PROGRAMMING LOGICS

AN INTRODUCTION TO VERIFICATION AND SEMANTICS

Raymond D. Gumb

PROGRAMMING LOGICS

AN INTRODUCTION TO VERIFICATION AND SEMANTICS

Raymond D. Gumb

John Wiley & Sons

New York Chichester Brisbane Toronto Singapore

Library of Congress Cataloging in Publication Data:

Gumb Raymond D
Programming logics An Introduction to Verification and Semantics
by Raymond D Gumb
p cm

Includes indexes
ISBN 0-471-60539-5
1 Computer programs—Verification 2 Programming languages (Electronic computers)—Semantics I Title
QA76 76 V47G86 1989
005 1'4—dc19

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Preface

This text is intended for a first course in program verification systems, and consistent and complementary definitions of the semantics of programming languages. In order of emphasis, it treats axiomatic, operational, translational, and denotational semantics. Although program proving is covered extensively, this text also shows that the different semantics are appropriately related. Finally, because of its focus on Hoare-style Axiomatizations and semantics of **while** programs, this text serves as an introduction to programming (or Hoare) logics.

I began writing this text after being drafted to teach advanced undergraduate and graduate-level courses in program verification and the semantics of programming languages, a subject for which there were no adequate introductory texts. Excellent advanced texts existed, but they were too difficult for the graduate students who had not taken the undergraduate course. Because most computer science students enjoy programming, I feel that an introductory text should emphasize program proving and suggest relevant programming projects. At the same time, students need to realize that verification systems and semantics are not self-evident formalisms to the *cognoscenti* that should be applied uncritically. Consequently, this text highlights the importance of consistent and complementary definitions of programming

language semantics by, for example, stressing the role of soundness proofs in vindicating verification systems. Furthermore, relevant aspects of first-order logic and Peano arithmetic are accented here for two reasons: first, they are needed in program proving, and second, they facilitate the explication of key semantic concepts (e.g., the weakest precondition).[1] A novel feature of this text is its use of "free" logic and arithmetic to handle run-time errors in the final two chapters.

Advanced undergraduates and beginning graduate students in computer science at three universities have used drafts of this text. Professional programmers should also find it suitable for independent study (without formal classroom instruction). The text presumes that the student has some background in concepts of programming languages and in discrete structures (in particular, sets, relations, functions, mathematical induction, first-order logic, formal proofs, and the Peano axioms for the natural numbers). However, Chapter 0 reviews the basic mathematical tools used in the text and, in general, the text covers preliminary material in enough detail so that it should be comprehensible to a diligent reader with little background in some of these areas. Students who have mastered the material in this text should be prepared for a second course using one of the more advanced texts listed in *Suggestions for Further Reading*.

When feasible, the text follows standard notational conventions as represented in de Bakker's excellent advanced text. And, to make semantic analysis tractable in this introductory text, we impose a number of restrictions on the syntax of our programming languages. Our approach is to introduce a minilanguage in each chapter to simplify treatment of new concepts. Although we deal primarily with extensions of the class of **while** programs, represented by dialects of Pascal (Chapters 1 and 4 through 6), we also treat flowchart programs, represented by dialects of BASIC (Chapters 2 and 3). Despite the fact that BASIC's **goto** command complicates semantic description, so do the restricted transfer of control statements advocated by proponents of program intelligibility. In program verification today, the methods of Floyd remain a viable alternative to Hoare Axiomatizations, and flowchart programs provide a convenient framework for developing the power and simplicity of Floyd's methods. Furthermore, the commands of flowchart programs are more similar to the instruction sets of conventional machines than the statements of **while** programs, and flowchart programs are the "meanings" of **while** programs under the translational semantics presented in Chapter 4.

Figure P.1 presents a checklist of chapter contents. To avoid developing the intricacies of many-sorted logic, only integer-type variables are used.

[1] Applications of the Consequence Rule in programming (Hoare) logic are justified, in part, by proofs in arithmetic. In Chapter 5, the symbiosis of arithmetic and programming logic is most perfect: Proving programmer-defined functions correct in programming logic justifies the introduction of function-call axioms into arithmetic.

The sections numbered 0 in Chapters 5 and 6 introduce two critical mathematical concepts—free arithmetic and concepts of denotational semantics that are essential for understanding subsequent sections.

In Chapter 0, it is important for students to work through enough natural deduction proofs and proofs in arithmetic to develop an intuition of when, in the Hoare Axiomatization, the Consequence Rule can be applied and, in the Floyd Method, when a verification condition holds. Furthermore, by working through some formal proofs in arithmetic, students will come to appreciate the fact that a natural deduction system is truth-preserving, as contrasted with the Hoare Axiomatization of Chapter 1, which is only validity-preserving. Chapter 0 can be reviewed in about a week by mathematically mature students. In Chapter 1, it is essential for students to work through at least three or four program-proving exercises for them to understand the Hoare Axiomatization. Because program proofs in the formalism of Chapter 1 can be rather long, I generally postpone the proofs of more intricate programs until Chapter 2. And, as students will learn, program-proving can be fun! In Chapters 3 and 4, more interesting algorithms such as sort programs can be proven, but I generally emphasize finding loop invariants and semantic issues because thorough program proofs are long and complex. Students should work through one or two of the program-proving exercises in enough detail to appreciate the aspects of the minilanguage giving rise to these complexities (e.g., assignments to subscripted variables), as these aspects are obstacles to practical program proving. Either program proving or semantic issues can be emphasized in Chapters 5 and 6. Many students will be motivated by the programming exercises. I usually do not assign more than one programming exercise in a semester because meaningful programming exercises tend to be long.

I recommend working through the chapters in the order they are presented. If there is not enough time in a one-semester course to cover all seven chapters and you wish to spend some time on subprograms and rudimentary denotational semantics (Chapters 5 and 6), you can cover lightly—or even skip altogether—Chapters 3 and 4. If you wish to skim over the more mathematical material, it can be found in Sections 0.2, 0.3, 0.5, and 0.6, Section 1.4, Section 2.4, Sections 3.4 and 3.6, Section 4.3, Sections 5.0 and 5.4, and Sections 6.0, 6.5, and 6.6.

I wish to thank Russ Abbott, Paul Kenison, Hugues Leblanc, Donald Martin, Edward Smith, Ivan Sudburough, George Weaver, Anita Gleason, Paul Lee, Faith Lin, Phil Mahler, Paul Mayer, Ridge McGhee, Kevin Meier, and Weidong Wang for their comments on earlier versions of this text. Faith Lin created the subject index and assisted in proofreading.

Tyngsboro, Massachusetts
June 1988

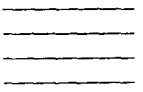

Figure P.1 Checklist of Topics by Chapter

CHAPTER 0

First-order logic, natural deduction, axiomatization and the intended semantics of the first-order theory of the integers (Peano arithmetic extended to the negative integers), properties of relations (e.g., orderings), and Nötherian induction.

CHAPTER 1

While programs in a Pascal dialect, operational semantics (Cook – de Bakker style), the concepts of partial and total correctness, Hoare Partial Correctness Axiomatization, and soundness.

CHAPTER 2

Flowchart programs in a BASIC dialect, operational semantics, the Floyd Method (inductive assertions for proving partial correctness and well-founded sets for proving termination), and soundness.

CHAPTER 3

Extension of the BASIC dialect in Chapter 2 to cover arrays and sequential input/output.

CHAPTER 4

Extension of the Pascal dialect in Chapter 1 to cover arrays, stacks, and input/output. Hoare Total Correctness Axiomatization, correctness of a translation of the new dialect of Pascal into the BASIC dialect of Chapter 3, comparison of the Hoare and Floyd Methods, and difficulties expressing specifications in first-order languages.

CHAPTER 5

Free logic for handling run-time errors, extension of the Pascal dialect in Chapter 1 to cover nonrecursive functions and procedures, environments, static scoping, call-by-value and call-by-reference, Hoare Total Correctness Axiomatization, and soundness.

CHAPTER 6

Rudimentary concepts of denotational semantics, modification of the Pascal dialect in Chapter 1 yielding a class of tail recursive procedures, Hoare Total Correctness Axiomatization, correctness of a translation of tail recursive programs into **while** programs, equivalence of the denotational and operational semantics, weakest preconditions, and relative completeness.

Contents

Introduction 1

CHAPTER 0

Mathematical Preliminaries 7

0.1 Syntax 8

0.2 Natural Deduction 13

0.3 The Theory of Integers: Axiomatization 24

0.4 The Theory of Integers: Semantics 34

0.5 Properties of Relations 39

0.6 Nötherian Induction 42

Exercises 49

CHAPTER 1

The Partial Correctness of **while** Programs 54

1.1 Syntax 55

1.2 Operational Semantics 60
1.3 Proving Partial Correctness: The Hoare Axiomatization 64
1.4 The Soundness of the Axiomatization 70
1.5 Some Sample Proofs 75
Exercises 84

CHAPTER 2
The Total Correctness of Flowchart Programs 90

2.1 Syntax 91
2.2 Operational Semantics 93
2.3 Floyd's Method for Proving Program Correctness 96
2.4 The Soundness of Floyd's Method 103
2.5 Some Sample Proofs 109
Exercises 120

CHAPTER 3
The Total Correctness of Flowchart Programs with Arrays and Input and Output 128

3.1 Syntax 129
3.2 Operational Semantics 132
3.3 Proving Total Correctness 135
3.4 Soundness 142
3.5 Some Sample Proofs 146
3.6 Comparison of First-Order and Second-Order Specifications 158
Exercises 162

CHAPTER 4
The Translation of **while** Programs with Arrays, Input and Output, and a Stack into Flowchart Programs 171

4.1 Syntax, Operational Semantics, and Total Correctness Axiomatization 173
4.2 The Translation 180
4.3 Correctness of the Translation 184
4.4 Comparison of the Hoare Axiomatization and the Floyd Method 189
Exercises 194

CHAPTER 5
The Total Correctness of **while** Programs with Functions and Procedures 200

5.0 Free Arithmetic: Natural Deduction, Axiomatization, and Semantics 202
5.1 Syntax 218
5.2 Operational Semantics: States and Environments 224
5.3 Proving Total Correctness 234
5.4 Soundness 244
5.5 Some Sample Proofs 251
Exercises 255

CHAPTER 6
The Translation of Tail Recursive Procedures into **while** Programs 261

6.0 Complete Partial Orders 263
6.1 Syntax 268
6.2 Denotational Semantics 271
6.3 Proving Total Correctness 279
6.4 The Translation 284
6.5 Correctness of the Translation 289
6.6 Comparison of the Hoare Axiomatization and the Operational and Denotational Semantics 295
Exercises 302

Suggestions for Further Reading 309

Index of Definitions of Arithmetic 323
Index of Lemmas, Propositions, and Theorems 325
Index of Notations 317
Subject Index 329

Introduction

Some time ago, while working as a programming consultant, I was asked to implement an algorithm for scheduling plant shutdown times. The specification for the "algorithm" had been written by an accountant in an informal, but seemingly thorough, manner. I developed a program and, after considerable program testing, turned it over to the plant's computer center for production runs. A few months later, I received a frantic call. My program had begun assigning plant workers negative vacation times, which, needless to say, the workers did not appreciate! After studying the unfortunate runs for some time, I discovered that the problem lay in the specification. It became apparent that no program could be written for the specification. That is, the specification was unsatisfiable. When I pointed out the problem to the accountant, he simply replied that "the world is filled with inconsistencies." At first, I was impressed by his response, but not for long, as higher-level management soon made it clear that they did not want any of the world's inconsistencies running around in their scheduling programs.

Computer users in government and industry constantly are demanding more reliable computer systems. Part of this demand is for fault-tolerant systems that will continue to perform *correctly* when components fail or faults occur. But what does it mean to speak of a system, consisting of

hardware and programs (software and firmware), "performing correctly"? In this text, we study this question in the case of programs. We analyze what it means for a program to be correct, and present verification techniques for proving programs correct. For as my accountant friend and I discovered at the expense of the plant workers, testing a program does not always establish its correctness; crucial test cases may be overlooked or there may be too many cases to test. Program correctness must be understood in the context of a mathematical theory.

No user wants an incorrect program, but some users cannot afford even a single "bug" in their programs, as program failures could result in the loss of life and expensive resources. To that end, for example, the Department of Defense Security Center has developed standards for evaluating the security features of commercially available computer systems. For a computer system to receive the highest security rating, the tools of formal logic must be used in establishing the correctness of the system. The reliability of the "Star Wars" system, for example, hinges in part on the possibility of proving that programs meet their specifications. Another government agency has funded development of a provably fault-tolerant computer system to control future commercial aircraft that will be inherently unstable to improve fuel efficiency. Similar efforts in program verification are contemplated in England and other countries. Like it or not, ready or not, program verification is with us. Soon we will see demands for provably correct computer systems for controlling nuclear power plants, life support equipment in hospitals, electronic funds transfer systems, and other critical applications.

Program correctness is a semantic concept. In keeping with the tradition of mathematical semantics developed by Alfred Lindenbaum, Alfred Tarski, Leon Henkin, Saul Kripke, Dana Scott, Christopher Strachey, and others, we understand semantics to give meaning to linguistic expressions by assigning them denotations. Typical denotations are an object (an integer), a set of objects (the even integers), and a set of ordered pairs of objects (the greater than relation $>$ on the integers), and so forth. In the study of logics, one typically distinguishes between syntax, semantics, and a deductive system. In the case of a programming language, one typically specifies the context-free aspects of the syntax in terms of a BNF grammar. The semantics assigns denotations to the syntactic objects specified by the grammar. For example, the semantics assigns to an integer expression (a syntactic or linguistic object) an integer (a mathematical, nonlinguistic object), and it assigns to an assertion (a syntactic object) a truth-value (true or false) as a denotation. The deductive system (verification system) provides a means of arranging assertions into proofs. In other words, the deductive system lets us prove (verify) that our programs do what they were intended to do (as formalized in specifications, which we discuss in the following paragraphs).

In programming language semantics, many writers distinguish between "axiomatic" semantics (discovered by Anthony Hoare), "translational" semantics, "operational" semantics (developed, as we shall study it, by Ste-

phen Cook and Jaco de Bakker), and "denotational" semantics (discovered by Dana Scott and Christopher Strachey). Roughly, these semantics may be characterized as follows. The axiomatic semantics is a program verification system consisting of axioms and rules of inference. The translational semantics is provided by a programming language translator that specifies a mapping of a source program to a target program (the meaning of the source program). The operational semantics is given by a description of the sequence of machine states passed through during program execution. The denotational semantics is more abstract, as it denotes the meaning of a program in the form of a mapping from machine state to machine state.

These different semantics are said to be "complementary" because they can serve different purposes, and they are said to be "consistent" because they can define one and the same programming language. Complementary semantics are needed because many writers believe that the fundamental semantics is denotational semantics for the language designer, operational semantics for the language implementor developing an interpreter, translational semantics for the implementor developing a compiler, and axiomatic semantics for the language user writing reliable programs. The different semantics must be mutually consistent (or equivalent in some sense) if the implementors are to realize properly the language design and the users are to write software correctly. To illustrate the point on consistency in the extreme, consider the absurdity of selecting a Pascal compiler to translate LISP (as opposed to Pascal) source code.

The terminology used in the last paragraph underscores the importance of different semantic definitions being consistent and complementary, but it also has some problems. The terms "axiomatic semantics" and "translational semantics" are somewhat out of keeping with existing mathematical practice, because axiomatic semantics does not (directly) assign denotations to programs, and translational semantics does not assign the usual sort of mathematical entities (set theoretic objects). The "operational"–"denotational" terminology is somewhat misleading as the operational semantics as well as the denotational semantics can deliver denotations for syntactic expressions. In fact, whenever both the operational and denotational semantics assign a denotation to the same syntactic object, they must deliver the same denotation (or denotations that are equivalent in some suitable sense). If the two semantics do not deliver the same denotation, something is seriously wrong with one or both of them. The difference between the two semantics is in how the denotations are "delivered." In particular, the meaning of a program is a function mapping the set of states (memory configurations) into itself. The operational semantics is described in terms of a machine executing program statements, leaving a trace or sequence of machine states. Roughly, it specifies the meaning of a program point-wise: given an initial state, the meaning of a program is the state (if it exists) in which the program terminates. The denotational semantics abstracts from machine implementations, describing the semantics in terms of various mathematical

constructions (e.g., the least fixed points of appropriate functions). If we restrict our attention to nonrecursive programs, the only structured programming construct for which the operational and denotational semantics differ substantially is the **while** statement. We shall examine that difference in Chapter 6.

Program correctness, as we shall study it, is a relativized concept because it is defined in terms of a specification and we will presume that any given specification is correct. A specification consists of a pair of assertions—the precondition (or input assertion) and the postcondition (or output assertion). The precondition describes the permissible values of variables when program execution begins, and the postcondition prescribes the values variables must have when the program terminates. A program is totally correct with respect to a specification consisting of a precondition p and a postcondition q provided that, if p is true when execution begins, then the program *will* terminate and q will be true on termination. The program is partially correct provided that, if p is true when execution begins and *if* the program terminates, then q will be true on termination. Total correctness is the more natural concept in most contexts. However, partial correctness is usually easier to work with, and many important works on program correctness have dealt with it exclusively. We shall define these two concepts of program correctness more rigorously in the chapters that follow.

A deductive or verification system is said to be sound if no incorrect program can be "proven" correct. Beginning in Chapter 1, we will prove the soundness of most of our verification systems immediately after presenting them. We do this to make sure that each verification system contains no "bugs" as well as to fortify your intuitions with the sense of what makes each verification system tick. A deductive system is also said to be complete if every correct program can be proven correct. As we shall see, our verification systems are not complete in this sense, but most of them are complete in a weaker, relative sense. A verification system is called "relatively complete" if the programming part of the verification system does its work fully. The incompleteness of arithmetic (which Kurt Gödel proved in 1931) is entirely responsible for the incompleteness of relatively complete verification systems. The relationships between syntax, operational and denotational semantics, and verification system are depicted in Figure I.1.

A verification system must be proven correct with respect to a semantics —operational or denotational—for it to be trustworthy. An unsound verification system is at best useless and at worst dangerous, for in it one can prove falsehoods. Suppose, for instance, that you were given a verification system in which you could write down (prove) any assertion you choose. You could prove many properties about programs or whatever. You could prove, say, that your very first program gave a constructive proof of Fermat's Last Theorem, that your program established that $x=x+1$, or that (assuming an appropriate encoding of assertions into the language of arithmetic) the moon was made of green cheese. You could prove many things, but that certainly

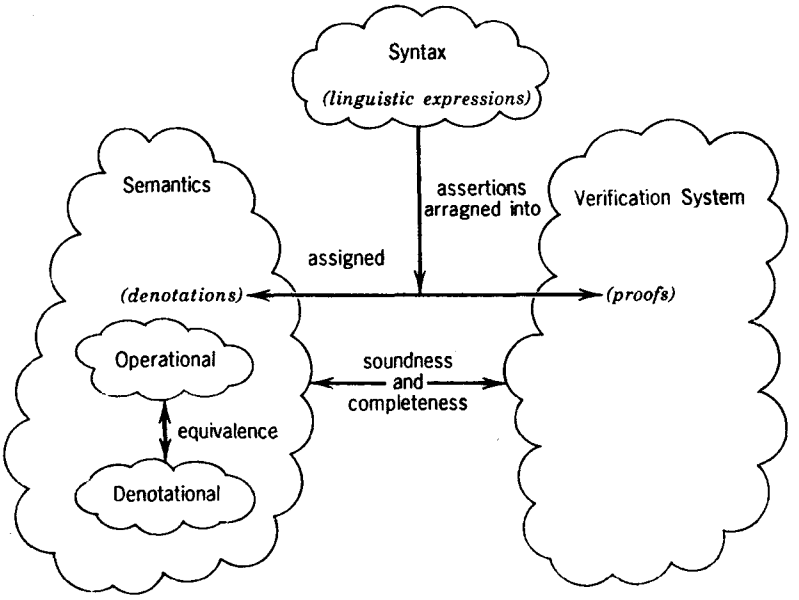

FIGURE 1.1. Relationships between syntax, semantics, and verification systems.

would not mean that they were true. You would have been swept away by the *runaway inference ticket* that licenses one to prove anything.

You may say that this is all very silly, that you would never use an unsound verification system. In the unfortunate history of program verification technology, however, many systems for specifying and verifying programs have been developed without a mathematical semantics. And although these systems have been used to "prove" programs correct, many of these systems themselves have not been correct. As a result, incorrect programs have been "proven" correct. And, as a case in point, technical papers on the Assignment Axiom for arrays were, for years, filled with errors. (We shall explore some of the intricacies of the Axiom of Assignment for arrays in Chapter 3.)

Unsound verification practices can surface anywhere. For instance, in the late 1970s, an office of the Department of Defense funded the development of the programming language EUCLID, which was intended to facilitate the development of programs that could be easily verified. The designers of EUCLID did develop proof rules, but they neglected to provide a semantics. In the end, they were forced to admit that they could not vouch for the correctness of their verification system. Indeed, they could not show correctness without first providing a semantics, for correctness is a semantic con-

cept. In another instance, as recently as 1983, the Department of Defense Security Center's evaluation criteria for secure computer systems included only one sentence about semantics, and nothing about the correctness of specification and verification systems.

In my opinion, program verification and semantics are most worthy of study as mathematical aspects of computer science. Further, I am convinced that study of these subjects has practical import. At the least, some basic understanding of them is required to understand the advanced literature on programming languages as well as the ongoing debate on the practicality of verification. More importantly, their study enhances one's ability to reason informally about programs and write correct programs. Furthermore, their study may suggest criteria for evaluating and designing programming languages if, as I believe, only programming languages with reasonable formal verification systems and semantics can be used to develop reliable software.

I do not believe, however, that large-scale program verification, as demanded by some of the systems mentioned previously, is practical in the near future. As will be seen in the following chapters, verification of even moderately realistic programs is complicated. Due to results in undecidability and complexity theory, we know that mechanical theorem-proving techniques are of limited assistance, and that we must depend on human-oriented proof techniques such as the natural deduction system presented in Chapter 0. There are also technical problems in expressing certain specifications, both of a formal and an informal nature. The formal problems are touched upon in Chapter 3. The informal problems relate to the difficulty of spelling out a user's needs in a formal specification, as might be gleaned from the scheduling algorithm fiasco mentioned at the beginning of this introduction. A formal specification may not capture the user's informally stated objectives, and, even if it does, the user's articulated objectives may be misguided, immoral, or just plain wrong.

On the other hand, I am not convinced that program verification will never become a practical activity. Some events that would make program proving more practical are basic discoveries in the mathematical theory of definitions, acceptance of semantic standards for programming language designs, development of software tools to assist humans in program proofs, and initiation of training programs for "proof engineers." I make no prediction on the practicality of program verification in the twenty-first century.