Alias Analysis for Exceptions in Java

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

Jehak Woo
Esse Soft Inc.
Seoul, 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 alias analysis algorithm* that computes safe and efficient alias sets in Java. For that, we propose a *references-set* representation of aliased elements, its *type table*, and its propagation rules. Also, for an *exception* construct, we consider *try/catch/finally* blocks as well as *potential exception statement* nodes while building a *control flow graph*. Finally, for the safe alias computation on a *control flow graph*, we present a *structural order* traverse of each block and node.

Keywords: Alias Analysis, Reference-Set Representation, Java, Exceptions, High-Performance Computing.

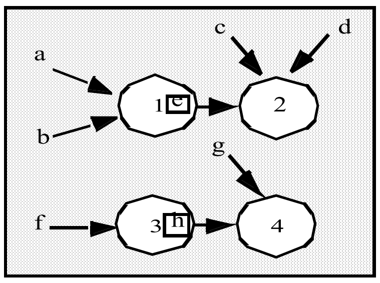
1 Introduction

Java has become a popular language because it is platform independent, object-oriented, and server site applicable. Even though it is easier to analyze Java statically because it does not have pointer operations, Java has other complicated aspects to analyze such as *exception*s and *thread*s. 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. In this paper, we propose an alias analysis algorithm on our *reference-set* representation to compute possible aliases. And, the analysis is applicable to *exception* constructs, one of complicated aspects to analyze.

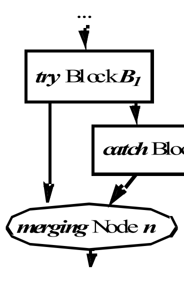
Some research [BCC97, CBC93, CR97, CS95, EGH94, PR94, PR95, WWG00] has analyzed aliases statically for high performance computing such as parallelizing compiler as well as compiler optimization. Those works proposed alias analyses for C/C++ by representing the alias relation with pointers and pointer-to-pointer objects. However, the representation of alias relations is not sufficiently optimized to apply to Java. Thus, we have proposed *referred-set* representation [WWG00] for Java. Even though it reduces the alias computation, it might have a large time complexity in the usage of the computed alias information. Thus, in our previous paper [WWACGW011, WWACGW016], we presented an alternative alias relation, *reference-set* representation to achieve better efficiency and accuracy without losing its safety than the previous works. However, it is not sufficient to analyze exception properties in Java.

People has analyzed *exceptions* statically in Java. Choi [CGH99] presented a model of exceptions by analyzing *exception* instructions in a Java intermediate code. We present a Control Flow Graph (*CFG*) to compute aliases safely on *exception* constructs of source codes in Java. Based on these *CFG*s and propagation rules on *reference-set* representation, we propose a safe and efficient alias analysis algorithm.

In this paper, section 2 presents our reference-set alias representation for Java. Section 3 describes the structures of our algorithm such as *CFG*, *CG*, and type table. Section 4 explains our propagation rules on *reference-set* representation. Section 5 shows our alias analysis algorithm. Section 6 computes the time and space complexities of the algorithm. Finally, our conclusions are presented.



(a) The relationships between references and objects in *try* Block $\mathbf{B_1}$ of (b)



(b) A part of a *CFG*

Figure 1: The relationships between objects in a block of *CFG*

2 Reference-set Representation

An alias analysis algorithm computes the alias sets in a code. 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.

The *reference-set* representation is proposed to improve the accuracy and the efficiency of the alias computation and the type inference for Java.

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 $j \geq 2$ 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 computation in an alias analysis, $j \geq 1$ is allowed when passing references forward and backward at a call site.

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

Initially, we only consider *reference-set*s that contain more than two elements since a *reference-set* with one element is redundant for an alias analysis. For example, in a statement *s* of a code, each *reference-set* and *alias set* for the alias relation in Figure 1 (a) are represented as follows:

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

```
public class ThrowTest {
  public static void main(String args[]){
    int i;

    try { i = Integer.parseInt(args[0]);}
    catch (ArrayIndexOutOfBoundsException e){
      System.out.println("needs an argument!");
      return;
    } catch (NumberFormatException e){
      System.out.println("needs an integer argument!");
      return;
    }

    a(i);
  }

  public static void a(int i){
    try { b(i); }
    catch (MyException e) {
      if (e instanceof MySubException)
          System.out.println("MySubException!");
      else System.out.println("MyException!");


      System.out.println(e.getMessage());
    }
  }

  public static void b(int i) throws MyException {
    int result;
    try {
      System.out.println("i= " + i);
      result = c(i);
      System.out.println("c(i)= " + result);
    } catch (MyOtherException e) {
      System.out.println("MyOtherException:"                    +
e.getMessage());
    }finally{
      System.out.println("\n" + "fin" + "\n");
    }
  }


class MyException extends Exception {
  public MyException() {
            super();
  }
  public MyException(String s) {
           super(s);
  }
}


class MyOtherException extends Exception {
  public MyOtherException() {
           super();
  }
  public  MyOtherException(String s) {
           super(s);
  }
}


class MySubException extends MyException {
  public MySubException() {
            super();
  }
  public  MySubException(String s) {
           super(s);
  }
}
```

Figure 2: An exception handling example code [Flan97]

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

The space complexity of the *reference-set* representation is $O(R_n \times A_r)$, where R_n is the number of objects and A_r is the maximum number of aliased references for an object. It is less than $O(R_t \times A_r)$ of the conventional compact representation, where R_t is the maximum number of

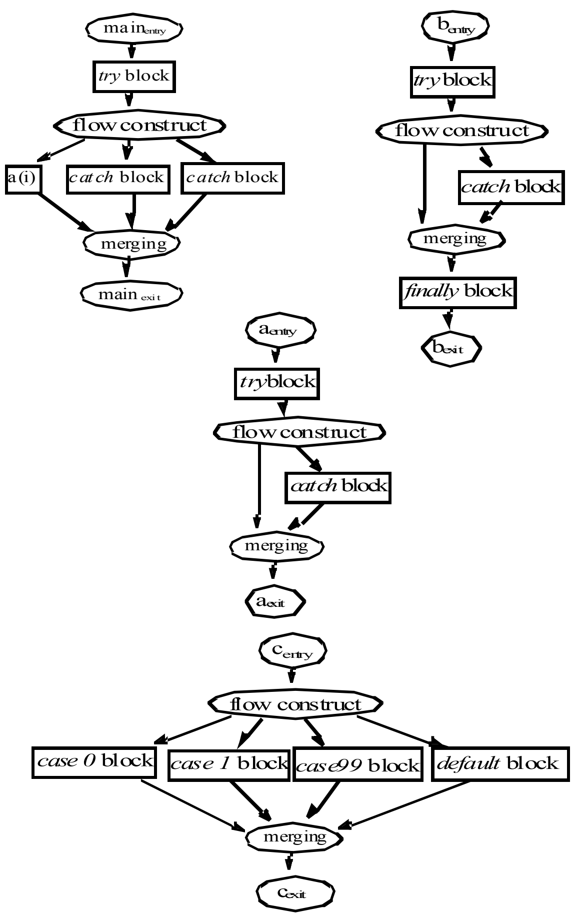


Figure 3: *CFG*s of example classes in Figure 2

objects in the program. Practically, R_n is less than R_t because R_n is initially the number of objects that include more than two references.The time complexity of a alias computation depends on the space complexity of each representation. Thus, the efficiency of whole algorithms is improved via the *reference-set* representation.

3 Data Structures

We can maintain the safety of an alias analysis in Java with the structures such as *CFG, CG, and Type Table*. Java provides an exception handling mechanism with *try/catch/ finally* construct. The *try* block handles its exceptions and abnormal exits with zero or more *catch* blocks. The *catch* clauses catch and handle specified exceptions. The *finally* block should be executed even though an exception is caught or not. A programmer's own exceptions are generated by the *throw* statement. Figure 2 shows example *exception* classes that were written by Flanagan [Flan97].

For the computation of the aliases, *CFG*s can be used for intraprocedural alias analysis. Our *CFG* is a directed graph defined for each method as $<N_{CFG}, E_{CFG}, n_{entry}, n_{exit}, B_{CFG}>$; 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 alias set information between a predecessor and a

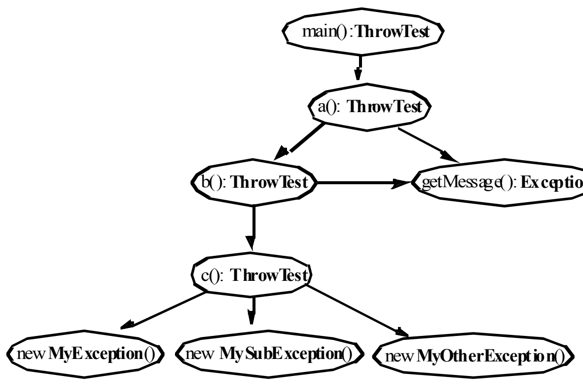


Figure 4: *CG* of example classes in Figure 2

successor statements; n_{entry} and n_{exit} represent the entry and the exit node of the method; B_{CFG} is a set of blocks that consist of the set of nodes for *try, catch,* and *finally* blocks.

We assume that each *potential exception statement* (*PES*) in a *try* block has its corresponding *catch* block for an *exception* construct. An *exception* edge is connected to a *catch* block from the block that contains the exception statement. And, the *catch* block is connected to the *exit* node or a *finally* block.

Also, we need to consider *PES*s of a *CFG*. Runtime *Exceptions* are caused by wrong array indexes, string indexes, and class casts, by qualified expressions of a null pointer, and by dividing-by-zero. n_{exit} of a *CFG* includes an out alias set of the last statement and out alias sets of *PES*s. Figure 3 shows *CFG*s of example classes in Figure 2.

A *CG* is needed for an interprocedural alias analysis between a calling and a called methods at a call statement. Our *CG* is a directed graph defined as $< N_{CG}, E_{CG}, n_{main}>$; 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; 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; n_{main} is the main method that executes initially in a Java program. Figure 4 shows the *CG* of example classes in Figure 2.

A type table contains all possible types of each reference variable. The type table is built during the executing of our algorithm and it contains three columns: a reference variable, its declared type, and its overridden method types. 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. We can improve the efficiency of the analysis with the type table by inferring types of each reference variable in a constant time.

4 Propagation Rules of Alias Information

In a flow-sensitive alias analysis, each statement includes all the alias information at the point. The information is propagated to the next statement and subsequently computed in the *CFG* of a method. This analysis starts with the alias set holding at the entry node n_{entry}, traverses all nodes, and computes the *out* alias set of each node. The analysis ends at the exit node n_{exit}. Let *in(n)* be the input alias set and *out(n)* be the output alias set held on the exit of a node *n*. The effect of a node *n* can be computed by the following equation:

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

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}. In our *reference-set* representation, an operator $\cup$ works within the same *reference-set*s between alias sets as well as the alias sets themselves. For example, when an alias set $A_1 = \{R_1, R_2\}$ where $R_1 = \{a, b\}$ and $R_2 = \{b, c\}$ and an alias set $A_2 = \{R_1, R_3\}$ where $R_1 = \{a, c\}$ and $R_3 = \{b, d\}$, $A = A_1 \cup A_2$ is:

$A = \{R_1, R_2, R_3\}$
where $R_1 = \{a, b, c\}$, $R_2 = \{b, c\}$, $R_3 = \{b, d\}$
because $R_1 = \{a, b\} \cup \{a, c\}$

For an exception block *B* that consists of nodes: n_1, n_2, ..., n_m, we can relate the block and its nodes by the following equation:

$$in(B) = in(n_1)$$
$$out(B) = out(n_m)$$

Also, if a node *n* has a array, a qualified expression, a divide operation, and a class cast expressions, we can consider it as a *PES* node. We describe propagation rules in the following sections according to *CFG* node types.

4.1 Rules for Intraprocedural Analysis

The *intraprocedural analysis* rule consists of premises and conclusions divided by a horizontal line. The premises are a set of equations that define an input alias set, information about a node, and intermediate sets. When all premises hold, the equations in the conclusions are solved for *out(n)*.

First, we define a flow construct node type rule that has several out going edges with the same *out* information:

$in(n) = out(n_{pred})$
n_{pred} *: predecessor node of n or predecessor block of n*
———————————————— *[Flow Construct Node]*
$out(n) = in(n)$

The merging node type rule is as follows:

$in(n) = \bigcup_{p \in n_{pred}} out(p)$
n_{pred} *: predecessor node of n or predecessor block of n*
———————————————— *[Merging Node]*
$out(n) = in(n)$

In the rule, 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.

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

$in(n) = out(n_{pred})$
n_{pred} *: predecessor node of n*
$x = LHS,$
$y = RHS,$
$\forall i, j \quad R_i, R_j \in in(n) \rightarrow [Mod_{kill}(in(n))$
$\quad = \{R_i \mid kill\ x \in R_i\}$
$\quad\quad \cup \{R_i \mid kill\ R_j.f \in R_i \text{ when } q \in R_j \text{ and } x = q.f\}]$
$\quad\quad \wedge [in(n) = in(n) - Mod_{kill}(in(n))] \wedge [KILL(in(n)) = \{x, R_j.f\}],$
$\forall k \quad R_k \in in(n) \rightarrow [Mod_{gen}(in(n))$
$\quad = \{R_k \mid R_k = R_k \cup KILL(in(n)) \text{ when } y \in R_k\}]$
$\quad\quad \wedge [in(n) = in(n) - Mod_{gen}(in(n))],$
n: the first node of an exception block B $\rightarrow in(B) = in(n),$
n is a Potential Exception Statement $\rightarrow PES = PES \cup \{n\}$
———————————————— *[Assignment Node]*
$out(n) = Mod_{kill}(in(n)) \cup Mod_{gen}(in(n)) \cup in(n)$

In this rule, *LHS* and *RHS* respectively stand for the left and the right hand side of an assignment statement. *KILL(in(n))* is a *reference-set* of references killed by Mod_{kill}(in(n)). Also, *out(B) = out(n)* if *n* is the last node of the block *B*. This *out* equality between a block and a node can be applicable for all of the rules in this paper. If the node *n* is one of the *PES*s: an array, a qualified expression, a divide operation, and class cast expressions, the node *n* becomes an element of a set *PES* for all of the rules.

For example, when there is an assignment statement *a.e = f.h* in the *catch* block B_2 of Figure 1 (b), alias set *out(*B_1)

and $in(B_2)$ are expressed with *must alias reference-set* R_1, R_2, R_4 in the *try* block B_1 of Figure 1 (a), (b) as follows:

$$R_1 = \{a, b\}, \quad R_2 = \{R_1.e, c, d\}, \quad R_4 = \{f.h, g\}$$
$$in(B_2) = out(B_1) = \{R_1, R_2, R_4\}$$

Because *LHS* is a qualified expression related to both R_1 and R_2, $Mod_{kill}(in(B_2))$, $in(B_2)$, and $KILL(in(B_2))$ are computed as follows:

$R_1 = \{a, b\}$ and $R_2 = \{R_1.e, c, d\}$ then $R_2 = \{c, d\}$
$Mod_{kill}(in(B_2)) = \{R_2\} \qquad in(B_2) = \{R_1, R_4\}$
$KILL(in(B_2)) = \{R_1.e\}$

Since R_4 includes *RHS*, $Mod_{gen}(in(B_2))$ and $in(B_2)$ are computed as follows:

$$Mod_{gen}(in(B_2)) = \{R_4 \mid R_4 = R_4 \cup \{R_1.e\} = \{R_1.e, f.h, g\}\} = \{R_4\}$$
$$in(B_2) = \{R_1\}$$

Thus, $out(B_2)$ is the union set of $Mod_{kill}(in(B_2))$, $Mod_{gen}(in(B_2))$, and $in(B_2)$ as follows:

$$out(B_2) = Mod_{kill}(in(B_2)) \cup Mod_{gen}(in(B_2)) \cup in(B_2) = \{R_1, R_2, R_4\}$$
$$\text{when } R_1 = \{a, b\}, R_2 = \{c, d\}, R_4 = \{R_1.e, f.h, g\}$$

Finally, $in(n)$ for the merging node n is the union set of $out(B_1)$ and $out(B_2)$ as follows:

$$in(n) = out(B_1) \cup out(B_2) = \{R_1, R_2, R_4\}$$
$$\text{where } R_1 = \{a, b\}, R_2 = \{c, d, R_1.e\}, R_4 = \{f.h, g, R_1.e\}$$

Each *reference-set* of $in(n)$ consists of *may alias* elements; R_1 consists of *must alias* element; R_2 may contain an aliased element R_1.e from the block B_1; R_4 may contain an aliased element R_1.e from the block B_2.

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

$in(n) = out(n_{pred})$
n_{pred} : *predecessor node of n*
M: callee, $LOCAL(M) = \{v \mid v \text{ is a local variable of } M\}$
$\forall i \;\; R_i \in in(n)$ *for r: return reference*
$\rightarrow [Mod_{kill}(in(n)) = \{R_i \mid kill\ x \in R_i\ for\ x \in LOCAL(M)\}]$
$\wedge [R_r = \{R_r \mid kill\ x \in R_r\ for\ x \in LOCAL(M)\ when\ r \in R_r\}]$,
n is a Potential Exception Statement $\rightarrow PES = PES \cup \{n\}$
———————————————————— *[Return Node]*
$out(n) = Mod_{kill}(in(n))$

The next is the rule for an exit node type.

$in(n) = \bigcup_{p \in n_{pred}} out(p)$
n_{pred} : *predecessor node of n*
$PES = \bigcup_{p \in PES} out(p)$
PES: Potential Exception Statement
M: callee, $LOCAL(M) = \{v \mid v \text{ is a local variable of } M\}$
$\forall i \;\; R_i \in in(n) \rightarrow [Mod_{kill}(in(n))$
$= \{R_i \mid kill\ x \in R_i\ for\ x \in LOCAL(M)\}]$
$\wedge [in(n) = in(n) - Mod_{kill}(in(n))]$,
———————————————————— *[Exit Node]*
$out(n) = Mod_{kill}(in(n)) \cup in(n) \cup PES$

4.2 Rules for *Interprocedural* Analysis

We virtually divide a call node into a *precall* 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 and killed in the calling method. It reduces the inefficiency of the previous approaches [PR94, PR95, CR97] which compute redundant alias relations to be modified in a called method.

A *postcall* node collects the modified kill set of the *precall* node and exit nodes alias sets of all possible called methods. By selecting references accessible from the calling method, we can compute the result alias set of the *postcall* node which is the out alias set of the call node as follows:

$in(n) = out(n_{pred})$,
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,
$\forall i$, $R(a_i) \in in(n)$
$\rightarrow [R_{pass}(a_i) = \{a_i, f_i\}] \wedge [R(a_i) = R(a_i) - R_{pass}(a_i)]$,
$RHS = M_c$, $\forall i$, $R(a_i) \in in(n)$, *v is a non local variable in the callee* M_c, $\forall f \forall v\ R(v.f) \in in(n)$
$\rightarrow [R(v) = R(v) - \{v\}] \wedge [R(v.f) = R(v.f) - \{v.f\}]$
$\wedge [R_{pass}(v) = \{v\}] \wedge [R_{pass}(v.f) = \{v.f\}]$
$\wedge [PASS(M_c) = \cup \{R_{pass}(a_i), R_{pass}(v), R_{pass}(v.f)\}]$,
$RHS = E_c.M_c$, $\forall i$, $R(a_i) \in in(n)$, $\forall f\ \forall a_i$, $R(a_i.f) \in in(n)$,
$\forall f$, $R(E_c.f) \in in(n)$

$$\begin{aligned}
&\rightarrow [R(E_c.f) = R(E_c.f) - \{E_c.f\}] \wedge [R_{pass}(E_c.f) = \{E_c.f\}]\\
&\quad \wedge [R(a_i.f) = R(a_i.f) - \{a_i.f\}]\\
&\quad \wedge [R_{pass}(a_i.f) = \{a_i.f\}]\\
&\quad \wedge [PASS(M_c) = \cup \{R_{pass}(a_i), R_{pass}(a_i.f), R_{pass}(E_c.f)\}],\\
&n\text{: the first node of an exception block } B \rightarrow in(B) = in(n)\\
&\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \text{ [Precall Node]}\\
&out(n) = in(n)
\end{aligned}$$

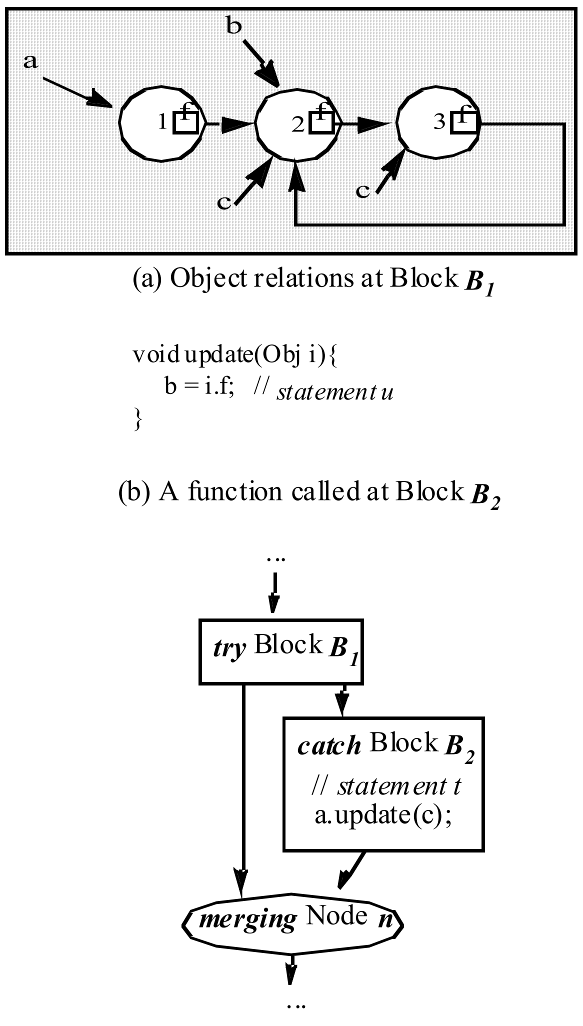
PASS(M_c) represents the set of actual, formal parameters and non-local variables in a called method M_c. $R_{pass}(a_i)$ is a set of actual parameters accessible by a called method when passing from a caller to the called method M_c. $R_{pass}(v)$ is a set of non-local variables accessible by a called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

PRECALL(M_c) is a *precall* node of call statement nodes that invoke this called method. An entry node merges alias sets from the *precall* nodes and then propagates the merged set to its subsequent node.

$$\begin{aligned}
&PRECALL(M_c)\text{: a precall node of the callee } M_c,\\
&in(n) = \bigcup_{p \in PRECALL(M_c)} PASS(p),\\
&n\text{: the first node of an exception block } B \rightarrow in(B) = in(n)\\
&\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \text{ [Entry Node]}\\
&out(n) = in(n)
\end{aligned}$$

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

$$\begin{aligned}
&in(n) = \bigcup_{p \in n_{precall}} out(n_{precall})\\
&{}_{nprecall}\text{: a precall node of } n\\
&RHS = E_c.M_c\\
&\rightarrow FIELD(E_c) =\\
&\qquad\qquad \{f \mid f \text{ is a field name in an object referred by } E_c\},\\
&RHS = M_c \rightarrow FIELD(E_c) = \varnothing,\\
&RHS = new\ M_c \rightarrow FIELD(E_c) = \varnothing \wedge A(r),\\
&EXIT(M_c)\\
&= \{e \mid e \text{ is an exit alias set from a possible callee method } M_c\},\\
&LHS = \varnothing, \forall R_{passb} \in EXIT(M_c)\\
&\rightarrow [R_{passb} = R_{passb} - \{v \mid v \text{ is a local variable in the callee } M_c\}]\\
&\quad \wedge [EXIT(M_c) = \bigcup_{\text{for all } M_c} R_{passb}],\\
&LHS \neq \varnothing, \forall R_{passb} \in out(\text{return node})\\
&\rightarrow [R_{passb} = R_{passb} - \{v \mid v \text{ is a local variable in the callee } M_c\}]\\
&\quad \wedge [EXIT(M_c) = \bigcup_{\text{for all } M_c} R_{passb}],\\
&\forall i\ R_i \in EXIT(M_c),\ \forall j\ R_j \in in(n)\\
&\rightarrow [R_i \mid R_i = R_i \cup R_j \text{ when } i{=}j] \wedge [EXIT(M_c) = EXIT(M_c) - R_i]\\
&\quad \wedge [in(n) = in(n) - R_j],\\
&exit(RHS) =\\
&\quad \bigcup_{e \in EXIT(M_c)} out(e) \cup \bigcup_{p \in n_{precall}} \cup\, out(\text{precall node}) \cup \bigcup_{\text{for all } i} R_i,
\end{aligned}$$



(a) Object relations at Block B_1

```
void update(Obj i){
    b = i.f;  // statement u
}
```

(b) A function called at Block B_2

```
catch Block B2
// statement t
a.update(c);
```

(c) *CFG* of an example

Figure 5: Example of an interprocedural Analysis

$$\begin{aligned}
&LHS = \varnothing \rightarrow out = exit(RHS),\\
&LHS = x, \forall i, j\ \ R_i, R_j \in exit(RHS), R(RHS) \in exit(RHS)\\
&\quad \rightarrow [Mod_{kill}(exit(RHS))\\
&= \{R_i \mid kill\ x \in R_i\} \cup \{R_i \mid kill\ R_j.f \in R_i \text{ when } q \in R_j \text{ and } x = q.f\}]\\
&\qquad \wedge [exit(RHS) = exit(RHS) - Mod_{kill}(exit(RHS))]\\
&\qquad \wedge [KILL(exit(RHS)) = \{x, R_j.f\}],\\
&R(RHS) \in exit(RHS) \rightarrow [Mod_{gen}(exit(RHS)) = \{R(RHS)\\
&\quad \mid R(RHS) = R(RHS) \cup KILL(exit(RHS))\}]\\
&\quad \wedge [exit(RHS) = exit(RHS) - Mod_{gen}(exit(RHS))]\\
&\quad \wedge [out = Mod_{kill}(exit(RHS))\\
&\qquad\quad \cup Mod_{gen}(exit(RHS)) \cup exit(RHS)],\\
&n \text{ is a Potential Exception Statement} \rightarrow PES = PES \cup \{n\}\\
&\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \text{ [Postcall Node]}\\
&out(n) = out
\end{aligned}$$

exit(RHS) is a set of exit nodes of all possible called methods. We can compute *exit(RHS)* in a *CG* by integrating all *out(precall node)* and outgoing edges from callers and their exit nodes.

Figure 5 is the example where to compute an alias set on the interprocedural analysis rule. If we assume that Figure 5 (a) represents the state at *try* block B_1, the *out* alias set of the block B_1 is:

$out(B_1) = \{R_2, R_3\}$
where $R_2 = \{a.f, b, c, R_3.f\}$ and $R_3 = \{R_2.f, c\}$

After executing the call statement t at the block B_2 in Figure 5 (c), the alias set of its *precall node* becomes as follows:

$in(t) = in(B_2) = \{R_2, R_3\}$,
$RHS = a.update(c)$,
$a_i = c, f_i = i$,
$R_{pass}(a_i) = \{c, i\}$, $R(a_i) = R_2 = R_2 - \{c\}$
$= \{a.f, b, R_3.f\}$
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 $R_{pass}(a.f) = \{a.f\}$,
$PASS(a.update) = \{R_{pass}(a_i), R_{pass}(a.f)\}$,
$out(t_{precall}) = \{R_2, R_3\}$

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:

$in(t_{postcall}) = out(t_{precall}) = \{R_2, R_3\}$
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,
$EXIT(update) = \{R(b), R_{passb}(R_2), R_{passb}(R_3)\}$,
$R_2 = R_2 \cup R_{passb} \cup R_{passb}(R_2) = \{b, R_3.f, c.f, c, a.f\}$,

```
Algorithm Alias Analysis
construct an initial $CG$ with main method;
**repeat** {
  **for** each method $T.M \in N_{cg}$,
  alternating between topological and reverse topological order
   **for** each node $n \in N_{cfg\,(T.M)}$ 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 $t \in TYPES$ {
       **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 6: Alias Analysis Algorithm

$R_3 = R_3 \cup R_{passb} \cup R_{passb}(R_3) = \{R_2.f, b, c.f, c\}$,
Thus, $out(B_2) = out(t) = out(t_{postcall}) = \{R_2, R_3\}$
Finally, $in(n) = out(B_1) \cup out(B_2) = \{R_2, R_3\}$
where $R_2 = \{a.f, b, c, R_3.f\}$ and $R_3 = \{R_2.f, b, c\}$

5 Alias Analysis Algorithm

Our alias analysis algorithm in Figure 6 visit all nodes of a *CG* until fixed data status and nodes are achieved. The algorithm traverses each node of a *CG* in a topological and a reverse topological order in order to possibly shorten the execution time for the fixed point [CBC93, CS95, BCC97]. 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. While processing the algorithm, resolved methods with the possible class types of each reference make the *CG* grow.

Each node in our algorithm is visited in structural order; while visiting nodes from an entry node to an exit node, for the *if* flow construct node, each branch is traversed then finally its merging node is visited; for the *exception* blocks, each block is traversed then finally its merging node is visited. With the structural order, we do not only maintain the safety of the alias computation, including *exception* constructs, but also we improve the efficiency in Java than the previous work [CS95] without losing the accuracy of a resulting set.

6 Complexity of the Algorithm

For the most outer loop, R_n and A_r are the number of reference-sets and the maximum number of aliased reference variables for each *reference-set*. $R_n \times A_r$ means the maximum number of refer-to relations between references and objects existing in each node of a *CFG* except its exit node. We can estimate the worst time complexity of the loop as $O(R_n \times A_r \times N_{pes} \times E_{cg})$; N_{pes} is the number of *potential exception statement*s in a *CFG*; $R_n \times A_r \times N_{pes}$ denotes the maximum number of refer-to relations between references and objects existing in exit node of a *CFG*; E_{cg} is the number of edges in a *CG* since each relation from an entry node to an exit node is traversed once per each iteration. For the second out loop, the time complexity becomes $O(N_{cg})$ if N_{cg} is the final number of nodes in a *CG*. For the inner most 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 most dominant parts of the execution in an inner loop are the call statement nodes so that the worst time complexity depends on the number of call statements. The time complexity of a set of inferred types is $O(R_m)$; R_m is the number of reference variables in a program code and a type table contains all possible types of an reference variable.

The time complexity for the possible method resolution is $O(T_i \times H)$; 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 $O(R_p \times R)$; R_p is the maximum number of *reference-set*s propagated; 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 $O(R_n \times A_r \times N_{pes} \times E_{cg} \times N_{cg} \times N_{cfg} \times (R_p \times R \times R_m + T_i \times (H + N_{cg} + C_c)))$. The worst space complexity becomes $O(R_n \times A_r \times N_{pes} \times N_{cg} \times N_{cfg})$ to include an alias set and $O(R_m)$ for a type table. The worst space complexity of the outgoing edges for a call statement is $O(T_i \times H)$ and then the worst space complexity of a *CG* is $O(N_{cg} \times C_s \times T_i \times H)$; C_s is the maximum number of call statements in a method.

Existing alias relations for C++ generates the number of aliased elements in an exit node as $((O \times A_o + O) \times N_{pes})$ for Java; O is the number of objects in a program; A_o is the maximum number of aliased element for an object. For the most outer loop with the same iterative algorithm as in Figure 6, we can estimate the worst time complexity as $O((O \times A_o + O) \times N_{pes} \times E_{cg})$. The number of columns of a type table are the number of object names O. The time complexity of a set of inferred types is $O((O \times A_o + O) + O)$; $O((O \times A_o + O))$ is the time to search the aliased elements; $O(O)$ is the time to search the type table for all possible types of an object name. The worst time complexity of a call statement node is $O((O \times A_o + O) \times N_{pes})$ when a caller propagates an alias set to both the callee and next node. Therefore, the worst time complexity of the existing works [CBC93, CS95, CS97] is $O((O \times A_o + O) \times N_{pes} \times E_{cg} \times N_{cg} \times N_{cfg} \times (O \times A_o + O) \times (O \times A_o + O) + O) + T_i \times (H + N_{cg} + C_c)))$.

Practically, R_n is much less than O even though A_r equals to A_o so that our $O(R_n \times A_r)$ is less than $O(O \times A_o + O)$. For the type inference, our constant time complexity $O(R_m)$ is less than the time complexity of $O((O \times A_o + O) + O)$. For the call statement, our $O(R_p \times R)$ is bigger than the $O(O \times A_o + O)$ but it reduces the redundant aliased elements of the caller. As a result, our worst time complexity of the main algorithm is less than the existing works [CBC93, CS95, BCC97].

7 Conclusion

We have presented a *reference-set* alias representation for Java since existing alias representations of C/C++ are not efficient for Java. We also have presented *CFG*s of *exception* constructs with *PES*s. Then, we have proposed our *flow sensitive alias analysis algorithm* by adapting existing alias analyses [CS95, BCC97] for C/C++ to for Java. The algorithm is more precise and efficient than previous works [BCC97, CBC93, CR97, CS95, EGH94, PR94, PR95, WWG00] based on the *reference-set* alias representation and its data propagation rules. Besides, the algorithm is the safe alias analysis for *exception* statements.

In the complexity analysis, we have shown that our *reference-set* alias representation is more precise and efficient for a type inference and data propagation rules that are main parts of the alias analysis algorithm. We have shown that, by using structural traverse of a *CFG*, our algorithm becomes more efficient than the previous works.

8 References

[BCC97] M. Burke, P. Carini, and J. Choi. Interprocedural Pointer Alias Analysis. Research Report RC 21055, IBM T. J. Watson Research Center, December 1997.

[CBC93] J. Choi, M. Burke, and P. Carini. Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects. The 20th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 232-245, January 1993.

[CGHS99] J. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs. Proceedings of the ACM SIGPLAN SIGSOFT workshop on Program analysis for software tools and engineering September 6, 1999.

[CR97] R. Chatterjee and B. G. Ryder. Scalable, flow-sensitive type inference for statically typed object-oriented languages. Technical Report DCS-TR-326, Rutgers University, August 1997.

[CS95] P. Carini and H. Srinivasan. Flow-Sensitive Type Analysis for C++. Research Report RC 20267, IBM T. J. Watson Research Center, November 1995.

[EGH94] 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, SIGPLAN Notices, 29(6), 1994

[Flan97] David Flanagan, Java in a nutshell, 2nd Edition, O'REILLY, May 1997.

[Much97] S. S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Academic Press, July 1997.

[PR94] H. D. Pande and B. G. Ryder. Static Type Determination and Aliasing for C++. Technical Report LCSR-TR-236, Rutgers University, December 1994.

[PR95] H. D. Pande and B. G. Ryder. Static Type Determination and Aliasing for C++. Technical Report LCSR-TR-250-A, Rutgers University, October 1995.

[WWACGW016] Jongwook Woo, Jehak Woo, Isabelle Attali, Denis Caromel, Jean-Luc Gaudiot, and Andrew L Wendelborn. Alias Analysis On Type Inference In Class Hierarchy In Java. In proceedings of Twenty-Fourth Australasian Computer Science Conference, Jan 2001.

[WWACGW011] Jongwook Woo, Jehak Woo, Isabelle Attali, Denis Caromel, Jean-Luc Gaudiot, and Andrew L Wendelborn. Alias Analysis for Java with Referrence-Set Representation. In proceedings of Eighth International Conference on Parallel and Distributed Systems, June 2001.

[WWG00] 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 Pacific Region, May 2000.