5CCS2FC2: Foundations of Computing II

The Limits of Computation

Week 2

Dr Christopher Hampson

Department of Informatics

King's College London

Warm-up : A simple domino game

- **A simple domino game**
  - Can you arrange the following dominos so that the string spelled out by the **top row** matches the string spelled out by the **bottom row**?

c	ccb	ac	aa	bca
aac	ac	a	ccb	bc

(you are not permitted to *rotate* the tiles)

Decidablility and Undecidability

Decidablility and Undecidability

- **Decision Problems**

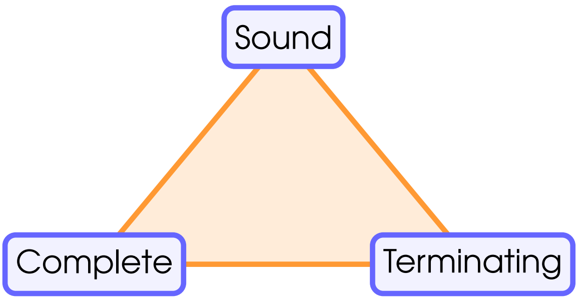
  - Every **problem**, for which the possible answers are **YES** or **NO** (or, alternatively, **True** or **False**) can be interpreted as a language,

$$L_P = \left\{ w \in \Sigma^* \ : \ \begin{array}{c} w \text{ is an instance of the problem } P \\ \text{whose answer is } \textbf{YES} \end{array} \right\}$$

(for some suitable choice of alphabet Σ)

Decidablility and Undecidability

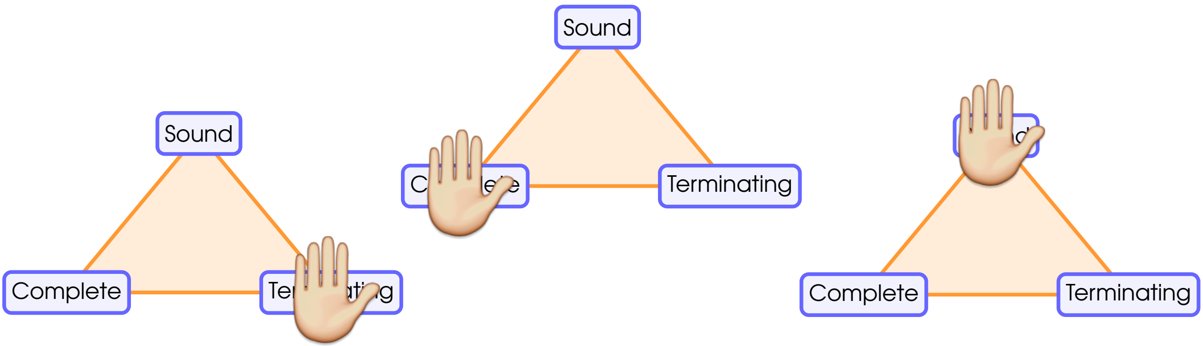
- **Decidable Languages / Problems**
  - A problem P is said to be **decidable** if there is *some* Turing Machine / algorithm that is:
    - **Sound** If the algorithm returns **True** then $w \in L_P$
    - **Complete** If $w \in L_P$ then the algorithm returns **True**
    - **Teminating** The algorithm always terminates on all inputs



Decidablility and Undecidability

- **Undecidable Languages / Problems**
  - A problem P is said to be **undecidable** if it is *not* decidable.

    In other words, it is **impossible** to construct an algorithm for P that is **sound**, **complete** *and* **terminating**.



Do any Undecidable problems exist?

Decidablility and Undecidability

- **Examples of Decidable Problems**
  - Checking if a Finite Automata accepts an input word
  - Checking whether a Finite Automata does not accept any words
  - Checking whether two Finite Automata are equivalent
  - Checking whether a word is a palindrome
  - Checking whether a Boolean formula is satisfiable,
  - etc. etc.

- **Encoding Turing Machines as Strings**
  - Since Turing Machines contain only **finitely many** states and instructions, we can encode them as a finite **string** over some alphabet.

(*e.g.* by writing each transition in unicode/ASCII)

```java
public static boolean progM(String w) {
  // An example of a Turing Machine / algorithm
  String[] tape = w.split("");  // write w on tape
  String state = "q_init";  // curent state of TM
  int position = 0; // current head position

  if (tape[position] == "0") {
    tape[position] = "1";
    position = position + 1;
    state = "q1";
  }
}
```

- **Encoding Turing Machines as Strings**
  - Since Turing Machines contain only **finitely many** states and instructions, we can encode them as a finite **string** over some alphabet.

(*e.g.* by writing each transition in unicode/ASCII)

```
String M = "public static boolean progM(String w) {
        // An example of a Turing Machine / algorithm
        String[] tape = w.split(‘’);  // write w on tape
        String state = ‘q_init’;  // curent state of TM
        int position = 0; // current head position

        if (tape[position] == ‘0’) {
          tape[position] = ‘1’;
          position = position + 1;
          state = ‘q1’;
        }
      }";
```

Universal Turing Machines

- **Why is this relevant?**
    - If a Turing Machines can be encoded as a string, is should be possible to construct a Turing Machines that can read the encodings of other Turing Machines.

Caution!

Be mindful to distinguish between the TM as an abstract algorithm and the encoding of the TM which is a string.

Only the *encodings* of TMs can be read by other TMs.

Universal Turing Machines

- **Universal Turing Machine**
  - A **Universal Turing Machine** is a Turing Machine $\mathcal{M}_u$ that takes as input a pair $\langle M, w \rangle$, where
    - M is an **encoding** of a Turing Machine $\mathcal{M}$,
    - $w \in \Sigma^*$ is an **input word** intended for $\mathcal{M}$

    with the property that

$$\mathcal{M}_U \text{ accepts } \langle M, w \rangle \iff \mathcal{M} \text{ accepts } w$$
$$\mathcal{M}_U \text{ rejects } \langle M, w \rangle \iff \mathcal{M} \text{ rejects } w$$

A Universal Turing machine acts like a **compiler** or an **interpreter** that reads the *software* M, and simulates the operation of $\mathcal{M}$ on the input w. It will accept if $\mathcal{M}$ accepts, reject if $\mathcal{M}$ rejects and get struck if $\mathcal{M}$ gets stuck.

Universal Turing Machines

- **Universal Turing Machine**

```
public static boolean UTM(String M, String w) {
  // This algorithm returns ‘true’ if M
  // accepts on input w and returns ‘false’
  // if M rejects on input w.

  // Parse M and return a machine with a method ‘run’
  Machine progM = parse_string(M)

  // Run progM on input w
  return progM.run(w)
}
```

(for some subroutine `parse_string` that returns a new type `Machine`)

Common Software Issues

- **What common issues do you face when writing programs?**
  - Syntax Error
  - Missing end-of-statement delimiters
  - Unmacthed brackets / parenteses
  - Doesn't do what it is supposed to do (Human error)
  - Get's stuck in a loop!
  - Uses too much memory / takes too long.

Can we write an algorithm to check our code for bugs?

The Halting Problem

The Halting Problem

- **The Halting Problem**
  - The **Halting Problem** the the decision problem that asks whether a given Turing Machine will **terminate** on a given input word.

$$\boxed{\textbf{HALT}_{TM} = \{\, \langle M, w \rangle \ : \ \mathcal{M} \text{ terminates on input } w \}}$$

The Halting Problem

Theorem The Halting Problem $\textbf{HALT}_{TM}$ is Undecidable.

Proof:

Step 1) Suppose, to that contrary that $\textbf{HALT}_{TM}$ is **decidable**, then there must be some algorithm `HALT`.

```
public static boolean HALT(String M, String w) {
  // This algorithm returns 'true' if M
  // terminates on input w and returns 'false'
  // if M does not terminate on input w.
  // This algorithm always terminates.
       ⋮
}
```

Step 2) We want to construct a program on which `HALT` will not give the correct answer.

```java
public static void progX(String M) {

    // Ask HALT whether program M halts on input M
    boolean ans = HALT(M,M);

    if (ans == true) {
        // Loop forever
        while (true) {
            System.out.println("Hello World");
        }
    } else {
        // Terminate successfully
        System.out.println("Goodbye!");
    }
}
```

The Halting Problem

Step 3) What does `progX` do when we run it on its **own code**?

It's behavior will depend on whether `HALT(progX,progX)` returns **True** or **False**.

Case 1) Suppose that `HALT(progX,progX)=true`,

(i.e. progX halts on its own input, according to `HALT`).

However, in line 4, we then enter an infinite loop, and so progX does not halt on its own input.

Case 2) Suppose that `HALT(progX,progX)=false`,

(i.e. progX does not halts on its own input, according to `HALT`).

However, line 4, we jump to the `else` condition and halt after printing `Goodbye!`.

Step 4) If we assume that `HALT` works as advertised, we end up with a **paradox** that we cannot resolve.

Hence, there can be no algorithm for HALT$_{TM}$

Q.E.D

This argument is similar to the argument that no Finite Automata accepted the language $a^n b^n$, for $n = 0, 1, 2, \ldots$:

(i) Assume there is a machine that solves the problem,

(ii) Construct an input for which the machine does not return the correct answer,

(iii) Conclude that no possible machine exists.

Other Undecidable Problems

- **(Incomplete) List of Undecidable Turing Machine Problems**
  - **The Accepting Problem**

$$\mathbf{A}_{TM} = \left\{ \langle M, w \rangle \ : \ \begin{array}{c} M \text{ encodes a TM that accepts} \\ \text{the input word } w \end{array} \right\}$$

  - **The Emptiness Problem**

$$\mathbf{E}_{TM} = \left\{ \langle M \rangle \ : \ \begin{array}{c} M \text{ encodes a TM that} \\ \text{rejects } \textit{all} \text{ input words} \end{array} \right\}$$

  - **The Equivalence Problem**

$$\mathbf{EQ}_{TM} = \left\{ \langle M_1, M_2 \rangle \ : \ \begin{array}{c} M_1 \text{ and } M_2 \text{ encode two TMs that} \\ \text{accept precisely the same words} \end{array} \right\}$$

Other Undecidable Problems

- **(Incomplete) List of Undecidable Turing Machine Problems**
  - **The Regular Language Problem**

$$\textbf{REGULAR}_{TM} = \left\{ \langle M \rangle : \begin{array}{c} M \text{ encodes a TM that} \\ \text{accepts a regular language} \end{array} \right\}$$

Bonus Fact!

A surprising example not directly related to Turing Machines is **Post's Correspondence Problem**, which asks whether a collection of word dominos (like the ones we saw in the warm-up exercise) has a solution. We allow for solutions where the same domino can be used multiple times if required.

(see Siper, Theorem 5.15 if interested)

Mapping Reductions

Mapping Reductions

- **Mapping Reduction**
  - A **mapping reduction** from a problem A to a problem B is a *computable* function $\boxed{f : \Sigma^* \to \Sigma^*}$ that maps instances A to instances of B such that

$$\boxed{w \in A \quad \Longleftrightarrow \quad f(w) \in B}$$

(we say that A is *reducible* to B, and write $A \leq_m B$)

If we have an algorithm for solving B, we can use the reduction to solve A by first **translating** the input w into $f(w)$ and using the algorithm for B as a **subroutine**.

The problem A must be ***at least as easy*** as B, since we can always solve A if we known how to solve B.

Theorem The Accepting Problem $\mathbf{A}_{TM}$ is Undecidable.

Proof:

We want to show that $\mathbf{A}_{TM}$ is **at least as hard** to solve as $\mathbf{HALT}_{TM}$.

Step 1) **Assume** that there is an program `ATM` that solves the accepting problem $\mathbf{A}_{TM}$.

```
public static boolean ATM(String M, String w) {
  // This algorithm returns ‘true’ if M accepts
  // w and returns ‘false’ if M rejects
  // or does not not terminate on input w.
  // This algorithm always terminates.
      ⋮
}
```

Mapping Reductions

Step 2) Construct a **new program** that solves the 'easier' problem $\textbf{HALT}_{TM}$, using `ATM` as a **subroutine**.

```
public static boolean HALT(String M, String w) {

    // Generate a new string M' of a program that
    // accepts w whenever M terminates on w
    String M' = "public static progM'(String w) {...}";

    return ATM(M',w);
}
```

Step 3) Note that

$$\texttt{HALT(M,w) = true} \iff \texttt{ATM(M',w) = true}$$

Mapping Reductions

Step 4) Or in other words

$$\langle M, w \rangle \in \textbf{HALT}_{TM} \iff \langle M', w \rangle \in \textbf{A}_{TM}$$

Step 5) We still need to demonstrate that we can **construct** M', else the program `HALT` will be incomplete.

```
public static boolean progM'(String w) {
  // Run M on input w (with Universal Turing
       Machine)
  boolean ans = UTM(M,w);

  if (ans == true) { return true; }
  if (ans == false) { return true; }
}
```

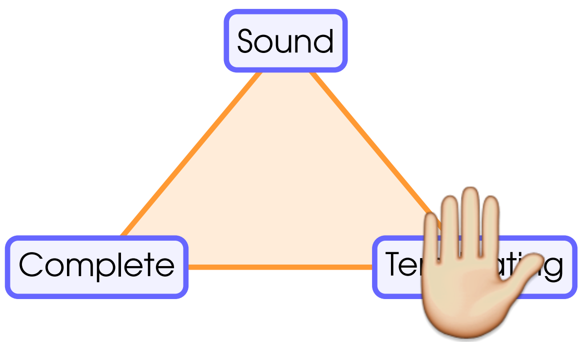
Q.E.D

Beyond Undecidability

Recursive Enumerability

- **Recursive Enumerability**
  - A decision problem A is said to be **recursively enumerable** (or **Turing-recognisable**) if there is an an algorithm such that
    - **Sound** If the algorithm returns `true` then $w \in A$
    - **Complete** If $w \in A$, then the algorithm returns `true`.

(the algorithm may not terminate for some $w \notin A$)



Recursive Enumerability

Theorem Let A be an arbitrary decision problem. If both A and $\overline{A}$ are recursively enumerable, then A is decidable.

Proof:

Step 1) Let $\mathcal{M}_1$ be a Turing Machine that **recognises** A, and let $\mathcal{M}_2$ be a Turing Machine that **recognises** $\overline{A}$.

Step 2) Run both $\mathcal{M}_1$ and $\mathcal{M}_2$ in **parallel** — Eventually one will terminate since they are both sound and complete

Step 3) **Case 1)** If $w \in A$ then $\mathcal{M}_1$ will eventually terminate, so return **True**.

Case 2) If $w \notin A$, then $w \in \overline{A}$ and so $\mathcal{M}_2$ will eventually terminate, so return **False**.

Q.E.D

Recursive Enumerability

> **Theorem** The Halting problem **HALT**$_{TM}$ is undecidable but recursively enumerable.

Proof:

Step 1) Since **HALT**$_{TM}$ is undecidable, it is enough to **construct** an algorithm that will return **True** on input $\langle M, w\rangle$ whenever $\mathcal{M}$ terminates on w.

```
1 public static boolean HALT(String M,String w) {
2     // Run M on input w (with universal Turing Machine)
3     boolean ans = UTM(M,w);
4     return true;
5 }
```

(the algorithm always returns `true`, or gets stuck in a loop in line 3)

Q.E.D

Corollary The *Non*-halting Problem $\overline{\textbf{HALT}_{TM}}$ is *not* recursively enumerable.

Proof:

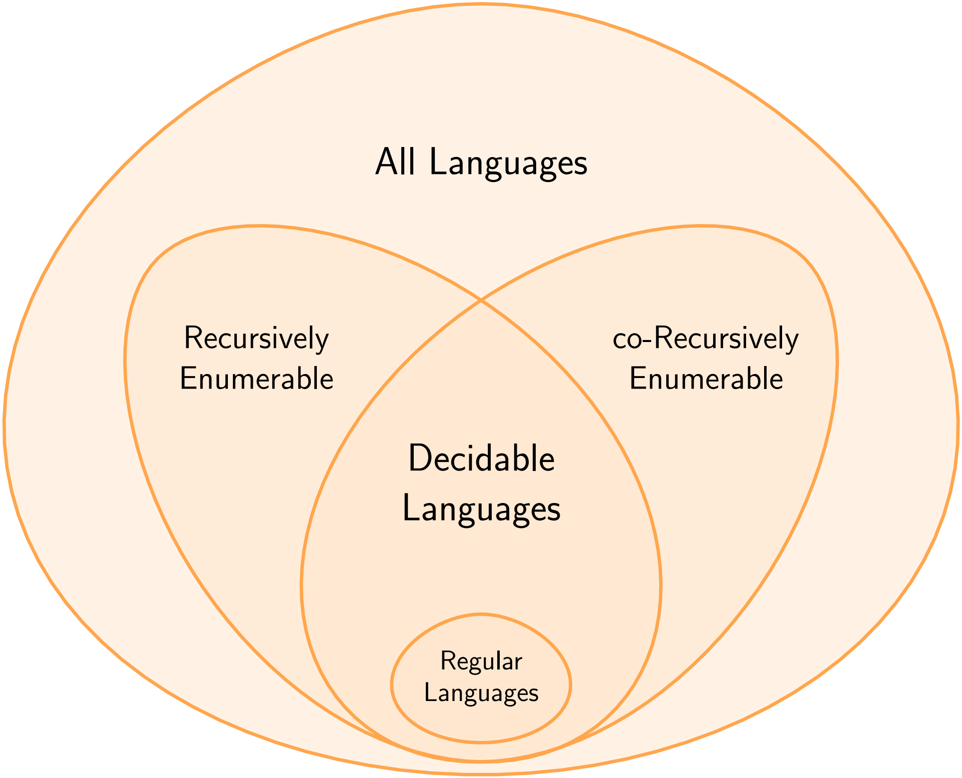
Step 1) We know that $\textbf{HALT}_{TM}$ is **undecidable** but **recursively enumerable**.

Step 2) If $\overline{\textbf{HALT}_{TM}}$ were also **recursively enumerable**, then by our earlier theorem, $\textbf{HALT}_{TM}$ would be decidable!

Q.E.D

We say that the Non-halting problem is **co-recursively enumerable**, since its *complement* is recursively enumerable.

Complexity Hierarchy



Appendix

Appendix

Theorem The emptiness problem $\mathbf{E}_{TM}$ is Undecidable.

Proof:

Step 1) Assume there is a program `ETM` that solves the emptiness problem $\mathbf{E}_{TM}$, and construct a **new program** that solves the 'easier' problem $\mathbf{A}_{TM}$, using `ETM` as a **subroutine**.

```
public static boolean ATM(String M,String w) {
    // Generate a new string Mw of a machine whose
    // langauge is empty if and only if M accepts w
    String Mw = "public static progMw(String s) {...}";
    return ETM(Mw);
}
```

Appendix

Step 2) We note that

$$\langle M, w\rangle \in \mathbf{A}_{TM} \iff \langle M_w\rangle \in \mathbf{E}_{TM}$$

Step 3) However, we still need to demonstrate that we can construct $\mathcal{M}_w$

```java
public static boolean progMw(String s) {
    // Run M on built-in constant w
    String w = "[whatever the word w was]";
    boolean ans = progM(w);

    if (ans == true) { return false; }
    if (ans == false) { return true; }
}
```

(note that this *input* s is always ignored, and uses the constant w)

Q.E.D

End of Slides!

🎉

Feedback

- **Let me know how you found today's lecture!**

https://goo.gl/forms/5lDHqW9JOFzN9KSC2