Halting problem undecidability and infinitely nested simulation (V5)

This is an explanation of a key new insight into the halting problem provided in the language of software engineering. Technical computer science terms are explained using software engineering terms.

To fully understand this paper a software engineer must be an expert in: the C programming language, the x86 programming language, exactly how C translates into x86 **and the ability to recognize infinite recursion at the x86 assembly language level.** No knowledge of the halting problem is required.

The computer science term “halting” means that a Turing Machine terminated normally reaching its last instruction known as its “final state”. This is the same idea as when a function returns to its caller as opposed to and contrast with getting stuck in an infinite loop or infinite recursion.

> In computability theory, the halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever. Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.
>
> For any program H that might determine if programs halt, a "pathological" program P, called with some input, can pass its own source and its input to H and then specifically do the opposite of what H predicts P will do. No H can exist that handles this case. https://en.wikipedia.org/wiki/Halting_problem

Technically a halt decider is a program that computes the mapping from a pair of input finite strings to its own accept or reject state based on the actual behavior specified by these finite strings. In other words it determines whether or not its input would halt and returns 0 or 1 accordingly.

> Computable functions are the basic objects of study in computability theory. Computable functions are the formalized analogue of the intuitive notion of algorithms, in the sense that a function is computable if there exists an algorithm that can do the job of the function, i.e. **given an input of the function domain it can return the corresponding output**.
> https://en.wikipedia.org/wiki/Computable_function

The most definitive way to determine the actual behavior of the actual input is to simply simulate this input and watch its behavior. This is the ultimate measure of the actual behavior of the input. A simulating halt decider (SHD) simulates its input and determines the halt status of this input on the basis of the behavior of this correctly simulated of its input.

The x86utm operating system was created so that all of the details of the the halting problem counter-example could be examined at the much higher level of abstraction of the C/x86 computer languages. It is based on a very powerful x86 emulator.

The function named P was defined to do the opposite of whatever H reports that it will do. If H(P,P) reports that its input halts, P invokes an infinite loop. If H(P,P) reports that its input is non-halting, P immediately halts.

The technical computer science term "halt" means that a program will reach its last instruction technically called its final state. For P this would be its machine address `[0000136c].`

H simulates its input one x86 instruction at a time using an x86 emulator. As soon as H(P,P) detects the same infinitely repeating pattern (that we can all see), it aborts its simulation and rejects its input.

Anyone that is an expert in the C programming language, the x86 programming language, exactly how C translates into x86 and what an x86 processor emulator is can easily verify that the correctly simulated input to H(P,P) by H specifies a non-halting sequence of configurations.

Software engineering experts can reverse-engineer what the correct x86 emulation of the input to H(P,P) would be for one emulation and one nested emulation thus confirming that the provided execution trace is correct. They can do this entirely on the basis of the x86 source-code for P with no need to see the source-code or execution trace of H.

The function named H continues to simulate its input using an x86 emulator until this input either halts on its own or H detects that it would never halt. If its input halts H returns 1. If H detects that its input would never halt H returns 0.

```
#include <stdint.h>
#define u32 uint32_t

void P(u32 x)
{
  if (H(x, x))
    HERE: goto HERE;
  return;
}

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

_P()
[00001352](01)  55              push ebp
[00001353](02)  8bec            mov ebp,esp
[00001355](03)  8b4508          mov eax,[ebp+08]
[00001358](01)  50              push eax      // push P
[00001359](03)  8b4d08          mov ecx,[ebp+08]
[0000135c](01)  51              push ecx      // push P
[0000135d](05)  e840feffff      call 000011a2 // call H
[00001362](03)  83c408          add esp,+08
[00001365](02)  85c0            test eax,eax
[00001367](02)  7402            jz 0000136b
[00001369](02)  ebfe            jmp 00001369
[0000136b](01)  5d              pop ebp
[0000136c](01)  c3              ret
Size in bytes:(0027) [0000136c]
```

```
_main()
[00001372](01)  55              push ebp
[00001373](02)  8bec            mov ebp,esp
[00001375](05)  6852130000      push 00001352 // push P
[0000137a](05)  6852130000      push 00001352 // push P
[0000137f](05)  e81efeffff      call 000011a2 // call H
[00001384](03)  83c408          add esp,+08
[00001387](01)  50              push eax
[00001388](05)  6823040000      push 00000423 // "Input_Halts = "
[0000138d](05)  e8e0f0ffff      call 00000472 // call Output
[00001392](03)  83c408          add esp,+08
[00001395](02)  33c0            xor eax,eax
[00001397](01)  5d              pop ebp
[00001398](01)  c3              ret
Size in bytes:(0039) [00001398]

    machine   stack     stack     machine    assembly
    address   address   data      code       language
    ========  ========  ========  =========  =============
...[00001372][0010229e][00000000] 55         push ebp
...[00001373][0010229e][00000000] 8bec       mov ebp,esp
...[00001375][0010229a][00001352] 6852130000 push 00001352 // push P
...[0000137a][00102296][00001352] 6852130000 push 00001352 // push P
...[0000137f][00102292][00001384] e81efeffff call 000011a2 // call H

Begin Local Halt Decider Simulation   Execution Trace Stored at:212352
...[00001352][0021233e][00212342] 55         push ebp      // enter P
...[00001353][0021233e][00212342] 8bec       mov ebp,esp
...[00001355][0021233e][00212342] 8b4508     mov eax,[ebp+08]
...[00001358][0021233a][00001352] 50         push eax      // push P
...[00001359][0021233a][00001352] 8b4d08     mov ecx,[ebp+08]
...[0000135c][00212336][00001352] 51         push ecx      // push P
...[0000135d][00212332][00001362] e840feffff call 000011a2 // call H
...[00001352][0025cd66][0025cd6a] 55         push ebp      // enter P
...[00001353][0025cd66][0025cd6a] 8bec       mov ebp,esp
...[00001355][0025cd66][0025cd6a] 8b4508     mov eax,[ebp+08]
...[00001358][0025cd62][00001352] 50         push eax      // push P
...[00001359][0025cd62][00001352] 8b4d08     mov ecx,[ebp+08]
...[0000135c][0025cd5e][00001352] 51         push ecx      // push P
...[0000135d][0025cd5a][00001362] e840feffff call 000011a2 // call H
Local Halt Decider: Infinite Recursion Detected Simulation Stopped
```

H sees that P is calling the same function from the same machine address with identical parameters, twice in sequence. This is the infinite recursion (infinitely nested simulation) non-halting behavior pattern.

```
...[00001384][0010229e][00000000] 83c408     add esp,+08
...[00001387][0010229a][00000000] 50         push eax
...[00001388][00102296][00000423] 6823040000 push 00000423 // "Input_Halts = "
---[0000138d][00102296][00000423] e8e0f0ffff call 00000472 // call Output
Input_Halts = 0
...[00001392][0010229e][00000000] 83c408     add esp,+08
...[00001395][0010229e][00000000] 33c0       xor eax,eax
...[00001397][001022a2][00100000] 5d         pop ebp
...[00001398][001022a6][00000004] c3         ret
Number_of_User_Instructions(1)
Number of Instructions Executed(15892) = 237 pages
```

The correct simulation of the input to H(P,P) and the direct execution of P(P) are not computationally equivalent thus need not have the same halting behavior.

The following is the same idea a shown above this time it is applied to the Peter Linz Halting Problem proof. It can only be undertood within the context of this proof.

A simulating halt decider (SHD) computes the mapping from its inputs to its own final states on the basis of the behavior of its correctly simulated input.

All of the conventional halting problem counter-example inputs are simply rejected by a simulating halt decider as non-halting because they fail to meet the Linz definition of halting:

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

USENET comp.theory: On 4/11/2022 3:19 PM, Malcolm McLean wrote:
> PO's idea is to have a simulator with an infinite cycle detector.
> You would achieve this by modifying a UTM, so describing it as
> a "modified UTM", or "acts like a UTM until it detects an infinite
> cycle", is reasonable. And such a machine is a fairly powerful
> halt decider. Even if the infinite cycle detector isn't very
> sophisticated, it will still catch a large subset of non-halting
> machines.

The following simplifies the syntax for the definition of the Linz Turing machine $\hat{H}$.
There is no need for the infinite loop after H.qy because it is never reached. The halting criteria has been adapted so that it applies to a simulating halt decider (SHD).

$\hat{H}.q_0\ \langle\hat{H}\rangle \vdash^* H\ \langle\hat{H}\rangle\ \langle\hat{H}\rangle \vdash^* H.qy$
If the correctly simulated input $\langle\hat{H}\rangle\ \langle\hat{H}\rangle$ to H would reach its own final state of $\langle\hat{H}.qy\rangle$ or $\langle\hat{H}.qn\rangle$.

$\hat{H}.q_0\ \langle\hat{H}\rangle \vdash^* H\ \langle\hat{H}\rangle\ \langle\hat{H}\rangle \vdash^* H.qn$
If the correctly simulated input $\langle\hat{H}\rangle\ \langle\hat{H}\rangle$ to H would never reach its own final state of $\langle\hat{H}.qy\rangle$ or $\langle\hat{H}.qn\rangle$.

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

Then these steps would keep repeating: (unless their simulation is aborted)
$\hat{H}_0$ copies its input $\langle\hat{H}_1\rangle$ to $\langle\hat{H}_2\rangle$ then H_0 simulates $\langle\hat{H}_1\rangle\ \langle\hat{H}_2\rangle$
$\hat{H}_1$ copies its input $\langle\hat{H}_2\rangle$ to $\langle\hat{H}_3\rangle$ then H_1 simulates $\langle\hat{H}_2\rangle\ \langle\hat{H}_3\rangle$
$\hat{H}_2$ copies its input $\langle\hat{H}_3\rangle$ to $\langle\hat{H}_4\rangle$ then H_2 simulates $\langle\hat{H}_3\rangle\ \langle\hat{H}_4\rangle$...

Since we can see that the simulated input: $\langle\hat{H}_0\rangle$ to H would never reach its own final state of $\langle\hat{H}_0.qy\rangle$ or $\langle\hat{H}_0.qn\rangle$ we know that it is non-halting.

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

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

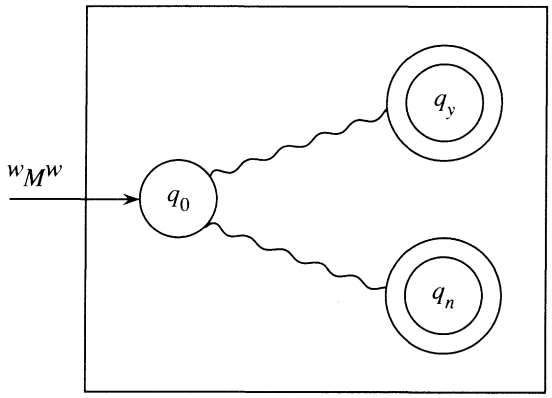
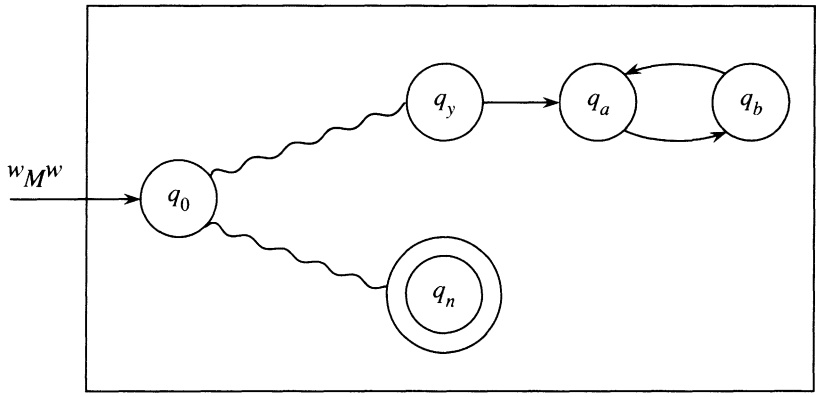


Figure 12.2



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. ■