Defining a Decidability Decider for the Halting Problem

When the Halting Problem is analyzed as a David Hilbert formalist mathematical proof from finite strings to final states then previously undiscovered semantic details emerge. Treating the task of a potential halt decider as deriving a formal mathematical proof from its inputs to its own final states requires the potential halt decider to trace the sequence of state transitions of the input TMD as syntactic logical consequence inference steps of this formal proof.

Halt Decider definition
A Halt Decider (HD) determines whether or not a finite string represents a correct Turing Machine Description that would halt on its input. It does this by performing a mathematical proof on a pair of finite strings as a David Hilbert formalism within the language of Turing Machine Descriptions.

Decidability of the Halting Problem as a David Hilbert formalism
∃a ∈ Turing_Machines_Descriptions
∀b ∈ Turing_Machines_Descriptions
∀c ∈ Finite_Strings
Halts(a, b, c) ∨ Loops(a, b, c)

The syntactic logical consequence of this proof transitions to one of two finite strings:
(1) "Y" input pair (b,c) would halt on its input.
(2) "N" input pair (b,c) would loop on its input.

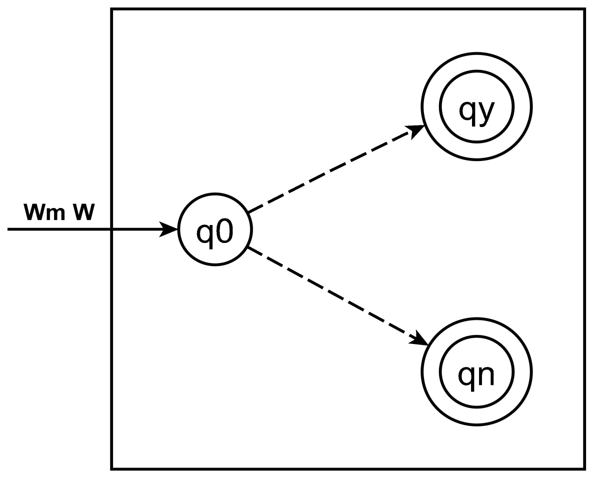
If the first element of the input finite string pair is a correct TMD then this proof must necessarily proceed from the specified TMD start state through all of the state transitions of this TMD to the halting or non halting behavior of this TMD.

The proof would essentially be a hypothetical execution trace** of the state transition sequences of the input TMD. It cannot be an actual execution trace or the halt decider could have non-halting behavior. **Like step-by-step mode in a debugger.

The following has been adapted from material from the this book:
An Introduction to Formal Languages and Automata by Peter Linz 1990 pages 318-320

We begin our analysis by constructing a hypothetical halt decider: H.

Figure 12.1 Turing Machine H



The dashed lines proceeding from state (q0) are represented in the text definition as the asterisk ⊢* wildcard character. These conventions are used to encode unspecified state transition sequences.

Definition of Turing Machine H (state transition sequence)
H.q0 Wm W ⊢* H.qy // Wm would halt on its input W
H.q0 Wm W ⊢* H.qn // Wm would loop on its input W

The diagram and the state transition sequence indicate that H begins at its own start state H.q0 and is applied to finite string pair (Wm, W).

Then it proceeds through an unspecified set of state transitions to one of its final states.

H.qy is the final state of H indicating that Wm would halt on input W.
H.qn is the final state of H indicating that Wm would loop on input W.

In an attempt to provide a valid counter-example proving by contradiction that a Halt Decider cannot possibly exist for every possible TM / Input pair we create Turing Machine Ĥ by making the following changes to H:

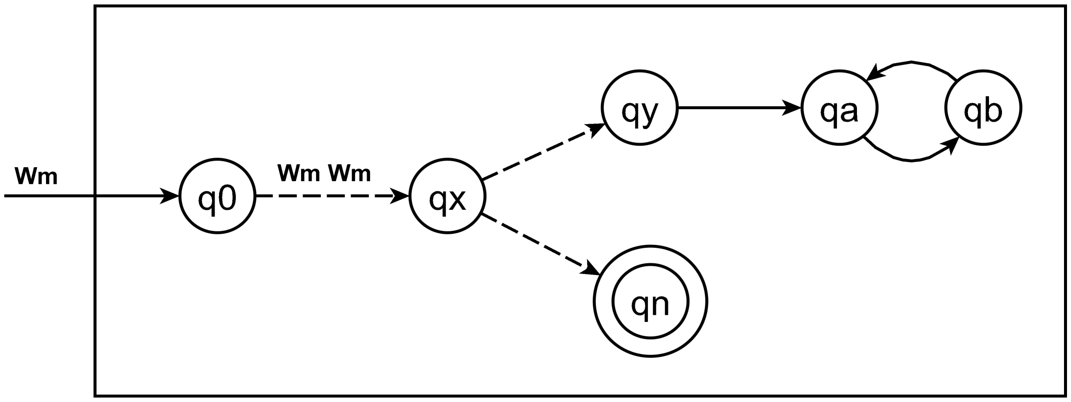
(1) Ĥ copies its input Wm a its Ĥ.q0 state and transitions to its Ĥ.qx state.

(2) Ĥ would begin to evaluate Wm Wm at its Ĥ.qx state in exactly the same way that H would begin to evaluate its Wm W input at its H.q0 state.

(3) States (qa) and (qb) are appended to existing final state ((qy)) such that any transition to state (qy) will cause Ĥ to loop.

Since Turing Machine Ĥ is created by adapting H, it would have exactly the same behavior at its Ĥ.qx state as H would have at its H.q0 state.

Figure 12.3 Turing Machine Ĥ



Definition of Turing Machine Ĥ (state transition sequence)
Ĥ.q0 Wm ⊢* Ĥ.qx Wm Wm ⊢* Ĥ.qy ∞
Ĥ.q0 Wm ⊢* Ĥ.qx Wm Wm ⊢* Ĥ.qn

If Turing Machine H is applied to Turing Machine descriptions [Ĥ] [Ĥ]
would H transition to H.y or H.n ?

H [Ĥ] [Ĥ2] // We append a “2” to the second Ĥ for clarity

To answer this question we perform an execution trace on H
(1) H begins at its start state H.q0.
(2) H begins to evaluate what [Ĥ] would do on its input [Ĥ2].

Because H is performing a mathematical proof on finite strings [Ĥ] [Ĥ2] the required syntactic logical inference chain is simply the state transitions that [Ĥ] would make on its input [Ĥ2]. In simplest terms H is performing a step-by-step debug trace of [Ĥ].

Step-by-step debug trace of what [Ĥ] would do on its input [Ĥ2]
(3) [Ĥ] would begin at its start state [Ĥ].q0
(4) [Ĥ] would make a copy of its input [Ĥ2], we will call this [Ĥ3].
(5) [Ĥ] would transition to its state [Ĥ].qx.
(6) [Ĥ] would begin to evaluate what [Ĥ2] would do on its input [Ĥ3].
Can you see the infinite recursion?

H [Ĥ] [Ĥ] specifies an infinitely recursive evaluation sequence. Every HP proof by contradiction depends this same infinite recursion. What no one ever noticed before is that the debug trace by the halt decider must abort its evaluation long before it ever reaches the appended infinite loop. Because the formal proof is aborted before ever reaching the final states of H, H is not a decider for H [Ĥ] [Ĥ].

Because of this every TMD in the infinitely recursive sequence is defined in terms of H each would reject the whole sequence as semantically incorrect before even beginning any halting evaluation.

Therefore H would correctly transition to its H.qn state rejecting the infinitely recursive evaluation sequence specified by [Ĥ] [Ĥ] as the semantic error of Pathological self-reference(Olcott 2004).

As further evidence that infinitely recursive evaluation sequence has been overlooked we only need to know that every TMD always requires its own TM/Input pair. For any finite sequence of input H could always decide halting.

If Ĥ did not copy its input and instead simply took two inputs then the Halting Decision would be easy. Ĥ would transition to its Ĥ.qy state because Ĥ2 would transition to its Ĥ2.qn state on null input. This would cause H to transition to its H.qn state deciding halting for [Ĥ] [Ĥ2].

Rice's theorem: all non-trivial, semantic properties of programs are undecidable.

Defining a Decidability Decider for the Halting Problem
∃a ∈ Turing_Machines_Descriptions
∀b ∈ Turing_Machines_Descriptions
∀c ∈ Finite_Strings
(Halts(a, b, c) ∨ Loops(a, b, c)) → Decidable(a,b,c)
(~Halts(a, b, c) ∧ ~Loops(a, b, c)) → Undecidable(a,b,c)

Putting this in the proper terminology of Rice's Theorem:

DD is a UTM that executes (a,b,c) segregating (a,b,c) into two sets:
(1) (a) Decides halting for (b,c)
(2) (a) Does not decide halting for (b,c)