ACM MONOGRAPH SERIES

A Computational Logic

Robert S. Boyer
J Strother Moore

A Computational Logic

ROBERT S. BOYER and J STROTHER MOORE

SRI International
Menlo Park, California

ACADEMIC PRESS

A Subsidiary of Harcourt Brace Jovanovich, Publishers

New York London Toronto Sydney San Francisco

ACADEMIC PRESS, INC.
111 Fifth Avenue, New York, New York 10003

United Kingdom Edition published by
ACADEMIC PRESS, INC. (LONDON) LTD.
24/28 Oval Road, London NW1 7DX

Library of Congress Cataloging in Publication Data

Boyer, Robert S
A computational logic.

(ACM monograph series)
Includes bibliographical references and index.
1. Automatic theorem proving. I. Moore,
J Strother, Date joint author. II. Title.
III. Series: Association for Computing Machinery.
ACM monograph series.
QA76.9.A96B68 519.4 79-51693
ISBN 0-12-122950-5

PRINTED IN THE UNITED STATES OF AMERICA

79 80 81 82 9 8 7 6 5 4 3 2 1

A Computational Logic

This i
ACM MONOGRAPH SERIES

Editor: THOMAS A. STANDISH, *University of California at Irvine*

A complete list of titles in this series appears at the end of this volume.

To our wives,
Anne and Liz

Preface

Mechanical theorem-proving is crucial to the automation of reasoning about computer programs. Today, few computer programs can be mechanically certified to be free of "bugs." The principal reason is the lack of mechanical theorem-proving power.

In current research on automating program analysis, a common approach to overcoming the lack of mechanical theorem-proving power has been to require that the user direct a proof-checking program. That is, the user is required to construct a formal proof employing only the simplest rules of inference, such as *modus ponens,* instantiation of variables, or substitution of equals for equals. The proof-checking program guarantees the correctness of the formal proof. We have found proof-checking programs too frustrating to use because they require too much direction.

Another approach to overcoming the lack of mechanical theorem-proving power is to use a weak theorem-proving program and to introduce axioms freely. Often these axioms are called "lemmas," but they are usually not proved. While using a proof checker is only frustrating, introducing axioms freely is deplorable. This approach has been abused so far as to be ludicrous: we have seen researchers "verify" a program by first obtaining formulas that imply the program's correctness, then running the formulas through a simplifier, and finally assuming the resulting slightly simplified formulas as axioms. Some researchers admit that these "lemmas" ought to be proved, but never get around to proving them because they lack the mechanical theorem-proving power. Others, however, believe that it is reasonable to assume lots of "lemmas" and never try to prove them. We are strongly opposed to this latter attitude because it so completely undermines the spirit of proof, and we therefore reply to the arguments we have heard in its defense.

(1) It is argued that the axioms assumed are obvious facts about the concepts involved. We say that a great number of mistakes in computer programs arise from false "obvious" observations, and we have already seen researchers present proofs based on false lemmas. Furthermore, the concepts involved in the complicated computer systems one hopes eventually to certify are so insufficiently canonized that one man's "obvious" is another man's "difficult" and a third man's "false."
(2) It is argued that one must assume some axioms. We agree, but observe that mathematicians do not contrive their axioms to solve the problem at hand. Yet often the "lemmas" assumed in program verification are remarkably close to the main idea or trick in the program being checked.
(3) It is argued that mathematicians use lemmas. We agree. In fact, our theorem-proving system relies heavily on lemmas. But no proof is complete until the lemmas have been proved too. The assumption of lemmas in program proving often amounts to sweeping under the rug the hard and interesting inferences.
(4) It is argued that the definition of concepts necessarily involves the addition of axioms. But the axioms that arise from proper definitions, unlike most "lemmas," have a very special form that guarantees two important properties. First, adding a definition never renders one's theory inconsistent. Second, the definition of a concept involved in the proof of a subsidiary result (but not in the statement of one's main conjecture) can safely be forgotten. It does not matter if the definition was of the "wrong" concept. But an ordinary axiom (or "lemma"), once used, always remains a hypothesis of any later inference. If the axiom is "wrong," the whole proof may be worthless and the validity of the main conjecture is in doubt.

One reason that researchers have had to assume "lemmas" so freely is that they have not implemented the principle of mathematical induction in their theorem-proving systems. Since mathematical induction is a fundamental rule of inference for the objects about which computer programmers think (e.g., integers, sequences, trees), it is surprising that anyone would implement a theorem-prover for program verification that could not make inductive arguments. Why has the mechanization of mathematical induction received scant attention?

Perhaps it has been neglected because the main research on mechanical theorem-proving, the resolution theorem-proving tradition (see Chang and Lee [15] and Loveland [29]), does not handle axiom schemes, such as mathematical induction.

We suspect, however, that the mechanization of mathematical induction has been neglected because many researchers believe that the only need for induction is in program semantics. Program semantics enables one to obtain from a given program and specification some conjectures ("verification conditions") which imply that the program is correct. The study of program semantics has produced a plethora of ways to use induction. Because some programs do not terminate, the role of induction in program semantics is fascinating and subtle. Great effort has been invested in mechanizing induction in program semantics. For example, the many "verification condition generation" programs implicitly rely on induction to provide the semantics of iteration.

But program semantics is not the only place induction is necessary. The conjectures that verification condition generators produce often require inductive proofs because they concern inductively defined concepts such as the integers, sequences, trees, grammars, formulas, stacks, queues, and lists. If you cannot make an inductive argument about an inductively defined concept, then you are doomed to assume what you want to prove.

This book addresses the use of induction in proving theorems rather than the use of induction in program semantics.

We will present a formal theory providing for inductively constructed objects, recursive definitions, and inductive proofs. Readers familiar with programming languages will see a strong stylistic resemblance between the language of our theory and that fragment of the programming language LISP known as "pure LISP" (see McCarthy et al. [35]). We chose pure LISP as a model for our language because pure LISP was designed as a mathematical language whose formulas could easily be represented within computers. Because of its mathematical nature (e.g., one cannot "destructively transform" the ordered pair ⟨7, 3⟩ into ⟨8, 3⟩), pure LISP is considered a "toy" programming language. It is an easy jump to the *non sequitur:* "The language and theory presented in this book are irrelevant to real program analysis problems because they deal with a toy programming language." But that statement misses the point. It is indeed true that our theory may be viewed as a programming language. In fact, many programs are naturally written as functions in our theory. But our theory is a mathematical tool for making precise assertions about the properties of discrete objects. As such, it can be used in conjunction with any of the usual program specification methods to state and prove properties of programs written in any programming language whatsoever.

When we began our research into proving theorems about recursive functions [7, 38], we thought of ourselves as proving theorems only about pure LISP and viewed our work as an implementation of McCarthy's [34] functional style of program analysis. However, we now also regard recursion as a natural alternative to quantification when making assertions about programs. Using recursive functions to make assertions about computer programs no more limits the programming language to one that implements recursion than using the ordinary quantifiers limits the programming language to one that implements quantification! In this book we use both the functional style and Floyd's inductive assertion style [18] of program specification in examples. (For the benefit of readers not familiar with the program verification literature, we briefly explain both ideas when they are first used.) We have relegated the foregoing remarks to the preface because we are not in general interested in program semantics in this book. We are interested in how one proves theorems about inductively constructed objects.

Our work on induction and theorem-proving in general has been deeply influenced by that of Bledsoe [3, 4]. Some early versions of our work have been previously reported in [7, 38, 39, 40, 8]. Work closely related to our work on induction has been done by Brotz [11], Aubin [2], and Cartwright [14].

We thank Anne Boyer, Jacqueline Handley, Paul Gloess, John Laski, Greg Nelson, Richard Pattis, and Jay Spitzen for their careful criticisms of this book. We also thank Jay Spitzen for showing us how to prove the prime factorization theorem. We thank Bernard Meltzer for the creative atmosphere in the Metamathematics Unit of the University of Edinburgh, where we began our collaboration. Finally, we thank our wives and children for their usually cheerful long-suffering through the years of late hours behind this book.

Our work has been supported by the National Science Foundation under Grant MCS-7681425 and by the Office of Naval Research under Contract N00014-75-C-0816. We are very grateful to these agencies for their support.

Contents

I

Introduction

Unlike most texts on logic and mathematics, this book is about how to prove theorems rather than the proofs of specific results. We give our answers to such questions as

When should induction be used?

How does one invent an appropriate induction argument?

When should a definition be expanded?

We assume the reader is familiar with the mathematical notion of equality and with the logical connectives "and," "or," "not," and "implies" of propositional calculus. We present a logical theory in which one can introduce inductively constructed objects (such as the natural numbers and finite sequences) and prove theorems about them. Then we explain how we prove theorems in our theory.

We illustrate our proof techniques by using them to discover proofs of many theorems. For example, we formalize a version of the propositional calculus in our theory, and, using our techniques, we formally prove the correctness of a decision procedure for that version of propositional calculus. In another example, we develop elementary number theory from axioms introducing the natural numbers and finite sequences through the prime factorization theorem.

Since our theory is undecidable, our proof techniques are not perfect. But we know that they are unambiguous, well integrated, and successful on a large number of theorems because we have programmed a computer to follow our rules and have observed the program prove many interesting theorems. In fact, the proofs we describe are actually those discovered by our program.

A. MOTIVATION

Suppose it were practical to reason, mechanically and with mathematical certainty, about computer programs. For example, suppose it were practical to prove mechanically that a given program satisfied some specification, or exhibited the same output behavior as another program, or executed in certain time or space bounds.[1] Then there would follow a tremendous improvement in the reliability of computer programs and a subsequent reduction of the overall cost of producing and maintaining programs.

To reason mechanically about programs, one must have a formal program semantics, a formal logical theory, and a mechanical theorem-prover for that theory. The study of formal program semantics has provided a variety of alternative methods for specifying and modeling programs. But all the methods have one thing in common: they reduce the question, Does this program have the desired property? to the question, Are these formulas theorems? Because of the nature of computers, the formulas in question almost exclusively involve inductively constructed mathematical objects: the integers, finite sequences, n-tuples, trees, grammars, expressions, stacks, queues, buffers, etc. Thus, regardless of which program semantics we use to obtain the formulas to be proved, our formal theory and mechanical theorem-prover must permit definition and proof by induction. This book is about such a theory and a mechanical theorem-prover for it.

B. OUR FORMAL THEORY

We will present a logical theory that we have tailored to the needs of thinking about computer programs. It provides for the introduction of new "types" of objects, a general principle of induction on well-founded relations (Noetherian Induction [6]), and a principle permitting the definition of recursive functions. Recursive functions offer

[1] See Manna and Waldinger [31] for a description of the many other ways that formal reasoning can be usefully applied in computer programming.

such a powerful form of expression when dealing with discrete mathematics (such as underlies computer programs) that we do not use any additional form of quantification.[2]

C. PROOF TECHNIQUES

After defining our formal theory, we describe many techniques we have developed for proving theorems in it. We devote eleven chapters to the description of these techniques and how, when, and where they should be applied to prove theorems. The most important of these techniques is the use of induction. The formulation of an induction argument for a conjecture is based on an analysis of the recursive definitions of the concepts involved in the conjecture. Thus the use of recursively defined functions facilitates proving theorems about inductively defined objects. Many of the other proof techniques are designed to support our induction heuristics.

D. EXAMPLES

All the techniques are illustrated with examples. Most of our techniques are first illustrated with simple theorems about functions on lists and trees. These elementary functions are simple to define and are worth knowing if one is interested in mechanical theorem-proving (as we assume many readers will be). In addition, it is more fun to work through the proofs of novel theorems than through the proofs of, say, the familiar theorems of elementary number theory.

We have also included four complicated examples, chosen from several different subject domains, to illustrate the general applicability of the theory and our proof methods.

In the first such example, we write a tautology-checker as a recursive function on trees representing formulas in propositional calculus. We exercise the theory and proof techniques in an interesting way by stating and proving that the tautology-checker always returns an answer, recognizes only tautologies, and recognizes all tautologies.

[2] The program of using recursive functions and induction to understand computer programs, and the use of computers to aid the generation of the proofs, were begun by McCarthy [33, 34]. See also Burstall [12]. The idea of using recursive functions and induction but no other form of quantification in the foundations of mathematics (or at least of arithmetic) was first presented by Skolem in 1923 [52]. See also Goodstein [22].