CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 21: The Church–Turing Thesis

Harvard SEAS - Fall 2021 *Nov. 16, 2021*

1 Announcements

Recommended Reading:

- MacCormick §5.6–5.7, §7.7–7.9
- PS10 distributed tomorrow, due Dec 1
- PS11 due at end of reading period
- Please do not distribute solutions + put code on Github

2 The Church–Turing Thesis

We have seen that RAM programs cannot solve many computational problems about other RAM programs. To what extent does this apply to other models of computation?

Theorem 2.1 (Turing-equivalent models)**.** *If a computational problem* Π *is solvable in one of the following models of computation, then it is solvable in all of them:*

- *RAM programs*
- *Word-RAM programs*
- *XOR-extended RAM or Word-RAM programs*
- *%-extended RAM or Word-RAM programs*
- *Python programs*
- *OCaml programs*
- *C programs (modified to allow a variable/growing pointer size)*
- *Turing machines*
- *Lambda calculus*
- $\vdots$

Moreover, there is an algorithm (e.g. a RAM program) that can transform a program in any of these models of computation into an equivalent program in any of the others.

Proof idea: A theorem like this is proven via "compilers" and simulation arguments like we have seen several times, giving a procedure to transform programs from one model to another (e.g. simulating XOR-extended Word-RAMs by ordinary Word-RAMs, as on PS4). Like you have seen on PS4, we can write simulators for RAM programs in high-level languages like Python and OCaml, and conversely those high-level languages are compiled down to assembly code, which is essentially Word-RAM code.

Simple and elegant models: The λ calculus and Turing machines are extremely simple (even moreso than the RAM model) and mathematically elegant models of computation, coming from the work of Church and Turing, respectively, in 1936, in their attempts to formalize the concept of an algorithm (prior to, and indeed inspiring, the development of general-purpose computer technology). We will describe Turing machines in more detail in the next lecture and compare them to the Word-RAM model. We won't have time to describe the lambda calculus, but it provided the foundation for future functional programming languages like OCaml, and one of the theorems in Turing's paper established the equivalence of Turing machines and the λ calculus.

Input encodings: One detail we are glossing over in Theorem 2.1 is that the different models have different ways of representing their inputs and outputs. For example, natural numbers can be represented directly in RAM programs, but in a Turing machine they need to be encoded as a string (e.g. using binary representation), and in the lambda calculus, they are represented as an operator on functions (which maps a function $f(x)$ to $f^{(n)}(x) = f(f(\cdots f(x)))$. So to be maximally precise, these models are equivalent up to the representation of input and output.

Given Theorem 2.1, we can replace both the RAM program being analyzed and the RAM program carrying out the analysis with any of the equivalent models of computation. For example:

Input	: A Turing Machine M and an input x
Output	: `accept` if M halts on input x, `reject` otherwise

Computational Problem TM-Halting Problem

Corollary 2.2. *There is no Python program that solves the TM-Halting Problem.*

Proof.
Suppose for contradiction that there is a Python program P that solves the TM-Halting Problem. Then by Theorem 2.1, there is a RAM program P' that solves the TM-Halting Problem. We can then obtain a RAM program P'' that solves the original Halting Problem (for RAM programs), by having $P''(Q, x)$ first convert the RAM program Q into an equivalent Turing machine M (by Theorem 2.1) and then running $P'(M, x)$. Since we know that there is no RAM program that solves the Halting Problem for RAM programs, we have a contradiction.

□

The Church–Turing Thesis: The equivalence of many disparate models of computation leads to the Church–Turing Thesis, which has (at least) two different variants:

1. The (equivalent) models of computation in Theorem 2.1 capture our intuitive notion of an algorithm.

2. Every physically realizable computation can be simulated by one of the models in Theorem 2.1.

The Church–Turing Thesis is not a precise mathematical claim, and thus cannot be formally proven. However, it has stood the test of time very well, even in the face of novel technologies like quantum

computers (which have yet to be built in a scalable fashion); every problem that can be solved by a quantum algorithm can also be solved by a RAM program, albeit much more slowly (as far as we know). Considering computational efficiency when comparing models of computation will be the subject of next class!

3 Unsolvability Summary

Many computational problems about general algorithms/programs are unsolvable. For example:

- Every nontrivial problem about the input–output behavior of the program. (Rice's Theorem. Includes Halting, HaltOnEmpty, Solves3Coloring as special cases.)
- Estimating the asymptotic runtime of the program. (Today's Active Learning exercise)
- Detecting bugs (e.g. buffer overflows) that can occur in arbitrarily long computations of the program

In general, questions that relate to arbitrarily long computations of general programs (i.e. from one of the models of computation in Theorem 2.1) are usually unsolvable. However, as you are seeing on PS9, there are nontrivial problems about programs that *can* be solved, such as:

- Questions about what an execution does within a bounded amount of time steps.
- Syntactic properties of programs.
- Compiling programs into other models/languages (as in Theorem 2.1) or to enforce certain properties of the program (e.g. type safety, run time, no memory leaks).
- Problems about some restricted sets of programs that don't have the full power of those in Theorem 2.1.

The phenomenon of unsolvability is not limited to problems about programs. One important example is the following.

Input	: A multivariate polynomial $p(x_0, x_1, \ldots, x_{n-1})$ with integer coefficients
Output	: `accept` if there are natural numbers $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$ such that $p(\alpha_0, \alpha_1, \ldots, \alpha_{n-1}) = 0$, `reject` otherwise

Computational Problem Diophantine Equations

Example input:

$$p(x_0, x_1, x_2, x_3) = 7x_0^8 x_1^2 x_3 - 21x_2^3 + 3x_1^8 x_3^7 - \cdots + 5x_0 x_2 x_3$$

Algorithm for univariate polynomials $p(x) = ax^2 + bx + c$ of degree 2: there *is* an algorithm to solve this problem; namely by checking whether $\frac{-b \pm \sqrt{b^2-4ac}}{2a}$ lie in $\mathbb{N}$. More concretely, if we assume $a = 1/2$, we can solve the problem by checking if the discriminant $b^2 - 4ac$ is a perfect square.

Hilbert's 10th Problem: find an algorithm to decide solvability of Diophantine Equations in general. Posed by Hilbert in a famous address to the International Congress of Mathematicians

in 1900, as part of a list of 23 problems that Hilbert laid out as challenges for mathematicians to tackle in the 20th century. Several of Hilbert's problems were part of a general project to fully formalize and mechanize mathematics. Hilbert's 2nd problem was to prove consistency of the axioms of mathematics, which was shown to be impossible by Gődel's Incompleteness Theorem in 1931. Turing's work on the undecidability of the Halting Problem was also motivated by Hilbert's program, and was used by Turing to show that there is no algorithm to decide to truth of well-defined mathematical statements (since whether or not a Turing machine halts on a given input is a well-defined mathematical statement), resolving Hilbert's Entscheidungsproblem ("Decision Problem", posed by Hilbert and Ackermann in 1928). Hilbert's 10th problem asks about a very special case of the Entscheidungsproblem, and was finally resolved in 1970 through the work of several mathematicians:

Theorem 3.1 (Matiyasevich, Robinson, Davis, Putnam)**.** *Diophantine Equations is unsolvable.*

Another very simple unsolvable problem is the following:

Input	**:** A finite set T of square tiles with each of the four edges colored (using an arbitrarily large palette of colors)
Output	**:** `accept` if the entire 2-d plane be tiled using tiles from T so that colors of abutting edges match, `reject` otherwise

Computational Problem Tiling

Theorem 3.2. *Tiling is unsolvable.*