Lecture 15

Encodings and Universality

15.1 Encodings

The notion of encoding things into strings should be familiar to anyone who uses computers: after all, we all know that the words, images, sounds, and videos that a computer can store and display are all encoded in 0s and 1s in the computer's memory. Still, even though the notion that things can be encoded into binary strings is intuitive to an inhabitant of the modern world, we will go through a bit of mathematical formalism for it.

For a mathematical object M, we will use $\langle M \rangle$ to denote the encoding of M as a string. The exact details of the encoding won't matter too much; we go through some examples.

First, integers $n \in \mathbb{N}$ can be encoded as strings $\langle n \rangle$ in binary, i.e. $\langle n \rangle$ is a string over the alphabet $\{0, 1\}$. We can encode pairs of objects (M_1, M_2) by using a separator symbol #, and then setting $\langle (M_1, M_2) \rangle = \langle M_1 \rangle \# \langle M_2 \rangle$, where # is a symbol that does not occur in the encodings $\langle M_1 \rangle$ and $\langle M_2 \rangle$; then, if desired, we can switch alphabets to the Boolean alphabet $\{0, 1\}$ by encoding each alphabet symbol by some binary string (e.g. using ASCII notation, for example). This allows us to encode pairs of numbers (i, j) with $i, j \in \mathbb{N}$, and also any other finite tuple of numbers in a similar way.

We can also encode negative integers; to do so, we can just use a new symbol $-$ in front of the binary encoding of a number, and then we can again switch alphabets to $\{0, 1\}$ to encode everything in binary. We can encode rational numbers as a pair of integers (numerator and denominator). However, encoding real numbers cannot be done to infinite precision, since that would require infinitely many characters (this is not allowed, as strings must be finite). Therefore, we can only encode real numbers to some fixed precision, which effectively rounds them to some nearby rational number.

We can encode finite sets of mathematical objects by encoding each object in the set (the same way we would encode a tuple of the objects), and then an additional marker to remind ourselves these objects are supposed to be inside a set; for example, we could use the characters "{" and "}" and the beginning and end of the tuple to denote that the objects belong in a set (and then, as usual, we could switch to the alphabet $\{0, 1\}$ if we so desire).

We can even encode functions as strings, so long as the function has finite domain. This is because mathematically, a function is merely a set of ordered (input,output) pairs. That is, the function $f \colon \mathbb{N} \to \mathbb{N}$ defined by $f(x) = x + 1$ is actually the set $\{(0, 1), (1, 2), (3, 4), \dots\}$. This function has infinite domain, so we cannot encode it, but we can encode functions acting on finitely many elements because they correspond to finite sets of pairs.

Putting together all of these encodings, it becomes clear that we can encode machines such as

a DFA, NFA, PDA, or TM as strings. For example, if M is a DFA, then $M = (Q, \Sigma, \delta, q_0, F)$. The encoding of M as a string is denoted by $\langle M \rangle$, and we can implement it because we've already seen how to encode tuples, finite sets, and functions on finite domain (like δ is). Note that the names of the states in Q are arbitrary, so we can first rename the states in Q so that $Q = \{0, 1, 2, \ldots, n-1\}$ where $n = |Q|$ and so that $q_0 = 0$; this will allow us to use the encoding system for integers to encode the states in the finite sets Q and F. We can do a similar thing to the finite set Σ if we want to. The definitions of an NFA, PDA, and TM similarly lend themselves to such encodings. We can also encode context-free grammars and regular expressions as strings without too much difficulty.

In other words, as long as everyone agrees on some fixed encoding method, all the finite mathematical objects we considered in this course can be represented by strings over the alphabet $\{0, 1\}$ (and this is how your computer would represent them). Moreover, a Turing machine can convert any one such encoding into another (the same way a computer program can do so).

15.1.1 Unary encoding and lexicographic order

It turns out we can also encode things in unary, over only the alphabet $\{0\}$. To see how to do so, we only need to describe how to encode binary strings using the alphabet $\{0\}$ (since we already know how to encode things in binary). The trick is that we can order all the binary strings in *lexicographic order*; this order is first sorted by length, and then sorted alphabetically (like a dictionary). The first few binary strings in lexicographic order are

$$\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, \ldots$$

Note that every binary string s occurs exactly once in this ordering. The string ϵ occurs in position 0, the string 0 occurs in position 1, and so on. If we let $n(s)$ be the position of the string $s \in \{0, 1\}^*$ in the lexicographic ordering, then we can represent s as a string over the alphabet $\{0\}$ by writing $0^{n(s)}$. That is, the length of this string tells us the position of s in the lexicographic order, uniquely specifying the string s. This encoding using the alphabet $\{0\}$ is called *unary* encoding. It is much less efficient than binary encoding, but we are not worried about efficiency for now.

15.2 Some interesting languages

The concept of encodings allows us to describe some more interesting types of languages. So far, we have studied relatively boring languages such as $\{0^n 1^n 0^n : n \in \mathbb{N}\}$. We now introduce some more interesting types of languages. For example, consider the language

$$A_{DFA} = \{\langle \langle D \rangle, w \rangle : D \text{ is a DFA and } w \in L(D)\}.$$

This is the language consisting of (encodings of) all pairs (D, w), where D is a DFA and w is a string accepted by this DFA. Is A_{DFA} decidable?

We already saw that every regular language is decidable, so that for every DFA D there is a TM M that decides $L(D)$. However, this doesn't necessarily tell us anything about A_{DFA}. After all, to convert a single DFA D into a Turing machine M, we just need to describe a machine which takes a string w as input and decides whether D accepts w. Such a machine can depend on D, and effectively have the code of D "hardcoded" into it.

On the other hand, to decide A_{DFA}, we want a Turing machine M which takes as input both the description of a DFA D and the input w to that DFA, and decides whether D accepts w. This time, our Turing machine cannot depend on D, since D is part of the input; our Turing machine

must simulate any DFA it receives on the fly, without having that DFA be hardcoded into the states of the Turing machine.

Actually, the task of a Turing machine deciding A_{DFA} is slightly harder: not only does it need to determine whether a DFA D accepts an input string w, it also needs to handle the case that it receives a garbage input, which doesn't look like a description of a (DFA, input) pair at all. Note that garbage inputs are by definition not in A_{DFA}, so the Turing machine can safely reject them, but it still needs to handle them properly (and not accidentally accept them or loop on them).

Fortunately, this is not too difficult for a Turing machine to do. As will usually be the case, formally describing a Turing machine for A_{DFA} will be too tedious for us. Instead, we will make do with an informal argument. Recall that we argued Turing machines can do whatever computers can. Consider how you would program a computer program which decides the language A_{DFA}. We will talk in a similar way about Turing machines. First, the TM will check whether the input s is formatted like (D, w), where D is a valid DFA and w is a string in Σ^*, where Σ is the alphabet of the DFA. If the input string s is not formatted like this, the TM will reject, since $s \notin A_{DFA}$ in that case.

Next, the TM will simulate D on w. How? It will place a marker on the state of D that D is currently in, which starts at its start state q_0. The TM will also have a marker placed at the current position of the input string, starting right before the first symbol of w. Then the Turing machine will repeatedly "read" a symbol of w and take the corresponding transition in D, moving the markers each time (the state marker moves to the new state of D, and the position marker to the new position inside w, which is one symbol to the right). This is repeated until the TM is done processing the input string w, at which point the TM checks the final marked state of D. It will compare this state to the states in F one by one, and if it finds a match, it accepts. If no match is found, the TM will reject. It is not hard to see that this Turing machine decides A_{DFA}, so A_{DFA} is decidable.

We can construct other interesting languages in a similar way. For example, consider the language

$$A_{CFG} = \{\langle G, w\rangle\} : G \text{ is a CFG and } w \in L(G)\}.$$

Is A_{CFG} decidable?

As in the previous case, designing a TM which decides A_{CFG} is a potentially harder task than showing that every context-free language is decidable, since in the latter case you can build a TM that depends on the context-free language in question (for example, by hard-coding a CFG in Chomsky normal form into the Turing machine).

It turns out that A_{CFG} is decidable as well. We describe a Turing machine M that decides it. This machine M will start by checking if the input string s is formatted like $\langle\langle G\rangle, w\rangle$ for some context-free grammar G and some string w in the alphabet of G. If not, M will reject.

Next, M will convert G into Chomsky normal form. Here it is important to note that when we proved that every context-free grammar can be converted into an equivalent CFG in Chomsky normal form, our proof actually provided an algorithm for doing so; that is, we didn't just prove abstractly that such a Chomksky-normal-form CFG must exist, but we actually described how to build it algorithmically. The machine M will apply that algorithm to convert the input CFG G into an equivalent CFG G' in Chomsky normal form. It will then write down all strings of length $|w|$ that G' can generate (we saw how to do this when we proved that every context-free language is decidable). Finally, it will compare w to each of these listed strings, to see if it matches any of them. If so, M will accept; otherwise it will reject.

15.3 A universal Turing machine

What about the language

$$A_{TM} = \{\langle\langle M\rangle, w\rangle : M \text{ is a TM and } w \in L(M)\}?$$

This is the language consisting of all descriptions of (M, w) pairs where M is a Turing machine and w is an input accepted by M. Is this language decidable?

It turns out that A_{TM} is not decidable, intuitively because when given as input such (M, w), it is impossible to tell whether $M(w)$ runs forever using a finite number of steps. We will prove that A_{TM} is undecidable next class.

On the other hand, it turns out that A_{TM} is recognizable. In fact, we will prove something stronger: we will show there is a Turing machine U, called a *universal* Turing machine, which takes inputs of the form $\langle\langle M\rangle, w\rangle$, and simulates the behavior of M on w: that is, if M accepts w then U accepts $\langle\langle M\rangle, w\rangle$, if M rejects w then U rejects $\langle\langle M\rangle, w\rangle$, and if M loops on w then U loops on $\langle\langle M\rangle, w\rangle$. (The Turing machine U will also reject badly-formatted inputs which do not look like $\langle\langle M\rangle, w\rangle$ for any M or w.)

Such a Turing machine is called universal because it is one machine that can do anything: all you need to do is give it the description of some other machine, and it will mimic the behavior of that machine. The hardware on your computer is an example of a universal machine: modernly, we don't have separate physical machines, one for playing Starcraft and another for computing digits of π. We have one machine, a *universal* machine, which can do either of these tasks if the right software is provided – but software is, of course, just a string, a sequence of 0s and 1s. Universal Turing machines are similarly programmable: by giving a universal TM the right input, you can make it do anything any other Turing machine can do.

How shall we construct a universal Turing machine? This will be easiest to describe using multiple tapes (though remember that we can always convert a multi-tape Turing machine into a regular single-tape Turing machine). The machine U will have three tapes: one with the input $\langle\langle M\rangle, w\rangle$, one "work" tape, and one which will represent the tape of the machine M.

At the beginning, U will check that the input is correctly formatted; if not, it will reject. From now on, we can assume the input looks like $\langle\langle M\rangle, w\rangle$ for some Turing machine M and some string w over the alphabet of M. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$.

Next, U will copy over w onto its third tape, the one which represents the tape of M. It will place a special marker on the blank symbol right before the string w, representing the position of the head of M. U will also place a marker on the start state of M (in its first tape, where the input was). This marker will represent the fact that M is currently at its start state. U can do so by scanning Q to find a state name which equals q_0, and mark it (for example, using additional symbols).

From there, U will repeatedly implement transitions of M. More explicitly, recall that the function δ is a set of ordered pairs, with each pair being of the form $((q, a), (p, b, \leftarrow))$ or $((q, a), (p, b, \rightarrow))$. The machine U will scan all the ordered pairs of δ to find a pair $((q, a), (p, b, D))$ (with $D \in \{\leftarrow, \rightarrow\}$) in which q is the same as the marked state of M (the current state of M), and also where a is the same as the marked symbol on the third tape, the symbol read by the head of M. Then U will overwrite the symbol a in that position with b, move the marker left or right according to D, and then will scan across Q to look for a state whose name matches p, and move the marker to that state.

After M enters each state, U will also compare that state to q_{acc}; if they have the same name, U will accept. Similarly, U will compare the state to q_{rej}, and reject if they match. Otherwise, U will continue implementing the next transition of M.

It is hopefully clear that this machine achieves a faithful simulation of M, which it runs on the input w. It then follows that U accepts exactly when M accepts w, and rejects when M rejects w (or if the input is badly formatted), while looping if M loops on w.

15.4 A note about notation

To denote the encoding into a string, we've used notation such as $\langle\langle M\rangle, w\rangle$ above, where M is a Turing machine and w is a string. It is also OK to simply write $\langle M, w\rangle$ (or even to write other variations of this, like $\langle(M, w)\rangle$ for example). We will not be very strict about the exact notation, and we will such notation interchangeably. The key thing to remember is that both M and w are encoded as strings, and that there is a separator between them, so that the machine that reads this input knows when the description of M ends and the description of w begins.

15.5 The Church-Turing Thesis

As a final note for this lecture, we mention the Church-Turing thesis, which is the conclusion that all reasonable models of computation are equivalent to each other, and to Turing machines. Here by "reasonable" we mean something you might be able to build in real life, but also something which is sufficiently powerful. DFAs, NFAs, and PDAs are not sufficiently powerful; they are not capable of storing an manipulating memory in arbitrary ways. Machines that *are* capable of such manipulation always end up being equivalent to Turing machines, according to the Church-Turing thesis.

Note that it is called a "thesis" rather than a theorem. That's because it's not a formal mathematical statement; it is merely an observation that seems to hold in practice. It was first observed by researchers Church and Turing, who each independently defined a mathematical model of computation, and then discovered that their models were equivalent (despite appearing quite different on the surface). We've also seen (informally) that a Turing machine can simulate assembly code, and hence anything a computer can do, but it should also be clear that a modern computer (if given sufficient memory) would be able to simulate a Turing machine; hence the two models are equivalent.

Effectively, the Church-Turing thesis says that Turing machines are the "right" model of computation: something you can build in real life, but also something that can simulate every other machine you can build in real life. Of course, it is not a theorem; we don't yet know all the laws of physics, so it is at least conceivable that physics allows the construction of machines more powerful than Turing machines. However, this seems unlikely. Even quantum computers can be simulated on Turing machines (albeit extremely inefficiently), and no theories of physics appear to suggest more computational power is available in the universe.

Surprisingly, however, despite its power, not every language can be computed by a Turing machine. We've already claimed that A_{TM} is not decidable; next class, we will formally prove this claim (and we will also show some languages that are not recognizable).