

SE 3310b

Theoretical Foundations of Software Engineering

Universal Turing Machines, the Halting Problem and the Existence of Undecidable Languages

Aleksander Essex



Implications of the Church-Turing Thesis

Recall the **Church-Turing thesis** essentially states that if a problem P can be solved using an algorithm A it can be solved by some Turing machine M :

Algorithm A solves $P \rightarrow$ Turing Machine M solves problem P

Implications of the Church-Turing Thesis

Consider a computational system \mathcal{S} . If I could somehow prove it can simulate any Turing machine, i.e., if I could somehow prove:

Turing Machine $M \rightarrow$ Some instance of system \mathcal{S} ,

then by the Church-Turing thesis I can prove:

Algorithm $A \rightarrow$ Some instance of system \mathcal{S} .

The implication is that the computational system \mathcal{S} can be used to solve any problem that is algorithmically solvable.

Turing Completeness

Definition 1 (Turing Completeness).

A computational system S is called *Turing complete* if it can be used to simulate any Turing machine M

That is, S is Turing complete if it can be used to *simulate* the functionality of any Turing machine M . This notion of simulation will become important when we talk about Universal Turing machines. But for now, its also useful to talk about real-world programming languages

Turing Complete Programming Languages

Turing completeness is useful in the context of real-world computational systems (e.g., programming languages). For example, if you designed a new programming language and could prove it was Turing complete, you will have proved (via the C-T thesis) that your programming language can compute anything that any other Turing complete language can.

What do you need for a language to be Turing complete? Actually not that much. For an imperative language (e.g., C) you need:

1. The ability to read/write from memory
2. If statements
3. Goto statements

Example: Turing Completeness of C

Consider the following description of a Turing machine (a 2-state **Busy Beaver**):

Current	Read	Write	Move	Next
A	0	1	R	B
A	1	1	L	B
B	0	1	L	A
B	1	1	N	H

We can encode each state transition directly into C (see right):

```
// 2-State Busy Beaver Turing machine in C
#include <stdio.h>
int main(){

    int tape[10] = {0}; //init tape to all 0s
    int *head = &tape[5]; //init head in middle

stateA:
    if (!*head){ //if read 0
        *head = 1; //write 1
        head++; //move head right
        goto stateB;
    }
    else{ //if read 1
        *head = 1; //write 1
        head--; //move head left
        goto stateB;
    }

stateB:
    if (!*head){ //if read 0
        *head = 1; //write 1
        head--; //move head right
        goto stateA;
    }
    else{ //if read 1
        *head = 1; //write 1
        goto halt;
    }

halt:
    return 1;
};
```

Universal Computation

A computational system that is Turing complete is capable of *universal computation*, and is (in most cases) what we mean today by the term "computer."

This is an important distinction because there are computational systems (e.g., a clock, a calculator, etc) that are capable of some form of computation, but are not Turing complete. Something to consider next time you see a commercial where they're claiming your toothbrush as an on-board computer.

Consider the Dilemma

For each of the following statements mark True if the statement is true, and mark false if the statement is false:

1. The sky is blue: T / F
2. $1+1=3$: T / F
3. You'll mark this question False: T / F

If you mark False then the statement was True, which means you should mark True. But if you mark True, then the statement was False, which means you should mark False. But if you mark false...

Universal TMs and Simulation

A Universal Turing Machine (UTM) is a TM that can simulate the behavior of any Turing machine on any input.

Let U be a TM and let $s = \langle M, w \rangle$ be an input string where M is the encoding of a TM and w is the string input to that TM.

Some Aspects of U

- ▶ U is a computer, M is a program, w is the input argument.
- ▶ The description of U is independent of M , and descriptions of U in as few as $|Q| = 2$ states and $|\Gamma| = 6$ tape symbols are known.
- ▶ The precise details of U aren't considered here. All U needs to keep track of is what state M is currently, and what M 's head is currently pointing at. Then at each step it can scan through the description of M to decide what to do next.

What U Does

U works as follows: Given input $\langle M, w \rangle$ where M is a TM and w is a string:

- ▶ Simulate M on input w
- ▶ If M accepts, halt and output accept.
- ▶ If M rejects, halt and output reject.

Notice that:

- ▶ If M accepts w , then U accepts $\langle M, w \rangle$
- ▶ If M rejects w , then U rejects $\langle M, w \rangle$
- ▶ If M loops on w , then U loops on $\langle M, w \rangle$.

The Language of U

U is a Turing machine, and every Turing machine has a set of strings that it accepts. In other words, recognizes some language. So what language does U recognize? It recognizes $\langle M, w \rangle$, the set of *accepting* Turing machine/string combinations:

$$A_{TM} = \{ \langle M, w \rangle : \text{TMM } M \text{ accepts string } w \}$$

A_{TM} is known as the *acceptance problem*. Clearly, U recognizes A_{TM} . Notice, however, U does not decide A_{TM} : if M loops on w , then U loops on $\langle M, w \rangle$.

A_{TM} is Undecidable

Well maybe there is some TM H that could check M and somehow *decide* if it would go into a loop on input w , and then could reject.

Theorem: A_{TM} is undecidable.

Proof: to follow.

A Decider for A_{TM}

Let's begin by assuming the existence of a program H that decides A_{TM} .

H works as follows:

- ▶ Given input $\langle M, w \rangle$ where M is a TM and w is a string:
- ▶ If M accepts w , halt and output accept.
- ▶ Otherwise halt and output reject.

Notice we define H as a decider. It always answers “Yes” or “No” to the questions of whether machine M accepts input w .

Define a TM Encoding

Define some Turing machine encoding scheme. All TMs now correspond to some string, and we can sort them lexicographically (e.g., alphabetically) to produce some kind of canonical ordering of TMs. Now we can talk about the 1st TM, the 10th TM, etc.

A Table of all TMs and Strings.

Define the following table: let row i correspond to the i -th Turing machine (sorted lexicographically under the encoding scheme). Let column j correspond to string j .

Run H on the Table

Run H on each $\langle M, w \rangle$ and fill in result in appropriate cell: If TM_i accepts string j , fill in (i, j) with accept. Otherwise fill in reject. We can now define two sets: those TMs that will accept their own encoding, and those that don't. Let's look at this latter category.

Running TM's on Their Own Description

Consider the language of all TMs that do not accept their own encoding as input, i.e., let:

$$L = \{ \langle i, i \rangle : TM_i \text{ does not accept input } i \}$$

This corresponds to all cells on the diagonal that are reject. Is there a Turing machine that can decide this language? Suppose the n -th Turing machine, i.e., TM_n can decide language L .

What is TM_n accept?

1. It accepts TMs on their own description, and
2. It does the *opposite* of what H for the same input.

Running TM's on Their Own Description

Thus TM_n can be described as follows:

Given input i where TM_i is the i -th Turing machine:

1. Run H on input $\langle i, i \rangle$
2. If H outputs accept, halt and output reject. If H outputs reject, halt and output accept.

So TM_n does two things: it runs on its own input, and it outputs the opposite of what H outputs.

TM_n on its Own Description

Consider now what happens when TM_n is run on its own description n ?

Inputting $\langle n, n \rangle$ in to our description of TM_n we have:

1. Run H on input $\langle n, n \rangle$
2. If H outputs accept, halt and output reject.
3. If H outputs reject, halt and output reject.

TM_n Cannot Exist

Let's consider both cases of TM_n on input $\langle n, n \rangle$:

- ▶ Case 1: TM_n outputs accept. That means H output reject, meaning TM_n did not accept itself input. But this is the case where TM_n accepted itself as input. A contradiction!
- ▶ Case 2: TM_n outputs reject. That means H output accept, meaning TM_n accepted itself as input. But this is the case where TM_n did not accept itself as input. Another contradiction!

Conclusion: TM_n cannot exist.

A_{TM} is Undecidable

Since TM_n cannot exist, but it merely does the opposite of what H does, then by extension H cannot exist. And since H decides A_{TM} , therefore A_{TM} is undecidable. \square

The Halting Problem

Now we're ready to consider the famous *Halting problem*:

$$HALT = \{ \langle M, w \rangle : \text{TMM } M \text{ halts on string } w \}$$

The language *HALT* consists of all Turing machines that halt i.e., either accept or reject a given input. Can you decide this language? Can you always tell the difference between a TM stuck in a loop, and one that's just taking a really long time?

The Halting Problem

It's easy to see why being able to decide this language would be useful: if you could decide ahead of time if a program was going to run forever, you wouldn't have to wait around forever to find out!

Theorem 2.

HALT is undecidable.

Proof: Toward obtaining a contradiction, let's begin by assuming *HALT* is decidable.

The Halting Problem

Let S be a Turing machine that decides $HALT$. We now show that we can use S to build a TM T that will help us decide A_{TM} . T works as follows:

Given input $\langle M, w \rangle$ where M is a TM and w is a string:

- ▶ Run S on $\langle M, w \rangle$
 - ▶ If S rejects (i.e., M does not halt on w), output *reject*
 - ▶ Else simulate M on input w (since we know from S) that it will halt eventually.
 - ▶ If M accepts, halt and output *accept*
 - ▶ If M rejects, halt and output *reject*

The Halting Problem

Clearly T decides A_{TM} : If M accepts, T accepts. If M rejects, T rejects. If M gets stuck in a loop (as decided by S), then T rejects.

Problem: If $HALT$ is decidable then clearly A_{TM} is decidable. But we already proved A_{TM} was undecidable. This contradicts our initial assumption that $HALT$ is decidable.

Therefore $HALT$ is not decidable. □

The Halting Problem

Why does the halting problem matter? For a few reasons.

1. It proves the existence of undecidable (i.e., uncomputable) problems, establishing a fundamental limit to what can be computed
2. In his seminal paper [On Computable Numbers](#), Turing reduced the Halting Problem to the Entscheidungsproblems, meaning if you could solve the latter, you could solve the former. But since we proved you cannot solve the former, you cannot solve the latter.
3. As a sort of byproduct of all of this, he invented the notion of the stored program computer!

A Turing Unrecognizable Problem

Consider the complement of A_{TM} :

$$\overline{A_{TM}} = \{ \langle M, w \rangle : \text{TMM does not accept string } w \}$$

We already showed A_{TM} was Turing-recognizable: simulate M on w using our universal machine U and output *accept* if M outputs *accept*. But what about $\overline{A_{TM}}$?

Theorem 3.

$\overline{A_{TM}}$ is Turing-unrecognizable.

A Turing Unrecognizable Problem

Once again, towards a contradiction let's begin by assuming $\overline{A_{TM}}$ is Turing-recognizable. Let U be a TM recognizing A_{TM} and V be a TM recognizing $\overline{A_{TM}}$. We now show that we can use U and V to build a TM W to decide A_{TM} .

W works as follows: Given input $\langle M, w \rangle$ where M is a TM and w is a string:

1. Run U and V in parallel on w
2. If U accepts, halt and output *accept*
3. If V accepts, halt and output *reject*

A Turing Unrecognizable Problem

Let's consider what we've accomplished with W :

- ▶ W is a decider. It accepts when U accepts, it rejects when V accepts.
- ▶ W decides A_{TM} . It accepts when $\langle M, w \rangle \in A_{TM}$ and rejects when $\langle M, w \rangle \notin A_{TM}$

But once again, A_{TM} is undecidable. Therefore we have reached a contradiction somewhere. But where? Well we know A_{TM} is recognizable, thus our assumption that $\overline{A_{TM}}$ is recognizable must be false. □