Halting problem undecidability and infinitely nested simulation

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.

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.

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

When halting is defined as any computation that halts without ever having its simulation aborted then it is understood that simulating partial halt decider H correctly decides that its input does not halt on the simplified version of the Linz Ĥ (shown below).

The x86utm operating system was created so that the halting problem could be examined concretely in the high level language of C. UTM tape elements are 32-bit unsigned integers. H examines the behavior of the x86 emulation of its input. 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 H is a Universal Turing Machine (UTM) that has been adapted to decide whether or not its input halts. H simulates the execution of its inputs exactly as if it was simply a UTM. After H simulates each instruction of its input it examines the full execution trace of this input. When an execution trace matches an infinite execution behavior pattern H aborts the simulation of this input and reports that this input does not halt.

This halt deciding principle overcomes the conventional halting problem proofs:
It is self-evidently true that every computation that never halts unless its simulation is aborted <is> a non-halting computation even after its simulation has been aborted.

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

int main()
{
  H((u32)P, (u32)P);
}
```

A simple example of simulating halt decider H correctly deciding that an infinite loop does not halt is provided on page 6 below.

So can you describe the exact conditions which cause H to detect a repeated state and declare that the program will never terminate?

Anyone knowing the x86 language very well can examine the two x86 execution traces of `H(P,P)` and directly see for themselves that it is completely certain that the input to `H(P,P)` would never halt unless the simulation of this input its was aborted. Every time that P calls `H(P,P)` H creates and executes another nested simulation of P.

H analyzes the (currently updated) stored execution trace of its simulation of P(P) after it simulates each instruction of input (P, P). 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.

P continues to call H with its own machine address endlessly repeating its first 8 lines of x86 code. There is no code that can escape this endlessly repeating cycle in these 8 lines of x86 code unless H aborts its simulation of P(P).

```
_P()
[00000af8](01)  55          push ebp
[00000af9](02)  8bec        mov ebp,esp
[00000afb](01)  51          push ecx
[00000afc](03)  8b4508      mov eax,[ebp+08]
[00000aff](01)  50          push eax
[00000b00](03)  8b4d08      mov ecx,[ebp+08]
[00000b03](01)  51          push ecx
[00000b04](05)  e81ffeffff  call 00000928 // Machine address of H
[00000b09](03)  83c408      add esp,+08
[00000b0c](03)  8945fc      mov [ebp-04],eax
[00000b0f](04)  837dfc00    cmp dword [ebp-04],+00
[00000b13](02)  7402        jz 00000b17
[00000b15](02)  ebfe        jmp 00000b15
[00000b17](02)  8be5        mov esp,ebp
[00000b19](01)  5d          pop ebp
[00000b1a](01)  c3          ret
Size in bytes:(0035) [00000b1a]

_main()
[00000b28](01)  55          push ebp
[00000b29](02)  8bec        mov ebp,esp
[00000b2b](05)  68f80a0000  push 00000af8 // Machine address of P
[00000b30](05)  68f80a0000  push 00000af8 // Machine address of P
[00000b35](05)  e8eefdffff  call 00000928 // Machine address of H
[00000b3a](03)  83c408      add esp,+08
[00000b3d](02)  33c0        xor eax,eax
[00000b3f](01)  5d          pop ebp
[00000b40](01)  c3          ret
Size in bytes:(0025) [00000b40]
```

Columns
(1) Machine address of instruction
(2) Machine address of top of stack
(3) Value of top of stack after instruction executed
(4) Machine language bytes
(5) Assembly language text

```
==============================
[00000b28][001014de][00000000] 55              push ebp
[00000b29][001014de][00000000] 8bec            mov ebp,esp
[00000b2b][001014da][00000af8] 68f80a0000      push 00000af8 // P
[00000b30][001014d6][00000af8] 68f80a0000      push 00000af8 // P
[00000b35][001014d2][00000b3a] e8eefdffff      call 00000928 // H
```

Begin Local Halt Decider Simulation at Machine Address:af8

```
[00000af8][0021157e][00211582] 55              push ebp
[00000af9][0021157e][00211582] 8bec            mov ebp,esp
[00000afb][0021157a][0020154e] 51              push ecx
[00000afc][0021157a][0020154e] 8b4508          mov eax,[ebp+08]
[00000aff][00211576][00000af8] 50              push eax      // P
[00000b00][00211576][00000af8] 8b4d08          mov ecx,[ebp+08]
[00000b03][00211572][00000af8] 51              push ecx      // P
[00000b04][0021156e][00000b09] e81ffeffff      call 00000928 // H
```

The above eight instructions of P are repeated here

```
[00000af8][0025bfa6][0025bfaa] 55              push ebp
[00000af9][0025bfa6][0025bfaa] 8bec            mov ebp,esp
[00000afb][0025bfa2][0024bf76] 51              push ecx
[00000afc][0025bfa2][0024bf76] 8b4508          mov eax,[ebp+08]
[00000aff][0025bf9e][00000af8] 50              push eax      // P
[00000b00][0025bf9e][00000af8] 8b4d08          mov ecx,[ebp+08]
[00000b03][0025bf9a][00000af8] 51              push ecx      // P
[00000b04][0025bf96][00000b09] e81ffeffff      call 00000928 // H
```

Local Halt Decider: Infinite Recursion Detected Simulation Stopped

In column 3 we can see that P pushed its own machine address 0xaf8 onto the stack in the pair of push instructions preceding the call to H(P,P) at 0x928. The first eight x86 instructions of P are repeated in an infinitely repeating cycle. P continues to call H in what is essentially infinite recursion.

The execution trace after the call to H only shows the first instruction of P because H ignores its own execution trace. The first thing that H does is simulate its input. So when P calls H all we see is H simulating P.

```
[00000b3a][001014de][00000000] 83c408          add esp,+08
[00000b3d][001014de][00000000] 33c0            xor eax,eax
[00000b3f][001014e2][00100000] 5d              pop ebp
[00000b40][001014e6][00000060] c3              ret
Number_of_User_Instructions(25)
Number of Instructions Executed(26445)
```

When a chain of function calls specifies infinite recursion is broken by a simulating halt decider aborting the simulation of any one of these function calls, then the whole chain of function calls is correctly decided to specify a computation that does not halt.

This same reasoning applies to the computation: `P((u32)P);` when P() invokes H() with its own machine address, this is the first invocation of an infinite chain of invocations. As the first element of an infinite chain of invocations where the third element of this chain is aborted the whole chain is understood to specify an infinite invocation sequence.

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

int main()    // The input to P is forced to halt.
{             // The invocation of H(P,P) from this P(P)
  P((u32)P);  // is the first element of an infinite
}             // chain of invocations of H(P,P)
```

```
_P()
[00000af8](01)  55          push ebp
[00000af9](02)  8bec        mov ebp,esp
[00000afb](01)  51          push ecx
[00000afc](03)  8b4508      mov eax,[ebp+08]
[00000aff](01)  50          push eax
[00000b00](03)  8b4d08      mov ecx,[ebp+08]
[00000b03](01)  51          push ecx
[00000b04](05)  e81ffeffff call 00000928 // Machine address of H
[00000b09](03)  83c408      add esp,+08
[00000b0c](03)  8945fc      mov [ebp-04],eax
[00000b0f](04)  837dfc00    cmp dword [ebp-04],+00
[00000b13](02)  7402        jz 00000b17
[00000b15](02)  ebfe        jmp 00000b15
[00000b17](02)  8be5        mov esp,ebp
[00000b19](01)  5d          pop ebp
[00000b1a](01)  c3          ret
Size in bytes:(0035) [00000b1a]


_main()
[00000b28](01)  55          push ebp
[00000b29](02)  8bec        mov ebp,esp
[00000b2b](05)  68f80a0000  push 00000af8 // Machine address of P
[00000b30](05)  e8c3ffffff  call 00000af8 // Machine address of P
[00000b35](03)  83c404      add esp,+04
[00000b38](02)  33c0        xor eax,eax
[00000b3a](01)  5d          pop ebp
[00000b3b](01)  c3          ret
Size in bytes:(0020) [00000b3b]
```

Columns

(1) Machine address of instruction
(2) Machine address of top of stack
(3) Value of top of stack after instruction executed
(4) Machine language bytes
(5) Assembly language text

```
==============================
[00000b28][001014cf][00000000] 55          push ebp
[00000b29][001014cf][00000000] 8bec        mov ebp,esp
[00000b2b][001014cb][00000af8] 68f80a0000  push 00000af8   // P
[00000b30][001014c7][00000b35] e8c3ffffff  call 00000af8   // P
[00000af8][001014c3][001014cf] 55          push ebp
[00000af9][001014c3][001014cf] 8bec        mov ebp,esp
[00000afb][001014bf][00000000] 51          push ecx
[00000afc][001014bf][00000000] 8b4508      mov eax,[ebp+08]
[00000aff][001014bb][00000af8] 50          push eax
[00000b00][001014bb][00000af8] 8b4d08      mov ecx,[ebp+08]
[00000b03][001014b7][00000af8] 51          push ecx
[00000b04][001014b3][00000b09] e81ffeffff  call 00000928   // H
```

Begin Local Halt Decider Simulation at Machine Address:af8

```
[00000af8][0021156f][00211573] 55          push ebp
[00000af9][0021156f][00211573] 8bec        mov ebp,esp
[00000afb][0021156b][0020153f] 51          push ecx
[00000afc][0021156b][0020153f] 8b4508      mov eax,[ebp+08]
[00000aff][00211567][00000af8] 50          push eax        // P
[00000b00][00211567][00000af8] 8b4d08      mov ecx,[ebp+08]
[00000b03][00211563][00000af8] 51          push ecx        // P
[00000b04][0021155f][00000b09] e81ffeffff  call 00000928   // H
```

The above eight instructions of P are repeated here

```
[00000af8][0025bf97][0025bf9b] 55         push ebp
[00000af9][0025bf97][0025bf9b] 8bec       mov ebp,esp
[00000afb][0025bf93][0024bf67] 51         push ecx        // P
[00000afc][0025bf93][0024bf67] 8b4508     mov eax,[ebp+08]
[00000aff][0025bf8f][00000af8] 50         push eax        // P
[00000b00][0025bf8f][00000af8] 8b4d08     mov ecx,[ebp+08]
[00000b03][0025bf8b][00000af8] 51         push ecx
[00000b04][0025bf87][00000b09] e81ffeffff call 00000928   // H
```

Local Halt Decider: Infinite Recursion Detected Simulation Stopped

In column 3 of the prior two push instructions we can see that P pushed its own machine address 0xaf8 onto the stack thus calling H(P,P) at 0x928 in an infinitely repeating cycle of the first eight x86 instructions of P.

The call to H from P only shows the first instruction of P because H ignores its own execution trace. The first thing that H does is simulate its input. So when P calls H all we see is H simulating P.

```
[00000b09][001014bf][00000000] 83c408     add esp,+08
[00000b0c][001014bf][00000000] 8945fc     mov [ebp-04],eax
[00000b0f][001014bf][00000000] 837dfc00   cmp dword [ebp-04],+00
[00000b13][001014bf][00000000] 7402       jz 00000b17
[00000b17][001014c3][001014cf] 8be5       mov esp,ebp
[00000b19][001014c7][00000b35] 5d         pop ebp
[00000b1a][001014cb][00000af8] c3         ret
[00000b35][001014cf][00000000] 83c404     add esp,+04
[00000b38][001014cf][00000000] 33c0       xor eax,eax
[00000b3a][001014d3][00100000] 5d         pop ebp
[00000b3b][001014d7][00000098] c3         ret
Number_of_User_Instructions(39)
Number of Instructions Executed(26459)
```

Anyone that knows the theory of computation well enough knows that any computation that never halts unless its simulation is aborted is a conventional non-halting computation: When UTM(⟨P⟩,I) never halts then we can know that P(I) never halts.

When P invoked from main calls H(P,P) this is the first element of an otherwise infinite invocation chain. Unless H aborts its simulation of P this simulation never terminates.

It is common knowledge that whenever any invocation of an infinitely recursive chain is terminated that this terminates the whole chain.

When H aborts the third element of this otherwise infinite chain the whole chain stops running including its first element.

```
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]

================================
...[00000c00][00101693][00000000](01)  55              push ebp
...[00000c01][00101693][00000000](02)  8bec            mov ebp,esp
...[00000c03][0010168f][00000000](01)  51              push ecx
...[00000c04][0010168b][00000ab0](05)  68b00a0000      push 00000ab0
...[00000c09][00101687][00000ab0](05)  68b00a0000      push 00000ab0
...[00000c0e][00101683][00000c13](05)  e84dfdffff      call 00000960
Begin Local Halt Decider Simulation at Machine Address:ab0
...[00000ab0][00211733][00211737](01)  55              push ebp
...[00000ab1][00211733][00211737](02)  8bec            mov ebp,esp
...[00000ab3][00211733][00211737](02)  ebfe            jmp 00000ab3
...[00000ab3][00211733][00211737](02)  ebfe            jmp 00000ab3
Local Halt Decider: Infinite Loop Detected Simulation Stopped
...[00000c13][0010168f][00000000](03)  83c408          add esp,+08
...[00000c16][0010168f][00000000](03)  8945fc          mov [ebp-04],eax
...[00000c19][0010168f][00000000](03)  8b45fc          mov eax,[ebp-04]
...[00000c1c][0010168b][00000000](01)  50              push eax
...[00000c1d][00101687][0000034b](05)  684b030000      push 0000034b
---[00000c22][00101687][0000034b](05)  e859f7ffff      call 00000380
Input_Would_Halt2 = 0
...[00000c27][0010168f][00000000](03)  83c408          add esp,+08
...[00000c2a][0010168f][00000000](02)  33c0            xor eax,eax
...[00000c2c][00101693][00000000](02)  8be5            mov esp,ebp
...[00000c2e][00101697][00100000](01)  5d              pop ebp
...[00000c2f][0010169b][00000050](01)  c3              ret
Number_of_User_Instructions(21)
Number of Instructions Executed(640)
```

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

When we assume that the halt decider embedded in Ĥ is simply a UTM does this define a computation that never halts when Ĥ is applied to its own Turing machine description?

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

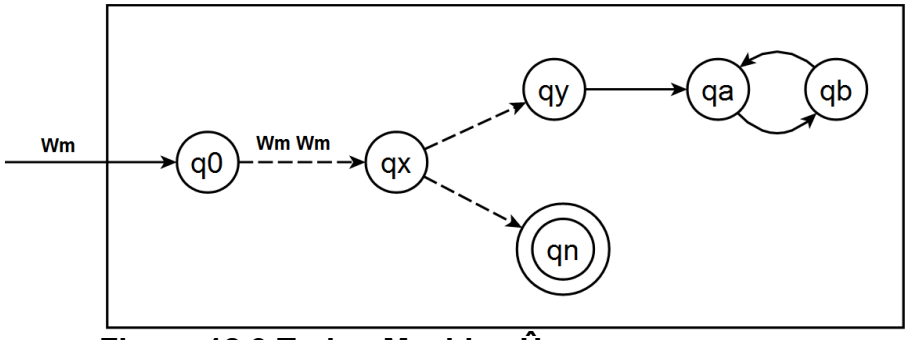


Figure 12.3 Turing Machine Ĥ

Ĥ.q0 copies its input then Ĥ.qx simulates this input with the copy then
Ĥ.q0 copies its input then Ĥ.qx simulates this input with the copy then
Ĥ.q0 copies its input then Ĥ.qx simulates this input with the copy then...
This is expressed in figure 12.4 as a cycle from qx to q0 to qx.

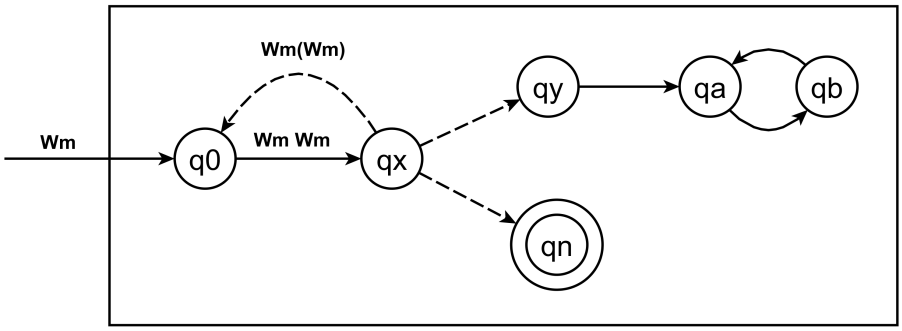


Figure 12.4 Turing Machine Ĥ

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

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

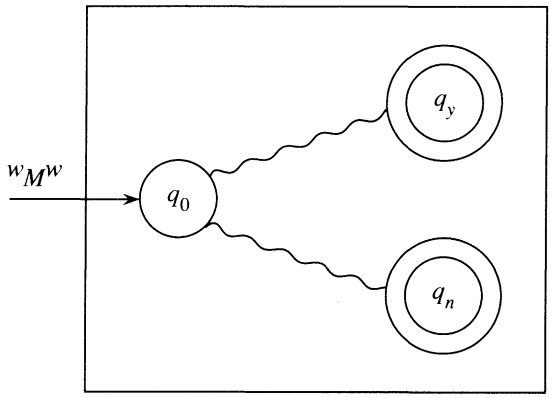
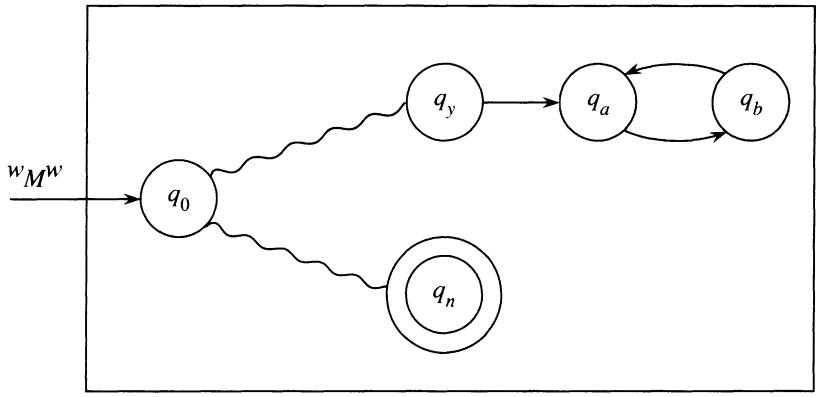


Figure 12.2



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. ■