Undecidability

Terminology

A **decision problem** is a question with a yes or no answer on a given input. We say a decision problem is **decidable** if and only if there exists an algorithm to solve it. Otherwise, we say that the decision problem is **undecidable**.

The Halting Problem

Input A string `P` and a string `I`. We think of `P` as a program.

Output `True` if `P` halts on `I` and `False` if `P` goes into an infinite loop on `I`.

Turing's Theorem There is no program to solve the Halting Problem.

Proof Assume towards a contradiction there exists a program `Halt(P, I)` that solves the halting problem, i.e. `Halt(P, I)` returns `True` if and only if `P` halts on `I`. Given this program for the halting problem, we could construct the following algorithm Z

```
Program (String X)
    If Halt(X, X) then
        Loop Forever
    Else Halt.
End.
```

Then if `Z` is run with `Z` as input:

Case 1 Program `Z` halts on `Z`, meaning the `if` guard `Halt(Z, Z)` fails, `Z` does not halt on `Z`. Contradiction.

Case 2 Program `Z` runs forever on `Z`, meaning the `if` guard `Halt(Z, Z)` holds, `Z` does halt on `Z`. Contradiction.

Hence the program `Halt(P, I)` cannot exist. □

Reduction and Consequences

A typical way to prove that a problem is undecidable is to use reduction. If a problem can be reduced to the halting problem, it is undecidable.

A problem A is **reducible** to problem B if a solution to B could be used to solve A. If A has been proven to be an undecidable problem, to prove that a new problem B is undecidable, it is sufficient to show that a solution B could be used to decide A. This yields a contradiction since it was already proven that A is undecidable, and therefore, B is also undecidable.

Proving Undecidability

Proof Template

Solution Assume towards a contradiction there exists an algorithm `B` that solves this problem.

```
/* Suppose B solves the given problem */
program B(program X) {
    ...
}

/* Show that A(P, I) is equivalent to Halt(P, I) */
program A (program P, input I) {
    program P’ (_) {
        return P(I);
    }
    return B(P’);
}
```

Halting-No-Input Problem

Problem Given a program `P` that requires no input, does `P` halt?

Solution Assume towards a contradiction there exists an algorithm `B` that solves this problem.

```
/* Suppose B solves the halting-no-input problem */
program B(program X) {
    if (Halt(X))
        return True;
}

/* Show that A(P, I) is equivalent to Halt(P, I) */
program A (program P, input I) {
    program P’ (_) {
        return P(I);
    }
    return B(P’);
}
```

`A` returns `True` iff `B(P’)` returns `True` iff `P’` halts iff `P(I)` halts. Hence `A(P, I)` solves the halting problem. By Turing's theorem, such `A` should not exists. Contradiction. Hence `B` does not exist. The problem is undecidable. □

Both-Halt Problem

Problem Given two programs P1 and P2 that take no input, do both programs halt?

```
program B(program X1, program X2) {
    if (Halt(X1) && Halt(X2))
        return True;
}

program A (program P, input I) {
    program P' (_) {
        return P(I);
    }
    return B(P', P');
}
```

A returns True iff B(P', P') returns True iff P' halts iff P(I) halts. Hence A(P, I) solves the halting problem. By Turing's theorem, such A should not exists. Contradiction. Hence B does not exist. The problem is undecidable. ☐

Program-agreement Problem

Problem Given two programs P1 and P2, do they agree on all inputs? We say that two problems agree on all input if and only if, for every input , either they both run forever, or they both halt and return the same value.

```
/* Suppose B solves the program-agreement problem */
program B(program X1, program X2) {
    if ((!Halt(X1) && !Halt(X2)) || (forall x:  X1(x) == X2(x)))
        return True;
}

/* We shall show A solves the halting problem */
program A (program P, input I) {
    program P1 (program _) { return P(I); }
    program P2 (program _) { return True; }
    return B(P1, P2);
}
```

A returns True iff B(P1, P2) returns True iff both runs forever or both returns the same value for all inputs. As P2 halts and returns True, P1 must halt as well. P1 halts iff P(I) halts. Hence A(P, I) solves the halting problem. By Turing's theorem, such A should not exists. Contradiction. Hence B does not exist. The problem is undecidable. ☐

Hoare-triple Total Correctness Problem

Problem Given a hoare triple, is the triple satisfied under total correctness?

```
/* Suppose B solves the hoare-triple total correctness problem */
program B(Hoare<P, C, Q>) {
    if (Halt(C) && (Hoare<P, C, Q> is valid))
        return True;
}

/* We shall show A solves the halting problem */
program A (program P, input I) {
    program P' (program _) { return P(I); }
    return B(Hoare<true, P', true>);
}
```

`A` returns `True` iff `B(Hoare<true, P', true>)` returns `True` iff `P'` halts (and the triple is valid) iff `P(I)` halts. Hence `A(P, I)` solves the halting problem. By Turing's theorem, such `A` should not exists. Contradiction. Hence `B` does not exist. The problem is undecidable. □

Hoare-triple Partial Correctness Problem

Problem Given a hoare triple, is the triple satisfied under partial correctness?

```
/* Suppose B solves the hoare-triple partial-correctness problem */
program B(Hoare<P, C, Q>) {
    if (!Halt(C) || (Hoare<P, C, Q> is valid))
        return True;
}

/* We shall show A solves the halting problem */
program A (program P, input I) {
    program P' (program _) { return P(I); }
    return (!B(Hoare<true, P', false>));
}
```

`A` returns `True` iff `B(Hoare<true, P', true>)` returns `False` iff `P'` halts and the triple is invalid. If `P'` halts (i.e. `P(I)` halts), as the postcondition is never satisfied so the triple is always invalid, `A` returns True. If `P'` does not halt (i.e. `P(I)` does not halt), `B` returns `True`, `A` returns `False`. Hence `A(P, I)` solves the halting problem. By Turing's theorem, such `A` should not exists. Contradiction. Hence `B` does not exist. The problem is undecidable. □