q_0 w_M w \vdash^*_{H'} \infty,$$

if M applied to w halts, and

$$q_0 w_M w \vdash^*_{H'} y_1 q_n y_2,$$

if M applied to w does not halt.

From H' we construct another Turing machine $\hat{H}$. This new machine takes as input w_M, copies it, and then behaves exactly like H'. Then the action of $\hat{H}$ is such that

$$q_0 w_M \vdash^*_{\hat{H}} q_0 w_M w_M \vdash^*_{\hat{H}} \infty,$$

if M applied to w_M halts, and

$$q_0 w_M \vdash^*_{\hat{H}} q_0 w_M w_M \vdash^*_{\hat{H}} y_1 q_n y_2,$$

if M applied to w_M does not halt.

Now $\hat{H}$ is a Turing machine, so that it will have some description in Σ^*, say $\hat{w}$. This string, in addition to being the description of $\hat{H}$ can also be used as input string. We can therefore legitimately ask what would happen if $\hat{H}$ is applied to $\hat{w}$. From the above, identifying M with $\hat{H}$, we get

$$q_0\hat{w} \vdash^*_{\hat{H}} \infty,$$

if $\hat{H}$ applied to $\hat{w}$ halts, and

$$q_0\hat{w} \vdash^*_{\hat{H}} y_1 q_n y_2,$$

if $\hat{H}$ applied to $\hat{w}$ 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. ■