International Journal of Parallel Programming, Vol. 32, No. 1, February 2004 (© 2004)

Alias Analysis in Java with Reference-Set Representation for High-Performance Computing

Jongwook Woo,[1] Jean-Luc Gaudiot,[2] and Andrew L Wendelborn[3]

Received December 11, 2002; revised October 1, 2003; accepted October 21, 2003

In this paper, a *flow-sensitive, context-insensitive* alias analysis in Java is proposed. It is more efficient and precise than previous analyses for C++, and it does not negatively affect the safety of aliased references. To this end, we first present a *reference-set* alias representation. Second, data-flow equations based on the propagation rules for the *reference-set* alias representation are introduced. The equations compute alias information more efficiently and precisely than previous analyses for C++. Third, for the constant time complexity of the type determination, a type table is introduced with reference variables and all possible types for each reference variable. Fourth, an alias analysis algorithm is proposed, which uses a popular iterative loop method for an alias analysis. Finally, running times of benchmark codes are compared for *reference-set* and existing *object-pair* representation.

KEY WORDS: Alias analysis; high-performance computing; reference-set representation; Java.

1. INTRODUCTION

An alias is defined as two or more reference (pointer) variables that may point to the same memory location. Aliases can be created by assignment

[1] Computer Information Systems Department, California State University, Los Angeles, California 90032. E-mail: jwoo5@calstatela.edu

[2] Department of EECS, University of California, Irvine, California 92697. E-mail: gaudiot@uci.edu

[3] Department of Computer Science, University of Adelaide, South Australia 5005, Australia. E-mail: andrew@cs.adelaide.edu.au

statements among reference variables, and by calling statements among procedures on global variables and parameters (call-by-reference) of reference variables. Aliases complicate the data-flow analysis which is performed during optimization and parallelization of a part of program. It causes side effects that make the code unsuitable for parallel execution.

As one optimization technique used in the static analysis of compilers, alias analysis has been developed to detect aliased variables and to reorder instructions in order to enhance performance. Aliases are detected on *assignment statements* through *intra-procedural* analysis within a procedure by using its control flow graph (CFG) and on *call statements* through *inter-procedural* analysis among procedures by using its calling graph (CG). There are the *context-sensitive* and *context-insensitive* approaches to building a *CG* for *inter-procedural* analysis. The *context-sensitive* approach is characterized by a data flow analysis based on path-sensitivity, so each procedure may be analyzed separately for different calling contexts. In the *context-insensitive CG*, each procedure is represented by a single node in a calling graph, so information is computed efficiently with a context-insensitive graph, but the computed data are approximate. In object-oriented languages such as C++ and Java, the static determination of run-time object types is a critical factor in the optimization. The static determination of run-time types makes inlining/cloning optimizations possible by predicting dynamic function calls. It is used to resolve a number of indirect function calls by limiting the number of possible functions invoked and by converting the indirect function calls to direct function calls. Also, it improves the precision of *inter-procedural* analyses and transformations. We can define "precision" as a measure of the extent to which an analysis accurately reports aliases; as stated by Hind: "a less precise alias analysis will conservatively report more alias relations representing real memory locations than a more precise analysis."(31) We also can define the "safety" as a measure of the extent to which an analysis correctly reports aliases, in that a less safe alias analysis will report a smaller subset of all the actual alias relations representing real memory locations, than will be reported by a more safe analysis. In addition, in case of alias analysis, the static determination of run-time object types are needed to build safe calling graphs for function calls including virtual function calls.

The integration of alias analysis with type information increases the precision of alias detections, particularly with regard to inheritance among classes. Although the integration of alias analysis with type information(15, 19, 24, 39) improves the precision and efficiency of alias analysis for virtual functions in C++ (overridden methods in Java), these approaches do not achieve the desired precision in Java because their alias representations are for pointer and object based languages. Also, for constructors and class inheritances, the type information may lose information regarding shadowed variables in Java: shadowed variables are variables defined in one class with the same named variable of its super class; overridden method is a method defined in one class with same named method and same argument types of its super class. Further, their type inference methods may not be safe for dynamic type determination(39) and the indirect type inference with objects may negatively affect performance.(15, 19, 39)

In this paper, we present a compile-time, *flow-sensitive, context-insensitive*, alias analysis algorithm with type information in Java. It addresses those issues in C++ while applying to Java. Our alias analysis algorithm adapts to Java an existing alias analysis algorithm for C++(15) by adding our type inference algorithm and data flow equations for our *reference-set* representation. In this scheme, the type information of references during alias analysis is inferred by using our type inference operation. The inferred type information is used to increase the precision of subsequent alias analyses not only for overridden methods but also for shadowed variables. The inference is much more efficient because a reference variable has all possible type information. Also, our algorithm proposes a data flow computing rule for possible statements by regarding constructors as functions. Compared to other algorithms,(15, 19, 24, 39) the precision of our analysis algorithm is improved by adding type information of shadowed variables and by regarding constructors as functions. Also, it computes safe aliases and improves efficiency by using *reference-set* representation. Further, the equations can compute aliases in *exception* statements.

Section 2 presents related work for alias analysis in C/C++. Section 3 shows motivation of our alias analysis in Java. Section 4 describes the differences between C++ and Java for alias computing. Section 5 introduces our *reference-set* alias representation for Java and describes the structure of our algorithm. Section 6 explains our propagation rules on the *reference-set* representation. Section 7 shows our alias analysis algorithm and computes the time and space complexities of the algorithm. Section 8 shows the experimental results. Finally, the conclusions are presented.

2. RELATED WORK

The computation of pointer aliases has been studied in several publications(5, 6, 10, 15, 19, 26, 35, 39) in the context of C and C++. C is a procedural language where a memory is handled by using pointer variables while C++ is an extension of C to object-oriented concepts, although it is not a pure object-oriented language. Since the existence of pointers implies many hidden aliases, earlier works(5, 6, 10, 15, 19, 26, 35, 39) have investigated the computation of pointer induced aliases. They focused on how to improve efficiency, preciseness, and safety of analyses and demonstrated that these goals are often contradictory: one must be traded off against the other. Thus, they each have their own alias-set representations and type inference methods.

Java has a syntax similar to C and C++, so we might hope that alias analysis in Java could be an extension of existing alias analyses in C/C++. Unfortunately, since Java is a pure object-oriented language with several properties including object references, the ability to invoke methods of other classes, late dynamic binding, class hierarchies, exceptions and multithreading, conventional studies(5, 6, 10, 15, 19, 26, 35, 39) are not sufficient solutions to compute aliases among objects in Java. Nonetheless, conventional *inter-procedural* analyses give useful backgrounds to the problem of alias analysis in Java.

2.1. Alias-Set Representation

This section introduces how alias representation for alias analysis has been developed from C and C++. Pande(39) presented the first algorithm which simultaneously solved type determinations and pointer aliases with *points-to* alias set representation in C++ programs. *Points-to* has the form $\langle loc, obj \rangle$ where *obj* is an object and *loc* is a memory location of the object *obj*. *Points-to* pair is essentially *points-to* relation as introduced by Emami.(26) Emami proposed it to reduce extraneous alias pairs generated in certain cases with *alias pairs* of Landi.(35)

Carini(15) proposed a *flow-sensitive* alias analysis in C++ with *compact representation*. The *compact representation* is an alias relation that has a name object or one level of dereferencing. The *compact representation* of alias relation was introduced by Choi(5, 6, 10) to eliminate redundant *alias pairs*.

Chatterjee(19) presented a *flow-sensitive* alias analysis in object-oriented languages with *points-to* alias set representation in C++. It improves the efficiency and safety of *points-to* alias set representation comparing to Pande's method.

The *compact* and *points-to* alias representation are highly similar. However, the *points-to* alias representation contains *may* or *must* alias information.(5, 6)

Woo(49) introduced a *flow-sensitive* alias analysis in Java with *referred-set* alias representation, which is an alternate formulation to this paper. *Referred-set* is a set of objects that may be pointed by a reference variable and an alias set is a collection of *referred-set*. It is used to reduce extraneous alias pairs while applying the *compact* and *points-to* alias representations in C/C++ to Java.

For example, in C++ (see (a) in Fig. 1), *Alias pairs* can be generated as $\langle {}^*x, y\rangle$, $\langle {}^{**}x, {}^*y\rangle$, $\langle {}^*y, z\rangle$, $\langle {}^{**}x, z\rangle$. The corresponding compact representations are $\langle {}^*x, y\rangle$, $\langle {}^*y, z\rangle$. *Points-to* pairs can be represented as $\langle x, y, D\rangle$, $\langle y, z, D\rangle$ where D represents the pair as *must* alias relation. *Referred-set* alias relation is $A(statement\,2) = null$ because each variable does not point to any object.

In Java (see (b) in Fig. 1), *Alias pairs* can be generated as $\langle z, obj\_A\rangle$, $\langle y, z\rangle$, $\langle x, y\rangle$, $\langle y, obj\_A\rangle$, $\langle x, obj\_A\rangle$ where obj_A is an object generated by statement 1. The *compact representations* of the statement are $\langle z, obj\_A\rangle$, $\langle y, z\rangle$, $\langle x, y\rangle$. *Points-to* pairs can be represented as $\langle z, obj\_A\rangle$, $\langle y, z, D\rangle$, $\langle x, y, D\rangle$ where D represents the pair as *must* alias relation. *Referred-set* alias relations are $A_z = \langle obj\_A\rangle$, $A_y = \langle obj\_A\rangle$, $A_x = \langle obj\_A\rangle$, $A(statement\,3) = \{A_z, A_y, A_x\}$.

Existing alias relations in C++, *compact* and *points-to* representations, are quite similar, especially when applied to Java, because it represents an alias as a pair of $\langle$*object name*, *object name*$\rangle$. *Object name* is an object generated or a reference variable that points to the object. Therefore, in this paper, we call *compact* and *points-to* representations as *object-pair* representation. The space complexity of the *object-pair* representation is $O(N_r + N_o \times nC_2)$, where N_r is the number of reference variables, N_o is the number of objects, n is the number of reference variables that refer to an object, and nC_2 is to generate the *combination* set with 2 elements among n. The space complexity of *Referred-set* alias relations is $O(N_r + N_o \times A_o)$, where N_r is the number of reference variables, N_o is the number of objects, and A_o is the maximum number of references aliased for an object. If n becomes larger, nC_2 will be larger than A_o. Thus, $O(N_r + N_o \times nC_2) > O(N_r + N_o \times A_o)$. Therefore, comparing to *object-pairs*, the collection of objects for a reference variable saves space. Further, it may improve performance in terms of the time complexity to compute alias set.

```
func() {
  A **x, *y, z;

  x = &y;  //statement 1
  y = &z;  //statement 2
}
```

(a) A Code in C++

```
bar() {
  A x, y, z;

  z = new A(); //statement 1
  y = z; //statement 2
  x = y; //statement 3
}
```

(b) A Code in Java

Fig. 1. Example programs for related work.

2.2. Type Inference for Objects

As an object-oriented language, C++ and Java often make use of inheritances, which leads to dynamic invocation. Dynamic invocation, such as through virtual functions, makes it hard to build a calling graph statically. Type inference statically computes possible run-time types for every expression in the program's source code. The static determination of possible run-time types is a key issue for compile-time optimization because the dynamic calls of an object-oriented language make it hard to build its calling graph statically.

Therefore, there are advantages of static type determinations. First, the inlining or the cloning optimization mechanisms of functions are applicable if the static type determination resolves a number of indirect function calls by limiting the number of possible functions invoked and by converting the indirect function calls to direct function calls. Second, the static type determination improves the precision of *inter-procedural* analyses and transformations in object-oriented language. Finally, the static type determination has solved problems upon the occurrence of virtual function calls for compile-time optimization. For alias analysis in object-oriented languages, the second and third benefits will improve the precision or, at least, maintain the safety of the analysis by recognizing the dynamic calls at compile time.

Pande(39) integrates alias analysis and dynamic type information in C++. His *points-to* alias relations contain type information for each object. However, his method loses safety of alias relations because it is not clear how to maintain type information for conditional statements.

Carini(15) proposed a type table to infer possible types of an object in C++. The type table is a pair of $\langle object, type \rangle$. We can statically infer possible types of an object with its alias relations and the type table. Its space complexity with *compact* alias representation is $O(N_o + N_r)$, where N_o is the number of objects and N_r is the number of references. Its time complexity with *compact* alias representation is $O((N_o + N_r) \times A_r)$, where A_r is the maximum number of references aliased for an object.

Chatterjee(19) computed a type of pointer variables with referenced and modified type relations and alias relations at each program point in C++. Its space complexity is the same as the compact representation. Its *points-to* relation contains alias pairs and their type information. Further, it maintains the safety of the possible types for conditional statements by adding *may* or *must* flag. The time complexity is the same as that of compact alias representation.

The time complexity of the *Referred-set*(49) alias relation for type inference is $O(R \times A_o)$, where R is the maximum number of accessible references at a program point and A_o is the maximum number of objects pointed by a reference. A type table is defined with a pair of an object and its type. In the table, we can search for each object to collect possible types of a reference because a *referred-set* contains all possible objects to which the reference points. Since, R is usually smaller than $N_o + N_r$, the *referred-set* alias relation will be more efficient than the others.

3. MOTIVATION

There are three motivations for developing an alias detecting algorithm in Java. First, aliases causes problems not only for optimizing a sequential compiler but also for a parallel compiler when using Java. Second, some mechanisms used in *inter-procedural* analysis cannot be adequately applied to object-oriented languages because procedural languages do not have objects. Third, conventional alias analysis with type information in C++ needs to be adapted for an alias representation, additional type information, and rules including exceptions when applied to Java. Those motivations are more specifically described as follows.

High performance computing on distributed computing systems has been widely studied.(12, 18) It has been shown that high performance computation can be achieved without any extra cost, using existing computers that are connected via some network to execute some applications concurrently as in distributed computing systems.

Since Java is a platform independent language, it can be used for integrating different computational platforms into a distributed computing system at no additional cost. Java is an object-oriented language which allows for easy maintenance, update and reuse of application programs. An application program can be implemented for applets or servlets that are executed on the World-Wide-Web. Thus, we can create a distributed or clustering computing system which is more scalable and allows accesses to arbitrary hosts with the owner's permission.

These properties are making Java a major interesting parallel language for distributed computing application. However, Java is an object-oriented language with reference variables which refer to objects instantiated. Reference variables become possible aliases if a pair of references are left-hand and right-hand side references on an assignment statement. Those aliased references may cause race conditions if each reference is assigned to a different process. Further, it may cause context switch and communication delay. Researchers have studied to avoid or detect aliases among pointers in C and C++.(5, 6, 10, 15, 19, 26, 35, 39) Since reference implies pointers or pointed objects, we could refer to those studies for Java in order to detect aliases among reference variables.

4. 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 (pointer variables) have their own names for an alias analysis. A pointer variable name is a name which points to an object that contains the address of a pointed-to object. In Fig. 2(a), the pointer variable name p is naming a pointer-valued object that contains its address value. The 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 works,(6, 10, 20) when pointer p points to an object of v, the alias relation is represented as $\langle {}^*p, v\rangle$.

Figure 2(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 nonpointer object. Those alias relations are represented as $\langle {}^*p, q\rangle$ and $\langle {}^*q, v\rangle$. Existing alias relations in C++ are similar, which are compact(6, 10, 20) and *points-to*(19, 39) representation. Those relations save space by representing all alias relations without using an exhaustive set. Those relations can be used in Java. However, there are some problems in the use of those representations because only references are used to name objects in Java. A reference is a variable that refers to 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 and that is only refered to by a reference variable. Thus, each object needs its own naming such as *obj1* and *obj2* by binding a reference name and an object name for the existing alias relations. However, we have another more efficient binding.

In Fig. 3(a), if there is an assignment statement $r = \&w$, the value of the addressed valued object named by r is changed to the value of the addressed valued object bound by w. Therefore, $\langle {}^*r, v\rangle$ can be killed and a new alias relation $\langle r, {}^*w\rangle$ is generated through the compact representation rule(5, 6, 10) as follows.

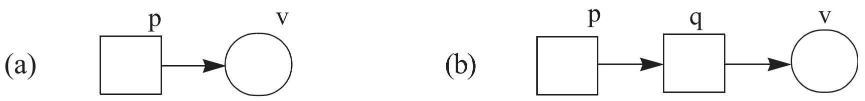


Fig. 2. Relation between a pointer and an object in C++.

Object-pairs A_{IN} before $r = \&w$:

$$A_{IN} = \{\langle {}^*p, r\rangle, \langle {}^*q, r\rangle, \langle {}^*r, v\rangle\}$$

Object-pairs after $r = \&w$:

$$Kill{:}A_{IN}({}^*r) = \{\langle {}^*r, v\rangle\}$$

$$Gen{:} \bigcup_{\langle r, u\rangle \in A_{IN},\, AR \in A_{IN}({}^*w)} \{AR({}^*u/{}^*w)\} = \{\langle r, r\rangle\} \otimes \{\langle {}^*w, {}^*w\rangle\}$$

$$= \{\langle r, {}^*w\rangle\}$$

finally,

$$\begin{aligned}\text{object-pairs } A_{OUT} &= (A_{IN} - Kill) \cup Gen\\ &= (\{\langle {}^*p, r\rangle, \langle {}^*q, r\rangle, \langle {}^*r, v\rangle\} - \{\langle {}^*r, v\rangle\})\\ &\quad \cup \{\langle r, {}^*w\rangle\}\\ &= (\{\langle {}^*p, r\rangle, \langle {}^*q, r\rangle, \langle r, {}^*w\rangle\}\end{aligned}$$

In Fig. 3(b), an alias relation via compact representation in Java is shown. If there is an assignment statement $p.d = w$ when the variable w refers to an object *obj3*, we can compute alias relations with the compact representation rule as follows:

Object-pairs A_{IN} before $p.d = w$:

$$A_{IN} = \{\langle p, r\rangle, \langle q, r\rangle, \langle p.d, v\rangle, \langle q.d, v\rangle, \langle w, obj3\rangle\}$$

Object-pairs after $p.d = w$:

$$Kill{:}A_{IN}(p.d) = \{\langle p.d, v\rangle\}$$

$$Gen{:} \bigcup_{\langle p.d, u\rangle \in A_{IN},\, AR \in A_{IN}(w)} \{AR(u/w)\} = \{\langle p.d, p.d\rangle\} \otimes \{\langle w, w\rangle\}$$

$$= \{\langle p.d, w\rangle\}$$

finally,

$$\begin{aligned}&\text{object-pairs } A_{OUT} = (A_{IN} - Kill) \cup Gen\\ &= (\{\langle p, r\rangle, \langle q, r\rangle, \langle p.d, v\rangle, \langle q.d, v\rangle, \langle w, obj3\rangle\} - \{\langle p.d, v\rangle\})\\ &\quad \cup \{\langle p.d, w\rangle\}\\ &= (\{\langle p, r\rangle, \langle q, r\rangle, \langle q.d, v\rangle, \langle w, obj3\rangle, \langle p.d, w\rangle\}\end{aligned}$$

However, for the correct relation, $\langle q.d, v\rangle$ of A_{OUT} should be killed. The incorrect result comes from the fact that, in Java, a reference name is used for naming an object without a dereferencing operator such as * in

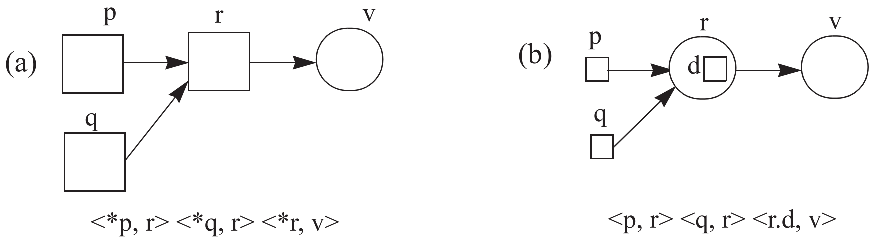


Fig. 3. Difference between a pointer in C++ and a reference in Java.

C++. Therefore, we believe that the traditional rule is not applicable to Java to detect precise alias relations as well as compact representation may have more aliased elements in Java as shown in Fig. 3(b).

To obtain the correct result in this example, $r.d$ should be recognized not only as a memory location that contains its addressed value in $\langle p.d, r.d \rangle$ but also 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 $\langle p.d, r.d \rangle$ alias relation for the alias computation should be analyzed differently. Particularly, the alias computation should consider that reference names in an assignment statement *l-value* and *r-value* should be used to assign the address value of the memory location.

5. REFERENCE-SET REPRESENTATION

An alias analysis algorithm computes alias sets in a piece of program. Each statement of the program collects an alias set from its predecessor and updates it with the statement itself and passes the resulting alias set to its successor(s). The alias computation should be iteratively performed until the alias sets and a calling graph have converged for the program. Since the computation mainly depends on the number of elements in the alias sets and the representation of the elements, alias set representation affects the efficiency of the entire algorithm.

We propose the *reference-set* representation to improve the accuracy and the efficiency of the alias computation and its type inference for Java.

Reference-set: a set of alias references that consist of more than two references which refer to an object; $R_i = \{r_1, r_2, \ldots, r_j\}$: for each j, initially $j \geqslant 2$ 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 the data flow computation in an alias analysis, $j \geqslant 1$ is allowed when passing references forward and backward at a call site.

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

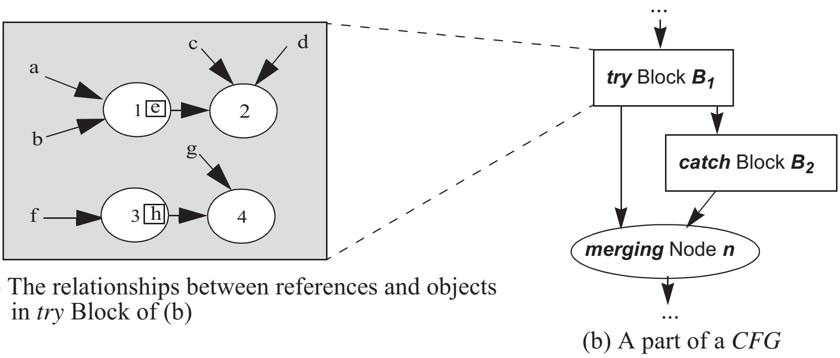
Initially, we only consider *reference-sets* that contain more than two elements since a *reference-set* with one element is redundant for alias analysis. For example, in a statement s, each *reference-set* and *alias set* for the alias relation in Fig. 4(a) are represented as follows:

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

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

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 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 an alias computation depends on the space complexity of each representation. Thus, the efficiency of whole algorithms is improved via the *reference-set* representation.

We can maintain the safety of alias analysis in Java with structures such as *CFG, CG*, and *Type Table*, especially for exception statement in Java. Java provides an exception handling mechanism with the *try/catch/finally* construct. The *try* block handles its exceptions and abnormal exits



(a) The relationships between references and objects in *try* Block of (b)

(b) A part of a *CFG*

Fig. 4. The relationships between objects in a block of *CFG*.

```
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");
  }
 }
 public static int c(int i) throws MyException, MyOtherException {
  switch(i){
       case 0: throw new MyException("too low input");
       case 1: throw new MySubException("still too low input");
       case 99: throw new MyOtherException("too high input");
       default: return i*i;
} } }
```

```
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);
 }
}
```

Fig. 5. An exception handling example code [Flan97].

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 5 shows example *exception* classes that were written by Flanagan.(27)

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 $\langle N_{CFG}, E_{CFG}, n_{entry}, n_{exit}, B_{CFG}\rangle$; 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 predecessor and 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

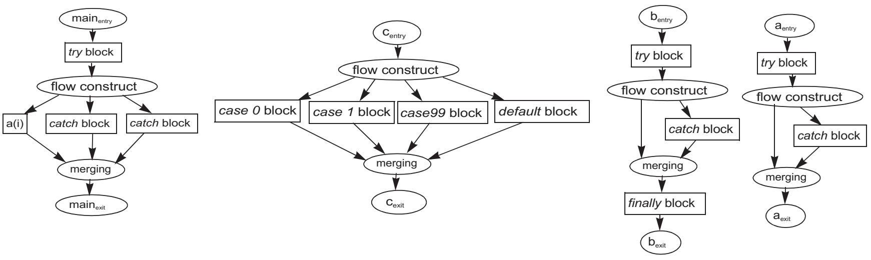


Fig. 6. *CFG*s of example classes in Fig. 5.

the exception statement. Further, 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 6 shows *CFG*s of example classes in Fig. 5.

A *CG* is needed for interprocedural alias analysis between a calling method and the methods it calls at a call statement. Our *CG* is a directed graph defined as $\langle N_{CG}, E_{CG}, n_{main} \rangle$; N_{CG} is a set of nodes and each node is a method shown once 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 7 shows the *CG* of example classes in Fig. 5.

A type table contains all possible types of each reference variable. The type table is built during the execution of our algorithm and it contains

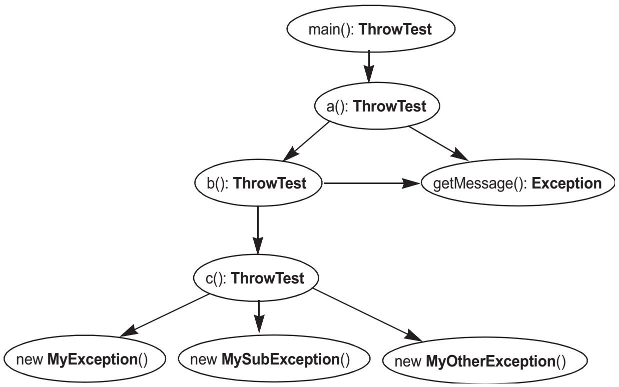


Fig. 7. *CG* of example classes in Fig. 5.

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 overrid-den 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.

6. PROPAGATION RULES OF ALIAS INFORMATION

In a flow-sensitive alias analysis, alias information is kept up-to-date for each program statement. The information is propagated to the next statement and subsequently updated according to the *CFG* of a method. Figure 8 presents possible control flow rules in a program. This analysis for Fig. 8 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 described by the following equation:

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

In this equation, *pred*(*n*) represents a predecessor node of the node *n*. Mod_{kill} denotes the alias set modified after killing some *reference-sets* of *in*(*n*) and Mod_{gen} is the subsequent alias set after generating the new *reference-sets* on Mod_{kill} . In our *reference-set* representation, an operator $\bigcup$ works within the same *reference-sets* between alias sets as well as the alias

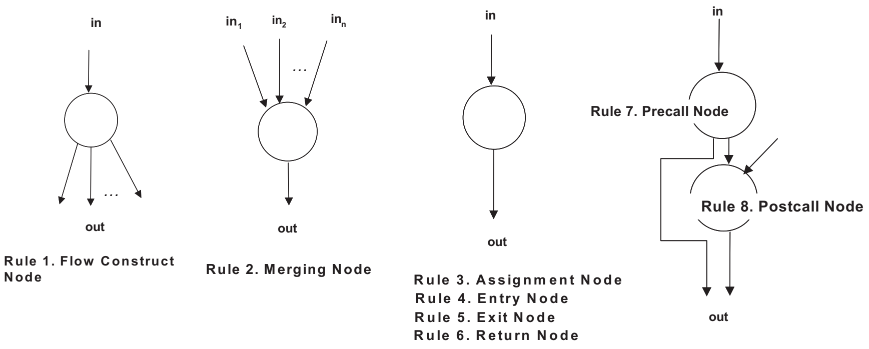


Fig. 8. Flow control rules.

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:

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

For an exception block B that consists of nodes: $n_1, n_2, \ldots, n_m$, we can relate the block and its nodes by the following equations:

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

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

6.1. 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 rule that has several out going edges with the same *out* information:

$$\frac{\begin{array}{l} in(n) = out(n_{pred}) \\ n_{pred}\text{: predecessor node of } n \text{ or predecessor block of } n \end{array}}{out(n) = in(n)} \quad [\text{Flow Construct Node}]$$

The merging node rule is as follows:

$$\frac{\begin{array}{l} in(n) = \bigcup_{p \in n_{pred}} out(p) \\ n_{pred}\text{: predecessor node of } n \text{ or predecessor block of } n \end{array}}{out(n) = in(n)} \quad [\text{Merging Node}]$$

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.

$$
\begin{aligned}
& in(n) = out(n_{pred}) \\
& n_{pred}\text{: predecessor node of } n \\
& x = LHS, \\
& y = RHS, \\
& \forall i, j\ R_i, R_j \in in(n) \rightarrow [Mod_{kill}(in(n)) = \{R_i \mid kill\ x \in R_i\} \\
& \qquad \cup \{R_i \mid kill\ R_j.f \in R_i \text{ when } q \in R_j \text{ and } x = q.f\}] \\
& \qquad \wedge [in(n) = in(n) - Mod_{kill}(in(n))] \\
& \qquad \wedge [KILL(in(n)) = \{x, R_j.f\}], \qquad (1) \\
& \forall k\ R_k \in in(n) \rightarrow [Mod_{gen}(in(n)) \\
& \qquad = \{R_k \mid R_k = R_k \cup KILL(in(n)) \text{ when } y \in R_k\}] \\
& \qquad\quad \wedge [in(n) = in(n) - Mod_{gen}(in(n))], \qquad (2) \\
& n\text{: the first node of an exception block B} \rightarrow in(B) = in(n), \\
& n \text{ is a } \textit{Potential Exception Statement} \rightarrow PES = PES \cup \{n\} \\
& \rule{10cm}{0.4pt} \text{ [Assignment Node]} \\
& out(n) = Mod_{kill}(in(n)) \cup Mod_{gen}(in(n)) \cup in(n)
\end{aligned}
$$

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 Fig. 4(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 Figs. 4(a) and (b) as follows:

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

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 based on (1) *of Assignment Node* rule as follows:

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

Since R_4 includes *RHS*, $Mod_{gen}(in(B_2))$ and $in(B_2)$ are computed based on (2) *of Assignment Node* rule as follows:

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

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:

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

Finally, $in(n)$ for the merging node n is the union set of $out(B_1)$ and $out(B_2)$ based on *Merging Node* rule as follows:

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

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.

Following is the another example to detect *reference-set* of Fig. 3(b) by assuming the current block of the assignment statement $p.d = w$ as B_1:

$$R_1 = r = \{p, q\}, \qquad R_2 = v = \{p.d, q.d\} = \{R_1.d\}, \qquad R_3 = \{w\}$$
$$in(B_1) = \{R_1, R_2\}$$

Because *LHS* is a qualified expression related to R_1, $Mod_{kill}(in(B_1))$, $in(B_1)$, and $KILL(in(B_1))$ are computed based on (1) *of Assignment Node* rule as follows:

$$R_1 = \{p, q\} \quad \text{and} \quad R_2 = \{R_1.d\} \qquad \text{then} \quad R_2 = \{\,\}$$
$$Mod_{kill}(in(B_1)) = \{R_2\} \qquad in(B_1) = \{R_1\} \qquad KILL(in(B_1)) = \{R_1.d\}$$

Since R_3 includes *RHS*, $Mod_{gen}(in(B_1))$ and $in(B_1)$ are computed based on (2) *of Assignment Node* rule as follows:

$$Mod_{gen}(in(B_1)) = \{R_3 \mid R_3 = R_3 \cup \{R_1.d\} = \{w, R_1.d\} = R_3 \quad in(B_1) = \{R_1\}$$

Thus, $out(B_1)$ is the union set of $Mod_{kill}(in(B_1))$, $Mod_{gen}(in(B_1))$, and $in(B_1)$ as follows:

$$out(B_1) = Mod_{kill}(in(B_1)) \cup Mod_{gen}(in(B_1)) \cup in(B_1) = \{R_2, R_3, R_1\}$$

$$\text{when} \quad R_1 = \{p, q\}, \qquad R_2 = \{\ \}, \qquad R_3 = \{w, R_1.d\}$$

Finally, $out(B_1)$ has the following reference set because R_2 has no element, which has the correct aliased elements:

$$out(B_1) = \{R_1, R_3\} \qquad \text{when} \quad R_1 = \{p, q\}, \quad R_3 = \{w, R_1.d\}$$

With this example for Fig. 3(b), we have shown that our reference set of $out(B_1)$ is more precise than the aliased elements A_{OUT} of the compact representation shown in Section 4.

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

$\rightarrow [Mod_{kill}(in(n)) = \{R_i \mid kill\ x \in R_i \text{ for } x \in LOCAL(M)\}]$

$\wedge [R_r = \{R_r \mid kill\ x \in R_r \text{ for } x \in LOCAL(M) \text{ when } r \in R_r\}]$,

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, $LOCAL(M) = \{v \mid v \text{ is a local variable of } M\}$

$$\begin{aligned}
&\forall i \;\; R_i \in in(n) \\
&\quad \rightarrow [Mod_{kill}(in(n)) = \{R_i \mid kill\ x \in R_i \text{ for } x \in LOCAL(M)\}] \\
&\qquad \wedge [in(n) = in(n) - Mod_{kill}(in(n))],
\end{aligned}$$

$$\overline{\hspace{30em}} \quad \text{[Exit Node]}$$

$$out(n) = Mod_{kill}(in(n)) \cup in(n) \cup PES$$

6.2. Rules for *Interprocedural Analysis*

We build our system based on *Context-Insensitive CG* to compare with the existing system.[5, 6, 10, 15, 19, 26, 35, 39] 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 the 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[19, 39] 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 as in Fig. 8. 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

$$RHS = E_c.M_c, \tag{1}$$

$RHS = M_c$,

$$\forall i,\, a_i = \text{the } i\text{th actual parameter of the callee } M_c,$$

$$\forall i,\, f_i = \text{the } i\text{th formal parameter of the callee } M_c,$$

$$\forall i,\, R(a_i) \in in(n) \rightarrow [R_{pass}(a_i) = \{a_i, f_i\}] \wedge [R(a_i) = R(a_i) - R_{pass}(a_i)], \qquad (2)$$

$$RHS = M_c, \forall i,\, R(a_i) \in in(n),\, v \text{ is a nonlocal variable in the callee } M_c,$$

$$\begin{aligned} &\forall f\ \forall v\ R(v.f) \in in(n) \\ &\rightarrow [R(v) = R(v) - \{v\}] \wedge [R(v.f) = R(v.f) - \{v.f\}] \\ &\wedge [R_{pass}(v) = \{v\}] \wedge [R_{pass}(v.f) = \{v.f\}] \\ &\wedge [PASS(M_c) = \bigcup \{R_{pass}(a_i), R_{pass}(v), R_{pass}(v.f)\}], \end{aligned}$$

$$RHS = E_c.M_c, \forall i,\, R(a_i) \in in(n), \forall f\ \forall a_i,\, R(a_i.f) \in in(n), \forall f,\, R(E_c.f) \in in(n)$$

$$\begin{aligned} &\rightarrow [R(E_c.f) = R(E_c.f) - \{E_c.f\}] \wedge [R_{pass}(E_c.f) = \{E_c.f\}] \\ &\wedge [R(a_i.f) = R(a_i.f) - \{a_i.f\}] \qquad (3) \\ &\wedge [R_{pass}(a_i.f) = \{a_i.f\}] \wedge [PASS(M_c) \\ &= \bigcup \{R_{pass}(a_i), R_{pass}(a_i.f), R_{pass}(E_c.f)\}], \end{aligned}$$

n: the first node of an exception block $B \rightarrow in(B) = in(n)$

———————————————————— [Precall Node]

$out(n) = in(n)$

$PASS(M_c)$ represents the set of actual, formal parameters and nonlocal 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 nonlocal variables accessible by a called method M_c.

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

$PRECALL(M_c)$: a precall node of the callee M_c,

$$in(n) = \bigcup_{p \in PRECALL(M_c)} PASS(p),$$

n: the first node of an exception block $B \rightarrow in(B) = in(n)$

———————————————————— [Entry Node]

$out(n) = in(n)$

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

$$in(n) = \bigcup_{p \in n_{precall}} out(n_{precall}) \quad (1)$$

$n_{precall}$: a precall node of n

$$\begin{aligned}
&RHS = E_c.M_c \rightarrow FIELD(E_c)\\
&\qquad = \{f \mid f \text{ is a field name in an object referred by } E_c\}, \quad (2)\\
&RHS = M_c \rightarrow FIELD(E_c) = \emptyset,\\
&RHS = new\ M_c \rightarrow FIELD(E_c) = \emptyset \wedge A(r),\\
&EXIT(M_c) = \{e \mid e \text{ is an exit alias set from a possible callee method } M_c\}, \quad (3)\\
&LHS = \emptyset, \forall R_{passb} \in EXIT(M_c)\\
&\qquad \rightarrow [R_{passb} = R_{passb} - \{v \mid v \text{ is a local variable in the callee } M_c\}]\\
&\qquad\quad \wedge \left[EXIT(M_c) = \bigcup_{for\ all\ M_c} R_{passb} \right], \quad (4)\\
&LHS \neq \emptyset, \forall R_{passb} \in out(return\ node)\\
&\qquad \rightarrow [R_{passb} = R_{passb} - \{v \mid v \text{ is a local variable in the callee } M_c\}]\\
&\qquad\quad \wedge \left[EXIT(M_c) = \bigcup_{for\ all\ M_c} R_{passb} \right],\\
&\forall i\ R_i \in EXIT(M_c), \forall j\ R_j \in in(n)\\
&\qquad \rightarrow [R_i \mid R_i = R_i \cup R_j \text{ when } i = j] \wedge [EXIT(M_c) = EXIT(M_c) - R_i]\\
&\qquad\quad \wedge [in(n) = in(n) - R_j], \quad (5)
\end{aligned}$$

$$\begin{aligned}
&exit(RHS) = \bigcup_{e \in EXIT(M_c)} out(e) \cup \bigcup_{p \in n_{precall}} out(precall\ node) \cup \bigcup_{for\ all\ i} R_i,\\
&LHS = \emptyset \rightarrow out = exit(RHS),\\
&LHS = x, \forall i, j\ R_i, R_j \in exit(RHS), R(RHS) \in exit(RHS)\\
&\qquad \rightarrow [Mod_{kill}(exit(RHS)) = \{R_i \mid kill\ x \in R_i\}\\
&\qquad\quad \cup \{R_i \mid kill\ R_j.f \in R_i \text{ when } q \in R_j \text{ and } x = q.f\}]\\
&\qquad\quad \wedge [exit(RHS) = exit(RHS) - Mod_{kill}(exit(RHS))]\\
&\qquad\quad \wedge [KILL(exit(RHS)) = \{x, R_j.f\}],\\
&R(RHS) \in exit(RHS)\\
&\qquad \rightarrow [Mod_{gen}(exit(RHS)) = \{R(RHS) \mid R(RHS)\\
&\qquad\quad = R(RHS) \cup KILL(exit(RHS))\}]\\
&\qquad\quad \wedge [exit(RHS) = exit(RHS) - Mod_{gen}(exit(RHS))]\\
&\qquad\quad \wedge [out = Mod_{kill}(exit(RHS)) \cup Mod_{gen}(exit(RHS)) \cup exit(RHS)],\\
&n \text{ is a Potential Exception Statement} \rightarrow PES = PES \cup \{n\}
\end{aligned}$$

$$\overline{} \quad \text{[Postcall Node]}$$

$$out(n) = out$$

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 9 is the example where we compute the alias set by following the interprocedural analysis rule; reference variable c might refer to the object 2 or 3. If we assume that Fig. 9(a) represents the state at *try* block B_1, the *out* alias set of the block B_1 is:

$$out(B_1) = \{R_2, R_3\}$$

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

After executing the call statement t at the block B_2 in Fig. 9(c), the alias set of its *precall node* becomes as follows:

$$in(t) = in(B_2) = \{R_2, R_3\},$$

$$RHS = a.update(c), \qquad \text{(1) of precall node}$$

$$a_i = c,\ f_i = i,$$

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

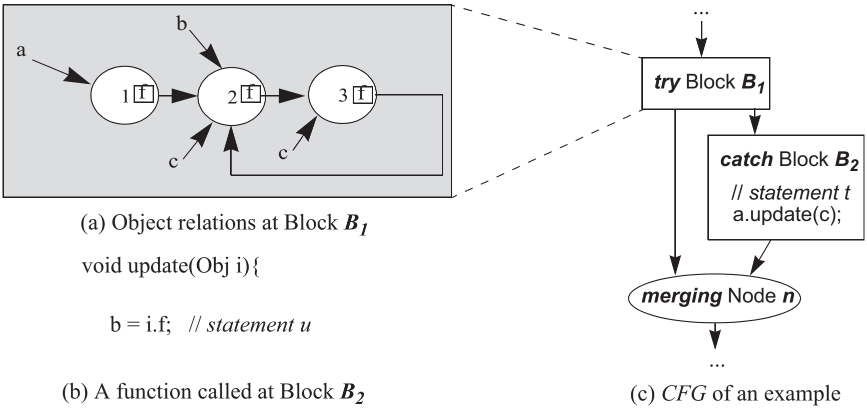
$$\text{or} \quad R(a_i) = R_3 = R_3 - \{c\} = \{R_2.f\}, \qquad (2)$$

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

$$\text{and} \quad R_{pass}(a.f) = \{a.f\}, \qquad (3)$$

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

$$out(t_{precall}) = \{R_2, R_3\}$$



(a) Object relations at Block B_1

```
void update(Obj i){

    b = i.f;   // statement u
```

(b) A function called at Block B_2

(c) *CFG* of an example

Fig. 9. Example of an interprocedural Analysis.

In this computation, (1) of *precall node* is selected because $a.update(c)$ is a qualified call statement. (2) of *precall node* is to make pairs between formal parameter i and actual parameter c. Thus, we can compute safe *reference sets* $R(a_i)$ for either block B_1 and B_2. (3) of *precall node* computes $R_{pass}(a.f)$ accessible by a called method $a.update(c)$. $R_{pass}(a.f)$ is computed as a set of nonlocal variables accessible by a called method $a.update(c)$. 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:

$$R_{pass}(a_i) = \{c, i\},\ R_{pass}(a.f) = \{a.f\},$$
$$R_{pass}(a_i) = R_{pass}(a.f) = \{c, i, a.f\} = R_{pass}(R_2) \text{ for } R_2,$$
$$R_{pass}(a_i) = \{c, i\} = R_{pass}(R_3) \text{ for } R_3, \dots\dots\dots\dots \text{ computed in } \textit{Precall Node}$$
$$in(u) = \{R_{pass}(R_2), R_{pass}(R_3)\},$$
$$R(b) = \{b, i.f, c.f\}, \dots\dots\dots\dots \text{ computed in } \textit{Assignment Node}$$
$$out(u) = update_{exit}$$
$$= \{R(b), R_{pass}(R_2), R_{pass}(R_3)\} \dots\dots\dots\dots \text{ computed in } \textit{Exit Node}$$

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:

$$in(t_{postcall}) = out(t_{precall}) = \{R_2, R_3\}$$
$$\text{where} \quad R_2 = \{b, c.f\},\ R_3 = \{c.f\}, \tag{1}$$
$$\text{FIELD(a)} = \{a.f\}, \tag{2}$$
$$EXIT(update) = update_{exit} = \{R(b), R_{pass}(R_2), R_{pass}(R_3)\}, \tag{3}$$
$$\text{where } R_{pass}(R_2) = \{c, i, a.f\} \text{ and } R_{pass}(R_3) = \{c, i\},$$
$$R(b) = R_{passb} = \{b, c.f\},\ R_{passb}(R_2) = \{c, a.f\},$$
$$R_{passb}(R_3) = \{c\} \text{ for the caller,}$$
$$EXIT(update) = \{R(b), R_{passb}(R_2), R_{passb}(R_3)\}, \tag{4}$$
$$R_2 = R_2 \cup R_{passb} \cup R_{passb}(R_2) = \{b, R_3.f, c.f, c, a.f\},$$
$$R_3 = R_3 \cup R_{passb} \cup R_{passb}(R_3) = \{R_2.f, b, c.f, c\}, \tag{5}$$
$$\text{Thus, } out(B_2) = out(t) = out(t_{postcall}) = \{R_2, R_3\}$$
$$\text{Finally, } in(n) = out(B_1) \cup out(B_2) = \{R_2, R_3\}$$
$$\text{where } R_2 = \{a.f, b, c, R_3.f\} \text{ and } R_3 = \{R_2.f, b, c\}$$

The propagation rule (1) of *postcall* node computes an input reference set with its *precall* node. (2) of *postcall* node presents FIELD(a), a set of field names, in an object referred by a reference a of $a.update(c)$. (3) $exit(update)$ is a set of exit nodes of the called method *update* and computed in a *CG* by integrating all $out(u)$ and outgoing edges from callers and their exit nodes. (4) is to pass back to the caller the elements of the *reference set* $exit(update)$ accessible from the caller. (5) updates *reference sets* R_2 and R_3 by considering elements computed in the caller $a.update(c)$. Finally, we can compute $in(n)$ of the node n.

7. ALIAS ANALYSIS ALGORITHM

7.1. Algorithm

Our alias analysis algorithm in Fig. 10 visits all nodes of a *CG* until a fixed point is reached. The algorithm traverses each node of a *CG* in topological and reverse topological order so as to possibly shorten the running time in reaching a fixed point.[6, 10, 20] The set *TYPES* has all possible class types as elements for a callee to build a safe CG. We define $TYPES_{table}(r)$ as 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 over previous work[20] without losing the accuracy of a resulting set.

7.2. Complexity of the Algorithm

For the outermost 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 statements* 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 the *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

```
1  Algorithm Alias Analysis
2   construct an initial CG with main method;
3   repeat {
4      for each method $T.M \in N_{cg}$,
5        alternating between topological and reverse topological order {
6        for each node $n \in N_{cfg\,(T.M)}$ in structural order {
7              if n is a call statement node {
8                    if ($RHS = E_c.M_c$) {
9                          compute the set of inferred types from the reference-set for $E_c$;
10                         compute the set TYPES resolved
11                               from the inferred types and class hierarchy;
12                   } else if ($RHS = M_c$) {
13                         TYPES:= {T};
14                   } else if (RHS = new $M_c$) {
15                         TYPES:= {$M_c$};
16                   }
17                   if LHS exists
18                         $TYPES_{table}(LHS) = TYPES_{table}(RHS)$;
19                   for each type $t \in TYPES$ {
20                         if $t.M_c$ is not in CG
21                               create a CG node for $M_c$;
22                         if no edge from T.M to $t.M_c$ with a label n
23                               connect an edge from T.M to $t.M_c$ with a label n;
24                   }
25             compute $out(n_{precall})$ for a precall node $n_{precall}$;
26             compute $out(n_{postcall})$ for a postcall node $n_{postcall}$;
27             } else {
28                   if n is an assignment statement node
29                         $TYPES_{table}(LHS) = TYPES_{table}(RHS)$;
30                   if n is a merging statement node
31                         $TYPES_{table}(LHS) = TYPES_{table}(LHS) + TYPES_{table}(RHS)$;
32                   compute out(n) using data-flow equation and propagation rule;
33             }
34       }
35  }
36  } until CG and alias set for every CFG node converge
```

Fig. 10. Alias analysis algorithm.

traversed once per each iteration. For the second-to-outer loop (line 4), the time complexity becomes $O(N_{cg})$ if N_{cg} is the final number of nodes in a *CG*. For the most inner loop (line 6), 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 dominant parts of the running in the inner loops are the call statement nodes, so the worst time complexity depends on the number of call statements. The time complexity of a set of inferred types is $O(R_m)$ where R_m is the number of reference variables in a program code and a type table contains all possible types of a 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 meth-ods in a calling method. The worst time complexity of a *precall* and a *postcall* node is $O(R_p \times R)$; R_p is the maximum number of *reference-sets* 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++ generate 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 outer most loop with the same iterative algorithm as in Fig. 10, 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 existing work(10, 19, 20) 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 larger 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 that of prior work.(6, 10, 20)

8. EXPERIMENTAL METHOD

8.1. Alias Detector Framework

Figure 11 shows a framework detecting aliases while examining application codes written in Java as benchmarks. It mainly consists of three parts: parser, syntax tree builder, and alias analysis.

Sun Microsystems gives a basic parser for a JDK-1.0.2 grammar in JavaCC(32) which can be converted to Java programs. Purdue university built JTB,(33) a Java syntax tree builder. The alias analysis systems are built on JavaCC and JTB. It automatically generates a JavaCC grammar with the proper annotations to build the syntax tree during parsing. The syntax tree is extended by adding the data structures of *reference-set* and *object-pair* representations with class structures of TT and CFG.(45) Our system adds to these parser and syntax tree builder semantic actions such as type and scope checking. Based on this, during executing alias analysis algorithm, the CFG and CG are built; dynamic type information is stored in the type table; alias set is detected on alias computing rules.

As shown in Fig. 11, the parser reads the example input classes and stores attribute information of the classes. Also, type and scope check operations are done during parsing. The syntax tree builder builds a syntax tree of each input and modifies the class information. Our alias analysis algorithm shown in Fig. 11 uses the information of classes constructed on

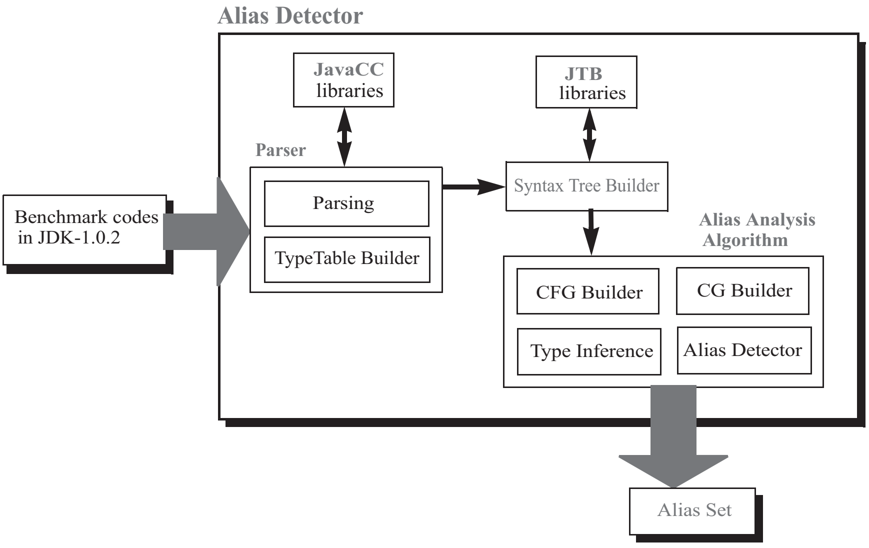


Fig. 11. Alias detection for Java codes.

Table I. Characteristics of Benchmark

	Num of classes	Num of lines	Num of methods/ constructors	Num of overridden methods
Dynamic CG	5	54	7	4
Binary Tree	5	154	8/2	1
Ray Tracer	12	1,213	53/13	4
Exception Block	4	67	4/6	0
Recursive Call	5	160	10/2	1

parsing so that it builds CFG and CG. The algorithm converges all hidden alias sets of each class and then collects final alias set. As a result, our alias detector detects alias set of given Java codes (Kottos, Ceng, and Asadal are the names of hosts used in the experiment).

We have executed alias analysis algorithms on benchmark codes with the *reference-set* and the existing *object-pair* representations.(6, 19, 20, 39) We only focus on the running time for the experiment because the aliases detected for these two executions are implicitly the same—the benchmark does not have the indirect object relations that may generate imprecise aliases with the existing work. We have executed five benchmark codes on the systems: *Dynamic CG, Binary Tree, Ray Tracer, Exception Block*, and *Recursive Call*. The characteristic of each code is presented in Table I. *Dynamic CG* is written in C++ initially by Carini(5) and adapted in Java by ourselves. It has conditional statements and overridden methods. *Binary Tree* is provided by Pro-active group.(41) 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.(34) We adapt to this work *Exception Block* built by Flanagan.(27) It contains *try/catch* and *try/catch/finally* constructs for Java *exceptions*. It is used to measure the safety of the exception blocks in type inference and alias set. *Recursive Call*, modified from *Binary Tree*, includes more calling statements to extend a tree so that

Table II. 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

it will show whether our approach can yield an improvement in performance for dynamic calls. Table II presents properties of the hosts to execute those benchmark codes.

8.2. Experimental Results

Figure 12 presents the running times of *Dynamic CG*. For all hosts, the running time of *reference-set* is faster than *object-pair* because our *Type Table* has a more efficient structure to search possible types of methods than Carini's.(15) Figure 13 is the running 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 are for JVM performance measurement, do not have any big difference in the running time of alias analysis for any alias representation. Figure 14 presents the running times of *Exception Block*. For all hosts, the running time of *object-pair* is almost same as or 4% faster than *Reference-Set*. These differences come from the usage of the different alias representations. It means that our algorithm will take longer to compute aliases if a code does not contain possible dynamic types for references. Figures 16 and 17 shows that the running time of *reference-set* is faster than *object-pair*. Figure 17 presents that the running time of *reference-set* is much faster than *object-pair* on Ceng and Asadal

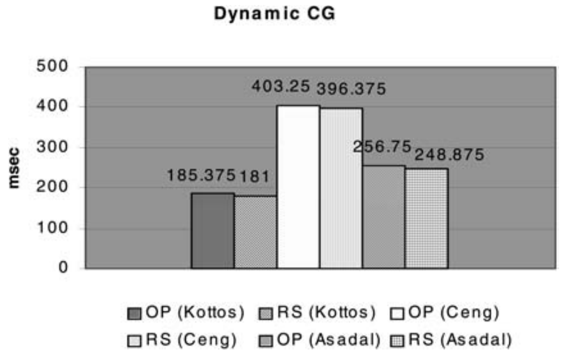


Fig. 12. Running time of *Dynamic CG*.

Table III. The Relative Speed of Fig. 12

	Kottos	Ceng	Asadal
Object-Pair	1.02	2.23	1.42
Reference-Set	1	2.19	1.38

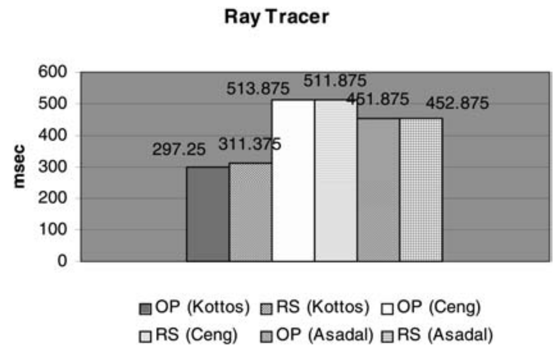


Fig. 13. Running time of *Ray Tracer*.

Table IV. The Relative Speed of Fig. 13

	Kottos	Ceng	Asadal
Object-Pair	0.95	1.65	1.45
Reference-Set	1	1.64	1.45

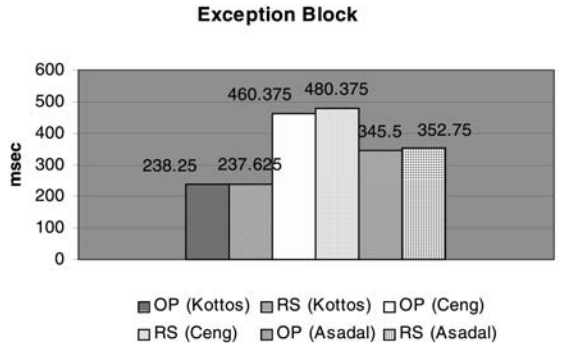


Fig. 14. Running time of *Exception Block*.

Table V. The Relative Speed of Fig. 14

	Kottos	Ceng	Asadal
Object-Pair	1	1.94	1.45
Reference-Set	1	2.02	1.48

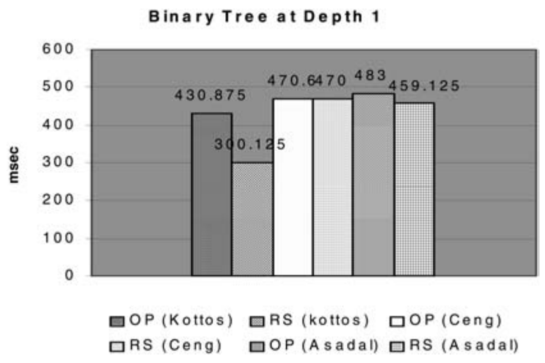


Fig. 15. Running time of *Binary Tree* at depth 1.

Table VI. The Relative Speed of Fig. 15

	Kottos	Ceng	Asadal
Object-Pair	1.44	1.57	1.61
Reference-Set	1	1.57	1.53

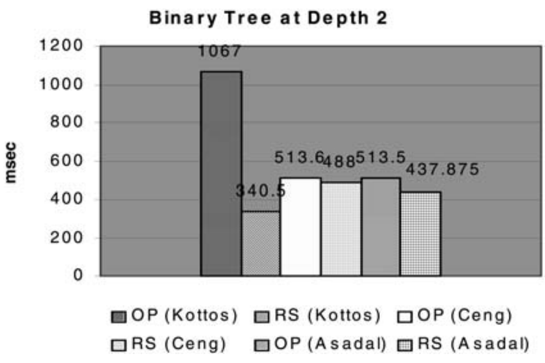


Fig. 16. Running time of *Binary Tree* at depth 2.

Table VII. The Relative Speed of Fig. 16

	Kottos	Ceng	Asadal
Object-Pair	3.13	1.51	1.51
Reference-Set	1	1.43	1.29

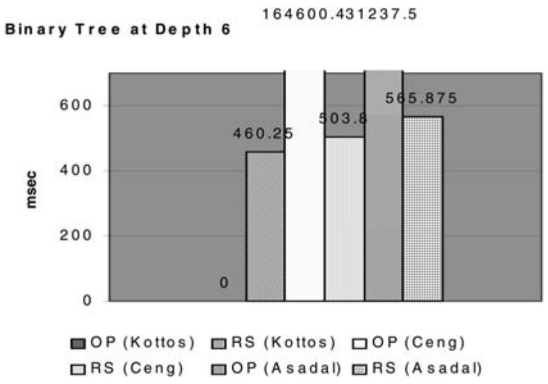


Fig. 17. Running time of *Binary Tree* at depth 6.

Table VIII. The Relative Speed of Fig. 17

	Kottos	Ceng	Asadal
Object-Pair	NA	357.63	67.87
Reference-Set	1	1.09	1.23

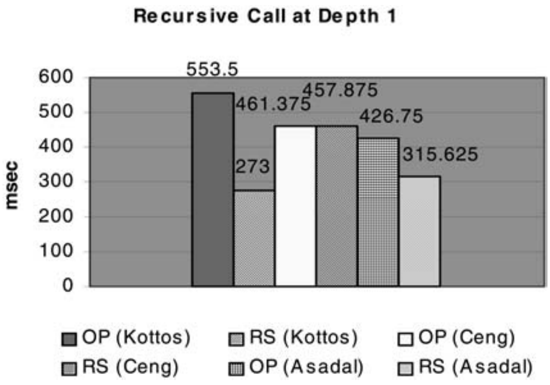


Fig. 18. Running time of *Recursive Call* at depth 1.

Table IX. The Relative Speed of Fig. 18

	Kottos	Ceng	Asadal
Object-Pair	2.03	1.69	1.56
Reference-Set	1	1.68	1.16

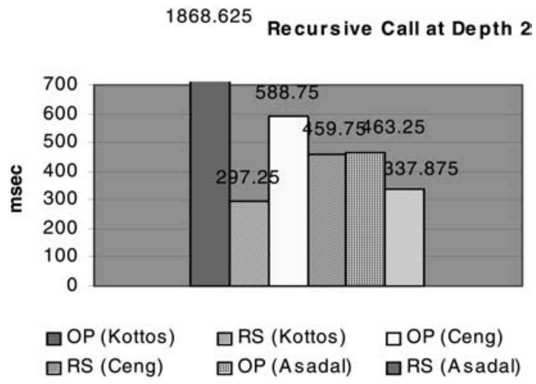


Fig. 19. Running time of *Recursive Call* at depth 2.

Table X. The Relative Speed of Fig. 19

	Kottos	Ceng	Asadal
Object-Pair	6.29	1.98	1.56
Reference-Set	1	1.55	1.14

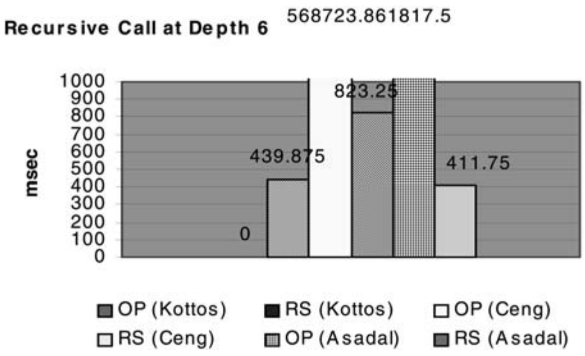


Fig. 20. Running time of *Recursive Call* at depth 6.

Table XI. The Relative Speed of Fig. 20

	Kottos	Ceng	Asadal
Object-Pair	NA	1292.92	140.53
Reference-Set	1	1.87	0.94

respectively. For Kottos, *object-pair* is not measurable because the running time is too long. From Fig. 18 to Fig. 20, it presents the similar result to in *Binary Tree*. It shows that our algorithm with *Reference-set* representation has better performance comparing to existing *object-pair* representation in particular for benchmark codes such as *Binary Tree* and *Recursive Call*, which contains accumulated aliased objects inside many conditional and recursive call statements.

Also, benchmark results(8, 34, 44) show that JVM of Windows NT has the best score and JVM of AIX has the worst score among AIX, Sparc, and NT. Our results meet with the results particularly in *object-pair representation* for *Binary Tree* and *Recursive Call* when those have the larger depths such as depth 2 and 6 that make those codes iterate much longer.

9. CONCLUSION

There is considerable interest in Java as a language for high performance computing, in good part due to its portability, increasingly widespread use, and its support for distributed computation; the latter point is especially significant as internet and grid-based computing becomes more widespread. Hence, it is important to investigate, in the context of high performance computing applications, the object-based parallelism which is the natural idiom of object-oriented Java programming. We have shown our work makes advances in the area of detecting aliased references statically in a Java environment. This contributes to the effective use of Java in HPC by: making it possible to exploit instruction level parallelism; to detect side effects, and avoid their adverse effects of; to better analyze programs for parallelism and distribution, and thereby context switch overhead, and communication overhead for distributed and cluster computing. Our alias analysis among objects in Java is applicable to the parallelizing of Java by detecting possible side effects.

We have presented a *reference-set* alias representation for Java, and we showed that it is a better representation for Java than earlier representations developed for C++, because it takes into account the fact that Java has only object references, rather than arbitrary pointers. Further, we have proposed our *flow sensitive alias analysis algorithm* by adapting existing alias analyses(6, 20) for C/C++ to for Java. The algorithm is more precise and efficient than previous work(6, 10, 19, 20, 26, 39, 49) based on the *reference-set* alias representation and its associated type table and data propagation rules. Besides, the algorithm is the safe alias analysis including *exception* statements.

By proposing our additional type information, and by combining an alias analysis algorithm with that type information, we can detect shadowed variables which cannot be detected through conventional means or overridden methods. This algorithm detects more precise alias sets for both shadowed variables and overridden methods. Our algorithm also regards a constructor as a procedure in order to analyze shadowed variables so that calling graph contains constructors to compute the alias set of each constructor by using our proposed equation. Its efficiency is not negatively affected even though the precision is improved by adding extra type information. Our work is the first implementation of alias analysis with type inference for Java. The type information and rules for *reference-set* representation presented in this paper are applicable to C++. In the complexity analysis, we have shown that our *reference-set* alias representation is more precise and efficient for the case of type inference and data propagation rules combined together in the alias analysis algorithm. By using a structural traversal of a *CFG*, the algorithm achieves additional efficiency, surpassing previous work. Further, a possible multithreading solution has been proposed.

Finally, we have built our alias algorithm in Java with the JavaCC parser and JTB syntax tree builder and executed benchmark codes. The benchmark does not have the indirect object relations that may generate imprecise aliases in the *object-pair* relations so that we only focus on the running time. The first experimental result on *Dynamic CG* shows that our dynamic type determination is as safe as *object-pair* representation. The second result on *Ray Tracer* shows that our alias analysis does not show any improvement of efficiency in regular application codes which do not contain many aliases. The third result on *Exception Block* shows that our analysis succeeds in analyzing exceptions in Java. But, it also shows that if a code does not have references with many dynamic types and aliases, our analysis might be less efficient than *object-pair* representation. The final result on *Binary Tree* and *Recursive Call* shows that if a code does have references with many aliases and objects generation, our analysis should be much more efficient than *object-pair* representation.

REFERENCES

1. O. Agesen, D. Detletfs, and J. E. B. Moss, Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines, *ACM on Programming Language Design and Implementation* (1998).
2. O. Agesen, J. Palsberg, and M. I. Schwartzback, Type Inference of Parametric Polymorphism, *Proceedings of ECOOP'95*, Aarhus (Aug. 1995).
3. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison–Wesley (1986).
4. J. P. Banning, An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables, *Proceedings of Sixth POPL* (Jan. 1979).

5. M. Burke, P. R. Carini, and J. Choi, Efficient Flow-Insensitive Alias Analysis in the Presence of Pointers, IBM Research Report, RC 19546 (1994).
6. M. Burke, P. Carini, and J. Choi, Interprocedural Pointer Alias Analysis, Research Report RC 21055, IBM T. J. Watson Research Center (December 1997).
7. D. F. Bacon and P. F. Sweeney, Fast Static Analysis of C++ Virtual Function Calls, *Proceedings of Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'96*, ACM SIGPLAN Notices, Vol. 31, No. 10, San Jose, California (October 1996).
8. CaffeineMark 3.0, Pendragon Software, http://www.pendragon-software.com/pendragon/cm3/results.html
9. D. C. Cann, Compilation Techniques for High Performance Applicative Computation, Technical Report, CS-89-108, Colorado State University (1989).
10. 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*, pp. 232–245 (January 1993).
11. D. Caromel, F. Belloncle, and Y. Roudier, *The C++// Language*, Chapter 7, Parallel Programming using C++, Massachusetts Institute of Technology (1996).
12. B. O. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu, Javaelin: Internet-Based Parallel Computing Using Java, *ACM 1997 Workshop on Java for Science and Engineering Computation, June 21, 1997, PPoPP Las Vegas*.
13. C. Chambers, J. Dean, and D. Grove, Whole-Program Optimization of Object-Oriented Languages, Technical Report, UW-CSE-96-06-02, Department of Computer Science, University of Washington.
14. 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 1999).
15. Paul R. Carini, M. Hind, and H. Srinivasan, Flow-Sensitive Type Analysis for C++, IBM Research Report, RC20267 (Nov. 1995).
16. K. D. Cooper and K. Kennedy, Interprocedural Side-Effect Analysis in Linear Time, *Proceedings of SIGPLAN 88 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 23(7) (July 1988).
17. K. D. Cooper and K. Kennnedy, Fast Interprocedural Alias Analysis, *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 49–59 (Jan. 1989).
18. D. Caromel, W. Klauser, and J. Vayssiere, Towards Seamless Computing and Metacomputing in Java, *Concurrency-Pract. Ex.*, **10**(11–13):1043–1061 (Sept.–Nov. 1998).
19. 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).
20. P. Carini and H. Srinivasan, Flow-Sensitive Type Analysis for C++, Research Report RC 20267, IBM T. J. Watson Research Center (November 1995).
21. Java How to Program. Deitel & Deitel, Prentice–Hall International, Inc. (1997).
22. J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers, Vortex: An Optimizing Compiler for Object-Oriented Languages, *OOPSLA'96 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications* (1996).
23. G. DeFouw, D. Grove, and C. Chambers, *Fast Interprocedural Analysis*, Technical Report 97-07-02, Department of Computer Science and Engineering, University of Washington (July 1997).
24. A. Diwan, K. S. McKinley, and J. E. B. Moss, Type-Based Alias Analysis, *Proceedings of SIGPLAN 98 Conference on Programming Language Design and Implementation* (1998).

25. M. Fahndrich, J. Rehof, and M. Das, Scalable Context-Sensitive Flow Analysis Using Instantiation Constraints, *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, Vancouver, BC, Canada (June 2000).
26. 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*, SIGPLAN Notices, Vol. 29(6), pp. 242–256 (1994).
27. D. Flanagan, *Java in a Nutshell*, 2nd ed., O'Reilly (May 1997).
28. D. Grove, G. DeFouw, J. Dean, and C. Chambers, Call Graph Construction in Object-Oriented Languages, *OOPSLA '97 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications* (1997).
29. J.-L. Gaudiot, T. DeBoni, J. Feo, W. Bohm, W. Najjar, and P. Miller, The Sisal Project: Real World Functional Programming, *Languages, Compilation Techniques, and Run Time Systems for Scalable Parallel Systems Recent Advances and Future Perspectives*, Lecture Notes in Computer Science Series, Springer-Verlag (2001).
30. U. Holzle and O. Agesen, Dynamic vs. Static Optimization Techniques for Object-Oriented Languages, *Theor. Pract. Obj. Syst.*, **1**(3) (1996).
31. M. Hind, M. Burke, P. Carini, and J.-D. Choi, Interprocedural Pointer Alias Analysis, *ACM T. Progr. Lang. Sys. (TOPLAS)*, **21**(4):848–894 (July 1999).
32. Sun Micro Systems, Java Compiler Compiler, *The Parser Generator*, http://www.suntest.com/JavaCC/Version0.8pre2 (April 1998).
33. Purdue University, West Lafayette, Indiana, USA, *Java Tree Builder*, http://www.cs.pur-due.edu/jtb/index.html (May 2000).
34. *Java Grande Forum*, http://www.javagrande.org/.
35. W. Landi, B. G. Ryder, and S. Zhang, A Safe Approximating Algorithm for Interprocedural Pointer Aliasing, *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 235–248 (June 1992).
36. S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, Academic Press (July 1997).
37. E. W. Myers, A Precise Inter-procedural Data Flow Algorithm, *8th Annual ACM Symposium on the Principles of Programming Languages* (1981).
38. J. Plevyak and A. A. Chien, Precise Concrete Type Inference for Object-Oriented Languages, *OOPSLA'94 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications* (1994).
39. H. D. Pande and B. G. Ryder, Static Type Determination and Aliasing for C++, Technical Report LCSR-TR-250-A, Rutgers University (October 1995).
40. J. Palsberg and M. I. Schwartzbach, Object-Oriented Type Inference, *OOSLA '91, Object-Oriented Programming Systems, Languages, and Applications*, pp. 146–161, Phoenix, Arizona (Oct. 1991).
41. Inria, Sophia, France, http://www.inria.fr/oasis/ProActive.
42. B. K. Rosen, Data Flow Analysis for Procedural Languages, *JACM*, **26**(2):322–344 (April 1979).
43. R. Rugina and M. Rinard, Pointer Analysis for Multithreaded Programs, *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, Atlanta, Georgia (May 1999).
44. SciMark 2.0, National Institute Standards and Technology, http://math.nist.gov/cgi-bin/ScimarkSummary.
45. J. Woo, I. Attali, D. Caromel, J.-L. Gaudiot, and A. L Wendelborn, Alias Analysis on Type Inference for Class Hierarchy in Java, *Proceedings of Twenty-Fourth Australasian Computer Science Conference*, ACSC 2001, pp. 206–214 (Jan. 2001).

46. R. P. Wilson and M. S. Lam, Efficient Context-Sensitive Pointer Analysis for C Programs, *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation* (June 1995).
47. J. Woo, J. Woo, I. Attali, D. Caromel, J.-L. Gaudiot, and A. L Wendelborn, Alias Analysis for Java with Reference-Set Representation, *Proceedings of Eighth International Conference on Parallel and Distributed Systems*, ICPADS 2001, pp. 459–466 (June 2001).
48. J. Woo, J. Woo, I. Attali, D. Caromel, J.-L. Gaudiot, and A. L Wendelborn, Alias Analysis for Exceptions in Java, *25th Australasian Computer Science Conference—Australian Computer Science Communications*, Vol. 24, No. 1, IEEE Press (Jan. 2002).
49. J. Woo, J. Woo, and J.-L. Gaudiot, Flow-Sensitive Alias Analysis with Referred-Set Representation for Java, *Proceedings of Fourth International Conference on High Performance Computing in Asia-Pacific Region*, Vol. 1, HPC-ASIA 2000, pp. 485–494 (Dec. 2000).

Printed in Belgium