SE 3310b

Theoretical Foundations of Software Engineering

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

Aleksander Essex

Western Engineering

Turing Completeness and Universal Computation

Implications of the Church-Turing Thesis

Recall the **Church-Turing thesis** essentially states that if a problem P can be solved 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

Western Engineering

Implications of the Church-Turing Thesis

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

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

then by the Church-Turing thesis I can prove:

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

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

Western Engineering

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

Western Engineering

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

Western Engineering

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):

```c
// 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;
};
```

Western Engineering

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.

Western Engineering

Undecidability and the Halting Problem

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…

Western Engineering

Universal TMs and Simulation

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

Definition 2 (String encoding notation)**.**

Let M be a Turing machine. We use the notation $\langle M \rangle$ to denote the description of M encoded as a string.

Additionally we use notation $\langle M, w \rangle$ to denote a string describing a Turing machine M, and input w. We can provide $\langle M, w \rangle$ as input to U, and design U to simulate the execution of M on input w.

Western Engineering

Some Aspects of U

- U is a computer, M is a program, w is the input argument.
- The 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.

Western Engineering

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:

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

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 $\langle M, w \rangle$.

Western Engineering

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.

Western Engineering

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.

Western Engineering

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.

Western Engineering

A Table of all TMs and Their String Encodings.

Define the following table: let row i correspond to the i-th Turing machine M_i. Let column j correspond to the string encoding of M_i, i.e., $\langle M_i \rangle$.

Western Engineering

Run H on the Table

For each row M_i and each column $\langle M_j \rangle$, run H on $\langle M_i, M_j \rangle$, and fill in the results in cell (i, j), i.e., M_i accepts string $\langle M_j \rangle$, fill in (i, j) with `Accept`. Otherwise fill in `Reject`.

The diagonal of this table, i.e., all cells (k, k) for $k \geqslant 1$ represent all Turing machines M_k fed their own description $\langle M_k \rangle$ as input. Some will accept, some will reject, and some will loop.

We can now define two sets: those TMs that will accept their own encoding, and those that don't.

Why are we doing this?? There's no special significance to feeding a TM its own description as input, except toward helping us achieve our contradiction.

Western Engineering

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:

$$L = \{\langle M_i \rangle : M_i \text{ does not accept input } \langle M_i \rangle\}$$

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., M_n can decide language L.
What is the language M_n accepts? It accepts all strings $\langle M_k \rangle$ where Turing machine M_k accepts input $\langle M_k \rangle$.

Western Engineering

Running TM's on Their Own Description

Thus M_n can be described as follows:

Given input string $\langle M_i \rangle$:

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

Notice M_n does two things: it runs M_i own description $\langle M_i \rangle$, and outputs the *opposite* of what H outputs.

Western Engineering

M_n on its Own Description

Consider now what happens when M_n is run on its own description $\langle M_n \rangle$?

Given input string $\langle M_n \rangle$ and our description of M_n we have:

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

Recall, H accepts $\langle M_n, \langle M_n \rangle \rangle$ if M_n would have accepted $\langle M_n \rangle$ as input.

Western Engineering

M_n Cannot Exist

Let's consider both cases of M_n run on its own input $\langle M_n \rangle$:

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

Conclusion: M_n cannot exist.

Western Engineering

A_{TM} is Undecidable

By way of a logical paradox we see M_n cannot exist. But M_n merely does the *opposite* of what H does. Therefore by extension H cannot exist. And since H decides A_{TM}, therefore A_{TM} is undecidable. $\square$

Western Engineering

The Halting Problem

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

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

The language $HALT$ consists of all Turing machines that halt i.e., either acccept 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?

Western Engineering

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 3.

$HALT$ is undecidable.

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

Western Engineering

The Halting Problem

Let S be a Turing machine that decides $HALT$. To obtain the contradiction, we will show that if S decides $HALT$, we can use it to build a Turing machine T to 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 S told us it will halt eventually).
    - If M accepts, halt and output *accept*
    - If M rejects, halt and output *reject*

Western Engineering

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. □

Western Engineering

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 Entschiedungsproblems, 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!

Western Engineering

A Turing Unrecognizable Problem

Consider the complement of A_{TM}:

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

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 4.

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

Western Engineering

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*

Western Engineering

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 $< M, w > \notin A_{TM}$

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. □

Western Engineering