Texts & Monographs in Symbolic Computation

A Series of the Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria

Wolfgang Schreiner

Thinking Programs

Logical Modeling and Reasoning About Languages, Data, Computations, and Executions

Springer

Wolfgang Schreiner
RISC
Johannes Kepler University
Linz, Austria

ISSN 0943-853X ISSN 2197-8409 (electronic)
Texts & Monographs in Symbolic Computation
ISBN 978-3-030-80506-7 ISBN 978-3-030-80507-4 (eBook)
https://doi.org/10.1007/978-3-030-80507-4

Mathematics Subject Classification: 03B70, 03B10, 03B44, 68N15, 68N20, 68N30, 68Q55, 68Q60, 68Q65

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Meinen Eltern

Foreword

When I started my study of mathematics at the University of Innsbruck in 1960, like most freshmen, I was intimidated and impressed by the apparent intelligence of the professors who gave proofs of abstract knowledge, which was far from the concrete thinking about mathematical objects which we had seen in high-school. However, after some time, in secret, I started to doubt the quality of some of the hand-waving proofs and I wanted to look behind the scene. For this, in parallel and independent of the curriculum, I started to dig through the books on logic I found in the general library of the university. (For some reason, most of them had a yellow cover—the Springer Books on Logic. Subconsciously, this may have been the reason why, many years later, I decided that the cover the Journal of Symbolic Computation should be yellow, when I founded it in 1985.)

From that time on, it became more and more clear to me that logic is the essence of mathematical thinking and, luckily, very soon after my start as a mathematics student I got the chance to become one of the first programmers on the first computer at our university and the computer appeared to me as materialized logic. Since then, for me, mathematics, logic, and computer science was just one field and I am still fascinated and convinced by the repeated algorithmic cycles through object and meta-levels to reach higher and higher states of insight and a more and more efficient grasp of the thinking process. Understanding the spiral of logic for (algorithmic) mathematics and algorithmic mathematics for logic is so important also for steering a clear course in a time of frequent new and fancy catch words that may suggest that logical clarity and brilliance is not any more relevant in a time of intelligent machines.

In 1979, in contrast to the usual analysis/linear algebra approach, I dared to give an introduction to mathematics for first semester computer science students which was nothing else than a practical introduction to predicate logic as a working language and a unified frame for proving and programming. Over the years, many of my students shared this view on the fundamental theoretical and practical importance of logic for mathematics and informatics and did remarkable work developing ideas and tools for supporting this kind of thinking. Wolfgang Schreiner embarked on this type of research, teaching, and software development already in the mid-nineties and, over the years, accumulated enormous know-how and produced impressive and extensive teaching material and software tools for the theoretical foundation and the practical application of logic in mathematics and computer science, notably in mathematics and computer science teaching.

Now, he presents this enormous amount of work in a coherent book. I think there is hardly any other book that combines the foundation of logic, the applications of logic in computer science, and software for logic in an equally rich and comprehensive way. I wish the book a wide distribution. Given the outstanding didactic qualification of Wolfgang Schreiner, I am sure that the book will be extremely helpful for students of mathematics and computer science to get a profound training of the thinking technology that is in the center of the present age and will stay and become even more important in the next turns of the spiral of innovation.

It is also a special pleasure for me that the book appears in the RISC book series on symbolic computation with the Springer Verlag whose yellow books lured me into the field of (algorithmic) logic so many years ago. When I founded the RISC book series in 1993, for some reason, we decided that the cover should be gray. However, the contents of the books in this series were very much yellow all the time. It is great to see that, since Peter Paule took over the editorship and is giving enormous drive to the series, yellow is taking over also on the covers.

Hagenberg, Austria
March 2021

Bruno Buchberger

Preface

Motivation

The purpose of this book is to outline some basic principles that enable developers of computer programs (computer scientists, software engineers, programmers) to more clearly *think* about the artifacts they deal with in their daily work: data types, programming languages, programs written in these languages that compute from given inputs wanted outputs, and programs for continuously executing systems. In practice, thinking about these artifacts is often muddled by not having a suitable mental framework at hand, i.e., a *language* to appropriately express this thinking.

The core message that we want to convey is that clear thinking about programs can be expressed in a single universal language, the formal language of logic. In particular, with the help of logic we can achieve the following goals:

- *Modeling*: we can unambiguously describe the meaning of syntactic entities such as the behavior of computer programs.
- *Specifying*: we can precisely formulate constraints we impose on (the meaning of) these entities such as requirements on program executions.
- *Reasoning*: we can rigorously show that the entities indeed satisfy these constraints, e.g., that programs satisfy their specifications.

However, in order to enable this clear thinking about computer programs, we also need a framework to relate the syntactic artifacts (that have a priori not any content beyond their structure) to their formal meaning (characterized by logical formulas). The description of this relationship can be generally based on three principles:

- A *grammar* that describes the basic structure of syntactic phrases.
- A *type system* that further restricts the phrases to certain well-formed ones.
- A *function* that maps every well-formed phrase to its meaning.

Thus we can give arbitrary syntactic phrases a precise meaning about which we can rigorously reason. In fact, we will use this approach of *denotational semantics* uniformly throughout this book in order to define various formal languages, starting with the language of logic itself and ending with a language of concurrent systems.

Throughout most of this book, we understand by logic the classical first-order variant of predicate logic, short *first-order logic*, the lingua franca of formal modeling and reasoning today. While some aspects of computer programming may practically profit from more general frameworks such as higher-order logic or temporal logic (which we will also discuss in this book), first-order logic is conceptually sufficient for most purposes and in any case provides a solid basis for understanding all kinds of logical extensions.

Apart from its universal elegance and expressiveness, our logical approach to the formal modeling of and reasoning about computer programs has another advantage: due to advances in computational logic (automated theorem proving, satisfiability solving, model checking), nowadays much of this process can be supported by *software*. This book therefore accompanies its theoretical elaborations by practical demonstrations of various systems and tools that are based on respectively make use of the logical underpinnings. We hope that this will convincingly demonstrate also the actual usefulness of the presented approach.

This book has been written with a broad target audience in mind that encompasses students and practitioners in computer science and computer mathematics; it therefore tries to be as self-contained as possible and not assume a particular background in logic or mathematics. However, it focuses on a "logical" perspective to the overall areas of "formal methods" and "formal semantics" which in several aspects differs from other presentations of this topic. To get a more comprehensive picture (especially on alternative approaches), the reader might want to consult additional resources; the book gives various recommendations for further reading.

Content

To introduce the basic themes of this book and demonstrate their ultimate purpose, an introductory section Logic for Programming: A Perspective gives a short historical account on the development of logical modeling and reasoning about computer programs and presents examples of practical applications of these techniques in industrial software development today.

The main contents of this book are then organized into two parts:

Part I: The Foundations. This part introduces the basic language of logic and mathematics that is used throughout the remainder of the book. While an impatient reader (such as the author himself!) may be inclined to skip over this part, we advise to study at first reading at least Chapter 1 "Syntax and Semantics" which introduces the themes that are fundamental to this book:

- context-free grammars (inductive definitions of formal languages as sets of phrases represented by abstract syntax trees),
- type systems (logical inference systems which restrict these languages to certain well-formed phrases),
- semantics functions (inductively defined functions which give these phrases a meaning by mapping them to mathematical objects), and

- the accompanying principle of structural (more general: rule) induction which enables us to reason about these constructions.

The subsequent chapters elaborate these concepts in more detail (the impatient reader may skip over them at first reading and consult them later on demand). Chapter 2 "The Language of Logic" applies above principles to introduce the syntax and semantics of first-order logic, the core language of this book. Chapter 3 "The Art of Reasoning" continues this presentation by introducing the concept of logical proof and by introducing a variant of the sequent calculus as a formal framework for proof construction. Chapter 4 "Building Models" describes how with the language of logic models of reality (theories and data types) can be constructed in which we subsequently operate. Chapter 5 "Recursion" discusses the semantics of various forms of recursion, including inductive and coinductive definitions of functions and relations, using a restricted variant of fixed point theory. The material presented in these chapters (and much more) can be similarly found in various texts on logic and mathematics for computer science, however with non-uniform notions and notations; our goal here is to give a minimal consistent framework as a sufficient and necessary basis of Part II of this book.

Part II: The Higher Planes, this part contains the actual core contents of the book:

- Chapter 6 "Abstract Data Types" discusses the formal specification of abstract data types by logical axioms that the operations on the types must satisfy; the types may consist of finite values characterized by their constructor operations but also of potentially infinite values characterized by their observer operations. For this purpose, we gradually introduce a formal type specification language with a static type system and give it a semantics as models of first-order formulas; these models may be restricted to a particular class of candidates by special kinds of specifications (generated/free, cogenerated/cofree). We also discuss specifying in the large by various principles to compose smaller specifications to bigger ones. The chapter does not only describe the modeling of types but also the basic techniques for reasoning about them; it also discusses the refinement of more abstract types to more concrete ones.
- Chapter 7 "Programming Languages" discusses the formal semantics of programming languages. For this we extend the previously introduced type specification language to an imperative (command-based) programming language whose semantics we describe by two approaches. In denotational semantics, we first map commands to partial functions on program states; these functions are defined by logical terms. Later we generalize this classical functional style to a non-classical but much more flexible relational style where commands are mapped to state relations defined by logical formulas. In operational semantics, we give programs a semantics by mapping them to transition relations defined by logical inference systems; we then show the essential equivalence of denotational and operational semantics. We apply these techniques to model the translation of the command language to a low-level machine language and prove the correctness of the translation. Finally, we extend the command language by the abstraction mechanism of procedures and model their semantics.

- Chapter 8 “Computer Programs” discusses the formal verification of programs by various closely related calculi; the chapter thus lifts the level of reasoning from the previously discussed layer of programming languages to that of programs written in these languages. After discussing the formal specification of computational problems as the basis of program verification, we present the Hoare calculus and prove its soundness with respect to the semantics of the language. We continue with Dijkstra’s predicate transformer calculus that maps commands to functions on formulas over states; further on we complement this approach by presenting a relational calculus that maps programs to formulas over state pairs. A good part of the chapter is dedicated to the pragmatics of verifying the partial correctness and termination of programs, which requires the human to devise adequate loop invariants and termination measures; here we give several concrete verification examples. Finally we discuss the abstract concept of command refinement from which we derive the concrete principles of modular reasoning about the correctness of procedure-based programs.
- Chapter 9 “Concurrent Systems” discusses the formal modeling of and reasoning about systems exhibiting nondeterministic behavior (concurrent/reactive systems). For this we extend the previously introduced command language to a language of shared systems where concurrent activities interact via a common state as well as to a language of distributed systems whose components interact by exchanging messages. We give these languages a semantics by mapping them to labeled transition systems; these are described by logical formulas that denote initial state conditions and transition relations. For specifying properties of such systems we extend first-order logic to linear temporal logic whose formulas are interpreted over system runs. The proof-based verification of such properties requires the human to devise adequate system invariants; we discuss the verification of such invariants expressing safety properties of the system and also the verification of a particular class of liveness properties which ensure the progress of the system execution. Finally we investigate the refinement of more abstract systems (models) to more concrete systems (implementations).

Altogether these chapters thus present the syntax and semantics of a language that encompasses abstract type declarations, imperative programs whose behaviors are specified by formulas in first-order logic, and concurrent systems whose behaviors are specified in linear temporal logic; they discuss corresponding calculi for the verification of the programs with respect to specifications and for the refinement of more abstract programs to more concrete ones.

Software

Each chapter is accompanied by a section that illustrates the practical relevance of the presented theoretical material by some software system or programming language that is based on the presented concepts respectively makes use of them:

- The functional programming language OCaml.
- The proof assistant RISC ProofNavigator.
- The interactive theorem prover Isabelle/HOL.
- The algebraic specification language CafeOBJ.
- The common algebraic specification language CASL.
- The executable semantics framework K.
- The program verification environment RISC ProgramExplorer.
- The algorithm language and model checker RISCAL.
- The TLA$^+$ toolbox for modeling and checking concurrent systems.

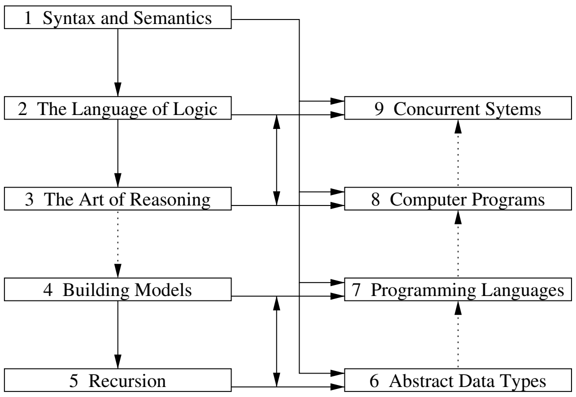
Most of these tools have been developed by other researchers; the RISC ProofNavigator, the RISC ProgramExplorer, and RISCAL are the results of the author's own work. All the presented software is freely available; the reader may download the examples presented in each section (see below) and run them on his/her own.

Teaching and Further Study

While of course the various chapters of this book in general linearly depend on each other in that each chapter may refer to some material presented earlier, Fig. 1 tries to outline the main dependencies by solid arrows (the dotted arrows represent weak dependencies); their consideration may be indeed useful in selecting material from this book for university courses that teach some specific topics such as

- Logic, Formal Modeling: Chaps. 1–3.
- Set Theory, Fixed Point Theory: Chaps. 4–5.
- Formal Specification of Abstract Data Types: Chap. 6.
- Formal Semantics of Programming Languages: Chap. 7 (with elements of Chap. 5).
- Formal Methods in Software Development: Chap. 8.
- Formal Models of Parallel and Distributed Systems: Chap. 9.

Fig. 1 Chapter dependencies



Indeed the author has taught courses (respectively participated in teaching) on most of these topics using material from this book (partially complemented by other more in-depth material).

In fact, since this book presents an integrated view on various (related but not identical) subjects, we had to choose from each field those aspects that we considered essential for conveying our core message: how by some universal principles of logical modeling and reasoning a software developer can better understand the artifacts that he/she is dealing with. This necessarily comes at the price that many other (also important) elements had to be neglected. To partially compensate for these gaps, each chapter concludes with a section Further Reading that suggests specific literature (mainly textbooks) that cover the presented topic in more depth or breadth, but typically from a different perspective. Lecturers, students, and readers may complement our presentation with additional material from these references.

Web Page and Exercises

This book is accompanied by electronic material available from the following URL:

https://www.risc.jku.at/people/schreine/TP

In particular, this web page contains all software examples presented in this book and exercises for the topics of the various book chapters.

Acknowledgments

The author has written this book as an associate professor at the Research Institute of Symbolic Computation (RISC), a stimulating Austrian institution for research, education, and application of algebraic and logic methods founded by Bruno Buchberger, formerly directed by Franz Winkler, and currently directed by Peter Paule; RISC is located in the Softwarepark Hagenberg and organizationally part of the Johannes Kepler University Linz. The book is based on the author's experience in teaching corresponding courses at this university, feedback received from students, and discussions with colleagues. In particular, Patrick Riegler has helped to improve the chapters of Part II of this book; Chaps. 2 "The Language of Logic" and 3 "The Art of Reasoning" have been influenced by the course module First-Order Logic jointly taught by the author and Wolfgang Windsteiger. We are also grateful to the anonymous reviewers whose suggestions helped to improve the manuscript (in particular by adding the introductory section with examples of industrial applications of formal methods). All errors, however, are the author's own.

Hagenberg, Austria Wolfgang Schreiner
March 2021

Contents

Logic for Programming: A Perspective

Wer nicht von dreitausend Jahren sich weiß Rechenschaft zu geben, bleib im Dunkeln unerfahren, mag von Tag zu Tage leben. (Who cannot draw on three thousand years may stay in the dark, inexperienced, living from day to day.)

—Johann Wolfgang von Goethe (West-östlicher Divan)

Reading this book may be conceived as traveling through a landscape of unfamiliar domains. Even if the relevance of these domains to computer programming will be continuously emphasized and also demonstrated by various software presentations, the reader may be more motivated to undertake this voyage, if she has some perspective on where the journey is heading. Therefore this section will demonstrate some real-life applications of logic to modeling and reasoning about computer programs, in particular also examples of industrial relevance. However, of equal importance is the dual understanding of where the journey has started; we will therefore begin with a short historical account on the creation of logic and its evolution to a tool of computer science. While this presentation is certainly incomplete and also influenced by the goals of this book and the personal preferences of its author, it may convey to the reader a first big picture of the landscape through which we are walking.

Logic and Language

Logic (from the Greek word logos which may be translated as reason) emerged in antiquity from the human desire to distinguish a valid argument from an invalid one. Initially such arguments were expressed in natural language and were intended for everyday discourse, but also for discussing philosophical and scientific questions (philosophical logic); the development of formal (symbolic or mathematical) logic occurred at much later times.

The roots of logic in the Western world (there have been alternative Eastern traditions in India and China) go back to the ancient Greece of the 4th century BCE. At that time, the Greek philosopher Aristotle developed in his writings later called

Organon a theory of syllogisms; a syllogism is a particular form of logical argument that derives from two sentences, the premises, a third sentence, the conclusion. These sentences have the form of particular subject-predicate statements, namely, every *A* is *B* or some *A* is *B*, respectively their negations; here *A* and *B* are terms that represent some basic concept like human or mortal, thus every human is mortal would be such a sentence. Although the expressiveness of this logic was quite limited, its main realization was that the validity of a logical argument (i.e., the correctness of a proof of its conclusion) only depends on the *form* of its sentences, not the interpretation of the terms in them. Indeed, Aristotle's logic (also called term logic) represented the essence of logic for more than two millennia, with only minor developments in medieval times.

A much more ambitious role for logic was envisioned in the 17th century by the German polymath Gottfried Wilhelm Leibniz (who also invented the binary numbers, the basis of digital computers today). He desired a characteristica universalis (a universal formal language), in which every mathematical, scientific, and metaphysical sentence could be expressed in such a way that a calculus ratiocinator (a logical calculation framework) could decide its truth by mechanical computation: thus, in order to solve disputes among persons with opposing opinions, Leibniz proposed calculemus! (Latin for let us calculate!). Indeed, Leibniz's writings sketched early forms of some concepts that would later appear in propositional logic and set theory. A resonating view was expressed in the 19th century by the English mathematician Ada Lovelace who wrote the world's first computer programs for Charles Babbage's analytical engine; she realized that this machine might act upon other things besides number, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations. Thus in the future computing machines might indeed be capable of fulfilling Leibniz's dream (Fig. 2).

Fig. 2 The Pioneers of Logic: Aristotle, Gottlob Frege, Kurt Gödel, Alfred Tarski

Logic and Mathematics

A first concrete step toward a more expressive kind of logic with also a more rigorous mathematical treatment was taken in the middle of the 19th century by the English mathematician and logician George Boole; he developed the principles of what was later called Boolean algebra (the formal basis of modern digital circuits), also known as propositional logic. The British mathematician Augustus De Morgane further refined Boole's system and extended it to a logic of binary relations; he also introduced the modern use of quantifiers. Based on these results, the American mathematician and logician Charles Peirce extended Boole's system to a logic of relations (the formal basis of relational databases today).

Independently of these developments in the English-speaking world, the modern kind of logic we use today was mainly developed by the German philosopher, logician, and mathematician Gottlob Frege; in his 1879 published Begriffsschrift (German for concept writing) he introduced what is today called predicate logic (later, distinctions would be made between first-order and second-order respectively higher-order predicate logic). While Frege's work was mainly ignored during his life time, it was later brought to attention by other scientists, such that Frege would be ultimately considered as the greatest logician since Aristotle. In this respect very influential was the Principia Mathematica of the English mathematician and philosopher Alfred North Whitehead and the British polymath Bertrand Russell; published in 1910, this work formulated mathematics on the basis of a variant of first-order logic and the theory of sets. This set theory had been informally developed by the German mathematician Georg Cantor in 1874 and was in 1908 formalized as an axiomatic theory in first-order logic by the German mathematician Ernst Zermelo; in 1922, this axiomatization was further refined by the German-born Isreali mathematician Abraham Fraenkel.

However, as Russel had already discovered in 1901, particular formalizations of set theory may lead to contradictions which are easy to overlook (Russel's paradox); thus the question remained open, whether mathematics was really based on a solid foundation. To settle this question, in 1920 the German mathematician David Hilbert proposed a research endeavor (Hilbert's program). The goal was to show that all of mathematics can be expressed in some formal logic as an axiomatic theory (i.e., by a list of logical formulas called *axioms*, this list may be infinite but must be enumerable) that is consistent (from the axioms no contradictory sentences can be proved) and complete (from the axioms every sentence or its negation can be proved). In 1928, this program was extended to the Entscheidungsproblem (German for decision problem): given any formula, a mechanical procedure should be able to decide in a finite amount of time whether this formula is provable or not.

Indeed Hilbert's program seemed well underway, when the Hungarian-American polymath John von Neumann proved in 1927 the consistency of a first-order axiomatization of a fragment of arithmetic and the Austrian logician Kurt Gödel proved in 1929 the completeness of first-order logic. However, in 1931 the program was utterly smashed by Gödel: he showed in his incompleteness theorems that in any logic every consistent axiomatization of arithmetic is necessarily incomplete, i.e., there are mathematical statements that can neither be proved nor disproved (this implies that second-order logic itself is incomplete, since arithmetic can be axiomatized by a single second-order formula). Even more, Gödel showed that no logical system can prove its own consistency. By these fundamental results, Gödel joined the ranks of Aristotle and Frege as one of the most significant logicians in history.

Therefore, while the formalization of mathematics today indeed rests upon first-order logic and Zermelo Fraenkel set theory, Gödel's incompleteness theorems set clear limits to this approach: neither can we be sure that the formalization is consistent (but no inconsistencies could ever be found), nor can every mathematical question be settled. For instance, in 1963 the American mathematician Paul Cohen showed that the "continuum hypothesis" (there is no set whose size is strictly between that of the integers and that of the real numbers) can neither be proved nor disproved from the first-order axioms of Zermelo-Fraenkel set theory. Thus there exist mathematical statements that are (in a certain sense) "neither true nor false".

But what if we contend ourselves with the more modest goal of deciding formulas in simpler theories which can be axiomatized in first-order logic? Indeed one aspect of the Entscheidungsproblem of first-order logic can be achieved (i.e., the problem is "semi-decidable"): given a provable formula, it is indeed possible by a mechanical procedure to find this proof in a finite amount of time. However, in 1936 the American mathematician Alonzo Church and the English polymath Alan Turing independently showed that the overall goal is impossible to reach (i.e., the Entscheidungsproblem is "undecidable"): if a first-order formula is unprovable, any mechanical procedure that attempts to prove/disprove it may run forever. Thus first-order logic can be only semi-automated: given a formula to be decided, we can never be sure whether the procedure just needs more time to find its proof or whether it runs forever in a doomed attempt to prove an unprovable formula (to show these results, Church developed the "λ-calculus" and Turing the "Turing machine", the first formal models of full-fledged "Turing-complete" programming languages).

A completely new approach to logic was established by the Polish-born American logician Alfred Tarski. From Aristotle to Gödel, logic had been treated as a purely syntactical game that investigated how formulas could be proved; thus (apart from an intuitive interpretation) a formula actually had no inherent meaning that was independent of its provability. In 1936, however, Tarski gave logic a formal "semantics" by the interpretation of formulas in some "model"; he thus created the field of "model theory" that investigates the interplay between "proof and truth". The original notion of the "consistency" of a calculus was thus widely replaced by the notion of its "soundness" (every proved formula is true) and the original notion of "completeness" was correspondingly redefined (every true formula can be proved). Nowadays, logic is mainly presented within the model-theoretic framework established by Tarski.

Logic with Computers

After the previous decades had clarified the capabilities and limitations of formal logic, since the 1960s more and more work was dedicated to turn the theoretical foundations into practical results by applying computer programs to logical reasoning. Here essentially three strands have emerged:

- *Automated Reasoning Systems:* these are systems that prove formulas in first-order logic, typically by applying some sound and complete proving calculus of which various have been developed since the 1930s. Since the "search space" for finding a proof is of infinite size, the main challenge is to define an efficient search strategy, Here a major step forward was the "resolution algorithm" invented by Alan Robinson [151] which considerably limits the search space and is applied in many modern first-order provers, e.g., the "Vampire" prover developed by Andrei Voronkov and colleagues at the University of Manchester [176]. However, systems may be also based on more "human-oriented" strategies such as the "Theorema" system developed by Bruno Buchberger and colleagues at the RISC institute of the Johannes Kepler University Linz [175].
- *Interactive Proving Assistants:* fully automated reasoning systems reach their practical limits when dealing with complex mathematical theories. Therefore, since the 1970s research has been pursued on interactive proving assistants where a proof is developed by a collaboration of human and machine: the human generally directs the proof construction but lets the computer elaborates the tedious details. Here the "Logic for Computable Functions" (LCF) developed by Michael Gordon, Robin Milner, and colleagues at the universities of Edinburgh and Stanford was very influential [59]. LCF introduced the idea of a "trusted core" that implements a small set of logical rules (in the functional programming language "ML" which was developed for this purpose); this core is represented by an abstract datatype whose constructors are the logical rules. The core can be arbitrarily extended by derived rules in the form of proof construction procedures that may be invoked by humans or by automated proof search procedures; still the correctness of a proof only depends on the soundness of the trusted core. Since these systems do not require a complete inference systems, they typically implement a higher-order logic; notable examples are HOL family of provers [79] and the Isabelle/HOL proving assistant [132] developed at the University of Cambridge and the Technische Universität München by Lawrence Paulson, Tobias Nipkow, and colleagues.
- *Satisfiability Solvers:* the problem of deciding the validity of a formula can be reduced to deciding the satisfiability of its negation. This "SAT problem" for propositional logic is NP-complete such that we cannot expect to solve it generally faster than in exponential time. Nevertheless, since the late 1990s heuristically fast SAT solvers have been developed that are effectively able to solve problems with tens of thousands of Boolean variables [21]; these solvers have found wide application in hardware design, planning, scheduling, and

optimization. Furthermore, also for the quantifier-free fragment of first-order logic the satisfiability problem is decidable in certain (combinations of) mathematical theories such as "linear arithmetic" or "uninterpreted functions with equality". Corresponding "Satisfiability Modulo Theories" (SMT) solvers are nowadays used as central components in the verification of computer programs [10]; a well-known representative of this category is the Z3 solver developed by Leonardo De Moura and Nikolaj Bjørner at Microsoft [183].

In the area of mathematics, interactive proving assistants have been used to establish new results such as the "four-color theorem", the "Kepler conjecture", or the "Boolean Pythagorean" triples problem; however, these are structurally simple "proofs by exhaustion" where an overwhelmingly large number of cases is checked by arithmetic calculation respectively satisfiability solving. Conceptually more interesting is the use of such assistants to formally verify versions of (perhaps previously disputed) proofs such as that of the Jordan curve theorem [64]. Fully automated reasoning systems have been mainly used to confirm the validity of already known theorems; occasionally these systems also found more elegant proofs than were previously known (e.g., that of a theorem in the Principia Mathematica); a really new mathematical result was established by the automated prover EQP that found a proof of the "Robbins conjecture" [112] by equational reasoning. Generally, most success has been achieved by procedures targeted to special problems such as proofs of algebraic identities which occur in combinatorics and in the theory of special functions, such as implemented by the summation package "Sigma" developed by Carsten Schneider at the RISC institute of the Johannes Kepler University Linz [162].

Summarizing, new mathematical results were established with the help of automated or interactive reasoning systems mainly if they involved activities for which computers are more suitable than humans, such as checking many cases or establishing long chains of identities. More pragmatic success was achieved in verifying mathematical truths by formalizing existing proofs (respectively in detecting errors and gaps in these proofs); in particular, the Mizar project established by Andrzej Trybulec at the University of BiaÅ‚ystok has developed a large library of strictly formalized mathematics on the basis of the Mizar proof assistant [120].

Be that as it may, while computers have become a *tool* for logic, they have simultaneously also opened a new and very fruitful *domain* for logic, that of computer systems and computer programs themselves.

Logic for Computer Science

With the advent of electronic computers in the 1940s, more and more complex problems were solved by computer programs; thus, however, it became more and more difficult to write computer programs that are indeed correct, i.e., that for all given inputs deliver the expected outputs respectively produce the expected effects.

Fortunately, it also became clear that it is not really necessary to run a computer program to predict its behavior. Only poor programmers write their programs by "trial and error"; good programmers apply some form of rational reasoning to deduce the behavior of their programs from the texts of the programs alone. It therefore should be possible to formalize this reasoning process in the form of a logical theory and predict the behavior of the program by logical deduction in that theory.

Indeed, based upon the pioneering works of the American computer scientists John McCarthy and Robert Floyd, the British computer scientist Tony Hoare developed in 1969 a logical inference system later called "Hoare calculus" which provides a suitable framework for this kind of reasoning [73]. Given a "precondition", a first-order formula that describes the possible inputs of a program, and a postcondition, a first-order formula that describes the desired outputs, a valid deduction in the Hoare calculus ensures that every execution of the program with inputs that satisfy the "precondition" yields outputs that satisfy the "postcondition". This deduction proceeds via the derivation of "verification conditions", first-order formulas whose truth implies the correctness of the program; thus the Hoare calculus reduces the problem to reasoning about the correctness of computer programs to the problem of proving formulas in first-order logic. However, the calculus itself does not give a strategy to build valid deductions. This was amended in 1975, when the Dutch computer scientist Edsger Dijkstra developed in his "predicate transformer semantics" an effective algorithm for deriving suitable verification conditions via the computation of "weakest preconditions" or (dually) "strongest postconditions"; it is this algorithm that was subsequently to be applied in most systems for program verification [41].

The Hoare calculus only defines the meaning of programs implicitly via rules that allow us to prove the correctness of programs with respect to specifications; in this sense it represents an "axiomatic semantics" of programs. However, in 1971 the American logician Dana Scott and the British computer scientist Christopher Strachey gave (on the basis of the "domain theory" developed by Scott) recursive functions and thus also iterative computer programs a "denotational semantics", i.e., they assigned an explicit meaning to programs [171]; this allows (in the spirit of Tarski's work on the semantics of first-order logic) to prove the soundness and completeness of the Hoare calculus with respect to the semantics of programs. Another approach was a new form of "operational semantics" of programs developed in the 1980s by the British computer scientist Gordon Plotkin ("structural operational semantics" [144]) and the French computer scientist Gilles Kahn ("natural semantics" [85]). Here the semantics of a program is defined by a logical deduction system whose inference rules mimic the execution steps of the program; thus it becomes possible to formally relate the mathematical denotation of a program to its operational interpretation. All in all, by these various approaches to formalizing the semantics of programs, computer science has established a firm basis for the execution and translation of computer programs (processors, interpreters, compilers).

So far, the main consideration was programs that transform given inputs to expected outputs; here it is only necessary to deal with *two* states, the input state and the output state of a program. However, this does not really address systems that repeatedly interact with their environment, such as the components of concurrent programs or computing systems which run through (potentially even infinite) sequences of states, each of which represents a possible point of interaction with other components. In 1977, the Israeli computer scientist Amir Pnueli proposed to apply a logic originally introduced by the New Zealand-born logician and philosopher Arthur Prior as a "temporal logic" to specify and reason about the behavior of such systems [145]. Temporal logic can be considered as an extension of first-order logic; while a first-order formula talks about a *single* program state, a temporal formula talks about *arbitrarily many* such states. This approach has become in the following decades the basis of numerous systems for concurrent system modeling and verification.

In the 1970s and 1980s, also another general approach to dealing with the problem of developing correct computer programs was widely pursued: the attempt to abandon conventional programming languages in favor of using logic itself as a much more "high-level" programming language; thus the gap between the specification of a problem and the implementation of a program solving this problem would be closed. In a certain sense, Gödel's proof of the completeness of first-order logic shows that first-order logic itself is a Turing-complete programming language: every computation can be expressed as the proof of a first-order formula. However, finding this proof is overwhelmingly more costly than performing the computation in a conventional language; therefore research was pursued on developing efficient execution mechanisms for fragments of first-order logic. Instances of this idea are "abstract data type languages" based on equational logic (in particular the "OBJ" language family initiated by the American computer scientist Joseph Goguen [58]), and "logic programming languages" based on "Horn clause" logic with applies a special form of resolution (in particular the language "Prolog" developed by the French computer scientists Alain Colmerauer and Philippe Roussel [38]). The Japanese "Fifth Generation Computer" Systems initiative even pursued the development of massively parallel computer systems based on concurrent logic programming languages [173]. While these approaches of a "direct" use of logic did ultimately not supplant conventional programming languages or computing systems, many principles developed in these endeavors formed the basis of the "indirect" use of logic for programming; in particular the type systems of modern programming languages and the theory of data types specification was substantially influenced by abstract data type languages, as well as automated reasoning over equational theories.

Returning to the topic of the verification of computer programs and systems, a core problem with corresponding logical calculi (Hoare calculus, predicate transformers, temporal logic) is that they crucially depend on additional information that adequately characterizes the behavior of iterative computations by logical formulas. These "invariants" are not inherent in the programs themselves but have to be provided by some external source, in practice by the human programmer. If the invariants are not adequate, the derived verification conditions do not hold and their proofs fail. Thus in practice program verification has to struggle with two kinds of uncertainties: the fundamental uncertainty in the correctness of the program and the additional uncertainty in the adequacy of the invariants; errors on both levels lead to unprovable verification conditions. This was especially problematic in the 1970s and 1980s, when automated reasoning systems and interactive proving assistants were only able to provide adequate reasoning support for verification conditions arising from simple "toy programs"; thus, as a third uncertainty, the failure of proving a verification condition could also be due to the inadequacy of the proof automation. For all these reasons, at that time the technique of program verification by logical deduction was generally not considered of much practical relevance.

However, in the 1980s and 1990s with the technique of "model checking" an alternative approach emerged [37]. Rather than investigating logical theories with arbitrarily many interpretations ("models") of generally infinite size, model checking focuses on a single model of finite size and analyzes its properties. This approach evolved in the area of hardware verification, because a digital circuit can be described by such a finite model (the finitely many combinations of the states of logic gates). Given a temporal logic formula that describes an expected property of such a model, sophisticated encoding and analysis techniques are able to fully automatically decide whether the model satisfies this property. Similar techniques can be also applied to computer programs if we consider the domains of all program variables as finite bit vectors and assume a finite bound for the number of execution steps ("bounded model checking" [20]); furthermore, the domain of an infinite-state system or program can be abstracted to a finite domain that is amenable to "abstraction model checking" [36]. While these approaches are typically not able to verify the general correctness of a program, they may still detect typical programming errors that lead to the abortion of programs (division by zero, null pointer dereferences, out-of-bound array indices).

In the 2000s, however, interest in the more general approach of program verification by logical deduction revived. By that time, substantial advances had been made in automated reasoning, especially fueled by the already discussed emerge of practical SMT solvers [10] which are able to decide the satisfiability of quantifier-free formulas in combinations of theories which are quite relevant in computer programming, such as linear integer arithmetic or the theory of arrays. Such "SMT solvers" have become the building blocks of many program verification environments which combine a program reasoning calculus (such as Dijstra's weakest preconditions) that automatically derives verification conditions with an automated reasoning system or interactive proving assistant that deals with the quantification structure of these conditions; finally SMT solvers can be applied to handle the resulting quantifier-free fragment. In this way, many interesting program verification problems can be nowadays successfully solved.

Logic and Software Development

After this historical excursion, we will fulfill our initial promise of discussing some concrete examples of non-trivial software whose correctness has been actually established by logical modeling and reasoning. Here we will consider only the verification of software and altogether omit the topic of hardware verification. Furthermore, we will focus entirely on the verification of the "functional" correctness of programs; there are also approaches to establish non-functional requirements, such as "security" guarantees or "real-time" constraints. Furthermore, we will not discuss the also important topic of the verification of "cyber-physical" systems which combine digital controllers with physical sensors respectively actuators and are therefore governed not only by the laws of computing but also by the laws of nature.

Some of the following examples do indeed describe "industrial" applications, others represent major research activities with a mid-term perspective of industrial impact, some are listed because they may give a glimpse into the long term future of industrial software development. While it is in a short space not possible to describe these examples in great detail, we roughly sketch their underlying logical approaches and relate these to the topics presented in this book.

Verified System Designs and Implementations Engineers at Amazon Web Services (AWS) have since 2012 used the temporal logic modeling language TLA$^+$ (described on page 591 of this book) to model and verify critical components of the AWS infrastructure [128]. This work started in the context of Amazon's S3 Simple Storage Service, when the designer of the replication and fault-tolerance mechanisms of the DynamoDB data store component of that service wrote a detailed logical model of these mechanisms and applied the TLC model checker to verify its expected properties. Thus a subtle bug in the design of the fault-tolerant algorithm was detected that could lead to losing data if a particular sequence of failures and recovery steps would be interleaved with other processing; this bug had previously passed unnoticed through extensive design reviews, code reviews, and testing. By more formal modeling and verification, later two more bugs were detected in other algorithms, both serious and subtle. After these initial successes, TLA$^+$ was presented to a broader engineering community at AWS who applied it to a new fault-tolerant network algorithm (revealing two bugs and some more bugs in extended and optimized versions of the algorithm) and another critical algorithm (revealing that a proposed fix to a bug previously detected in testing actually had not removed that bug); similar experiences were reported by other engineers. Subsequent new algorithms and protocols would be routinely modeled and verified with TLA$^+$ before translating the designs to actual production code.

While AWS only verified system *designs*, Microsoft Research extended its ambition toward verified system *implementations*. In its framework "IronFleet", this organization applied a TLA-like approach to build two complex distributed systems, IronRSL (a replicated state machine library) and IronKV (a key-value store), whose correctness (with respect to safety and liveness properties, see Sects. 9.5 and 9.6) was formally proved [67]. This verification proceeded in three layers and utilized various techniques presented in this book: On the highest layer, the system is described in Microsoft's modeling and verification language "Dafny" as a state machine in a logical form (similar to the transition systems described in Sect. 9.1). On the middle layer, in Dafny a distributed state machine is defined, again in a logical form (similar to the semantics of distributed systems described in Sect. 9.3). On the lowest layer, imperative Dafny code is written for each component of the distributed state machine; this code is then automatically translated to C# code and compiled to the executable code of the system. To semantically connect the layers, it is proved that each layer is refined by the next lower one. using an abstraction function that relates the states of both layers (see the refinement techniques presented in Sects. 7.4 and 9.7). To verify the correctness of the protocol layer, TLA-style property specifications are translated into corresponding Dafny predicates over state sequences, i.e., temporal logic reasoning is reduced to first-order reasoning (see the semantics of temporal operators specified in Sect. 9.4); for the verification of the lowest layer, Hoare-style reasoning is applied (see Sect. 8.2). The actual proofs are performed within the Dafny framework with the help of Microsoft's SMT solver Z3 and human-provided annotations to guide the prover through the quantifier instantiations.

Verified Program Components and Libraries Rather than attempting to verify whole systems, more often efforts concentrate on modeling and verifying individual critical program components. In modern object-oriented programming, these components are mainly represented by "classes" that encapsulate data and the methods operating on these data. For various object-oriented programming languages, "behavioral interface specification languages" have been developed [66] that describe not only the syntactic interfaces of classes but their semantic behavior via logical preconditions and postconditions of their methods and logical invariants of their objects (the basics reasoning about the correctness of methods respectively procedures are discussed in Chap. 8). Furthermore, to specify the external behavior of classes without exposing their internal representation, it is usually necessary to relate these classes to "models" (examples of such models are axiomatically specified abstract data types as discussed in Chap. 6). In the ecosystem of the object-oriented programming language Java, the "Java Modeling Language" (JML) has emerged as the de facto standard behavioral interface specification language which is supported by a variety of tools [34, 136].

A prominent representative of such a tool is "KeY", a formal verifier for Java that has since the late 1990s been jointly developed by the Karlsruhe Institute of Technology, the Chalmers University of Technology, and the Technische Universität Darmstadt [3, 148]. KeY is internally based upon "dynamic logic", a logic that combines classical first-order logic with a program logic (the principles of program reasoning are essentially the same as the calculi presented in Chap. 8). KeY has been used to verify non-trivial Java code respectively detect errors in such code. The original target was Java Card, a subset of Java for smartcards and embedded devices. Until 2005, various real-life industrial examples of Java Card applications were formally specified and verified with KeY; subsequently, KeY was extended to support many features of the full Java language. In 2015, an attempt to use KeY to verify the sorting algorithm of the Java library (a hybrid combination of merge sort and insertion sort called “Timsort”) revealed that this widely used Java method actually had a bug; a corrected version could be successfully verified. In 2017, the correctness of another sorting algorithm available in the Java library (Dual Pivot Quicksort) was shown. In 2018, KeY was used to verify the correctness of Hyperledger Fabric Chaincode, a protocol for smart contracts built upon blockchain technology. In 2020, core components of EVA (the Java-based main support system for elections in municipalities and counties in Norway) were formally specified in JML and verified with KeY.

Beyond individual classes, an interesting target of verification is whole class libraries such as the “container” libraries available in many programming languages; here the main challenge is to show the correctness of a concrete (optimized) internal representation with respect to an abstract mathematical specification. For instance, in 2015 researchers from MIT and ETH Zürich verified the full functional correctness of Eiffel-Base2, a container library (with more than 130 public methods and 8400 lines of code) for the object-oriented programming language Eiffel that offers all the features customary in modern language frameworks [146]. The proof was performed with the help of the automated deductive verifier AutoProof that translated Eiffel code annotated with logical specifications and invariants into Microsoft’s intermediate verification language “Boogie”; the Boogie verifier then generated verification conditions that were ultimately discharged by the Microsoft’s SMT solver Z3. The library specification relied heavily on mathematical model types (essentially abstract data types as discussed in Chap. 6), the relationship between the actual representation and the model type was provided by abstraction functions (as described in Sect. 6.8).

Verified Compilers The verification of computer programs is usually based on a logical model of the high-level language in which the source code of the program is written. However, since this source code itself is not executable, it is typically compiled into an actually executable form, a program in the machine language of the underlying computer processor. This translation process itself is a potential source of errors, i.e., the generated machine program might behave differently than expected from the logical analysis of the original source code; thus it is a worth-wile goal to verify the compiler itself.

This problem of compiler verification has been addressed by the “CompCert” project initiated in 2007 by the French computer scientist Xavier Leroy; its main outcome is a verified compiler for a large subset of the C99 programming language which generates efficient code for the PowerPC, ARM, RISC-V, x86, and x86-64 processors [82, 101]. The compiler is written in the specification language of the proof assistant Coq in purely functional style; from this Coq specification, executable code in the functional language Caml is generated. While the Caml implementation itself is unverified, the Coq specification ensures the correctness of the translation of the C99 code to machine code via a sequence of transformations through various intermediate languages. For each of the source, intermediate, and target languages, a formal operational semantics is defined (see Sect. 7.3) and for each transformation it is proved that the generated code preserves the semantics by showing that each step of the original program is simulated by a sequence of steps of the transformed program (see the techniques presented in Sect. 7.4). The whole Coq specification consists of 42000 lines of which approximately 14 represent the compilation algorithms, 10 defined the formal semantics of the various languages, and 76 represent the correctness proofs themselves. These proofs were performed in Coq by a combination of user interaction to guide the proof and the application of automated decision procedures to discharge proof obligations; the proofs are recorded in the form of proof terms that apply a small set of logical rules; their correctness can be independently verified by a trusted proof checker. Since 2015, the CompCert compiler has been commercially available; in 2017 it was used to certify a highly safety-critical industrial application (a digital engine unit that controls the backup diesel engines of nuclear power plants) according to the IEC 60880 standard for nuclear power plant control systems.

While CompCert still relies on an unverified implementation of Caml, the "CakeML" project initiated in 2012 has produced a verified compiler that is written itself in the language that it verifies, a subset of Standard ML [32, 94]. Thus the compiler can compile itself and so produce a verified executable program that provably implements the compiler itself; this "bootstrapping" process started (along the lines of the techniques used in CompCert) with a specification of the compiler in the language of the interactive theorem prover HOL4, which was also used to perform all proofs. CakeML was applied to the end-to-end verification of various Unix-like command line tools, a proof checker for the OpenTheory standard, and a certificate checker for floating-point error bounds.

Verified Operating System Kernels The most complex program running on a computer is usually not an application executed on behalf of some user but the "operating system", i.e., software which extends the basic capabilities of the computer processor by an additional set of services. Errors in operating systems may have devastating consequences: apart from the danger of computer crashes and data losses, they also represent security holes which malicious attackers may exploit, not only locally, but remotely over the Internet. Since an operating system is run on millions of computers worldwide, it pays off to invest some efforts to ensure its correct behavior by formal verification. Indeed, this already happens regularly, albeit in a limited form: the Microsoft Windows device driver frameworks (WDF) incorporate a "static driver verifier" (SDV), an abstract model checker that was developed by Thomas Ball and Sriram Rajamani at Microsoft Research since 1999 and originally called SLAM (the most recent version SLAM2 was released in 2010) [9, 114]. This model checker verifies that a device driver interacts correctly with the Windows operating system, e.g., by detecting illegal function calls or actions that may cause system corruption. The technique applied by SLAM is "counter-example guided abstraction refinement" which is based on the principle of "symbolic execution", essentially a form of Dijkstra's predicate transformer calculus applied to abstractions of program states; such an abstraction is represented by the values of a set of predicates on the state. If an error is detected in the abstract model of the program, Microsoft's SMT solver Z3 is applied to determine whether its abstraction corresponds to an error in the real program; if not, the model is refined by extending the predicate set. Based upon similar principles (but with many improvements) are the C language model checkers “BLAST” [17] and “CPAchecker” [16, 18]; these have been used to find bugs in kernel drivers of the GNU/Linux operating system.

A much more ambitious goal was followed by the “seL4” project pursued at the Australian NICTA Research Center under the direction of Gernot Heiser. In 2009, this project developed a formally verified microkernel and hypervisor with completely proved functional correctness and security guarantees; in 2014 the seL4 microkernel was released as open source [87, 172]. The main domain of application of seL4 are safety-critical systems such as industrial control systems, medical devices, and autonomous vehicles; for instance, it was utilized in Boeing's Unmanned Little Bird (ULB) helicopter prototype [86]. The development of seL4 started with an abstract formal specification expressed in the language of the proof assistant Isabelle/HOL (described on pages 141 and 191 of this book). This specification was essentially in the form of an abstract state machine (similar to the transition systems described in Sect. 9.1) for which certain correctness properties were proved on the basis of appropriate system invariants (see Sect. 9.5). Then a prototype of the microkernel was developed in the purely functional programming language Haskell; this prototype represented an executable form of the state machine with concrete data structures implementing the abstract types of the specification. After appropriate testing, the Haskell code was automatically translated into the language of Isabelle/HOL and it was proved that this translated form of the executable specification “refined” the previously specified abstract machine (see the refinement techniques presented in Sects. 7.4 and 9.1); thus the executable specification preserved the correctness properties of the abstract specification. Then the Haskell prototype was manually re-implemented in a subset of the C99 programming language where the high-level functional constructions of Haskell were now expressed in low-level imperative code. Based on a formal definition of the semantics of C99 in Isabelle/HOL (essentially a small step operational semantics as described in Sect. 7.3), this C99 code was automatically translated into its formal semantics in Isabelle/HOL. It was then proved that this semantics refined the executable specification; thus the C99 implementation of the seL4 microkernel preserved the correctness properties of its abstract specification. The seL4 project was one of the largest formal verification efforts so far; the various specification and coding efforts took about 2.5 person years, while the proving efforts required about 11 person years (including the setup of the proving infrastructure and learning curve); it was estimated that a subsequent effort would reduce the later figure to 6 person years, essentially twice as much as required by traditional quality assurance methods which do not provide formal correctness guarantees.

From the above descriptions, one can see that it is comparatively rarely the case that already existing (“legacy”) software is verified; generally more promising it is to design new software already with the goal of verifying its correctness in mind. Often this “codesign” of a software and the proof of its correctness proceeds in multiple layers where first an abstract design is specified and verified and then gradually refined to actual code; if we can prove that each layer is properly “refined” by the next layer, the lowest layer has inherited the correctness properties from the highest one.

Further Reading

As for the history of logic till the 20th century, good general resources are the Wikipedia pages on the corresponding persons and topics; more in-depth articles can be found in the online Stanford Encyclopedia of Philosophy [174].

For surveys on techniques and tools for the automation of logical reasoning, see for instance the various chapters of the handbook [152] edited by Robinson and Voronkov, the chapters of the book [179] edited by Wiedijk, or Harrison’s handbook [65].

A historical account of the application of formal methods to computer science is given in the article [23] by Bjørner and Havelund. For surveys on applications of formal methods in industrial practice, see the paper [182] of Woodcock and others and the more recent article [57] of Gleirscher and others. The VSTTE conference series [35] provides numerous examples of actual software verifications.