JAAT: A Practical Alias Analysis Tool for JAVA Programs

Fumiaki OHATA, Yusuke YAMANAKA, Kazuhiro KONDO and Katsuro INOUE

The authors are with the Software Engineering Research Group (C/O Katsuro Inoue), Division of Software Science, Graduate School of Engineering Science, Osaka University, 1-3 Machikaneyama-cho, Toyonaka, Osaka 560-8531, JAPAN. E-mail: {oohata, y-yamank, kondou, inoue}@ics.es.osaka-u.ac.jp.

Abstract

When an expression refers to a memory location that is referred to by another expression, we say that there is an alias relation between those expressions. Alias analysis, i.e, the extraction of such relations is essential for efficient maintenance of object-oriented programs.

Although many researchers have already proposed analysis methods and implemented prototype tools for object-oriented programs, difficulties still remain in applying such methods to practical tools in the sense of precision, extensibility, and scalability. Focusing mainly on practical implementation, we propose an alias analysis method for object-oriented programs. This method employs a two-phase, on-demand, instance-based, and extensible algorithm.

We have implemented the proposed method as JAAT. JAAT can analyze large programs such as JAVA Developer's Kit (JDK) class library. We have applied JAAT to various large JAVA programs and confirmed JAAT's performance.

Keywords

Alias, Object-Oriented programs, Scalability, JAVA

I. INTRODUCTION

An *alias relation* between two expressions e_0 and e_1 in a source program is a relation such that e_0 and e_1 may possibly refer to the same memory location during program execution. Alias relations are generated by various situations such as parameter passing, reference variables and indirect reference with pointer variables. We say that e_0 is an *alias* of e_1 (and vice versa) when there is an alias relation between e_0 and e_1. An alias relation is an equivalence relation, and we call its equivalence class an *alias set*. *Alias analysis* is an extraction method of alias sets by static analysis, and alias analysis is needed for various purposes such as *compiler optimization*[1] and *program slicing*[23].

Alias analysis was first proposed for traditional procedural languages such as C and Pascal as part of the static analysis of pointer variables[2], [16], [22], [28]. Concepts such as *class*, *inheritance*, *dynamic binding* and *polymorphism*[7] have been introduced through Object-oriented (OO) languages such as C++[3] and JAVA[14], and alias analysis methods for OO programs have been devised[24], [27]. [24], [27] focus mainly on analysis algorithms, and they have not explored practicability and scalability as analysis tools.

We are interested in developing a practical alias analysis tool for OO languages such as JAVA; however, implementation of already proposed approaches remains difficult as

discussed in [21].

The primary issue for such tools is scalability. Since programs being developed have become larger and class libraries associated with the developed programs tend to be huge and complex, the analysis should satisfy scalability in handling large programs. However, many analysis methods produce poor results due to inherent implementations and more studies need to be conducted[21].

Another issue is the usage and approach of the analysis. Most previous works focused mainly on compiler optimization and back-end for data-flow analysis as applications of the alias analysis. For such purposes, all alias relations in the program need to be extracted by the analysis. Nonetheless, we believe that alias analysis is useful for program debugging and understanding[21]. For program debugging and understanding, not all the alias relations are needed at one time; only user-requested relations are to be extracted quickly. Thus, we have to newly devise an on-demand, incremental analysis approach, which can be used effectively in an interactive environment.

To resolve these issues, we propose in this paper a new alias analysis method for OO programs. We have developed a two-phase approach, in which intra-module analysis is done in Phase 1 for whole programs and libraries, and inter-module analysis is done in Phase 2 only for an user-demanded target. This two-phase approach greatly contributes to the overall performance of the analysis.

In this paper, we also consider analysis precision. Flow-sensitiveness of the analysis is an important factor in determining the analysis precision and cost[18], [19], [20], [30]. A flow insensitive approach is easily implemented in general; however, this approach produces poor results. Also, for OO programs, an instance-based analysis for individual objects instantiated from a single class will improve the analysis precision programs when compared to class-based analysis, although the instance-based approach generally requires high analysis cost. We take a flow-sensitive instance-based analysis in this research to aim for high analysis precision with practically affordable analysis cost.

The two-phase approach is also suitable for extending analysis algorithms since software components for each phase and sub-phase are easily replaceable. Therefore, a new analysis policy for a precision-cost compromise will be introduced.

We have implemented the proposed algorithm as JAAT. JAAT considers scalability in the sense that it can analyze large programs with reasonable computation time. For example, the analysis time of 32,037 lines of JAVA programs with 119,564 lines of JDK library was 168 seconds in Phase 1 and less than 1 milli second in Phase 2. This result shows that the user can immediately get the resulting aliases on-demand for the user-specified analysis target after the preparation of Phase 1. Also, the result was fairly precise in the sense that for our sample programs, 4.42 – 15.37 aliases were found due to the instance-based approach, which is 32 – 63% less than the traditional class-based analysis.

As an additional feature of JAAT, it can save internal syntactic and semantic information as an external XML database, and restore the information, in order to improve reusability of analysis results. Also, JAAT provides useful Graphical User Interface (GUI), which shows the resulting aliases using several visualization ways and helps the user's program debugging and understanding activities.

In Section II, we give a brief overview of an alias and its analysis for OO programs. In Section III and Section IV, we propose an alias analysis method for OO programs. In Section VI, we introduce an implementation of the proposed method and evaluate its effectiveness using several sample programs. In Section VII, we discuss the evaluation results with respect to related works and also describe an extension of our method to alias analysis of pointer variables in C or C++. In Section VIII and Section IX, we conclude our discussion with a few remarks and describe our future works. In the Appendix, a detailed proposal of the alias analysis algorithm and its algorithm complexity are presented.

II. PRELIMINARY

A. Example of Aliases

Alias analysis is useful for *program debugging* and *program understanding*. To give an intuition of this, we present an example here. Fig. 1(a) shows a sample JAVA program and Fig. 1(b) shows its execution outputs. This program computes the salaries of employee `Emp` and manager `Mng`. The salary of the manager should be higher than the employee. However, the program execution output is incorrect because a salary addition was made

to `Emp`. When the user recognizes such a fault, he/she computes the aliases for reference variable `Emp` at line 32. In this paper, we call such a target expression of the alias analysis the *alias criterion* (or simply *criterion*), and it is specified by a tuple <s, e>, where s is a statement in the source program and e is an expression at s. The shadow expressions are the resulting aliases for <32, `Emp`> (which is also an alias itself, and therefore, boxed with a bold line). We can see around those shadow expressions, and can identify a fault at the salary addition statement at line 24. By modifying the statement `e.add_salary(200)` to `add_salary(200)` at line 24, the program will compute an expected result as shown in Fig. 1(c).

B. Alias Analysis

Alias analysis methods are roughly divided into two categories, *flow insensitive alias analysis (FI analysis)* and *flow sensitive alias analysis (FS analysis)*.

B.1 Flow Insensitive Alias Analysis (FI Analysis)

In FI analysis, we do not take into account the execution order of each statement in the source program[2], [22]. To compute FI aliases, we usually use an *alias graph* as shown in Fig. 2(b). An alias graph is an undirected graph, in which each node represents an expression that refers to a particular memory location. Each edge represents a possible alias relation between two nodes, which occurs on both sides of an assignment statement and on the actual and formal parameters.

In Fig. 2(a), when we specify <7, `c`> as the alias criterion, we get aliases {`a`, `b`, `new Integer(1)`, `new Integer(2)`}, which are all reachable nodes in the alias graph from the criterion node.

B.2 Flow Sensitive Alias Analysis (FS Analysis)

In FS analysis, we consider the execution order of statements using a *Reaching Alias set (RAset)*[16], [28]. A RAset for statement s, denoted by $RA(s)$, is a collection of *alias sets*, which possibly exists just before the execution of s. Each alias set is represented by a set of tuples (t, f) (t is a statement in the source program and f is an expression at t). Fig. 2(d) shows RAsets for each statement in Fig. 2(c). In order to compute the aliases

for $<s, e>$, we search $RA(s)$ for an alias set that contains e.

At $RA(7)$ in Fig. 2(d), since an alias set [(6, `c`), (2, `a`), (6, `a`), (2, `new Integer(1)`)] contains variable `c`, we get the result as shown in Fig. 2(c).

Since FS analysis considers the execution order, it generally requires a larger amount of CPU time and memory space than FI analysis; however, FS analysis can extract more accurate alias relations than FI analysis. In Fig. 2, we can see the difference in the accuracy between the two methods. Previous works [18], [19], [20], [30] show empirical comparisons of these analyses. In this paper, we focus on FS analysis for more accurate analysis results.

C. Alias Analysis for Object-Oriented Programs

Alias analysis methods for OO programs have been proposed as the extension of analysis methods for procedural programs[16], [17], [22], [28], [30]; however, some issues still remain to be solved for the implementation of practical alias analysis tools for OO programs. Here, we consider three issues, as follows:

[a] Overall computation time for analysis

In order to implement a practical analysis tool, overall computation time is one of our main concerns. We have chosen relatively expensive FS approach, where all possible method-call-paths should be analyzed to compute the RAset. When the target program contains loops or recursive method calls, it should be analyzed until the increase of the RAset 'settles down'. In other words, when a $RA(s)$ for statement s changes during alias computation, we must re-compute the RAsets for all statements that are possibly affected by $RA(s)$. Thus, convergence and total computation time are important factors of the tool.

[b] Effective reuse of analysis results

The alias analysis tool should be used repeatedly with various alias criteria or with slightly distinct target programs. In such cases, we do not want to re-analyze the over all programs. In traditional FS alias analysis, $RA(s)$ for statement s is computed by analyzing all statements in the source program that contain s along the execution order. Therefore, when another statement t is modified, we might simply re-compute

$RA(s)$ for each statement s in the source program even if $RA(s)$ is not affected by the modification of t. We are interested in finding an effective approach that re-computes RAsets only affected by the modification.

[c] Improvement of analysis precision by separating each instance

In OO programs, each object has its own state and behavior even if they are instantiated from the same class. Fig. 3 shows part of a JAVA program that refers to attribute `s` of objects `x` and `y`. In this program, we prefer to have two independent alias sets [(3, `x.s`), (8, `s`)] and [(4, `y.s`), (8, `s`)]. However, if we apply a simple analysis approach such that all objects instantiated from the same class share the inner alias information, we get only one alias set [(3, `x.s`), (4, `y.s`), (8, `s`)] where (3, `x.s`) and (4, `y.s`) unwillingly fall in the same alias set. In order to increase the analysis precision, each object should retain its inner alias relations separately; however, applying this notion to traditional FS analysis methods would cause consumption of large memory space. We devise an efficient approach.

III. Analysis Overview

A. Approach

To solve above the mentioned issues, we will adopt the following analysis policies.

Policy 1 Compute intra-module alias relations for each module such as class, method, and so on.

Policy 2 Compute inter-module alias relations on-demand.

Utilizing these two policies, modularity and independence of the analysis will be established. This is particularly important because in OO programming, we usually have to analyze class libraries in addition to the developed codes. Those class libraries tend to be large and their analysis cost also becomes large. Thus, the modularized analysis approach is essential.

In this paper, we divide alias relations into two categories, *inner alias relation* and *outer alias relation*. The followings are their definitions.

inner alias relation: alias relations that are common among all objects instantiated from the same class

outer alias relation: alias relations that are not common among objects instantiated from the same class

Outer alias relations are generated by external objects.

Policy 3 Distinguish outer alias relations on the objects instantiated from the same class, and analyze inner alias relations in advance, and compute outer alias relations on-demand.

This policy holds for issue [c].

We also distinguish method invocation information for each instance. Such information is named *object context*, which is used where computing outer alias relations for each instance.

The object context for alias set $\mathcal{A}$, denoted by $OC(\mathcal{A})$, is a set of instance methods that might be invoked on the objects in $\mathcal{A}$. This context is defined by the following process:

$OC(\mathcal{A}) := \phi$.
repeat
When method m appears in expressions such as
$a.m(\ldots)$ and a is in $\mathcal{A}$,
$OC(\mathcal{A}) := OC(\mathcal{A}) \cup \{m\}$.
When method m is an instance method of the objects
in $\mathcal{A}$ and its invocation appears in $OC(\mathcal{A})$,
$OC(\mathcal{A}) := OC(\mathcal{A}) \cup \{m\}$.
until $OC(\mathcal{A})$ is unchanged.

Using OC($\mathcal{A}$) when we compute the aliases for $a.i$ on condition that a is in $\mathcal{A}$, we can limit the candidate methods to be considered further. In other words, we can exclude the instance methods that can not be invoked in the objects referred to by expressions in $\mathcal{A}$.[1]

We also consider two variants of object context, *flow insensitive object context (FIOC)* and *flow sensitive object context (FSOC)*. The former takes into account the method invo-

[1]If we can not specify the unique $\mathcal{A}$'s type, we might have more than one method; *method overriding* causes such a situation.

cation order, whereas the latter does not. FSOC extracts more accurate alias information than FIOC.

For example, suppose we compute the aliases for <13, `return(i)`> in Fig. 4[2] assuming that the underlined methods `Calc::Calc()`, `Calc::add()` and `Calc::result()` are invoked in this order. Since Fig. 4(b) uses FSOC, it can generate more precise results than Fig. 4(a) which uses FIOC. FSOC can exclude two expressions, `i` and `new Integer` in `Calc::Calc()`, since we know that the definition of `i` at line 4 is always discarded at line 10.

FSOC requires more analysis cost than FIOC; on the other hand, FIOC might be less precise than FSOC. Therefore, we need to compromise between analysis cost and analysis precision. In this paper, we use the FIOC approach.

Based on Policy 1 – 3 and object context, we propose the following two-phase approach.

Phase 1: Intra-module analysis in advance:

(a) Construction of AFG (defined later) by intra-method analysis

(b) Construction of MFG (defined later) by intra-class and inter-method analysis

Phase 2: Inter-module analysis on demand: Alias computation using AFG and MFG along with object context

The details of this approach are discussed in Section IV.

B. Other Issues with OO Program Analysis

OO languages like JAVA, contain more features than traditional procedural languages. We use the following approaches for each feature.

- Inheritance:

  The inheritance concept causes other features such as method overriding and dynamic binding. In addition, we must take virtual method invocation mechanisms into account, so that MFG construction algorithms consider the inheritance. The details of our consideration will be shown later.

- Method overriding and Dynamic binding:

[2]Since $return(x)$ and x always satisfy alias relations, we only show x as aliases in the figures.

Method overriding might generate two or more methods that have the same signature in a class hierarchy. Since the determination of the invoking method depends on the reference-type of the object that receives a message, static identification of the actually invoked method is difficult. This difficulty stems from the undecidable type of receiver object without program execution; however, with alias analysis we can infer such a type more accurately.

For example, when we identify the invoking method of expression $a.b()$, we can use the alias information of a in order to infer the reference-types of the instances that might be referred to by a.

- Thread and Exception:

  Here, we do not deal with threads and exceptions. We concentrate on more general control flows such as loop statement, conditional statement and method call discussed in Section VII.

IV. Details of Analysis

A. Phase 1: Construction of AFG and MFG

A.1 Phase 1(a): Construction of AFG

An *alias flow graph (AFG)* is an undirected graph, which shows FS alias relations inside a single method. A node represents either

- an expression that refers to an object (e.g., a reference variable, an instance creation expression, or a method invocation) or
- parameters to/from a method or an instance.

The former node is called an *AFG normal node* and the latter is called an *AFG special node*. We prepare special nodes as listed in TABLE I. An edge in AFG denotes an alias relation immediately determined by the intra-method alias analysis. We call such an alias relation a *direct alias relation*. Examples are aliases created by assignment statement, variable's definition and its use (def-use relation), and assignment of parameters to/from special nodes. Such aliases are easily obtained by RAset-based FS may-alias analysis inside methods and classes[16], [28]. Also, a path formed with more than one edge shows an *indirect alias relation*.

TABLE I

AFG SPECIAL NODE

Node	Description	Location
Actual-Alias-in (AA-in)	Alias passed by actual parameter to callee.	caller
Formal-Alias-in (FA-in)	Alias passed by formal parameter to callee.	callee
Actual-Alias-out (AA-out)	Alias passed by actual parameter to caller.	caller
Formal-Alias-out (FA-out)	Alias passed by formal parameter to caller.	callee
Method-Alias-out (MA-out)	Alias passed by return statement to caller.	callee
Method-Invocation (MI)	Alias passed by return statement to caller. (Parent node for AA-in(out) node)	caller
Instance-Alias-in (IA-in)	Alias passed by instance attribute to method.	-
Instance-Alias-out (IA-out)	Alias passed by instance attribute from method.	-

Note that an expression specifying an attribute b (or a method $b()$) associated with an instance a is denoted by $a.b$ (or $a.b()$) in JAVA. In such a case, we say that the node for a in AFG is a *parent* of the node for b, and the node for b is called a *child* of the node for a. Such a parent-child relationship is used for the alias computation in Phase 2. Also, parent-child relationships between an MI node and its corresponding AA-in nodes exist.

A.2 Phase 1(b): Construction of MFG

A *method flow graph (MFG)* is a directed graph, which represents the caller-callee relations of methods in a single class. An *MFG node* denotes the definition of each method, and when a method A possibly calls a method B, an *MFG edge* is drawn from the node for A to the node for B.

The consideration of inheritance and method overriding concepts of methods are also important. We have to make MFG while considering such concepts (this will be discussed in Section V-A.2). Each MFG is constructed by the intra-class analysis of caller-callee relations using class inheritance analysis.

B. Phase 2: Alias Computation Using AFG and MFG

We compute aliases $\mathcal{A}(X)$ for a reference-type expression X by AFG traversal. $\mathcal{A}(X)$ means a set of expressions that might refer to the same memory location as X. X is also an element of $\mathcal{A}(X)$. Note that Fig. 5 includes some of the definitions of the symbols used in this paper.

We have adopted the following principles for alias computation using AFG:

1. When we compute the aliases for node C with a parent node P, first we compute P's aliases $\mathcal{A}(P)$, and then we collect information about $\mathcal{A}(P)$, such as
   - types for $\mathcal{A}(P)$ and
   - $\mathrm{OC}(\mathcal{A}(P))$,

   after which we compute C's aliases.

   There are many cases where we can not compute the aliases for C without $\mathcal{A}(P)$ and their types. If we omitted P's alias computation, we would have had to consider that P could refer to all the objects instantiated from the classes derived from P, so that C's alias result would be enlarged.

   We have named this approach '*parent-first-child-last* approach'.
2. When we reach an MI, MA-out, an FA-in, FA-out, an AA-in or AA-out node during AFG traversal, using MFG we determine the callee or caller method corresponding to the node, and then we traverse from the corresponding MA-out, MI, AA-in, AA-out, FA-in or FA-out node, respectively.

AFG is traversed on the whole program or several classes beyond the class boundary with MFG information. When programs hold recursive method calls, AFG traversal continues until the increase of the resulting aliases settles down. The details of the AFG traversal algorithm and its termination are shown in Appendix I. The complexities of Phase 1 and 2 are shown in Appendix II.

V. Example

A. Phase 1: Construction of AFG and MFG

A.1 Phase 1(a): Construction of AFG

Fig. 6 shows a small Java program and its AFG. Nodes in AFG are shown as circles with expressions inside, and edges are denoted with solid lines. Other strings out of those nodes (e.g., `Integer`, `=`, `Integer b, c;`) are comments used to identify the occurrences of expressions and to help the reader imagine the original source text. In Fig. 6(b), since `new Integer(0)` is assigned to `a` in the source program, you can see that a node for `new Integer(0)` is connected to node `a` with an edge. This edge represents a direct alias

relation.

Fig. 7 is a JAVA program with two class definitions, and its AFG. Variable `i` appearing at each righthand-side expression (line 7 and 10) is a reference-type instance variable. Thus, AFG special nodes IA-in[`i`] and IA-out[`i`] are employed for each method in class `Calc`. Also, the expression for the return value, `return(i)`, at line 13 is a reference to an object. Therefore, an AFG special node MA-out is created for method `Calc::result()`.

Also, `b.result()` at line 24 in Fig. 7(a) is represented in AFG with a parent node φ(`b`) and a child node φ(`result()`).

A.2 Phase 1(b): Construction of MFG

Fig. 8 shows a sample program and its MFG. Method `p()` is not defined in class `B`, and method `A::p()` is executed when `p()` is activated on the `B`'s object. In this case, method call to `q()` appearing in `A::p()` causes activation of `B::q()`, not `A::q()`. Thus, the resulting MFG for class `B` is as shown in Fig. 8(b).

Fig. 9 shows MFGs for class `Calc` and class `Test` shown in Fig. 7(a). Since neither classes do not have intra-class method calls, there is no MFG edge.

B. Phase 2: Alias Computation Using AFG and MFG

We show an alias computation process for <24, `c`> (boxed with a bold line) in Fig. 10, which is the same as Fig. 7(a). Also, lower-case letters `b` and `c` represent expressions in the program, and capital letters B and C represent nodes in AFG.

1. Start AFG traversal from AFG normal node φ(`c`), and immediately reach MI node φ(`result()`) (Fig. 11).
2. Since φ(`result()`) has parent node φ(`b`), first compute `b`'s aliases to specify the object related to `result()`'s aliases.
   (a) Compute `b`'s aliases $\mathcal{A}(B)$.
   (b) Compute the types of $\mathcal{A}(B)$ using class instance creation expressions included in $\mathcal{A}(B)$. In this case, the type is determined to be `Calc`.
   (c) Compute OC($\mathcal{A}(B)$). The result is {`Calc::Calc()`, `Calc::add(int c)`, `Calc::result()`}.
3. Since alias computation for `b` indicates that it refers to the objects that are instantiated from class `Calc`, traverse AFG from MA-out node in ψ(`Calc::result()`) (Fig. 12).

(a) Reach IA-in[i] in ψ(`Calc::result()`).

(b) Traverse AFG from IA-out[i] in $\{M \mid \psi^{-1}(M) \in \mathrm{OC}(\mathbf{this})(\equiv \mathrm{OC}(\mathcal{A}(B)))\}$.

Fig. 10 shows `c`'s aliases (masked) and OC($\mathcal{A}(B)$) (underlined). Since OC($\mathcal{A}(B)$) does not contain `Calc::inc()`, expressions in `Calc::inc()` are excluded from the candidates for `c`'s aliases.

VI. A Java Alias Analysis Tool (JAAT)

We have implemented the proposed method as a Java *Alias Analysis Tool (JAAT)*. Using JAAT, we have executed several programs and obtained data.

A. Overview of JAAT

JAAT consists of three subsystems, the *analysis subsystem*, the *XML database subsystem* and the *User Interface (UI) subsystem*. Fig. 13 shows the structure of JAAT. We will overview each subsystem.

A.1 Analysis Subsystem

The analysis subsystem consists of three components: The *syntax analyzer* analyzes Java source files and generate syntax trees[3]. The *semantic analyzer* proceeds with a semantic analysis that creates symbol tables and extracts declare-refer relations among identifiers and generates semantic trees. The *alias analyzer* that generates MFGs and AFGs as Phase 1, and computes the aliases for the alias criterion specified by the user's request as Phase 2. The alias analyzer returns the resulting aliases to the UI subsystem.

A.2 XML Database Subsystem

Generated trees and graphs are proprietary data structures, and are placed into the memory space of JAAT, as are many other analysis tools[4], [25]. Since the translation from a source program to the corresponding semantic tree is a fairly time-consuming process, we would not want to discard analysis results for the analysis sessions. Thus, we build a database for semantic trees. This feature improves the reusability of the analysis results along with the AFG/MFG approach.

[3]The syntax analyzer is generated by ANTLR[9].

We use an *eXtensible Markup Language (XML)*[11] database that holds semantic tree information. The *XML converter*[4] converts semantic trees to XML documents and vise versa. The XML database is a collection of XML documents, each of which represents one JAVA class or interface. In order to keep the XML database compact, we do not preserve source text information, such as line breaks, spaces and comments. The formatted source text in JAVA can be re-generated by the application that we have also implemented. In Section VII, we will discuss the effectiveness of the XML database with experimental results and some XML applications.

A.3 UI Subsystem

The UI subsystem[5] has two main functions, editing programs and visualizing the resulting aliases. When the user specifies an expression as an alias criterion, the UI subsystem sends a query to the analysis subsystem. It displays the analysis results on the *text window* (Fig. 14(a)). The text window shows the resulting aliases with colored backgrounds. Statements without any aliases can be compressed on the screen with smaller fonts by the user's commands. The user can focus attention to the statements with aliases, and other statements, masked by the compressed display mode, can be easily enlarged and displayed normally if required.

Fig. 14(b) is the *alias tree window* that shows the resulting aliases like a tree, on which each node denotes class, method or expression. Since aliases tend to be found widely in many methods or classes, the alias tree window enables the user to grasp the overall distribution. In Fig. 14(b), two alias trees (left tile) for two alias criteria are shown, on which the resulting alias trees are spread from **Alias[0]** and **Alias[1]** nodes, respectively. When clicking a node, the user can see the information (right tile) about that node.

B. Evaluation

In order to explore the applicability of JAAT, we have applied it to various sample programs. TABLE II shows features of the sample programs. Note that we must analyze not only these sample programs but also all related classes in JDK for inter-method alias

[4]The XML converter uses libxml[13] as an XML parser.
[5]The UI subsystem uses Gtk−− tool kit[8].

TABLE II

CHARACTERISTICS OF ANALYZED PROGRAMS

Programs	Sample Program		Related Classes in JDK	
	Number of Files	Number of Lines	Number of Files	Number of Lines
TextEditor	1(0.1%)	915(0.8%)	802(99.9%)	114,887(99.2%)
WeirdX (X server)	47(5.5%)	16,703(12.6%)	815(94.5%)	115,977(87.4%)
ANTLR (Parser Generator[9])	129(32.6%)	18,775(35.7%)	267(67.4%)	33,847(64.3%)
DynamicJava (JAVA Interpreter)	242(22.7%)	32,037(21.1%)	825(77.3%)	119,564(78.9%)

analysis. For example, `TextEditor` is composed of one file 915 lines long. `TextEditor` directly and indirectly uses the classes in JDK, which is in 802 files with a total of 114,887 lines (99.2% of the overall total lines). This data shows that a heavy effort on analysis must be done for the related classes.

B.1 Computation Time of Phase 1(a)

Our modularized analysis is effective in that we only have to re-analyze modified parts of the programs when small parts of the program are modified. On the other hand, the RAset-based analysis[16], [28] can not retain analysis results for each module separately; therefore, the overall program might have to be re-analyzed. Although user-development programs are often modified when developing a JAVA program, their related classes are seldom modified.

TABLE III(a) shows AFG construction time for sample programs and their related classes. The analysis time for its related classes is much longer than for those of the sample programs. For example, `TextEditor` itself requires only 100 milli seconds, and its related classes require 99,980 milli seconds. When we modify `TextEditor`, we do not need to re-analyze its related classes, but only the `TextEditor`.

B.2 Computation Time of Phase 1(b)

TABLE III(b) shows MFG construction time for sample programs and their related classes. Since MFG construction time does not depend on the programs size, but on the number of intra-class method calls, the MFG construction time of the sample program is not always longer than that of its related classes. For example, `DynamicJava` itself

TABLE III

EXPERIMENTAL RESULTS

Programs	Sample Program	Related Classes in JDK
TextEditor	100	99,980
WeirdX	14,220	100,540
ANTLR	12,830	23,480
DynamicJava	56,260	110,150

(a) computation time of Phase 1(a) [ms]

Programs	Sample Program	Related Classes in JDK
TextEditor	10	390
WeirdX	20	390
ANTLR	100	80
DynamicJava	960	450

(b) computation time of Phase 1(b) [ms]

Programs	Average
TextEditor	0.65
WeirdX	0.29
ANTLR	0.17
DynamicJava	0.07

(c) computation time of Phase 2 [ms]

Programs	Instance-based	Class-based
TextEditor	4.42	8.31
WeirdX	15.37	24.54
ANTLR	5.94	18.77
DynamicJava	9.16	17.19

(d) average number of detected aliases [nodes]

— PentiumIII-667MHz-512MB(Debian GNU/Linux)

requires 960 milli seconds, but its related classes require only 450 milli seconds. However, the overall MFG construction time is much smaller than the AFG construction time.

B.3 Computation Time of Phase 2

TABLE III(b) shows average AFG traversal time for all AFG normal nodes in each sample program. According to TABLE III(b), it is clear that Phase 2 takes much less computation time than Phase 1. In the case of `TextEditor`, 0.65 milli seconds is much smaller than the AFG construction time (100,080 milli seconds = 100 milli seconds + 99,980 milli seconds) for `TextEditor` and its related classes.

Our on-demand approach might be unsuitable as back-end for data-flow analysis and compiler optimization which needs whole alias analysis results. However, when we do

TABLE IV

COMPARISON BETWEEN JAVA SOURCE FILES AND THE XML DATABASE IN CONSTRUCTION OF THE SEMANTIC TREE

From	Construction time[s]	Data Size[MB]
JAVA source files	37	25
XML database	24	62

not need to compute the aliases for all expressions, or when we implement an interactive programming support tool with alias analysis features, our method is a practical choice.

B.4 Average Number of Detected Aliases

The proposed method uses the *instance-based* approach that can distinguish outer alias relations on objects instantiated from the same class. On the other hand, if we use the *class-based* approach that shares outer alias relations with other objects instantiated from the same class, analysis precision will decrease. Traditional alias analysis methods are not concerned with the instance-based approach, but only with the class-based approach.

TABLE III(c) shows the comparison results between those two approaches with regard to the average number of detected aliases for all AFG normal nodes in each sample program. For example, the instance-based approach generates more accurate results than the class-based approach (4.42 nodes v.s. 8.31 nodes) in `TextEditor`. The average size of aliases is about 32 – 63% of the class-based approach; therefore, our approach is of practical value.

B.5 Effects of The XML Database

Since the XML database is a text-based database, it requires higher analysis costs than in a proprietary database. However, we think that the reusability of the data by the XML database is important. As mentioned in Section VI, since there are many programs and libraries that can deal with XML documents, we can easily implement XML applications for program debugging and understanding activities that access the XML database. The advantages of the XML database are as follows:

- Decreased analysis costs

  The implemented XML database holds semantic tree information. In order to evaluate its effectiveness, we have compared the XML database and the JAVA source files from

the viewpoints of time and space.

We have used all source codes for the JDK 1.3 class library. The XML database for semantic trees took 45.6 seconds to be generated.

We compared the construction time for all semantic trees from the XML database and the JAVA source files, and compared disk space to store them as shown in TABLE IV. It took 37 seconds to construct semantic trees from the JAVA source files and only 24 seconds to construct the trees from the XML database. While the data size became large, the improvement of the construction was satisfactory.

- Rapid development of applications

  We have implemented three XML applications (see Fig. 13):
  - *XML-HTML converter* (2000 lines in *eXtensible Stylesheet Language Transformations (XSLT)*[12]),
  - *XML-JAVA converter* (2000 lines in XSLT, 1500 lines in C++) and
  - *XML-XML converter* (identifier replacement program, 600 lines in XSLT).

  Fig. 15 shows the result of applying the XML-HTML converter. We can see the JAVA source text with HTML tags that are generated from the XML representation of class `java.lang.ClassLoader`. Using WWW browsers, we can also traverse the declaration-reference chains for identifiers (these chains are represented by links) such as types and methods and variables.

VII. DISCUSSION

Our proposed method for alias computation, which consists of two analysis phases, has produced effective results, and problems with the alias analysis of OO programs are sufficiently resolved.

In this section, we compare our results and related works, and also show an extension of our method to programs with pointer variables in ordinary languages.

A. Comparison with Related Works

In this section, we discuss our work and related works from several viewpoints.

A.1 Two-phase Approach

Several prior research studies also analyze each module in advance[17], [27]. However, in these studies, each element $\mathcal{A}$ (alias relation) in an RAset holds conditions if a specific alias relation $\mathcal{A}_0$ really exists (if **true**, $\mathcal{A}$ exists). These conditions are used for indirect alias relations; however, all combinations of accessible variables should be taken into account as candidates for $\mathcal{A}_0$. In our method, since each AFG edge represents a direct alias relation, we can easily extract each indirect alias relation as an AFG path.

Such conditional-based algorithms used in [17], [27] would be suitable as back-end for data-flow analysis (e.g., program slicing), which needs whole alias relations in target programs. Since we focus mainly on program debugging and understanding activities using alias information itself, more simple representation is useful. On the other hand, since our AFG traversal algorithm is designed in order to compute the aliases for single alias criterion specified by the user, it is unsuitable for data-flow analysis (however, we will be able to define new AFG traversal algorithm for computing whole alias relations in target programs if need).

Also, [17] focuses on ordinary procedural languages only, and [27] is not applied to aliases with pointers, where both methods have been implemented as prototype tools only.

A.2 Instance-based Analysis

The instance-based approach was proposed in *Object slicing*, which is a slice extraction method for OO programs[5]. [5] extends the *System Dependence Graph (SDG)* and defines the traversal algorithm to compute slices with regard to a specific object; however, [5] assumes that the pointer or alias analysis have been already performed by another method. To get a practical alias analysis tool, combining an alias analysis method and the instance separation method into a single method is very important, as we have proposed here.

A.3 On-demand Alias Extraction

We applied an on-demand approach to alias analysis. Although alias information has been used for other analyses such as compiler optimization, data-flow analysis and so on[1], [2], [16], [22], [24], [28], alias information is itself useful for program debugging and understanding of JAVA programs, because, in general, JAVA programs have many aliases,

which are caused by reference-type expressions.

Since all alias information in the target program is not necessary on program debugging and understanding activities, we believe on-demand (or query-based) analysis will be a cost-effective approach.

A.4 Extensibility of The Algorithm

In Section II, we introduced FS analysis and FI analysis. In general, their algorithms are different, and are implemented independently; the former uses RAsets, the latter uses point-to or alias graphs. Since each analysis has merits and demerits on analysis costs and analysis precision, we think that practical alias analysis tools should be implemented with both methods. In order to compare various alias analysis methods, many prototype tools have been implemented[18], [19], [20], [30]. Since those tools are for prototypes, changing and extending analysis algorithms are not considered.

Using our AFG-based alias analysis approach, we can share intra-method analysis results with several alias analysis algorithms. Each AFG holds intra-method FS alias analysis results for its corresponding method. Although FS analysis requires much larger computation time than FI analysis, intra-method analysis is generally more effective than inter-method analysis; applying FS policy to the intra-method alias analysis would be a practical choice.

Since various alias analysis methods can be described in the form of the AFG traversal algorithm, we can compromise between analysis cost and precision by changing traversal algorithms. FSOC and FIOC are examples of this approach. So are instance-based and class-based algorithms (their comparison results were shown in TABLE III(c)).

Currently, the AFG traversal algorithm in Appendix I does not completely satisfy FS due to the FIOC approach. In order to apply FSOC, flow directions should be represented using direct edges.

Since distinguishing may-alias relations and must-alias relations is effective for more precise analysis results, applying this approach to AFG construction and traversal algorithms would be more powerful by its flexible mechanism on algorithm selection.

A.5 XML Database

JavaML also describes semantic tree information using XML[10]. [10]'s approach resembles our XML database. However, since JavaML considers only intra-class declaration-reference relations about identifiers, we are afraid that JavaML can not restore semantic trees from generated XML documents. Since our generated XML documents can take into account both inter-class and intra-class declaration-reference relations, semantic trees can be easily restored from XML documents.

A.6 Thread and Exception

Since FS analysis considers the execution order of statements, its analysis precision depends on the precision of control-flow information. Thus, if we can collect more precise control-flow information, FS analysis results should become more precise.

Currently, since JAAT's control-flow representation does not consider the control-flows caused by threads and exceptions, its resulting aliases will become imprecise with the programs that contain threads and exceptions. However, since many researchers have already proposed control-flow analysis methods for thread and exception[15], [29], we will be able to adopt the algorithms in [15], [29] to construct the improved control-flow representation that takes thread and exception into account.

B. Alias Analysis for Pointer Variables

In Section IV, we have described an alias analysis method for reference-type expressions in JAVA. This method has been also extended to the alias analysis of ordinary procedural programs with pointer variables such as C or C++. In such a case, we need special consideration of indirect reference by pointers (especially, higher-order pointers) because a callee can modify its caller's alias relations using n-order (n $\geq$ 2) pointer variables even if parameter passing uses a passed-by-value mechanism. The following are the core parts of the extension.

B.1 Phase 1(a): Construction of AFG

We prepare n AFG nodes for each n-order pointer variable. Fig. 16 shows a C program and the direct alias relations (i.e, the AFG edge and each element denotes an AFG node)

extracted from the program. Since procedure `assign(char **y, char *x)` uses a second-order pointer variable `y` as the first parameter, we prepare the following nodes:

- AA-in[`&b`] and AA-in[`b`] for actual parameter `b` in procedure `main()`
- FA-in[`y`] and FA-in[`*y`] for formal parameter `y` in procedure `assign(char **y, char *x)`

In addition to AA-in and FA-in nodes, we add the corresponding AA-out and FA-out nodes, respectively.

B.2 Phase 2: Alias Computation using AFG and MFG

There are two operators for pointer variables – indirect operator '$*$' and address operator '&'. Expression $*x$ represents the value in the memory location to which x refers, and expression $\&x$ represents x's memory location. To adapt the alias computation algorithm to these operators, we have also applied the parent-first-child-last approach to them. For example, when we compute the aliases for $*a$, first we compute the aliases for a. The details of the AFG traverse algorithm for '$*$' and '&' will be represented in Fig. 17 as [Cond.8] and [Cond.9] at Step 2.

VIII. Conclusions

We have proposed an alias analysis method for OO programs, which is a scalable and on-demand method with high precision and extensibility. Also, we have implemented this method as JAAT, and evaluated its effectiveness. JAAT consists of three subsystems, the analysis subsystem, the XML database subsystem, and the UI subsystem. The analysis subsystem can analyze large programs such as a JDK class library. We can save semantic tree information to the XML database to decrease the analysis cost and improve reusability of the analysis results.

IX. Future Research

We plan to implement the FSOC approach, and compare this approach with the existing FIOC approach. In addition, we also plan to extend our AFG construction and AFG traversal algorithms for exceptions and threads.

Appendix

I. AFG Traversal Algorithm

In this section, we will describe the AFG traversal algorithm in Phase 2. Phase 2 is the most important phase for our AFG-based alias analysis because we use query-based alias computation, and we can control the analysis cost and precision by changing the AFG traversal algorithm.

Fig. 17 shows the AFG traversal algorithm used in JAAT. The algorithm uses the FIOC and FS may-alias approaches.

In Step 1, we identify AFG node E for e that is specified by the user as an alias criterion.

In Step 2, we start AFG traversal from E where we find reachable nodes in AFG from E. This step requires not only simply traversing AFG edges but also additional processes. Traversing AFG edges is only for intra-method alias relations identified in Phase 1(a); thus, more complicated alias relations such as inter-method or outer alias relations have to be identified here. We perform the following processes according to the classification of nodes during the AFG traversal.

If we arrive at AFG node C on the traversal, we do the following based on the node type of C.

1. C is an AFG normal node that has parent node P:

   Compute P's aliases $\mathcal{A}(P)$ by applying Step 2 recursively to P, and infer $\mathcal{A}(P)$'s types from the instance creation expressions in $\mathcal{A}(P)$. Next, compute OC($\mathcal{A}(P)$). If C's corresponding expression c is an instance variable, traverse AFG from IA-in[c] and IA-out[c] nodes held by OC($\mathcal{A}(P)$)'s corresponding methods, and traverse AFG from the nodes (here, we define each node as N) that satisfy all the following conditions:

   - N's corresponding expression is an instance variable named c, and
   - N's parent node is in $\mathcal{A}(P)$.

2. C is a MI node that has parent node P:

   Compute P's aliases $\mathcal{A}(P)$, infer $\mathcal{A}(P)$'s type, and compute OC($\mathcal{A}(P)$). Next, traverse AFG from MA-out nodes whose corresponding methods are in OC($\mathcal{A}(P)$) and their signatures are the same as that of C's corresponding method invocation

expression.

3. C is an IA-in[c] or IA-out[c]:

   Traverse AFG from IA-out[c] or IA-in[c] held by OC(**this**)'s corresponding AFGs. OC(**this**) means the object context for the currently focused instance (i.e, **this** object). If an alias criterion is in **this** object, OC(**this**) is initialized as a set of constructors and methods that are directly or indirectly called from the constructors. Otherwise, OC(**this**) is initialized with OC($\mathcal{A}(P)$) ($\mathcal{A}(P)$ is an alias set that refer to the instances that have c as an instance variable).

   This process is for the instance-based approach.

   For example, suppose that we have reached IA-in[b] or IA-out[b] computing the aliases for $a.b$. After finishing the alias computation for a, we begin alias computation for b using information about a's type and OC(a). When a's type is A, the alias computation for IA-out[b] or IA-in[b] is proceeded under OC($this$)($\equiv$ OC(a)) in class A. If we can not determine the unique $\mathcal{A}(a)$'s type, we should traverse AFG in two or more classes inferred from $\mathcal{A}(a)$.

4. C is an AA-in[c] or AA-out[c] that has parent node P:

   Traverse AFG from FA-in[c] or FA-out[c] whose corresponding methods are called by MI node P.

   When P has parent node Q, we will apply a dynamic binding policy to the alias computation for P. Thus, P's invoked methods are determined by Q's type, i.e, Q's alias analysis results.

5. C is an FA-in[c] or FA-out[c]:

   Suppose that method m_B is a method that has formal parameter c. First, collect methods m_A that are in OC(**this**) and calls m_B, using MFG. Next, traverse AFG from AA-in[c] or AA-out[c] in m_A's corresponding AFGs.

6. C is an MA-out node:

   Suppose that method m_B is a method that has C's corresponding `return` expression. First, collect methods m_A that are in OC(**this**) and calls m_B, using MFG. Next, traverse AFG from MI nodes in m_A's corresponding AFGs.

7. C is an MI node:

TABLE V

ELEMENTS OF AFG-BASED ALIAS ANALYSIS

Symbol	Description
A	Maximum number of attributes in each class (inherited attributes are also counted)
M	Maximum number of methods in each class (inherited methods are also counted)
L	Maximum number of sum of local variables and parameters in a method
C	Total number of classes in the target program
E	Total number of expressions in the target program
k	Maximum length of parent-child chains (when a chain forms a recursive loop, we do not count the previously visited expressions)

First, extract methods m_A that are in OC(**this**) and are called by C. Next, traverse AFG from the MA-out nodes in m_A's corresponding AFGs.

In our AFG traversal algorithm, we applied the following rules to detect its termination.

- When AFG traversal reaches node N_0 to compute the aliases for c, we record N_0 to RNlist(c); RNlist(c) represents *reached nodes list (RNlist)* for c.
- When the AFG traversal reaches the nodes in the RNlist again, we no longer traverse AFG.
- A RNlist is created for each alias set that may refer to the same memory location. For example, when $a.b$ is an alias criterion, RNlist(a) and RNlist(b) is created.
- Suppose that $a.b.c$ is an alias criterion and we have reached node N_1 on the alias computation for a. When we check if N_1 is in RNlist(a), we should also check if $N_1 \in$ RNlist(b) and check if $N_1 \in$ RNlist(c), respectively.

Since the size of each RNlist is less than E and the number of the RNlist is less than k (the maximum length of the parent-child chain), we can prevent creating RNlists infinitely so that AFG traversal must terminate.

II. Algorithm Complexity

We discuss algorithm complexity for each phase.

TABLE V shows the elements of AFG-based alias analysis.

- Phase 1(a): Construction of AFG

  When the target program has loop statements, we must analyze expressions at most E^2 times. In order to analyze each expression, two or three set operations

are needed. Since set operation cost is proportional to the number of elements in the target set, the time complexity of set operation is covered with $O(A + L)$. Thus, in the worst case, the time complexity is $O((A + L) \cdot E^2)$. The number of AFG nodes is $O(E)$ and the number of AFG edges is $O(E^2)$. The space complexity is $O(E^2)$ in the worst case.

In our experimentation, both the time complexity and the space complexity were near liner order.

- Phase 1(b): Construction of MFG

  One MFG is constructed for each class. Since we must check each expression once in order to find method calls, the time complexity is $O(E)$. Since the number of methods in a class is less than M, the number of MFG nodes is $O(C \cdot M)$, and the number of MFG edges is $O(C \cdot M^2)$. Thus, the space complexity is $O(C \cdot M^2)$ in the worst case.

  Also, the complexity was near liner order in our experimentation.

- Phase 2: Alias Computation using AFG and MFG

  Since we use the instance-based approach, we must also compute the aliases for parent nodes, recursively.

  In the worst case, the time complexity and the space complexity are both $O(E^k)$; however, such a case is quite rare. In our experimentation, k's values were 2 or 3 on average.

References

[1] A. V. Aho, S. Sethi and J. D. Ullman, *Compilers : Principles, Techniques, and Tools*, Addison-Weseley, 1986.

[2] B. Steensgaard, *Points-to analysis in almost linear time*, Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Pages 32–41, 1996, St. Petersburg Beach, Florida, USA.

[3] B. Stroustrup, *The C++ Programming Language(Third edition)*, Addison-Wesley, 1997.

[4] D. C. Atkison and W. G. Griswold, *The Design of Whole-Program Analysis Tools*, Proceedings of the 18th International Conference on Software Engineering, Pages 16–27, 1996, Berlin, Germany.

[5] D. Liang, and M. J. Harrold, *Slicing Objects Using System Dependence Graphs*, Proceedings of the International Conference on Software Maintenance, pp.358–367, 1998, Washington, D.C., USA.

[6] F. Ohata, and K. Inoue, *Alias analysis for object-oriented programs*, Technical Report on IPSJ-SIGSE-126, Vol.2000, No.25, 2000-SE-126, pp.57-64 (in Japanese).

[7] G. Booch, *Object-Oriented Design with Application*, The Benjamin/Cummings Publishing Company, Inc, 1991.

[8] `http://gtkmm.sourceforge.net/`, *Gtk−−*.

[9] `http://www.ANTLR.org/`, *ANTLR Website*.

[10] G. J. Badros, *JavaML: A Markup Language for* JAVA *Source Code* Computer Networks, Vol.33, No.1-6, pp.159–177, 2000.

[11] `http://www.w3c.org/XML/`, *Extensible Markup Language(XML)*.

[12] `http://www.w3.org/TR/1999/WD-xslt-19990421`, *XSL Transformations (XSLT) Specification*.

[13] `http://xmlsoft.org/`, *The XML C library for Gnome*.

[14] J. Gosling, B. Joy, and G. Steele, *The* JAVA^{TM} *Language Specification*, Addison-Weseley, 1996.

[15] J. Zhao, *Slicing Concurrent Java Programs*, Proceedings of the 7th IEEE International Workshop on Program Comprehension, Pages 126–133, 1999, Pittsburgh, Pennsylvania, USA.

[16] M. Enami, R. Ghiya, and L. J. Hendren, *Context-sensitive interprocedural points-to analysis in the presence of function pointers*, Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation, Pages 242–256, 1994, Orlando, Florida, USA.

[17] M. J. Harrold and G. Rothermel, *Separate Computation of Alias Information for Reuse*, IEEE Transactions on Software Engineering, Special section of best papers of the 1996 International Symposium on Software Testing and Analysis, Vol. 22, No. 7, Pages 442–460, 1996.

[18] M. Hind, and A. Pioli, *An empirical comparison of interprocedural pointer alias analysis*, IBM Research Report, #21058, 1997.

[19] M. Hind, and A. Pioli, *Assessing the Effects of Flow-Sensitivity on Pointer Alias Analysis*, IBM Research Report, #21251, 1998.

[20] M. Hind, and A. Pioli, *Which Pointer Analysis Should I Use?*, Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, Pages –, 2000, Portland, Oregon.

[21] M. Hind, *Pointer Analysis: Haven't We Solved This Problem Yet?*, Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Pages 54 – 61, 2001, Snowbird, Utah.

[22] M. Shapiro, and S. Horwitz, *Fast and accurate flow-insensitive point-to analysis*, Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Pages 1–14, 1997, Paris, France.

[23] M. Weiser, *Program slicing*, Proceedings of the 5th International Conference on Software Engineering, Pages 439–449, 1981, San Diego, California, USA.

[24] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo, *Flow insensitive C++ pointers and polymorphism analysis and its application to slicing*, Proceedings of the 19th International Conference on Software Engineering, Pages 433–443, 1997, Boston, Massachusetts, USA.

[25] *The Wisconsin Program-Slicing Tool 1.0, Reference Manual*, Computer Sciences Department, University of Wisconsin-Madison, 1997.

[26] T. Kamiya, F. Ohata, K. Kondo, S. Kusumoto, and K. Inoue, *Maintenance support tools for Java programs: CCFinder and JAAT*, Proceedings of the 5th International Conference on Software Engineering, Pages 837–838, Formal Research Demonstrations, 2001, Toronto, Canada.

[27] R. K. Chatterjee and B. G. Ryder, *Modular Concrete Type-Inference for Statically Typed Object-Oriented Programming Languages*, Department of Computer Science, Rutgers University, no. DCS-TR-349, 1997.

[28] R. P. Wilson, and M. S. Lam, *Efficient context-sensitive pointer analysis for C programs*, Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation, Pages 1–12, 1995, La Jolla, California, USA.

[29] S. Sinha, and M. J. Harrold, *Analysis of Programs With Exception-Handling Constructs*, Proceedings of the International Conference on Software Maintenance, Pages 358–367, 1998, Washington, D.C., USA.

[30] S. Zhang, B. G. Ryder, and W. A. Landi, *Experiments with Combined Analysis for Pointer Aliasing*, Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Pages 11–18, 1998, Montreal, Canada.

```
1:  class Employee {
2:    String name; int salary; Employee supervisor;
3:    Employee(String n, int s) {
4:      name = n;
5:      salary = s;
6:      supervisor = null;
7:    }
8:    void add_salary(int n) {
9:      salary += n;
10:   }
11:   void set_supervisor(Employee e) {
12:     supervisor = e;
13:   }
14:   void print() {
15:     System.out.println(name + " Salary:" + salary);
16:   }
17: }
18: class Manager extends Employee {
19:   Manager(String n, int s) {
20:     super(n, s);
21:   }
22:   void manage(Employee e) {
23:     e.set_supervisor(this);
24:     e.add_salary(200);
25:   }
26: }
27: class Office {
28:   public static void main(String args[]) {
29:     Employee Emp = new Employee("Emp", 750);
30:     Manager Mng = new Manager("Mng", 750);
31:     B.manage(Emp);
32:     Emp.print();
33:     Mng.print();
34:   }
35: }
```

(a) source program

```
% java Office
Emp Salary: 950
Mng Salary: 750
```

(b) program execution (with error)

```
% java Office
Emp Salary: 750
Mng Salary: 950
```

(c) program execution (without error)

Fig. 1. example of alias

```
1: Integer a, b, c;
2: a = new Integer(1);
3: b = new Integer(2);
4: c = b;
5: System.out.println(c);
6: c = a;
7: System.out.println(c);
```

(a) FI alias analysis

(b) alias graph

```
1: Integer a, b, c;
2: a = new Integer(1);
3: b = new Integer(2);
4: c = b;
5: System.out.println(c);
6: c = a;
7: System.out.println(c);
```

(c) FS alias analysis

St.(s)	Reaching alias set($RA(s)$)
1	ϕ
2	ϕ
3	{[(2, a), (2, **new** Integer(1))]}
4	{[(2, a), (2, **new** Integer(1))], [(3, b), (3, **new** Integer(2))]}
5	{[(2, a), (2, **new** Integer(1))], [(4, c), (3, b), (4, b), (3, **new** Integer(2))]}
6	{[(2, a), (2, **new** Integer(1)), [(4, c), (5, c), (3, b), (4, b), (3, **new** Integer(2))]}
7	{[(6, c)], [(2, a)], [(6, a)], [(2, **new** Integer(1))], [(3, b), (4, b), (3, **new** Integer(2))]}

(d) reaching alias set (RAset)

Fig. 2. FI alias analysis and FS alias analysis

```
1:  x = new A;
2:  y = new A;
3:  x.s = ...;
4:  y.s = ...;
```

```
5:  class A {
6:      Integer s;
7:      ...(...) {
8:          ... = s;
```

Fig. 3. aliases without instance separation

```
1:  public class Calc {
2:    Integer i;
3:    public Calc() {
4:      i = new Integer(0);
5:    }
6:    public void inc() {
7:      i = new Integer(i.intValue() + 1);
8:    }
9:    public void add(int c) {
10:     i = new Integer(i.intValue() + c);
11:   }
12:   public Integer result() {
13:     return(i);
14:   }
15: }
```

(a) flow insensitive object context (FIOC)

```
1:  public class Calc {
2:    Integer i;
3:    public Calc() {
4:      i = new Integer(0);
5:    }
6:    public void inc() {
7:      i = new Integer(i.intValue() + 1);
8:    }
9:    public void add(int c) {
10:     i = new Integer(i.intValue() + c);
11:   }
12:   public Integer result() {
13:     return(i);
14:   }
15: }
```

(b) flow sensitive object context (FSOC)

Fig. 4. example of object context

$\varphi(x)$: mapping from expression x to its corresponding AFG node

$\varphi^{-1}(X)$: mapping from AFG node X to its corresponding expression

$\mathcal{A}(X)$: aliases for AFG node X

$N_P(X)$: a parent AFG node for AFG node X

$\psi(m)$: mapping from method m to its corresponding AFG

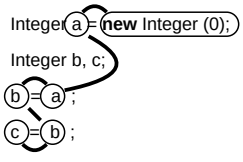
$\psi^{-1}(M)$: mapping from AFG M to its corresponding method

$e =_{id} e'$: **true** when e and e' are the same identifiers, otherwise **false**

Fig. 5. definition of symbols

```
1:  Integer a = new Integer(0);
2:  Integer b, c;
3:  b = a;
4:  c = b;
```

(a) source program



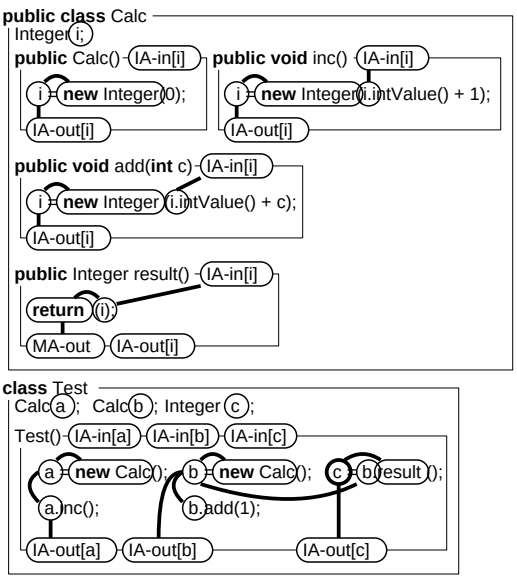
(b) AFG

Fig. 6. sample program and its AFG (1)

```
1:  public class Calc {
2:    Integer i;
3:    public Calc() {
4:      i = new Integer(0);
5:    }
6:    public void inc() {
7:      i = new Integer
            (i.intValue() + 1);
8:    }
9:    public void add(int c) {
10:     i = new Integer
            (i.intValue() + c);
11:   }
12:   public Integer result() {
13:     return(i);
14:   }
15: }
```

```
16: class Test {
17:   Calc a, b;
18:   Integer c;
19:   Test() {
20:     a = new Calc();
21:     b = new Calc();
22:     a.inc();
23:     b.add(1);
24:     c = b.result();
25:   }
26: }
```

(a) source program

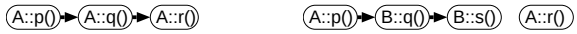


(b) AFG

Fig. 7. sample program and its AFG (2)

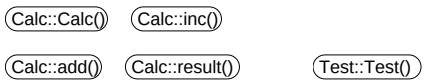
```
1:  class A {
2:      void p() { q(); }
3:      void q() { r(); }
4:      void r() { }
5:  }
```

```
6:  class B extends A {
7:      void q() { s(); }
8:      void s() { }
9:  }
```



(a) class `A` (b) class `B`

Fig. 8. sample program and its MFG



(a) class `Calc` (b) class `Test`

Fig. 9. MFGs for Fig. 7(a)

```
1:  public class Calc {
2:      Integer i;
3:      public Calc() {
4:          i = new Integer(0);
5:      }
6:      public void inc() {
7:          i = new Integer
                (i.intValue() + 1);
8:      }
9:      public void add(int c) {
10:         i = new Integer
                (i.intValue() + c);
11:     }
12:     public Integer result() {
13:         return(i);
14:     }
15: }
```

```
16: class Test {
17:     Calc a, b;
18:     Integer c;
19:     Test() {
20:         a = new Calc();
21:         b = new Calc();
22:         a.inc();
23:         b.add(1);
24:         c = b.result();
25:     }
26: }
```

Fig. 10. alias set for <24, `c`> (masked expressions are aliases)

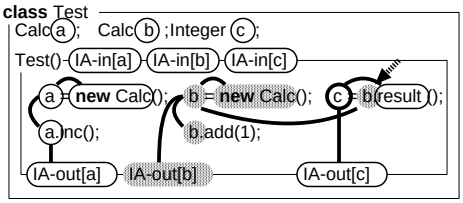


Fig. 11. alias set for <24, `b`> in Fig.10 (masked expressions are aliases)

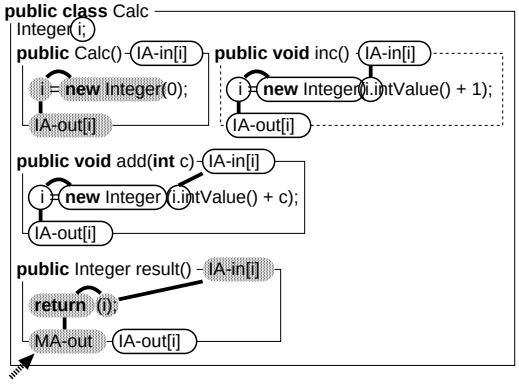


Fig. 12. alias set for <24, `result()`> in Fig.10 (masked expressions are aliases)

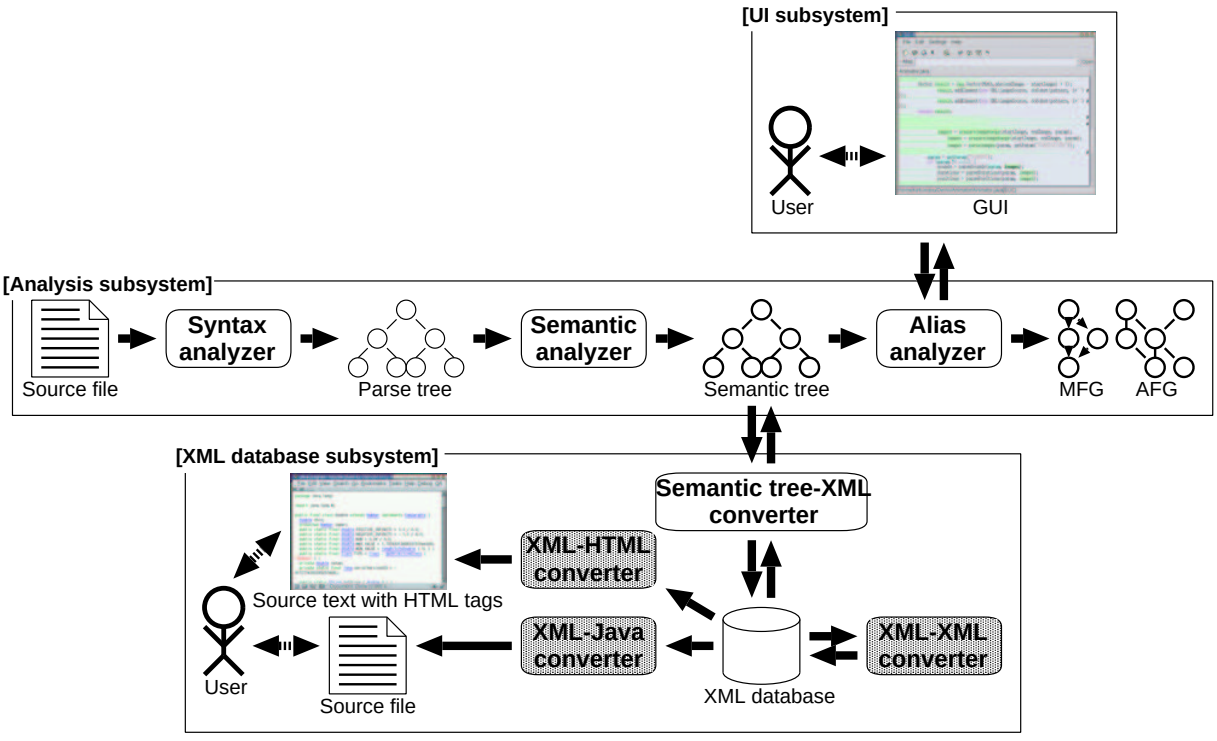
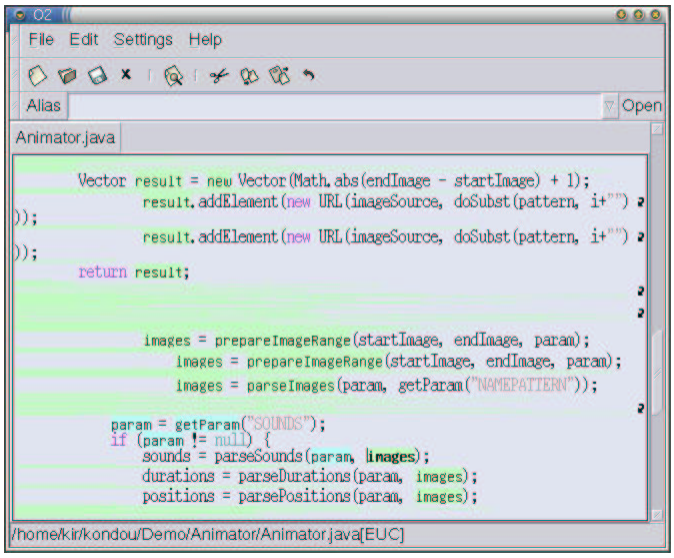
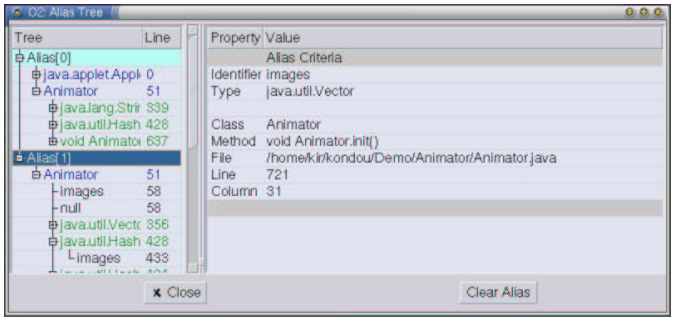


Fig. 13. JAAT (structure)



(a) text window



(b) alias tree window

Fig. 14. JAAT (UI)

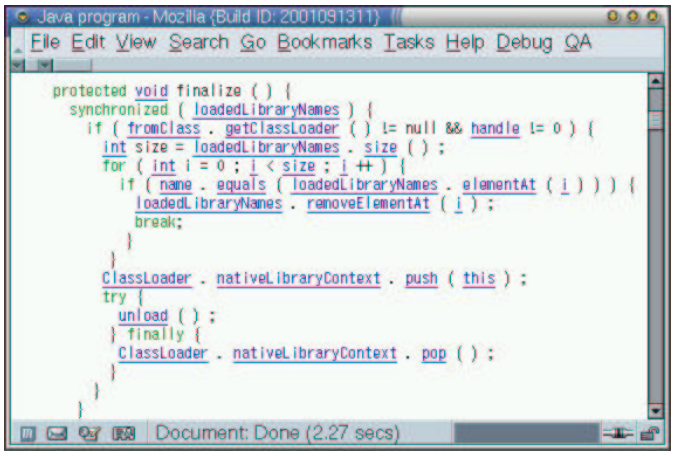
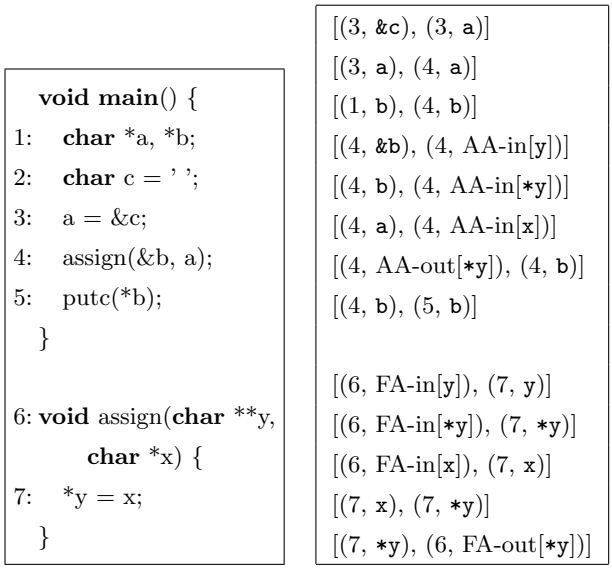


Fig. 15. HTML representation of JAVA source file



(a) sample C program

(b) extracted direct alias relations

Fig. 16. alias analysis for programs with pointer variables

Step 1: Specify an alias criterion e ($E := \varphi(e)$).

Step 2: Start AFG traversal from E.

When we traverse AFG, we perform following processes at each reachable node C:

[Cond.1] C is an AFG normal node that has a parent node:

1. $c := \varphi^{-1}(C)$, $P := N_P(C)$
2. Compute P's aliases $\mathcal{A}(P)$.
3. Compute $\mathcal{A}(P)$'s type using class instance creation expressions included in $\mathcal{A}(P)$
4. Compute OC($\mathcal{A}(P)$).
5. Traverse AFG from IA-in[c] and IA-out[c] nodes in $\{G_M \mid \psi^{-1}(G_M) \in \mathrm{OC}(\mathcal{A}(P))\}$, and Traverse AFG from $\{N \mid N_P(N) \in \mathcal{A}(P)$ **and** $\varphi^{-1}(N) =_{id} c\}$.

[Cond.2] C is an MI node that has a parent node:

1. $c := \varphi^{-1}(C)$, $P := N_P(C)$
2. Compute P's aliases $\mathcal{A}(P)$.
3. Compute $\mathcal{A}(P)$'s type using class instance creation expressions included in $\mathcal{A}(P)$.
4. Compute OC($\mathcal{A}(P)$).
5. Traverse AFG from MA-out nodes in $\{M \mid \psi^{-1}(M) \in \mathrm{OC}(\mathcal{A}(P))\}$.

[Cond.3] C is an IA-in or IA-out node:

1. $c := \varphi^{-1}(C)$
2. Traverse AFG from IA-out[c] or IA-in[c] in $\{M \mid \psi^{-1}(M) \in \mathrm{OC}(\mathbf{this})\}$.

[Cond.4] C is an AA-in or AA-out node:

1. Compute method m that MI node $N_P(C)$ calls.
2. Traverse AFG from the corresponding FA-in or FA-out node in $\psi(m)$.

[Cond.5] C is an FA-in or FA-out node:

1. $M_B := \{M \mid C \in M\}$, $m_B := \psi^{-1}(M_B)$
2. $m_A := \{m \mid m \in \mathrm{OC}(\mathbf{this})$ **and** m calls $m_B\}$ (using MFG)
3. $M_A := \psi(m_A)$
4. Traverse AFG from the corresponding AA-in or AA-out node in M_A.

[Cond.6] C is an MA-out node:

1. $M_B := \{M \mid C \in M\}$, $m_B := \psi^{-1}(M_B)$
2. $m_A := \{m \mid m \in \mathrm{OC}(\mathbf{this})$ **and** m calls $m_B\}$ (using MFG)
3. $M_A := \psi(m_A)$
4. In M_A, traverse AFG from MI nodes that call m_B

[Cond.7] C is a MI node:

1. $m_A := \{m \mid m \in \mathrm{OC}(\mathbf{this})$ **and** C calls $m\}$
2. $M_A := \psi(m_A)$
3. Traverse AFG from MA-out nodes in M_A.

..... [Cond.8] and [Cond.9] are for programs that have pointer variables.

[Cond.8] C is an AFG normal node with '$*$' operator, such as $*x$:

1. $X := \varphi(x)$
2. Compute X's aliases $\mathcal{A}(X)$.
3. Traverse AFG from $\{\varphi(y) \mid \varphi(\&y) \in \mathcal{A}(X)\}$ and $\{\varphi(*y) \mid \varphi(y) \in \mathcal{A}(X)\}$.

[Cond.9] C is an AFG normal node with '&' operator, such as $\&x$:

1. $X := \varphi(x)$
2. Compute X's aliases $\mathcal{A}(X)$.
3. Traverse AFG from $\{\varphi(y) \mid \varphi(*y) \in \mathcal{A}(X)\}$ and $\{\varphi(\&y) \mid \varphi(y) \in \mathcal{A}(X)\}$.

.....

Step 3: Expressions corresponding to reachable nodes are e's aliases.

Fig. 17. alias computation algorithm