CS411 Theory of Computation
Lecture 6

Sergey Kitaev

University of Strathclyde

4 October 2017

Enumerators

- An enumerator is a variant of a Turing machine with a printer attached.
- However, an enumerator E starts with a blank input tape.
- If E doesn't halt then it may print an infinite list of strings.
- The language enumerated by E is the collection of all strings that it eventually prints out.
- (These strings can be generated in any order and with possible repetitions.)
- How are enumerators and TMs related?

Theorem (3.21)

A language is Turing-recognizable if and only if some enumerator enumerates it.

Proof $\Leftarrow$.

(Enumerator E enumerates a language $A \Longrightarrow$ TM M recognizes A.)

The TM is specified in the following way:

$M =$ "On input w:

1. Run E. Every time E outputs a string, compare it with w.
2. If w ever appears in the output of E, then accept."

From this, M accepts those strings that appear in E's list. $\square$

Proof $\Rightarrow$.

TM M recognizes a language $A \Longrightarrow$ we can construct an enumerator of it.

Suppose that $\Sigma^* = \{s_1, s_2, \ldots\}$.

$E =$ "Ignore the input:

1. Repeat the following for $i = 1, 2, 3, \ldots$
2. Run M for i steps on each input s_1, s_2, $\ldots$, s_i.
3. If any computations accept then print out the corresponding s_j."

If M accepts a particular string s, it will eventually appear on the list generated by E. $\square$

Summary of what we have achieved

It has been shown that several variants of Turing machines are equivalent in power.

- All these shared the essential feature of having unrestricted access to unlimited memory.
- All models with this feature turn out to be equivalent in power so long as they satisfy some reasonable requirements.

Consider the analogous situation for programming languages:

Q: Can some algorithm be programmed in one language but not another?

A: No.

The tasks programming languages can complete are in a way equivalent.

This equivalence phenomenon $\Rightarrow$ the algorithms one can accomplish are machine-independent.

The definition of algorithm

An algorithm is (intuitively) some set of instructions for doing something with an input.

Usually this intuitive description is sufficient when it comes to easy questions.

However, the intuitive notion is not much use when it comes to challenging questions: ones which touch on the limits of computability.

Polynomial digression

A polynomial is a sum of products of collections of variables.

Suppose we have variables x, y and z. Then

$$x \cdot y + y \cdot z + x \cdot y \cdot z + z \cdot x \cdot y + x + z \cdot y$$

is a polynomial, and this equals $x + xy + 2yz + 2xyz$.

The coefficient of yz is 2 and the coefficient of x is 1.

A root of a polynomial is an assignment of values such that the polynomial evaluates to *zero*.

For example if $(x, y, z) = (-1, -1, -1)$ then $x + xy + 2yz + 2xyz = 0$, so $(-1, -1, -1)$ is an integral root of the polynomial.

An old question

Hilbert (1900) asked whether there was an algorithm (a process) that tests whether a polynomial has an integral root.

Then

How to approach problem?

Intuitive notion of algorithm adequate for certain things, but not for non-existence proofs.

Needs something more.

Now

1936: Church used λ calculus to define algorithms, Turing used his machines.

It turns out the two definitions are equivalent.

Church-Turing Thesis:
Intuitive notion of algorithm $\equiv$ Turing machine algorithms

1970: Matijasevic showed that no algorithm exists for testing whether a polynomial has integral roots, algorithmically unsolvable.

In our terminology: let

$$D = \{p \mid p \text{ is a polynomial with an integral root}\}.$$

Hilbert's question: Is D decidable?

Answer: No

However: D is Turing-recognizable.

Consider the simpler problem:

$$D_1 = \{p \mid p \text{ is a polnomial in } x \text{ with an integral root}\}.$$

Here is a Turing machine $\mathcal{M}_1$ that recognizes D_1:

$\mathcal{M}_1 =$ "Input is polynomial p over variable x

1. Evaluate p with $x = 0, 1, -1, 2, -2, 3, -3, \ldots$
   If at any point p evaluates to 0, then accept."

A similar Turing machine $\mathcal{M}$ recognizes D.

Note that $\mathcal{M}_1$ and $\mathcal{M}$ are recognizers, but not deciders.

We can convert $\mathcal{M}_1$ to be a decider for D_1- we do this by using a mathematical result that provides us with bounds on where the roots may be (if there are any!).

- If $p(x) = c_1x^{k-1} + c_2x^{k-2} + \ldots + c_{k-1}x + c_k$ then the roots (if any) lie in the range:

$$\left[-k\frac{c_{max}}{c_1}, +k\frac{c_{max}}{c_1}\right]$$

  where c_{max} is a value one can read from the polynomial.

- If a root is not found in this range then **Reject**.

Matijasevic's theorem proves that calculating such bounds for multivariate polynomials is impossible.

Describing Turing machines

Standardize the way we describe Turing machine algorithms:

Formal description: (Lowest level) Specifies everything.

Implementation description: English prose describes way head moves and things are stored.

High-level description: English used to describe algorithm, implementation details ignored.

The high-level description is now sufficient for our purposes now that you 'understand' low-level.

Moving forward:

- the input to any Turing machine will be a string $\langle Obj \rangle$ that represents an object Obj.
- Sometimes the input will be a string $\langle Obj_1, \ldots, Obj_k \rangle$ that represents a collection of objects $Obj_1, \ldots, Obj_k$.