Pointer Analysis in Java Programs using Execution Path Information

By

WANG Jian

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for the
Degree of Master of Philosophy
in Computer Science and Engineering

April 2008, Hong Kong
AUTHORIZATION

I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

WANG Jian
Pointer Analysis in Java Programs using
Execution Path Information

by

WANG Jian

This is to certify that I have examined the above MPhil. thesis
and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by
the thesis examination committee have been made.

Prof. Fangzhen LIN, Thesis Supervisor

Prof. Lionel Ni, Head of Department

Department of Computer Science and Engineering

April 2008
ACKNOWLEDGEMENTS

I cannot help expressing my deep gratitude to my mentor, Prof. Fangzhen LIN. It was him who did all the good on my research and the scientific way during the whole period of my MPhil. study. In the research, there is nothing more satisfying to me than to learn from his venerable spirit of science, decent character and upbeat attitude.

I am also sincerely grateful to the members of my Examination Committee, Prof. Zonghua GU and Prof. Shing-Chi CHEUNG, for their advice and guidance, and for spending their time on examining my thesis.

I would like to thank my parents for all your support and care. You set a very good example for me to follow. I will always remember to be an honest and useful person to others.

Last but not the least, I want to give my thanks to my girlfriend, my group members, my roommate and all the friends I have met in UST. I really cannot imagine how pale the life would be without their encouragement and help. Living here make me realize more things. Not only the goal is important, being here, enjoying and learning is at least as important. I learn to love the journey, not only the destination.
# Table of Contents

Title page i
Authorization page ii
Signature page iii
Acknowledgments iv
Table of Contents v
Abstract vii

**Chapter 1 Introduction** 1

**Chapter 2 Pointer Analysis** 5

2.1 Pointer Analysis 5

2.1.1 Types of Pointer Analysis 6

2.1.2 Call graph discovery 9

2.2 Steensgaard’s and Andersen’s algorithm 9

2.2.1 Steensgaard’s algorithm 9

2.2.2 Andersen’s algorithm 10

**Chapter 3 Execution Trace Generation** 11

3.1 JavaCC (the Java Compiler Compiler) 12

3.2 Generation of Execution Trace 14

**Chapter 4 Algorithm for the Maybe-alias Discovery** 23

4.1 Overview of the solution 23

4.2 Algorithm for maybe-alias discovery 26

**Chapter 5 Comparison** 30
5.1 Static Analysis vs. Dynamic Analysis
5.2 Application of the dynamic points-to set

Chapter 6 Conclusion and Future work

Reference
Abstract:

This thesis investigates the pointer alias information in Java programs. The basic idea is to get the pointer alias information by generating and analyzing the trace of one-time-run of the target programs. A trace tracks the complete information of one-time-run of the target program with certain input. After analyzing a particular trace, we are sure to discover the portion of the pointer alias pairs related to that run. After collecting traces of different runs of the target programs with various inputs, we perform the analysis on the reference variables with the same type, which are equivalent to pointers. With different inputs, we can get different traces of the program approximating the complete behavior of the program. Our approach is different from the previous research in that it is a hybrid of static and dynamic analysis. We obtain the execution trace by running the program, and then we statically analyze the trace to discover the pointer alias information.

To the best of our knowledge, this is the first to analyze the pointer alias information in Java programs by traces of unrolled program statements. The contribution of our work is that we systematically implement the algorithm to
calculate and collect the dynamic points-to set information, which turns out to be useful for debugging and program slicing. We believe that with the development of computer hardware and the application of software testing techniques, our method shall be useful because of its preciseness.
Chapter 1

Introduction

Program analysis is the process of analyzing the behavior of a computer program. And it has many important applications. In the literature, it has been used primarily for optimizing programs so that they will run faster. More recently, with the ever more powerful hardware, people change their focus to use program analysis as a tool to aid software development.

Program analysis has the potential to be extremely useful in finding bugs and security vulnerabilities in software. Research has shown that 60% of software faults that were found in released software products could have been detected with static analysis tools[2]. And 40% of the faults that could have been found through static analysis would eventually become a defect in software production [5].

Despite its promise, program analysis has not substantially improved software reliability in the real world, mainly because that the results of program analysis may not be precise. Imprecision can lead to false positives, which means the tool reports a bug when none exists.

One major source of analysis imprecision comes when a program uses pointers. Pointers are important, and they existing in almost every major programming language. Pointer analysis is essential to the understanding of every computer program. Despite its importance, pointer analysis is still one of the most bewildering
problems in program analysis. What’s more, from [4] we can know statically determining intraprocedural May Alias for language with if-statements, loops, dynamic storage and recursive data structures is recursively enumerable but not recursive even when all paths through the program executable. Therefore, the known algorithms are necessarily a trade-off between efficiency and precision. Efficiency of the alias analysis can be defined by the computing resources (for example, the time or the memory) that are needed to perform the analysis. Precision can be defined as the accuracy of the alias-analysis results and the usefulness of these results to the clients that use them.

Without the detailed knowledge of how a program manipulates the memory, compilers have to make very conservative assumptions about statements that use pointers. This limits the use of standard compiler optimizations such as constant propagation, common subexpression elimination, loop-invariant code motion, strength reduction, dead code removal, etc. As modern object-oriented languages, such as Java make heavy use of pointers or reference variables, the analysis of pointers becomes really important for compiler optimization, program understanding and debugging.

In its most general definition, the pointer analysis field tries to detect useful properties about pointers. It occurs to me that there are at least four types of analyses that one can do: points-to analyses, alias analyses, escape analyses, and shape analyses. A points-to analysis identifies the memory locations which are pointed to by a specific pointer variable or heap reference. An alias analysis determines whether or not separate memory references point to the same area of memory. An escape analysis
detects all the places where a pointer can be stored and whether the lifetime of the pointer can be proven to be restricted only to the current procedure and/or thread. The scope of escape analyses is usually a method, but can also be the current iteration of a loop, a group of methods, etc. Finally, shape analyses are about the linked structures in the heap or the pointers dumped into the heap from stacks. Shape analyses can be used to check properties such as “a method to reverse a linked list does not introduce cycles to the list”. In practice, all these analyses are closely related.

Despite the importance of the problem and the large amount of resources invested for more than two decades, pointer analysis remains an “exciting research problem”. To the best of our knowledge, the pointer analysis used by the industrial compiler is not very precise. This can be attributed to the following situations: the difficulty of the problem itself, chances of big improvements with simple analyses, steady increases in the power of hardware that make work on optimizing compilers look unnecessary, etc.

This thesis investigates the problem of pointer alias analysis by using the information of execution traces. An execution trace tracks executed statements of the program given a particular input. In the trace, we no longer have the control structures such as “if-then-else”, “while-do” and recursive definition. All the statements are tracked according to the exact execution order of the program run. In this way, the execution trace gives the complete and correct information related to a program run with the given input, thus we can just use trace for further analysis without losing any information.
In addition to execution trace information, we use type information from the original program to get the type information. Since Java is a strong-type language, we can assume the pointer alias pair only happens to the variable of the same type. With the type information of different variables, we only need to care about the reference variables in Java which are equivalent to pointers as we usually call. The next step is to check the statements in the trace to find out the relationship of the reference variables. We compute the pointer alias pair based on the reference variables of the same type and the “assignment” operation between two variables. The direct alias pair can be obtained easily, and we also need to find out the indirect pointer alias, such as “A points B, B points to C, A-B, B-C and A-C are all pointer alias”. The indirect pointer alias is calculated by scanning the tree-like structure that keeps the information the direct alias pair found. We will talk about the algorithm in detail in Chapter 4.

The rest of the paper is organized as follows. In Chapter 2 we give an introduction of pointer analysis. In Chapter 3, we consider how the execution traces can be generated. Next, in Chapter 4, we describe the algorithm for calculating the may-be alias pairs from an execution trace and the original program. Chapter 5 describes the difference and similarity of our method to previous research and the practical applications of our method in software engineering. We conclude in Chapter 6.


Chapter 2

Pointer Analysis

In this chapter, we give some background introduction which is necessary for the understanding of the rest of the thesis. It includes the definition of the terminology in use and some research areas that are related to this thesis.

2.1 Pointer Analysis

Simply speaking, pointer analysis is the analysis to determine the memory locations that the pointer variables could point to. And pointer analysis is very important to the understanding of computer programs since pointer is the key feature of almost every major programming language. Without knowing the location that a pointer can point to, we can only gather very limited information. For example, it is hard to image that we can perform correct pointer dereference based on incomplete pointer information.

Pointer analysis has been an active research field for the last two decades. In an invited talk at PASTE’01[6], Michael Hind give a list of seventy-five papers and nine PhD thesis published on pointer analysis. And the first pointer analysis was published by Weihl in 1980[10].

Despite the importance of pointer analysis, the problem itself is undecidable in general, even when the program has only a single procedure and no dynamic allocation[4]. This means even if we are allowed to write exponential algorithms, the analysis would still have to be approximate. Different algorithms provide trade-offs between the efficiency of the analysis and the precision of the computed answer. Efficiency of the alias analysis can be defined by the computing resources (for
example, the time or the memory) that are needed to perform the analysis. Precision can be defined as the accuracy of the alias-analysis results and the usefulness of these results to the clients that use them.

2.1.1 Types of Pointer Analysis

The alias-analysis algorithm can be designed along different dimensions, which will affect the efficiency and the precision of the alias-analysis algorithms[6,8]. The dimensions include flow sensitivity, context sensitivity, object representation, field sensitivity and directionality. We will talk about each dimension in the following discussion.

Flow Sensitivity

Flow sensitivity refers to whether to take the control flow information of the computer programs into account. Usually in a programming language, we have the control constructs like goto, conditional and loop etc. Different runs of a same computer program behave differently due to the effects of these control constructs, i.e. different statements of the program are executed. Flow-insensitive analysis computes a solution for the whole program (or a procedure of the program) by viewing the statements inside a certain block as a whole. This means the execution order of the assignments inside this block does not influence the final result. A flow-sensitive analysis computes a solution for each program point, such as program statement. Intuitively, flow-insensitive analysis is less precise but simpler thus can be implemented efficiently.

In terms of pointer analysis, the key difference of flow-sensitive algorithm is that
it takes strong updates[11] into account. Here strong update is a kill assignment which means it will overwrite the previous contents of the destination. This can only happen when the memory location pointed to is a single unique location, also known as “must-point-to”. Obviously, flow-sensitive analysis should distinguish the memory state before and after an update instruction.

**Context Sensitivity**

Context sensitivity refers to whether different calling context will influence the results of a method. It is important for inter-procedural analyses because it keeps the information of different calling contexts separate, whereas context-insensitive analyses mixed the information of calling contexts together. As different calling contexts usually have their own characteristics, mixing them together certainly sounds a bad idea and it certainly leads to imprecision.

The most common way to define a calling context is based on the chain of call sites starting with main and ending with the particular procedure call under analysis. This is known as the call strings approach or k-CFA, where k indicates the length of the call string maintained, that is the number of call sites considered in the context. One notable version of k-CFA is ∞-CFA, where the entire call path to the root, is used as the context.

In order to achieve context-sensitivity, there are two important techniques, one is cloning-based, the other is summary-based. In cloning-based technique, for each context it “clones” a method and computes the result separately for each clone. And in summary-based technique, it calculates a summary of the effects of a method and
then applies that summary at different call sites.

Object Representation

The resolution of object names can have huge effect on analysis precision. There are two common representation choices for pointer analysis. First, an analysis can use one abstract object per class to represent all possible instantiations of that class. Second, objects can be identified by their creation site; in this case, all objects created by a specific `new` statement are represented by the same abstract object.

Field Sensitivity

Field sensitivity refers to whether or not to preserve information associated with distinct reference fields inside an object. For a C pointer analysis, it combines the analysis information for all fields of the object, i.e. $x.f$ and $x.g$ would be indistinguishable to $x$. Field-sensitive analyses distinguish different fields inside an object. Most analyses of the type-safe language such as java are field-sensitive. However, most analyses simply treat all elements of an array as a single location.

Directionality

Directionality is also known as Inclusion-based or Unification-based analysis. The differences lie on the treatment of the assignment operator.

An assignment statement “$x=y$” can be explained in two ways. First, “the points-to set of $x$ includes everything that $y$ points to” which is known as inclusion constraints; second, “the points-to set of $x$ and $y$ collapsed into a single equivalence set and become indistinguishable” which is known as unification constraints. In the second treatment, the assignment statement loses directionality. So the statement
“x=y” is equivalent to the statement “y=x”. The unification constraints are similar to those used in Steensgaard’s pointer analysis for C [9]; the inclusion constraints are similar to those used by Andersen’s pointer analysis for C [1]. Solution procedures for both types of constraints are polynomial time (in the size of the constraint set), but unification constraints can be solved in almost linear time [9], whereas inclusion constraints need cubic time.

2.1.2 Call graph discovery

In order to analyze the whole program, usually we need a call graph to record the relationship of different procedures. Without a complete call graph, the analysis result can never be sound. And there are two ways to deal with call graph --- a pre-computed call graph or compute the call graph on-the-fly based on the analysis information collected so far.

In order to generate a call graph, the analysis starts with a set of root methods that are the entry point of the program. It then iteratively adds the target methods to the analysis after calculating the possible call targets.

2.2 Steensgaard’s and Andersen’s algorithm

Steensgaard’s and Andersen’s algorithms are two of the best-known alias-analysis algorithm. Both algorithms are flow-insensitive and context-insensitive. The difference between Steensgaard’s algorithm and Andersen’s algorithm is the way the algorithms treat assignments.

2.2.1 Steensgaard’s algorithm

Steensgaard’s algorithm is a very efficient algorithm: it can compute the alias-analysis
information in almost linear time. Steensgaard’s algorithm uses an unification-based approach to treat assignments.

By using unification-based approach, Steensgaard’s algorithm treats assignment as an equality relation. If one pointer variable is assigned to another pointer variable, the points-to sets of both pointer variables are unified, and consequently, both variables have the same points-to set. The points-to sets of both pointer variables are merged using the “fast union-find algorithm”. Because Steensgaard’s algorithm uses a unification-based approach, each pointer-related assignment needs to be processed only once.

2.2.2 Andersen’s algorithm

Andersen’s algorithm can compute more precise point-to information than Steensgaard’s algorithm. However, the worst-case time complexity of Andersen’s algorithm is $O(n^3)$. Andersen’s algorithm uses an inclusion-based approach, which implies a subsume relation.

If one pointer variable is assigned to another pointer variable of form “$l = r$”, the points-to set of pointer variable $l$ must subsume the points-to set of pointer variable $r$ (i.e., the points-to set of $l$ is a super set of the points-to set of $r$). Because of the subsume relation, assignment need to be processed more than once when a points-to set that is subsumed by another points-to set changes.
Chapter 3
Execution Trace Generation

The main research topic of this thesis is to discover the pointer alias information of java programs. Instead of using the traditional methods that try to discover this information by looking at the source codes, we choose to gather the information from the generated execution traces. For this to be effective, the target program must be executed with sufficient test inputs to produce interesting behaviors. To achieve this we can use software testing techniques such as code coverage helps to ensure that an adequate slice of the program's set of possible behaviors will be observed.

In this thesis, we consider only the maybe-alias pair instead of must-alias. Maybe-alias means two reference variables may point to the same memory location in some cases. Must-alias means two reference variables must point to the same memory location in all cases. The characteristic of the problem justify the technique we are using.

- Given certain input, whether in correct form or not, a computer program will run. If the program can terminate, the internal state (value of the variables) may change and there will be an execution trace faithfully record every statement the program has executed.
- In the execution trace, only the assignment operation will update the information of maybe-alias.
- By supplying the program with different inputs, if the inputs cover all the cases the program to consider, we will get a complete set of maybe-alias pair after
analyzing these different execution traces.

In this chapter we first briefly introduce JavaCC (section 3.1), and then describe
the generation procedure of execution trace (section 3.2).

3.1 JavaCC (the Java Compiler Compiler)

JavaCC stands for “the Java Compiler Compiler”; it is a parser generator and lexical
 analyzer generator.

Figures 3.1 and 3.2 show the relationship between a JavaCC generated lexical
 analyzer (called a “token manager” in JavaCC parlance) and a JavaCC generated
 parser. In the figures, standard C is used as the input language as an example, but
 JavaCC can handle almost any language —and not only programming languages— if
 you can describe the rules of the language to JavaCC.

![Diagram](image)

**Figure 3.1** The token manager converts a sequence
 of characters to a sequence of Token objects

The job of token manager is to read in a sequence of characters and produces a
 sequence of objects called “tokens”. It uses rules supplied by the programmer as a
collection of “regular expressions” to break the sequence of characters into a sequence of tokens. A simple example is : < #DECIMAL_LITERAL: ["1".-"9"] (["0".-"9"])*. This regular expression is used to construct “decimal literal” tokens from the input stream of characters. Any decimal integer begins with a number from 1 to 9, followed by zero or more numbers from 0 to 9.

After the token manager finishes its job, a sequence of tokens will be passed to the parser. In figure 3.2, the parser consumes the sequence of tokens, analyses its structures and produces an “abstract syntax tree”. To produce, say,

![Figure 3.2: The parser analyzes the sequence of tokens](image)

a file of assembly language (if you are writing a one-pass compiler) or a modified sequence of characters (if you are writing a text processor). What one needs to do is to provide a complete set of production rules in the form of “Extended BNF”. JavaCC will generate the parser as a Java class based on this set of rules.

Figure 3.3 is a simple example that specifies how to construct a conditional
Figure 3.3 An example of Extended BNF production rule statement. “LOOKAHEAD(1)” is a lookahead specification that is used to deal with conflicts or ambiguity. In this example, the semantic is that “if” will make a pair with the closest “else”.

The set of regular expressions that specifies the token manager and the set of Extended BNF production rules that specifies the parser are all included in the JavaCC input file which has the extension “.jj”. JavaCC’s generated parser classes work by the method of “recursive descent”. A recursive descent parser is a top-down parser built from a set of mutually-recursive procedures (or a non-recursive equivalent) where each such procedure usually implements one of the production rules of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes. Generally speaking, JavaCC limits itself to the LL(k) class of grammars. In particular, left recursion is not used in it.

3.2 Generation of Execution Trace

When a user tries to use computer software, usually he will consider the software as a “black box”. What he needs to do is to give the software appropriate inputs, assuming the software does its job, he will receive the correct answer from the software. Just like when we are using a calculator, we do not need to know how the calculator works,
what we need to do is to press the number and the arithmetic operator buttons in a correct sequence, then we will get our desired answer. All of this sounds perfect and natural. Unfortunately, we cannot always assume the software does its job. It may happen that we key in “2+2=”, and we get the result “5” instead of the correct result “4”. In this case, the programmer needs to see into the “black box”, and find out the bugs that make things wrong. What makes things worse is that usually the code is poorly documented, and in some case, programmer may only get a portion of the sources.

Instead of mindlessly reading the source code, if we have it, people usually choose to debug the program by tracing the code at runtime. The main method of the program takes the input from the user. Inside the program, there will be different control structures like conditional, loop and recursive definition. Also there are different procedure calls in a run of the program. In this thesis, the trace of the program is the actual statements the program executes in a run. In the following are two examples that illustrate the traces.

Figure 3.4a is an example of a Java program. Inside it, there are one main method and three user-defined methods. The control structure of conditional, loop and procedure call are all presented. Figure 3.4b is the corresponding trace. Figure 3.5a is an example of Factorial function that is recursively defined, and Figure 3.5b is the corresponding trace.

From these two examples we can see that the trace of the program completely record every statement program executed in time order.
Figure 3.4a An Example of Java Program

```java
public class Test {
    public static void main(String[] args) {
        int a, b;
        a = 0;
        b = f();
        System.out.println(a);
        System.out.println(b);
    }

    static int f() {
        int j = 0;
        int z = 0;
        while (j < 7)
            j++;
        i++;
        return 3 * g(j);
    }

    static int g(int k) {
        int z = k;
        if (z < 4)
            z = 5 * z;
        return z;
    }

    static int h() {
        int x = 4;
        return 4 * 4;
    }
}
```

Figure 3.5a The Factorial Function

```java
package com.xyz;

public class Factorial {
    public static void main(String[] args) {
        Factorial fac = new Factorial();
        System.out.println(fac.factorial(3));
    }

    public int factorial(int n) {
        // assert(n >= 0);
        if (n == 0 || n == 1)
            return 1;
        else
            return n * factorial(n - 1);
    }
}
```


```java
{ // void main (string[] args)
    int a, b;
    a = b = 0;
    {
        // int f ()
        int i = 0;
        int i = 0;
        i += 1;
        i += 4;
        i += 4;
        i += 1;
        i += 1;
        i += 1;
        i += 1;
        i += 1;
        i += 1;
        i += 1;
        // int g (int k)
        int z = k;
        {
            // int h ()
            int x = 7;
            return 7 * 7;
        }
        // int h ()
        int x = 7;
        return 7 * 7;
        z = z + 1;
        return z;
    }
    return 3 * g(1);
} a = f();
{
    // int g (int k)
    int z = k;
    {
        // int h ()
        int x = 7;
        return 7 * 7;
    }
    return z;
}
    b = 2 * g(0);
    System.out.println(a);
    System.out.println(b);
}
```

**Figure 3.4b The Trace of the Java program**

```java
// void main (String[] args)
Factorial fac = new Factorial();
{
    // int factorial (int n)
    {
        // int factorial (int n)
        {
            // int factorial (int n)
            {
                // int factorial (int n)
                return 1;
            }
            return n * factorial(n - 1);
        }
        return n * factorial(n - 1);
    }
    System.out.println(fac.factorial(3));
}
```

**Figure 3.5b The Trace of Factorial Function**
In the rest of this chapter we will talk about how to use JavaCC to construct the execution trace, and in next chapter we will focus on how to discover the maybe-alias pair from the generated execution trace.

From the discussion in part 3.2, we know that JavaCC is a parser generator and lexical analyzer generator. The execution trace generator consists of two parts corresponding to parser and lexical analyzer respectively. In the lexical analyzer part, we specify the set of regular expressions that is used to distinguish the tokens in a java program.

To be specific we have:

- **White space**
  The set of characters that can be skipped, e.g. “\t”, “\n”, “\r”

- **Comment**
  In java we have multiline comment that begin with “/*” and end with “*/”, and we also have single line comment begin with “//”

- **Reserved words and literals**
  Reserved words are the set of words that have a special grammatical meaning to a language and cannot be used as an identifier in that language, e.g. “double”, “if”, “super”, “try” etc.
  Literals include the characters represent the numbers (integer or float), character and strings.

- **Identifiers**
  Identifiers (IDs) are lexical tokens that name language entities. Some of the kinds
of entities an identifier might denote include variables, types, labels, subroutines, and packages

- **Separators**

  Separators include “(”, “)”, “{”, “}”, “[”, “]”, “;”, “,”, “.”, “@”

- **Operators**

  Operators include all the operators that exist in Java, e.g. “==”, “?”, “|”, “>”

  After this process, the source file provided to the JavaCC is no longer a character stream, it becomes the meaningful token set in Java. The next step is how to construct the java programs based on this token set by Java language grammar.

  We specify the java language grammar in four parts, program structuring syntax, declaration syntax, expression syntax and statement syntax. Program structuring syntax includes the package declaration, import declaration and modifiers (public, abstract, final etc). Declaration syntax includes the declaration of class or interface, enumeration, variable, method, constructor etc. Expression syntax includes how to construct an expression in java. And finally statement syntax specifies how to construct the statement based on expressions. Figure 3.3 is an example of the syntax to construct an “if-else” statement in Java.

  The reason why we need this two part is that we want the execution trace records every statement in chronological order. From figure 3.3 we can see, in the statement construction process we add some code in order to record the statement in the trace. From figure 3.6 we can see, after we get the source file f we will add the necessary trace statement into the source file, the new file f’ is still a runnable java file and do
exactly what the original source file $f$ does plus providing the trace information; next we run the new file $f'$ as any normal .java file, we will get execution trace $t$ in .txt file. By giving $f'$ different inputs, we will get different traces $t_1$, $t_2$, $t_3$ … etc. And these different $t_i$s are used to discover the maybe-alias pair in next chapter.

Figure 3.6 Mechanism of Execution Trace Generator

- Declaration

This part mainly includes the declaration of class or interface, enumeration, variable, method, constructor etc.

- Class or Interface Declaration

For class we need to specify the Identifier, the parameters. If the class inherits other class, we also need to specify the extended class. Inside the class, there will also be declaration of field, variable, constructor and method.

For interface, we need to specify the implementing class.

- Enumeration

For an enumeration, we need to specify the name and the body.

- Variable

Variable will have type information. And there are also special variables like
array and using expression to specify variable.

- **Method**

  Method has type, identifier information. It also has the special feature of “throw” that throws an exception.

- **Constructor**

  Constructor is similar to method except that it has the same name as the class.

- **Type and Expression**

  - **Type**

    In the type part we need to distinguish the primitive type and the reference type.

  - **Expression**

    For an expression, we use different operators to distinguish different expressions. For example, we have conditional expression (or, and, inclusive, exclusive etc), assignment expression (add, minus, multiply etc), relational expression (bigger, smaller etc), unary expression (increment, decrement), etc. Expressions are the building blocks of the statements.

- **Statement**

  Statements are separated according to the control structures of the program. For example, we have switch statement, for statement, try statement, if statement, and return statement etc…

  Most of the statements are easy to deal with, we just need to distinguish them by
the reserved words, and fill the program with the corresponding reserved words.

While two types of statements need special treatment. One is the for statement, the other is the return statement.

- **For statement**

  Insider for statement, we need to specify the ForInit, Expression and ForUpdate. These three parts are all need to be recorded in the trace since they are not simple relation expression like the judgement of whether 3<5, there are variable updates inside these three parts.

- **Return statement**

  The treatment of return statement matters both to the procedure call and the recursive data structure. When meet with a return statement, we substitute the expression with a new variable, execute the expression to calculate the answer, assign the new variable with the answer and then return the new variable to the caller.

Since the trace $t$ is generated by executing the annotated file $f'$, the user-defined methods will be unrolled automatically. However, the system build-in function will not be unrolled. It is like the inline function, user will get the answer as if the function is just one statement.
Chapter 4

Algorithm for the Maybe-alias Discovery

From the discussion of part 3.1 we know that the pointer-alias analysis is imprecise in general. And we have the different dimensions like flow sensitivity, context sensitivity and directionality. Refer to these dimensions, our algorithm is essentially a flow-sensitive, context-sensitive and inclusion-based ($x = y$ is not equivalent to $y = x$) algorithm. Theoretically speaking, the result will be relatively precise.

4.1 Overview of the solution

Context-sensitivity is based on the fact that with a specific input there will only be one fixed execution trace if no random procedure is included in the program. Thus we achieve the context sensitivity by separating different context in different runs of the program, i.e. every execution trace represents one calling context from the root to the end of the program. Flow-sensitive is based on that the algorithm given here does not consider the execution trace as a whole, rather it distinguish different program points, i.e. if P first point to A and A is alias to other pointers $A_1$, $A_2$ …$A_n$, then P together with any one in the set of {$A$, $A_1$, $A_2$ …$A_n$ } are alias; but if in the next statement P changes to point to another pointer B, the information will update immediately, and at that program point, P and B are alias while P is no longer the alias of {$ A$, $A_1$, $A_2$ …$A_n$ }. The directionality will be talked about later in detail in the algorithm explanation.

Because Java is a strong type language, variables can only point to objects of assignable types. Assignability is similar to the subtype relation, with allowances for
interfaces, null values, and arrays[7]. By dropping targets of unassignable types in assignments and load statements, we can eliminate many impossible points-to relations.

In Java, there are two major categories of data types, one is the primitive type which include int, char, double…etc, these primitive data types in Java are not objects; the other is reference type. Reference types in Java are types where the name of the variable evaluates to the address of the location in memory where the object referenced by the variable is stored.

Figure 4.1 Reference Type

From Figure 4.1 we can see, there are three kinds of reference types: class types, interface types and array types. Class and array are similar to the definition as in C++, an interface declaration introduces a new reference type whose members are classes, interfaces, constant and abstract methods. This type has no implementation, but otherwise unrelated classes can implement it by providing implementation for its abstract methods. A variable whose declared type is an interface type may have its
value a reference to any instance of a class which implements the specified interface. What’s more, it is not sufficient that the class happen to implement all the abstract methods of the interface; the class or one of its superclass must be declared to implement the interface, or else the class is not considered to implement the interface.

Two reference types are the same compile-time type if they have the same binary name, in which case they are sometimes said to be the same class or the same interface. In java, a variable always contains a value that is assignment compatible with its type. A variable’s value is changed by an assignment or by prefix or postfix ++(increment) or --(decrement) operator.

Figure 4.2 The overview of the solution

From figure 4.2, we can get an overview of the solution. When we get a java source file and our task is to find out the maybe-alias pairs, the task is separated mainly into two parts. First part is generation of the execution trace and collection all the reference variables; second part is using the collected information to discover all the maybe-alias pairs in this execution of the program.

The first part is discussed in chapter 2 with the collection of reference variables
untouched. Actually this step is natural. What we need to do is just adding some specific tag in the declaration of reference variables to the annotated file $f'$. After running the program of $f'$, execution trace and reference variables can be obtained simultaneously.

4.2 Algorithm for maybe-alias discovery

From the discussion of 4.1 we know that Java is a strong type language, thus we can limit our search to the set of reference variables. Besides, the way to change the value of a reference variable is assignment. So we can use the operator “=” to further limit our search domain. Since our analysis of pointer-alias is inclusion-based, the operator “=” has its directionality.

Input: Set of Reference Variables, Execution Trace
Output: Set of pointer alias pair
Algorithm:
{   While (!EOF)
    { 
        Scan the current line;
        If “=” is met and two operand op1, op2 are reference variable
            Call aliasUpdater(op1,op2);
        Else
            Continue;
    }
    Check the forest, every two nodes in a tree are pointer alias pair;
}
aliasUpdater(op1,op2)
If op1 has not been assigned before

{ op1->op2;
  Record op1=op2; }

Else if the assigned value v is not in the same tree of op2

{ Find out the position of op1 in the forest;
  Break the link op1->v;
  op1 -> op2; // the subtree of op1 points to op2 indirectly
  Record op1=op2; }

Else

{op1->op2;
  if the operation introduce a cycle
    {Mark the original op1->v;
     Update the cycle set;}
  Record op1=op2;
}

Figure 4.3 Algorithm For Pointer Alias Discovery

In this algorithm we need to keep four data structures, first the set of all the reference variables, this is the search domain of our algorithm; second, the latest updated value to every reference variable, this information is necessary for the flow-sensitivity of our pointer alias analysis; third the forest which is constructed through the algorithm, every tree in the forest represent a set of pointer alias pairs, the whole forest record all the different set of pointer alias pairs of this execution; fourth is the cycle that is constructed during the forest-like data construction. The reason why we are using a tree structure is based on the characteristic of reference variables. Operator “=” has its directionality that the left operand points to the right operand. The execution trace is
generated in time sequencing, so the later assignment of the left operand will update
the previous assignment. In this manner, at any time point, one reference variable can
only directly points to one memory location, it is impossible for a reference variable
to simultaneously points to two different memory locations. Meanwhile, it is possible
that different reference variables point to the same memory locations and it is the
basic reason why we have to do the pointer alias analysis research. There may be
cycle in certain tree which makes it a tree like structure.

Therefore, forest-like structure should be the correct data structure to store the
alias pair information. The reason why it is a forest-like is that there usually exist
different sets of alias pair in the memory, all the reference variables directly or
indirectly pointing to the same location will be a rare case. The root of every tree-like
will be a memory location that is initialized to certain reference variable(s). Other
nodes under the first level of the tree-like are constructed gradually by the assignment
operation that assigns a reference variable to another. And every edge has the
direction that children point to parent.

When an assignment statement of two reference variables “op1=op2” is
discovered, there will be three situations:

- **op1 has not assigned a value before**
  
  Direct add a link from op1 to op2;

  Add op1=op2 to the latest updated value table

- **op1 has been assigned before and the value is not in the same tree of op2**
  
  break the link that op1 is current pointing to;
Add a link from op1 to op2, thus the subtree rooted from op1 is under op2 now;

Updated op1=op2 in the latest updated value table;

- **op1 has been assigned before and the value is in the same tree of op2**

  This will be the tricky part. It is possible that after the assignment, a cycle will be created. The cycle is easy to check, if the level of op1 is higher than op2, then the assignment will introduce a cycle. Under this situation, record the cycle; let op1 point to op2 and mark the original link op1 pointing to. In this way, the nodes in the cycle can be considered as a big node, at the same time this big node is not detached from the original tree.

  Please notice that the right operand op2 should always be assigned a value, or it cannot appear in the right side of the operator.

  If the size of the execution trace is M, the size of the reference variables set is N, then the time complexity of the algorithm is $O(M^2N)$. Usually when running a program, the lines of the execution trace will vary greatly with different input because of the unwinding of the loop. This explains the running time variation of the same program with different input.

  In the next chapter I will discuss the experiments I have done using the algorithm discussed here.
Chapter 5
Comparison

In this chapter we will discuss the differences between the method we propose and the methods that have been used in the literature. As far as pointer alias concerned, researchers have made good effort to get a “precise enough” points-to set for the use of compiler optimization, bug detection, program understanding even security. We use the word “precise enough” because all the efforts researchers have done is a trade-off between efficiency and preciseness. The problem of pointer alias is intractable to solve in theory. Fortunately, what we want is the better usage of the points-to set information.

Section 5.1 will discuss the differences and similarities between static and dynamic analysis. And in Section 5.2, we will point out the possible applications of the result calculated by the dynamic analysis we propose.

5.1 Static Analysis vs. Dynamic Analysis

Static and dynamic analyses arose from different communities and also evolve in different tracks. Yet, from the discussion of [3], we can see that both communities shall complement each other.

Static analysis has been a success for a long time in the area of optimizing compiler, debugging, program slicing, etc. Static analysis examines the program code and reasons over all possible behaviors that might happen at run time without actually running the program. The result of the static analysis is usually sound but conservative. Soundness means the analysis results are an accurate description of the
program, no matter on what inputs of the program. Conservative means the property reported is weaker than it actually is. For example, given a function \( f \), the property "\( f \) returns non-negative number" is weaker than "\( f \) returns even number". And a conservative analysis might report the latter, or even weaker property like "\( f \) returns a number".

Static analysis works by building a model of the state of the program, then reasoning over these states. Since it is infeasible to consider every possible runtime state, static analysis builds an abstracted model of the program state that loses some information, but which is compact and easier to manipulate.

Dynamic analysis works by executing a program and then analyze the information generated through the execution. Dynamic analysis is precise because no approximation or abstraction is needed. And its speed can be as fast as the program execution. The disadvantage of dynamic analysis is that the results may not apply to future executions. There is no guarantee that the test suite will fully represented all possible program executions. A sound dynamic analysis need to observe every possible execution of a program. This simple goal is not achievable, nontrivial programs usually have infinitely many possible executions, and only a small number of them can be considered due to the time and money constraints. Unsoundness of dynamic analysis has been an issue for the widely adoption of the method. However, in certain circumstance, soundness is not necessarily demanded. Tools designed to assist a human in a software engineering task are free of the soundness restriction. Moreover, in many cases the results of a static analysis, although sound, maybe
considerably less useful than the potentially unsound results of a dynamic analysis, because the conservativeness of the result of static analysis usually overwhelms the user with too much data. Dynamic analysis can also increase the confidence of a system.

There is a key observation need mentioning: both static and dynamic analysis are only able to consider only a subset of program executions. Generalization over this subset is the reason why dynamic analysis is unsound and static analysis is imprecise.

From the discussion in previous chapters we know that the subset chosen by dynamic and static analysis is different. Dynamic analysis considers the executions that are in the test suite. This subset is very much input-related. On the other hand, static analysis considers the executions that induce executions of a certain variety. For example, in 2.1.1, we talked about context-sensitivity. The k-CFA abstraction considers the calling context with the maximum length k.

By recognizing this duality, we shall agree the possible transformation of approaches from one domain to the other and the combinations of both static and dynamic analyses.

5.2 Application of the dynamic points-to set
The result of dynamic analysis is unsound, but this doesn’t necessarily mean the result is useless. In the application of compiler optimization, we need to make sure that two variables will never share the same memory location, therefore we can assign different registers to them to speed up the calculation, in this case, we have to use the sound result reported by static analysis. However, we know the fact that the result of
static analysis is an upper bound while the result of dynamic analysis is a lower bound. When the dynamic and static sets agree, we are sure that the static information is in fact optimal and not just a conservation approximation. What’s more, when debugging a program, it is actually more important to present only the dynamically observed targets rather than all the possible targets from different executions reported by static analysis.

Program slicing is answering the question of “What part of the program affect the computation of variable $v$ at statement $s$?” And the power and utility of program slicing is enormous. It can help in the area of: **Differencing** – Whether or not two program is doing the same thing?; **Software maintenance** – Understanding existing software and making changes that will not have negative impact on the unchanged part; **Debugging** – Given a piece of code with bug in it, find out the portion of the code that creates the bug and fix it; **Reverse Engineering**, **Software Quality Assurance**, etc.

The definition of program slice is: For statement $s$ and variable $v$, the slice $S$ of program $P$ with respect to the slicing criterion $<s, v>$ is any executable program with the following properties:

1. $S$ is obtained by deleting zero or more statements from $P$.
2. If $P$ halts on input $I$, then the value of $v$ at statement $n$ each time $n$ is executed in $P$ is the same in $P$ and $S$. If $P$ fails to terminate normally, $n$ may execute more times in $S$ than in $P$, but $P$ and $S$ compute the same values each time $n$ is executed by $P$. 

A program slice is computed by transitively following the dependences between variables, and smaller points-to sets will result in fewer pointer-induced dependences and therefore also result in smaller slices. A dynamic analysis records the targets of program pointers during actual program execution. That is it only captures the information in that particular program execution. This perfectly gives us a lower bound of the whole analysis. And with a sufficient number of program inputs and execution runs, the answer will be close to optimal.

Since the dynamic sets are a lower bound for the results of any static analysis, we can use them to obtain an upper bound on the potential improvement of slice sizes that might be achieved by using more precise pointer analysis algorithm in slicing. Slices computed using the dynamic points-to sets are unsound since they may exclude the statements that should in fact be present. However, if we focus on the potential improvement in slice size and not in their use to determine program behavior, their soundness is kind of irrelevant to think about.

The conservative static analysis will result in conservative assumptions about data dependences in a program, contributing to a larger slice size. We shall bear in mind that the information provided for the usage like program understanding and debugging is a hint for the human. We cannot expect a human to quickly find a needle in the hay. Just like in the internet, we need to use search engine to find what we want; with the slice build from dynamic points-to set, people can narrow down their search space. Besides, with the development of software test techniques and input generation techniques, researchers can expand the slice build from dynamic points-to set,
eventually approximate the optimal slice. To sum up, we believe the construction of
dynamic points-to set will be beneficial to many real applications in software
engineering.
Chapter 6

Conclusion and Future work

In this thesis, we have presented a new method to compute the pointer alias in a strong-typed language Java. This method uses different program execution traces to replace program calling graph. Union the information discovered in different traces, we can obtain the result of “maybe-alias”.

The time complexity of the proposed algorithm is $O(M*N^2)$ which is same as the worst-case of Andersen’s $O(n^3)$. In order to make this algorithm work better, next step we need to find out how to choose representative inputs for programs. With a small domain of inputs, the discovered information approximates to be complete. Testing and debugging techniques in software engineering shall be a complement to our algorithm. Besides, the combination of dynamic and static analysis shall work better than solely using only one side. The soundness of static analysis is important to decide the property of a program, yet the unsound but precise dynamic analysis turns out to be much helpful in the environment of program debugging and slicing.
Reference


