# Alias Analysis for Java with Reference-Set Representation

Jongwook Woo
*Department of EE-Systems*
*University of Southern California*
*Los Angeles, CA 90089-2563*
*jongwook@usc.edu*

Jehak Woo
*Computer Engineering Department*
*Daejin University, Korea*
*jhwoo@computer.org*

Isabelle Attali Denis Caromel
*INRIA Sophia Antipolis*
*University of Nice Sophia Antipolis*
*BP 93, 06902 Sophia Antipolis Cedex - France*
*{Isabelle.Attali, Denis.Caromel}@sophia.inria.fr*

Jean-Luc Gaudiot
*Department of EE-Systems*
*University of Southern California*
*Los Angeles, CA 90089-2563*
*gaudiot@usc.edu*

Andrew L Wendelborn
*Department of Computer Science*
*University of Adelaide, SA 5005*
*andrew@cs.adelaide.edu.au*

## Abstract

*We propose a flow-sensitive context-insensitive alias analysis in Java that is more efficient and precise than previous analyses in C++. For that, we propose a reference-set alias representation. Second, we present the propagation rules for the reference-set alias representation. Third, for the type determination, the type table is built with reference variables and all possible types of the reference variables. Fourth, we propose an algorithm in a popular iterative loop method with a structural traverse of a CFG. Finally, we show that our reference-set representation has better performance for the alias analysis algorithm than the existing object-pair representation.*

## 1. Introduction

Java has become a popular language because it is platform independent and object-oriented. More specifically, in Java, objects are accessed by references, consequently, there might be many aliases in a piece of Java code. An alias situation is said to have occurred when an object is bound by more than two names. Recent studies [1, 2, 3, 4, 5, 6, 8, 9, 13] have analyzed aliases to avoid side effects statically because codes with alias information are useful and better candidates for high performance computing such as parallelizing compiler as well as compiler optimization. With detected aliases information, we may avoid race conditions, context switch and communication overhead for parallelizing compilers.

This material is based upon work supported by the National Science Foundation under Grants No. CSA-0073527 and INT-9815742. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Previous studies [2, 3, 4, 5, 6, 8, 9] proposed alias analyses for C/C++ by representing the alias relation with objects because of the concepts of pointers and pointer-to-pointer in C/C++. This research aims at improving the efficiency and the accuracy based on the safety of the alias information detected. However, the representation of alias relations is not sufficiently optimized to apply to Java. Thus, we have proposed *referred-set* representation [1] that makes a more efficient and precise analysis for Java than previous methods. However, it might have a large time complexity in the usage of the computed alias set. Thus, in this paper, we propose an alternative alias relation, *reference-set* representation by extending our *reference-pair* representation which is a pair of reference variables [13].

Our alias analysis in Java presents four contributions for the efficiency and preciseness without losing its safety. First, we analyze the existing alias representations in C++ [2, 3, 4, 6] in order to apply to Java. Second, we introduce the *reference-set* representation to present an alias information in Java. Third, we propose more precise data propagation rules of aliases for the *reference-set* representation. Finally, based on existing

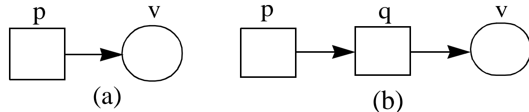


Figure 1. Relation between a pointer and an object in C++

algorithms [1, 6, 13], our calling graph, control flow graphs and a type table, we propose a more precise and efficient flow-sensitive alias analysis algorithm.

In this paper, section 2 presents the differences between C++ and Java and introduces the *reference-set* alias representation for Java. Section 3 describes the structures of our algorithm. Section 4 explains our propagation rules for *intraprocedural* and *interprocedural* analyses. Section 5 shows the alias analysis algorithm. Section 6 computes the complexities of our and existing algorithms. Section 7 shows the experimental results of the *reference-set* and existing *object-pair* representations. Finally, the conclusion is presented.

## 2. Alias Relation Representation in Java

### 2.1. Differences between C++ and Java

Naming of an object should be considered to represent aliases in C++ and Java. In C++, static objects declare object names. Also, dynamic objects and pointer-valued objects have their own names for an alias analysis. A pointer variable name is a name to point an object that contains the address of a pointed-to object. In Figure 1 (a), pointer variable name $p$ is naming a pointer-valued object that contains its address value. Dereferenced pointer $p$ is naming the object that is pointed to by $p$. A variable name $p$ that is not a pointer is naming an object that contains the address of the variable. There exist alias relations among pointer-valued objects because of pointer-to-pointer relationships. Therefore, in the previous studies [5, 6, 7], when pointer $p$ points to an object of $v$, the alias relation is represented as <*p, v>. Figure 1 (b) shows that a pointer points to another pointer variable that complicates the alias analysis, where a box depicts a pointer-valued object and a circle is a non-pointer object. Those alias relations are represented as <*p, q> and <*q, r>.

Existing alias relations in C++ are similar compact [5, 6, 7] and points-to [2, 3, 4] representation. In this paper, we call them as *object-pair* representation because those are a pair of objects. Those relations save spaces by representing all alias relations without using an exhaustive set. Those relations can be used in Java. However, there are some problems to use those representations because only references are used to name objects in Java. A reference is a variable that refers to

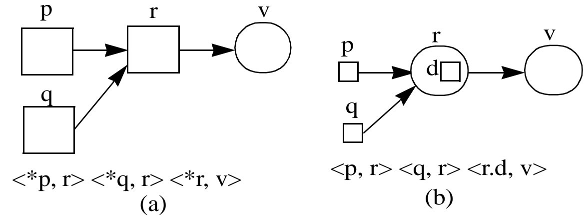


Figure 2. Difference between a pointer in C++ and a reference in Java

an object as a pointer in C++. There are no pointer-to-pointer concepts and no pointer operations in Java. An object in Java is created dynamically so that the object becomes an anonymous object that does not have its own name. Thus, each object needs its own naming by binding a reference name and an object name in an alias relation.

In Figure 2 (a), if there is an assignment statement $*p = w$, the value of the addressed valued object named by $p$ is changed to the value of the addressed valued object bound by $w$. Thus, the object pointed by $r$ is changed via <*p, r>. Therefore, <*r, v> can be killed and a new alias relation <*r, w> is generated.

In Figure 2 (b), an alias relation via compact representation in Java is shown. If there is an assignment statement $p.d = w$, the addressed value of the reference $p.d$ is changed so that <p.d, r.d> is inferred and $p.d$ and $r.d$ are considered as an alias of the same address-valued object. However, if $v$ and $r.d$ are recognized as an alias of the same address-valued object in <r.d, v>, an object referred to by the reference $v$ is changed and the wrong alias relations <r.d, w> and <v, w> are generated. For the correct relation, <r.d, v> should be killed and <r.d, w> should be generated. The wrong result comes from the fact that, in Java, a reference name is used for naming an object without a dereferencing operator such as * in C++. To obtain a right result in this example, $r.d$ should be recognized as a memory location that contains its addressed value in <p.d, r.d>. Also, $r.d$ should be recognized as an object that is referred to by the reference $r.d$. To solve this problem, an alias relation for an address-valued object should be presented by extending a compact representation. Otherwise, a data flow equation for aliases should recognize the difference. Therefore, reference names for an alias relation should be meant as dereferencing and the <p.d, r.d> alias relation for the alias computation should be analyzed differently.

### 2.2. *Reference-Set* Representation

For a more precise alias analysis in object-oriented languages, the type information of the objects accessed are needed and this information can be collected more safely via alias information [2, 3]. It is known as a type

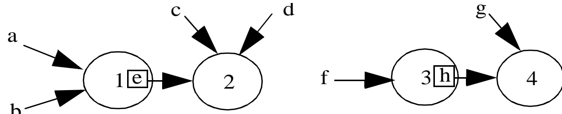


Figure 3. The relationships between reference and objects

inference. The type information can be used for overridden methods resolution in Java. The more precise the type information, the more precise alias analysis becomes.

In this section, the *reference-set* representation is proposed to improve the efficiency of the alias computation and the type inference.

***Reference-set:*** *a set of alias references that consist of more than two references which refer to an object;* $R_i = \{r_1, r_2, ..., r_j\}$*: for each j, initially* $j > 2$ *and* $r_j$ *is a reference for an object; when* $r_j$ *and* $r_k$ *are in the same path and qualified expressions with a field f,* $r_j$ *and* $r_k$ *can be represented with a* $R_i.f$ *with a reference-set* $R_i$ *for an object i; During data flow computing in an alias analysis,* $j > 1$ *when passing and passing back an object at a call site.*

The *alias set* contains the entire alias information at the statement.

***Alias set:*** *a set of reference-sets at a statement s;* $A_s = \{R_1, R_2, ..., R_i\}$

In a statement *s* of a program, each *reference-set* and *alias set* for the alias relation in Figure 3 are represented as follows.

$R_1 = \{a, b\}$ $R_2 = \{R1.e, c, d\}$ $R_4 = \{f.h, g\}$

$A_s = \{R_1, R_2, R_4\}$

An alias analysis algorithm computes the alias sets in a program. Each statement collects an alias set from its predecessor and updates it with the statement itself and passes the resulting alias set to its successor(s). Since the alias computation should be iteratively done until the alias sets and a calling graph have converged for the program, it affects the efficiency of the whole algorithms. For example, there is an assignment statement $a = g$ in Figure 3. It means that the reference *a* refers to an object that the reference *g* refers to. Therefore, the element *a* of the reference-set $R_1$ is killed and the elements *a* should be copied to the reference-set $R_4$. The time complexity of this computation depends on the space complexity of each representation. Thus, the efficiency of whole algorithms is improved via *reference-set* representation.

## 3. Data Structures

Aliases can be computed with data-flow equations. For the computation of the aliases, we define our dataflow equations, calling graph (*CG*), and control flow graph (*CFG*) in this section. A type table contains all possible types of reference variables.

A *CG* is needed to compute the alias set of an interprocedural analysis between a calling and a called methods at a call statement. Our *CG* is a directed graph defined as $< N_{CG}, E_{CG}, n_{main}>$, where $N_{CG}$ is a set of nodes and each node is a method shown one time in a *CG* even though it may be called many times; where $E_{CG}$ is a set of directed edges connected from caller(s) to callee(s) and one edge is connected even though a caller may invoke the callee many times and all edges are connected when many callers invoke one callee; $n_{main}$ is the main method that executes initially in a Java program. During our algorithm proceeds, our *CG* grows as in the previous works [4, 5, 6, 7] by adding nodes.

*CFG*s can be used to compute the alias sets of the intraprocedural propagations. Our *CFG* is a directed graph defined for each method as $<N_{CFG}, E_{CFG}, n_{entry}, n_{exit}>$, where $N_{CFG}$ is a set of nodes with $n_{entry}$, $n_{exit}$, and each statement of the method; $E_{CFG}$ is the set of directed edges that represent the control and alias set information between a predecessor and a successor statements; $n_{entry}$ represents the entry node of the method; $n_{exit}$ represents the exit node of the method.

In our *CFG*, seven node types are proposed based on their purposes: *Entry* that is the $n_{entry}$ of the *CFG*, *Exit* that is the $n_{exit}$ node, *Assignment statement*, *Call statement*, *Return Statement*, *Flow construct node* (*if, while* etc.), and *Merging nodes*.

The *flow construct node* is a node which signifies the start of the *if* or *while* clause. In an *If* node, each clause is branched from the node. All the branched clauses are merged into a merging node. In a *while* node, a *merging node* is not necessary and a directed edge is connected from the last node of the *while* loop to the *flow construct node*.

Reference variables dynamically refer to objects in Java. Thus, the types of a reference variable are determined statically at the type declaration and dynamically during the processing of the algorithm. A type table is built during the process and it contains three columns: the reference variable, its declared type, and its overridden method types. Type inference can be processed with a type table which contains the declared and dynamic types of each reference variable. The declared type represents static and shadowed variable type information of a reference variable. The dynamic types represent possible overridden method types of the reference variable. Types of each reference variable can be given in a constant time.

## 4. Propagation Rules of Alias Information

The propagation and computation of alias information is made through the nodes in a *CFG* of each method. Let *in(n)* and *out(n)* be the input alias set of a node *n* transferred from predecessor nodes and output alias set held on exit from a node *n* respectively.

$$in(n) = \cup\, out(pred(n))$$

$$out(n) = Trans(in(n)) = Mod_{gen}\,[Mod_{kill}(in(n))]$$

In this equation, *pred(n)* represents a predecessor node of the node *n*. $Mod_{kill}$ denotes the alias set modified after killing some *reference-set*s of *in(n)* and $Mod_{gen}$ is the subsequent alias set after generating the new *reference-set*s on $Mod_{kill}$.

### 4.1. Rules for Intraprocedural Analysis

The propagation rules for intraprocedural analysis are described below for every *CFG* node type except an *entry* and a *call statement node* type. The rule consists of premises and conclusions divided by a horizontal line. The premise can have the form of conditional implication that is interpreted as follows: when a given condition holds, the implied equation has a meaning and can be solved. When all premises hold, the equations in the conclusions are solved for *out(n)*.

First, we define a *flow construct* and a *merging node* type rule as follows: $n_{pred}$ is a predecessor set of node *n*. Given $n_{pred}$ , *out(n)* of node *n* is the union of all predecessor node sets.

$$\frac{in(n) = \bigcup_{p \in n_{pred}} out(p) \qquad n_{pred} : \text{predecessor node of } n}{out(n) = in(n)} \quad \text{[Flow Construct/Merging Node]}$$

The next rule concerns the node type of an assignment statement.

$$\frac{\begin{array}{l} in(n) = out(n_{pred}) \\ n_{pred} : \text{predecessor node of } n \\ x = LHS, \\ y = RHS, \\ \forall i, j \;\; R_i, R_j \in in(n) \;\rightarrow [Mod_{kill}(in(n)) = \{R_i \mid kill\ x \in R_i\} \\ \qquad\qquad \cup \{R_i \mid kill\ R_j.f \in R_i\ when\ q \in R_j\ and\ x = q.f\}] \\ \qquad\quad \wedge [in(n) = in(n) - Mod_{kill}(in(n))] \wedge [KILL(in(n)) = \{x, R_j.f\}], \\ \forall k\ R_k \in in(n) \rightarrow [Mod_{gen}(in(n)) = \{R_k \mid R_k = R_k \\ \quad \cup KILL(in(n))\ when\ y \in R_k\}] \;\wedge [in(n) = in(n) - Mod_{gen}(in(n))], \end{array}}{out(n) = Mod_{kill}(in(n)) \cup Mod_{gen}(in(n)) \cup in(n)} \quad \text{[Assignment Node]}$$

*LHS* and *RHS* respectively stand for the left hand side and the right hand side of an assignment statement. *KILL(in(n))* is a *reference-set* of references killed by $Mod_{kill}(in(n))$. *out(n)* of the node *n* is a union of $Mod_{kill}(in(n))$, $Mod_{gen}(in(n))$, and *in(n)*.

In order to show how the above rule can be applied to alias analysis, we analyze an assignment statement *a.e = f.h* in a statement of Figure 3. Initially, *reference-set R1, R2, R3* and alias set *in(n)* are expressed as follows for the statement:

$$R1 = \{a, b\} \qquad R2 = \{R1.e, c, d\} \qquad R3 = \{f.h, g\}$$

$$in(n) = \{R1, R2, R3\}$$

Because *LHS* is a qualified expression related to both *R1* and *R2*, $Mod_{kill}(in(n))$, *in(n)*, and *KILL(in(n))* are computed as follows:

$R1 = \{a, b\}$ and $R2 = \{R1.e, c, d\}$ then $R2 = \{c, d\}$

$$Mod_{kill}(in(n)) = \{R2\} \qquad in(n) = \{R1, R3\} \qquad KILL(in(n)) = \{R1.e\}$$

Since *R3* includes *RHS*, $Mod_{gen}(in(n))$ and *in(n)* are computed as follows:

$$Mod_{gen}(in(n)) = \{R3 \mid R3 = R3 \cup \{R1.e\} = \{R1.e, f.h, g\}\} = \{R3\}$$

$$in(n) = \{R1\}$$

Finally, *out(n)* is the union set of $Mod_{kill}(in(n))$, $Mod_{gen}(in(n))$, and *in(n)* as follows:

$$out(n) = Mod_{kill}(in(n)) \cup Mod_{gen}(in(n)) \cup in(n) = \{R2, R3, R1\}$$

$$when\ R1 = \{a, b\},\ R2 = \{c, d\},\ R3 = \{R1.e, f.h, g\}$$

The rule for the *return statement node* type is presented as follows with the *reference-set* of a return variable *r*. In the rule, *LOCAL* stands for a local variable set defined in a method *M* such as local variable and formal parameter variables.

$$\frac{\begin{array}{l} in(n) = out(n_{pred}) \\ n_{pred} : \text{predecessor node of } n \\ M: callee,\ LOCAL(M) = \{v \mid v\ is\ a\ local\ variable\ of\ M\} \\ \forall i \;\; R_i \in in(n)\ for\ r:\ return\ reference \\ \quad \rightarrow [Mod_{kill}(in(n)) = \{R_i \mid kill\ x \in R_i\ for\ x \in LOCAL(M)\}] \\ \;\; \wedge [R_r = \{R_r \mid kill\ x \in R_r\ for\ x \in LOCAL(M)\ when\ r \in R_r\}], \end{array}}{out(n) = Mod_{kill}(in(n))} \quad \text{[Return Node]}$$

The next is the rule for an *exit node* type.

$$\frac{\begin{array}{l} in(n) = \bigcup_{p \in n_{pred}} out(p) \\ n_{pred} : \text{predecessor node of } n \\ M: callee,\ LOCAL(M) = \{v \mid v\ is\ a\ local\ variable\ of\ M\} \\ \forall i \;\; R_i \in in(n) \\ \quad \rightarrow [Mod_{kill}(in(n)) = \{R_i \mid kill\ x \in R_i\ for\ x \in LOCAL(M)\}] \\ \qquad \wedge [in(n) = in(n) - Mod_{kill}(in(n))], \end{array}}{out(n) = Mod_{kill}(in(n)) \cup in(n)} \quad \text{[Exit Node]}$$

### 4.2. Rules for Interprocedural Analysis

Interprocedural propagation rules should be considered for a *call statement node* and an *entry node*. The data flow of an alias set in a call statement denotes that an alias information of the statement is propagated to a called method and it affects an alias information of the

called method. The affected information are passed back to the call statement of the calling method after computing the alias set of the called method. The alias set from the called method modifies the alias set of the call statement when the return alias set includes non-local variables and actual parameters.

We virtually divide a call node into a *precall* node and a *postcall* node to simplify the computation of a call statement. A *precall* node collects an alias set from a predecessor node of a current call node and computes its own alias set *out(n)* with the collected set. This alias set is propagated to the entry node of the called method. During the propagation, the *reference-set*s for references which are inaccessible from the called method are killed. Since this set is an input of the *postcall* node and is not modified, it does not need to propagate to the called method. The *out(n)* of the *precall* node is not propagated to the *postcall* node because the called method might modify the set. As in previous approaches [2, 3, 4], if we do not kill the alias relations affected by the called method for the subsequent analysis, it might build nonexistent call relations and cause the subsequent analysis to become inefficient.

A *postcall* node collects the modified kill set of the *precall* node and exit nodes alias set of all possible called methods. The following rule computes an out set of a *precall* node.

$in(n) = out(n_{pred})$,
$n_{pred}$ : predecessor node of n
$RHS = E_c.M_c$,
$RHS = M_c$,
$\forall i,\ a_i =$ the $i$th actual parameter of the callee $M_c$,
$\forall i,\ f_i =$ the $i$th formal parameter of the callee $M_c$,
$\forall i,\ R(a_i) \in in(n)$
$\quad \rightarrow [R_{pass}(a_i) = \{a_i, f_i\}] \wedge [R(a_i) = R(a_i) - R_{pass}(a_i)]$,
$RHS = M_c,\ \forall i,\ R(a_i) \in in(n)$, $v$ is a non local variable in the callee,
$M_c \rightarrow [R(v) = R(v) - \{v\}] \wedge [R_{pass}(v) = \{v\}]$
$\quad \wedge [PASS(M_c) = \cup \{R_{pass}(a_i), R_{pass}(v)\}]$,
$RHS = E_c.M_c,\ \forall i,\ R(a_i) \in in(n)\ \forall f$,
$R(E_c.f) \in in(n)$
$\quad \rightarrow [R(E_c.f) = R(E_c.f) - \{E_c.f\}] \wedge [R_{pass}(E_c.f) = \{E_c.f\}]$
$\quad \wedge [PASS(M_c) = \cup \{R_{pass}(a_i), R_{pass}(E_c.f)\}]$
——————————————————————— [Precall Node]
$out(n) = in(n)$

*PASS*($M_c$) represents the set of actual, formal parameters and non-local variables in a called method $M_c$. $R_{pass}(a_i)$ is a set of reference variables accessible by a called method when passing from a caller to the called method $M_c$. $R_{pass}(v)$ is a set of non-local variables accessible by a called method in the called method $M_c$.

In the following propagation rule of an *entry node*, the propagated set can be computed as in an *assignment statement node*. *PRECALL*($M_c$) is a *precall* node of *call statement node*s that invoke this called method node. This set can be computed by considering ingoing edges of the called method $M_c$ in a *CG*. An entry node merges alias sets from the *precall* nodes and then propagates the merged set to its subsequent node.

$PRECALL(M_c)$ : a precall node of the callee $M_c$,
$in(n) = \bigcup_{p \in PRECALL(M_c)} PASS(p)$
——————————————————————— [Entry Node]
$out(n) = in(n)$

The rule of the *postcall* node is defined as follows.

$in(n) = \bigcup_{p \in n_{precall}} out(n_{precall})$
$n_{precall}$ : a precall node of n
$RHS = E_c.M_c \rightarrow$
$\quad FIELD(E_c) = \{f \mid f \text{ is a field name in an object referred by } E_c\}$,
$RHS = M_c \rightarrow FIELD(E_c) = \emptyset$,
$RHS = new\ M_c \rightarrow FIELD(E_c) = \emptyset \wedge A(r)$,
$EXIT(M_c) = \{e \mid e \text{ is an exit alias set from a possible callee method } M_c\}$,
$LHS = \emptyset, \forall R_{passb} \in EXIT(M_c)$
$\quad \rightarrow [R_{passb} = R_{passb} - \{v \mid v \text{ is a local variable in the callee } M_c\}]$
$\quad \wedge [EXIT(M_c) = \bigcup_{\text{for all } M_c} R_{passb}]$,
$LHS \neq \emptyset, \forall R_{passb} \in out(\text{return node})$
$\quad \rightarrow [R_{passb} = R_{passb} - \{v \mid v \text{ is a local variable in the callee } M_c\}]$
$\quad \wedge [EXIT(M_c) = \bigcup_{\text{for all } M_c} R_{passb}]$,
$\forall i\ R_i \in EXIT(M_c),\ \forall j\ R_j \in in(n) \rightarrow [R_i \mid R_i = R_i \cup R_j \text{ when } i=j]$
$\quad \wedge [EXIT(M_c) = EXIT(M_c) - R_i] \wedge [in(n) = in(n) - R_j]$,
$exit(RHS) = \bigcup_{e \in EXIT(M_c)} out(e) \cup \bigcup_{p \in n_{precall}} out(\text{precall node}) \cup \bigcup_{\text{for all } i} R_i$,
$LHS = \emptyset \rightarrow out = exit(RHS)$,
$LHS = x, \forall i, j\ \ R_i, R_j \in exit(RHS), R(RHS) \in exit(RHS)$
$\quad \rightarrow [Mod_{kill}(exit(RHS)) = \{R_i \mid kill\ x \in R_i\} \cup \{R_i \mid kill\ R_j.f \in R_i$ when $q \in R_j$ and $x = q.f\}] \wedge [KILL(exit(RHS)) = \{x, R_j.f\}]$
$\quad \wedge [exit(RHS) = exit(RHS) - Mod_{kill}(exit(RHS))]$,
$R(RHS) \in exit(RHS) \rightarrow [Mod_{gen}(exit(RHS))$
$\quad = \{R(RHS) \mid R(RHS) = R(RHS) \cup KILL(exit(RHS))\}]$
$\quad \wedge [exit(RHS) = exit(RHS) - Mod_{gen}(exit(RHS))]$
$\quad \wedge [out = Mod_{kill}(exit(RHS)) \cup Mod_{gen}(exit(RHS)) \cup exit(RHS)]$
——————————————————————— [Postcall Node]
$out(n) = out$

*Exit(RHS)* is a set of *exit node*s of all possible called methods as explained before. We can compute *exit(RHS)* in a *CG* by integrating all *out(precall node)* and outgoing edges from callers and their *exit node*s. *Out* is an alias set of the *exit node* of a callee.

If we assume the Figure 4 (a) as a status after executing a statement $s$, the alias set $A_s$ of the statement $s$ is:

$A_s = \{R_2, R_3\}$
where $R_2 = \{a.f, b, c, R_3.f\}$ and $R_3 = \{R_2.f, c\}$

After executing the call statement $t$ in Figure 4 (b), the result alias set of its *precall* node can be computed in the following sequence of rule applications:

$in(t) = \{R_2, R_3\}$,
$RHS = a.update(c)$,

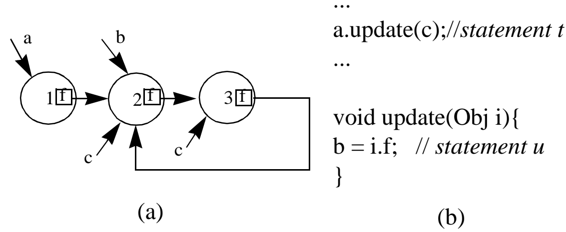


Figure 4. Example of an interprocedural analysis

$a_i = c, f_i = i,$

$R_{pass}(a_i) = \{c, i\}, R(a_i) = R_2 = R_2 - \{c\} = \{a.f, b, R_3.f\}$

$or\ R(a_i) = R_3 = R_3 - \{c\} = \{R_2.f\},$

$R(a.f) = R_2 = R_2 - \{a.f\} = \{b, R_3.f\}\ and\ R_{pass}(a.f) = \{a.f\},$

$PASS(a.update) = \{R_{pass}(a_i), R_{pass}(a.f)\},$

$out(t) = \{R_2, R_3\}$

The *PASS(a.update)* of the *precall* node propagates to the *entry node* of the callee *update()*. The result alias set of the *exit node* can be computed as follows:

$R_{pass}(a_i) = \{c, i\}, R_{pass}(a.f) = \{a.f\},$

$R_{pass}(a_i) = R_{pass}(a.f) = \{c, i, a.f\} = R_{pass}(R_2)\ for\ R_2,$

$R_{pass}(a_i) = \{c, i\} = R_{pass}(R_3)\ for\ R_3,$

$in(u) = \{R_{pass}(R_2), R_{pass}(R_3)\},$

$R(b) = \{b, i.f, c.f\},$

$out(u) = update_{exit} = \{R(b), R_{pass}(R_2), R_{pass}(R_3)\}$

The result set of the *postcall* node at the statement *t* is computed with the exit alias set of the *update()* and the propagation rule of the *postcall* node as follows:

$in(t) = out(t_{precall}) = \{R_2, R_3\}$ where $R_2 = \{b, c.f\}, R_3 = \{c.f\},$

FIELD(a) $= \{a.f\},$

$EXIT(update) = update_{exit} = \{R(b), R_{pass}(R_2), R_{pass}(R_3)\},$

where $R_{pass}(R_2) = \{c, i, a.f\}$ and $R_{pass}(R_3) = \{c, i\}$ ,

$R(b) = R_{passb} = \{b, c.f\}, R_{passb}(R_2) = \{c, a.f\}, R_{passb}(R_3) = \{c\}$

for the caller,

$EXIT(update) = \{R(b), R_{passb}(R_2), R_{passb}(R_3)\},$

$R_2 = R_2 \cup R_{passb} \cup R_{passb}(R_2) = \{b, R_3.f, c.f, c, a.f\},$

$R_3 = R_3 \cup R_{passb} \cup R_{passb}(R_3) = \{R_2.f, b, c.f, c\},$

Thus, $out(t) = \{R_2, R_3\}$

## 5. Alias Analysis Algorithm

Figure 5 represents our alias analysis algorithm. This algorithm is adapted on the interprocedural type analysis algorithm [6]. It is one of iterative methods for interprocedural data flow analysis based on a *CG* [10]. The iterative algorithm executes its computation by visiting all nodes of a *CG* in order until fixed point of the data status and nodes are achieved.

Our algorithm traverses each node of a *CG* in a topological order and a reverse topological order in order to possibly shorten the execution time for the fixed point [5, 6, 7]. The ending point in our algorithm means that the topology of a *CG* and alias set *out(n)* are not changed anymore.

```
Algorithm AliasAnalysis
construct an initial CG with main method;
repeat {
  for each method T.M ∈ N_cg,
    alternating between topological and reverse topological order {
    for each node n ∈ N_cfg (T.M) in structural order {
      if n is a call statement node {
        if (RHS = E_c.M_c) {
          compute the set of inferred types
              from the reference-set for E_c;
          compute the set TYPES resolved
            from the inferred types and class hierarchy;
        } else if (RHS = M_c) {
          TYPES:= {T};
        } else if (RHS = new M_c) {
          TYPES:= {M_c};
        }
        if LHS exists
          TYPES_table(LHS) = TYPES_table(RHS);
        for each type t ∈ TYPES {
          if t.M_c is not in CG
            create a CG node for M_c;
          if no edge from T.M to t.M_c with a label n
            connect an edge from T.M to t.M_c
          with a label n;
        }
        compute out(n_precall) for a precall node n_precall;
        compute out(n_postcall) for a postcall node n_postcall;
      } else {
        if n is an assignment statement node
          TYPES_table(LHS) = TYPES_table(RHS);
        if n is a merging statement node
          TYPES_table(LHS) = TYPES_table(LHS) + TYPES_table(RHS);
        compute out(n) using data-flow equation
        and propagation rule;
      }
    }
  }
} until CG and alias set for every CFG node converge
```

Figure 5. Alias Analysis Algorithm

The set *TYPES* represents the possible class types for a callee to build a safe *CG*. $TYPES_{table}(r)$ is a set of dynamic types of a reference variable *r* in a type table. The reference *r* also maintains its static type in the type table. Inner loop is for intraprocedural alias analysis for each method. While proceeding the inner loop, each node of the *CFG* of a method is traversed with computing an alias set of each node and the computed alias set is propagated to the next node. Each node in our algorithm is visited on structural order for this. Structural order is defined that while visiting nodes from an entry node to an exit node, for the *if* flow construct node, each branch is traversed first then finally its merging node is visited. We improve the efficiency with the structural order than the previous work [6].

When the method of a *call statement node* is an overridden method, its resolution should be considered for the safety of the alias set. It is not possible to precisely predict the dynamic overridden method statically. However, we can store all possible types of each reference into a type table during computation. Thus, we can safely predict all possible methods invoked with the type table. If the method is defined in a type

inferred by this type inference, the method defined is a resolved method. By adding all possible methods to the *CG* via the searching, we can update the *CG*.

An alias set of each node can be computed as a result alias set *out(n)* with type information and our data-flow equations of the propagation rules.

## 6. Complexity of Algorithm

Our algorithm has three outer loops. For the most outer loop, $R_n$ and $A_r$ are the number of *reference-set*s and the maximum number of aliased reference variables for each *reference-set*. We can estimate the worst time complexity of the loop as $O(R_n \times A_r \times E_{cg})$ - $E_{cg}$ is the number of edges in a *CG*.

For the second outer loop, the time complexity becomes $O(N_{cg})$ if $N_{cg}$ is the final number of nodes in a *CG*. For the most inner loop, the time complexity is $O(N_{cfg})$ if $N_{cfg}$ is the maximum number of nodes in a *CFG* that consists of the maximum number of nodes.

The time complexity of a set of inferred types is $O(R_m)$ when $R_m$ is the number of reference variables in a program code.

The time complexity for the possible method resolution is $O(T_i \times H)$ when $T_i$ is the maximum number of subclasses for a superclass and $H$ is the maximum number of the levels in its hierarchy. The time complexity for the resolution of overridden methods and the updating of a *CG* is $O(T_i \times (H + N_{cg} + C_c))$ when $C_c$ is the maximum number of call statements to invoke same called methods in a calling method. The worst time complexity of a *precall* and a *postcall* nodes is $O(R_p \times R)$ when $R_p$ is the maximum number of *reference-set*s propagated and $R$ is the maximum number of reference variables in $R_p$ on a call statement.

Therefore, the worst time complexity of the main algorithm is $O(R_n \times A_r \times E_{cg} \times N_{cg} \times N_{cfg} \times (R_p \times R \times R_m + T_i \times (H + N_{cg} + C_c)))$.

## 7. Experimental Results

We have executed benchmark codes on alias analysis algorithms with the *reference-set* and the existing *object-pair* representations [2, 3, 4, 6, 7]. The alias analysis systems are built on JavaCC (Java Compiler Compiler) [11] and JTB (Java Tree Builder) [12]. JavaCC is the parser generator and JTB is a syntax tree builder to be used with the JavaCC parser generator. It automatically generates a JavaCC grammar with the proper annotations to build the syntax tree during parsing [12]. The syntax tree is extended by adding the data structures of *reference-set* and *object-pair* representations with class structures of TT and CFG [13]. We have executed three benchmark codes on the systems: *Dynamic CG*, *Binary Tree*, and *Ray Tracer*. The characteristic of each code is presented in Table 1. *Dynamic CG* is written in C++ initially by Carini [6] and adapted in Java by ourselves. It has conditional statements and overridden methods. *Binary Tree* is provided by Proactive group [14]. The binary tree contains many conditional statements and recursive calls that generate potential aliases dynamically. Both can be used to measure the safety, preciseness, and efficiency of the algorithms. *Ray Tracer* is one of Java Grande's benchmarks to measure the performance of a 3D raytracer [15]. Table 2 presents properties of the hosts for those benchmark codes.

**Table 1: Characteristics of Benchmark**

| | Num of classes | Num of lines | Num of methods /const. | Num of overrid. methods |
|---|---|---|---|---|
| **Dynamic CG** | 5 | 54 | 7 | 4 |
| **Binary Tree** | 5 | 154 | 8/2 | 1 |
| **Ray Tracer** | 12 | 1,213 | 53/13 | 4 |

**Table 2: Characteristics of hosts**

| | Kottos | Ceng | Asadal |
|---|---|---|---|
| **Host Type** | RS6000 | Sun4 | Windows 2000 |
| **OS** | AIX4 | SunOS5.6 | Windows NT |
| **Java VM** | JDK-1.1.1 | JDK-1.2.1_02 | JDK-1.2.2 |

Figure 6 presents the execution times of *Dynamic CG*. For all hosts, the execution time of *reference-set* is faster than *object-pair* because our *Type Table* has more efficient structure to search possible types of methods than Carini's [6]. Figure 7 is the execution times of *Ray Tracer*. It implies that the benchmark codes such as *Ray Tracer*, which do not contain many aliased references among objects and which is for JVM performance measurement, do not have any big difference in the execution time of alias analysis for any alias representation. Figure 8 and Figure 9 shows that the execution time of *reference-set* is faster than *object-pair*. Figure 10 presents that the execution time of *reference-set* is much faster than *object-pair* on *Ceng* and *Asadal* respectively. For *Kottos, object-pair* is not measurable because the execution time is too long.

## 8. Conclusion

We propose the flow sensitive alias analysis algorithm with *reference-set* alias representation for Java by

adapting existing alias analyses [6, 7] for C or C++. The algorithm is more precise and efficient than previous studies [2, 3, 4, 6, 7] by using the *reference-set* alias representation, the structural traverse of a CFG, and the data propagation rules for the representation. It also does not negatively affect the safety and preciseness of the analysis.

## References


[1] Jehak Woo, Jongwook Woo and Jean-Luc Gaudiot. Flow-Sensitive Alias Analysis with Referred-Set Representation for Java. The Fourth International Conference/Exhibition on High Performance Computing in Asia, May 2000.

[2] H. D. Pande and B. G. Ryder. Static Type Determination and Aliasing for C++. LCSR-TR-236, Rutgers Univ., 1995.

[3] H. D. Pande and B. G. Ryder. Data-flow-based Virtual Function Resolution, Static Analysis: Third International Symposium (SAS'96), LNCS 1145, Sept., 1996.

[4] R. Chatterjee and B. G. Ryder. Scalable, flow-sensitive type inference for statically typed object-oriented languages. Technical Report DCS-TR-326, Rutgers Univ., Aug. 1997.

[5] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects. The 20th ACM SIGACT-SIGPLAN Symposium on POPL, 232-245, January 1993.

[6] Paul Carini and Harini Srinivasan. Flow-Sensitive Type Analysis for C++. Research Report RC 20267, IBM T. J. Watson Research Center, November 1995.

[7] Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural Pointer Alias Analysis. Research Report RC 21055, IBM T. J. Watson Research Center, December 1997.

[8] Barry K. Rosen. Data flow analysis for procedural languages. JACM, 26(2):322-344, April 1979.

[9] M. Emami, R.Ghiya, and L. J. Hendren. Context-sensitive interprocedural point-to analysis in the presence of function pointers. SIGPLAN '94 Conference on Programming Language Design and Implementation, 242-256, 29(6), 1994

[10] S. S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Academic Press, July 1997.

[11] Sun Micro Systems, JavaCC, The parser Generator, http://www.suntest.com/JavaCC/, V0.8pre2, 1998.

[12] Purdue University, West Lafayette, Indiana, USA, Java Tree Builder, http://www.cs.purdue.edu/jtb/index.html, 2000.

[13] Jongwook Woo, Isabelle Attali, Denis Caromel, Jean-Luc Gaudiot, and Andrew L Wendelborn, Alias Analysis On Type Inference For Class Hierarchy In Java, Accepted The 24th ACSC 2001, Jan 29-Feb 2, 2001.

[14] Inria, Sophia, http://www.inria.fr/oasis/ProActive

[15] Java Grande Forum, http://www.javagrande.org/


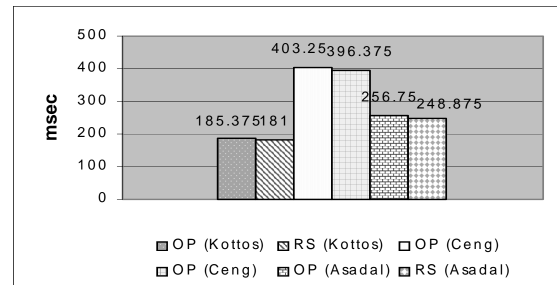


Figure 6. Execution Time of *Dynamic CG*

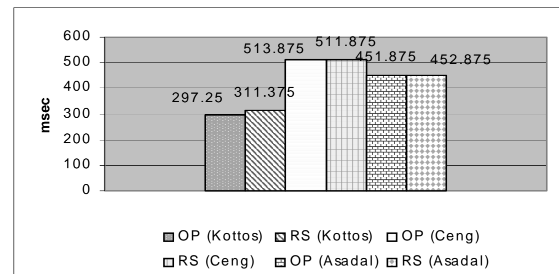


Figure 7. Execution Time of *Ray Tracer*

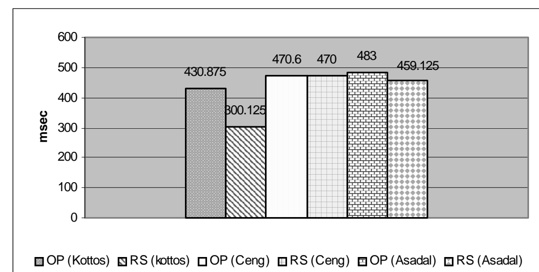


Figure 8. Execution Time of Binary Tree at Depth 1

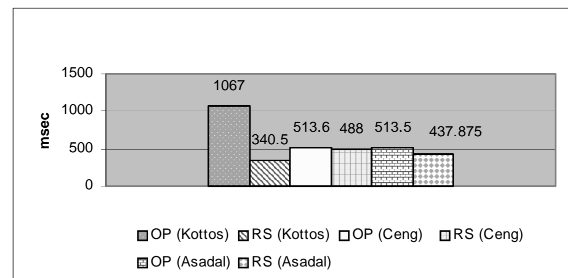


Figure 9. Execution Time of Binary Tree at Depth 2

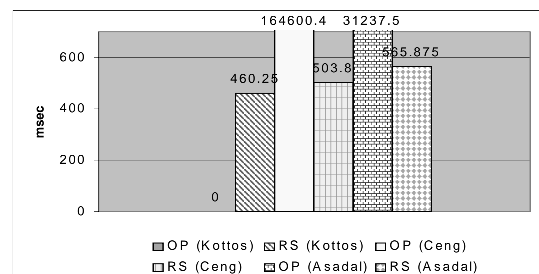


Figure 10. Execution Time of Binary Tree at Depth 6