

Halting problem undecidability and infinite recursion

The simplest way to define a halt decider is to make a program that runs its input to see what it does. In technical terms this would be a universal Turing machine (UTM) that has been adapted to become a halt decider.

The UTM would simulate its input and maintain an execution trace of this simulation as a pure function of this input. The halt decider would be invoked by the UTM immediately after it simulates each instruction. The halt decider is a pure function of its input execution trace.

This adapted UTM simply simulates the execution of its input until its input halts on its own or its halt decider determines that its input would never halt on its own and stops simulating it. The following finite string has the property of infinite execution even after it halts because the halt decider stopped simulating it:

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

Every (at least partial) halt decider that decides the halting status of its input on the basis of its examination of the execution trace of its own simulation of this input would correctly decide the conventional halting problem undecidability proof counter-examples would not halt. (see Appendix: Key basis of the {Sipser, Kozen, Linz} halting problem undecidability proofs).

The fact that the only reason these input programs halted was that their simulation was aborted by the halt decider proves that they are non-halting computations.

The x86utm operating system was created so that halt deciders written in the C programming language would be computationally equivalent to the execution of actual Turing machines. These (at least partial) halt deciders base their halting decision on examining the execution trace of the x86 machine language of their input. The input is the COFF object file output of a C compiler and is directly executed by the x86 emulator.

The halt decider Halts() determines that H_Hat() called it in infinite recursion, on the basis of the x86 machine language execution trace shown below, stops simulating this input, and decides not halting on the basis that its input would not halt.

Partial halt decider architectural design

Every partial halt decider H that bases its halting decision on simulating its input P until criteria (1)(2)(3) are met by the execution trace of P correctly decides not halting on input P.

If the execution trace of function P() [i.e. the input to H] shows:

- (1) Partial halt decider H is called twice in sequence from the same machine address of P. ("in sequence" means that the block of instructions containing the call to H repeats).
- (2) with the same machine address parameters to H.
- (3) with no conditional branch or indexed jump instructions in P.

Then H correctly decides not halting on P.

```

void H_Hat(u32 P)
{
    u32 Input_Halts = Halts(P, P);
    if (Input_Halts)
        HERE: goto HERE;
}

int main()
{
    u32 Input_Would_Halt = Halts((u32)H_Hat, (u32)H_Hat);
    Output("Input_Would_Halt = ", Input_Would_Halt);
}

```

```

_H_Hat()
[0000b1f] (01) 55          push ebp
[0000b20] (02) 8bec        mov ebp,esp
[0000b22] (01) 51          push ecx
[0000b23] (03) 8b4508      mov eax,[ebp+08]
[0000b26] (01) 50          push eax
[0000b27] (03) 8b4d08      mov ecx,[ebp+08]
[0000b2a] (01) 51          push ecx
[0000b2b] (05) e2ffeffff    call 000095f
[0000b30] (03) 83c408      add esp,+08
[0000b33] (03) 8945fc      mov [ebp-04],eax
[0000b36] (04) 837dfc00     cmp dword [ebp-04],+00
[0000b3a] (02) 7402          jz 0000b3e
[0000b3c] (02) ebfe          jmp 0000b3c
[0000b3e] (02) 8be5          mov esp,ebp
[0000b40] (01) 5d          pop ebp
[0000b41] (01) c3          ret
Size in bytes:(0035) [0000b41]

```

```

_main()
[0000b4f] (01) 55          push ebp
[0000b50] (02) 8bec        mov ebp,esp
[0000b52] (01) 51          push ecx
[0000b53] (05) 681f0b0000    push 0000b1f
[0000b58] (05) 681f0b0000    push 0000b1f
[0000b5d] (05) e8fdfdffff    call 000095f
[0000b62] (03) 83c408      add esp,+08
[0000b65] (03) 8945fc      mov [ebp-04],eax
[0000b68] (03) 8b45fc      mov eax,[ebp-04]
[0000b6b] (01) 50          push eax
[0000b6c] (05) 682b030000    push 000032b
[0000b71] (05) e8e9f7ffff    call 000035f
[0000b76] (03) 83c408      add esp,+08
[0000b79] (02) 33c0          xor eax,eax
[0000b7b] (02) 8be5          mov esp,ebp
[0000b7d] (01) 5d          pop ebp
[0000b7e] (01) c3          ret
Size in bytes:(0048) [0000b7e]

```

Columns

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

```

01. [00000b4f] [00101542] [00000000] (01) 55          push ebp
02. [00000b50] [00101542] [00000000] (02) 8bec       mov ebp,esp
03. [00000b52] [0010153e] [00000000] (01) 51          push ecx
04. [00000b53] [0010153a] [00000b1f] (05) 681f0b0000 push 00000b1f
05. [00000b58] [00101536] [00000b1f] (05) 681f0b0000 push 00000b1f
06. [00000b5d] [00101532] [00000b62] (05) e8fdfdffff call 0000095f

```

Begin Local Halt Decider Simulation at Machine Address:b1f

```

07. [00000b1f] [002115e2] [002115e6] (01) 55          push ebp
08. [00000b20] [002115e2] [002115e6] (02) 8bec       mov ebp,esp
09. [00000b22] [002115de] [002015b2] (01) 51          push ecx
10. [00000b23] [002115de] [002015b2] (03) 8b4508     mov eax,[ebp+08]
11. [00000b26] [002115da] [00000b1f] (01) 50          push eax
12. [00000b27] [002115da] [00000b1f] (03) 8b4d08     mov ecx,[ebp+08]
13. [00000b2a] [002115d6] [00000b1f] (01) 51          push ecx
14. [00000b2b] [002115d2] [00000b30] (05) e82ffeffff call 0000095f

```

```

15. [00000b1f] [0025c00a] [0025c00e] (01) 55          push ebp
16. [00000b20] [0025c00a] [0025c00e] (02) 8bec       mov ebp,esp
17. [00000b22] [0025c006] [0024bfda] (01) 51          push ecx
18. [00000b23] [0025c006] [0024bfda] (03) 8b4508     mov eax,[ebp+08]
19. [00000b26] [0025c002] [00000b1f] (01) 50          push eax
20. [00000b27] [0025c002] [00000b1f] (03) 8b4d08     mov ecx,[ebp+08]
21. [00000b2a] [0025bffe] [00000b1f] (01) 51          push ecx
22. [00000b2b] [0025bffa] [00000b30] (05) e82ffeffff call 0000095f

```

Local Halt Decider: Infinite Recursion Detected Simulation Stopped

(1) Halts is called twice in sequence from the same machine address of H_Hat.

```

14. [00000b2b] [002115d2] [00000b30] (05) e82ffeffff call 0000095f
22. [00000b2b] [0025bffa] [00000b30] (05) e82ffeffff call 0000095f

```

(2) With the same machine address of H_Hat parameters to H

(Pair of push instructions preceding the call to Halts are its input)

```

11. [00000b26] [002115da] [00000b1f] (01) 50          push eax
13. [00000b2a] [002115d6] [00000b1f] (01) 51          push ecx

19. [00000b26] [0025c002] [00000b1f] (01) 50          push eax
21. [00000b2a] [0025bffe] [00000b1f] (01) 51          push ecx

```

(3) with no conditional branch or indexed jump instructions in H_Hat

None of the machine instructions from line 07 to line 22 are this type.

```

23. [00000b62] [0010153e] [00000000] (03) 83c408     add esp,+08
24. [00000b65] [0010153e] [00000000] (03) 8945fc     mov [ebp-04],eax
25. [00000b68] [0010153e] [00000000] (03) 8b45fc     mov eax,[ebp-04]
26. [00000b6b] [0010153a] [00000000] (01) 50          push eax
27. [00000b6c] [00101536] [0000032b] (05) 682b030000 push 0000032b
28. [00000b71] [00101532] [00000b76] (05) e8e9f7ffff call 0000035f

```

Input_would_Halt = 0

```

29. [00000b76] [0010153e] [00000000] (03) 83c408     add esp,+08
30. [00000b79] [0010153e] [00000000] (02) 33c0       xor eax,eax
31. [00000b7b] [00101542] [00000000] (02) 8be5       mov esp,ebp
32. [00000b7d] [00101546] [00100000] (01) 5d          pop ebp
33. [00000b7e] [0010154a] [00000080] (01) c3         ret

```

Number_of_User_Instructions(33)
Number of Instructions Executed(26560)

Appendix

Key basis of the {Sipser, Kozen, Linz} halting problem undecidability proofs.

The following code implements the key basis of the {Sipser, Kozen, Linz} halting problem undecidability proofs:

```
void H_Hat(u32 P)
{
    u32 Input_would_Halt = Halts(P, P);
    if (Input_would_Halt)
        HERE: goto HERE;
}

int main()
{
    u32 Input_would_Halt = Halts((u32)H_Hat, (u32)H_Hat);
    Output("Input_would_Halt = ", Input_would_Halt);
}
```

In that the input program is defined to "do the opposite of whatever the halt decider decides". Here are quotes from {Sipser, Kozen, Linz} showing that key basis:

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)

Suppose (for a contradiction) that there existed a total machine K accepting the set HP...

- K halts and accepts if M halts on x, and
- K halts and rejects if M loops on x.

Consider a machine N ...

accepting if K rejects and going into a trivial loop if K accepts. **(Kozen:1997:233)**

...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, qy or qn...

Next, we modify H to produce a Turing machine H'... in situations where H reaches qy and halts, the modified machine H' will enter an infinite loop. **(Linz:1990:318-319)**

Linz, Peter 1990. An Introduction to Formal Languages and Automata. Lexington/Toronto: D. C. Heath and Company. (315-320)

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

Kozen, Dexter 1997. Automata and Computability. New York: Springer-Verlag. (231-234).

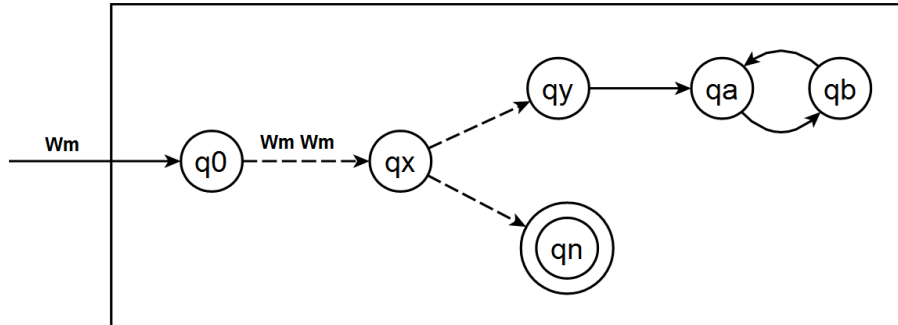


Figure 12.3 Turing Machine \hat{H}

$$\hat{H}.q_0 w_M \vdash^* \hat{H}.q_x w_M w_M \vdash^* \hat{H}.q_y \infty$$

$$\hat{H}.q_0 w_M \vdash^* \hat{H}.q_x w_M w_M \vdash^* \hat{H}.q_n$$

The above is adapted from (Linz:1990:319).

It shows that Turing machine \hat{H} copies its input at (q_0) and begins executing an embedded copy of the original halt decider with this input at (q_x).

The (q_y) state indicates that the halt decider decides that its input would halt. The (q_n) state indicates the input would not halt. The appended (q_a) and (q_b) states cause \hat{H} to infinitely loop if the halt decider decides that its input would halt.

It can be understood from the above specification that when the embedded halt decider @ $\hat{H}.q_x$ bases its halting decision on simulating its input, and it has (\hat{H} , \hat{H}) as its input that:

- $\hat{H}.q_0$ would copy its input and then $\hat{H}.q_x$ would simulate its input with this copy then
- $\hat{H}.q_0$ would copy its input and then $\hat{H}.q_x$ would simulate its input with this copy then
- $\hat{H}.q_0$ would copy its input and then $\hat{H}.q_x$ would simulate its input with this copy...

unless and until the halt decider @ $\hat{H}.q_x$ stops simulating its input.

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