ECS 120 Lesson 17 – Church's Thesis, The Universal Turing Machine

Oliver Kreylos

Monday, May 7th, 2001

In the last lecture, we looked at the computation of Turing Machines, and also at some variants of Turing Machines – nondeterministic Turing Machines and enumerating Turing Machines. From now on, we will shift our attention away from machines and languages, and will start reasoning about algorithms and problems instead. Before we can do so, we have to understand how different models of computation – one of them being Turing Machines – are related to each other. This will allow us to reason about algorithms without necessarily having to use the low-level models that we have seen so far.

1 The Church-Turing Thesis

Parallel to the development of the Turing Machine by Alan Turing, the mathematician Alonzo Church invented a different model of computation, the λ-*calculus*. As opposed to Turing Machines, this model is not "machine-based," but relies on the application of mathematical functions to their arguments. λ-calculus inspired development of the LISP programming language by John McCarthy in the late 1950's. LISP, in fact, is an almost direct implementation of Alonzo Church's ideas. Though these two models are very different in their structure, Church and Turing were later able to prove that their models are equivalent – any computation that can be described by one can also be described by the other. Later, other models of computation were invented (random access machine, register machine, higher-level programming languages,...); all of these could be proven to be equivalent to each other.

This somewhat surprising observation led computer scientists to believe that all these models are in fact describing the same thing – algorithms, in the intuitive meaning that they are "recipes for computation." Recall: An algorithm is a strategy for solving a problem, which satisfies the following three properties:

1. An algorithm is a *clearly* and *unambiguously defined*.
2. An algorithm is *effective*, in the sense that each of its steps is *executable*.
3. An algorithm is *finite*, in the sense that the textual description itself is of finite length, and that it always terminates after a finite number of steps.

The equivalence of all known models of computation led Alonzo Church to formulate the following hypothesis:

Hypothesis 1 (Church–Turing Thesis) *Every conceivable model of computation is equivalent to the intuitive notion of computation, defined by algorithms.*

The reason for this hypothesis not being called a theorem is that it can not be proven true. Since the "intuitive notion of algorithm" does not have an exact mathematical definition, one cannot reason about it with mathematical means. Though it is not a theorem, the Church–Turing thesis is very valuable for reasoning about algorithms. It has an analogous impact to the theorem that for every regular language there must be a finite state machine that accepts it. This fact, for example, led us to discovery of the Pumping Lemma for regular languages, the most important tool in proving that certain languages are not regular. In the context of algorithms, the Church–Turing thesis allows us to write down an algorithm for a specific problem, and then to conclude that there must be a Turing Machine that computes this algorithm. From that point on, we can apply the theory of Turing Machines to prove certain properties of this hypothetical machine, without ever having to actually build or define it. Since Turing Machines performing non-trivial tasks are typically extremely complicated, this "higher-level reasoning" based on algorithms and the Church-Turing thesis is what makes theory of computation practically useful.

2 The "Hello World"-Problem

As an example, let us look at a typical problem in real-life programming: The question whether a certain program performs according to its specification. To pick a simple example that every student of the `C` programming language knows: Write a program that prints the words "Hello World" and then terminates. While it is pretty easy to write a program that conforms with this specification, see Figure 1, it is very difficult to determine whether a given program does. Figure 2 shows a non-trivial program that might or might not print "Hello World" and then exit.

```
#include <stdio.h>

void main(void)
    {
    printf("Hello World\n");
    }
```

Figure 1: A simple program that prints "Hello World" and then exits.

The program in Figure 2 works in the following way: It expects an integer n as its only command-line parameter, and then performs a search for any triple $x, y, z \in \mathbf{N}$ of positive integers, such that $x^n + y^n = z^n$. If such a triple is found, the program prints "Hello World" and exits. In other words, for any input parameter $n \geq 3$, this program only conforms to its specification if Fermat's last theorem is false. This statement, first made by Fermat about 330 years ago, was proven correct just recently, on November 19th, 1994, by Andrew Wiles at Princeton.

Now it would be very nice if there was an algorithm that could automatically deduct that a certain program conforms to its specification, as this would make extensive testing obsolete, and would probably make non-working programs disappear. Alas, as it turns out, there is no such algorithm. The example program in Figure 2 may be a rather contrived example, but it is one of the possible cases an automatic correctness prover would have to deal with. We now know that the program in Figure 2 does not work for any $n \geq 3$, but it took the world's best mathematicians more than 300 years to prove this. Still believing that there is an algorithm to do things like this automatically is overly optimistic. This is only a result of intuition about

```
#include <stdio.h>
#include <stdlib.h>

int ipow(int basis, int exponent)
    {
    int result = 1;
    while(exponent > 0)
        {
        result *= basis;
        --exponent;
        }

    return result;
    }

void main(int argc, char* argv[])
    {
    int n, sum, x, y, z;

    n = atoi(argv[1]);

    for(sum = 3; true; ++sum)
        for(x = 1; x <= sum - 2; ++x)
            for(y = 1; y <= sum - x - 1; ++y)
                {
                z = sum - x - y;
                if(ipow(x, n) + ipow(y, n) == ipow(z, n))
                    {
                    printf("Hello World\n");
                    return;
                    }
                }
    }
```

Figure 2: A program that might or might not print “Hello World” and then exit.

algorithms, but Church's hypothesis will allow us to prove that there can be no algorithm to prove the correctness of a program, based on the theory of Turing Machines. The basic idea of that proof is that every algorithm can be computed by a Turing Machine, and the assumption that there is a correctness-proving Turing Machine is then shown to violate basic properties of Turing Machines. Without the Church–Turing Thesis, we could not have proved this fundamental result, because there is no formal mathematical definition of "algorithm."

3 Encoding Turing Machines

The idea of the proof alluded to above requires that a Turing Machine can somehow read the description of an algorithm, and then reason about that algorithm. Since one model of algorithm is again a Turing Machine, this is equivalent to a Turing Machine that can read the description of another Turing Machine and work on it.

By the definition of Turing Machines, the input to a machine can only be a word, i. e., a string of characters. To be able to feed a Turing Machine as input into another one, we have to find a way to encode a Turing Machine as a string. Luckily, this is not too complicated. To recall, a Turing Machine M is defined as a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ and Γ are finite sets, and δ is a function from a finite set (the cartesian product $Q \times \Gamma$) to another finite set (the cartesian product $Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$). The sets Q, Σ and Γ can be converted into finite strings by listing their elements: $Q = \{q_0, q_1, \ldots, q_n\}$, $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$, $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_o\}$. For example, the string for Q starts with an opening brace, then some name for the first state, a comma, some name for the second state, another comma, etc. The string ends with a name for the last state, and then a closing brace.

To write down the transition function δ, we observe that any function can be written as a set of argument/value pairs: If $f\colon X \to Y$ is some function, then $f = \left\{ x \mapsto y \;\middle|\; x \in X, y = f(x) \in Y \right\}$. If both the domain set X and the range set Y are finite, so is the set describing the function. For example, the transition function for a hypothetical Turing Machine could be written as $\delta = \big\{(q_0, \mathtt{a}) \mapsto (q_1, \mathtt{b}, \mathrm{R}), (q_0, \mathtt{b}) \mapsto (q_0, \mathtt{a}, \mathrm{R}), \ldots, (q_1, \mathtt{a}) \mapsto (q_1, \mathtt{b}, \mathrm{L})\big\}$.

Now let q, s, g and d be strings defining the sets Q, Σ, Γ and δ for some Turing Machine M, and let q_0, q_{accept} and q_{reject} be the names used for those states in q, then we can describe the complete machine by the

string $(q, s, g, d, q_0, q_{\text{accept}}, q_{\text{reject}})$.

One could think about encodings like this in a less abstract way, by recalling that any file (in the computer sense) is a finite string over the alphabet containing the 256 different values a byte can have, and that an encoding for a class of objects is a file format for that class. We just defined a file format for Turing Machines; in fact, there is a file format for every kind of object that can be represented in a computer. Generally, if O is an object of some type (a graph, a finite state machine, a Turing Machine, etc.) we denote the encoding of O as $\langle O \rangle$.

4 The Universal Turing Machine

Using encodings as described in the last section, we can build a Turing Machine U that can read the encoding $\langle M \rangle$ of any other Turing Machine M and a word w over M's input alphabet as input, and can then *simulate* the computation of M when reading input word w. If after simulation, machine U accepts exactly if M would accept input w, and rejects exactly if M would reject input w, then the result of running machine M directly on input w, or running machine U on input $\langle M \rangle, w$ cannot be distinguished. In other words, machine U is an *interpreter* for Turing Machines, that can behave exactly like any other Turing Machine. Due to this property, machine U is called *Universal Turing Machine* (UTM). A Universal Turing Machine is a mathematical model for a common general-purpose computer – a machine that can read a program (from disk) and then execute that program on some input.

To see how a Universal Turing Machine U can simulate another machine, we have to go back to the definition of computation: The current configuration of machine M is determined by its ID (u, q, v), where u is the left tape string, q is the current state, and v is the right tape string. Since the left and right tape strings are always of finite length, an ID itself can be encoded as a string. To simulate execution of machine M, all machine U has to do is to keep track of M's ID (by writing it somewhere onto its work tape), and to update the ID for every step of computation by consulting the encoding of M's transition function δ. If, at some point during computation, M reaches a halting configuration, U will also move to a halting configuration and accept if M accepted and reject otherwise. Before it starts simulating, machine U should also check if the given input is really a valid description

for a Turing Machine,i. e., if the string $\langle M \rangle$ conforms to the specification we gave above, and if w is really a word over M's input alphabet. Altogether, the algorithm for U works as follows:

1. Check the input word $\langle M \rangle, w$ for correctness, i. e., check whether $\langle M \rangle$ describes a valid Turing Machine, and whether w is a string over the input alphabet of M.
2. Write the starting configuration (ϵ, q_0, w) of M onto the work tape.
3. If the current configuration for M is a halting configuration, go to step 7.
4. Find an argument pair matching the current state and input character of M in the encoding of M's transition function.
5. Update the current state and left and right tape string according to the value triple found in M's transition function.
6. Repeat from step 3.
7. If M's current state is the accept state, accept; if the current state is the reject state, reject.

5 The Halting Problem

The first problem for which we will prove that it cannot be solved by an algorithm is the *halting problem*. It is very similar to the "Hello World"-problem discussed above: It asks if a given Turing Machine M ever reaches a halting configuration when fed the input word w. Since Turing Machines are equivalent to other models of computation, the halting problem could as well have been stated as asking whether a given `C` program terminates when reading the input word w. In the "Hello World" example we have seen that the first version of the program always halts (it does not even read its input), but that the second version halts exactly if its input word is the ASCII representation of an integer $n \leq 2$.

To reason about the existence of an algorithm solving the halting problem, we use the Church–Turing Thesis and conclude that an algorithm exists exactly if there exists a Turing Machine that solves it. Such a Turing Machine

would work very similarly to the Universal Turing Machine U we defined above: It would read an encoding of a machine M and an input word w over M's alphabet as input, and accept exactly if M ever reaches a halting configuration. We can generate such a machine H' easily, by changing the algorithm given for U: In step 7, instead of accepting/rejecting according to M, the machine H' would always accept.

This machine, however, does not completely solve the halting problem: If a machine M halts on word w, machine H' can detect this and accept. If, on the other hand, M does not terminate on w, then H' can never reach the rejecting stage. Since H' does not have knowledge about the inner workings of M, it can never be certain that M will not reach a halting configuration at some point of time in the future.

Thus, the situation at hand is very similar to the situation with a Turing-recognizable language $L(M)$: If a word w is an element of $L(M)$, then the Turing Machine M must accept; if, on the other hand, w is not an element of $L(M)$, M can either reject or loop. This seeming similarity between problems and languages can be exploited to apply language theory to problems as well: A problem P with a yes/no answer can be converted to a language by defining $L(P) := \{ w \mid \text{The answer of problem } P \text{ to } w \text{ is yes} \}$. This way, the language of the halting problem would be $L(H) = \{ \langle M \rangle, w \mid M \text{ is a Turing Machine that halts on input } w \}$. Usually, we will identify problems and their languages, and just call the language $L(H)$ the halting problem.

Using language terminology, we know that Turing Machine H' accepts the halting problem, but it does not decide it. In other words, the halting problem is a recursively enumerable language. In the next lecture, we will prove that the halting problem is not a decidable language.