1

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

using the rules developed in the following chapters to rewrite the trace into the form that directly reveal the answers.

In particular, we develop a set of rules that can be used to move a constraint up or down the control flow without changing the semantics of the program (see Chapter 3). By repeatedly applying this set of rules to the subprogram, we can obtain an equivalent subprogram with all constraints on the top and all assignment statements at the bottom.

If we apply this step to Trace 1.2, it becomes the following program.

Program 1.3

```
int atoi(string& s)
{
    int i, n, sign;
    /\!(isspace(s[0])) &&!(s[0] == '-');
    /\!(s[0] == '+' || s[0] == '-');
    /\ (isdigit(s[0]));
    /\!(isdigit(s[0+1]));
    i = 0;
    sign = 1;
    n = 0;
    n = 10 * n + (s[i] -- '0');
    i = i + 1;
    return sign * n;
}
```

Note that when constraints are placed immediately next to one another, their relationship becomes that of a logical conjunction. This conjunction of constraints on the upper half of the subprogram often can be greatly simplified because most constraints in the same program are not entirely independent of one another. The most common relationship among constraints is that one is implied by the other. This afford the possibility of simplification because, if A *implies* B, "A *and* B" is reduced to B.

Also note that the long sequence of assignment statements at the bottom of the subprogram can also be simplified by use of the technique of symbolic execution (King, 1975; Khurshid et al., 2003) and the rules developed in this book (Huang, 1990).

Program 1.3 can thus be simplified to Program 1.4.[2]

Program 1.4

```
/\ (isdigit(s[0])) && !(isdigit(s[1]));
return s[0] - '0';
```

Program 1.4 says that the computation performed by Program 1.1 can be reduced to a single statement `return s[0] - '0'` if its definition is constrained by the predicate `(isdigit(s[0])) && !(isdigit(s[1]))`. In words, this simplified subprogram says that this subprogram is defined for all input strings consisting of a single digit, and it computes the difference between the ASCII representation of the input digit and that of digit `'0'`.

In a quality-related software process, such as testing, understanding, walkthrough, and constructing correctness proof, it is essential that we be able to describe an execution path precisely and concisely, to determine the condition under which it will be traversed, and to determine the computation it performs while the path is being traversed. As just illustrated, we can use the present method to accomplish this effectively.

It is important to understand clearly that Program 1.4 is not equivalent to the statement

```
if ((isdigit(s[0])) && !(isdigit(s[1])))
return s[0] - '0';
```

Although this conditional statement will do exactly the same as Program 1.4 if the condition `(isdigit(s[0])) && !(isdigit(s[1]))` is satisfied, it will act differently when the condition is not satisfied: This conditional statement would do nothing (i.e., will maintain *status quo*) whereas Program 1.4 would become undefined, i.e., it would give no information whatsoever!

In abstract, if f is the function implemented by Program 1.1, f is decomposed by the programmer into many subfunctions, each of which is implemented by an execution path in the program. The path described by Trace 1.2 implements one of those. The present method allows us to treat each trace as a subprogram and to transform it to explicate the properties of that subprogram.

[2] Details are given in Appendix A.

Two possible relationships among programs are now introduced. Given two programs S_1 and S_2, we say "S_1 is (logically) equivalent to S_2" and write $S_1 \Leftrightarrow S_2$ if and only if S_1 and S_2 have the same input domain and compute the same function. We say "S_2 is a subprogram of S_1" and write $S_1 \Rightarrow S_2$ if and only if the input domain of S_2 is a subset of that of S_1, and, within the input domain of S_2, S_1 and S_2 compute the same function. The formal definitions are given in Chapter 2.

Evidently, Program 1.1 $\Rightarrow$ Program 1.4.

Different constraints can be inserted into Program 1.1 in different ways to create different subprograms. For instance, consider the execution path subsequently listed. It is similar to Trace 1.2 except that the last loop construct in Program 1.1 is iterated one more time.

Program 1.5

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

This trace, when treated as a subprogram of Program 1.1, can be similarly simplified to Program 1.6.

Program 1.6

```
/\ (isdigit(s[0])) && (isdigit(s[1]))
   && !(isdigit(s[2]));
return 10 * (s[0] - '0') + (s[1] - '0');
```

The first line says that this path will be traversed if the input is a string of two, and only two, digits. The function will return an integer value equal to that of the first digit times 10, plus that of the second digit. Again, that is precisely what `atoi` is designed to do.

State constraints can be used to decompose a program pathwise into subprograms, each of which is smaller and less capable than the original. A *program set* is a construct (as defined in Chapter 4) designed to "glue" two or more programs together to form a new program that is more complex and capable than each of its constituent components.

In abstract, we use ordinary set notation to denote a program set. Thus if we have two programs P_1 and P_2, we can form a program set $\{P_1, P_2\}$, the formal semantics of which are given in Chapter 4. For now it suffices to say that, if program $P = \{P_1, P_2\}$, P is not only defined in the subdomain in which P_1 is defined, it is also defined in the subdomain in which P_2 is defined as well. Furthermore, it has the capability of P_1 as well as that of P_2.

Because the curly braces "{", "}", and comma "," used in ordinary set notation have special meanings in many programming languages, the trigraphs "{{{", "}}}" and ",,," are used instead to prevent possible confusion when those symbols are used in a real program.

Thus a singleton program set consisting of Program 1.4 as its element is written as

```
{{{
    /\ (isdigit(s[0])) &&!(isdigit(s[1]));
    return s[0] - '0';
}}}
```

We can add Program 1.6 into this program set to form a new one:

```
{{{
    /\ (isdigit(s[0])) &&!(isdigit(s[1]));
    return s[0] - '0';
,,,
    /\ (isdigit(s[0])) && (isdigit(s[1]))        &&!(isdigit(s[2]
    return 10 * (s[0] - '0') + (s[1] - '0');
}}}
```