Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

130

Robert Goldblatt

Axiomatising the Logic of Computer Programming

Springer-Verlag
Berlin Heidelberg New York

TP31
G618

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

130

Robert Goldblatt

Axiomatising the Logic of Computer Programming

Springer-Verlag
Berlin Heidelberg New York 1982

Author
Robert Goldblatt
Department of Mathematics
Victoria University, Private Bag
Wellington, New Zealand

CR Subject Classifications (1981): 5.21, 5.24

ISBN 3-540-11210-3 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-11210-3 Springer-Verlag New York Heidelberg Berlin

Library of Congress Cataloging in Publication Data
Goldblatt, Robert. Axiomatising the logic of computer programming. (Lecture notes in computer science; 130) Bibliography: p. Includes index. 1. Programming languages (Electronic computers)– Semantics. 2. Proof theory. I. Title. II. Series.
QA76.7G65 519.4 82-863 ISBN 0-387-11210-3 (U.S.) AACR2

Printing and binding: Beltz Offsetdruck, Hemsbach/Bergstr.
2145/3140-543210

To Helen,

and to Jed and Hannah

PREFACE

This is a small step for Computer Science: a step towards a systematic proof-theory for programming-language semantics. We study a language that is designed to formalise assertions about how programs behave. In this language each program determines a modal connective that has the meaning "*after the program terminates* ...". Such connectives appear in the "algorithmic logic" of A. Salwicki at Warsaw, but the explicit use of techniques from modal logic in this area was initiated more recently by V.R. Pratt at M.I.T., and has become known as "dynamic logic". It is to the latter that the present work is directly related.

Our approach contains a number of distinctive features. Notably, a contrast is made between *external* and *internal* logic : between the operations performed by the programmer in reasoning about program behaviour, and those performed by the computer in evaluating Boolean expressions. The programmer's external propositional logic is the classical two-valued one, while the computer may sometimes leave certain expressions undefined - e.g. if their evaluation fails to terminate. This leads us to a three-valued model of computer logic, based on a "sequential" interpretation of Boolean connectives. (This is not claimed to be the official model, but it is a natural interpretation, and readers interested in others are encouraged to adapt our techniques to cater for them.) The external language includes the internal one (the programmer can talk about the machine, but not conversely), and so the presence of undefined expressions has implications for the programmer's *quantificational* logic. The version used here is a variant of "logic without existence assumptions": it accommodates the possibility that the value of a quantifiable variable may not exist.

The general purpose of this book is to establish a methodological framework for proof-theory and axiomatisation. Within that, our central aim is to analyse the operation of assigning a value to a program variable. This is the most basic of commands, and - although representable as a dynamic form of logical substitution - is the fundamental departure that takes computation theory beyond the traditional province of mathematical logic. In Part I a complete axiomatisation is developed of the class of valid assertions about programs of the following kinds :

assignments	($x := \sigma$)
composites	(compound statements)
conditionals	(*if-then-else*)
iterations	(*while-do*)
alternations	(non-deterministic choice).

This would appear to provide an adequate formalisation of the system used by E.W. Dijkstra in his well-known book *A Discipline of Programming*. Moreover it is known that by using all except the last of these concepts a program can be written to compute each partial recursive function. Hence, by Church's Thesis, this language is *in theory* as powerful as can be: it contains programs for all possible algorithms. But of course in comparison to real programming languages it is extremely limited. Its relationship to the latter is perhaps comparable to the relationship between Turing machines and actual computers. Just as Turing machines are crucial to a theoretical understanding of the nature of algorithms, the above constructs are crucial to a theoretical understanding of the structure of programs (and structured programming).

However, an adequate semantical theory must eventually be applicable to the concepts and devices found in actual programming practise, and so in Part II we begin to move in this direction. We study function declarations, procedure calls, and the syntactic and semantic roles of the indexed variables used to denote components of arrays. This enables us to investigate the various proof rules that have been proposed by C.A.R. Hoare for such notions, and to develop an analysis of the parameter-passing mechanisms of call-by-value, call-by-name, and call-by-reference.

In the more abstract realms of mathematical thought it is sometimes possible for a person to single-handedly exhaust the investigation of a particular topic, and then produce the definitive account of it. Programming-language semantics is not like that. It is an inherently open-ended subject that depends on the perspectives and ideas of many contributors for its development. Its character is as much that of an empirical study as that of an intellectually creative one: it uses mathematics to model real-world phenomena that are produced in response to practical need as much as theoretical principle. An appropriate analogy is with the linguistics of natural languages - no-one would claim to have had the final say about the semantics of English.

In such disciplines it is often necessary to produce an exposition of the current state-of-the-art in order to stand back, evaluate, and thereby move on to new understandings. This book should be seen as a stage in such a process. Its object, as the title is intended to convey, is to pursue the problem of proof-theoretically generating *all* the valid assertions about programs in a given language. Its major contributions in this regard can be seen as

(1) the adaptation to quantificational programming logics of the methodology of "Henkin-style" completeness proofs via canonical model constructions; and

(2) the analysis of *while*-commands in terms of an infinitary rule-schema. The techniques and ideas used originate in the mathematical studies of intensional logics that have taken place in the two decades or so since the advent of "Kripke semantics". Thus the work may well be of interest to logicians who are unfamiliar with computer science, as well as to computer scientists who have little background in formal logic. For this reason an initial chapter is provided that gives an informal overview of the necessary conceptual background. But it should be understood that the text does not purport to provide an exposition of the general study of Programming Logics. It is simply an individual contribution to an aspect of that discipline, and as such is not unlike a large research paper. In an appendix to Part I, a survey is given of works by others, but this is little more than an annotated bibliography: its purpose is to lend perspective and context to the present work, and to point the reader in some appropriate directions. By pursuing these references s/he will become aware of the numerous important contributions that have not been cited here.

This typescript has been prepared by Shelley Carlyle, to whom the author is indebted once again for her expertise and cooperation. The cost of preparation was generously subsidised by a grant from the Internal Research Committee of the Victoria University of Wellington.

CONTENTS

PART I FOUNDATIONS

It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and mathematical elegance.

John McCarthy, 1963.

CHAPTER 1

CONCEPTUAL BACKGROUND

1.1 INTERNAL AND EXTERNAL LOGIC

In the following pages, techniques and ideas from mathematical logic are applied to an aspect of computer science. Our concern is to analyse and formalise the patterns of thought that are used in reasoning about the behaviour of computers and the algorithms that they process. A formal system will be developed on the basis of a distinction between two kinds of logical activity. On the one hand we have the logical operations performed by the computer itself when it calculates the truth-value of certain basic expressions, in order to thereby determine its next action. This will be called *internal* logic. It is, for example, the logic of the expression ε in a command of the form

$$\textit{while } \varepsilon \textit{ do } \alpha.$$

External logic, on the other hand, is concerned with the structure of assertions *about* programs and the effects of their execution. When people write programs, they have in mind certain tasks that are to be carried out. Such an intention might be expressed by a programmer in a sentence of the form

(1) *when program* α *terminates,* ψ *will be true,*

or, more generally,

(2) *if the assertion* φ *is true before initiation of program* α*,*
then the assertion ψ *will be true on its completion.*

For instance, if α is intended to find the remainder r upon division of x by y (where x and y denote natural numbers), then we might express this by taking

φ as

$$0 < y$$

and ψ as

$$\exists q(x = q \times y + r) \wedge (r < y),$$

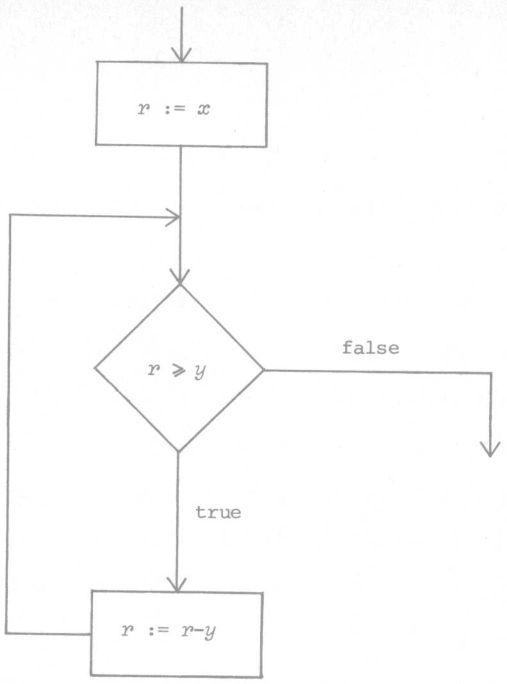
where $\exists q$ is the existential quantifier "for some q" and $\wedge$ is the logical connective "and".

Assertions of the type (1) and (2) belong to external logic, which itself includes, and is intermeshed with, the internal logic. Both aspects are represented within the formal language that will be the object of our investigations. But in addition to this we have an informal *metalanguage* (the language in which this book is written) that is used to make assertions about the formal language (e.g. that certain sentences are provable/imply other sentences/have no models etc.), and so there are all told *three* levels of logical activity. This is in marked contrast to the situation in traditional mathematical logic, where generally we emphasise only two levels - metalanguage and object language, the latter being designed to formally express the properties of conventional algebraic operations. The reason for the extra dimension here is that whereas traditional logic attempts to describe the static properties that these operations have, in the present context we wish to study their dynamic *performance*, in time, by some computing agent. In this introductory chapter we will sketch the historical background to this kind of study, and outline the main conceptual features of the theory to be developed in the chapters to follow.

1.2 CORRECTNESS AND PROOF

How can the programmer be sure that the algorithm he devises actually does what he wants it to do? This is known as the problem of program *correctness*. Consider, for example, the flowchart (3), in which x, y, r are natural-number-valued variables.

(3)



First of all we may ask "what operation does this flowchart perform?" Given an answer, we may then ask for a rigorous mathematical proof that it is the *correct* answer. But then the interrogation could be pushed even deeper, by demanding an explication of the meaning of "rigorous proof".

In fact this flowchart computes the remainder upon division of x by y, and to an experienced computer-scientist this answer may spring off the page almost immediately. We could then inquire as to what aspects of his knowledge and experience allowed him to make this conclusion.

On the other hand, to people unfamiliar with the graphical notation of (3) we would first have to explain what command boxes and decision diamonds are, and perhaps what the symbol := means in an assignment command. Once this has been grasped, it is a matter of their understanding that the particular series of operations specified do indeed lead to the remainder upon division. Of course if they are not sure what "remainder" and "division" mean then we may still have a long way to go towards a proof that is convincing *for them*.

It becomes apparent that the notion of an adequate proof is relative to the person who is trying to grasp it. At the least we can say that a proof, by its nature, consists in a demonstration that a certain fact depends upon, or can be reduced to, certain other facts. The latter may in turn require proving, and so on. But as far as programmers are concerned, we can reasonably claim that in order to be able to carry out their function properly they must understand the characteristic properties of the data types that they write programs for. These structures, initially the integers, the real numbers, and the two-element Boolean algebra, are just the sort of thing that traditional mathematical logic has been developed to analyse. Thus an adequate correctness proof for a program might well consist in the reduction of an assertion to certain facts about data types, with the standard machinery of logic being invoked to provide "proofs" of these later facts.

The kind of standard logical machinery for data types that we have in mind here is called *elementary* ("of elements") or *first-order* logic. Its statements ("well-formed formulae") are built up from an alphabet that comprises

(i) variables $x,y,z,\ldots$ whose values are the individual elements of some data type;

(ii) symbols that denote operations on these elements, e.g. the operations $+$, $\times$, $-$, $\div$ on numbers : the sort of operation that a computer performs;

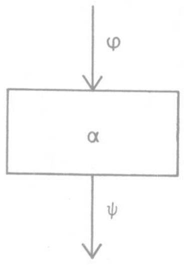
(iii) symbols for relations between, or properties (predicates) of, individuals, e.g. the relations $=$, $\neq$, $<$, $>$, $\leqslant$, and $\geqslant$: the sort of relations whose truth-values are tested by a computer in applying its internal logic;

(iv) the statement-forming connectives $\wedge$ ("and"), $\vee$ ("or"), $\neg$ ("not"), $\rightarrow$ ("implies"), $\leftrightarrow$ ("if and only if");

(v) the universal quantifier $\forall$ ("for all"), and the existential quantifier $\exists$ ("for some").

The name "first-order" refers to the fact that the variables x, y,... that may be quantified denote basic *elements* of the data type, and not any "higher-order" entities such as sets of elements, sets of sets of elements, etc.

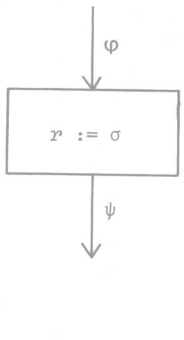
A technique for proving correctness of flowcharts was first developed by R.W. Floyd (1967). (The essential idea appears to have been suggested by von Neumann and Goldstine in 1946 (cf. von Neuman (1963)) and Turing (1949), while a similar notion was independently studied by Naur (1966)). Floyd's method can be understood as follows. A flowchart could be a complex diagram, built up from simpler flowcharts by various constructions, but it will always be assumed to have a unique entry arrow and a unique exit arrow. An *interpretation* of such a flowchart is made by attaching assertions to its entry and exit arrows. The interpretation



is called *correct* if the assertion ψ is true when the computation process reaches the exit point of flowchart α, provided that φ was true when it reached the entry point to α. Thus correctness of this interpretation amounts to the truth of the statement (2) above. The method then requires the development of rules or conditions for correctness of interpretations, and the correctness of a particular algorithm is given by showing that the desired interpretation obeys these rules. We now consider four such conditions.

I. Rule of Assignment.

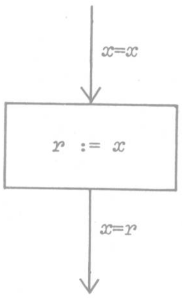
The interpretation



is correct if φ *is the assertion obtained by replacing all free occurrences of* r *in* ψ *by* σ.

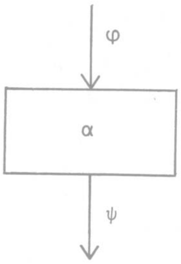
The idea behind this rule is that when the assignment has been completed, the variable r has the same value that the expression σ had beforehand, so that any statement that is true of (the value of) r afterwards must have been true of (the value of) σ to begin with. For example, the following is a correct interpretation of an assignment command.

(4)



II. Rule of Consequence.

This is a rule that allows us to form a new correct interpretation from a given one. It stipulates that *if*



is correct, and moreover

(i) φ' *implies* φ , *and*

(ii) ψ *implies* ψ',

then