Halting Problem Proof from Finite Strings to Final States

If there truly is a proof that shows that no universal halt decider exists on the basis that certain tuples: (H, Wm, W) are undecidable, then this very same proof (implemented as a Turing machine) could be used by H to reject some of its inputs. When-so-ever the hypothetical halt decider cannot derive a formal proof from its input strings and initial state to final states corresponding the mathematical logic functions of Halts(Wm, W) or Loops(Wm, W), halting undecidability has been decided.

Wikipedia: Formalism (philosophy of mathematics)
In foundations of mathematics, philosophy of mathematics, and philosophy of logic, formalism is a theory that holds that statements of mathematics and logic can be considered to be statements about the consequences of certain string manipulation rules.

If we form this proof as a Formalism (philosophy of mathematics) then such a proof could be directly executed by a Turing machine. When the finite strings of the proof specify Turing machine descriptions then their corresponding rules-of-inference (string manipulation rules) are provided by the formal language of Turing Machine descriptions.

A Hypothetical Halting Decider H could be mathematically specified as:
(1) A David Hilbert formalist proof (defined above).
(2) Where the inference steps are construed as an execution trace in language of
Turing machine descriptions.
(3) From the initial state of a Turing machine description H.q0 and
(4) Its finite string input pair: (Wm, W) to final states corresponding to
(5) The mathematical logic predicates: Halts(Wm, W) and Loops(Wm, W).

[I had to simplify this for less sophisticated reviewers]
Where Loops(Wm, W) means that Wm would get stuck in an infinite loop on input W.

Basis of the mathematical formalist version of the Halting Problem proof
∃ H ∈ Turing_Machine_Descriptions
∀ Wm ∈ Turing_Machine_Descriptions
∀ W ∈ Finite_Strings
{H.q0, Wm, W} ⊢ Halts(Wm, W) // formalist proof from premises to logic predicate
{H.q0, Wm, W} ⊢ Loops(Wm, W) // formalist proof from premises to logic predicate

The formal proof involves tracing the sequence of state transitions of the input TMD as syntactic logical consequence inference steps in the formal language of Turing Machine Descriptions.

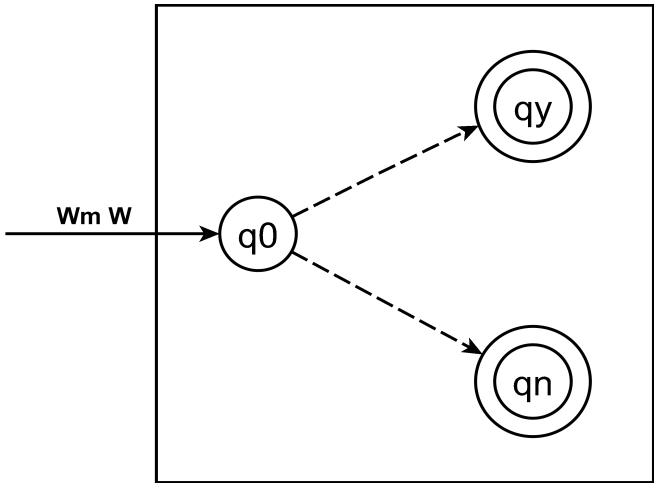
The proof would essentially be a hypothetical execution trace (like step-by-step mode in a debugger) of the state transition sequences of the input TMD. It cannot be an actual execution trace or the halt decider could have non-halting behavior.

When-so-ever Turing machine H decides that no finite sequence of state transitions would correspond to the mathematical logic predicate Halts(Wm, W) or Loops(Wm, W) it transitions to H.Undecidable.

Although Rice's Theorem (1951) “proves” that no Turing machine can divide an infinite set of finite strings into halting decidable and halting undecidable I show how this limitation might be circumvented.

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.

Turing Machine H (state transition sequence)
H.q0 Wm W ⊢* H.qy // Wm is a TMD that would halt on its input W
H.q0 Wm W ⊢* H.qn // else

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 is a TMD would halt on input W.
H.qn is the final state of H indicating that Wm is a TMD would halt 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:

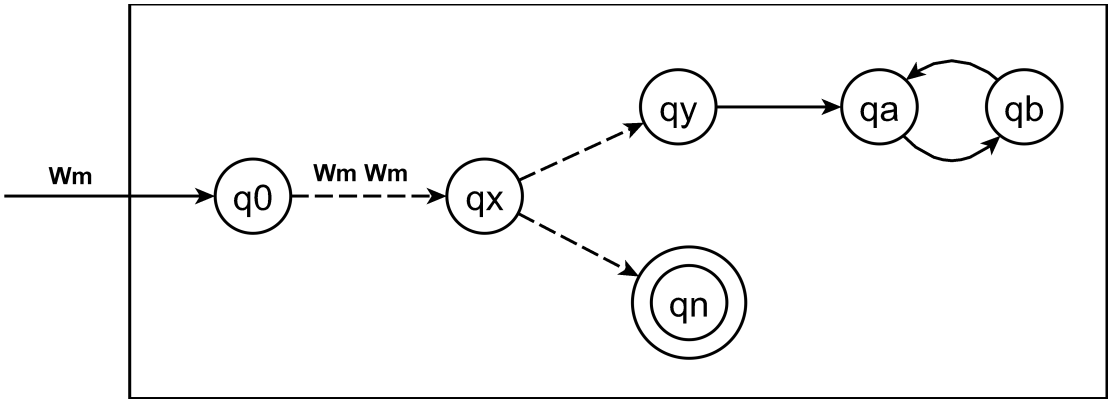
(1) Ĥ copies its input Wm a its Ĥ.q0 state and transitions to its Ĥ.qx state.

(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 ((halt)) such that any transition to state (halt) 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



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

Categorically exhaustive reasoning reverse engineers how H(Wm, W) could be analyzed more deeply

(1) Determining if (Wm, W) halts requires the category:
(2) Determining what (Wm, W) does which requires the category:
(3) Determining what (Wm, W) does in the same order that (Wm, W) does it which requires the category:
(4) Some level of execution trace of (Wm, W) which includes (but does not require) the category:
(5) Step-by-step debug trace of (Wm, W) which is selected as the simplest alternative.

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 [H] would do on its input [Ĥ][Ĥ2]
(01) H begins at its start state H.q0.
(02) H begins to evaluate what [Ĥ] would do on its input [Ĥ2].

Step-by-step debug trace of what [Ĥ] would do on its input [Ĥ2]
(03) [Ĥ] would begin at its start state [Ĥ].q0
(04) [Ĥ] would make a copy of its input [Ĥ2], we will call this [Ĥ3].
(05) [Ĥ] would transition to its state [Ĥ].qx.
(06) [Ĥ] would begin to evaluate what [Ĥ2] would do on its input [Ĥ3].

Step-by-step debug trace of what [Ĥ2] would do on its input [Ĥ3]
(07) [Ĥ2] would begin at its start state [Ĥ2].q0
(08) [Ĥ2] would make a copy of its input [Ĥ3], we will call this [Ĥ4].
(09) [Ĥ2] would transition to its state [Ĥ2].qx.
(10) [Ĥ2] would begin to evaluate what [Ĥ3] would do on its input [Ĥ4].
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 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, and transition to H.qn.

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].

If we assume that a halting decidability decider could use something like the Prolog predicate unify_with_occurs_check/2 to detect and report the infinite recursive evaluation sequence of H(Ĥ, Ĥ2), then an actual halt decider might be defined as follows:

What happens if H decides infinite recursion at the recursion depth of three?
Ĥ2 aborts Ĥ3 and transitions to Ĥ2.qn.
which causes Ĥ1 to transition to Ĥ1.qy and loop.
which causes Ĥ to transition to Ĥ.qn.
which causes H to transition to H.qy. // thus H has decided halting for (Ĥ, Ĥ2)

What happens if H decides infinite recursion at the recursion depth of two?
Ĥ1 aborts Ĥ2 and transitions to Ĥ1.qn.
which causes Ĥ to transition to Ĥ.qy and loop.
which causes H to transition to H.qn. // thus H has decided halting for (Ĥ, Ĥ2)

What happens if H decides infinite recursion at the recursion depth of one?
Ĥ aborts Ĥ2 and transitions to Ĥ.qn.
which causes H to transition to H.qy. // thus H has decided halting for (Ĥ, Ĥ2)

Turing Machine Description Language
(Generalized from: TM, The Turing Machine Interpreter by David S. Woodruff)
http://www.lns.mit.edu/~dsw/turing/turing.html

The following is space delimited 'quintuples', each one of which is a five-symbol Turing Machine instruction. It is written in AWK regular expression syntax:

1) Initial State
2) Current Tape Symbol
3) Next State
4) Write Tape Symbol
5) Move Tape Head Left or Right

```
--1--       --2--      --3--       --4--      --5--
[0-9]+ [\x20-\x7e] [0-9]+ [\x20-\x7e] [LR]
```

If we make a fixed number of (hexadecimal) states then The UTM could simply hold a sorted list of std::strings. A sorted list would allow state transitions to occur by binary search. This would allow a very simple DFA based syntax checker.

The tape could be constructed using a std::vector. Binary Zero end markers would be in the tape alphabet and not the input alphabet. These trigger memory reallocation to enlarge the tape.

Clocksin and Mellish 2003, Programming in Prolog Using the ISO Standard Fifth Edition Chapter 10 The Relation of Prolog to Logic, page 254.

> Finally, a note about how Prolog matching sometimes differs from the unification used in Resolution. Most Prolog systems will allow you to satisfy goals like:
>
> **equal(X, X).**
>
> **?- equal(foo(Y), Y).**
>
> that is, they will allow you to match a term against an uninstantiated subterm of itself. In this example, foo(Y) is matched against Y, which appears within it. As a result, Y will stand for foo(Y), which is foo(foo(Y)) (because of what Y stands for), which is foo(foo(foo(Y))), and so on. So Y ends up standing for some kind of infinite structure. Note that, whereas they may allow you to construct something like this, most Prolog systems will not be able to write it out at the end. According to the formal definition of Unification, this kind of “infinite term” should never come to exist. Thus Prolog systems that allow a term to match an uninstantiated subterm of itself do not act correctly as Resolution theorem provers. In order to make them do so, we would have to add a check that a variable cannot be instantiated to something containing itself. Such a check, an occurs check, would be straightforward to implement, but would slow down the execution of Prolog programs considerably. Since it would only affect very few programs, most implementors have simply left it out [1].
>
> [1] The Prolog standard states that the result is undefined if a Prolog system attempts to match a term against an uninstantiated subterm of itself, which means that programs which cause this to happen will not be portable. A portable program should ensure that wherever an occurs check might be applicable the built-in predicate unify_with_occurs_check/2 is used explicitly instead of the normal unification operation of the Prolog implementation. As its name suggests, this predicate acts like =/2 except that it fails if an occurs check detects an illegal attempt to instantiate a variable.