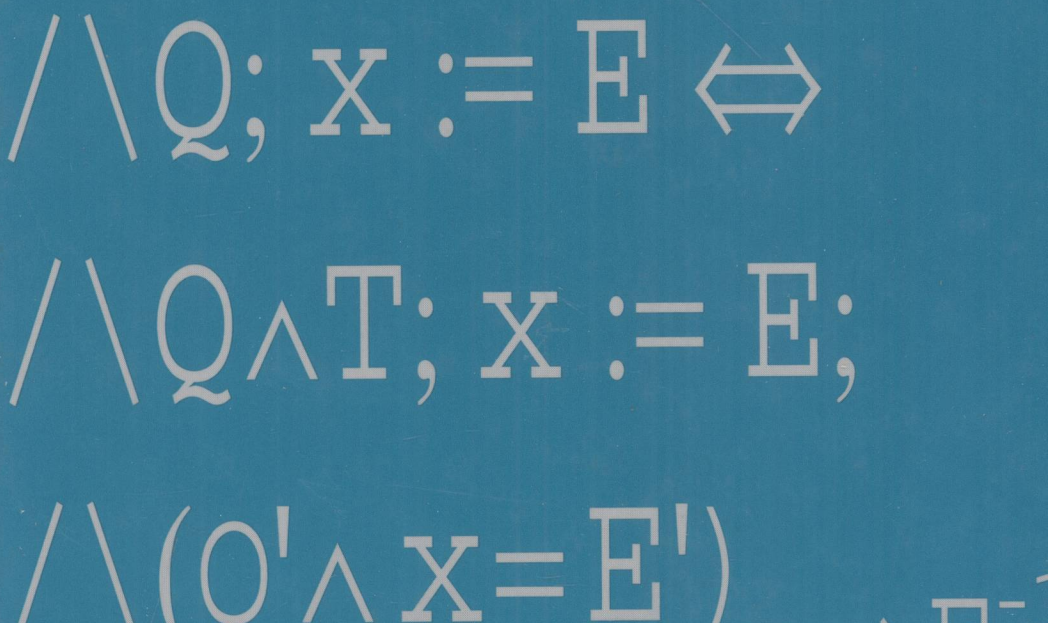
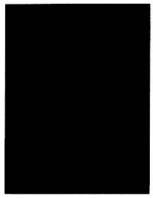
Path-Oriented Program Analysis

J. C. Huang



CAMBRIDGE



Path-Oriented Program Analysis

J. C. Huang
University of Houston, Houston, Texas

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo, Delhi

Cambridge University Press
32 Avenue of the Americas, New York, NY 10013-2473, USA

www.cambridge.org
Information on this title: www.cambridge.org/9780521882866

First published 2008

Printed in the United States of America

A catalog record for this publication is available from the British Library.

Library of Congress Cataloging in Publication Data

Huang, J. C., 1935–
Path-oriented program analysis / J. C. Huang
p. cm.
Includes bibliographical references and index.
ISBN-978-0-521-88286-6 (hardback)
1. Computer software – Development. 2. Computer software –
Development – Computer programs. I. Title.
QA76.76.D47H83 2008
005.1 – dc22 2007026404

ISBN 978-0-521-88286-6 hardback

Path-Oriented Program Analysis

This book presents a unique method for decomposing a computer program along its execution paths, for simplifying the subprograms so produced, and for recomposing a program from its subprograms. This method enables us to divide and conquer the complexity involved in understanding the computation performed by a program by decomposing it into a set of subprograms and then simplifying them to the furthest extent possible. The resulting simplified subprograms are generally more understandable than the original program as a whole. The method may also be used to simplify a piece of source code by following the path-oriented method of decomposition, simplification, and recomposition. The analysis may be carried out in such a way that the derivation of the analysis result constitutes a correctness proof. The method can be applied to any source code (or portion thereof) that prescribes the computation to be performed in terms of assignment statements, conditional statements, and loop constructs, regardless of the language or paradigm used.

J. C. Huang received a Ph.D. in electrical engineering from the University of Pennsylvania in 1969. He is a Professor Emeritus in the Department of Computer Science at the University of Houston, where he served as chair from 1992 to 1996.

His practical experience in computer software includes serving as the chief architect of a software validation and verification system developed for the U.S. Army's Ballistic Missile Defense Command, and as a senior consultant to the U.S. Naval Underwater Systems Center on submarine software problems.

To my wife

Preface

Many years ago, I was given the responsibility of leading a large software project. The aspect of the project that worried me the most was the correctness of the programs produced. Whenever a part of the product became suspect, I could not put my mind to rest until the product was tested successfully with a well-chosen set of test cases and until I was able to understand the source code in question clearly and completely. It was not always easy to understand. That was when I started to search for ways to facilitate program understanding.

A program can be difficult to understand for many reasons. The difficulty may stem, for example, from the reader's unfamiliarity with the application area, from the obscurity of the algorithm implemented, or from the complex logic used in organizing the source code. Given the different reasons that difficulty may arise, a single comprehensive solution to this problem may never be found.

I realized, however, that the creator of a program must always decompose the task to be performed by the program to the extent that it can be prescribed in terms of the programming language used. If the

reader could see exactly how the task was decomposed, the difficulty of understanding the code would be eased because the reader could separately process each subprogram, which would be smaller in size and complexity than the program as a whole.

The problem is that the decomposition scheme deployed in any program may not be immediately obvious from the program text. This is so because programmers use code sharing to make source code compact and avoid unnecessary repetition. Code sharing, together with certain syntactic constraints imposed by programming languages, tends to obscure the decomposition scheme embodied in any program. Some analysis is required to recover this information.

Mathematically speaking, there are three basic ways to decompose a function. The first way is to divide the computation to be performed into a sequence of smaller steps. The second way is to compute a function with many arguments in terms of functions of fewer arguments. The third way is to partition the input domain into a number of subdomains and prescribe the computation to be performed for each subdomain separately. Methods already exist to recover and exploit information relevant to the first two decomposition schemes: They are known as the techniques of symbolic execution and program slicing, respectively. This book presents an analysis method that allows us to extract, analyze, and exploit information relevant to the third decomposition scheme.

Do not be intimidated by the formalisms found in the text. The theorems and corollaries are simply rules designed to manipulate programs. To be precise and concise, formulas in first-order predicate calculus are used to describe the rules. Only elementary knowledge of symbolic logic is needed to interpret those rules.

Typically, the method described in this book is to be used as follows. The program in question is test-executed with an input. If the program produces an incorrect result, it is a definite indication that the program is faulty, and appropriate action must be taken to locate and correct the fault. On the other hand, if the program produces a correct result, one can conclude with certainty only that the program is correct for that particular input. One can broaden the significance of the test result, however, by finding the execution path traversed during the test-execution and then applying the analysis method presented in this book to determine

(1) the conditions under which the same path will be traversed, and (2) the exact nature of the computation performed during execution. This information about execution paths in the program can then be integrated to obtain a better understanding of the program as a whole. This method is illustrated in Appendix A with example programs in C++.

This book contains enough information for the reader to apply the method manually. Manual application of this method, however, is inevitably tedious and error prone. To use the method in a production environment, the method must be mechanized. Software tool designers will find the formal basis presented in this work useful in creating a detailed design.

Being able to understand programs written by others is of practical importance. It is a skill that is fundamental to anyone who reuses software or who is responsible for software quality assurance and beneficial to anyone who designs programs, because it allows designers to learn from others. It is a skill that is not easy to acquire. I am not aware of any academic institution that offers a course on the subject. Typically, students learn to understand programs by studying small examples found in programming textbooks, and they may never be challenged, while in school, to understand a real-world program. Indeed, I have often heard it said – and not only by students – that if a program is difficult to understand, it must be badly written and thus should be rewritten or discarded. Program analysis is normally covered in a course on compiler construction. The problem is that what is needed to make a compiler compile is not necessarily the same as what is needed to make a programmer understand. We need methods to facilitate program understanding. I hope that publication of this book will motivate further study on the subject.

I would like to take this opportunity to thank William E. Howden for his inspiration; Raymond T. Yeh for giving me many professional opportunities that allowed this method to develop from conception, through various stages of experimentation, and finally to the present state of maturity; and John L. Bear and Marc Garbey for giving me the time needed to complete the writing of this book. I would also like to thank Heather Bergman for seeking me out and encouraging me to publish this work and Pooja Jain for her able editorial assistance in getting the book produced. Finally, my heartfelt thanks go to my daughter, Joyce, for her

active and affectionate interest in my writing, and for her invaluable help in the use of the English language, and to my wife, Shihwen, for her support, and for allowing me to neglect her while getting this work done.

J. C. Huang
Houston

Contents

Contents

Introduction

Program analysis is a problem area concerned with methodical extraction of information from programs. It has attracted a great deal of attention from computer scientists since the inception of computer science as an academic discipline. Earlier research efforts were mostly motivated by problems encountered in compiler construction (Aho and Ullman, 1973). Subsequently, the problem area was expanded to include those that arise from development of computer-aided software engineering tools, such as the question of how to detect certain programming errors through static analysis (Fosdick and Osterweil, 1976).

By the mid-1980s, the scope of research in program analysis had greatly expanded to include, among others, investigation of problems in data-flow equations, type inference, and closure analysis. Each of these problem areas was regarded as a separate research domain with its own terminology, problems, and solutions.

Gradually, efforts to extend the methods started to emerge and to produce interesting results. It is now understood that those seemingly disparate problems are related, and we can gain much by studying them in a unified conceptual framework. As the result, there has been a dramatic

shift in the research directions in recent years. A great deal of research effort has been directed to investigate the possibilities of extending and combining existent results [see, e.g., Aiken (1999); Amtoft et al. (1999); Cousot and Cousot (1977); Flanagan and Qadeer (2003); and Jones and Nielson (1995)].

In the prevailing terminology, we can say that there are four major approaches to program analysis, viz., data-flow analysis, constraint-based analysis, abstract interpretation, and type-and-effect system (Nielson et al., 2005).

The definition of data-flow analysis appears to have been broadened considerably. In the classical sense, data-flow analysis is a process of collecting data-flow information about a program. Examples of data-flow information include facts about where a variable is assigned a value, where that value is used, and whether or not that value will be used again downstream. Compilers use such information to perform transformations like constant folding and dead-code elimination (Aho et al., 1986). In the recent publications one can now find updated definitions of data-flow analysis, such as "data-flow analysis computes its solutions over the paths in a control-flow graph" (Ammons and Larus, 1998) and the like.

Constraint-based analysis consists of two parts: constraint generation and constraint resolution. Constraint generation produces constraints from the program text. The constraints give a declarative specification of the desired information about the program. Constraint resolution then computes this desired information (Aiken, 1999).

Abstract interpretation is a theory of sound approximation of the semantics of computer programs. As aptly explained in the lecture note of Patrick Cousot at MIT, the concrete mathematical semantics of a program is an infinite mathematical object that is not computable. All nontrivial questions on concrete program semantics are undecidable.

A type system defines how a programming language classifies values and expressions into types, how it can manipulate those types, and how they interact. It can be used to detect certain kinds of errors during program development. A type-and-effect system builds on, and extends, the notion of types by incorporating behaviors that are able to track information flow in the presence of procedures, channels based on communication, and the dynamic creation of network topologies (Amtoft et al., 1999).

Introduction

This book presents a path-oriented method for program analysis. The property to be determined in this case is the computation performed by the program and prescribed in terms of assignment statements, conditional statements, and loop constructs. The method is path oriented[1] in that the desired information is to be extracted from the execution paths of the program. We explicate the computation performed by the program by representing each execution path as a subprogram, and then using the rules developed in this work to simplify the subprogram or to rewrite it into a different form.

Because the execution paths are to be extracted by the insertion of constraints into the program to be analyzed, this book may appear to be yet another piece of work in constraint-based analysis in light of the current research directions just outlined. But that is purely coincidental. The intent of this book is simply to present an analysis method that the reader may find it useful in some way. No attempt has been made to connect it to, or fit it into, the grand scheme of current theoretical research in program analysis.

The need for a method like this may arise when a software engineer attempts to determine if a program will do what it is intended to do. A practical way to accomplish this is to test-execute the program for a properly chosen set of test cases (inputs). If the test fails, i.e., if the program produces at least one incorrect result, we know for sure that the program is in error. On the other hand, if all test results produced are correct, we can conclude only that the program works correctly for the test cases used. The strength of this conclusion may prove to be inadequate in some applications. The question then is, what can we do to reinforce our confidence in the program? One possible answer is to read the source code. Other than an elegant formal proof of correctness, probably nothing else is more reassuring than the fact that the source code is clearly understood and test-executes correctly.

It is a fact of life that most of a real-world program is not that difficult to read. But occasionally even a competent software engineer will find

[1] This is not to be confused with the term "path sensitive." In some computer science literature (see, e.g., WIKIPEDIA in references), a program analysis method is characterized as being path sensitive if the results produced are valid only on feasible execution paths. In that sense, the present method is not path sensitive, as will become obvious later.

a segment of code that defies his or her effort to comprehend. That is when the present method may be called on to facilitate the process.

A piece of source code can be difficult to understand for many different reasons, one of which is the reader's inability to see clearly how the function was decomposed when the code was written to implement it. The present method is designed to help the reader to recover that piece of information.

To understand the basic ideas involved, it is useful to think of a program as an artifact that embodies a mathematical function. As such, it can be decomposed in three different ways.

The first is to decompose f into subfunctions, say, f_1 and f_2, such that $f(x) = f_1(f_2(x))$. In a program, f, f_1, and f_2 are often implemented as assignment statements. The computation it prescribes can be explicated by use of the technique of symbolic execution (King, 1976; Khurshid et al., 2003).

The second is to decompose f into subfunctions, say, f_3, f_4, and f_5, such that

$$f(x, y, z) = f_3(f_4(x, y), f_5(y, z)).$$

In a real program, the code segments that implement f_4 and f_5 can be identified by using the technique of program slicing (Weiser, 1984).

The third way of decomposition is to decompose f into a set of n subfunctions such that

$$f = \{f_1, f_2, \ldots, f_n\},$$
$$f : X \rightarrow Y,$$
$$X = X_1 \cup X_2 \cup \ldots \cup X_n,$$
$$f_i : X_i \rightarrow Y \text{ for all } 1 \leq i \leq n.$$

An execution path in the program embodies one of the subfunctions. The present method is designed to identify, manipulate, and exploit code segments that embody such subfunctions.

Examples are now used to show the reader some of the tasks that can be performed with the present method.

Two comments about the examples used here and throughout this book first.

Programs in C++ are used as examples because C++ is currently one of the most, if not the most, commonly used programming languages at present.

Furthermore, some example programs have been chosen that are contrived and unnecessarily difficult to understand. The reason to keep example programs small is to save space, and the reason to make them difficult to understand is so that the advantages of using the present method can be decisively demonstrated.

Consider the C++ program listed below.

Program 1.1

```
#include <iostream>
#include <string>
using namespace std
int atoi(string& s)
{
     int i, n, sign;
     i = 0;
     while (isspace(s[i]))
          i = i + 1;
     if (s[i] == '-')
          sign = -1;
     else
          sign = 1;
     if (s[i] == '+' || s[i] == '-')
          i = i + 1;
     n = 0;
     while (isdigit(s[i])) {
          n = 10 * n + (s[i] -- '0');
          i = i + 1;
     }
     return sign * n;
}
```

This is a C++ version of a standard library function that accepts a string of digits as input and returns the integer value represented by that string.

Now suppose we test-execute this program with input string, say, "7," and the program returns 7 as the value of function `atoi`. This test result is obviously correct. From this test result, however, we can conclude only that this piece of source code works correctly for this particular input string. As mentioned before, we can bolster our confidence in the correctness of this program by finding the execution path traversed during the test-execution and the answers to the following questions: (1) What is the condition under which this execution path will be traversed? and (2) what computation is performed in the process?

The execution path can be precisely and concisely described by the symbolic trace subsequently listed. The symbolic trace of an execution path is defined to be the linear listing of the statements and true predicates encountered on the path (Howden and Eichorst, 1977).

Trace 1.2

```
i = 0;
/\!(isspace(s[i]));
/\!(s[i] == '-');
sign = 1;
/\!(s[i] == '+' || s[i] == '-');
n = 0;
/\ (isdigit(s[i]));
n = 10 * n + (s[i] - '0');
i = i + 1;
/\!(isdigit(s[i]));
return sign * n;
```

Note that every true path-predicate in the preceding trace is prefixed with a digraph "/\" and terminated with a semicolon.

In comparison with the whole program (Program 1.1), the symbolic trace is simpler in logical structure and smaller in size because all statements irrelevant to this execution are excluded. This symbolic trace contains answers to the two questions previously posed, but the answers are not immediately obvious.

It turns out that we can obtain the desired answers by treating the true predicates on the path as state constraints (see Chapter 2) and by