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 what an x86 process emulator is. 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.

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.

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.

Without seeing any of the details of the 264 pages of the execution trace of H we can verify on the basis of the x86 source code for P and the execution trace that H derives that H does correctly simulate its input until it detects the same infinitely repeating pattern that we all can see.

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 [000009f0].

If the actual behavior of the correctly simulated input to H(P,P) specifies a non-halting sequence of instructions then this actual behavior supersedes and overrules anything and everything else that disagrees.

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()
[000009d6](01) 55         push ebp
[000009d7](02) 8bec       mov ebp,esp
[000009d9](03) 8b4508     mov eax,[ebp+08]
[000009dc](01) 50         push eax         // push P
[000009dd](03) 8b4d08     mov ecx,[ebp+08]
[000009e0](01) 51         push ecx         // push P
[000009e1](05) e840feffff call 00000826    // call H
[000009e6](03) 83c408     add esp,+08
[000009e9](02) 85c0       test eax,eax
[000009eb](02) 7402       jz 000009ef
[000009ed](02) ebfe       jmp 000009ed
[000009ef](01) 5d         pop ebp
[000009f0](01) c3         ret              // Final state
Size in bytes:(0027) [000009f0]
```

The simulated input to H(P,P) cannot possibly reach its own final state of [000009f0] it keeps repeating [000009d6] to [000009e1] until aborted.

```
Begin Local Halt Decider Simulation
...[000009d6][00211368][0021136c] 55         push ebp         // enter P
...[000009d7][00211368][0021136c] 8bec       mov ebp,esp
...[000009d9][00211368][0021136c] 8b4508     mov eax,[ebp+08]
...[000009dc][00211364][000009d6] 50         push eax         // Push P
...[000009dd][00211364][000009d6] 8b4d08     mov ecx,[ebp+08]
...[000009e0][00211360][000009d6] 51         push ecx         // Push P
...[000009e1][0021135c][000009e6] e840feffff call 00000826    // Call H
...[000009d6][0025bd90][0025bd94] 55         push ebp         // enter P
...[000009d7][0025bd90][0025bd94] 8bec       mov ebp,esp
...[000009d9][0025bd90][0025bd94] 8b4508     mov eax,[ebp+08]
...[000009dc][0025bd8c][000009d6] 50         push eax         // Push P
...[000009dd][0025bd8c][000009d6] 8b4d08     mov ecx,[ebp+08]
...[000009e0][0025bd88][000009d6] 51         push ecx         // Push P
...[000009e1][0025bd84][000009e6] e840feffff call 00000826    // Call H
Local Halt Decider: Infinite Recursion Detected Simulation Stopped
```

Because the correctly simulated input to H(P,P) cannot possibly reach its own final state at [000009f0] it is necessarily correct for H to reject this input as non-halting.

The correctly simulated 27 bytes of machine code at [000009d6] would never reach its own final state at [000009f0] when correctly simulated by the simulating halt decider (SHD) at machine address [00000826].

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 Ĥ.
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).

Ĥ.q0 ⟨Ĥ⟩ ⊢* H ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* H.qy
If the correctly simulated input ⟨Ĥ⟩ ⟨Ĥ⟩ to H would reach its own final state of ⟨Ĥ.qy⟩ or ⟨Ĥ.qn⟩.

Ĥ.q0 ⟨Ĥ⟩ ⊢* H ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* H.qn
If the correctly simulated input ⟨Ĥ⟩ ⟨Ĥ⟩ to H would never reach its own final state of ⟨Ĥ.qy⟩ or ⟨Ĥ.qn⟩.

When Ĥ is applied to ⟨Ĥ⟩
Ĥ copies its input ⟨Ĥ0⟩ to ⟨Ĥ1⟩ then H simulates ⟨Ĥ0⟩ ⟨Ĥ1⟩

Then these steps would keep repeating: (unless their simulation is aborted)
Ĥ0 copies its input ⟨Ĥ1⟩ to ⟨Ĥ2⟩ then H0 simulates ⟨Ĥ1⟩ ⟨Ĥ2⟩
Ĥ1 copies its input ⟨Ĥ2⟩ to ⟨Ĥ3⟩ then H1 simulates ⟨Ĥ2⟩ ⟨Ĥ3⟩
Ĥ2 copies its input ⟨Ĥ3⟩ to ⟨Ĥ4⟩ then H2 simulates ⟨Ĥ3⟩ ⟨Ĥ4⟩...

Since we can see that the simulated input: ⟨Ĥ0⟩ to H would never reach its own final state of ⟨Ĥ0.qy⟩ or ⟨Ĥ0.qn⟩ 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)