The Church-Turing Thesis

We now know that every Σ set or relation is effectively enumerable. The central thesis of recursion theory is that the converse also holds, so that we have:

Church-Turing Thesis. A set or relation is effectively enumerable iff it's Σ.

Since we know that a set is decidable iff it and its complement are both effectively enumerable, and we also know that a set is Δ iff it and its complement are both Σ, we see that the Church-Turing thesis entails that a set or relation is decidable iff it's Δ. Also, a partial function is calculable iff it's Σ, and a total function is calculable iff it's Δ.

"Calculable" here means calculable in principle: there is a mechanical procedure that, if carried out tirelessly and without error, will compute the function. The notion makes no allowance for senility, death, or limited disk space. If a computation takes up more bits of memory than there are particles in the universe, we still allow it. The theoretical aim is to give an extreme outer limit of what it's possible to compute, then to leave it to more practical-minded engineers to attempt to approximate that ideal in practice.

One cannot hope rigorously to prove the Church-Turing. Before we can rigorously prove things about decidability and effective enumerability, we first have to have mathematically precise characterizations of those notions, and we are looking to the thesis to give us such characterizations. There is, however, quite a body of evidence in the thesis' favor. Let us now survey some of it.

The biggest piece of evidence is simply that every known enumeration procedure generates a Σ set and every known decision procedure determines a Δ set. It's more than that. Every know enumeration procedure produces a set that is *obviously* Σ. Once you know a few tricks of the trade, it will be easy, once you know an algorithm of producing a set, to write down a Σ formula that describes the set. The only times we get stuck is when we don't really know the algorithm, or we don't know it explicitly. For example, we don't know how to write down a Σ formula that lists the code numbers of grammatical English sentences, but that's because, even though we presume there is an algorithm that generates the set, we don't know what it is.

After the Church-Turing thesis was proposed during the 1930s, a fair amount of effort was devoted specifically to the program of showing the thesis to be false by presenting a decision procedure that wasn't Δ. None of these efforts got anywhere. The utter failure to obtain a counterexample to the thesis makes it quite likely that there is no such counterexample, and it makes it a moral certainty that, if there is a counterexample, it must be an algorithm that is quite unlike any algorithm that we have today.

There are certain standard procedures for taking familiar algorithms and using them to create new algorithms; for example, substitution and recursive definition. The fact that the class of Σ partial functions is closed under all known techniques of this sort is another bit of evidence in favor of the Church-Turing thesis. Any counterexample to the thesis would have to involve some completely novel method of computation. There is no hope of getting a counterexample by combining familiar algorithms in complex ways.

During the 1930s and 1940s, many different people were working on the problem of understanding computability, which may different approaches and perspectives. All of them came up with the same answer. This convergence gives us reason to suppose that the answer they came up with was the right one. Generally speaking, if different clever people working independently on a problem all arrive at the same answer, this is reason to think that the answer arrived at is the right one. This is particularly so if the methods employed by different reseachers are highly dissimilar, since this lessens the likelihood that they have all made the same mistake.

If you come upon the same concept starting from several quite different approaches, this is reason to suppose that the concept you have reached is a basic and natural one; for it is likely to avoid the arbitrariness that often afflicts concepts that are only constructed from a single point of view.

Let me describe some of these methods that arrive at the identification of the effectively enumerable sets with the Σ sets. The proofs that these methods all yield the same result is labor intensive and not a little tedious, so I won't attempt it here.

Turing machines. Alan Turing, in his senior undergraduate thesis at Cambridge, attempted to give a model, pared down to its bare essentials, of what a human computing agent does when solving a computational problem by applying a system of rules. The agent writes symbols down in a zigzag fashion, starting at the left, proceeding until she reaches the right margin, then starting again at the left, this time one line down. Our first simplification is to cut the lines apart and glue them together, so that they all lie along a single very long line. Thus our agent writes symbols side-by-side along a very long tape.[1] Actually, we shall assume the tape is infinite, since

1 Being able to depict a problem pictorially may help us find a solution that would elude us if we were forced to do all our symbolic representations in one dimension. So if we were attempting to describe the creative processes by which new problem solutions are discovered, demanding that all our symbols be laid in a line would introduce terrible distortions. But that's not our aim. We want to look at the purely mechanical implementation of computational algorithms, a stage of problem-solving at which

we don't want to count a problem as unsolvable in principle if the only reason we've failed to solve it is that we've run out of scratch paper.

In deciding what to write next, the agent will sometimes look back over her previous work. There is no real loss of generality in supposing that the agent looks over her previous work one symbol at a time, so that we can think of the agent as having a movable tape reader that she passes over the tape one symbol at a time, moving sometimes one square to the left and sometimes one square to the right, and pausing sometimes to erase an old symbol or write a new one. In having her work one symbol at a time, we are just breaking down what the agent does into the simplest possible steps.

A finite, numbered list of detailed instructions tells the agent how to proceed with the computation. Things like, "If the square you are examining is blank, write the letter 'Q' in it and go to instruction 112" and "If the square you are examining has 'W' written in it, erase it and proceed to instruction 13" and "If the square you are examining has 'B' written in it, move one square to the right, and go to instruction 99." If the input number is n, the computation begins with the reader at the left end of a sequence of n "1" on an otherwise blank tape. If, when the computation finally ends, the reader is at the left end of a sequence of exactly m "1"s, then m is the output. If the computation never ends, then n isn't in the domain of the partial function being computed. If the machine halts on a certain input, that input is said to be *accepted* by a machine.

A mechanical device that implements such a system of instructions is called a *Turing machine*. Turing proposed such machines as a model of human computation. Of course, what the machine does isn't what the human computing agent does; it's a highly simplified and stylized

creativity is no longer called for.

mimicry of what the human agent does. But the differences are in inessential details, not in fundamental computational capacities. Everything a human computer[2] can do can be simulated, or so Turing proposes, by a Turing machine.

The details here are pretty arbitrary. The number of symbols can be few or many, as long as it's finite. The inputs and outputs can be Arabic numerals, instead of strings of "1"s. We can allow auxiliary tapes for scratch work. We can even allow *indeterministic* computations. These are machines programmed with conflicting sets of instructions, so that, at certain junctures, the machine chooses, arbitrarily, which instruction to follow.

However we work out the details, the result is the same. A function is computed by a Turing machine iff it is Σ. A set is accepted by a Turing machine iff it is Σ.

Register machines. Turing's machines were intended to provide a model of what human computing agents do. Register machines are intended as a model of what electronic computers do. A machine contains an unlimited number[3] of memory locations or *registers*, and a program for the machine consists of simple instructions for manipulating the numbers contained in those memory locations. Thus we can add 1 to the number in a particular register; we can set a particular register equal to 0; we can compare the contents of two different registers, then decide what to do next on the basis of whether the two contents are equal; and so on.

---

2 When Turing talked about a "computer," he meant a human computing agent, since at the time he wrote, in the early 30s, elecronic computers hadn't been invented yet. During the 40s (after the war; during the war he was busy breaking the German naval code), he went on to build one of the first electronic computers.

3 Keep in mind that we are attempting to characterize computability *in principle.*

We say that a register machine *accepts* a set of sentences S iff, for any n, if the machine is started with n in register 0 and 0 in all the other registers, the machine will eventually halt if n is in S, whereas if n isn't in S the computation will do on forever.

There is a register machine that accepts S if and only if S is Σ. Once again, this result is resilient, so that, for example, we get nothing new if we permit indeterministic register machines.

μ-recursive functions. Our next characterization looks at the ways calculable functions are defined in arithmetic, specifically, at the ways new calculable functions are defined on the basis of old ones. Two such methods we have discussed previously, substitution and recursive definition.

For example, if we have 0 and s, we can recursively define +, •, and E:

$x + 0 = x$

$x + (y+1) = s(x + y)$

$x \bullet 0 = 0$

$x \bullet (y+1) = (x \bullet y) + x$

$xE0 = 1$

$xE(y+1) = (xEy) \bullet x$

Here is another way to make new calculable functions out of old:

Definition. Given an n-ary partial function f, $\mu x_0[f(x_0,x_1,...,x_n) = 0]$ is the n-ary partial function defined as follows: Given $<x_1,...,x_n>$ as input, $\mu x_0[f(x_0,x_1,...,x_n) = 0]$ will be defined and equal to y iff $f(y,x_1,..,x_n)$ is defined and equal to 0 and, for each $z < y$, $f(z,x_1,...,x_n)$ is defined and different from 0.

Notice that, even if f is total, $\mu x_0[f(x_0,x_1,...,x_n) = 0]$ needn't be total, for there might not be any value of x_0 that makes $f(x_0,x_1,...,x_n)$ equal to 0.

It's clear that, if f is calculable, $\mu x_0[f(x_0,x_1,...,x_n) = 0]$ is also calculable. Just plug in successive numbers until you get the output 0.

> **Definition.** The *μ-recursive functions* constitute the smallest class of total partial functions that includes the successor function, the constant function 0 (the unary function that gives 0 for every input), and the projection functions (for each j and n, $j \leq n$, there is a projection function that takes $<x_1, x_2,...,x_n>$ to x_j; we need them for bookkeeping purposes), and is closed under substitution, recursive definition, and the μ operator.

A partial function is μ-recursive iff it is Σ.

Markov algorithms. Our next characterization of the Σ sets is based upon syntactic, rather than arithmetical, computations. We start with a finite alphabet. A *word* is a finite string of letters (including the empty string). A *production system* is a finite set of ordered pairs of words. A word σ is *derivable* from a word ρ iff there is finite system of transformations of the form

$$\alpha\hat{}\beta\hat{}\delta \rightarrow \alpha\hat{}\gamma\hat{}\delta,$$

where $<\beta,\gamma>$ is in the production system, that begins with ρ and ends with σ; here "^" denotes the concatenation operation. A set of words S is *accepted* by given production system iff, for any word σ, $\sigma \in S$ iff there is a derivation of the empty word from S. A set is Σ iff there is a production system that accepts it.

Representability in a theory. Our final characterization takes seriously the idea that, if there is a "proof procedure" for S, then, if n is in S, one ought to be able to right down a description of S on the basis of which it is possible to prove that n is in S. There must be some theory Γ that describes the natural number system and some predicate σ describing S such that σ([n]) is a consequence of Γ if and only if n is in S. We introduce a name for this state of affairs:

Definition. The formula σ *weakly represents* S in the theory Γ just in case, for any n, n is an element of S if and only if σ([n]) is a consequence of Γ.

If n is in S, you can prove that n is in S by providing a proof of σ([n]). If n isn't in S, you don't necessarily have any way of proving that n isn't in S. In cases, whenever n isn't in S, you can prove that n isn't in S by proving ¬σ([n]), σ is said to *strongly represent* S:

Definition. The formula σ *strongly represents* S in the theory Γ just in case, for any n, n is an element of S if and only if σ([n]) is a consequence of Γ, whereas n is outside S if and only if ¬σ([n]) is a consequence of Γ.

We'll actually prove the following result, instead of me asking you to take my word for it:

Theorem. A set natural numbers S is Σ iff there is a finite set of axioms Γ and a formula σ of the language of arithmetic that weakly represents S in Γ. S is Δ iff there is a formula σ that strongly represents S in Γ.

When we attempt to formalize ordinary mathematical reasoning, often we find ourselves working, not within a finite system of axioms, but within a finite system of axioms and axiom *schemata*. For example, the principle of mathematical induction is represented within the language of arithmetic, not by a single axiom, but by the *induction axiom schema*:

$$((R(0) \wedge (\forall x)(R(x) \rightarrow R(sx))) \rightarrow (\forall x)R(x))$$

Each of the infinitely many sentences you get from this schema by plugging in a formula for "R," then prefixing universal quantifiers to bind the resulting free variables is an *induction axiom.* Our earlier theorem is upheld when we allow axiom schemata as well as single axioms:

Theorem. For S a set of natural numbers, the following are equivalent:

S is Σ.

S is weakly represented within some finite system of axioms.

S is weakly represented within some finite system of axioms and axiom schemata.

S is weakly represented within some Σ system of axioms and axiom schemata.

Likewise, the following are equivalent:

S is Δ.

S is strongly represented within some finite system of axioms.

S is strongly represented within some finite system of axioms and axiom schemata.

S is strongly represented within some Σ system of axioms and axiom schemata.

Throughout its history, mathematics has been a demonstrative science. Because of the preeminence of the axiomatic method in mathematical reasoning, this theorem provides potent evidence for the Church-Turing Thesis. If there is an algorithm that enumerates the set S, then it ought to be possible to describe the algorithm precisely, and then to verify mathematically that computations are correct. If this verification looks anything like traditional mathematics, it can be formalized within a finite system of axioms and axiom schemata. S will be weakly represented within the axiom system, and so, according to the theorem, Σ.