Copyrighted Material

For more information on Alan Turing's SYSTEMS OF LOGIC: THE PRINCETON THESIS, click here: http://press.princeton.edu/titles/9780.html

The Birth of Computer Science at Princeton in the 1930s

ANDREW W. APPEL

The "Turing machine" is the standard model for a simple yet universal computing device, and Alan Turing's 1936 paper "On computable numbers . . . " (written while he was a fellow at Cambridge University) is the standard citation for the proof that some functions are not computable. But earlier in the same decade, Kurt Gödel at the Institute for Advanced Study in Princeton had developed the theory of recursive functions; Alonzo Church at Princeton University had developed the lambda-calculus as a model of computation; Church (1936) had just published his result that some functions are not expressible as recursive functions; and he had stated what we know as Church's Thesis: that the recursive functions characterize exactly the *effectively calculable* functions. In hindsight, the first demonstration that some functions are not computable was Church's.

It was only natural that the mathematician M. H. A. Newman (whose lectures on logic Turing had attended) should suggest that Turing come to Princeton to work with Church. Some of the greatest logicians in the world, thinking about the issues that in later decades became the foundation of computer science, were in Princeton's (old) Fine Hall in the 1930s: Gödel, Church, Stephen Kleene, Barkley Rosser, John von Neumann, and others. In fact, it is

Copyrighted Material

hard to imagine a more appropriate place for Turing to have pursued graduate study. After publishing his great result on computability, Turing spent two years (1938–38) at Princeton, writing his PhD thesis on "ordinal logics" with Church as his adviser.

If Turing was not the first to define a universal model of computable functions, why is the Turing machine the standard model? These three models—Gödel's recursive functions, Church's λ-calculus, and Turing's machine—were all proved equivalent in expressive power by Kleene (1936) and Turing (1937). But Turing's model is, most clearly of the three, a *machine,* with simple enough parts that one could imagine building it. As Solomon Feferman explains in his introduction to Turing's PhD thesis later in this volume, even Gödel was not convinced that either λ-calculus or his own model (recursive functions) was a sufficiently general representation of "computation" until he saw Turing's proof that unified recursive functions with Turing machines. That is, Church proved, and Turing independently re-proved, that some functions are not computable, but Turing's result was much more convincing about the *definition* of "computable."

Turing's "On computable numbers" convinced Gödel, and the rest of the world, in part because of the *philosophical* effort he put into that paper, as well as the mathematical effort. Turing described a process of computation as a human endeavor, or as a mechanical endeavor, in such a way that no matter which of these endeavors was dearest to the reader's heart, the result would come out the same: the Turing machine would express it. In contrast, it was not at all obvious that the Herbrand-Gödel recursive functions or the λ-calculus really constitutes the essence of "computation." We know that they do only because of the proof of equivalence with Turing machines.

The real computers of the 1940s and 1950s, like those of today, were never actually *Turing machines* with a finite control and an unbounded tape. But the electronic computers that *were* built, on both sides of the Atlantic, by von Neumann and others, were heavily (and explicitly) influenced by Turing's ideas, so that from the very beginning the field of computer science has often referred to computers in general as Turing machines—especially when considering their expressive power as universal computation devices.

Copyrighted Material

What became of the other two models—recursive functions and λ-calculus? Most mathematicians working in computability theory use the theory of recursive functions; computer scientists working in computational complexity theory use both Turing machines and recursive functions. Turing himself used λ-calculus in his own PhD thesis, but, as Feferman explains,

> One reason that the reception of Turing's [PhD thesis] may have been so limited is that (no doubt at Church's behest) it was formulated in terms of the λ-calculus, which makes expressions for the ordinals and formal systems very hard to understand. He could instead have followed Kleene, who wrote in his retrospective history: "I myself, perhaps unduly influenced by rather chilly receptions from audiences around 1933–35 to disquisitions on λ-definability, chose, after general recursiveness had appeared, to put my work in that format. I cannot complain about my audiences after 1935."

For Feferman and Kleene, and for other mathematicians working in the field known as "recursive function theory," the particular *implementations* of functions (as described in λ-calculus) are rarely useful, and it is usually sufficient (and simpler) to talk more abstractly about the *existence* of implementations, that is, about definability and about enumerations of computable functions. Soare (1996) points out that the very name of the field (in mathematics) "recursive function theory" was invented by Kleene; Soare suggested "computability theory" as a more descriptive name for the field, and pointed out that Turing and Gödel used "computable" in preference to "recursive." Of course, Soare is both a mathematician and a computer scientist, and it is my impression that many of the latter used the term "computable" more frequently than "recursive" for decades before 1996, influenced (for example) by Martin Davis (PhD 1950 under Church).

So there were several models of computation, all known (by the end of the 1930s) to be equivalent: recursive functions, λ-calculus, Turing machines, and in fact others; for a few decades, mathematicians studied what can be represented as recursive functions, while the computer scientists studied what can be calculated by Turing machines.

Copyrighted Material

But the λ-calculus did not disappear. In 1960 it became the explicit model of the Lisp programming language (invented by John McCarthy, 1927–2011; Turing Award 1971, PhD Princeton 1951 under Solomon Lefschetz). And λ-calculus is the implicit model of the Algol programming language (Perlis and Samelson 1958). Almost all programming languages in use today are descended from Lisp and Algol. Notions and mechanisms of variable binding, scope, functions, parameter passing, expressions, and type checking are all imported directly from Church's λ-calculus.

When the computers ("von Neumann machines") of the 1950s were built, with their (necessarily) sequential and mechanical universal control systems à la Turing, it was noticed that they were difficult to program. Programming these computers became easier with languages for specifying recursive functions (i.e., computations) that emphasized, to the degree possible with the technology of the time, *functions* (instead of procedures), *variables* (instead of registers), *binding* (instead of the "move" instruction), and *typed data* (instead of bit strings). All of this is from Church, and none of it is from Turing, Gödel, Kleene, or von Neumann.

Some mathematicians' criticisms of Church have to do with his reputation for pedanticism and excessive rigor: Hodges (1983, p. 119) writes that Turing "was reduced to attending Church's lectures, which he found ponderous and excessively precise." In part this reputation is undeserved. Feferman (1988, p. 120) writes that this "characterization of Church's style and personality" is "fair enough. . . . But it should be added that Church was (and is) noted for the great care and precision of his writing and lecturing, and these virtues probably benefited Turing—whose own writing was rough and ready and prone to minor errors." Robert Soare, who took classes from Church as an undergraduate at the beginning of the 1960s, says that Church's lectures on computability theory were indeed precise but "made the subject exciting"; Church was a better teacher than most math professors at Princeton.[1]

Still, Kleene and Feferman clearly agree that λ-calculus was not the most perspicuous vehicle for Turing to use in his PhD thesis, or for mathematicians to do many kinds of computability theory. This is because (typically) the mathematics they are doing depends only on the *computability* of a function,

1 Robert Soare, personal communication, December 12, 2011.

Copyrighted Material

not on *which method* is used to compute it. In contrast, the engineers and programmers who have written programs from 1950 to the present are forced to write down a precise formulation of the function; otherwise we have *bugs*. So the notation for writing down representations of computable functions must be precise, and must also be both readable and writable (by humans and by other computer programs) both in the small and in the large. This is where the descendants of Church's notation work better than those of Turing's.

Some of the ways in which early programming languages differed from λ-calculus were forced by the limitations of early computers. Turing machines with an infinite tape and unbounded time can nicely simulate the λ-calculus. The slow computers of the 1960s and 1970s with their tiny memories forced programmers, even those who used Lisp and Algol, to split the difference between Church and von Neumann in how they wrote down their algorithms. But in the 1980s and 1990s, as computers became more powerful, it was possible to develop and apply programming languages (such as ML and Haskell[2]) that were even closer to Church's λ-calculus, and consciously so.

This brings me to the subject of Turing's Princeton PhD thesis, the content of the current volume. Here, Turing turns his attention from computation to logic. Gödel and Church would not have called themselves computer scientists: they were mathematical logicians; and even Turing, when he got his big 1936 result "On computable numbers," was answering a question in *logic* posed by Hilbert in 1928.

Turing's thesis, "Systems of Logic Based on Ordinals," takes Gödel's stunning incompleteness theorems as its point of departure. Gödel had shown that if a formal axiomatic system (of at least minimal expressive power) is consistent, then it cannot be complete. And not only is the system incomplete, but the formal statement of the consistency of the system is true and unprovable if the system is consistent. Thus if we already have informal or intuitive reasons for accepting the axioms of the system as true, then we ought to accept the statement of its consistency as a new axiom. And then we can apply the same considerations to the new system; that is, we can iterate the process of adding consistency statements as new axioms. In his thesis, Turing investigated that process systematically by iterating it into the constructive transfinite, taking

2 Named after another great logician, Haskell Curry, who was also visiting Princeton in 1938.

Copyrighted Material

unions of logical systems at limit ordinal notations. His main result was that one can thereby overcome incompleteness for an important class of arithmetical statements (though not for all).

It is clear that Turing regards the formalization of mathematics as a desirable goal. He excuses himself at one point (on pp. 9–10 of the manuscript):

> There is another point to be made clear in connection with the point of view we are adopting. It is intended that all proofs that are given [in this thesis] should be regarded no more critically than proofs in classical analysis. The subject matter, roughly speaking, is constructive systems of logic, but as the purpose is directed towards choosing a particular constructive system of logic for practical use; an attempt at this stage to put our theorems into constructive form would be putting the cart before the horse.

Here it is clear that Turing is a logician and not just a great mathematician; few mathematicians believe that it would be a useful purpose to choose a constructive system of logic for *practical* use, and no ordinary mathematician would excuse himself for being no more rigorous than a mathematician.

Just as one of the strengths of Turing's great 1936 paper was its philosophical component—in which he explains the motivation for the Turing machine as a model of computation—here in the PhD thesis he is also motivated by philosophical concerns, as in section 9 (p. 60 of the manuscript):

> We might hope to obtain some intellectually satisfying system of logical inference (for the proof of number theoretic theorems) with some ordinal logic. Gödel's theorem shows that such a system cannot be wholly mechanical, but with a complete ordinal logic we should be able to confine the non-mechanical steps entirely to verifications that particular formulae are ordinal formulae.

Turing greatly expands on these philosophical motivations in section 11 of the thesis. His program, then, is this: We wish to reason in some logic, so that our proofs can be mechanically checked (for example, by a Turing machine). Thus we don't need to trust our students and journal-referees to check our proofs. But no (sufficiently expressive) logic can be complete, as Gödel

Copyrighted Material

showed. If we are using a given logic, sometimes we may want to reason about statements unprovable in that logic. Turing's proposal is to use an ordinal logic sufficiently high in the hierarchy; checking proofs in that logic will be completely mechanical, but the one "intuitive" step remains of verifying ordinal formulas.

Unfortunately, it is not at all clear that verifying ordinal formulas is in any way "intuitive." Feferman (1988, sec. 6) estimates that "the demand on 'intuition' in recognizing 'which formulae are ordinal formulae' is somewhat greater than Turing suggests." Feferman concludes his essay included in this volume with a mention of his and Kreisel's subsequent approaches to this problem, between 1958 and 1970.

Turing, in the thesis, recognizes significant problems with his ordinal logics, which can be summarized by his statement (manuscript, p. 73) that "with almost any reasonable notation for ordinals, completeness is incompatible with invariance" (and see also Feferman's essay).

But the PhD thesis contains, almost as an aside, an enormously influential mathematical insight. Turing invented the notion of oracles, in which one kind of computer consults from time to time, in an explicitly axiomatized way, a more powerful kind. Oracle computations are now an important part of the tool kit of both mathematicians and computer scientists working in computability theory and computational complexity theory (see Feferman 1992; Soare 2009). This method may actually be the most significant result in Turing's PhD thesis.

So the thesis exhibits Turing as logician. Alonzo Church also continued to be a logician, as in 1940 he published "A Formulation of the Simple Theory of Types," setting out the system now known as higher-order logic. As a practical means of actually doing mechanized reasoning, Turing's 1938 result was not nearly as influential as Church's higher-order logic.

In many other fields of engineering, such as the construction of bridges, chemical processes, or photonic circuits, the applicable mathematics is from analysis or quantum mechanics (see Wigner 1960, "The Unreasonable Effectiveness of Mathematics in the Natural Sciences"). But software does not (principally) rely on continuous or quantum artifacts of the natural world,

Copyrighted Material

where that kind of math works so well. Instead, software follows the discrete logic of bits, and it obeys axioms specified by the engineers who designed the instruction-set architecture of the computer, and by those who specified the semantics of the programming languages. Thus the applicable mathematics is, in fact, logic (see Halpern et al. 2001, "On the Unusual Effectiveness of Logic in Computer Science").

It might seem that the Boolean algebra of bits is simpler than real analysis, but the problem is that software systems are so complex that the reasoning is difficult. Thus in the twenty-first century many computer scientists do mechanized formal reasoning, and the most significant application domain for mechanized proof is in the verification of computer software itself. Software is large and complex, and for at least some software it is very desirable that it conform to a given formal specification. The theorems and proofs are too large for us to reliably build and maintain by hand, so we mechanize.

Mechanized proof comes in two flavors; the first flavor is fully automated. *Automated theorem proving* is the use of computer programs to find proofs automatically. *Automatic static analysis* is the use of computer programs to calculate behavioral properties of other computer programs, sometimes by calling upon automated theorem provers as subroutines to decide the validity of logical propositions. Do not be frightened by Turing's result that this problem is uncomputable; his result is simply that no automated procedure can decide the provability of *every* mathematical proposition, and no automated procedure can test nontrivial properties of *every* other program.[3] We do not need to prove *every* theorem or analyze *every* program; it will suffice to automatically prove many useful theorems, or analyze useful programs. Some automated provers work in undecidable logics, and (therefore) sometimes fail to find the proof. In those cases, the user is expected to simplify or reformulate the theorem as necessary, or provide hints. We would not ask Fermat to reformulate his Last Theorem for the convenience of Wiles; but when the theorem is "This horrible program meets its specification," we might well rewrite the program to make it more easily reasoned about. Other automated provers work in decidable logics—for example, Presburger arithmetic or Boolean satisfiability.

3 Actually, this generalization of Turing's 1936 result about halting is known as Rice's theorem (1953).

Copyrighted Material

Do not be frightened by Cook's result (1971) that satisfiability is NP-complete; that result is simply that no (known) automated procedure can solve *every* instance in polynomial time. In practice, SAT-solvers are now a big industry; they are quite effective in solving the actual cases that come up in theorem-proving applications. (Of course, SAT-solvers are not so effective in solving problems that arise from *deliberately* intractable problems, such as cryptography.) The extension of SAT-solvers to SMT (satisfiability modulo theories) is also now a big academic and commercial industry. Many of these solvers use variants of the Davis-Putnam algorithm for resolution theorem proving, discovered in 1960 by Martin Davis (PhD 1950 under Church) and Hilary Putnam (PhD UCLA 1951; in 1960 a colleague of Church's at Princeton).

The other flavor of mechanized proof is the use of computer programs to *check* proofs automatically, and to *assist* in the bureaucratic details of their construction. These are the proof assistants. One of the earliest of these was Robin Milner's LCF (Logic for Computable Functions) system (Gordon et al. 1979). Milner was influenced by the work of Church and by that of Dana Scott (PhD 1958 under Church), Christopher Strachey (a fellow student of Turing's at Cambridge, and one of the first to program the ACE computer in 1951), and Peter Landin (a student of Strachey's). Strachey, Landin, and Milner, all British computer scientists, were important figures in the application of Church's λ-calculus and logic to the design of programming languages and formal methods for reasoning about them.

Although some proof assistants use first-order logics (i.e., logics where each quantifier ranges over elements of a particular fixed type), for the expression of mathematical ideas it is much more convenient to use higher-order logics (i.e., where the type of a quantifier can itself be a variable bound in an outer scope). One of the earliest higher-order logics is Church's "simple theory of types" (1940), but even more expressive (and, to my taste, more useful) logics have dependent types, where the type of one variable may depend on the value of another. Such logics include LF (the Logical Framework) and CoC (the Calculus of Constructions). Proof assistants such as HOL (using the simple theory of types), Twelf (using LF), and Coq (using CoC) are now routinely used to specify and prove substantial theorems about computers and computer programs.

Copyrighted Material

Not only theorems about software; sometimes these proof assistants are even used to prove theorems in mathematics. Georges Gonthier (2008) used Coq to implement a proof of the four-color theorem end-to-end in "Church/Turing-style" fully formal logic. Gonthier's implementation improved on the 1976 proof by Kenneth Appel and Wolfgang Haken that relied in part on "von Neumann–style" Fortran programs to calculate reducibility and in part on "Pythagoras-style" traditional mathematics. (In 1976 the reaction of some mathematicians was to distrust those parts of Appel and Haken's proof that were calculated by computer, whereas the reaction of some computer scientists was to distrust the parts that were checked only "by hand.") In the twenty-first century, computer programs that prove mathematical theorems are expected themselves to be formalized within a mechanically checked logical system. Thomas Hales (2005) proved the Kepler conjecture about sphere packing, using computer programs written in Mathematica and C++, about which the referees were "99% certain." In order to reach 100%, Hales's current project (nearly complete) is to formalize this proof in the HOL Light proof assistant.

In Cambridge, Turing (1936) had brilliant, unprecedented ideas about the nature of computation. He was certainly not the first to build an actual computer; there was already work in progress at (for example) the University of Iowa. But when Turing came to Princeton to work with Church, in the orbit of Gödel, Kleene, and von Neumann,[4] among them they founded a field of computer science that is firmly grounded in logic. In some of Turing's other work (1950) he foresees the field (now within computer science) of artificial intelligence. But in his PhD thesis he makes it clear that he looks to a day when, in proving mathematical theorems, "the strain put on the intuition should be a minimum" (manuscript, page 83). That is, to the extent possible, every step in a proof should be mechanically checkable. We all know the Church-Turing thesis: that no realizable computer will be able to compute more functions than λ-calculus or a Turing machine. But in reading Turing's "Systems of Logic …" we can see quite clearly another kind of Church-Turing thesis, that came

4 Gödel was away from Princeton during Turing's time here, and Kleene had already finished his PhD and left; but clearly they had an enormous influence on Turing's PhD thesis. Turing worked with von Neumann during his time at Princeton, but on other kinds of mathematics than logic and computation.

Copyrighted Material

half a century later as a consequence of their work: mathematical reasoning *can* be done, and often *should* be done, in mechanizable formal logic.

BIBLIOGRAPHY

Church, Alonzo (1936), An unsolvable problem of elementary number theory, *American Journal of Mathematics* vol. 58, pp. 345–363.

Church, Alonzo (1940), A formulation of the simple theory of types, *Journal of Symbolic Logic* vol. 5, pp. 56–68.

Cook, Stephen A. (1971), The complexity of theorem-proving procedures, in *Proceedings 3rd ACM Symposium on the Theory of Computing*, pp. 151–158.

Davis, Martin, and Hilary Putnam (1960), A computing procedure for quantification theory, *Journal of the ACM* vol. 7, number 3, pp. 201–215.

Feferman, Solomon (1988), Turing in the land of O(z), in R. Herken, ed., *The Universal Turing Machine: a Half-Century Survey*, Oxford University Press.

Feferman, Solomon (1992), Turing's "oracle": From absolute to relative computability—and back, in J. Echeverria, et al., eds., *The Space of Mathematics,* Walter de Gruyter, pp. 314–348 .

Gonthier, Georges (2008), Formal proof—the four-color theorem, *Notices of the AMS* vol. 55, number 11, pp. 1382–1393.

Gordon, Michael J., Robin Milner, and Christopher P. Wadsworth (1979), *Edinburgh LCF: a Mechanised Logic of Computation,* Springer-Verlag.

Hales, Thomas C. (2005), A proof of the Kepler conjecture, *Annals of Mathematics* vol. 162, pp. 1065–1185.

Halpern, Joseph Y., Robert Harper, Neil Immerman, Phokion G. Kolaitis, Moshe Y. Vardi, and Victor Vianu (2001), On the unusual effectiveness of logic in computer science, *Bulletin of Symbolic Logic* vol. 7, number 2, pp. 213–236.

Hodges, Andrew (1983), *Alan Turing: the Enigma,* Burnett Books. (New American edition, Princeton University Press 2012.)

Kleene, S. C. (1936), λ-definability and recursiveness, *Duke Mathematical Journal* vol. 2, pp. 340–353.

McCarthy, John (1960), Recursive functions of symbolic expressions and their computation by machine, Part I, *Communications of the ACM* vol. 3, number 4, pp. 184–195.

Perlis, A. J., and K. Samelson (1958), Preliminary report: international algebraic language, *Communications of the ACM* vol. 1, number 12, pp. 8–22.

Rice, H. G. (1953), Classes of recursively enumerable sets and their decision problems, *Transactions of the American Mathematical Society* vol. 74, pp. 358–366.

Copyrighted Material

Soare, Robert I. (1996), Computability and recursion, *Bulletin of Symbolic Logic* vol. 2, pp. 284–321.

Soare, Robert I. (2009), Turing oracle machines, online computing, and three displacements in computability theory, *Annals of Pure and Applied Logic*, vol. 160, issue 3, pp. 368–399.

Turing, A. M. (1936), On computable numbers, with an application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society*, ser. 2, vol. 42, pp. 230–265.

Turing, A. M. (1937), Computability and λ-definability, *Journal of Symbolic Logic* vol. 2, number 4, pp. 153–163.

Turing. A. M. (1950), Computing machinery and intelligence, *Mind* vol. 59, pp. 433–460.

Wigner, Eugene P. (1960), The unreasonable effectiveness of mathematics in the natural sciences, *Communications on Pure and Applied Mathematics* vol. 13, number 1, pp. 1–14.

For more information on Alan Turing's SYSTEMS OF LOGIC: THE PRINCETON THESIS, click here: http://press.princeton.edu/titles/9780.html