1.5 Universal Turing Machines

Up to now, we have had to construct different specific Turing machines for each language we wished to recognize. In fact, we have had to construct specific machines for *every* language we wished to recognize in this course, whether that machine be a finite automaton, or a pushdown automaton, or indeed, a Turing machine.

However, we know that Turing machines are capable of performing quite complicated computations, and we know also that we can construct Turing machines that can simulate the computations of other variant Turing machines. What if we took this idea and generalized it as much as possible? That is, what if we constructed some Turing machine that could simulate the computation of *any* other Turing machine?

Alan Turing actually had this same idea in the paper that introduced the model of computation that would eventually be named after him. Turing described the process of constructing a machine $\mathcal{U}$ that is capable of simulating the computation of any other machine $\mathcal{M}$, as long as we give an appropriate encoding of $\mathcal{M}$ as part of the input to $\mathcal{U}$:

> "It is possible to invent a single machine which can be used to compute any computable sequence. If this machine $\mathcal{U}$ is supplied with a tape on the beginning of which is written the S.D. [standard description] of some computing machine $\mathcal{M}$, then $\mathcal{U}$ will compute the same sequence as $\mathcal{M}$."
> — Alan Turing, *On Computable Numbers, with an Application to the Entscheidungsproblem* Proceedings of the London Mathematical Society, Series 2, 42(1), 1937.

Here, like Turing did before us, we will show how to construct such a machine $\mathcal{U}$, which is nowadays called a *universal Turing machine*.[4] The main benefit of having such a machine is that we will no longer have to construct specific Turing machines for each language we consider; now, we can just give a high-level description of the Turing machine's computation, and we can feasibly "program" the universal Turing machine to perform its computation in a similar way. These high-level descriptions will be quite similar to what we saw in the proofs of Theorems 10, 12, and 13, where we simply listed the steps of the machine's computation instead of explicitly writing out each component of the machine.

Suppose that the input we give to our universal Turing machine $\mathcal{U}$ is of the form $\langle \mathcal{M}, w \rangle$, where $\mathcal{M}$ is an encoding of the Turing machine we wish to simulate and w is the input word given to $\mathcal{M}$. Given some input of this form, our machine $\mathcal{U}$ must satisfy three criteria:

1. $\mathcal{U}$ halts its computation on input $\langle \mathcal{M}, w \rangle$ if and only if $\mathcal{M}$ halts its computation on input w;
2. $\mathcal{U}$ enters its accepting state if and only if $\mathcal{M}$ enters its accepting state; and
3. $\mathcal{U}$ enters its rejecting state if and only if $\mathcal{M}$ enters its rejecting state.

We can now present the construction of this machine $\mathcal{U}$.

Theorem 14. *There exists a universal Turing machine $\mathcal{U}$ that, given an input $\langle \mathcal{M}, w \rangle$, is capable of simulating the computation of a Turing machine $\mathcal{M}$ on an input word w.*

Proof. We will construct $\mathcal{U}$ in the form of a multitape Turing machine, just as we did in our procedure to determinize a nondeterministic Turing machine. For this construction, we only need three tapes:

- The first tape will initially contain the input $\langle \mathcal{M}, w \rangle$, and after the computation of $\mathcal{U}$ begins, it will simulate the contents of the tape of $\mathcal{M}$.
- The second tape will contain the encoding of the machine $\mathcal{M}$.
- The third tape will keep track of which state we are currently in during the computation of $\mathcal{M}$ by maintaining the current state as a binary number.

[4] It's important to note that, in this context, "universal" does not mean that the Turing machine $\mathcal{U}$ can compute *everything*. It only means that $\mathcal{U}$ can compute whatever other Turing machines can compute.

At the beginning of its computation, $\mathcal{U}$ will contain only the input $\langle \mathcal{M}, w \rangle$ on its first tape, and its other two tapes will be blank.

$\cdots$	$\sqcup$	$\langle$	...	$\mathcal{M}$	...	,	...	w	...	$\rangle$	$\sqcup$	$\cdots$
$\cdots$	$\sqcup$	$\sqcup$	...	$\sqcup$	...	$\sqcup$	...	$\sqcup$	...	$\sqcup$	$\sqcup$	$\cdots$
$\cdots$	$\sqcup$	$\sqcup$	...	$\sqcup$	...	$\sqcup$	...	$\sqcup$	...	$\sqcup$	$\sqcup$	$\cdots$

Now, we can describe how $\mathcal{U}$ simulates the computation of $\mathcal{M}$ on w:

1. Initialize the tapes in the following way:
   (a) Transfer the encoding of $\mathcal{M}$ from the first tape to the second tape by writing $\langle \mathcal{M} \rangle$ to the second tape and erasing it from the first tape.
   (b) Read the encoding of $\mathcal{M}$ on the second tape to determine the number of states in $\mathcal{M}$. If $\mathcal{M}$ contains k states, then write $\lceil \log_2(k) \rceil$ copies of `0` to the third tape.

   After initialization, the tape will look like the following:

$\cdots$	$\sqcup$	$\sqcup$	...	$\sqcup$	...	$\langle$	...	w	...	$\rangle$	$\sqcup$	$\cdots$
$\cdots$	$\sqcup$	$\langle$	...	$\mathcal{M}$	...	$\rangle$	...	$\sqcup$	...	$\sqcup$	$\sqcup$	$\cdots$
$\cdots$	$\sqcup$	0	...	0	...	0	...	$\sqcup$	...	$\sqcup$	$\sqcup$	$\cdots$

2. Move the input head of the first tape to the first symbol of w. Move the input head of the second tape to the first symbol of $\langle \mathcal{M} \rangle$. Move the input head of the third tape to the first symbol of the sequence of `0`s.
3. Repeat the following steps until $\mathcal{M}$ halts:
   (a) For each computation step of $\mathcal{M}$, scan the second tape to find a transition that matches the current input symbol and state written on the first and third tapes, respectively.
   (b) Modify the contents of the first and third tapes to reflect the transition that was just taken.
   (c) If no transition exists for the current state/symbol pair, then halt and go to step 4.
4. Transition to q_{accept} if $\mathcal{M}$ transitioned to its accepting state. Transition to q_{reject} if $\mathcal{M}$ transitioned to its rejecting state. □

2 The Church–Turing Thesis

In the early days of computer science, long before physical computers as we know them existed, mathematicians and logicians wanted to know whether it was possible to use mechanical procedures to solve mathematical problems. Perhaps the most well-known of these problems at the time was the *Entscheidungsproblem*, which is German for "decision problem". The Entscheidungsproblem asks whether there exists a general procedure to decide whether a given statement is valid and provable using a predetermined system of logic. The Entscheidungsproblem was formalized by the German mathematician David Hilbert in 1928, although the seeds of the idea go all the way back to 1900, when Hilbert presented his famous set of problems at the International Congress of Mathematicians.

The problem back then was that there was no universally-agreed upon definition of a "procedure" that could decide the problem. The most appropriate definition was eventually taken to be that of an *effective method*. If we're given a class of problems, then a method for that class of problems is called effective if

1. the method consists of a finite number of exact instructions; and
2. the method always terminates and produces a correct answer when it is applied to a problem from its class.

In principle, an effective method is one that a human can perform on paper in a purely mechanical manner; it requires no creative thought or insight to arrive at an answer. It is computation in its purest form. If we view the class of problems in a way that allows us to sort instances of the problem into "yes" and "no" outcomes, then we essentially have a function that takes an input and returns either "yes" or "no". Then, any function with an associated effective method to solve it is called an *effectively calculable* function.

Throughout the 1930s, then, mathematicians and logicians focused on developing effective methods for solving the Entscheidungsproblem. In 1931, the Austrian-born logician Kurt Gödel published his famous paper introducing his incompleteness theorems. In this paper, Gödel introduced the notion of *recursive functions*, which was his approach to defining effective calculability.

The first major breakthrough with the Entscheidungsproblem came in a series of papers by the American mathematician Alonzo Church, wherein he introduced the notion of the *lambda calculus*. The lambda calculus is a system of mathematical logic that allows us to express computations in terms of function applications. Church proposed that the class of effectively calculable functions should correspond to the class of functions that can be defined in the lambda calculus. Indeed, along with Stephen Kleene and J. Barkley Rosser, Church showed that the class of lambda-definable functions corresponded exactly to the class of recursive functions. In 1936, Church then proved that the Entscheidungsproblem was unsolvable.

The next breakthrough came, again, in 1936 with a presentation by Alan Turing to the London Mathematical Society. Like Church, Turing showed that the Entscheidungsproblem was unsolvable, but he used a different formalization: a machine model, which would later come to be called a *Turing machine*. Turing was aware of Church's work, and he added as an appendix to his paper a proof sketch showing that his machine formalization was equivalent to Church's lambda calculus. Turing would go on to complete his PhD under the supervision of Church just a couple of years later.

Despite this flurry of results throughout the 1930s, it would take until 1952 for someone to formally define the notion of effective calculability. Stephen Kleene, in his book *Introduction to Metamathematics*, introduces what he calls *Church's thesis*...

> "Every effectively calculable function (effectively decidable predicate) is general recursive."

...and what he calls *Turing's thesis*...

> "Turing's thesis that every function which would naturally be regarded as computable under his definition, i.e. by one of his machines, is equivalent to Church's thesis [...]"

...which, taken together, give us the *Church–Turing thesis*: a connection between effectively calculable procedures and Turing machines. Note that, since this result is more definitional rather than a statement that we can formally prove, we refer to it as a "thesis" instead of a "theorem".

In modern language, we can state the thesis as follows.

Church–Turing thesis. Any function that can be computed by an algorithm can also be computed on a Turing machine.

In recent times, the Church–Turing thesis has allowed researchers to prove that all sorts of formal models are capable of behaving like a real-world computer running an algorithm. If some model of computation or some system of rules can be used in a way that allows it to simulate the computation of any Turing machine, then we say it is *Turing-complete*. We've already seen one example of a Turing-complete model—the universal Turing machine—but there exist many more (and much weirder) examples:

- Most general-purpose programming languages, and some specialized languages (like LaTeX, the typesetting system used to create these lecture notes!)
- Conway's Game of Life and, more generally, cellular automata
- Enzyme-based DNA computers and chemical reaction networks
- Microsoft Excel and Microsoft PowerPoint
- Minesweeper, Dwarf Fortress, Minecraft, and Magic: The Gathering