Church's thesis and the universal Turing machine

Vassos Hadzilacos

Church's thesis

Church's thesis, also often referred to as the Church-Turing thesis, is an assertion that identifies the concept of what it means for a procedure to be "algorithmic" or "effectively computable" with the concept of being computable by a Turing machine. It can be stated as follows.

- **Church's Thesis:** A computational procedure is an algorithm if and only if it can be carried out by a Turing machine.

Although this has the form of a mathematical definition it is not: it relates an informally understood concept (algorithmic procedure) to a precise, mathematical one (computable by Turing machine). It can be thought of as a hypothesis or a point of view. As we will see, there is strong evidence in support of this hypothesis. Its utility lies in that it allows us to bypass the rather painful task of describing computational tasks in terms of the very low-level operation of a Turing machine in favour of using familiar high-level programming constructs and arithmetic operations that we intuitively know can be carried out algorithmically, such as computing (the representation of) $\lfloor\sqrt{n}\rfloor$ given (the representation of) a natural number n. There is both empirical and mathematical evidence supporting Church's thesis.

The ***empirical evidence*** consists in that anything that a modern electronic computer can do given enough time and memory (presumably therefore everything that is "effectively computable") can be done by a Turing machine — and (less surprisingly) vice versa. To establish this fact one can define what a modern electronic computer does in terms of a typical CPU's instruction set (arithmetic and Boolean operations, comparison operations, branching instructions, etc.) and then show how to simulate each of these instructions in a Turing machine. This is not difficult, but it is tedious and so we will not discuss it further.

The ***mathematical evidence*** in support of Church's thesis derives from three sources. First, the various extensions to Turing machines that have been proposed — making the tape infinite in both directions, adding more tapes, allowing the tape to be a two-dimensional array of cells, and allowing non-determinism, among others — turn out not to change the computational power of the basic model: The same class of functions is computed by all these variations, so it seems that we have hit an inherent limit in what Turing machines can do.

A second mathematical evidence for the soundness of Church's thesis is that the many independently proposed and diverse approaches to defining "effective computation" turned out to be equivalent: they all define the same class of functions even as they do so in very different ways. In 1933, Kurt Gödel and Jacques Herbrandt formalized this notion as computation by so-called partial recursive functions. This is the set of functions built inductively as follows: For the basis we start with three kinds of functions: the ***constant functions*** ($F_n(x_1, x_2, \ldots, x_k) = n$, for every $n \in \mathbb{N}$), the ***projection functions*** that select one of their inputs ($P_i^k(x_1, x_2, \ldots, x_k) = x_i$ for every $k \in \mathbb{N}$ and i, $1 \leq i \leq k$), and the ***successor function*** ($S(x) = x + 1$). For the induction step, we build the set further by applying three operations to functions already in the set: ***composition*** of functions already in the set, ***recursion*** (which defines a function by giving its value $f(0)$ at 0 explicitly, and its value $f(n)$ for any natural number $n > 0$ as the value of another function, already in the set, at $f(n-1)$), and ***minimization*** (defining a function at some natural number as the smallest natural number for which another function, previously in the set, is zero). In 1936, Alonso Church and Stephen Kleene defined the notion of effective computation in terms of the so-called λ-calculus, a precise notation for defining functions, which later became the foundation for functional programming languages such as LISP and Haskell. Also in 1936, Alan Turing proposed his α-machines, now known as Turing machines, as the definition for what it means to compute a function effectively. In 1951, Andrei Markov proposed what became known as "Markov algorithms", a formalism for generating strings — in spirit similar to, but much more general than, the context-free grammars that you learned about in CSCB36. All these formalisms are equivalent, providing more evidence that again we have reached a limit, that being what can be computed "effectively".

The third mathematical evidence in support of Church's thesis is the concept of ***universality***, developed by Turing in the 1936 paper in which he introduced his machines. Universality shows that Turing machines (and all the other equivalent formalisms) are powerful enough to describe themselves. This ability for self-reference, which is not shared by weaker models of computation such as finite state automata and pushdown automata, also points to a certain completeness in the notion of effective computability. Universality turns out to be extremely important for its mathematical implications, which we will be exploring in the first part of this course, but also for its practical implications (see the box on page 5).

Universal Turing machine

There is a Turing machine M_U, called a ***universal Turing machine***, that takes as input the "encoding" of any Turing machine M and any input x to M, and simulates the steps of M on x. Note that this is a single Turing machine that simulates all Turing machines, including, of course, itself — the self-reference, or introspection, that we mentioned earlier. Also note that M_U, being a specific Turing machine, must have a fixed and finite set of states and a fixed and finite tape alphabet — even though it is to simulate any Turing machine, including those that have more states and more tape symbols than it does! This may seem impossible until we remember that in computers everything (numbers, programs, pointers, data structures, text, images, audio recordings) is represented by strings over the alphabet consisting of just two symbols: 0 and 1.

The first task in discussing the universal Turing machine M_U is to show how to encode an arbitrary Turing machine using a finite and fixed alphabet. There are many ways to do this. We will sketch here a simple one that uses the alphabet $\Sigma_U = \{0, 1, \#\}$. We will use binary strings to encode various elements of a Turing machine, and # as a separator for these elements.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, h_A, h_R)$ be a Turing machine.

- Encode each state $q \in Q$ by a unique binary string of length $\lceil \log_2 |Q| \rceil$.[1] We denote by $\langle q \rangle$ the binary string that encodes state q.

- Encode the set Q by listing the codes of the states, separated by #, with the convention that the first three codes listed are those of the initial state q_0, the accept state h_A, and the reject state h_R. So, if Q consists of these three states and the states $q_1, q_2, \ldots, q_k$, the code of Q is the string $\langle q_0 \rangle \# \langle h_A \rangle \# \langle h_R \rangle \# \langle q_1 \rangle \# \ldots \# \langle q_k \rangle$; we denote this as $\langle Q \rangle$.

- Encode each tape symbol $a \in \Gamma$ by a unique binary string, denoted $\langle a \rangle$, of length $\lceil \log_2 |\Gamma| \rceil$. Note that this also encodes the symbols of the input alphabet $\Sigma \subset \Gamma$.

- Encode the input alphabet $\Sigma = \{a_1, a_2, \ldots, a_m\}$ by the string $\langle a_1 \rangle \# \langle a_1 \rangle \# \ldots \# \langle a_m \rangle$; we denote this as $\langle \Sigma \rangle$.

- Encode the tape alphabet Γ in a similar manner, with the convention that the blank symbol is listed first. The code of Γ is denoted $\langle \Gamma \rangle$.

[1] So that we have enough bits to assign distinct codes to the states.

- Encode any string $x \in \Gamma^*$ by concatenating the codes of the symbols that appear in x in the order in which they appear there. So, if $x = a_1 a_2 \ldots a_\ell$, the code of x is the string $\langle a_1 \rangle \langle a_2 \rangle \ldots \langle a_\ell \rangle$. Note that there is no need for separator symbols here, since we use a fixed-length encoding for the symbols in the alphabet, so we (and a Turing machine!) can figure out the boundaries of individual symbol codes. The code of x is denoted $\langle x \rangle$.

- Encode an individual state transition $\delta(q, a) = (p, b, D)$ by the string $\langle q \rangle \langle a \rangle \langle p \rangle \langle b \rangle \langle D \rangle$, where $\langle D \rangle$ is the string 0 (respectively 1) if $D = L$ (respectively $D = R$).[2]

- Encode the entire transition function δ by concatenating the codes of the individual state transitions; the code of δ is denoted $\langle \delta \rangle$.

- Encode the entire Turing machine M as $\langle Q \rangle \#\# \langle \Sigma \rangle \#\# \langle \Gamma \rangle \#\# \langle \delta \rangle$. As noted earlier, the initial state q_0 and the two halting states h_A and h_R are identified by their position in $\langle Q \rangle$, and the blank symbol of M by its position in $\langle \Gamma \rangle$. The encoding of M is denoted $\langle M \rangle$.

In this manner, any Turing machine, with any number of states and an arbitrarily large tape alphabet, can be encoded by a string in Σ_U^*. Of course, many strings in Σ_U^* do not encode Turing machines according to the rules described above. It is convenient to make the mapping from Σ_U^* (i.e., strings encoding Turing machines) to the Turing machines that these strings encode a total function (defined for all such strings), so we will adopt the convention that syntactically unacceptable strings encode some fixed Turing machine, say one that rejects all inputs without taking any steps, i.e., $q_0 = h_R$. Similarly is convenient that all strings over Σ_U^* encode strings over the input alphbet Σ_M or the tape alphabet Γ_M of Turing machine M. To that end, we will adopt the convention that strings in Σ_U^* that do not properly encode a string in Σ_M^* or Γ_M^* according to the encoding rules actually encode some fixed string, say the empty string ϵ.

We could have encoded Turing machines using different conventions. The important feature of any such encoding is that, given a string that encodes a Turing machine M, it is possible (even for for a Turing machine!) to "decode" the string and extract from it the codes of the various components of M: the codes of its states, its tape alphabet, and the various state transitions. It should be fairly clear that this is the case for the encoding described above.

You may wonder whether it is possible to encode Turing machines using only two symbols, rather than three. This is indeed the case. Note that the basic idea of our encoding is to associate natural numbers with the various elements of the machine: state number 0, number 1, and so on; symbol number 0, number 1, and so on; transition number 0, number 1, etc. We can encode natural numbers in unary using the string 1^{n+1} to denote the number n, and use the symbol 0 as a separator. This makes the encodings longer, since unary encoding of numbers is very inefficient. If we are not worried about efficiency, however, (and for the time being we are only concerned with computability, not efficiency — that's the subject of complexity theory, the second part of the course) this is a perfectly valid way to encode Turing machines.

The universal Turing machine takes as input the encoding $\langle M \rangle$ of a Turing machine M and the encoding $\langle x \rangle$ of an input string x of M. This pair can be encoded in our alphabet $\{0, 1, \#\}$ by the concatenation of the two encodings separated by #, i.e., by the string $\langle M \rangle \# \langle x \rangle$. We will denote this as $\langle M, x \rangle$.

Notational convention about encodings. We will have occasion to represent many different kinds of finite mathematical objects as strings over some alphabet, just as we did for Turing machines. For example, we will need to represent natural numbers, sets, sequences, and graphs. The details of such encodings will not be important for our purposes, and we will take them for granted. We will assume, however, that these encodings are computable in the sense that, given the encoding of an object, a Turing machine can determine and manipulate the encodings of that object's constituent parts. For example,

[2] We could separate the various components using #s; do you see why this is not necessary?

given the encoding of a graph, a Turing machine can determine the (encoding of the) graph's nodes and edges — and therefore can perform operations such as determining if there is a path between two nodes. If O is a (finite) mathematical object, we will denote by $\langle O\rangle$ its encoding by a string in some suitable alphabet; and we will denote by $\langle O_1, O_2, \ldots, O_k\rangle$ the encoding of the ordered collection of mathematical objects $O_1, O_2, \ldots, O_k$. For example, if we are discussing the problem of determining whether there is a path of length ℓ connecting two nodes u and v of a graph G, we will use notation such as $\langle G, u, v, \ell\rangle$ to describe the string that encodes the collection of these objects of interest.

We are now ready to state and prove the following important theorem.

Theorem 3.2 (Turing, 1936) *There is a* ***universal Turing machine*** M_U *that, on input* $\langle M, x\rangle$*, simulates the operation of Turing machine* M *on input* x*; that is,*

- M_U *accepts* $\langle M, x\rangle$ *if* M *accepts* x*;*
- M_U *rejects* $\langle M, x\rangle$ *if* M *rejects* x*; and*
- M_U *loops on* $\langle M, x\rangle$ *if* M *loops on* x*.*

PROOF SKETCH. We will describe M_U as a three-tape Turing machine; as we have seen this can be converted to a single-tape Turing machine (Theorem 2.1). The input alphabet of M_U is $\{0, 1, \#\}$, i.e., the alphabet used to encode Turing machines and pairs consisting of a Turing machine and its input. The tape alphabet of M_U is its input alphabet together with $\sqcup$, the blank symbol of M_U. The tapes of M_U play the following roles:

- Tape 1 holds the input $\langle M, x\rangle$; it is read-only.
- Tape 2 holds the contents of M's tape during the simulation. More precisely, it holds the concatenation of the encodings of the symbols in M's tape. Thus, the cells of M_U's tape contain symbols in M_U's alphabet, not those of M's alphabet (which could be much larger).
- Tape 3 holds (the encoding of) the current state of M.

Initially, the input $\langle M, x\rangle$ of M_U is on tape 1; the other two tapes are empty, and the head of each tape is on the leftmost cell. M_U works as follows:

1. Copy $\langle x\rangle$ to tape 2, and $\langle q_0\rangle$ to tape 3 ($\langle q_0\rangle$ is found in $\langle M\rangle$, based on the conventions described earlier for encoding Turing machines). Move the head of each of these tapes back to its leftmost cell.
2. If tape 3 contains $\langle h_A\rangle$, then accept; if it contains $\langle h_R\rangle$, then reject.
3. Let $\langle q\rangle$ be the encoding of M's state in tape 3; let $\langle a\rangle$ be the encoding of M's tape symbol that starts where the head of tape 2 is positioned, and return the head to the start of the encoding of that symbol.
4. Search tape 1 for the encoding of the state transition that starts with $\langle q\rangle\langle a\rangle$. Let the encoding of that transition be $\langle q\rangle\langle a\rangle\langle p\rangle\langle b\rangle\langle D\rangle$.
5. Simulate the execution of that transition; that is,
   - replace $\langle q\rangle$ by $\langle p\rangle$ on tape 3, and move that tape's head back to its leftmost cell;
   - replace $\langle a\rangle$ by $\langle b\rangle$ in the cells that start at the current position of tape 2's head;
   - if $\langle D\rangle$ is 0 (respectively, 1) then move the head of tape 2 to the beginning of the encoding of the previous (respectively, next) symbol of M's simulated tape. (M_U can determine this based on the length of the encoding of M's tape symbols, which can be found in $\langle M\rangle$.)
6. Go to Stage 2.

Since M_U simulates every step of M by a sequence of its own steps as described above, on input $\langle M, x\rangle$ M_U carries out the simulation of M's computation on input x. Therefore M_U accepts, rejects, or loops on its input $\langle M, x\rangle$ according as M accepts, rejects, or loops on x. □

Those of you who have taken CSCB58 or otherwise know the basics of computer architecture will recognize in this proof the way computer hardware works: The program (represented here by the code

$\langle M\rangle$ of the simulated Turing machine) and its input (represented here by $\langle x\rangle$) are stored on the computer's memory (tape 1). The computer fetches the next instruction of the program to execute (M_U searches tape 1 to find the appropriate state transition), then it executes that instruction (M_U simulates the step of M), and the cycle continues until the computer executes the "stop" instruction (M_U halts after accepting or rejecting because it simulated a step of M that caused it to accept or reject its input x).

In view of this, the proof of the preceding theorem may leave you cold: There is nothing new here, it is just how computers work. In fact, it should leave you very excited because the causality is exactly reversed: Computers work like this because of Turing's universal machine!

Universal Turing machines and the stored-program computer. The story goes like this: In 1935, the year after completing his undergraduate studies, Turing attended a series of lectures on the foundations of mathematics given by his Cambridge professor, the topologist Max Neuman. In these lectures Turing learned of the "Entscheidungsproblem", Hilbert and Ackermann's question whether there is an algorithmic procedure to determine if any given formula in first-order logic is valid. He set to work on this problem and showed Newman a draft of what was to become his famous 1936 paper. Newman recognized the importance of Turing's work and, knowing that Church was working on that topic, arranged for Turing to visit Princeton where Church taught. Turing completed his doctorate at Princeton under Church's supervision in 1938 — in less than two years! The Institute for Advanced Study at Princeton is an august establishment (independent of the university) that assembles some of the world's greatest scientists including, at that time, Einstein, Gödel, and John von Neumann — another of the towering figures of 20th century mathematics. In the mid-1940s, von Neumann was involved in the design of one of the early electronic computers, the EDVAC (Electronic Discrete Variable Automatic Computer), and in 1945 he wrote a report describing that computer's design. A distinguishing feature of EDVAC's architecture was that it was a ***stored-program computer***, meaning that the program controlling the behaviour of the computer was to be stored in the computer's memory. This was in contrast to earlier electronic computers in which the program was hardwired; these were not general-purpose, i.e., universal, computers, but electronic devices dedicated to specific types of computations. To reprogram such machines effectively required re-wiring them, a much more awkward process than loading the new program on the computer's memory. The stored-program concept, a key element of what has come to be known as the ***von Neumann architecture***, has been the blueprint for all computers ever since. von Neumann, who was intimately familiar with Turing's work, acknowledged the intellectual debt of this concept to Turing's seminal paper.

This is how we have now come to have one computer on our desk running our browser, our text-processing software, our spreadsheets, our scientific computations, and so on, as opposed to having different devices, one for each of these tasks! In some sense, the software industry owes its existence to Turing's genius. No doubt, had he not thought of the universal Turing machine and had not von Neumann read and deeply understood Turing's paper, someone else would sooner or later have come up with the idea. It is interesting, however, that Turing came up with it in 1936, long before anyone was designing electronic computers. You never know how a great idea will germinate or where it will lead: A topologist happened to give lectures on the foundations of mathematics, inspiring a brilliant young mathematician to think about a somewhat obscure but deep unsolved problem, the resolution of which hinged on an idea that had radical implications for the engineering of electronic computers.