JAAT: Java Alias Analysis Tool
for Program Maintenance Activities

Fumiaki Ohata
Toshiba
Software Engineering Center
Kawasaki, Japan
fumiaki.oohata@toshiba.co.jp

Katsuro Inoue
Osaka University
Department of Computer Science
Osaka, Japan
inoue@ist.osaka-u.ac.jp

Abstract

Alias analysis is a method for extracting sets of expressions which may possibly refer to the same memory locations during program execution. Although many researchers have already proposed analysis methods for the purpose of program optimization, difficulties still remain in applying such methods to practical software engineering tools in the sense of precision, extensibility and scalability.

Focusing mainly on a practical use for program maintenance activities such as program debugging and understanding, we propose an alias analysis method for object-oriented programs and discuss our implementation. Using this method, we have developed a tool named JAAT.

Our proposed method employs a two-phase, on-demand, and instance-based algorithm, in which intra-class analysis is done in Phase 1 for whole programs and libraries, and inter-class analysis is done in Phase 2 only for a user-demanded target. JAAT can analyze large programs or libraries such as JDK class library. Also, JAAT includes various features for program maintenance activities, such as GUI for displaying aliases, and an XML database for storing analysis information.

1 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. Also, we call the set of expressions in which each element pair satisfies an alias relation, an *alias set*. *Alias analysis* is a method for extracting alias sets by static analysis. Alias analysis can be used for various purposes such as compiler optimization and program slicing.

Alias analysis was first proposed for traditional procedural languages such as C as part of the static analysis of pointer variables [7, 11, 19]. Concepts such as class, inheritance, dynamic binding, and polymorphism have been introduced into object-oriented (OO) languages such as C++ and JAVA, and various alias analysis methods for OO programs have been devised [5, 22]. These researches focus mainly on analysis algorithms as compiler optimization, but practicability and scalability of those algorithms as software engineering tools have not been explored.

We are interested in developing a practical software engineering tool for the alias analysis targeting JAVA; however, simply using already proposed approaches remains difficulties in scalability and usage as addressed by Hind et. al. [10]. To resolve these difficulties, we have newly devised an on-demand, incremental analysis approach for JAVA programs, which can be used effectively in an interactive environment.

In this paper, "alias analysis" means to extract a single set of expressions which are in alias relation to the user-specified expression, although a traditional meaning would be to extract all alias sets in a source program.

The alias analysis method proposed here is characterized by a two-phase and on-demand algorithm, flow-sensitive instance-based algorithm, and extensible algorithm.

We have implemented the proposed algorithm in a tool named JAAT. The analysis time of 58,300 lines of JAVA programs with 364,721 lines of the JDK library was 30 sec. in Phase 1, and less than 1 milli-sec. 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 focused in the sense that for our sample programs, about 5 - 120 aliases were found due to the instance-based approach, which are 30 – 97% smaller than the class-based approach. JAAT does not provide whole alias relations as the compiler optimization algorithm requires, but it produces the focused or scoped results useful for the program maintainers.

An additional feature of JAAT is that 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 a useful

GUI for the program maintainers.

In Section 2, we give a brief overview of alias analysis for OO programs. In Section 3 and Section 4, we propose an alias analysis method for OO programs. In Section 5, we introduce an implementation of the proposed method and evaluate its effectiveness using several sample programs. In Section 6, we discuss the evaluation results with respect to related works. In Section 7, we conclude our discussion.

2 Preliminary

2.1 Example of Aliases

Alias analysis is useful for program debugging and program understanding. To show this, we present an example.

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 that of the employee. However, the program execution output is incorrect since a salary addition was made to `Emp`, not to `Mng`. 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. In the figure, shadowed expressions represent the resulting aliases for $<s_{32}$, `Emp`$>$. `Emp` at line 32 is the alias criterion and is also an alias itself. Therefore, it is boxed and shadowed. We can easily see around those shadowed 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).

2.2 Alias Analysis

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

In FI analysis, we do not take into account the execution order of each statement in the source program [14, 19, 21, 25]. To compute FI aliases, an *alias graph*, as shown in Fig.2(b), is used. 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 $<s_7$, `c`$>$ as the alias criterion, we get aliases {`a`, `b`, `new Integer(1)`, `new Integer(2)`}, which are all reachable nodes from the criterion node in the alias graph.

In FS analysis, we consider the execution order of statements [7, 12, 23]. To compute FS aliases, Landi et al. have introduced *a reaching alias set (RAset)* [12]. A RAset for statement s, denoted by $RA(s)$, is a collection of *alias sets*, which exists just before the execution of s. Each alias set is composed of sets of tuples (t, f) (t is a statement in the source program and f is an expression at t), meaning that each f at t in the set possibly refers to the same memory location. Fig.3(b) shows RAsets for each statement in Fig.3(a). In order to compute the aliases for $<s, e>$, we search $RA(s)$ for an alias set that contains e. At $RA(s_7)$ in Fig.3(b), since an alias set {(s_6, `c`), (s_2, `a`), (s_6, `a`), (s_2, `new Integer(1)`)} contains variable `c`, we get the result as shown in Fig.3(a).

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 and Fig.3, we can see the difference in the accuracy between the two methods. In this paper, we focus on FS analysis for more accurate analysis results.

2.3 Alias Analysis for Object-Oriented Programs

Alias analysis methods for OO programs have been proposed as an extension of analysis methods for procedural programs [5, 22]; however, we have to further consider the nature of OO programs.

In OO programs such as JAVA, each object has its own state and behavior even if they are instantiated from the same class. In sample JAVA program shown at Fig.4, we prefer to have three independent alias sets:

- {(s_1, `new Integer(1)`), (s_5, `x.get()`), (s_9, `id`), (s_{10}, `ref`), (s_{11}, `ref`), (s_{11}, `id`), (s_{15}, `id`)}
- {(s_2, `new Integer(2)`), (s_6, `y.get()`), (s_9, `id`), (s_{10}, `ref`), (s_{11}, `ref`), (s_{11}, `id`), (s_{15}, `id`)}
- {(s_4, `new Integer(3)`), (s_7, `z.get()`), (s_9, `id`), (s_{12}, `ref`), (s_{13}, `ref`), (s_{13}, `id`), (s_{15}, `id`)}

However, if we apply a simple analysis approach such that all objects instantiated from the same class share the alias information of their attributes and their calling-contexts, we get only one alias set which is the union of these 3 alias sets: {(s_1, `new Integer(1)`), (s_5, `x.get()`), (s_2, `new Integer(2)`), (s_6, `y.get()`), (s_3, `null`), (s_4, `new Integer(3)`), (s_7, `z.get()`), (s_9, `id`), (s_{10}, `ref`), (s_{11}, `ref`), (s_{11}, `id`), (s_{12}, `ref`), (s_{13}, `ref`), (s_{13}, `id`), (s_{15}, `id`)}

In this case, many expressions are unwillingly in the same alias set. In order to increase the analysis precision, we will separately hold the alias information for each attribute of each object instance, although this approach generally requires more analysis cost. However, we will devise an efficient approach to resolve this.

```
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:         Mng.manage(Emp);
32:         Emp.print();
33:         Mng.print();
34:     }
35: }
```

(a) Java source program

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

(b) Program execution result with error

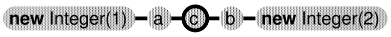
```
% java Office
Emp Salary: 750
Mng Salary: 950
```

(c) Program execution result without error

Figure 1. Simple debugging process of Java program with aliases

```
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 aliases



(b) Alias graph

Figure 2. Example of FI alias analyses

```
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) FS aliases

Statement(s)	Reaching alias set($RA(s)$)
s_1	ϕ
s_2	ϕ
s_3	$\{\{(s_2, \text{a}), (s_2, \texttt{new Integer(1)})\}\}$
s_4	$\{\{(s_2, \text{a}), (s_2, \texttt{new Integer(1)})\}, \{(s_3, \text{b}), (s_3, \texttt{new Integer(2)})\}\}$
s_5	$\{\{(s_2, \text{a}), (s_2, \texttt{new Integer(1)})\},$ $\{(s_4, \text{c}), (s_3, \text{b}), (s_4, \text{b}), (s_3, \texttt{new Integer(2)})\}\}$
s_6	$\{\{(s_2, \text{a}), (s_2, \texttt{new Integer(1)})\},$ $\{(s_4, \text{c}), (s_5, \text{c}), (s_3, \text{b}), (s_4, \text{b}), (s_3, \texttt{new Integer(2)})\}\}$
s_7	$\{\{(s_6, \text{c}), (s_2, \text{a}), (s_6, \text{a}), (s_2, \texttt{new Integer(1)})\},$ $\{(s_3, \text{b}), (s_4, \text{b}), (s_3, \texttt{new Integer(2)})\}\}$

(b) Reaching alias set (RAset)

Figure 3. Example of FS alias analyses

3 Analysis Overview

3.1 Approach

Here, we divide alias relations into two categories, *intra-class alias relations* and *inter-class alias relations*. Intra-class alias relations do not depend on their usage contexts. Inter-class alias relations are obtained by analyzing expressions with method invocations and reference variables over classes. For example, in Fig.4, $\{(s_9$, `id`), (s_{10}, `ref`), (s_{11}, `ref`), (s_{11}, `id`)$\}$, $\{(s_9$, `id`), (s_{12}, `ref`), (s_{13}, `ref`), (s_{13}, `id`)$\}$ and $\{(s_9$, `id`), (s_{15}, `id`)$\}$ are intra-class alias relations, and $\{(s_1$, `new Integer(1)`), (s_{10}, `ref`)$\}$, $\{(s_5$, `x.get()`), (s_{15}, `id`)$\}$, $\{(s_2$, `new Integer(2)`), (s_{10}, `ref`)$\}$, $\{(s_6$, `y.get()`), (s_{15}, `id`)$\}$, $\{(s_3$, `null`), (s_{10}, `ref`)$\}$ and $\{(s_4$, `new Integer(3)`), (s_{12}, `ref`)$\}$ are inter-class alias relations.

```
1:  Ident x = new Ident(new Integer(1));
2:  Ident y = new Ident(new Integer(2));
3:  Ident z = new Ident(null);
4:  z.set(new Integer(3));
5:  ... = x.get();
6:  ... = y.get();
7:  ... = z.get();
```

```
8:  class Ident {
9:      private Integer id;
10:     public Ident(Integer ref)
11:         { id = ref; }
12:     public void set(Integer ref)
13:         { id = ref; }
14:     public Integer get()
15:         { return id; }
16: }
```

Figure 4. Example of aliases across instances

Here, we will adopt the following analysis policies:

Policy 1: Compute intra-class alias relations in advance.

Policy 2: Compute inter-class alias relations on-demand.

Utilizing these two policies, the modularity and independence of the analysis will be established. This is particularly important in OO programming, since we usually use large class libraries in addition to user-developed classes. The analysis cost of these class libraries is generally high. Thus, modularizing the analyses of class libraries and user-developed classes is essential.

Note that mixing inter-class alias relations reduces precision. In the case of Fig.4, mixing all alias expressions will unwillingly generate a large and useless alias set. In order to resolve this problem, we use the following policy:

Policy 3: Compute inter-class alias relations based on the individual invocations and references of instance methods and attributes (we call this *instance-based analysis*).

The instance-based analysis of OO programs can be considered to be an extension of the context-sensitive analysis of procedural programs.

3.2 Object Context

To further improve the analysis precision of the instance-based analysis, we introduce *object context*.

Consider an alias set $\mathcal{A}$, which contains expressions referring to the object instances. Some instance methods of these objects are invoked directly or indirectly from the expressions in $\mathcal{A}$, and some are never invoked from the context of $\mathcal{A}$. We delete unnecessary alias expressions which appear in the body of never-called instance methods.

The *object context* for alias set $\mathcal{A}$, denoted by $OC(\mathcal{A})$, is a set of instance methods of instance objects pointed to by expressions in $\mathcal{A}$, and that may be invoked from some expression associated with expressions in $\mathcal{A}$. The object context is formally obtained by the algorithm shown in [15].

Fig.5 shows an example of the object context. Assume that $\mathcal{A}$ is the aliases for `a` at line 22 such that $\{(s_{20}$, `a`), (s_{20}, `new Calc()`), (s_{22}, `a`)$\}$. Now we know that $\mathcal{A}$ is `a` Calc type, and possible instance method invocations are `new Calc()` at line 20 and `a.inc()` at line 22. Thus, `Calc::Calc()` and `Calc::inc()` are included in $OC(\mathcal{A})$. Since these two methods have no further invocations of other instance methods in a `Calc` class, we finally know that $OC(\mathcal{A})$ is $\{$`Calc::Calc()`,`Calc::inc()`$\}$.

Also, if we assume that $\mathcal{A}$ is an alias set for `b` at line 21, then $OC(\mathcal{A})$ is $\{$`Calc::Calc()`, `Calc::add()`, `Calc::result()`$\}$.

When we compute the aliases for $e.i$ such that e is in $\mathcal{A}$, we can limit the candidate methods to be considered further by using $OC(\mathcal{A})$. In other words, we can exclude the instance methods that can not be invoked in the objects referred to by expressions in $\mathcal{A}$.

4 Analysis Process

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

Phase 1: Intra-class analysis for all source programs are composed of the following two sub-phases:
(a) Construction of AFG (Alias Flow Graph) by analyzing inside each method.
(b) Construction of MFG (Method Flow Graph) by analyzing methods in each class.

Phase 2: Inter-class analysis for a specified alias criterion, i.e., computation of the aliases by traversing AFG and MFG along with the object context.

```
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: }
```

Figure 5. Example program for object context analysis

4.1 Phase 1: Construction of AFG and MFG

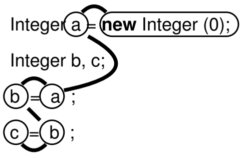
4.1.1 Phase 1(a): Construction of AFG

An AFG (Alias Flow Graph) 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
- a parameter to/from a method or an instance.

An edge in AFG denotes an alias relation immediately determined inside each method. Alias relations created by assignment statements, variable definitions and their uses (def-use relations), and assignments of parameters to/from special nodes are called *direct alias relations*. Direct alias relations are easily obtained by RAset-based FS may-alias analysis inside methods [12]. Also, a path formed with more than one edge is called an *indirect alias relation*.

```
Integer a = new Integer(0);
Integer b, c;
b = a;
c = b;
```

(a) Source program



(b) AFG

Figure 6. Example of AFG

Fig.6 shows a small JAVA program and its AFG. Nodes in AFG are shown as circles with expressions inside, and the 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 (s_1, `new Integer(0)`) is assigned to (s_1, `a`) in the source program, we can see that the node for (s_1, `new Integer(0)`) is connected to the node for (s_1, `a`) with an edge. This edge represents a direct alias relation.

Fig.7 is an AFG for the program shown in Fig.5. Variable `i` appearing at each right-hand side expression (line 7 and 10) is a reference-type instance variable. IA-in, IA-out, MA-in, and MA-out are the special nodes for parameter passing. `b.result()` at line 24 is represented in AFG with an associated node `b` and node `result()`.

4.1.2 Phase 1(b): Construction of MFG

An MFG (Method Flow Graph) is a directed graph, which represents the caller-callee relations of methods in a single class[1]. An *MFG node* denotes the definition of each method, and when method A possibly calls method B, an *MFG edge* is drawn from the node for A to the node for B.

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 `B`'s object. In this case, method call to `q()` appearing in `A::p()` causes activation

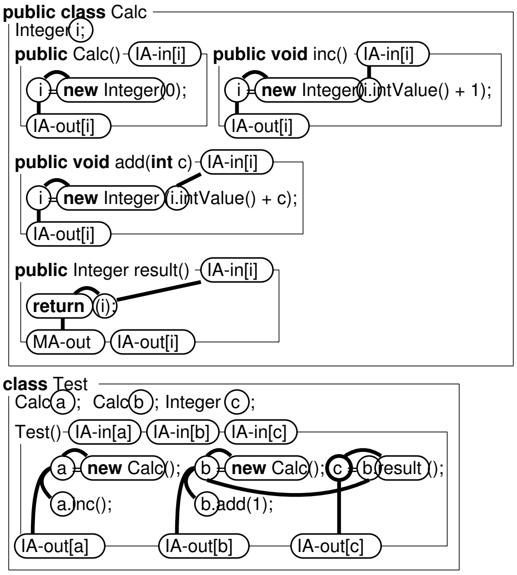
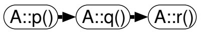

public class Calc
Integer i;
public Calc() IA-in[i]
i = new Integer(0);
IA-out[i]
public void inc() IA-in[i]
i = new Integer(i.intValue() + 1);
IA-out[i]
public void add(int c) IA-in[i]
i = new Integer(i.intValue() + c);
IA-out[i]
public Integer result() IA-in[i]
return(i);
MA-out IA-out[i]
class Test
Calc a; Calc b; Integer c;
Test() IA-in[a] IA-in[b] IA-in[c]
a = new Calc(); b = new Calc(); c = b.result();
a.inc();
b.add(1);
IA-out[a] IA-out[b] IA-out[c]

Figure 7. AFG for Fig.5

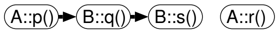
[1] MFG corresponds to a caller-callee graph (call graph) in procedural languages.

```
class A {
    void p() { q(); }
    void q() { r(); }
    void r() { }
}
```



(a) class A

```
class B extends A {
    void q() { s(); }
    void s() { }
}
```



(b) Class B

Figure 8. Example of MFG

of `B::q()`, not `A::q()`. Thus, the resulting MFG for class `B` is as shown in Fig.8(b).

4.2 Phase 2: Alias Computation Using AFG and MFG

Using AFG and MFG, we compute aliases $\mathcal{A}(e)$ for an alias criterion e, which is a reference-type expression. e itself is also an element of $\mathcal{A}(e)$. The nodes in AFG are visited beyond the class boundary using MFG information.

We will show an intuitive description of the traversal as follows. The formal definition, termination, and complexity of the traversal algorithm are presented in our technical report [15].

1. When we compute the aliases for e with a parent p (i.e., the expression is $p.e$), 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 OC($\mathcal{A}(p)$).

   After computing these, we compute e's aliases.

2. When we reach an terminal node in AFG during the AFG traversal, we determine the callee or caller method using MFG and the object context, and resume the traversal from the corresponding terminal nodes.

The result of the traversal is a set of reachable nodes from the alias criterion. The masked expressions in Fig.9 are the resulting aliases for (s_{24},c). Since OC($\mathcal{A}(b)$) does not contain `Calc::inc()`, expressions in `Calc::inc()` are excluded from the candidates for (s_{24}, `c`)'s aliases.

Together with Phase 1, the overall analysis algorithm establishes a FS(Flow Sensitive), may-alias, instance-based, and FIOC(Flow Insensitive Object Context) approach.

```
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: }
```

Figure 9. Resulting Aliases

5 A JAVA Alias Analysis Tool (JAAT)

We have implemented the proposed method in the tool *Java Alias Analysis Tool (JAAT)*. Using JAAT, we have analyzed several programs and obtained various data.

5.1 Overview of JAAT

JAAT consists of three subsystems, the *analysis subsystem*, the *XML database subsystem*, and the *user interface (UI) subsystem*. Fig.10 shows the structure of JAAT.

Analysis subsystem: The analysis subsystem consists of three components, the syntax analyzer, the semantic analyzer, and the alias analyzer. The *syntax analyzer* analyzes JAVA source files and generates syntactic trees. 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* generates MFGs and AFGs at Phase 1, and computes the aliases for the alias criterion specified by the user's request at Phase 2. The alias analyzer returns the resulting aliases.

XML database subsystem: Since the translation from a source program to the corresponding semantic tree is a fairly time-consuming process, we do not want to discard analysis results from the analysis sessions. Thus, we build a database for semantic trees. This feature

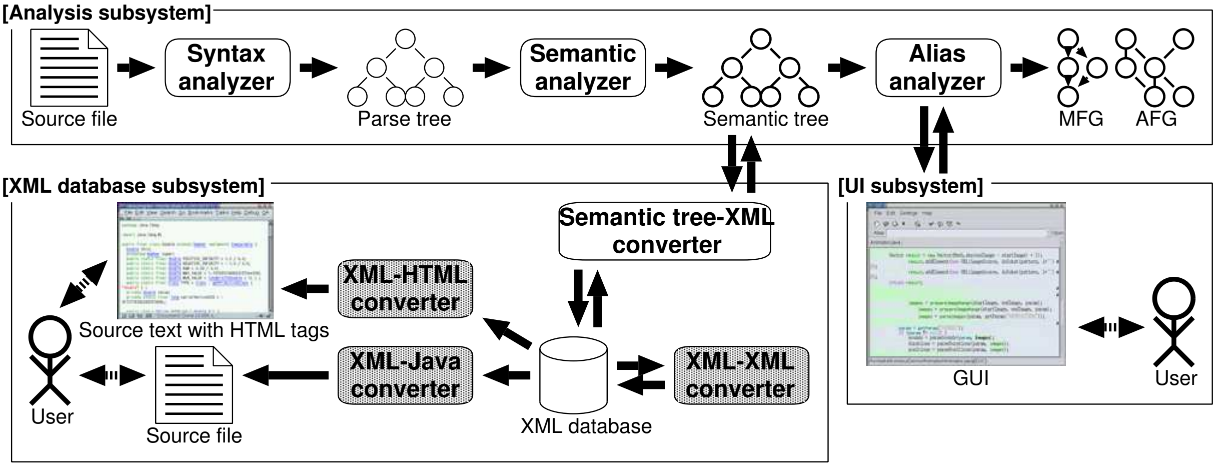


Figure 10. Architecture of JAAT

improves the reusability of the analysis results along with the AFG and MFG approaches. We use an XML database that holds semantic tree information. The *XML converters* converts semantic trees to XML documents and vice versa.

UI subsystem: The UI subsystem has two main functions, editing programs and visualizing the resulting aliases. Examples will be shown in Section 5.2.5.

5.2 Evaluation

In order to explore the applicability of JAAT, we have applied it to various sample programs. Table 1 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-class alias analysis. For example, `TextEditor` is composed of one file with 1,125 lines. `TextEditor` directly and indirectly uses the classes in JDK, which are in 878 files with a total of 351,890 lines (99.7% of the overall total lines). We have used a Pentium IV machine (2GHz) with 2GB memory running FreeBSD 4.6.

5.2.1 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, there are several FS analysis algorithms already proposed[7, 12, 23]. Those algorithms mainly focus on language-specific problems such as pointers to the stack, and they do not concern the separation of analysis results for each module. Therefore, if we would employ those algorithms, the overall program have to be re-analyzed. It should be noted that user-written programs are often modified but their related classes in JDK are seldom modified.

Table 2(a) shows AFG construction time for sample programs and their related classes in JDK. The sum of these two is the total time for Phase 1(a). The analysis time for the related classes in JDK is much longer than for those of the sample programs. For example, `TextEditor` itself requires only 10 milli-sec., and its related classes require 14,224 milli-sec. When we modify `TextEditor`, we do not need to re-analyze its related classes, but only the `TextEditor`.

5.2.2 Computation Time of Phase 1(b)

Table 2(b) shows MFG construction time for sample programs and their related classes. Since MFG construction time does not depend on the program's 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 related classes in JDK. For example, `DynamicJava` itself requires 1,892 milli-sec., but its related classes require only 843 milli-sec. However, the overall MFG construction time is much smaller than the AFG construction time.

5.2.3 Computation Time of Phase 2

Table 2(c) shows the average AFG traversal time for various alias criteria. According to Table 2(c), it is clear that Phase 2 takes much less computation time than Phase 1. For `TextEditor`, only 0.01 milli-sec. is required, which is much smaller than the overall Phase 1 of 14,915 milli-sec.

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

5.2.4 Average Number of Detected Aliases

The proposed method uses the *instance-based* approach that can distinguish inter-class alias relations on objects instantiated from the same class. On the other hand, if we use the *class-based* approach that shares inter-class alias relations with other objects instantiated from the same class, analysis precision will decrease. On each test suite, we select a

Table 1. Characteristics of analyzed sample programs

Programs	Sample Program		Related Classes in JDK	
	Number of Files	Number of Lines	Number of Files	Number of Lines
TextEditor	1(0.1%)	1125(0.3%)	878(99.9%)	351,890(99.7%)
JLex (Parser Generator)	1(0.4%)	7,835(6.9%)	275(99.6%)	105,234(93.1%)
java_cup (Parser Generator)	35(11.3%)	10,610(9.1%)	274(88.7%)	105,598(90.9%)
JFlex (Parser Generator)	40(4.3%)	13,029(3.6%)	882(95.7%)	353,067(96.4%)
WeirdX (X server)	47(5.0%)	19,701(5.2%)	892(95.0%)	356,217(94.8%)
ANTLR (Parser Generator)	129(31.6%)	25,283(19.3%)	279(68.4%)	105,483(80.7%)
Ant (Build Tool)	98(24.0%)	26,428(18.8%)	310(76.0%)	114,262(81.2%)
DynamicJava (JAVA Interpreter)	242(21.1%)	58,300(13.8%)	903(78.9%)	364,721(86.2%)

Table 2. Computation time [ms]

(a) Phase 1(a)

Programs	Sample Program	Related Classes in JDK
TextEditor	10	14,224
JLex	892	3,863
java_cup	844	3,813
JFlex	16,140	14,339
WeirdX	2,835	14,666
ANTLR	6,154	7,856
Ant	1,845	4,005
DynamicJava	12,255	15,646

(b) Phase 1(b)

Programs	Sample Program	Related Classes in JDK
TextEditor	13	768
JLex	10	100
java_cup	10	99
JFlex	10	759
WeirdX	10	823
ANTLR	304	99
Ant	22	104
DynamicJava	1,892	843

(c) Phase 2

Programs	Average
TextEditor	0.01
JLex	0.76
java_cup	0.37
JFlex	0.41
WeirdX	0.62
ANTLR	0.69
Ant	0.78
DynamicJava	0.07

class which plays an dominant role in the test suite, and we compute the aliases for each AFG normal node in that class.

Table 3 shows the comparison results between those two approaches with regard to the average number of detected aliases for various alias criteria in the main classes. For example, the instance-based approach generates more accurate results than the class-based approach (9.16 nodes v.s. 17.19 nodes) in DynamicJava. The average size of aliases is about 30 – 97% of the class-based approach; therefore, we think that our approach is of practical value.

In some cases, the average number of detected aliases is not small. This is because we repeatedly count each expression, even if they have the same signature. If a specific variable is repeatedly used in a class, the number of the aliases becomes large. For example, in the case of JLex, the average number of the aliases is 69.17, for instance variable JLex.CLexGen.m_outstream, where m_outstream is referred to more than 400 times in the JLex.CLexGen class. However, the average number of unique variables in those aliases in JLex.CLexGen is only 1.50. This suggests that the users can easily focus their attention on only those few variables.

5.2.5 Case study

We focus on program maintenance activities as an application of JAAT. To examine JAAT's effectiveness, we have applied JAAT to the following program debugging case.

> SpreadSheet.java (1000 lines) is a small spreadsheet JAVA applet contained in JDK. We assume that an error occurred on the execution of SpreadSheet.java as shown in Fig.11(a). Since cell C1 was defined A1*B1, C1 should be 5000; however, C1's value was incorrectly 10.

First we computed aliases for a String-type actual parameter formula in a method parseFormula(), which is a parser for the input expressions.

Using *alias tree window*[2], we knew the resulting aliases are in parseFormula() and parseValue() only as shown in Fig.12(a). We examined each expression on the alias tree using the alias tree window and the *text window* [3] as shown in Fig.12(b). After checking all aliases in parseFormula(), we recognized that the return expression in parseValue() is variable formula.

By examining several statements near the last return expression, we noticed that the return variable should be the variable restFormula, instead of formula. After fixing it, a new SpreadSheet.java was executed correctly (Fig.11(b)).

[2] The alias tree window shows an aliases tree, in which each node denotes class, method, or an expression which contains aliases.

[3] The text window shows the resulting aliases with colored backgrounds. Statements without any aliases can be compressed on the screen with smaller fonts according to the user's request.

Table 3. Average number of detected alias expressions (UNIQUE VARIABLES)

Programs (target class)	Instance-based	Class-based
TextEditor (`TextEditor`)	5.09(1)	5.09(1)
JLex (`JLex.CLexGen`)	69.17(2.02)	231.5(5.43)
java_cup (`java_cup.parser`)	101.6(1.48)	104.6(2.17)
JFlex (`JFlex.LexParse`)	124.2(1.50)	127.5(2.36)
WeirdX (`com.jcraft.weirdx.Client`)	68.33(2.97)	84.35(3.41)
ANTLR (`antlr.Tool`)	6.62(1.49)	11.37(2.30)
Ant (`org.apache.tools.ant.Main`)	20.94(3.25)	36.62(6.26)
DynamicJava (`koala.dynamicjava.interpreter.TypeChecker`)	9.16(1.89)	17.19(2.40)

In this example, there are only 20 aliases, so that the user's attention can be focused on only 20 lines out of the original source program of about 1000 lines. The alias tree windows can help the user to grasp the overall resulting aliases, and the text windows can help the user to get detailed information about each alias.

6 Discussion

Our proposed method for alias computation, which consists of two analysis phases, has produced effective results. In this section, we compare our method and related works, and also show an extension of our method to programs with pointer variables in ordinary languages.

6.1 Alias Analysis for JAVA

Most prior studies on alias analysis for Java programs are for compiler optimization, such as *synchronization removal* and *escape analysis* [2, 6, 16]. For such purposes, all alias relations in the program need to be extracted by the analysis. Since the optimized program should compute the same execution results as the original program, analysis results must satisfy conservative approximation. We believe that alias analysis is useful for program maintenance activities as a software engineering tool. For such activities, not all the alias relations are needed at one time; only user-requested relations on the specific scope or object are to be extracted quickly. Also, good GUI that intuitively presents the analysis results to the user is very important.

Currently, since JAAT's control-flow representation does not consider possible control-flows caused by exceptions or threads, the resulting aliases contain surplus alias relations for exceptions and they lack the aliases caused by shared variables in threads. For exceptions, we applied a conservative approach that assumes all possible exceptions that might occur. However, since many researchers have already proposed control-flow analysis methods for threads and exceptions, we will adopt them to construct the improved control-flow representation [17, 20].

6.2 Two-phase Approach

Several prior studies also propose two-phase approaches such as intra-procedural analysis in advance using FS approaches [5, 8] or a FI approach [14].

In those FS approaches, each element $\mathcal{R}$ (alias relation) in an RAset holds conditions if a specific alias relation $\mathcal{R}_0$ really exists (if **true**, $\mathcal{R}$ exists). These conditions are used for indirect alias relations; however, all combinations of accessible variables should be taken into account as candidates for $\mathcal{R}_0$. In our method, since each AFG edge represents a direct alias relation, we can easily extract each indirect alias relation as an AFG path. These conditional-based algorithms are suitable as back-end for data-flow analysis (e.g., program slicing), which requires whole alias relations in the target programs.

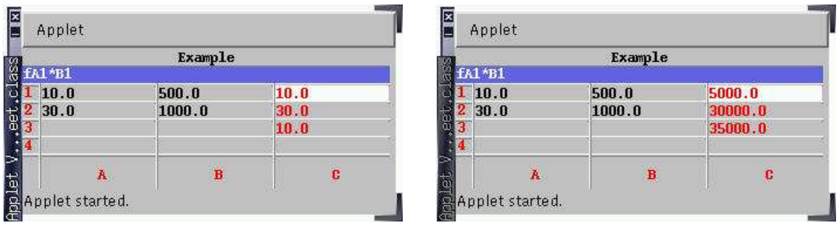
Since we focus mainly on program maintenance activities using the alias information itself, a simpler representation is useful. In such cases, the maintainers prefer the local alias information on which they focus. Also, we believe that they would request quick and simple answers even if the resulting aliases might be insufficient.

6.3 Instance-based Analysis

The instance-based approach was proposed in *object slicing*, which is a method for slicing OO programs proposed by Liang et al. [13]. They extend the *system dependence graph (SDG)* and define the traversal algorithm to compute slices with regard to a specific object; however, they assume that the 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.

6.4 On-demand Alias Extraction

We have applied an on-demand approach to alias analysis. Although alias information has been used for compiler optimization, data-flow analysis and so on, alias information is itself useful for program maintenance activities such as program debugging and understanding of Java programs. Java programs generally have many aliases caused by reference-type expressions, and some are not easily identified by the developer. Since all alias information in the target program is not necessary on program maintenance activities, we believe on-demand (or query-based) analysis will be a cost-effective approach. Although Heintze et al. have proposed a demand-driven pointer analysis, their approach is FI for procedural language C, and their goal is to compute the full point-to graph [9].



fA1*B1

1	10.0	500.0	10.0
2	30.0	1000.0	30.0
3			10.0
4			
	A	B	C

Applet started.

(a) Incorrect output

fA1*B1

1	10.0	500.0	5000.0
2	30.0	1000.0	30000.0
3			35000.0
4			
	A	B	C

Applet started.

(b) Correct output

Figure 11. Output of case study

7 Conclusions

We have proposed an alias analysis algorithm for JAVA programs, which is a scalable and on-demand algorithm with high precision and extensibility. Also, we have implemented this algorithm in the tool JAAT, and evaluated its effectiveness.

It has been presented that the analysis cost is reduced by our modularized approach which requires to re-analyze only modified modules, without re-analysis of other stable modules including libraries. In the future, we will include a feature to automatically identify modified modules and stable modules.

Acknowledgments
Authors are grateful to Yusuke Yamanaka, Kazuhiro Kondo, and Kazuhiro Kondo for the development of JAAT GUI.

References

[1] G. Agrawal and L. Guo, *Evaluating explicitly context-sensitive program slicing*, Program Analysis For Software Tools and Engineering, pp.6–12, Snowbird, UT, 2001.

[2] B. Blanchet, *Escape Analysis for Object-Oriented Languages: Application to Java*, Object-Oriented Programming Systems, Languages & Applications, pp.20–34, Denver, CO, 1999.

[3] M. Burke, P. Carini, J. D. Choi and M. Hind, *Flow-Insensitive Interprocedural Alias Analysis in the Presence of Pointers*, 7th Languages and Compilers for Parallel Computing, 1994, Ithaca, NY.

[4] D. R. Chase, M. N. Wegman and F. K. Zadeck, *Analysis of Pointers and Structures*, PLDI, pp.296–310, White Plains, NY, 1990.

[5] R. K. Chatterjee, B. G. Ryder and W. Landi, *Relevant Context Inference*, POPL, pp.133–146, San Antonio, TX, 1999.

[6] J. D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar and S. P. Midkiff, *Escape Analysis for Java*, OOPSLA, pp.1-19, Denver, CO, 1999.

[7] M. Enami, R. Ghiya and L. J. Hendren, *Context-sensitive interprocedural points-to analysis in the presence of function pointers*, PLDI, pp.242–256, Orlando, FL, 1994.

[8] M. J. Harrold and G. Rothermel, *Separate Computation of Alias Information for Reuse*, IEEE TSE,22-7, pp.442–460, 1996.

[9] N. Heintze and O. Tardieu, *Demand-Driven Pointer Analysis*, PLDI, pp.24–34, Snowbird, UT, 2001.

[10] M. Hind, *Pointer Analysis: Haven't We Solved This Problem Yet?*, Program Analysis for Software Tools and Engineering, pp.54–61, Snowbird, UT, 2001.

[11] W. Landi and B. G. Ryder, *A Safe Approximate Algorithm for Interprocedural Pointer Aliasing*, PLDI, pp.235–248, San Francisco, CA, 1992.

[12] W. Landi, B. G. Ryder and S. Zhang, *Interprocedural Modification Side Effect Analysis With Pointer Aliasing*, PLDI, pp.56–67, Albuquerque, NM, 1993.

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

[14] D. Liang and M. J. Harrold, *Efficient Points-to Analysis for Whole-Program Analysis*, 7th ESEC/FSE, pp.199–215, Toulouse, France, 1999.

[15] F. Ohata and K. Inoue, *JAAT: A Practical Alias Analysis Tool for Java Programs*, Technical Report of Osaka University, Department of Information and Computer Sciences, SE-lab-362, Dec, 2001.

[16] A. Rountev, A. Milanova and B. G. Ryder, *Points-To Analysis for Java using Annotated Constraints*, OOPSLA, pp.43–55,Tampa, FL, 2001.

[17] R. Rugina and M. C. Rinard, *Pointer Analysis for Multithreaded Programs*, PLDI, pp.77–90, 1999, Atlanta, GA.

[18] B. G. Ryder, W. Landi, P. Stocks, S. Zhang and R. Altucher, *A schema for interprocedural modification side-effect analysis with pointer aliasing*, ACM TOPLAS, 23-2, pp.105–186, 2001.

[19] M. Shapiro and S. Horwitz, *Fast and accurate flow-insensitive point-to analysis*, POPL, pp.1–14, Paris, France, 1997.

[20] S. Sinha and M. J. Harrold, *Analysis of Programs With Exception-Handling Constructs*, ICSM, pp.358–367, Washington, D.C., 1998.

[21] B. Steensgaard, *Points-to analysis in almost linear time*, POPL, pp.32–41, St. Petersburg Beach, FL, 1996.

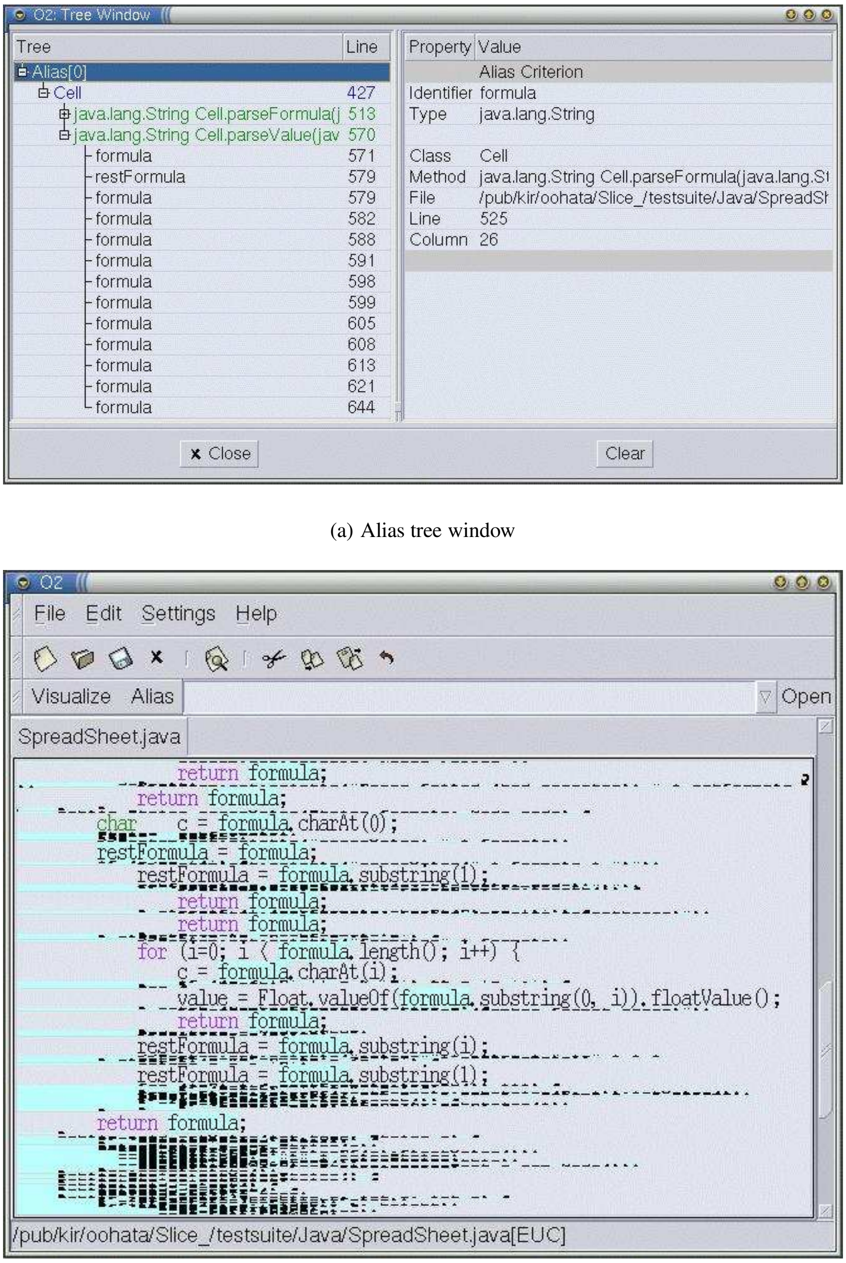
[22] P. Tonella, G. Antoniol, R. Fiutem and E. Merlo, *Flow insensitive C++ pointers and polymorphism analysis and its application to slicing*, 19th ICSE, pp.433–443, Boston, MA, 1997.

[23] R. P. Wilson and M. S. Lam, *Efficient context-sensitive pointer analysis for C programs*, PLDI, pp.1–12, 1995, La Jolla, CA.

[24] S. H. Yong, S. Horwitz and T. W. Reps, *Pointer Analysis for Programs with Structures and Casting*, PLDI, pp.91–103, Atlanta, GA, 1999.

[25] S. Zhang, B. G. Ryder and W. A. Landi, *Experiments with Combined Analysis for Pointer Aliasing*, Program Analysis for Software Tools and Engineering, pp.11–18, Montreal, Canada, 1998.

[26] S. Zhang, B. G. Ryder and W. A. Landi, *Program Decomposition for Pointer Aliasing: A Step Toward Practical Analyses*, 4th FSE, pp.81–92, San Francisco, CA, 1996.



Tree	Line
Alias[0]	
Cell	427
java.lang.String Cell.parseFormula(j	513
java.lang.String Cell.parseValue(jav	570
formula	571
restFormula	579
formula	579
formula	582
formula	588
formula	591
formula	598
formula	599
formula	605
formula	608
formula	613
formula	621
formula	644

Property	Value
	Alias Criterion
Identifier	formula
Type	java.lang.String
Class	Cell
Method	java.lang.String Cell.parseFormula(java.lang.St
File	/pub/kir/oohata/Slice_/testsuite/Java/SpreadSh
Line	525
Column	26

(a) Alias tree window

```
                return formula;
            return formula;
        char    c = formula.charAt(0);
        restFormula = formula;
            restFormula = formula.substring(1);
                return formula;
                return formula;
            for (i=0; i < formula.length(); i++) {
                c = formula.charAt(i);
                value = Float.valueOf(formula.substring(0, i)).floatValue();
                return formula;
            restFormula = formula.substring(i);
            restFormula = formula.substring(1);
        return formula;
```

/pub/kir/oohata/Slice_/testsuite/Java/SpreadSheet.java[EUC]

(b) Text window

Note: only the lines with alias expressions are displayed normally, and other lines are compressed as dotted lines.

Figure 12. JAAT windows showing the result of case study