Inaugural lecture at Collège de France

Software, between mind and matter

Xavier Leroy

November 15, 2018

Foreword

I delivered my inaugural lecture at Collège de France on November 15, 2018. The text of the lecture, in French, was then published as a book by Fayard [22], and will also be made freely available in the near future.

This document is an English translation of my manuscript for the inaugural lecture. I translated myself; proofreading by native English speakers is welcome.

Xavier Leroy
November 10, 2019

Software, between mind and matter

Mr Administrator,
Dear colleagues,
Dear friends,
Ladies and gentlemen,

On the Web site of the France Culture radio station, we can read the following:

> *Autrefois nommée «harangue», la leçon inaugurale prononcée par chaque professeur élu au Collège de France est à la fois la description de l'état d'une discipline et la présentation d'un programme de recherche. Bien que le nouveau titulaire soit aguerri, cet exercice demeure une épreuve formidable.* [25]
>
> Previously called "harangue", the inaugural lecture delivered by every professor newly elected at Collège de France is both a description of the state of a discipline and the presentation of a research program. Even though the new incumbent is experienced, this exercise remains a formidable ordeal.

As I begin this inaugural lecture — or shall I say this harangue? — "formidable ordeal" feels about right, but "experienced" feels exaggerated: "deeply moved" would better describe my current state of mind. Emotion at the honor you have conferred upon me, dear colleagues, and the trust you have placed in me by receiving me here. Emotion, as well, to see my discipline, computer science, thus recognized and its place at Collège de France thus confirmed.

Computing, as a practice and as a science, has long been ignored by the French academic system [27]. It was not until the mid-1960s that computing appeared in the French higher education and research system; not until circa 1980 that the main French scientific universities opened computer science departments; not until 1997 that a computer scientist, Gilles Kahn, was elected member of the Academy of Sciences; not until 2000 that CNRS created a department dedicated exclusively to information and communication sciences; and not until 2007 that computer science entered Collège de France, the last step towards academic respectability.

Gérard Berry, exceptional educator and indefatigable ambassador of our field, introduced computer science to Collège de France, first on the annual chair of technological innovation in 2007, then in 2009 as first guest of the annual chair of informatics and numerical sciences, created jointly with Inria, and finally in 2012 as incumbent of the first permanent chair in computer science, titled "algorithms, machines and languages". Bourdieu described Collège de France as "*lieu de sacralisation des hérétiques*" ("a place where heretics get sanctified"). He did not mention that several attempts can be required before reaching the sacred.

This chair of software sciences that you are entrusting me with is, therefore, the second permanent chair in computer science in the history of Collège de France. It is a confirmation that this field of study fully belongs in this institution. It is also a strong sign of recognition addressed to all computer programmers — academics, industry professionals, free software enthusiasts, amateurs — who create software and work relentlessly to make it better. (And there are many of them in the audience tonight.) On behalf of this community, I thank the assembly of the pro-

fessors of Collège de France for having created this chair, and Pierre-Louis Lions for having promoted its project.

A brief history of software

The first *programmable* mechanical devices appear in Europe during the 18th century: street organs and Jacquard-style weaving looms. It is the presence or absence of holes on a punched card that determines the musical tune or the visual pattern that are produced. One machine — the *hardware*, as we call it today — can produce an infinity of visual or audible results just by changing the information presented on the punched card — the *software*. The only limitations are the length of the punched card and the imagination of those who prepare it: the *programmers*.

That was a strong idea, which could have been food for thought. Yet, it went mostly unnoticed outside of weaving plants and fairgrounds for two centuries. The wave of automation that swept through industry since the 19th century then through daily life after 1950 was based on machines and equipment that were mechanical at first, then electrical, then electronic, but not programmable at all and built to perform one and only one action.

That was the case of numerical computation, even though it is identified with the modern computer in popular culture. Programmable pocket calculators equip high-school students since the 1980s; yet, nothing can be programmed in the calculating equipment widely used well after World War II: from Pascal's arithmetical machine (1645) to the Curta pocket mechanical calculator dear to Gérard Berry; from the Hollerith tabulating machines that enabled the wide-scale censuses of the 20th century

to the cash registers of my childhood years; from the artillery ballistic computers to the fly-by-wire systems of the first supersonic airplanes [12]. Even the "Bombe" computer that enabled British intelligence to break the German Enigma cipher during World War II was not programmable and had to be re-cabled often. This might come as a surprise since the architect of the "Bombe", Alan Turing, had, a few years earlier, developed the theoretical foundations for programmable computers — the famous universal machine that we will discuss again soon. But there was a war to win against Nazi Germany, and no time to study computer programming.

One exception: in 1834, Charles Babbage, a British mathematician, designed an *analytical engine*, where a Jacquard-style punched card controls the operation of a sophisticated mechanical calculator, the *difference engine* [12]. Despite generous funding from the British government, Babbage's analytical engine could never be fabricated because it was too complex for the mechanical engineering of the time. However, his friend Ada Byron, countess of Lovelace, wrote, on paper, a few programs for the analytical engine, including a program that computes Bernouilli numbers, making her the first computer programmer in history [24]. Did Babbage invent the modern programmable computer? Opinions differ. However, he is clearly a pioneer of the modern research project. It is all there: funding by a governmental agency, excessively ambitious objectives that could not be realized, and a side result (the first numerical calculation program) that turns out to be a major scientific result — said result being obtained by a woman who was not even funded by the project!

It is only at the end of Word War II that the concept of the universal programmable computer becomes widely accepted. The seminal work of Eckert and Mauchly (1943) and of von Neumann (1945) establish the architecture of modern computers: an arithmetic and logical unit communicating with a memory containing both the program that drives computation steps and the data they process. Several prototypes are developed in academic laboratories at the end of the 1940s (ENIAC, Manchester Mark 1, EDSAC, EDVAC, ...). The first electronic programmable calculators that we call computers today become commercially available from 1952 (Ferranti Mark 1, Univac 1, IBM 701, ...) [12].

The rest of the story is well known. Computers have spread from computing centers to factories, offices, homes, payment methods, phones, cars, medical equipment, cameras, television sets, home appliances, all the way to light bulbs, which are now "connected" and claim to be "intelligent". Computers take an ever increasing part in everyday life and in public life, from entertainment to the management of critical infrastructures and the preservation of human lives.

This explosive growth of computing owns much to tremendous advances in micro-electronics, resulting in mass production of computers and systems on chip ever more powerful at constant price, but also to the amazing flexibility of software that runs on these electronic systems. Hardware becomes a *tabula rasa*, a blank slate, infinitely reprogrammable. For example, the Voyager space probes were remotely reprogrammed many times during the 40 years of their journey through the Solar system. Free of almost all physical constraints, software can achieve an unbelievable level of complexity. A Web browser consists of about 10 millions line of code; the embedded software on board a mod-

ern car, 100 millions; Google's whole code base, about 2 billions. That an assembly of 2 billion different components works more or less is unprecedented in the history of technology. Equally unprecedented is the great vulnerability of software to design and programming errors — the infamous *bugs* — and to security attacks and malicious use.

I could talk at length about the feats and the misery of modern computing, dazzle you with figures, frighten you with risks. Instead, let me go back to the fundamental concepts of computing and the history of their birth. These concepts are rooted not in numerical computing and analysis, but in a completely different branch of mathematics, bordering with philosophy, namely: logic.

Logical foundations

Many times in the past, mathematicians and philosophers turned to computation for a source of unquestionable and universally-accessible truths [8]. As early as 1670, Leibniz set out to represent philosophical concepts by mathematical symbols, and to identify the symbolic calculation rules that support reasoning over these concepts. Thanks to this *calculus ratiocinator*, as he called it, philosophical controversies could be solved just by calculation:

> *Quando orientur controversiae, non magis disputatione opus erit inter duos philosophus, quam inter duos computistas. Sufficiet enim calamos in manus sumere sedereque ad abacos, et sibi mutuo (accito si placet amico) dicere: calculemus.* [20]

> If controversies were to arise, there would be no more need of disputation between two philosophers than between two calculators. For it would suffice for them to take their pencils in their hands and to sit down at the abacus, and say to each other (and if they so wish also to a friend called to help): let us calculate.

It is a long journey from this dream of Leibniz to modern mathematical logic [7]. However, this imperative, *calculemus*, remained as a rallying cry among computer scientists: in particular, it is the title of a series of conferences on symbolic computation. Let us compute, brothers and sisters! it builds truth!

During the second half of the 19$^{\text{th}}$ century, logic progressed tremendously, with the formalization of propositional logic (Boole, 1854) then of predicate logic (Frege, 1879), and the birth of set theory (Cantor, 1875–1884) [7, ch. 2–4]. Around 1900 it became conceivable to ground all of mathematics on a small basis of a formal logic. However, this beautiful project was marred by logical paradoxes. A famous example is Russell's paradox (1903): if we can define $A = \{x \mid x \notin x\}$, "the set of all sets that do not belong to themselves", then A belongs to A, but at the same time A does not belong to A. This internal contradiction makes naive set theory *inconsistent* and therefore unusable as a mathematical logic.

Could it be that the fine edifice of mathematics, which we often picture like the Eiffel tower, rising towards the sky while grounded on solid foundations, actually resembles the tower of Pisa, leaning dangerously because of weak foundations? This is called the foundational crisis of mathematics, and it preoccupied some of the greatest mathematicians and philosophers of the

early 20th century [8]. In 1900, David Hilbert, in his list of 23 major unsolved mathematical problems, mentioned the problem of proving the consistency of arithmetic – an important fragment of mathematics, but a fragment nonetheless. Twenty years later, he formulated what we now call Hilbert's program. It consists in formalizing arithmetic as a deductive system[1] and proving that this system satisfies three essential properties:

- *Consistency:* there are no propositions P such that P and its negation not-P can both be deduced. (The logic is free of paradoxes.)
- *Completeness:* for every proposition P we can deduce one of P or not-P. (The logic cannot say "I don't know".)
- *Decidability* (*Entscheidungsproblem*, "decision problem"): there exists a systematic process — an algorithm, as we say today — which, given a proposition P, decides whether it can be deduced or not.

The decidability requirement shows that Hilbert, like Leibniz with the *calculus ratiocinator*, found it very important to be able to compute the truth value of a logical proposition.

Like a Collège de France professor who is interviewed on France Culture, Hilbert popularized this program through a famous speech broadcast on German radio in 1930, which he concluded by stating *Wir müssen wissen; wir werden wissen*: "we must know; we will know" [7, ch. 5]. Shortly thereafter, we came to know… that Hilbert's program is impossible. In 1931, Kurt Gödel published his famous first incompleteness theorem [13],

[1] Comprised of axioms (such as "$n + 0 = n$ for all n") and of deduction rules (such as *modus ponens*, "from $P \Rightarrow Q$ and P we can deduce Q").

which shows that any consistent axiomatization of arithmetic contains a statement P such that neither P nor not-P can be deduced. In 1936, Alonzo Church and Alan Turing proved, independently and via different approaches, the first undecidability results (of the halting problem), from which it follows that the *Entscheidungsproblem* cannot be solved by an algorithm [3, 34].

This is the end of Hilbert's program, but also the beginning of a new knowledge. Fundamental computer science was born out of this failure of Hilbert's program like a wild flower growing on the ruins of a collapsed temple. To prove his incompleteness result, Gödel demonstrated how to represent any logical formula by a natural number. Today we would use a sequence of bits (zeros and ones), and a more compact encoding than Gödel's, but the key idea is there: any piece of information — number, text, sound, picture, video, logical formula, computer program, etc — can be coded by a sequence of bits, then transmitted, or stored, or processed by a computer. To prove their undecidability results, Church and Turing characterized precisely what an algorithm is, thus giving birth to computability theory, each in his own style. Turing formalized a "universal machine", an imaginary robot that moves a tape and reads and writes symbols on this tape, capturing the essence of the modern computer: the programmable calculator with stored program. Church developed his "lambda calculus", an algebraic notation centered on the notion of function, which is the ancestor of modern programming languages.

This birth of computability theory is such an important moment in the history of computing that it deserves to be explained with a cooking analogy. Algorithms are often compared to cooking recipes. To explain an algorithm that solves a given problem,

as to communicate a recipe for a given dish, natural language suffices: there is no need to formulate the recipe in mathematical language. This is no longer the case if we need to reason about all possible recipes and all dishes they produce. Consider — with apologies to Hilbert — the recipe problem (*Rezeptproblem*):

> Can any food be produced by a recipe?

To answer in the negative, we need to identify a food that is not cookable, such as ambrosia in Greek mythology, then prove that no recipe can produce this food. To this end, we must have a mathematically-precise definition of what constitutes a recipe. (For example, "knock on Zeus' door and ask him for leftover ambrosia" is not a recipe.) Thus, we are led to develop a *cookability theory* that is more general and perhaps more interesting than the *Rezeptproblem* we started with.

Mutatis mutandis and proportionally speaking, Church and Turing followed a similar approach to answer the *Entscheidungsproblem* in the negative. Moreover, the two models of computability they proposed, while coming from different horizons, are equivalent, meaning that any one of the models can simulate the other, and both are equivalent to a third model inspired by mathematics, the μ-recursive functions studied by Kleene around the same time. A machine (Turing's universal machine), a language (Church's lambda-calculus), and a class of mathematical functions (the μ-recursive functions) completely agree on which functions are computable and which problems are decidable. This is the birth of a universal notion of computability, called Turing completeness. Today, we know hundreds of models of computation, from mathematical games to bio-inspired models to quantum computing; all these models compute exactly the same functions as a Turing machine.

Towards efficiency and effectiveness

Something is still missing before Turing's dreams and Church's lambda-obsessions lead up to modern computing: a quantitative dimension, completely absent from computability theory. For instance, a problem can be decidable just because the solution space is finite: it "suffices" to test all 2^N possible solutions, even though it would exhaust all the energy of the Sun if N reaches about 200 [23]. Likewise, a model of computability such as Conway's game of life can be Turing-complete yet totally unsuitable for programming.

The last step that leads to modern computer science is precisely to account for the *efficiency* of algorithms and the *effectiveness* of programming these algorithms. On one side, a science of algorithms emerges: how to design efficient algorithms and characterize mathematically the time, memory space, or energy they consume. On the other side, the implementation of these algorithms in software reveals new needs: expressive programming languages; precise semantics for these languages; interpretation and compilation techniques to execute these languages on hardware; verification methods to rule out programming errors — the dreaded *bug*. This knowledge, half empirical, half mathematical, delineates the long path leading from the abstract specification of a software system to its effective implementation. This is the core of *software sciences*, the field of study that I will teach and research at Collège de France.

I will not discuss algorithms further in this lecture, and will not discuss them much in my first courses, because the science of algorithms has been brilliantly exposed already by Gérard Berry and by several incumbents of the annual chair in informatics

and numerical sciences: Bernard Chazelle in 2012, Jean-Daniel Boissonat in 2016, Claire Mathieu in 2017, and Rachid Guerraoui in 2018. These professors communicated the richness and the diversity of this area, from probabilistic algorithms to computational geometry, from approximation algorithms to distributed computing. Instead, I would rather talk about the journey from an abstract algorithm to its concrete execution by the computer, with special emphasis on programming languages and software verification.

Programming languages

In movies, computer experts glance at screens full of fast-scrolling zeros and ones and immediately spot the virus that the bad guys are trying to inject. Reality begs to differ. The programming language made of zeros and ones does exist: it is called *machine code* and it can be executed directly by the electronic circuits of the computer. However, machine code is completely unsuitable for writing programs, because it is illegible and lacks structure. Since the dawn of computers, programmers have invented many higher-level programming languages, clearer, more expressive, more intuitive than machine code, so as to facilitate writing, reviewing, maintaining and evolving programs. In parallel, they have also developed the programming tools — interpreters, compilers, assemblers, linkers — that execute programs written in these new programming languages, often by translation into machine code.

In 1949 — the classic Antiquity of computing history —, *assembly languages* appear. While remaining close to machine code, these languages replace bits by text: mnemonic words and ab-

breviations represent machine instructions (`add` for addition, `cmp` for comparison, etc); labels give names to program points and memory locations; comments, written in natural language and ignored during execution, help documenting the program and explaining what it does. It becomes clear that a program is intended not only to be executed by machines, but also, and as importantly, to be read, studied and modified by humans.

The Renaissance of programming languages starts in 1957 with Fortran, the first programming language that supports arithmetic expressions close to familiar mathematical notation. For instance, the solutions to the quadratic equation $Ax^2 + Bx + C = 0$ are clearly expressed in Fortran:

```
D = SQRT(B*B - 4*A*C)
X1 = (-B + D) / (2*A)
X2 = (-B - D) / (2*A)
```

In assembly language, a dozen instructions would be needed, at least one per basic arithmetic operation. This is the beginning of a humanistic approach where the programming language must clearly express the main concepts of its application area. Thus, Fortran, dedicated to numerical computing, emphasizes algebraic notations and arrays and matrices. Likewise, in 1959, Cobol is designed for business applications and introduces the concepts of records and structured files, the ancestors of modern databases.

The age of Enlightenment starts in 1960 with Algol and Lisp. The new priority for programming languages is to express the structure of programs and to make it easy to write, reuse and combine program fragments. Algol and Lisp are centered on the twin notions of *procedures* and *functions*. Freely reusable and

equipped with a well-defined interface, procedures and functions are the building blocks of software. *Recursion* (the ability for a function to call itself on sub-problems) opens a new way to design and express algorithms, beyond the usual iteration schemes. Lisp opens another new direction: symbolic computation. A Lisp program manipulates mathematical expressions, logical formulas, or other programs as easily as a FORTRAN program manipulates arrays of numbers.

The industrial revolution hits the software world around 1965. Software is less and less handcrafted by end-users, but increasingly developed by specialized firms. A primary objective is to coordinate the work of large teams of average-skilled programmers, so as to produce programs whose size and complexity exceed the abilities of a single highly-skilled programmer. The programming languages of this time, such as PL/I and C, are highly pragmatic and care less about elegance than about efficient utilization of hardware resources. The need for training many new programmers gives birth to languages dedicated to teaching computer programming, such as Basic, which make it possible to learn quickly how to program badly.

In parallel, the years 1970–1985 see the emergence of a computer counter-culture. Computer programming can liberate itself from the mainstream software industry. Rejecting the domination of the so-called imperative approach to programming, other programming paradigms emerge and materialize in radically original programming languages: object-oriented programming (Simula, Smalltalk), logic programming (Prolog), functional programming (Scheme, ML, Hope, FP), synchronous programming (Esterel, Lustre), …

During the 1980s and 1990s, counter-culture is largely absorbed by the mainstream. Hippies join advertising firms and object-oriented programming becomes the new industry standard, but through languages (C++, Java) that distort the original ideas. Functional programming spreads and becomes more respectable through a marriage of convenience with imperative programming. The OCaml language, to which I contributed much, is the result of such an alliance between the functional, imperative, and object-oriented approaches.

In the same time frame, it becomes fashionable to proclaim the end of programming and to question the relevance of programming languages. Soon we will no longer need to program: instead, we will assemble software components. Granted, it is no longer expected of every programmer to write their own sort routine: better reuse one from a library. However, assembling software components is still programming, in a different way, at a higher level, with new problems that arise. What is the interface of a component? how to reuse it correctly? what guarantees does it provide? So many questions that a good programming language can help to answer, through linguistic mechanism that support large-scale programming: classes and packages (Java, C#, …); modules and functors (Standard ML, OCaml); contracts (Eiffel, Racket, …); etc.

Today, we hear about another end of programming: thanks to artificial intelligence and its machine learning techniques, software is no longer written, but largely learned from examples. Can a language-based approach contribute to this "new frontier" in computing? Recent work on probabilistic programming clearly goes in this direction [35].

In parallel with this evolution of ideas and concept, the formal understanding of programming languages — their *syntax* and their *semantics* — progressed tremendously, so much so that, today, we can describe and define a programming language with mathematical precision. The 1960s saw the emergence of grammatical frameworks, often inspired by general linguistics, that can not only describe the syntax of a language but also generate automatically the corresponding *parser* (syntax analyzer).

Semantics resisted formalization longer. Programming constructs such as `x = x + 1` (meaning "increment variable `x`") look mathematically absurd at first sight: how could it be that `x` is equal to `x + 1`? We need to distance ourselves from this strange syntax and introduce the notion of a program state — a mapping from variables to their current values — to finally explain this assignment as a mathematical transformation from the state "before" executing the assignment to the state "after". Equally clever ideas are required to account for other programming constructs found in many languages. Today, we master many approaches to assigning a mathematically-precise meaning to a program: denotational semantics, where a program is interpreted as an element of a mathematical structure (Scott domains, two-player games, Church's lambda-calculus, etc); operational semantics, describing the successive steps in executing the program; and axiomatic semantics, characterizing the logical assertions satisfied by all executions of the program [28].

It can come as a surprise that we have formal syntaxes and semantics for widely-used programming languages such as C. If the behavior of a C program can be known with mathematical precision, how comes that this behavior is so often erratic, including

bugs, crashes, and security holes? This is the central question of formal software verification, which we now introduce on a simple instance: the type-checking of programs.

Type checking

In daily life, we do not compare apples and oranges, nor do we add cabbages and carrots. In physics, dimensional homogeneity prohibits absurd combinations: each quantity has a dimension (duration, distance, mass, etc); an equation that equates or adds two quantities having different dimensions is physically absurd and always wrong.

Generalizing this folk wisdom and this physical wisdom, *type-checking* a program consists in grouping data by *types* (such as integers, character strings, arrays, functions, etc) and checking that the program manipulates the data in accordance with their types. For example, the expression `"hello"(42)`, which applies a character string as if it were a function, is ill-typed. This simple verification, which can be performed *statically*, before the program executes, detects many programming errors and provides basic guarantees about software reliability.

The benefits of type-checking goes further. It guides the design and structuring of programs: programmers are encouraged to declare the types of the data structures and of the interfaces for software component, so that the consistency of these declarations can be verified automatically by a type-checking algorithm. One notch above, types also influence the design of programming languages and their comparative analysis [31]. Which language features lend themselves to effective type-checking? How

can we give precise types to the features we care about? These are questions that shape a programming language. For example, the design of the Rust language was guided by the need to type mutable data structures with high precision.

Even deeper, types reveal a connection between programming languages and mathematical logics: the Curry-Howard correspondence, where types are viewed as logical propositions and programs as constructive proofs. This correspondence between proving and programming was first observed on simple instances by Curry in 1958, then by Howard in 1969. The result looked so insignificant that Curry mentioned it in 4 pages of one of his books [6], and Howard did not submit for publication, circulating photocopies of his manuscript instead [15]. Rarely photocopies have had such an impact: the Curry-Howard correspondence started to resonate with the renewal of logic and the explosion of computer science of the 1970s, then established itself in the 1980s as a deep structural connection between languages and logics, between programming and proving. Today, it is commonplace to study the logical meaning of a programming language feature, or the computational content of a mathematical theorem — that is, which algorithms hide inside its proofs. My lectures for the year 2018–2019, titled "Programming = proving? The Curry-Howard correspondence today", discuss this ferment of ideas, at the border between logic and computer science.

In the end, this simple idea of attaching types to data and checking type consistency in programs carried us quite far! Now, let us see how to extend this approach to the verification of other desirable properties of a program, beyond type soundness.

Formal verification

On the one hand, programming languages, tools, and methodologies have progressed immensely. On the other, the complexity of software keeps increasing, and we entrust it with more and more responsibilities. Thus, the central issue with software is no longer *to program*, but *to convince*: convince designers, developers, end-users, regulation or certification authorities, perhaps a court of justice (if something bad happens) that the software is correct and harmless.

To build this conviction, we need to *specify* what the program is supposed to do ("what do we want?"); *verify* that the program satisfies this specification ("did we write the program right?"); and *validate* that the program and its specification match user expectations ("did we write the right program?"). Specifications take on many forms, from English prose to rigorous mathematical definitions through test suites (examples of expected behaviors) and semi-formal notations (diagrams, pseudocode).

The most widely used verification and validation, by far, is testing: execute the program on well-chosen inputs and check the results. In essence, this is an experimental approach, where software is treated like a natural phenomenon and its specification like a theory in need of experimental validation. Empirically, testing can reach high levels of software assurance, provided much effort is invested in constructing the test suite. For critical software systems, where human lives are at stake, the assurance levels are so high that testing becomes prohibitively expensive. Scientifically, no certainty arises from the results of testing, except for a few finite-state systems that can be tested exhaustively. As Dijkstra quipped in 1969,

> Testing shows the presence, not the absence of bugs.

Beyond testing, formal verification techniques establish, by computation and logical deduction, properties that hold of all possible executions of a program [30]. The properties being established range from type soundness through robustness (absence of "crashes" during execution) to full correctness with respect to a mathematical specification. As mentioned previously, a program can be viewed as a mathematical definition through a formal semantics for the language in which the program is written. Formal verification proves properties of the program as if these properties were theorems about this mathematical definition.

The fundamental ideas and formalisms for verification have a long history. As early as 1949, in a short communication at a conference, Turing himself suggested to annotate programs with logical assertions (formulas linking the values of program variables) [26]. This approach was rediscovered by Floyd in 1967 [11], then generalized by Hoare in 1969 [14] as *program logics* that make it possible to reason by deduction on the behavior of a program without having to construct its denotational or operational semantics beforehand. Two verification techniques more specialized than program logics but easier to automate appeared shortly after: abstract interpretation [5] and model checking [9, 32].

In spite of these early conceptual advances, it was not until the 2000s that formal verification made inroads in the critical software industry, especially in the railway sector and the aviation sector. An early success, in 1998, was the verification of the control software for the driverless Paris métro line 14. Why such

a delay? The formal verification of a realistic software system demands enormous amounts of deductive reasoning and calculations, well beyond human capabilities. To overcome this limitation, it was necessary to develop formal verification tools that automate these tasks, completely or partially, and to invent new algorithms for efficient verification. This is one of the major achievements in computer science of the last 20 years.

We can choose between several kinds of tools, depending on the desired degree of automation and precision of verification.

- A *static analyzer* automatically infers simple properties of a variable (for example, its variation interval in the case of a numerical variable) or of several variables (for example, a linear inequality). Often, this suffices to show that the program is robust.
- A *model checker* can automatically verify other kinds of properties, expressed in temporal logic, by exploring the states that are reachable at run-time.
- A *program prover*, also called *deductive verifier*, is able to check full correctness of a program with respect to its specification, provided that the program is manually annotated with logical assertions: preconditions (hypotheses on entry to functions and commands), postconditions (guarantees on exit), and invariants. The tool, then, verifies that preconditions logically imply postconditions, and that invariants are preserved, using automated theorem proving.
- A *proof assistant* such as Coq or Isabelle enables us to conduct mathematical proofs that are too difficult to be automated. The proof is constructed interactively by the user, but the tool verifies automatically that the proof is sound and exhaustive.

```
bigint[] firstprimes(int n)
{
    assert (n > 0);
    bigint p[] = new bigint[n];
    p[0] = 2;
    bigint m = 3;
  loop:
    for (int i = 1; i < n; m = m + 2) {
        int j = 0;
        while (j < i ∧ p[j] <= √m ) {
            if (m % p[j] == 0) continue loop;
            j = j + 1;
        }
        p[i] = m; i = i + 1;
    }
    return p;
}
```

Figure 1: Computing the first n prime numbers. From Knuth [19].

An example will illustrate the capabilities of these modern verification tools. The program shown in figure 1 is one of the first examples in Knuth's treatise on algorithms [19, section 1.3.2, program P], expressed in a Java-like language. The function `firstprimes` computes the first n prime numbers. It iterates over odd numbers $m = 3, 5, 7, \ldots$, excluding those that are multiple of a prime number already found and keeping the others.

A static analyzer based on abstract interpretation can infer the inequalities $\texttt{n} > 0$ and $0 \leq \texttt{i} < \texttt{n}$ and $0 \leq \texttt{j} < \texttt{i}$, without any programmer assistance. It follows that the accesses `p[0]`, `p[i]` and `p[j]` are always within the bounds of array `p`. The analyzer,

```
for (int i = 1; i < m; m = m + 2) {
  /*  invariants:
    ∀k, 0 ≤ k < i ⇒ prime(p[k])
    ∀n, 2 ≤ n < m ∧ prime(n) ⇒ ∃k, 0 ≤ k < i ∧ p[k] = n
    ∀j, k, 0 ≤ j < k < i ⇒ p[j] < p[k]
  */
}
```

Figure 2: The invariant for the outer loop of the program from figure 1. It expresses the fact that the sub-array `p`$[0] \ldots$`p`$[$`i`$ - 1]$ contains all prime numbers between 2 and `m`, and only those numbers. Moreover, this sub-array is sorted in strictly increasing order, and therefore contains no duplicates.

therefore, proved the absence of a common programming bug, source of many security holes.

A deductive verification tool can prove many other properties of this code fragment, all the way to *partial correctness*: if function `firstprimes` terminates, it does return as result the sequence of the first n prime numbers, in ascending order. The tool cannot succeed all by itself: the programmer must annotate the code with numerous logical assertions, especially loop invariants, that guide verification (figure 2).

Termination of the program is much harder to verify automatically, because it follows from the fact that there are infinitely many prime numbers: if there were only 10 prime numbers and the function were called with $n = 11$, it would loop forever searching for the 11^{th} prime number. We can add Euclid's theorem "there is no largest prime number" as an axiom, but I am not aware of any automated verification tool able to deduce ter-

mination from this axiom; an interactive proof seems in order, using a proof assistant such as Coq or Isabelle.

Another example where higher mathematics are used is an optimization suggested by Knuth [19, section 1.3.2, exercise 6], consisting in removing the condition `j < i` of the inner loop, on the grounds that the other condition `p[j] <=` $\sqrt{\texttt{m}}$ suffices to stop this loop at the right time. Proving the correctness of this optimization is difficult, because it hinges on a subtle property of the density of prime numbers: Bertrand's postulate, first proved by Chebychev in 1852, showing that for all $n > 1$ the interval $(n, 2n)$ contains at least one prime number. Mechanically proving this theorem and its use to justify the optimized code requires an interactive proof in Coq [33]. However, the Why3 automatic verification tool is able to verify the optimized code when taking Bertrand's postulate as an axiom [10].

Verifying the full correctness of a program can carry us quite far indeed — especially if the program was optimized by Knuth. However, there are no insurmountable obstacles, despite what many programmers feel, let alone fundamental impossibilities, despite what a few academics claimed [1]. Software was never expelled from mathematical paradise. No deity ever decreed "in pain thou shalt bring forth programs! using only tests and a few UML diagrams!".

The main prerequisite to formal verification is the availability of a mathematically-precise specification of the expected properties. We must express the problem as equations before we can solve it! This should come as no surprise to physicists, but is sometimes misunderstood by programmers and often by managers. Some application areas have long known how to put problems into equations: this is the case for control-command laws, which are

central to many critical software systems such as train traffic regulation or fly-by-wire aircraft. Other areas have recently made remarkable progress towards formal specification and verification: cryptographic protocols and applications [4], operating system components [18]. At the other end of the spectrum, machine learning, which plays a central role in contemporary artificial intelligence, is often applied to perception tasks that lack any precise specification. How can we characterize what is "good" speech recognition or "good" image classification more precisely than by a success rate on a given data set? However, in the rare case where a precise specification is available, a deep neural network can be formally verified once the training phase is over, as demonstrated by Katz *et al.* on the ACAS-Xu aerial collision avoidance system [17].

Another obstacle on the path to formally-verified software is the trust we can put in the computer tools that participate in its production and its verification. These tools are programs like any others, potentially buggy. A design or implementation mistake in a static analyzer or another verification tool can cause it to ignore a dangerous execution path or a source of bugs in the program under analysis, and therefore to wrongly conclude that the program is safe. Furthermore, verification tools rarely operate on the machine code that actually runs, but more commonly on the source code, written in a high-level language, or even on an abstract model of the software system. A second, more insidious risk appears here: compilers and automatic code generators, which translate the source code or the abstract model into executable machine code, could make mistakes during translation, producing the wrong executable code from a formally-verified source or model.

Quis custodiet ipsos custodes? If verification and compilation tools watch over software quality, who watches the watchmen? Testing is ineffective because these tools are complex programs that implement subtle algorithms for symbolic computation. However, these tools have fairly simple specifications, expressed in terms of the semantics for the languages involved; this paves the way to formal verification. My recent work and that of many colleagues explores this approach of formally verifying the tools that participate in the verification and the compilation of critical software. Using the Coq assistant, we developed and verified CompCert [21], a realistic compiler for the C programming language. (See figure 3.) CompCert's good performance and its extensive verification are milestones in this research area. Two follow-up projects use similar approaches to verify other tools: Verasco [16], a static analyzer based on abstract interpretation, and Velus [2], a code generator for the Lustre reactive language.

Conclusions

Look at how much progress has been made in the world of software since the inception of the programmable computer! High-level programming languages, compilers, semantics, type systems, formal specifications, program logics, tools that automate formal verification, verification of these tools and of compilers: so many steps taken in the last 60 years that tremendously increase our ability to produce safe and secure software.

Are we about to reach software perfection, this ideal where software behaves exactly as prescribed by its specifications, and where programming becomes invisible? Like most ideals, software perfection grows ever more distant as we progress towards it…

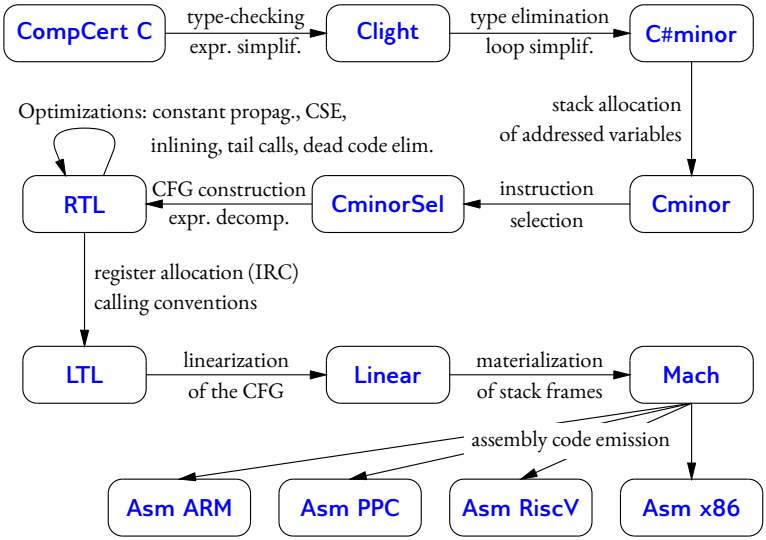

Figure 3: The formally-verified part of the CompCert compiler. It translates the CompCert subset of the C language to assembly language for 4 target architectures, going through 8 intermediate languages. Compilation comprises 16 passes, including optimization passes that eliminate inefficiencies and improve the performance of the generated code. This is a standard architecture for a modern compiler. CompCert's distinguishing feature is that every language (source, intermediate, target) has a formal semantics, and every compilation pass is formally verified to preserve these semantics, using the Coq proof assistant.

- Artificial intelligence succeeds at tasks, especially perception tasks, that were inaccessible to machines until very recently. However, AI produces "black boxes" that are highly vulnerable to bias, to noise, and to malicious use. As far as software reliability is concerned, this sets us back 20 years and calls for new verification and validation methods.
- Hardware turns out to be less infallible than software developers like to assume. Vulnerabilities such as those demonstrated by the Spectre attacks show that hardware has a hard time keeping a secret.
- Formal methods remain difficult to deploy, and keep frightening many programmers. A first step would be to make it easier to write specifications, for example via new, domain-specific specification languages.
- Finally, we need to question the way we teach computer science and mathematics. Our students often confuse the "for all" and "there exists" quantifiers. How can they write specifications if they do not understand basic logic?

Logic! We are back to logic, once more! It is the *leitmotiv* for this lecture: the rise of computer science from the ashes of Hilbert's program; the semantics of programming languages; specifications and program logics; all the way to type systems that bridge programming with proving… All things considered, is software just logic that runs on a computer? Not always (to err is human); not only (algorithmic efficiency matters too); but if only! One of the best things that could happen to software is to be the embodiment of mathematical logic. At a time when we entrust software with more and more responsibilities and we delegate more and more decisions to algorithms, in the naive hope that

they will make fewer mistakes than humans or for more sinister reasons [29], we need mathematical rigor more than ever: to express what a program is supposed to do, to reason upon what it actually does, and to mitigate the risks it poses. Software balances formal rigor with unbridled creativity. It is our duty, as computer scientists and as citizens, to put software at this balance point.

References

[1] A. Asperti, H. Geuvers, and R. Natarajan. Social processes, program verification and all that. *Mathematical Structures in Computer Science*, 19(5):877–896, 2009.

[2] T. Bourke, L. Brun, P. Dagand, X. Leroy, M. Pouzet, and L. Rieg. A formally verified compiler for Lustre. In *PLDI 2017: Proceedings of the 38th Conference on Programming Language Design and Implementation*, pages 586–601. ACM, 2017.

[3] A. Church. A note on the Entscheidungsproblem. *J. Symb. Log.*, 1(1):40–41, 1936.

[4] V. Cortier and S. Kremer. Formal models and techniques for analyzing security protocols: A tutorial. *Foundations and Trends in Programming Languages*, 1(3):151––267, 2014.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL 1977: Conference Record of the Fourth Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[6] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, 1958.

[7] M. Davis. *The Universal Computer: The Road from Leibniz to Turing*. CRC Press, 2012.

[8] G. Dowek. *Computation, Proof, Machine: Mathematics Enters a New Age*. Cambridge University Press, 2015.

[9] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Automata, Languages and Programming, 7th Colloquium*, volume 85 of

Lecture Notes in Computer Science, pages 169–181. Springer, 1980.

[10] J.-C. Filliâtre. Knuth's prime numbers. In *Gallery of verified programs*. http://toccata.lri.fr/gallery/knuth_prime_numbers. Last visited 2019/02/04.

[11] R. W. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia on Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[12] H. H. Goldstine. *The Computer from Pascal to von Neumann*. Princeton University Press, 1980.

[13] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. English translation in [36].

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576––585, 1969.

[15] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, page 479–490. Academic Press, 1980. Facsimile of the 1969 manuscript.

[16] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A formally-verified C static analyzer. In *POPL 2015: Proceedings of the 42nd Symposium on Principles of Programming Languages*, pages 247–259. ACM, 2015.

[17] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient SMT solver for verifying

deep neural networks. In *CAV 2017: Computer Aided Verification, 29th International Conference*, volume 10426 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2017.

[18] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, 2010.

[19] D. E. Knuth. *The Art of Computer Programming: Volume 1: Fundamental Algorithms*. Addison Wesley, July 1997.

[20] G. Leibniz. Nova Methodus pro Maximis et Minimis. *Acta Eruditorum*, Oct. 1684.

[21] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[22] X. Leroy. *Le logiciel, entre l'esprit et la matière*, volume 284 of *Leçons inaugurales du Collège de France*. Fayard, Apr. 2019.

[23] I. L. Markov. Limits on fundamental limits to computation. *Nature*, 512:147–154, Aug. 2014.

[24] L. F. Menabrea and Ada Augusta, Countess of Lovelace. Sketch of the analytical engine invented by Charles Babbage. http://www.fourmilab.ch/babbage/sketch.html, 1842.

[25] M. Moneghetti. Collège de France: 40 leçons inaugurales. https://www.franceculture.fr/emissions/college-de-france-40-lecons-inaugurales. Last visited 2019/02/04.

[26] L. Morris and C. B. Jones. An early program proof by Alan Turing. *Ann. Hist. Computing*, 6(2):129–143, 1984.

[27] P. Mounier-Kuhn. *L'informatique en France de la seconde guerre mondiale au Plan Calcul: L'émergence d'une science*. P. U. Paris-Sorbonne, Mar. 2010.

[28] H. R. Nielson and F. Nielson. *Semantics with Applications: An Appetizer*. Springer, 2007.

[29] C. O'Neil. *Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy*. Crown, 2016.

[30] D. A. Peled. *Software Reliability Methods*. Texts in Computer Science. Springer, 2001.

[31] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[32] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming, 5th Colloquium*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

[33] L. Théry. Proving pearl: Knuth's algorithm for prime numbers. In *TPHOLs 2003: Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 304–318. Springer, 2003.

[34] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[35] J. van de Meent, B. Paige, H. Yang, and F. Wood. An introduction to probabilistic programming. *Computing Research Repository*, abs/1809.10756, 2018.

[36] J. van Heijenoort. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, 1977.

A computer can perform many different functions just by changing the piece of software it executes. This amazing plasticity of software enabled the digital computer to escape data centers and spread everywhere, from everyday objects to critical infrastructures. Which fundamental concepts underlie this technical feat? How can we master the amazing and often frightening complexity of software? How can we avoid bugs and resist attacks? How can we establish that a piece of software is safe and trustworthy? To these questions, mathematical logic offers answers that enable us to build a scientifically rigorous approach to software.

Xavier Leroy is a computer scientist, specializing in programming languages and tools. He is one of the authors of the OCaml functional programming language and of the CompCert formally-verified compiler. Previously a research scientist at Inria, he was appointed professor at Collège de France on the chair of software sciences in May 2018.