CS 121: Introduction to Theoretical Computer Science Fall 2019

Section 3

Prof. Boaz Barak

1 RAM Machines and NAND-RAM

One of the limitations of Turing Machines and NAND-TM programs is that we can only access one location of our arrays/tape at a time. When we think of computers, we usually have a notion of **Random Access Memory (RAM)**, which allows us to directly access arbitrary memory locations. This motivates us to consider some new computational models.

A model that allows such access to memory is the **RAM machine**. We won't go into the formal details of how this is defined, but we'll briefly go over what it roughly looks like. Like Turing machines, RAM machines have infinite memory. They also have a finite sized set of registers that also store data. The RAM machine can freely move data between the memory and registers and perform computations such as comparisons, arithmetic operations, and logical operations. Similarly to Turing machines, RAM machines choose what to do at each step based on its state, in this case captured by the content of the registers.

Just like we had NAND-TM for Turing machines, we also have **NAND-RAM** for RAM machines. NAND-RAM differs from NAND-TM in that it allows integer valued variables instead of just bit valued, indexed access to arrays, basic arithmetic operations and comparisons in addition to just NAND, and conditionals.

2 Cellular automata

In cellular automata, we have an infinite number of cells, each of which has a constant number of possible states. At each time step, a cell updates to a new state by applying some simple rule involving its own state and the states of its neighbors.

We'll briefly discuss **one-dimensional cellular automata**, where we have an infinite line of cells. They consist of a finite alphabet of symbols Σ as well as a transition function $r : \Sigma^3 \to \Sigma$.

The state of the automaton is captured by its configuration, which is defined as a function $A : \mathbb{Z} \to \Sigma$. If an automaton with rule r is in configuration A, its next configuration $A' = \text{NEXT}_r(A)$ is the function A' such that $A'(i) = r(A(i-1), A(i), A(i+1))$ for all $i \in \mathbb{Z}$. The idea is that the configuration captures the state of the automaton in a given step, and we can progress the next state of the automaton by applying the transition rule to the values of a cell and its neighbors. We'll revisit the idea of configurations, except this time in the context of Turing machines, when discussing equivalence of models.

3 Lambda calculus

A **lambda expression** (or λ-expression) is one of the following forms:

- Application: $e = (e', e'')$ where e' and e'' are λ expressions.
- Abstraction: $e = \lambda x.(e')$ where e' is a λ expression and x is a variable identifier.

We say that two λ expressions are equivalent if we can repeatedly apply the following rules to them and be left with the same expression:

- Evaluation (β reduction): The expression $(\lambda x.e)e'$ is equivalent to $e[x \to e']$ (i.e. the expression e with all instances of x renamed to e').
- Variable renaming (α conversion): The expression $\lambda x.e$ is equivalent to $\lambda y.e[x \to y]$

There are two main ways we can simplify application expressions such as, for instance, $(\lambda f.f)((\lambda x.xx)(\lambda y.y))$

In call by name evaluation, we perform function applications as soon as possible, so taking steps from above with call by name evaluation, we would be left with

$$(\lambda f.f)((\lambda x.xx)(\lambda y.y)) \to ((\lambda x.xx)(\lambda y.y)) \to (\lambda y.y)(\lambda y.y) \to (\lambda y.y)$$

In call by value evaluation, we perform applications only when the right-hand side is a value (i.e. fully simplified), so the next steps for the expression would be the following:

$$(\lambda f.f)((\lambda x.xx)(\lambda y.y)) \to (\lambda f.f)((\lambda y.y)(\lambda y.y)) \to (\lambda f.f)((\lambda y.y)) \to (\lambda y.y)$$

4 Equivalence

Theorem 1. *For every function $F : \{0,1\}^* \to \{0,1\}^*$, F is computable by a NAND-TM program if and only if F is computable by a NAND-RAM program.*

Proof idea. We immediately have that NAND-RAM is more powerful than NAND-TM, so we need to show the other direction. There are two key ways in which NAND-RAM builds on NAND-TM: arbitrary indexing into arrays and integer valued variables. To show that we can emulate these using NAND-TM programs, we have to show that we can index into bit arrays and have two-dimensional bit arrays. These are a bit messy to achieve, but once we have them, we can have arrays of integers, and with syntatic sugar, that's enough for NAND-TM to emulate NAND-RAM. □

Theorem 2. *For every Turing machine M there is a one dimension cellular automaton that can simulate M on every input M.*

Proof idea. We begin by defining the idea of a **configuration** for a Turing machine, which contains all the information about the a Turing machine at a given step during execution. To do this, we

need to hold information about the symbols stored on the tape, as well as the position of the head and the state of the machine. For a Turing machine with tape alphabet Σ and state space $[k]$, a configuration is a string $\alpha \in \overline{\Sigma}^*$ where $\overline{\Sigma} = \Sigma \times (\cdot \cup [k])$ such that there is exactly one coordinate i for which $\alpha_i = (\sigma, s)$ for some $\sigma \in \Sigma$ and $s \in [k]$, and for all other coordinates j, $\alpha_j = (\sigma, \cdot)$ for some $\sigma \in \Sigma$. The idea is that in j^{th} tuple, the first element is the symbol on the tape at location j, the i^{th} tuple (the one without empty second element) indicates the head is at location i, and the second element of i^{th} tuple represents the state of the machine in the given configuration.

If $NEXT_M : \overline{\Sigma}^* \to \overline{\Sigma}^*$ is a function taking in a configuration of Turing machine M and returning the next configuration when we execute the next step of the Turing machine, we can show that the next value of tuple i depends only the values of tuples $i-1, i, i+1$. This reminds us of the transition rule in one-dimensional cellular automata, so we can show that this $NEXT_M$ function is something we can simulate with a cellular automaton with a properly chosen rule. □

Theorem 3. *For every function $f : \{0,1\} \to \{0,1\}^*$, f is computable in the enhanced λ-calculus if and only if it is computable by a Turing machine.*

Proof idea. We start with showing that a function f computable in λ-calculus is also computable with a Turing machine. NAND-RAM and Python are equivalent to a Turing Machine in power, so we can write a NAND-RAM or Python program that takes in an input string representing a λ-expression that computes f, applies the λ-calculus simplification rules to the expression string, and then outputs the simplified expression. Such simplification rules essentially involve repeated applications, which just involve "search-and-replace" style operations on the input expression string to rename bound variables inside abstractions. For example, the lambda expression represented by the string $(\lambda x.xx)(\lambda y.y)$ could be simplified to $(\lambda y.y)(\lambda y.y)$ by removing the string λx and replacing every instance of the string x with the string $(\lambda y.y)$, resulting in $(\lambda y.y)(\lambda y.y)$.

The other direction is more difficult. Much of the detail is omitted here, but essentially we can write a λ-calculus expression to compute the function $NEXT_M : \overline{\Sigma}^* \to \overline{\Sigma}^*$ that maps a Turing machine configuration M (encoded as a string $\alpha \in \overline{\Sigma}^*$) to the next configuration. To find the final configuration of M, we can write a function $FINAL(\alpha)$ that takes as input a configuration α representing M, and at each step, applies the $\lambda-$calculus expression for $NEXT_M$ on the output of $NEXT_M$ from the previous step, up until the point when a halting configuration is produced. Note that to actually compute $FINAL$ in λ-calculus, we can use the $RECURSE$ enhanced λ-calculus operation. Thus, we can write a $\lambda-$calculus expression to simulate any Turing machine, and thus any function computable by a Turing machine is also computable with a $\lambda-$calculus expression. □

5 Church-Turing Thesis

We previously defined a function as computable iff it can be computable by Turing machines, and we've seen how all the models we've discussed (RAM machines/NAND-TM, cellular automata, λ-calculus) are equivalent to Turing machines in power. This leads us to the famous Church-Turing thesis, which is that our notion of computability, as defined using Turing machines, is the only sensible definition of computable functions.