

# Motivating the Church–Turing Thesis in the Twenty-First Century

R. Gregory Taylor

New Jersey City University

Jersey City, New Jersey 07305

rgtaylor@interactive.net

## ABSTRACT

Theory of Computation students frequently fail to appreciate the significance of the Church–Turing Thesis for one of two reasons. First, there is a tendency, on the part of students, to regard Church–Turing as tautologous and, consequently, devoid of important content. Second, there is a contrary impulse to view Church–Turing as unmotivated or even implausible. We describe our experience using simulation software in an effort to combat these two tendencies.

## Keywords

Computability theory, Church–Turing Thesis, Turing machine, Markov algorithm, register machine, vector machine.

## 1. INTRODUCTION

We begin by recalling the usual formulation of the Church–Turing Thesis:

**Church–Turing Thesis.** If (number-theoretic) function  $f$  is effectively calculable, then  $f$  is Turing-computable. Equivalently, if function  $f$  is not Turing-computable, then  $f$  is not effectively calculable.

Properly understood, Church–Turing provides a concise summary of the classical Theory of Computation (see below). It constitutes a recurring theme in any good theory course. In addition, it provides an opportunity to relate Computer Science to the rest of the liberal arts curriculum (mathematics, the philosophy of mind, cognitive science, and even anthropology). Finally, it is a landmark of twentieth-century intellectual life. If students leave a theory course with anything, they should leave with an appreciation of Church–Turing. We believe that, often enough, this is not what happens.

## 2. Mathematics and Philosophy

Typically, if one’s goal is to introduce the Church–Turing Thesis within the classroom, one begins by reviewing the concept of an

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

SIGSCE '98 Dublin, Ireland

© 1998 ACM

*algorithm.* It will be emphasized that the algorithm concept is inexact to the extent that the notions used to characterize it—“next step,” “carrying out,” “result,” and so forth—are being left rather vague. Students are asked to view the algorithm concept as part of the philosophy of mathematics.

Next, the concept of an *effectively calculable function* or *effectively computable function* is introduced—any function  $f$ , partial or total, for which an algorithm exists. Due to the use of the algorithm concept in characterizing an effectively calculable function, the latter concept is, of course, similarly philosophical. One stresses the importance of not confusing the philosophical concept of an effectively computable function with the mathematically rigorous concept of a Turing-computable function: they are two distinct concepts—one philosophical (but concerning mathematics), the other genuinely mathematical.

- The philosophical notion of an effectively calculable function is one that underlies the culture of mathematics. It is part of every mathematician’s informal sense of the subject matter.
- There is nothing informal about the concept of a Turing-computable function: a Turing-computable function is one computed by a single-tape Turing machine, where the latter is a quintuple of sets and functions satisfying certain conditions.

This is not to say that the two concepts—one philosophical and one mathematical—are unrelated, however, and that is precisely what Church–Turing is all about. Our point is that the two terms “effectively calculable function” and “Turing-computable function” are not synonymous.

## 3. Empirical Justification

By virtue of the fact that Turing machines were intended as an analysis of the concept of computation, we take Turing in 1936 to be claiming that (1) *number-theoretic function  $f$  is effectively calculable if and only if  $f$  is Turing-computable.* Students are reminded that (1) is not itself a mathematical proposition, since it makes use of the inexact notion of effective calculability. Hence it would be misguided to try to prove (1). Rather, one seeks to provide justification for it. Of course, since (1) is a biconditional, it has two parts: (a) *If  $f$  is effectively calculable, then  $f$  is Turing-computable* (Church–Turing) and (b) *If  $f$  is Turing-computable, then  $f$  is effectively calculable.*

As claims about computation, (1a) and (1b) have different status. First, the student is encouraged to see that proposition (1b) is obvious. After all, if there is a Turing machine  $M$  that computes  $f$ , then evidently  $f$  is associated with an algorithm, namely, the algorithm embodied in the state diagram of  $M$ . On the other hand, (1a) is not obvious: might there not be some effectively calculable function  $f$  whose definition is so convoluted that no Turing machine could model the algorithm involved in computing this particular  $f$ ? If so, (1a) would be false. (The mere fact that this question even makes sense means that Church–Turing is not obviously true.) Once it has been established that Church–Turing is not obvious, are there reasons to think that it might be true? Of course, the answer is yes, and now it is time to review a series of equivalence results.

- Assume that the first mathematical model of an effectively calculable function considered is the Turing machine model. An uncountably infinite class of number-theoretic functions is seen to be partitioned into those that are Turing-computable and those that are not.
- Perhaps the partial recursive functions are introduced next. In that case, it is shown that the class of partial recursive functions is identical to the class of Turing-computable functions.
- If Markov algorithms are considered, it is shown that the class of Markov-computable functions is none other than the class of Turing-computable functions (see Examples 2 and 3 below).
- Analogous equivalence results relate Turing computability to register-machine computability, Post computability,  $\lambda$ -definability, and vector-machine computability.

It is expected that students will be impressed by the fact that such apparently dissimilar proposals regarding computability should turn out, in the end, to capture the very same class of functions.

What we call the *Argument from the Convergence of Dissimilar Ideas* is introduced so as to establish the bearing of these equivalence results on the question of the truth of the Church–Turing Thesis. First, the fact that, for over 60 years now, every attempt to characterize the concept of computability has netted the same class of functions suggests that this class of functions is a “natural” class—one such that the mathematical properties of its members make it the natural outcome of diverse analyses. The next step in the argument is to claim that this natural class can be none other than the class of effectively calculable functions. The empirical reasoning behind this second step is the following. We ask the question, If the class of Turing-computable functions were, in fact, a proper subset of the class of effectively calculable functions, might we not expect that one or more of the alternative characterizations of computability would turn out to encompass functions that are not Turing-computable? But the plain fact is that, after many years of alternative models, none has produced a single function that does not demonstrably fall under Turing’s model. This probably means that no such function exists.

#### 4. Decline of the Computability Concept

Some mention of arguments opposed to Church–Turing is always in order. But that is not our present focus. Rather, we wish to consider two obstacles to the instructor’s instilling a proper appreciation of Church–Turing and its importance for the Theory of Computation.

Although no student ever objects to either (1a) or (1b), we are convinced that many students do not understand either of them. And this is for one of two reasons. First, most students have no idea whatsoever of our intuitive concept of an effectively calculable function. Similarly, talk of our pretheoretic concept of computability is unlikely to have the intended effect. There is nothing unusual in this. It takes a long time to acquire a stance that is sufficiently objective—or sufficiently philosophical—to enable a student to distinguish *analysandum* from *analysans*. More likely than not, having just acquired the notion of a Turing-computable function, the student conflates that technical notion (analysans) with the philosophical concept of a computable function (analysandum).<sup>1</sup> In any case, no student ever questions Church–Turing, and we suspect that this is because, from the student’s perspective, the two terms involved are synonymous anyway. One way to discourage this misconception is to address it directly by emphasizing the distinction between an intuitive notion and the variety of technical concepts that have been introduced in order to characterize it. Since not much class time can be devoted to this issue, however, such direct efforts have, in the past, had but limited success.

As we approach the twenty-first century, there is a second reason why our students fail to appreciate the remarkable claim embodied in the Church–Turing Thesis: namely, our students may well lack any robust, pretechnological concept of computation. Increasingly, computation is held to be the exclusive domain of calculators and digital computers. This may in turn mean that, from our students’ point of view, the concept appearing in the antecedent of (1a) belongs not to the philosophy of mathematics, as we have suggested, but, rather, to computer design theory. (In that case, the two concepts appearing in (1a) would both be viewed as technical in character.) Moreover, no mere slide-show of the history of computing (written records of manual reckoning, numerical tables used in calculating, historical computing devices, and so forth) is likely to reinstate what has been lost as the result of an omnipresent technology.<sup>2</sup> Perversely, the solution that we propose involves introducing yet more technology.

---

<sup>1</sup> This tendency will be apparent to the extent that students use the terms “computable” and “Turing-computable” interchangeably without citing the Church–Turing Thesis. The situation becomes truly hopeless if the students’ textbook appears to be doing the same thing.

<sup>2</sup> Perhaps some readers will feel that we are overplaying this. But our experience has shown that at least some students—perhaps still a minority—have become dependent upon hand calculators even for very rudimentary computational tasks, e.g., integer division by 2. What does this portend for the viability of a widespread, transcendent concept of effective calculability? Even those who would deny it any decisive impact will surely





In § 9.1<sup>3</sup> we saw in effect that regular expressions may be used to characterize or define token classes. For example, setting aside the issue of reserved words for the moment, the class of Ada identifiers is the language denoted by regular expression<sup>4</sup>

(A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|\_|.|\?|(A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|0|1|2|3|4|5|6|7|8|9))\*

The token class *tok\_integer\_literal* is denoted by

(0|1|2|3|4|5|6|7|8|9)|\_|.|\?|(0|1|2|3|4|5|6|7|8|9))\*

The token class *tok\_end* is denoted by

E.N.D

and the token *tok\_relational\_operator* by

=|<.>|<|>|<.=|>.=

By Theorems 9.12 and 9.13, we have that each such class of strings may be associated with a deterministic finite-state automaton. Using the standard abbreviations introduced at the end of § 9.2, we present transition diagrams for four finite-state automata corresponding to the token classes just cited. (See Figure 2, where the finite-state automata presented are not fully defined. In fact, the finite-state automaton for *tok\_identifier* given in Figure 2 defines a superset of the set of Ada identifiers. Why? Hint: Is the string *procedure* an Ada identifier?) Largely as a consequence of this situation, it turns out that finite-state automata are adequate as models of lexical analysis. In fact, however, in programming a scanner, one implements not finite-state automata themselves but, rather, related machines that we shall call *extended finite-state automata* as represented in *extended transition diagrams* or *extended state diagrams*. An extended finite-state automaton will in general represent several token classes simultaneously—for example, the extended state diagram of Figure 3 represents the token classes *tok\_end*, *tok\_endif*, *tok\_if*, *tok\_else*, and (a superset of) *tok\_identifier*. Note that the extended finite-state automaton stipulates what character may follow any member of the token classes *tok\_end* or *tok\_else* and so forth: The label *other* might, in Figure 3, be taken to mean any alphanumeric character, whereas *delimiter* will designate either *white space*,<sup>5</sup> a semicolon, or a period. The reference to delimiters within transition diagrams is necessary to accommodate the fact that input tokens are not presented in isolation but are, instead, embedded within an input stream. Furthermore, this aspect of our diagrams captures a certain *greediness* on the part of a lexical analyzer: given input string *S* consisting of alphabetic characters and delimiters, the extended

<sup>3</sup> All such references are to R. Gregory Taylor, *Models of Computation and Formal Languages* (Oxford University Press: New York, 1998).

<sup>4</sup> For languages such as Pascal or Ada, which are not case sensitive, we assume that, say, a function *convert\_to\_upper( )* is being called to convert all lowercase letters to uppercase. This amounts to regarding the strings *num* and *NUM* as identical tokens requiring but a single entry in the symbol table. In any case, our regular expression here need account only for uppercase letters.

<sup>5</sup> By *white space*, one intends a space, a tab, a linefeed, or an end-of-file marker.