Chapter 5

The Church-Turing Thesis

"Effective computability" is not a mathematical notion. Informally, an effectively computable function is a total function whose values can be obtained by mindlessly following an explicit procedure whose successive steps are determined by unambiguously explicit rules. Let *Eff* denote the collection of intuitively effective total functions. It is important to remember that *Eff* is *not a set* since it is not clearly defined in set theory.

The set Tot of total recursive functions is a mathematically defined collection of total recursive functions. The set *Tot* is defined in terms of primitive recursion, composition and minimalization. It is pretty clear that each of these operators preserves effective computability, but it is far less clear that *Tot* exhausts the effectively computable functions, so *Eff* $\subseteq$ *Tot*.

The proposition *Eff* $\subseteq$ *Tot* is the **Church-Turing thesis.** The Church-Turing thesis is far from being obvious, in light of the development thus far. True, Tot skirts an obvious diagonal argument, but avoiding one diagonal argument implies nothing about susceptibility to others that have not been thought of yet.

Since *Eff* is not a mathematically defined concept (that's the whole point of defining Tot!) one can't *prove* that $Tot = Eff$. But one can try to provide a philosophical or empirical argument for the thesis. The two standard arguments are as follows.

5.0.1 The Amazing Coincidence Argument

No language for defining intuitively effective functions has ever yielded a definition of a function outside of *Tot* (although many such notations fall short of capturing all of *Tot*, as we have seen). This is the argument usually quoted in textbooks. It is a kind of physicist's argument, like discovering that wave mechanics and matrix mechanics are the same theory.

The proof strategy is straightforward. Let X be a given class of functions computed by a given computational formalism. To show

$$Part \subseteq X,$$

we show that the basic functions are in X and that X is closed under the three partial recursive operators C, R, M. This is usually easy (because the number of primitives is so small— that's why it pays to develop computability theory in a "small" formalism). Demonstrating the converse:

$$X \subseteq Part,$$

can be toughter, depending on how rich or fussy the new formalism is. The basic strategy is this:

1. Use finite sequences of numbers to represent in a natural way instantaneous computational states of computations directed by program M.

2. Code these sequences as single numbers using the primitive recursive Gödel coding.

3. Write a primitive recursive function $init(m, \vec{x}) = s$ that converts a list $\vec{x}$ of input arguments into the Gödel code s of the initial computational state of the computation.

4. Then write a primitive recursive function $transit(s) = s'$ that transforms each state code number into its successor state code number to simulate computations.

5. Finally, we write a primitive recursive relation $Output(s, y) = y$ that determines whether the process has yet halted with output y.

The work involved is not deep and is similar to what we did when we programmed the universal function (Cf. Cutland for *lots* of this).

The preceding argument didn't impress either Church or Gödel. The fact that all the analyses "stop" at the same place doesn't necessarily mean that the capabilities of algorithmic mathematics have been exhausted. It might mean only that everyone has seized upon a natural, easily axiomatized subclass of effectively computable functions whose extension requires a much more subtle insight. That's hardly wild skepticism: physical possibilities violating Newtonian mechanics eluded us for over two centuries. Why shouldn't arcane styles of effective reasoning prove equally elusive?

5.0.2 Turing Machines

What did impress Gödel was Turing's argument based on **Turing machines**. A Turing machine consists of an infinite tape divided into discrete squares and a "read-write-head" connected to a finite state control mechanism. That's not strictly true, for there are no infinite tapes in the world. At best, there is a paper factory that adds more squares of tape upon demand. So we will assume that the input is written on a finite scrap of tape and that whenever the read-write head moves off of this scrap, a new square is added just in time. Then the position of the read-write head on the scrap can be measured from the left end of the scrap. Paying attention to these details at the outset will be useful when it comes time to show that Turing-computable functions are partial recursive.

Each square is either blank (0) or printed with a discrete mark (1). Hence, the set of possible tape marks is $Sym = \{0, 1\}$. Non-existent squares will also be thought of as "blank". Thus, the current state of the tape is a finite, Boolean sequence, which is understood to be a mapping τ from a finite, initial segment of the natural numbers to *Sym*. The current position of the read-write head on τ is some $n \in \mathrm{dom}(\tau)$. Notice, the head position can never be off of τ, since we assume that new squares are added whenever the head moves off of the previous tape scrap.

The set of possible actions on the tape by the machine is $Act = \{0, 1, R, L\}$, where "0", "1" stand for "print zero", and "print one", respectively, and "L" stand for "move right" and "move left", respectively. A **(deterministic) Turing machine** over alphabet *Sym* consists of a pair quadruple $(Q, \alpha, \sigma, q_0, q_h)$, where

- Q is a finite set of natural numbers called **control states**;
- $q_0, q_h \in Q$, α is a total function $\alpha : Q \times Sym \to Act$;
- σ is a total function $\sigma : Q \times Sym \to Q$.

Interpret $\alpha(q, b) = a, \sigma(q, b) = q'$ to mean that if the finite state control is currently in state q and the currently scanned symbol is b, then perform tape act a and reset the control state to q'. State q_h indicates that the answer is ready and is called the **halting state**. In our setup, the machine will keep going forever after entering the "halting state" but you may as well turn it off because the answer is at hand. Usually one thinks of the machine as stopping when the halting state is reached, but that really makes no difference and adds nuisance cases to the mathematics. State q_0 is called the **initial state**.

A **computational state** for $M = (Q, \alpha, \sigma, q_0, q_h)$ is a triple (q, τ, p), where

1. $q \in Q$ is a control state;
2. τ is a tape state (i.e., a finite sequence of symbols from *Sym*); and
3. p is a position on the tape (i.e., an element of the domain of τ).

Let Sym^* denote the set of all finite sequences of symbols. Since in set theory n denotes the set of all its predecessors, a sequence of length n is a total map on domain n. Thus we may write:

$$Sym^* = \{\tau : (\exists n \in \mathbf{N})\ \tau : n \to Sym\}.$$

Then the possible computational states for M are defined as follows:

$$CS = \{(q, \tau, p) : q \in Q \ \wedge\ \tau \in Sym^* \ \wedge q \in \mathrm{dom}(\tau)\}.$$

Define:

$$\begin{aligned}
\text{state}(q,\tau,p) &= q;\\
\text{tape}(q,\tau,p) &= \tau;\\
\text{position}(q,\tau,p) &= p.
\end{aligned}$$

If τ is a sequence, then let $\tau * \delta$ denote the concatenation of finite sequence τ on the front of finite sequence δ. Denote the result of writing symbol b in the xth position in sequence τ as $\tau[b/x]$. If $b \in Sym$, then let b^n denote the sequence $(b,\ldots,b)$ of length n. The set-theoretical definitions of these operations are as follows:

$$\begin{aligned}
\text{lh}(\tau) &= |\tau|;\\
\tau * \delta &= \tau \cup \{(\text{lh}(\tau)+i, \delta(i)) : i < \text{lh}(\delta)\};\\
\tau[b/x] &= \tau - \{(x,\tau(x))\} \cup \{(x,b)\};\\
b^n &= \{(i,b) : i \leq n\};\\
(b) &= \{(0,b)\}.
\end{aligned}$$

Let $M[x_1,\ldots,x_n]$ denote the **computation** of M on inputs $x_1,\ldots,x_n$. A computation is a sequence (finite or infinite) of computational states. The kth state in the computation is denoted by $M[x_1,\ldots,x_n](k)$, so we may think of $M[x_1,\ldots,x_n](k)$ as a total mapping

$$M[x_1,\ldots,x_n] : \mathbf{N} \to CS.$$

This function is defined recursively as follows.

The **initial state** of computation $M[x_1,\ldots,x_n]$ is defined as

$$M[x_1,\ldots,x_n](0) = (q_0, 1^{x_1+1} * (0) * \ldots * (0) * 1^{x_n+1}, 0),$$

For example, the input state for computation $M[2,0]$ is given by $(q_0,(1,1,1,0,1),0)$. Notice that the read-write head is conventionally initialized at the beginning of the first block of units. Think of the initial head position as the Turing machine's "input prompt".

Next, one must specify the successive computational states. Assume that

$$M[x_1,\ldots,x_n](k) = (q,\tau,p).$$

Then define:

$$\begin{aligned}
M[x_1,\ldots,x_n](k+1) &= (q'(q,\tau,p)), \tau'(q,\tau,p), p'(q,\tau,p));\\
q'(q,\tau,p) &= \sigma(q,\tau(p));\\
p'(q,\tau,p) &= \begin{cases} p & \text{if } \alpha(q,\tau(p)) \in \{0,1\};\\ p+1 & \text{if } \alpha(q,\tau(p)) = R;\\ p-1 & \text{if } \alpha(q,\tau(p)) = L \wedge p > 0;\\ 0 & \text{if } \alpha(q,\tau(p)) = L \wedge p = 0;\end{cases}\\
\tau'(q,\tau,p) &= \begin{cases} \tau[\alpha(q,\tau(p))/p] & \text{if } \alpha(q,\tau(p)) \in \{0,1\};\\ (0) * \tau & \text{if } \alpha(q,\tau(p)) = L \wedge p = 0;\\ \tau * (0) & \text{if } \alpha(q,\tau(p)) = R \wedge p = lh(\tau) - 1;\\ \tau & \text{otherwise.}\end{cases}
\end{aligned}$$

It remains to define the output, if any, of computation $M[x_1, \ldots, x_n]$. Since the so-called "halt state" may be entered infinitely often (the computation never really halts in our setup). There is an epistemological moral here— all that matters is that the machine put up a flag indicating that the answer is correct and that the machine never gets to "take back" that signal later. Halting is just one way to provide an irrevocable signal (the machine has to commit suicide, so it can't change its mind later). If you don't like the idea of the machine living its own life after it correctly answers your question, you can kill it yourself!

$$\begin{array}{rcl} M[x_1, \ldots, x_n] \downarrow^k & \Leftrightarrow & \text{state}(M[x_1, \ldots, x_n](k)) = q_0 \ \wedge \\ & & \quad (\forall k' < k)\ \text{state}(M[x_1, \ldots, x_n](k')) \neq q_0; \\ M[x_1, \ldots, x_n] \downarrow^k y & \Leftrightarrow & M[x_1, \ldots, x_n] \downarrow^k \ \wedge \\ & & \quad (\exists i, j)\ \text{tape}(M[x_1, \ldots, x_n](k)) = 0^i * 1^y * 0^j; \\ M[x_1, \ldots, x_n] \downarrow y & \Leftrightarrow & (\exists k)\ M[x_1, \ldots, x_n] \downarrow^k y; \\ M[x_1, \ldots, x_n] \downarrow & \Leftrightarrow & (\exists k)\ M[x_1, \ldots, x_n] \downarrow^k . \end{array}$$

For all the above, let $\uparrow$ indicate $\not\downarrow$.

Now we can link up computations with functions. Say that M computes ϕ just in case for each $\overrightarrow{x}, y$,

$$\phi(\overrightarrow{x}) \simeq y \Leftrightarrow M[\overrightarrow{x}] \downarrow y.$$

Then say that ϕ is Turing-computable just in case some Turing machine M computes ϕ. Let Tur denote the set of all Turing-computable functions and $Ttot$ denote the set of all total Turing computable functions.

Exercise 5.1 *Show that the following are Turing-computable:*

1. *The projection p^i_k;*
2. *the zero function o;*
3. *the successor function s.*

In applications, it is convenient to represent Turing machine M as the finite set of quadruples

$$M' = \{(i, b, j, a)) : q_i \in Q \wedge b \in Sym \wedge q_j = \sigma(q_i, b) \wedge a = \alpha(q_i, b)\},$$

where without loss of generality, one can assume that $dom(\sigma) = dom(\alpha)$ so that each such quadruple is defined.

5.0.3 Turing's Simulation Argument

Turing computations are opaque and ugly, even for simple functions like multiplication. But Turing's point for introducing them was never to use them.

They were the linchpin of Turing's argument that $\mathit{Eff} \subseteq \mathit{Tot}$. The argument goes something like this. Consider a mathematician following an algorithm. The algorithm specifies a finite set of rules for modifying scribbles on a notebook in light of the current state of mind of the mathematician. The mathematician may have boundless originality, but the states of mind relevant to following the algorithm are finite and discrete. Also, only finitely many distinct notational states can occupy a page of the scratch pad, else a microscope would be required to follow the algorithm (Turing wryly notes that Chinese characters seem to be an attempt to challenge this assumption). Simple reduction arguments turn the scribblings into bits on a linear tape, the mathematician into a finite state automaton (for the purposes of his involvement in the computation) and the algorithm into a set of rules for elementary operations on the tape. So the mathematician is simulated in all relevant respects by a formal Turing machine. Let Tur denote the set of all Turing-computable functions. The preceding argument purports to show *philosophically* that

$$\mathit{Eff} \subseteq Tur.$$

One can now prove *mathematically* that $Tur = Part$ along the lines described above. Thus

$$Tot = \mathit{Eff}.$$

Here is an interesting autobiographical description of the reception of the two arguments by Stephen C. Kleene. Since Kleene was a student of Church and invented much of computability theory, he was in a fine position to report!

> Church had been speculating, and finally definitely proposed, that the λ-definaable functions are all the effectively calculable function— ... which I in 1952... called "Church's thesis". When Church proposed [the CT] thesis, I sat down to disprove it by diagonalizing out of the class of the λ-definable functions. But quickly realizing that the diagonalization cannot be done effectively, I became overnight a supporter of the thesis.
>
> Gödel came to the Institute for Advanced Study [at Princeton] in the fall of 1933. According to a ... letter from Church..., Gödel "regarded [the CT] thesis as thoroughly unsatisfactory". Soon thereafter, in his lectures in the spring of 1934, Gödel took a suggestion that had been made to him by Herbrand in a letter in 1931 and modified it to secure effectiveness. The result was what is now known as "Herbrand-Gödel general recursiveness." ...
>
> In a February 15, 1965, letter to Martin Davis, Gödel wrote, "However, I was, at the time of these lectures [1934] not at all convinced that my concept of recursion comprises all possible recursions...".
>
> Church (1936) and I (1936a) published equivalence proofs for Herbrand-Gödel general recursiveness to λ-definability. So, under Church's thesis, there were now two exact mathematical characterizations of the intuitive notion of all effectively calculable functions....

> The last of the original three equivalent exact definitions of effective calculablity is computability by a Turing machine [1936-37]. ...
>
> For rendering the identification with effective calculability the most plausible— indeed, I believe compelling— Turing computability has the advantage of aiming directly at the goal [i.e., the mathematician simulation argument]....
>
> It seems that only after Turing's formulation appeared did Gödel accept Church'es thesis, which had then become the Church-Turing thesis.[1]

5.0.4 Turing-Computable Functions are Partial Recursive

The technical side of Turing's argument is that *Tur* $\subseteq$ *Part*, from which it follows immediately that *Tur* $\subseteq$ *Tot*. The proof is by a simulation argument. Code Turing computational states as natural numbers (they are already pairs of form (σ, p), where σ is a finite, Boolean sequence, so that's trivial— that's the advantage of dealing with the paper factory from the outset) and then implement the transition function as a *primitive recursive* function from states to states. For totality, reserve some number (e.g., 0) to stand for "computation crashes or input number fails to code a state". Then use minimalization to look for a halting state and read off the output. This illustrates the importance of minimalization— it's what patiently waits for the machine to halt— primitive recursion would have to possess a prior bound on how long the computation would take.

Exercise 5.2 *Carry out the proof.*

5.0.5 Partial Recursive Functions are Turing Computable

Well, if Turing did his job on the philosophical argument that effectively computable functions are Turing computable, then the fact that partial recursive functions are effective should entail that *Part* $\subseteq$ *Tur*! But one can also prove it directly.

Exercise 5.3 *Prove that* Part $\subseteq$ Tur*. The proof proceeds by induction on partial recursive derivation tree depth. You have already shown that the basic functions are Turing computable. It remains only to show that Turing computable functions are preserved under the operators C, R, M. Thus, you need to show that Turing machines can implement these operations.*

5.0.6 What Church's Thesis doesn't say

Turing's reductive argument does not show that Turing machines can compute whatever a physically implementable computing machine could compute or that

[1]Kleene 1981, op. cit. pp. 59-61.

human intelligence is Turing computable. It is only the intuitively effective or algorithmic that is treated by the argument. The argument that the mental state may be treated as though it were a member of a discrete, finite space works only because the mind is assumed to be working out a "mind-less" mathematical algorithm. To extend this to arbitrary mental or mechanical processes is quite another matter. Many cognitive scientists do assume that mentation is effective, but the reasons for this assumption are not unclear, as a Turing-style argument has not been given. Nor is it clear that such an argument can be given in the absence of a philosophically clear pre-theoretical understanding of the nature of mentation.

5.1 Arguments "by Church's Thesis"

Granting the Church-Turing thesis, any crisp procedural specification entitles us to infer that a partial recursive index exists for the function.

To compute k-ary ψ on inputs $\vec{x}$ do blah blah blah.

By the Church-Turing thesis (CT), there exists an n such that $\psi = \phi^k_n$.

Yippee! But *don't do it until I say you may. And then make sure you provide a procedure.* CT doesn't do the programming for you!

5.2 Acceptable Indexings[2]

In physics, it is a disaster to confuse "coordinate effects" with physical realities. Imagine a scientist using a microscope to look for the equator! Here is a computational analogy: programming languages are to computable reality as coordinate systems are to physical reality. Physical coordinates allow us to refer to physical events. Programs allow us to refer to computable functions. The arbitrariness of coordinates is handled by the fact that physical laws are invariant under the relevant group of coordinate transformations (inertial translations). Such laws are said to be What sorts of transformation should preserve computable reality? You guessed it: computable transformations.

More abstractly, sufficiently powerful computer languages may be viewed as numberings of the partial recursive functions. To be really careful about it, a numbering of *Part* is a surjective (onto) mapping:

$$\psi^{-}_{-} : \mathbf{N}^2 \to Part$$

where ψ^k_i denotes the i^{th} k-ary partial recursive function according to numbering ψ. Now let δ, ψ be numberings. Say that δ^{-}_{-} compiles into ψ^{-}_{-} just in case for each k there exists a total recursive c_k such that for each i, k, $\delta^k_i = \psi^k_{c_k(i)}$. "Compiles into" is a pre-order (reflexive and transitive). Say that δ^{-}_{-} intercompiles with ψ^{-}_{-} just in case each numbering compiles into the other. Intercompilation is an equivalence relation.

[2](Rogers, exercise 2-10).

Exercise 5.4 *To make sure you are awake and understand the definitions: prove that compilation is a pre-order (reflexive and transitive) and that inter-compilation is an equivalence relation (reflexive, transitive, and symmetric).*

Our special numbering ϕ is not arbitrary. It has a special structure. For example, it satisfies the universal and *s-m-n* theorems. Presumably, it satisfies many other conditions as well. But perversely enough, we will focus on these two curious properties. Say that ψ is acceptable just in case satisfies the conditions of the universal and *s-m-n* theorems; i.e., just in case:

1. $\exists u\ \forall n, \vec{x}\ \psi^2_u(i, \langle \vec{x} \rangle) \simeq \psi^{lh(\vec{x})}_n(\vec{x})$
   (universal machine property);

2. $\forall n, m\ \exists$ total recursive $s\ \forall i, n$-ary $\vec{x}, m$-ary$\vec{y}\ (\psi^n_{s(i,\vec{x})}(\vec{y}) \simeq \psi^{m+n}_i(\vec{x}, \vec{y}))$.
   (*s-m-n* property)

Now why would acceptability be of any interest? Because it characterizes the set of all numberings intercompilable with our original numbering ψ. When you stop to think that the empirical evidence for the Church-Turing thesis is that all natural programming systems yield numberings intercompilable with ψ, we see that the universal and *s-m-n* theorems as it were axiomatize all the computationally invariant structure of our indexing! That means we may kick free of all the arbitrary scaffolding and work only with the universal and *s-m-n* theorems! But first, as they always say, we have to prove it. The proof is a beautiful illustration of how the universal and *s-m-n* constructions interact.

Proposition 5.1 *Numbering ψ^-_- is acceptable $\Leftrightarrow$ ψ^-_- is intercompilable with numbering ϕ^-_-.*

We proceed by a series of lemmas.

Lemma 5.2 *ψ^-_- satisfies the universal machine property*
$\Rightarrow$ ψ^-_- compiles into ϕ^-_-.

Proof. Suppose that ψ^-_- satisfies the universal property. Then for some u,

$$\forall n, \vec{x}\ (\psi^2_u(i, \langle \vec{x} \rangle) \simeq \psi^{lh(\vec{x})}_i(\vec{x})). \tag{5.1}$$

To get rid of the coding on the $\vec{x}$, define:

$$\delta(i, \vec{x}) \simeq \psi^2_u(i, \langle p^1_m(\vec{x}), \ldots, p^m_m(\vec{x}) \rangle). \tag{5.2}$$

Since $rng(\psi^-_-) = Part = rng(\phi^-_-)$, there exists a z such that

$$\delta = \phi^{m+1}_z. \tag{5.3}$$

Since ϕ^-_- has the *s-m-n* property, there is a total recursive s such that for all $i, x, \vec{y}$,

$$\phi^m_{s(i,x)}(\vec{y}) \simeq \phi^{m+1}_i(x, \vec{y}). \tag{5.4}$$

By composing in the constant function c_z, we obtain the total recursive r such that:

$$r(x) = s(z, x). \tag{5.5}$$

Now we use the above to calculate:

$$\begin{aligned}
\phi^m_{r(i)}(\vec{y}) &\simeq [5.4];\\
\phi^m_{s(z,i)}(\vec{y}) &\simeq [5.4];\\
\phi^{1+m}_z(i, \vec{y}) &\simeq [5.3];\\
\delta(i, \vec{y}) &\simeq [5.2];\\
\psi^2_u(i, \langle p^1_m(\vec{y}), \ldots, p^m_m(\vec{y})\rangle) &\simeq \psi^2_u(i, \langle\vec{y}\rangle)\\
\psi^2_u(i, \langle\vec{y}\rangle) &\simeq [5.1]\\
&\simeq \psi^m_i(\vec{y}).
\end{aligned}$$

Thus, r is the desired, total recursive compiler for arity m. ⊣

Lemma 5.3 *ψ^-_- compiles into ϕ^-_- $\Rightarrow$ ψ^-_- satisfies the universal property.*

Proof. Suppose that ψ^-_- compiles into ϕ^-_-. Then for each k there exists a total recursive c such that

$$\psi^m_i \simeq \phi^m_{c(i)}.$$

By the universal theorem for ϕ^-_-, there is a u such that for all $\vec{y}$:

$$\phi^m_{c(i)}(\vec{y}) \simeq \phi^2_u(c(i), \langle\vec{y}\rangle).$$

So we obtain a partial recursive function

$$\delta(i, \vec{y}) \simeq \phi^2_u(c(i), \langle\vec{y}\rangle).$$

Since ψ^-_- is onto $Part$, there exists a z such that for all $\vec{y}$:

$$\psi^2_z(i, \langle\vec{y}\rangle) \simeq \delta(i, \vec{y}).$$

Thus, for all $\vec{y}$, we have:

$$\begin{aligned}
\psi^2_z(i, \langle\vec{y}\rangle) &\simeq \delta(i, \vec{y})\\
&\simeq \phi^2_u(c(i), \langle\vec{y}\rangle)\\
&\simeq \phi^m_{c(i)}(\vec{y})\\
&\simeq \psi^m_i(\vec{y}).
\end{aligned}$$

So z is a universal index for ψ^-_-. ⊣

Lemma 5.4 *ψ^-_- satisfies the* s-m-n *property $\Rightarrow$ ϕ^-_- compiles into ψ^-_-.*

Exercise 5.5 *Prove it.*

Lemma 5.5 *ψ^-_- intercompiles with ϕ^-_- $\Rightarrow$ ψ^-_- satisfies the* s-m-n *property.*

Exercise 5.6 *Prove it.*