Halting problem undecidability and infinitely nested simulation

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 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.

H acts as a pure x86 simulator until its input demonstrates non-halting behavior. It is common knowledge that when-so-ever the pure simulation of the machine description of a machine never halts on its input that this logically entails that this machine never halts on its input. This proves that H uses the same halting criteria as the halting problem.

Because H acts as a pure simulator of its input until after it makes its halt status decision we know that the behavior of H cannot possibly have any effect on the behavior of P thus the behavior of H can be totally ignored in any halt status decision. This eliminates the pathological self-reference of the halting problem proof counter-example templates making them decidable.

In the concrete example shown below a simulating halt decider is based on a x86 emulator. In the Turing machine model it is based on a Universal Turing Machine (UTM). In both of these cases the input is simulated one instruction at a time. Then the stored execution trace is compared to patterns of behavior that never halt.

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 to the confounding input.

```
procedure compute_g(i):
  if f(i, i) == 0 then
    return 0
  else
    loop forever     // (Wikipedia:Halting Problem)
```

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.

The above pseudo-code is concretely implemented in the C programming language and fully executable in the x86utm operating system.

```
// Simplified Linz Ĥ (Linz:1990:319)
void P(u32 x)
{
  u32 Input_Halts = H(x, x);
  if (Input_Halts)
    HERE: goto HERE;
}

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

When a simulating partial halt decider H is applied to a simplified concrete example P of the Peter Linz Ĥ template (shown above) the details of this process show that P(P) is a computation that never halts unless it is aborted at some point. This same reasoning is then applied to the actual Ĥ(⟨Ĥ⟩) computation of the Peter Linz proof.

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 does not halt.

A simulating halt decider must abort the simulation of every input that never halts. For H to recognize the infinitely repeating pattern of P it only needs to see that same thing that humans see when they examine the x86 execution trace of the simulation of P. All of these details including the complete x86 execution trace of P(P) is provided below.

To anchor these ideas in a very simple concrete example we show how H decides that an infinite loop never halts and then show how H decides that an function calling itself in infinite recursion would be decided as non-halting.

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)
```

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

```
// Simplified Linz Â (Linz:1990:319)
void P(u32 x)
{
  u32 Input_Halts = H(x, x);
  if (Input_Halts)
    HERE: goto HERE;
}

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

When a simulating halt decider only simulates its input until it detects that its input exhibits non-halting behavior then we can know that this simulating halt decider has no effect what-so-ever on the behavior of this input.

This also means that while a simulating halt decider is examining the behavior of its input it can safely ignore its own behavior. When this simulating halt decider does detect an infinite execution behavior pattern then it can correctly stop simulating its input and report that its input does not halt.

Every H only acts as a pure x86 emulator until some P has demonstrated that it will never halt unless it is aborted.

Once it is fully understood that any computation that never halts unless its simulation is aborted is the correct non-halting criteria then it can be understood that H(P,P)==0 correct.

It can be easily verified that if no H(n) ever aborts any P(m) that P(P) will never halt. If any H(n) must abort any P(m) then this H(n) does correctly decide that this P(m) does not halt.

```
_P()
[00000b1a](01)  55              push ebp
[00000b1b](02)  8bec            mov ebp,esp
[00000b1d](01)  51              push ecx
[00000b1e](03)  8b4508          mov eax,[ebp+08]
[00000b21](01)  50              push eax        // 2nd Param
[00000b22](03)  8b4d08          mov ecx,[ebp+08]
[00000b25](01)  51              push ecx        // 1st Param
[00000b26](05)  e81ffeffff      call 0000094a   // call H
[00000b2b](03)  83c408          add esp,+08
[00000b2e](03)  8945fc          mov [ebp-04],eax
[00000b31](04)  837dfc00        cmp dword [ebp-04],+00
[00000b35](02)  7402            jz 00000b39
[00000b37](02)  ebfe            jmp 00000b37
[00000b39](02)  8be5            mov esp,ebp
[00000b3b](01)  5d              pop ebp
[00000b3c](01)  c3              ret
Size in bytes:(0035) [00000b3c]
```

```
_main()
[00000bda](01)  55              push ebp
[00000bdb](02)  8bec            mov ebp,esp
[00000bdd](01)  51              push ecx
[00000bde](05)  681a0b0000      push 00000b1a  // push address of P
[00000be3](05)  681a0b0000      push 00000b1a  // push address of P
[00000be8](05)  e85dfdffff      call 0000094a  // call H
[00000bed](03)  83c408          add esp,+08
[00000bf0](03)  8945fc          mov [ebp-04],eax
[00000bf3](03)  8b45fc          mov eax,[ebp-04]
[00000bf6](01)  50              push eax
[00000bf7](05)  683b030000      push 0000033b
[00000bfc](05)  e869f7ffff      call 0000036a
[00000c01](03)  83c408          add esp,+08
[00000c04](02)  33c0            xor eax,eax
[00000c06](02)  8be5            mov esp,ebp
[00000c08](01)  5d              pop ebp
[00000c09](01)  c3              ret
Size in bytes:(0048) [00000c09]
```

Execution Trace of H(P,P)

```
 machine   stack     stack     machine    assembly
 address   address   data      code       language
 ========  ========  ========  =========  ==============
[00000bda][00101647][00000000] 55         push ebp
[00000bdb][00101647][00000000] 8bec       mov ebp,esp
[00000bdd][00101643][00000000] 51         push ecx
[00000bde][0010163f][00000b1a] 681a0b0000 push 00000b1a // push P
[00000be3][0010163b][00000b1a] 681a0b0000 push 00000b1a // push P
[00000be8][00101637][00000bed] e85dfdffff call 0000094a // call H

Begin Local Halt Decider Simulation at Machine Address:b1a
[00000b1a][002116e7][002116eb] 55         push ebp
[00000b1b][002116e7][002116eb] 8bec       mov ebp,esp
[00000b1d][002116e3][002016b7] 51         push ecx
[00000b1e][002116e3][002016b7] 8b4508     mov eax,[ebp+08]
[00000b21][002116df][00000b1a] 50         push eax      // push P
[00000b22][002116df][00000b1a] 8b4d08     mov ecx,[ebp+08]
[00000b25][002116db][00000b1a] 51         push ecx      // push P
[00000b26][002116d7][00000b2b] e81ffeffff call 0000094a // call H
[00000b1a][0025c10f][0025c113] 55         push ebp
[00000b1b][0025c10f][0025c113] 8bec       mov ebp,esp
[00000b1d][0025c10b][0024c0df] 51         push ecx
[00000b1e][0025c10b][0024c0df] 8b4508     mov eax,[ebp+08]
[00000b21][0025c107][00000b1a] 50         push eax      // push P
[00000b22][0025c107][00000b1a] 8b4d08     mov ecx,[ebp+08]
[00000b25][0025c103][00000b1a] 51         push ecx      // push P
[00000b26][0025c0ff][00000b2b] e81ffeffff call 0000094a // call H
Local Halt Decider: Infinite Recursion Detected Simulation Stopped
```

In the above 16 instructions of the simulation of P(P) we can see that the first 8 instructions of P are repeated. The end of this sequence of 8 instructions P calls H with its own machine address as the parameters to H: 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 8 simulated instructions of P have any escape from their infinitely repeating behavior pattern. When H recognizes this infinitely repeating pattern it aborts its simulation of P(P) and reports that its input: (P,P) would never halt on its input.

```
[00000bed][00101643][00000000] 83c408     add esp,+08
[00000bf0][00101643][00000000] 8945fc     mov [ebp-04],eax
[00000bf3][00101643][00000000] 8b45fc     mov eax,[ebp-04]
[00000bf6][0010163f][00000000] 50         push eax
[00000bf7][0010163b][0000033b] 683b030000 push 0000033b
[00000bfc][0010163b][0000033b] e869f7ffff call 0000036a
Input_Halts = 0
[00000c01][00101643][00000000] 83c408     add esp,+08
[00000c04][00101643][00000000] 33c0       xor eax,eax
[00000c06][00101647][00000000] 8be5       mov esp,ebp
[00000c08][0010164b][00100000] 5d         pop ebp
[00000c09][0010164f][00000080] c3         ret
Number_of_User_Instructions(33)
Number of Instructions Executed(26452)
```

This is the sound deductive inference (proof) that H(P,P)==0 is correct.

Premise(1) (Axiom) When the pure simulation of the machine description ⟨P⟩ of a machine P on its input I never halts we know that P(I) never halts.

Every input that never halts while the simulating halt decider remains a pure simulator is an input that never halts.

Premise(2) (verified fact) The simulation of the input to H(P,P) never halts without being aborted is a verified fact on the basis of its x86 execution trace.

When the simulator determines whether or not it must abort the simulation of its input based on the behavior of its input the simulator only acts as an x86 emulator thus has no effect on the behavior of its input. This allows the simulator to always ignore its own behavior. H simply screens out its own address range when making its halt status decision.

When the simulating halt decider ignores its own behavior in its halt status analysis of the execution trace of its input this eliminates pathological self-reference from the conventional halting problem counter-example templates.

Conclusion(3) From the above true premises it necessarily follows that simulating halt decider H correctly reports that its input: (P,P) never halts.

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

```
void P(u32 x)
{
  u32 Input_Halts = H(x, x);
  if (Input_Halts)
    HERE: goto HERE;
}

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

_P()
[00000b25](01)  55              push ebp
[00000b26](02)  8bec            mov ebp,esp
[00000b28](01)  51              push ecx
[00000b29](03)  8b4508          mov eax,[ebp+08]
[00000b2c](01)  50              push eax
[00000b2d](03)  8b4d08          mov ecx,[ebp+08]
[00000b30](01)  51              push ecx
[00000b31](05)  e81ffeffff      call 00000955  // call H
[00000b36](03)  83c408          add esp,+08
[00000b39](03)  8945fc          mov [ebp-04],eax
[00000b3c](04)  837dfc00        cmp dword [ebp-04],+00
[00000b40](02)  7402            jz 00000b44
[00000b42](02)  ebfe            jmp 00000b42
[00000b44](02)  8be5            mov esp,ebp
[00000b46](01)  5d              pop ebp
[00000b47](01)  c3              ret
Size in bytes:(0035) [00000b47]

_main()
[00000c05](01)  55              push ebp
[00000c06](02)  8bec            mov ebp,esp
[00000c08](05)  68250b0000      push 00000b25  // push address of P
[00000c0d](05)  e813ffffff      call 00000b25  // call P
[00000c12](03)  83c404          add esp,+04
[00000c15](02)  33c0            xor eax,eax
[00000c17](01)  5d              pop ebp
[00000c18](01)  c3              ret
Size in bytes:(0020) [00000c18]
```

Execution Trace of P(P)

```
 machine   stack     stack     machine    assembly
 address   address   data      code       language
 ========  ========  ========  =========  =============
[00000c05][0010165e][00000000] 55         push ebp
[00000c06][0010165e][00000000] 8bec       mov ebp,esp
[00000c08][0010165a][00000b25] 68250b0000 push 00000b25
[00000c0d][00101656][00000c12] e813ffffff call 00000b25  // P0
[00000b25][00101652][0010165e] 55         push ebp
[00000b26][00101652][0010165e] 8bec       mov ebp,esp
[00000b28][0010164e][00000000] 51         push ecx
[00000b29][0010164e][00000000] 8b4508     mov eax,[ebp+08]
[00000b2c][0010164a][00000b25] 50         push eax
[00000b2d][0010164a][00000b25] 8b4d08     mov ecx,[ebp+08]
[00000b30][00101646][00000b25] 51         push ecx
[00000b31][00101642][00000b36] e81ffeffff call 00000955  // H0
```

```
Begin Local Halt Decider Simulation at Machine Address:b25
[00000b25][002116fe][00211702] 55          push ebp       // P1
[00000b26][002116fe][00211702] 8bec        mov ebp,esp
[00000b28][002116fa][002016ce] 51          push ecx
[00000b29][002116fa][002016ce] 8b4508      mov eax,[ebp+08]
[00000b2c][002116f6][00000b25] 50          push eax
[00000b2d][002116f6][00000b25] 8b4d08      mov ecx,[ebp+08]
[00000b30][002116f2][00000b25] 51          push ecx
[00000b31][002116ee][00000b36] e81ffeffff  call 00000955  // H1
[00000b25][0025c126][0025c12a] 55          push ebp       // P2
[00000b26][0025c126][0025c12a] 8bec        mov ebp,esp
[00000b28][0025c122][0024c0f6] 51          push ecx
[00000b29][0025c122][0024c0f6] 8b4508      mov eax,[ebp+08]
[00000b2c][0025c11e][00000b25] 50          push eax
[00000b2d][0025c11e][00000b25] 8b4d08      mov ecx,[ebp+08]
[00000b30][0025c11a][00000b25] 51          push ecx
[00000b31][0025c116][00000b36] e81ffeffff  call 00000955  // H2
Local Halt Decider: Infinite Recursion Detected Simulation Stopped
```

In the above computation (zero based addressing) H_1 aborts P_2.

```
[00000b36][0010164e][00000000] 83c408      add esp,+08
[00000b39][0010164e][00000000] 8945fc      mov [ebp-04],eax
[00000b3c][0010164e][00000000] 837dfc00    cmp dword [ebp-04],+00
[00000b40][0010164e][00000000] 7402        jz 00000b44
[00000b44][00101652][0010165e] 8be5        mov esp,ebp
[00000b46][00101656][00000c12] 5d          pop ebp
[00000b47][0010165a][00000b25] c3          ret
[00000c12][0010165e][00000000] 83c404      add esp,+04
[00000c15][0010165e][00000000] 33c0        xor eax,eax
[00000c17][00101662][00100000] 5d          pop ebp
[00000c18][00101666][00000098] c3          ret
Number_of_User_Instructions(39)
Number of Instructions Executed(26459)
```

The computation **int main() { P(P); }** specifies infinitely nested simulation.
Unless one of its infinitely nested simulations is aborted **int main() { P(P); }** never halts.

In the computation **int main() { P(P); }** when no P ever halts unless some H aborts some P this proves beyond all possible doubt that P(P) specifies an infinitely recursive chain of invocations.

The computation **int main() { P(P); }** calls H(P,P) which is the first invocation of an infinite chain of invocations. Whenever P calls H(P,P) H must abort its simulation of P.

In the computation **int main() { P(P); }** the third element of the infinite chain of invocations is terminated. The only reason that any P ever halts is that some H aborted some P. This proves (axiomatically) that P(P) really does specify an infinite invocation chain.

It can be easily verified that if no H ever aborts any P that P(P) will never halt. If any H must abort any P then this H does correctly decide that this P does not halt.

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 unless X() stops it.

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. The halt decider is embedded at state Ĥ.qx.

Ĥ.q0 wM ⊢* Ĥ.qx wM wM ⊢* Ĥ.qy ∞
if M applied to wM halts, and

Ĥ.q0 wM ⊢* Ĥ.qx wM wM ⊢* Ĥ.qn
if M applied to wM does not halt

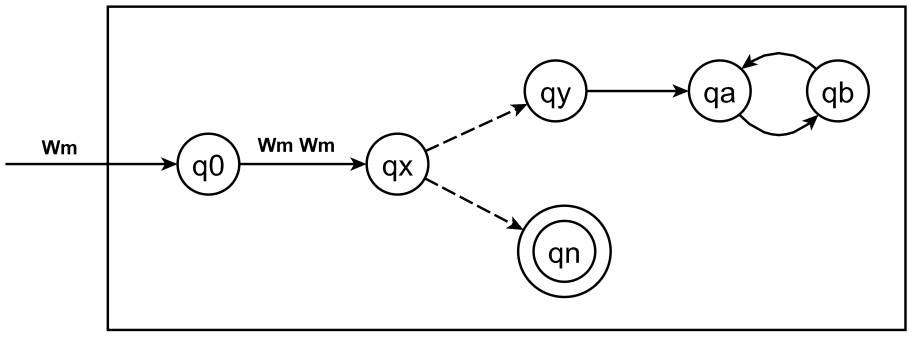


Figure 12.3 Turing Machine Ĥ

To provide a sketch of the idea of how a simulating halt decider would analyze the Peter Linz Ĥ applied to its own Turing machine description we start by examining the behavior of an ordinary UTM.

When we hypothesize that the halt decider embedded in Ĥ is simply a UTM then it seems that when the Peter Linz Ĥ is applied to its own Turing machine description ⟨Ĥ⟩ this specifies a computation that never halts.

$\hat{H}_0$.q0 copies its input $\langle \hat{H}_1 \rangle$ to $\langle \hat{H}_x \rangle$ then $\hat{H}_0$.qx simulates this input with the copy then
$\hat{H}_1$.q0 copies its input $\langle \hat{H}_2 \rangle$ to $\langle \hat{H}_y \rangle$ then $\hat{H}_1$.qx simulates this input with the copy then
$\hat{H}_2$.q0 copies its input $\langle \hat{H}_3 \rangle$ to $\langle \hat{H}_z \rangle$ then $\hat{H}_2$.qx simulates this input with the copy then ...

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

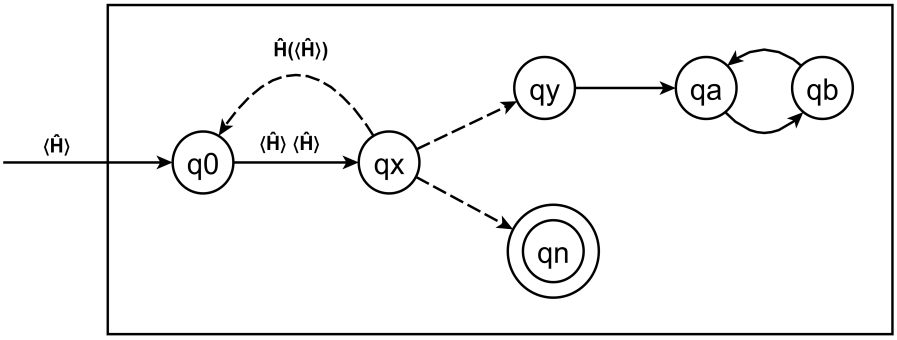


Figure 12.4 Turing Machine Ĥ applied to ⟨Ĥ⟩ input

Within the hypothesis that the internal halt decider embedded within Ĥ simulates its input Ĥ applied to its own Turing machine description ⟨Ĥ⟩ derives infinitely nested simulation, unless this simulation is aborted.

Self-Evident-Truth (premise[1])
When the pure simulation of a machine on its input never halts we know that the execution of this machine on its input never halts.

Self-Evident-Truth (premise[2])
The ⟨Ĥ⟩ ⟨Ĥ⟩ input to the embedded simulating halt decider at Ĥ.qx is pure simulation that never halts.

∴ **Sound Deductive Conclusion**
The embedded simulating halt decider at Ĥ.qx correctly decides its input: ⟨Ĥ⟩ ⟨Ĥ⟩ is a computation that never halts.

Ĥ.q0 ⟨Ĥ⟩ specifies an infinite chain of invocations that is terminated at its third invocation. The first invocation of Ĥ.qx ⟨Ĥ⟩, ⟨Ĥ⟩ is the first element of an infinite chain of invocations.

It is common knowledge that when any invocation of an infinite chain of invocations is terminated that the whole chain terminates. That the first element of this infinite chain terminates after its third element has been terminated does not entail that this first element is an actual terminating computation.

For the first element to be an actual terminating computation it must terminate without any of the elements of the infinite chain of invocations being terminated.

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

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

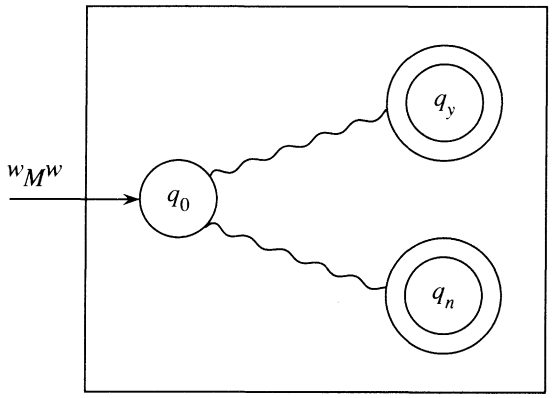
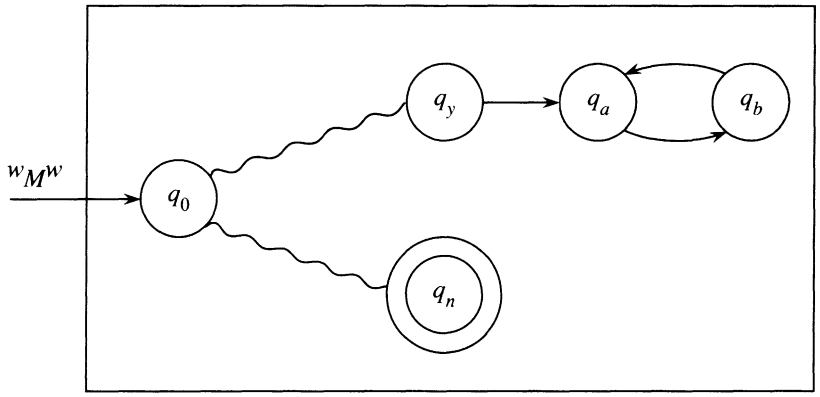


Figure 12.2



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. ■