Termination Analyzer H is Not Fooled by Pathological Input P

The notion of a simulating termination analyzer is examined at the concrete level of pairs of C functions. This is similar to AProVE: Non-Termination Witnesses for C Programs. The termination status decision is made on the basis of the dynamic behavior of the input. This paper explores what happens when a simulating termination analyzer is applied to an input that calls itself.

In computer science, termination analysis is program analysis which attempts to determine whether the evaluation of a given program halts for each input. This means to determine whether the input program computes a total function.
https://en.wikipedia.org/wiki/Termination_analysis

The halting problem proof is understood to be the logical impossibility of specifying a halt decider H that correctly reports the halt status of input P that is defined to do the opposite of whatever value that H reports. **Of course this is impossible.**

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

To understand this analysis requires a sufficient knowledge of the C programming language and what an x86 emulator does. It is also very helpful to have some basic understanding of the x86 programming language.

CODE SAMPLE 1

```
typedef void (*ptr)();
int H0(ptr P);

void Infinite_Loop()
{
  HERE: goto HERE;
}

void Infinite_Recursion()
{
  Infinite_Recursion();
}

void DDD()
{
  H0(DDD);
}

int main()
{
  H0(Infinite_Loop);
  H0(Infinite_Recursion);
  H0(DDD);
}
```

Analysis of CODE SAMPLE 1

Every C programmer that knows what an x86 emulator is knows that when **H0** emulates the machine language of **Infinite_Loop, Infinite_Recursion, and DDD** that it must abort these emulations so that itself can terminate normally.

When this is construed as non-halting criteria then simulating termination analyzer **H0** is correct to reject these inputs as non-halting by returning 0 to its caller.

Simulating termination analyzers must report on the behavior that their finite string input specifies thus **H0** must report that **DDD** correctly emulated by **H0** remains stuck in recursive simulation.

When we stipulate that the only measure of a correct emulation is the semantics of the x86 programming language then we see that when DDD is correctly emulated by H0 that its call to H0(DDD) cannot possibly return.

```
_DDD()
[00002172] 55              push ebp      ; housekeeping
[00002173] 8bec            mov ebp,esp   ; housekeeping
[00002175] 6872210000      push 00002172 ; push DDD
[0000217a] e853f4ffff      call 000015d2 ; call H0(DDD)
[0000217f] 83c404          add esp,+04
[00002182] 5d              pop ebp
[00002183] c3              ret
Size in bytes:(0018) [00002183]
```

When we define H1 as identical to H0 except that DDD does not call H1 then we see that when DDD is correctly emulated by H1 that its call to H0(DDD) does return. This is the same behavior as the directly executed DDD().

A partial halt decider is a computable function that computes the mapping from its finite string input to a Boolean value corresponding to the behavior that this finite string specifies. It does this for a limited set of inputs.

The following algorithm is used by the simulating termination analyzers:

<MIT Professor Sipser agreed to ONLY these verbatim words 10/13/2022>
If simulating halt decider H correctly simulates its input D
until H correctly determines that its simulated D would never
stop running unless aborted then

H can abort its simulation of D and correctly report that D
specifies a non-halting sequence of configurations.
</MIT Professor Sipser agreed to ONLY these verbatim words 10/13/2022>

The next example (uses the above algorithm) yet is not a termination analyzer because it only references a single program / input pair.

Unless every single detail is made 100% explicit false assumptions always slip through the cracks. This is why H(P,P) must be fully understood at the C level before its isomorphism is examined at the Turing Machine level.

H(P,P) has the classic halting problem proof relationship to its input. **H(P,P)** has the same behavior as the above **DDD** correctly simulated by **H0**. This prevents **P** correctly simulated by **H** from reaching past its own first line. This makes the classic halting problem question moot:

What Boolean value can H correctly return when input P is defined to do the opposite of every value that H returns? P correctly emulated by H cannot possibly reach this paradoxical point at its own second line.

```c
typedef int (*ptr2)();
int H(ptr2 P, ptr2 I);

int P(ptr2 x)
{
  int Halt_Status = H(x, x);
  if (Halt_Status)
    HERE: goto HERE;
  return Halt_Status;
}

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

When we understand that
(a) Decider **H** must report on the behavior that its input actually specifies.

(b) The measure of this behavior is **P** correctly simulated by **H** including its recursive call to **H(P,P)**.

Then we can see that **P** correctly simulated **H** cannot possibly reach past its own first line.

```
_P()
[000020e2] 55             push ebp           ; housekeeping
[000020e3] 8bec           mov ebp,esp        ; housekeeping
[000020e5] 51             push ecx           ; housekeeping
[000020e6] 8b4508         mov eax,[ebp+08]   ; parameter
[000020e9] 50             push eax           ; push parameter
[000020ea] 8b4d08         mov ecx,[ebp+08]   ; parameter
[000020ed] 51             push ecx           ; push parameter
[000020ee] e82ff3ffff     call 00001422      ; call H(P,P)
[000020f3] 83c408         add esp,+08
[000020f6] 8945fc         mov [ebp-04],eax
[000020f9] 837dfc00       cmp dword [ebp-04],+00
[000020fd] 7402           jz 00002101
[000020ff] ebfe           jmp 000020ff
[00002101] 8b45fc         mov eax,[ebp-04]
[00002104] 8be5           mov esp,ebp
[00002106] 5d             pop ebp
[00002107] c3             ret
Size in bytes:(0038) [00002107]
```

The same reasoning that applied to DDD correctly simulated by HH0 applies here. When we stipulate that the only measure of a correct emulation is the semantics of the x86 programming language then we see that when P is correctly emulated by H that its call to H(P,P) cannot possibly return.

As we can see from DDD correctly emulated by H0 the behavior of the input to H(P,P) is different than the behavior of the directly executed P(P). Computable functions including halt deciders are only accountable for the actual behavior of their actual inputs. No one is free to overrule the semantics of the x86 language.

Because the call from P correctly simulated by H to H(P,P) cannot possibly return this P cannot possibly reach past its own first line. This makes the paradoxical portion of P unreachable making it moot.

H uses the same non-halt status criteria that it uses to detect infinite recursion to detect and reject that P correctly simulated by H would halt. H returns 0 to it caller to indicate it rejected its input as non-halting.