

The Church-Turing Thesis

We now have four “general purpose” forms of computation: Turing machines, λ -calculus, grammars and partial recursive functions.

Each of the four can be transformed into one of the other three. Thus, they are all “equally powerful.”

Church-Turing Thesis: Turing machines are formal versions of algorithms, and no computational procedure will be considered an algorithm unless it can be presented as a Turing machine.

It is called a “thesis” because it is unprovable: it asserts that a certain informal concept (algorithm) corresponds to a mathematical concept (Turing machine).

How to disprove the Church-Turing Thesis: Devise another model of computation that does finite work at each step, and which can compute functions not computable by any Turing machine.

Universal Turing Machines

In order to be able to study the limits of Turing machines, we need a general method of specifying them, so that their specification can be used as input to other Turing machines (if we regard a Turing machine as a program, then we are treating programs as data).

We can create a *Universal Turing machine*:

- input is a specification for a Turing machine M and a string w .
- output is the output from M on w .

Thus, for a universal Turing machine U , we must have:

$$U(M, w) = M(w)$$

The Halting Problem

Problem: Given a Turing Machine M and a string w , will M halt when run on w ?

If there is a Turing machine M_0 that solves this problem, then an algorithm to convert a Turing-accepting machine M_1 to a Turing-deciding machine is:

- Feed M_1 and its input w to M_0 .
- If the answer is “halts” then output Y .
- Otherwise, output N .

This crucial question is called the *Halting Problem*.

The Halting Problem

Can a Java program be written to solve the halting problem?

- Input: A program P and input X .
- Output: “Halts” if P halts when run with X as input. “Does not Halt” otherwise.

Theorem: There is no program to solve the halting problem.

Proof: (by contradiction).

Assumption: There is a Java program that solves this problem.

```
boolean halt(Prog prog, String input)
{
    if (prog does halt on input)
        return true;
    else
        return false;
}
```

The Halting Problem

```

boolean selfhalt(Prog prog)
{
    // Return true if program halts
    // when given itself as input.
    if (halt(prog, prog))
        return true;
    else
        return false;
}

void contrary(Prog prog)
{
    if (selfhalt(prog))
        // Go into an infinite loop
        while(true);
}

```

The Halting Problem

What happens when the function `contrary` is run on itself?

Case 1: `selfhalt` returns `true`.

- `contrary` will go into an infinite loop.
- This contradicts the result from `selfhalt`.

Case 2: `selfhalt` returns `false`.

- `contrary` will halt.
- This contradicts the result from `selfhalt`.

Either result is impossible.

The only flaw in this argument is the assumption that `halt` exists.

Therefore, `halt` cannot exist.

The Halting Problem

The previous proof is essentially a proof by *diagonalisation*.

More explicitly, if the halting problem is decidable, then we can construct a table for all possible programs and all possible inputs, which indicates whether or not a given program halts for a given input. This table can be viewed as follows:

	$Input_1$	$Input_2$	$Input_3$	\dots
$Prog_1$	Yes	No	Yes	\dots
$Prog_2$	Yes	No	No	\dots
$Prog_3$	No	Yes	Yes	\dots
\vdots	\vdots	\vdots	\vdots	\dots

We then construct a program P such that $P(Input_i)$ halts if $Prog_i(Input_i)$ does not halt, and $P(Input_i)$ does not halt if $Prog_i(Input_i)$ does halt.

If the halting problem is decidable, then P must be contained in the enumeration of programs above. However, P differs from the j^{th} program for $Input_j$, so we have a contradiction.

Reducibility

In order to prove that other problems are undecidable, rather than perform more diagonalisation proofs, we show that these problems can be *reduced* to another problem which is known to be undecidable.

For example, the problem of determining whether a program (machine) halts on the empty input is undecidable.

Proof:

- Suppose that M_0 decided the language $\{L(M) : M \text{ accepts } \epsilon\}$
- Given arbitrary machine M and string w , we can create a new machine M_w that operates as follows on empty input:
 - Write w on the tape.
 - Simulate the execution of M .
- Now, we can apply M_0 to M_w to solve the original halting problem.
- Thus, M_0 must not exist.

Other Undecidable Problems

- Given a Turing machine M and an input string w , does M halt on input w ?
- Given a Turing machine M , is there any string at all on which M halts?
- Given a Turing machine M , does M halt on every input?
- Given two Turing machines M_1 and M_2 , do they halt on the same input strings?
- Given a Turing machine M , is the language M accepts regular? Is it context-free? Is it Turing-decidable?
- Does a particular line (transition) in a program (machine) get executed?
- Does a program contain a “computer virus”?

Rice’s Theorem: Let F be any proper, non-empty set of functions defined by Turing machines. The following problem is undecidable: Given any Turing machine, is its corresponding function a member of F ?