Preface

Logic has been called "the calculus of computer science" [66]. Much as mathematics has proven to be an indispensable tool in the natural sciences [113], logic has proven exceedingly useful in computer science. Its applications include databases; hardware; algorithm correctness and optimization; AI subjects such as planning, constraint solving, and knowledge representation; modeling and verification of digital systems in general; and all facets of programming languages, from parsing to type systems and compilation [44]. Perhaps this should not be surprising in view of the historical kinship between the two fields: Turing machines, the key theoretical innovation that gave birth to computer science, were introduced as a formal analysis of algorithms in order to settle the question of whether the satisfiability problem in predicate logic is mechanically decidable [24].[1]

However, even though the primary focus of this text is computer science, it is worth noting from the outset that logic in general and deductive proof in particular are not just useful in computer science or mathematics. Deductive proof is a tool of universal importance and applicability for a simple reason: It is the most rationally compelling species of argument known to humans, and rational argument is the foundation not just of mathematical disciplines but of all science and engineering. Chemistry and finance may have little subject matter in common, but both are underwritten by the same canons of rationality and follow largely the same approach to inquiry: formulating key concepts, articulating and establishing their properties and interrelationships, using these to issue predictions and explanations, and revising them as needed in the face of new empirical evidence or theoretical insights. All of these activities require the ability to distinguish between good and bad arguments, between sound and erroneous trains of thought. In particular, they require the ability to tell when a proposition is a logical consequence of ("follows from") a given body of information and when it is not. And they require the ability to provide correct arguments capable of demonstrating either of these cases. Logic provides a powerful set of conceptual and methodological resources for doing that.

In particular, logic provides a rich formal language in which we can unambiguously express just about anything we care to express, namely, the language of the first-order predicate calculus, which has become the lingua franca of computer science and related fields such as linguistics. It also provides a mathematically precise semantics for that language, with a rigorous definition of the key notion of logical consequence. And perhaps most importantly, it provides effective, sound, and complete mechanisms for constructing proofs, which can be viewed as arguments given to show that a conclusion follows from some premises. These mechanisms can be implemented on the computer, and are themselves amenable to mathematical analysis. Indeed, in this book almost all of our proofs are written in a computer language, Athena, and are checkable by machine. In what follows,

1 This was David Hilbert's famous *Entscheidungsproblem*.

we briefly explain why we believe proofs to be worth studying in general, and then, more narrowly, why we are advocating a computer-based approach to them.

Why proofs?

What is the value of proof? We have already mentioned a key advantage it confers: an exceptionally strong and domain-independent epistemic warrant. If we manage to prove something, whether it is a mathematical theorem, a prediction of an economics theory, or a statement of correctness for a piece of software, we can rest assured that the conclusion is valid, as long as we accept the premises underlying the proof (these might respectively be a body of mathematical axioms, the basic tenets and working assumptions of an economics theory, and the semantics of the programming language in which the software is written). In other words, if we accept the premises, then we are rationally compelled to accept the conclusion.

Proof is the primary vehicle for knowledge generation in the mathematical sciences, including theoretical computer science. But in computer science, proof has also found a more practical use with a stronger engineering flavor: verification, namely, demonstrating that a particular system (or a component, or an algorithm, or even a design sketch) works as it should, or at least that it has certain desirable properties. Verification is becoming increasingly commonplace and will continue to grow in importance as our lives become more dependent on digital artifacts and as the economic and social cost of software and hardware malfunctions becomes more prohibitive.

Verification has so far focused on automation, in an attempt to minimize the human effort necessary for analyzing large systems. However, due to fundamental theoretical limitations, human guidance will remain necessary in most cases. The modeling and verification of complex systems will continue to require the interaction of humans and machines, with people sketching out the main proof ideas and machines filling in the details. The many examples in this book will teach students how to reason about fundamental computer science structures in a style that will hold them in good stead should they ever undertake any verification projects.

Another reason to study proofs is to increase one's level of mathematical sophistication, both in generating one's own mathematical arguments and in reading those of others. Proof is central in mathematics, and if you understand its foundations you will find it easier to understand (and do!) mathematics. You will have a deeper appreciation of how and why putative mathematical proofs work when they are successful, but you will also be able to better understand what goes wrong when they fail. And this leads to a more general reason to master logic and proofs: to become better able to spot and steer clear of reasoning errors, whether in mathematics or elsewhere.

This is, of course, a key part of critical thinking in general. Now, we all have native logical intuitions, so to a certain extent we can all manage to detect logical pitfalls. Unfortunately, however, intuition can only take us so far. Psychologists have shown that our intuitions are often wrong and susceptible to systematic biases. These can only be rectified by rigorous training in normatively correct modes of reasoning, of the kind that are embodied in formal proof methods. And while detecting reasoning errors and questioning assumptions may be useful in guarding against demagogues and other assorted charlatans, it is even more important in debugging computer systems, especially software systems. Indeed, one of the best ways to analyze and to discover errors in our systems is to attempt to *prove that there are no errors*. The gradual refinement that takes place during that process, the unearthing and explicit formulation of assumptions and their consequences, is invaluable in making us understand how our systems work, and when they fail, how and why they fail. So another benefit of training in logic and proofs is cultivating a habit of correct and careful thinking that should pay handsome dividends in the course of an engineering career.

The mention of understanding in the previous paragraph points to yet another, frequently understated, strength of proofs: their potential as tools for *explanation*. The compelling epistemic warrant given by a deductive proof and a sharpened ability to detect and avoid errors are fine and good, but we would get both of these if we had, hypothetically, access to a benevolent oracle, a black box that could always tell us whether a conclusion follows from some premises. If it turned out, perhaps after a long period of time, that the oracle was never wrong, then we would be justified in accepting its verdicts. But while the oracle might always be correct in giving us "thumbs up" or "thumbs down" for a given question, and while we could rely on its answers as robust indicators of truth, we would still lack *understanding*. Suppose the oracle tells us that P is indeed not equal to NP. That's great, but *why*? That is where proofs again enter the picture. A proof is not only a means of *verifying* a claim. It is also, importantly, a means of *explaining* why a claim holds. Or at least a good proof is.

But if a proof is to serve as an explanation, it must be digestible by humans. It must be readable, it must be properly structured, it must abstract away details when they are not needed, and so on. To a large extent that is the responsibility of the proof writer, but the underlying medium—the *language*—in which the proof is written must be able to support these features. It must allow the expression of proofs in a style that is not far removed from informal mathematical practice. Athena, the proof language that we use, is based on *natural deduction*, a style of proof that was explicitly designed to capture the essential aspects of mathematical reasoning as it has been practiced for thousands of years. In combination with other features, such as abstraction mechanisms and complexity management tools borrowed from modern programming languages, Athena takes us a step closer to proofs that can explain and communicate our reasoning, but that are nevertheless entirely formal and checkable by machine.

Why this textbook?

We were motivated to write this textbook primarily by need. Many students whom we have asked to prove theorems in our courses, even in upper undergraduate—and sometimes graduate—courses, have been unable to write proofs. Some have struggled even to write down anything resembling a proof. Others have written what they thought were proofs but were in fact poor attempts at proof, full of reasoning errors. Many other colleagues we have talked to who teach at other universities have reported similar experiences. It is clear that neither the training nor the practice that students are currently getting in logic and proof methods in their undergraduate courses is anywhere near adequate. A major reason is that proof exercises are difficult to grade by hand, and thus only a few such exercises are assigned when in fact many are needed in order to measure students' understanding and help them improve their skills. This situation sorely needs to be remedied, but the way that logic and proof methods are presented in current computer science or logic textbooks, and the limited number of exercises that can be assigned and graded with human-intensive effort, are inadequate for the task.

The second consideration is opportunity. Due to recent research advances, programs are now available that permit proofs to be expressed in a format that is both human-readable and machine-checkable. These programs have been developed by one of the authors and used extensively in research and teaching by both authors. Although other mechanical proof assistants have been available for many years, most were developed as aids to teaching logic to students majoring in such fields as philosophy, cognitive science, or mathematics. By contrast, the newer programs—especially Athena, the program used in this book—support an approach to logic and proof methods that is much better suited to the needs of computer science students. For example, Athena fully supports proof by induction, especially for properties of data types defined by structural induction. Furthermore, Athena allows writing proof methods that are closely related to parameterized procedures for ordinary computation. This correspondence allows us to point out many analogies between proofs-as-programs and ordinary programs, thus leveraging the natural interest of computer science students in programming and computation in a way that one could not expect of students from other disciplines.

Unable to assume interest and experience in computer programming, authors of textbooks for "Introduction to Logic" and similar courses have had to find examples and exercises in artificial domains such as puzzle solving, which are often seen as insufficiently relevant and motivating by computer science students—or, for that matter, by a broader spectrum of science and engineering students. In this textbook we instead focus on practical computer science proof applications such as algorithm correctness properties, including input-output correctness, termination, and memory safety; algorithm efficiency

(correctness of optimizations); and fundamental data type properties, such as inductive properties of the recursive data structures that are pervasive across all branches of computer science. Furthermore, such applications serve to strongly motivate attention to properties of the fundamental mathematical concepts on which the correctness of important algorithms depends, such as algebraic and relational axioms and theorems.

There are several other textbooks that aim to introduce logic and proofs to a computer science audience, but in almost all cases the proof formalisms they use are unsupported by software, and therefore exercise solutions must be checked by hand. By contrast, all proofs in this textbook are machine-readable, and complete code is always given, so that students can try out the examples on a computer, experiment with them, and get immediate feedback. Student solutions can be mechanically checked for correctness.

A small number of much more recent textbooks do take a computer-based approach to proofs, and their goals are fairly similar to ours, including the goal of teaching proof methods that are fundamental in computer science; the use of a mechanical proof assistant as an integral tool in their course of study; substantial material on pervasive data types such as natural numbers and lists; and many examples and exercises, which are mechanically checkable. But there are also major differences. First, their main application area is programming languages. Our book does cover that topic, but to a somewhat lesser extent, focusing more on algorithms and data structures. To the best of our knowledge, our discussion of abstract algebraic structures and techniques, which leads into our extensive treatment of abstract algorithms and data structures, is unique. Our text also has a much more extensive treatment of equality and order relations (both for natural numbers and integers, and at an abstract level), as well as more extensive coverage of proof methods for sentential and predicate logic, including numerous heuristics for discovering proofs. Finally, material such as automated testing (falsification) of conjectures, as well as other applications of formal methods, such as SAT solving, are also not found in the aforementioned texts.

But the biggest difference lies in the style of proof supported by the respective inference systems. As we have already pointed out, Athena uses a true natural-deduction style of proof (the "Fitch style" of natural deduction), which allows for structured and readable proofs that resemble in certain key respects the informal proofs one encounters in practice. Other proof systems do not place a high premium on making proofs understandable, or even recognizable, and in fact the texts that use them seem to have taken it for granted that formal proof and human understanding are incompatible. We believe that readability is not only compatible with formal proofs, it is in fact *necessary* if formal proofs are to reach their full potential. A while back, in their preface to a classic computer science textbook [1], Abelson and Sussman urged the following: *Programs must be written for people to read, and only incidentally for machines to execute*. Few would debate this maxim. It might not always be attainable, but we should always strive to attain it. It should be no different for proofs.

Intended audience and prerequisites

The book is intended primarily for students majoring in computer science at the upper undergraduate or graduate level, but it will also be of interest to professional programmers and to practitioners of formal methods and researchers in logic-related branches of computer science, either for self-study or as a reference. More detailed suggestions are made at the end of this section. In its examples and exercises, since they are written in a machine-checkable language, it exhibits and requires complete rigor, but no more than what computer science majors and other programming practitioners are accustomed to when using other computer languages.

The only prerequisite is a basic level of programming experience and mathematical knowledge, typical of a second- or third-year computer science major. In many cases, a computer science major will take a discrete mathematics course in the first three to four semesters, and that is desirable before taking a course based on this book but not strictly necessary. The instructor must decide whether to require a discrete math background.

Prior exposure to functional programming would be helpful but not essential. A programming languages concepts or survey course, usually taken later in the curriculum, is often the first place students work with a functional language, but unless a programming languages text uses a functional language such as Scheme, ML, or Haskell throughout, as some do, the amount of time spent on functional language concepts is typically not that great. We believe that functional programming can be introduced along with other concepts, as this book does, and furthermore, that doing so provides more motivation than the typical treatment in a programming languages course. In practice, the main difficulty for students who have been taught to express computations in an imperative style, using loops and assignment statements, is learning to express them using recursion instead. There are many examples of recursive function definitions in the text, and as these can be directly executed, they should provide good practice in learning how to think recursively. But a fundamental claim that is often made about functional programs is that they are easier to reason about, mostly because the evaluation mechanism is based on substitution, reflecting standard mathematical practice; and because structural induction is a straightforward mechanism for reasoning about recursive functions defined on inductive data types. This text provides ample opportunity not only to compute with recursion, but also to reason about it mechanically. In particular, the material highlights the strong connection between structural induction over algebraic data types and recursion, a connection that is central to functional programming and yet is usually glossed over in existing functional programming texts [17].

Here are some more specific capacities in which this textbook could be used in computer science departments at the upper undergraduate or early graduate level:

1. Existing *Software Verification* courses could use the book as their primary or secondary source material, depending on whether they choose to focus on approaches based on

proofs or a combination of model checking and proofs. In addition, faculty who have interests in software verification but have not previously taught a course on it may be inspired by our approach and applications emphasis to introduce a new course using the book. There is more material than could be covered in a single course, but neither of the two main application areas, algorithms (Parts IV and V) and programming languages (Part VI), is dependent on the other. Instructors can thus choose depending on students' and their own background and interests. Not all students will need the depth of coverage of logic fundamentals provided in Chapters 4 and 5, so it should not be necessary to devote much class time to that material. The use of Athena as a high-bandwidth interface to SMT and SAT solvers and other ATPs (along the lines described in the "Automated Theorem Proving" appendix on the book's web site) may be of particular interest to instructors who want to introduce students to such tools.

2. *Theory of Computation* courses could use Parts I, II, and VI as secondary source material in lieu of the more abstract treatments of sentential and predicate logic often found in theory of computation textbooks.
3. *Logic* courses could use Parts I and II as their main source material, concentrating most heavily on Chapters 4 and 5. It may be useful for motivation to quickly survey the material in one or more of the later parts of the book. Related content includes:

   - syntax and semantics of sentential logic;
   - syntax and semantics of (many-sorted) first-order logic;
   - notions of interpretation, satisfiability, tautology, entailment, and monotonicity;
   - introduction and elimination proof rules;
   - heuristics for proof development;
   - equational and implicational chaining;
   - SAT solvers and theorem proving systems.

   Inclusion of the last two topics will help prepare students for writing proofs and using automated proof systems in later courses. Because of its focus on applications, the book does not cover metamathematical results such as completeness, compactness, the Löwenheim-Skolem theorems, Gödel's incompleteness, and so on. (Soundness and completeness results for Athena are stated but not proved here.) All of these are standard and can be found elsewhere.
4. In *Algorithms* courses, the book could be used as a supplementary text, using material selected from Parts I through V, with less depth of coverage of logic fundamentals and proof methods than in software verification or logic courses. Related content includes:

   - data structures: lists, binary search trees, finite sets and relations, Cartesian products, maps, memory range abstractions;

- algorithms: binary search on trees and on memory range abstractions, exponentiation, greatest common divisor (Euclid's), a few generic algorithms from the C++ Standard Template Library;
- programming techniques: recursion, replacing general recursion by tail recursion, expressing complex control structures using mutually recursive procedures, memory range traversal using iterators, behavioral abstractions as in generic programming;
- specification and verification: defining functions axiomatically, writing software specifications within a logic framework and using them together with model checking and proof attempts to detect errors in algorithm implementations, necessary and sufficient conditions on ordering relations for correct searching and sorting.

5. In *Programming Language Concepts* courses, the book could be used as a supplementary text, using material selected from Parts I, II, and VI. Related content includes:

   - Athena's programming language as an example of a higher-order functional language in the tradition of Scheme: strict and lexically scoped, encouraging a programming style based on function calls and recursion, but also offering imperative features (e.g., vectors and updatable memory cells);
   - Athena's deductive language as an example of a special-purpose language that benefits from custom-made syntax and semantics (as opposed to deductive systems that are programmed in a host programming language);
   - Athena's blend of functional and logic programming techniques, particularly the programmable interface to external Prolog systems described in Appendix B;
   - Athena's logic subset as an example of a many-sorted language with Hindley-Milner style parametric polymorphism and automatic sort inference;
   - examples of using infix operator precedence and associativity to reduce notational clutter (especially compared to Lisp-style prefix notation);
   - correctness of a compiler from an interpreter-defined toy language to a stack-based machine language, first without and then with error handling;
   - formal, executable, and mechanically analyzable definition of syntax and semantics of an imperative language, which, although still very simple compared to real programming languages, contains some of the most basic and essential features of imperative languages, providing sufficient context for discussion of many important issues related to programming language syntax and semantics.

   Logic fundamentals and proof methods could be covered in less depth than in software verification or logic courses, except perhaps when the last topic is to be treated in detail.

6. Senior students could use selected portions of the text as the basis of an independent study or capstone experience.

7. An "immigration" course or self-study requirement for new graduate students coming into computer science departments from other majors could use the book as its basis or as one of several assigned texts.

Examples and exercises

More than three hundred exercises are provided, and, in keeping with the book's emphasis, almost all require writing a complete proof or filling in one or more parts of a proof that is only sketched in the text. These exercises are generally designed to continue development of ideas presented in the book's many proof examples. (Any of the proof examples can itself be used to provide additional practice: Just glance at its overall structure, then put it aside and try to reproduce it, or even come up with a different and possibly better proof!) Exercises marked with a star are generally more difficult, and a mark of two stars indicates even greater difficulty.

Learning curve

Studying proofs via a computer language such as Athena does have a drawback: a somewhat steeper learning curve. After all, one must learn a new computer language from scratch. This is true regardless of whether one chooses to use Athena or some other computer-based proof system. However, as we discussed above, we believe that the effort required upfront is a worthwhile investment, well justified by the dividends (getting immediate feedback from the computer, a wealth of problems whose solutions can be checked by machine, capitalizing on the parallels between programs and proof methods, etc.). In the case of Athena, a minimal path to fluency can be carved out as follows: Start with Sections 2.1–2.10 and Section 2.17 from Chapter 2; continue with Sections 4.1–4.11 from Chapter 4 and Sections 5.1–5.4 from Chapter 5, and conclude with Chapters 6 and 7. This material, with assigned exercises included, can be covered in a few weeks and will ensure a basic mastery of Athena as well as a solid command of sentential and first-order proof methods, clearing the way forward to interesting applications.

Finally, it may be asked whether the learning curve can be justified if readers do not go on to use Athena later in their careers. Can the book still be of value? While Athena does have certain advantages that we hope to demonstrate in due course, ultimately it is only used as an instrument for expressing the underlying techniques concretely enough that a computer can check whether they are being used properly. As an analogy, programmers who become exposed to a functional programming language end up acquiring a set of concepts and techniques and perhaps a certain way of approaching problems and a broader view of the landscape of computation. The key benefit they derive is not learning the details

of a particular language, but becoming conceptually richer and enlarging their problem-solving arsenals. The same holds for Athena and proofs.

Support materials

The authors maintain a web site, `http://proofcentral.org`, whose primary role is to provide support materials for the book. It also allows downloading of Athena (for Windows, Mac OS, and Linux) and its libraries. All Athena code that appears in the book in examples and exercises is available in files on the web site, to be used in writing exercise solutions without having to type complete proof developments. In addition, solutions to many of the book's exercises are available on the site.