Flow-Sensitive Alias Analysis with Referred-Set Representation for Java

Jehak Woo
Department of Computer Engineering
Daejin University
Pocheon-Gun, Gyeonggi-Do Korea 487-711
jhwoo@computer.org

Jongwook Woo and Jean-Luc Gaudiot
Department of EE-Systems
University of Southern California
Los Angeles, CA 90089-2563 USA
{jongwook, gaudiot}@usc.edu

Abstract

Alias analysis refers to the determination of objects that may be accessed by two or more names. Many alias analysis algorithms have been proposed for several programming languages. In this paper, we propose a flow-sensitive alias analysis algorithm for Java, an object-oriented language. This algorithm is more efficient and precise than previous algorithms for C++, another object-oriented language. For the efficiency, we define a referred-set representation of an alias that is proper for Java, while the conventional representations in C++ cause an inefficient analysis for Java. We also present a data-flow equation based on propagation rules for the referred-set. The equation computes alias information more efficiently and precisely by removing redundant alias information at a call statement. Finally, we propose our algorithm that uses an iterative looping method for an alias analysis with a structural traverse of a CFG to improve its efficiency.

1. Introduction

Java is a programming language that becomes the standard for several areas including Internet, and high performance applications. Java is often examined in comparison with C++ as an object-oriented language. However, it has many distinguishable features from C++. Objects in Java are especially accessed by references so that there might be many aliases in a Java program. An alias is occurred when an object is bound by two or more names. People [1, 2, 3, 4, 5, 6, 8] have been analyzing aliases because the alias information is useful for high performance computing such as parallelizing compilers and compiler optimizations. If the information is detected statically, optimizations such as inlining can be applicable to compilers for high-performance computing. We can also avoid race conditions, context switching, and communication overheads in parallel computing [11].

Previous works have proposed alias analyses for C or C++ by representing an alias relation with objects and pointers as well as pointer-to-pointers. The research interests are focused on the efficiency and preciseness based on the safety of the alias information detected. The previous works provide useful hints to implement an alias analysis algorithm for Java. However, the representation is not well adaptable to Java without modification. Thus, we have to build a proper alias representation for Java.

Our goal is to propose an efficient and precise alias analysis algorithm for Java. We mainly present the following contributions based on the efficiency and preciseness without losing its safety. First, we analyze problems to adapt the previous alias representations [1, 8] to Java. Second, we devise a referred-set representation appropriate to Java. This representation improves the efficiency in the existing alias analysis algorithms for an object-oriented language [1, 2, 5]. Third, we propose data propagation rules for aliases using the referred-set representation. Finally, we propose a more efficient and precise flow-sensitive alias analysis algorithm based on the existing algorithm [5].

The remainder of the paper is structured as follows. Section 2 presents the difference in naming scheme between C++ and Java, and proposes the referred-set representation for aliases in Java. Section 3 describes the data structures such as a calling graph, control flow graphs, and a type table to implement our algorithm efficiently and precisely. Section 4 explains the propagation rules for intraprocedural and interprocedural analyses. Section 5 shows our alias analysis algorithm. Section 6 computes the time and space complexities of the algorithm. Finally, the conclusion is presented.

2. Alias relation representation in Java

Many alias analyses have been studied for C or C++. In this section, we compare C++ and Java, which are

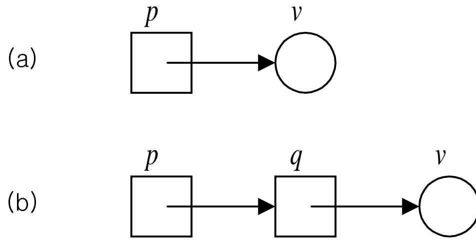


Figure 1. Relation between a pointer and an object in C++

similar object-oriented languages, in order to investigate the feasibility for adaptation of conventional approaches in C++. Thus, we analyze a difference between them, and show that difficulties due to the difference can be overcome by introducing a referred-set representation in Java.

2.1. Comparison between C++ and Java

Naming scheme of an object should be considered to represent aliases properly in C++ and Java. In C++, static objects have object names defined in their declarations but dynamic objects, which are created at run-time, do not have their own names. However, most alias analyses for C++ also assign object names to dynamic objects in order to represent alias relations consistently. An alias relation in C++ is mainly caused by a pointer variable. Figure 1 (a) shows that a pointer variable *p* points to a non-pointer variable *v*. This figure can be interpreted as follows. The pointer variable *p* is naming a pointer-valued object that stores the address of an object with a name of *v*. Thus, the name pointing to the object named by *v* is not *p*, but **p*, which is dereferenced pointer. Because the dereferenced pointer **p* and the object name *v* point to the same object, an alias relation occurs between the two names. This alias relation is represented as <**p, v*>. In C++, a pointer variable is also regarded as an object. Therefore, it is possible that a pointer points to another pointer variable. An alias analysis becomes more complicated because of this pointer-to-pointer variable. Figure 1 (b) shows this situation, where a box depicts a pointer-valued object and a circle depicts a non-pointer object. Alias relations in this figure can be represented as <**p, q*> and <**q, r*>.

Conventional works in C++ have described alias relations by two similar representations: a compact representation [4, 5, 6] and a points-to representation [3]. The representations save spaces by representing all of alias relations without using exhaustive set. It is possible that they represent alias relations in Java. However, there are some problems to use those representations because Java uses references in order to name objects. 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 always created dynamically so that the object becomes an anonymous object that does not have its own name. Thus, every object needs to be assigned its own name to accommodate a compact representation. However, there remains a problem to be solved.

We show which problem can be occurred, through an example in Figure 2. We consider that an assignment statement **p = &w,* where *w* is the name of an arbitrary non-pointer variable, is executed in a state of Figure 2 (a). After the execution, the address value of the pointer-valued object named by **p* is changed to the address value of the object named by *w*. Thus, the address value stored in *r* is changed via <**p, r*> and **r* points to the new object. Finally, <**r, v*> is killed and a new alias relation <**r, w*> is generated.

Figure 2 (b) shows a state of alias relations described by compact representation in Java. In this figure, a circle depicts an object, and a small box represents a reference variable which is a memory location storing the address of an object. This example displays the alias relations similar to the example in Figure 2 (a). When an

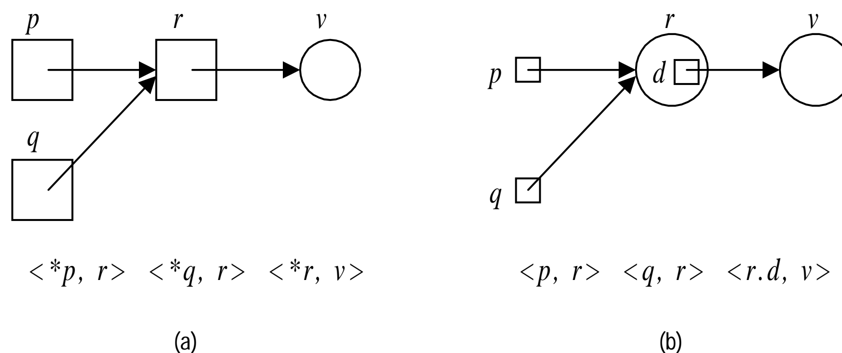


Figure 2. Difference between a pointer in C++ and a reference in Java

assignment statement *p.d* = *w*, where *w* is the name of an arbitrary object, is executed, the address value stored in the reference *p.d* is changed. The address value stored in *r.d* is changed via <*p.d, r.d*> which is inferred from <*p, r*>, and *p.d* and *r.d* are considered as an alias of the same memory location. However, if *v* and *r.d* are recognized as an alias of the same memory location from <*r.d, v*>, an object referred to by the reference *v* is changed and then wrong alias relations <*r.d, w*> and <*v, w*> are generated. For the correct result, <*r.d, v*> should be killed and <*r.d, w*> generated. The wrong result comes from that, in Java, a reference name is used for naming an object without a dereferencing operator such as * in C++. To obtain a right result in this example, *r.d* in <*p.d, r.d*> should be recognized as a memory location that contains the address of an object referred to, but *r.d* in < *r.d, v*> should be recognized as the name of an object referred to. It means that the reference should be interpreted as being dereferenced. To solve this problem, an alias relation for a memory location storing an address should be represented by the other representation. Otherwise, a data flow equation for aliases should recognize the difference. Therefore, reference names in an alias relation should be meant as dereferenced and names in the inferred alias relation such as <*p.d, r.d*> should be considered as a name of a memory location. Especially, during the alias computation, a reference name should be interpreted to refer to a memory location when it is used as an *l-value* or *r-value* of an assignment statement.

2.2. Referred-set representation of alias information

For alias analysis in object-oriented languages, type information of objects is required and the information can be collected more safely via alias information [1, 2]. The type information can be used for resolving virtual functions, which are called as overridden methods in Java. The more precise type information, the more precise an alias analysis becomes. Since the method resolution is executed iteratively whenever an algorithm meets a method, the type inference to collect all the possible type information affects the performance of the alias analysis algorithm.

We propose a *referred-set* representation that improves the efficiency of the type inference with containing the same alias information in compact representation. For each reference, the referred-set representation collects a set of all possible object names that it might refer to. Thus, the set of all possible object names is defined as a referred-set for the reference. All referred-sets that are accessible at a program point should be computed. We define the set of the referred-set as an *alias set*. The alias set contains the entire alias information at the program point.

When alias relations in Figure 3 hold at a program point n, referred-sets and an alias set at that point are represented as follows:

$$A_c = < o_1 >, A_d = < o_3, o_4 >, A_{o_1.f} = < o_2 >, A_{o_3.f} = < o_4 >$$
$$A(n) = \{A_c, A_d, A_{o_1.f}, A_{o_3.f}\}$$

This alias set may be also represented as follows:

$$A(n) = \{< o_1 >_c, < o_3, o_4 >_d, < o_2 >_{o_1.f}, < o_4 >_{o_3.g}\}$$

We use one of these two representations according to necessity.

In this example, we can search a type table, which is defined in section 3.3, to figure out each type of o_3 and o_4 when inferring possible types of the reference *d*.

The space complexity of the referred-set representation is $O(O_t \times A_o)$, where O_t is the number of objects and A_o is the maximum number of references aliased for an object. It is the same as the compact representation. However, the time complexity of the referred-set representation for the type inference is $O(R)$, where R is the maximum number of accessible references at a program point. It is $O(O_t \times A_o)$ in the compact representation. Since normally, R is smaller than O_t, the referred-set representation will be more efficient.

An alias analysis algorithm computes alias sets at every point in a program. Each statement gets an alias set from its predecessor, updates the set with the statement itself and transfers the result alias set to its successor. Since the computation of the alias sets should be iteratively performed until they are converged for the program, it affects the efficiency of the whole algorithm. For example, there is an assignment statement *c* = *d* in Figure 3. It means that the reference *c* refers to the same objects that the reference *d* refers to. Therefore, the referred-set elements of A_c are killed and the referred-set elements of A_d should be copied to the referred-set of A_c. This computation has the same time complexity as a type inference. Thus, the efficiency of the whole algorithm can be improved via the referred-set representation.

3. Data structures for alias analysis

The computation of alias information can be modeled by a data flow analysis [7, 9]. We define a *calling graph* (*CG*) and a *control flow graph* (*CFG*) used to perform the data flow analysis. A *type table* contains types of all

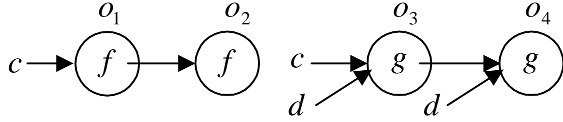


Figure 3. Example of alias relation in Java

objects created dynamically at run-time.

3.1. Calling graph

A *CG* is required to compute the alias set in an interprocedural analysis between a calling and a called method at a call statement. The *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 once in a *CG* even though it is called many times. E_{CG} is a set of directed edges, each of which is connected to every callee for each call site in a caller. n_{main} is a node denoting the main method that is executed initially in a Java program. The *CG* is constructed incrementally as an algorithm proceeds, because the precise method invocations can be achieved after the complete resolution of overridden methods.

3.2. Control flow graph

A *CFG* can be used to compute alias sets in the intraprocedural analysis. The *CFG* is a directed graph defined for each method as $< N_{CFG}$, E_{CFG}, n_{entry}, $n_{exit} >$. N_{CFG} is a set of nodes including n_{entry}, n_{exit}, and each statement of the method. E_{CFG} is a set of directed edges that represent the control flows between a predecessor and a successor statement; n_{entry} represents the entry node of the method; n_{exit} represents the exit node of the method.

In our *CFG*, nodes have seven types based on their purpose: an entry node, an exit node, an assignment statement node, a call statement node, a return statement node, a flow construct node, and a merging node.

A flow construct node represents the start of an *if* or *while* clause. Each clause in an *if* statement is branched from the *if* node. All the matching clauses are merged into a merging node. In *while* node, merging node is not necessary and directed edge is connected from the last node in the *while* loop to the *while* flow construct node.

3.3. Type table

An object in Java is created dynamically. Thus, the objects creation and their type determination are achieved during the process of an algorithm. Each dynamic object is assigned its own unique name. A type table contains the dynamic objects and their type information. The names of objects are used for the referred-set representation of aliases so that the objects referred to by a reference can be acquired from the referred set of the reference and then the possible types of the reference can be inferred by searching a type table.

4. Propagation rules of alias information

In a flow-sensitive alias analysis, the alias information of each statement should include all alias relations occurred at the point. The information is propagated to the next statement and the aliases affected by that statement are subsequently computed. This propagation and computation of alias information is made through all nodes in a *CFG* of each method.

The computation of an alias set for each node can be modeled by a data flow equation that computes an alias set on the effect of each node. Let $A_{IN}(n)$ be the input alias set of a node n transferred from predecessor nodes; Let $A_{OUT}(n)$ be the output alias set held on exit from a node n. The effect of a node n can be computed by the following data flow equation:

$$A_{OUT}(n) = (A_{IN}(n) - A_{KILL}(n)) \cup A_{GEN}(n)$$

In this equation, $A_{KILL}(n)$ and $A_{GEN}(n)$ denote the alias set to be killed and to be generated through the node n, which are called as a *kill* set and a *gen* set, respectively. After the alias analysis algorithm computes alias sets for all nodes in *CFG* repeatedly until they converge, the equation defines a relationship among alias sets $A_{IN}(n)$ and $A_{OUT}(n)$ of each node n.

The propagation of an alias set in a method is achieved by an intraprocedural analysis through nodes in a *CFG* of the method. This analysis starts with the alias set holding at the entry node n_{entry} of the method and computes the result alias set of each node, traversing all nodes in a structural order, which is explained in section 5. The analysis ends at the exit node n_{exit} of the method.

When it meets a call statement node, the alias set should be propagated into a called method because the set can affect aliases that are computed in the called method. Similarly, the alias set should be propagated back from the called method because its result alias set can affect aliases after the current call statement. Thus, aliases should be interprocedurally propagated from a caller to its callee, and from a callee to its caller. This propagation is made through an entry and an exit node in a called method. A calling method propagates its input alias set to

the entry node in a called method, and then reads the output alias set of the exit node in the callee in order to compute its own output alias set at the call statement node.

In this section, we describe propagation rules according to *CFG* node types. The rule for each node type consists of a killing and a generation rule, which are used to figure out a *kill* set and a *gen* set at each node, respectively. The sets are given to the data flow equation and thus the output alias set of a node can be computed.

4.1. Rules for intraprocedural anlaysis

The propagation rules for intraprocedural analysis are described for every *CFG* node type except an entry and a call statement node type. A rule consists of premises and conclusions divided by a horizontal line. The premises are a set of propositions that constrain an input alias set, information about a node, and intermediate sets. A premise can have a form of conditional implication that is interpreted as follows: When a given condition holds, the implied equation has a meaning and can be solved. The conclusions in a rule define the equations computing a *kill* alias set $A_{KILL}(n)$ and a *gen* alias set $A_{GEN}(n)$ for a node n. When all premises hold, the equations in the conclusions are solved for $A_{KILL}(n)$ and $A_{GEN}(n)$.

First, we define rules for a flow construct and a merging node type as follows. $PRED(n)$ is a predecessor set of node *n*.

$$\frac{PRED(n) = \{p \mid p \text{ is a predecessor node of } n\}, \quad A_{IN}(n) = \bigcup_{p \in PRED(n)} A_{OUT}(p)}{A_{KILL}(n) = \phi, \quad A_{GEN}(n) = \phi}$$

$$\frac{PRED(n) = \{p \mid p \text{ is a predecessor node of } n\}, \quad A_{IN}(n) = \bigcup_{p \in PRED(n)} A_{OUT}(p)}{A_{KILL}(n) = \phi, \quad A_{GEN}(n) = \phi}$$

These two rules have the same meaning and can be interpreted as follows: When $PRED(n)$ and $A_{IN}(n)$ are computed so that the two premises hold, $A_{KILL}(n)$ and $A_{GEN}(n)$ become null sets.

Next rule is regarding an assignment statement node type.

$$A_{IN}(n) = A_{OUT}(n_{pred}),$$
$$a = LHS,$$
$$b = RHS,$$
$$a = E_a.f \rightarrow AN(a) = \{o.f \mid o \text{ is an object name referred by } E_a\},$$
$$a = f \rightarrow AN(a) = \{f\},$$
$$b = E_b.g \rightarrow AN(b) = \{o.g \mid o \text{ is an object name referred by } E_b\},$$
$$b = g \rightarrow AN(b) = \{g\},$$
$$\frac{A_{rhs} = \bigcup_{r \in AN(b)} A_r}{A_{KILL}(n) = \{A_l \mid l \in AN(a)\}, \quad A_{GEN}(n) = \{A_l / A_{rhs} \mid l \in AN(a) \wedge A_{rhs} \neq \phi\}}$$

In this rule, n_{pred} denotes a predecessor node of a node n, and A_l / A_{rhs} is to present the new referred-set A_l whose elements are replaced with the elements in A_{rhs}. *LHS* and *RHS* stand for a left hand side and a right hand side expression for an assignment statement, respectively. The expression can have a qualified form such as $E_a.f$, where E_a is called a primary expression [12]. *AN* stands for a set of aliased names. This set is defined differently in according to whether the expression is qualified or not.

In order to show how the above rule can be applied to an alias analysis, we assume that an assignment statement *c.f = d.g* is analyzed in a state of Figure 3. An alias set for the n_{pred} is expressed as follows:

$$A_{OUT}(n_{pred}) = \{<o_1>_c, <o_3, o_4>_d, <o_2>_{o_1.f}, <o_4>_{o_3.g}\}$$

$A_{IN}(n)$ is equal to $A_{OUT}(n_{pred})$ by the rule. Because both *LHS* and *RHS* are qualified expressions, *AN(a)* and *AN(b)* can be computed as follows:

$$AN(a) = \{o_1.f\}$$
$$AN(b) = \{o_3.g, o_4.g\}$$

When $A_{o_4.g} = \phi$, $A_{rhs} = <o_4, null>$. Thus, $A_{KILL}(n)$ and $A_{GEN}(n)$ can be computed as follows:

$$A_{KILL}(n) = \{<o_2>_{o_1.f}\}$$
$$A_{GEN}(n) = \{<o_4, null>_{o_1.f}\}$$

The rule for a return statement node type is presented as follows with a virtual return reference variable *rr*, whose referred-set is used to analyze an assignment of a return value in a calling method.

$$
\frac{\begin{array}{l} A_{IN}(n) = A_{OUT}(n_{pred}), \\ RO = a \wedge A_a \in A_{IN}(n) \rightarrow A_{rr} = A_a, \\ RO = \text{new } T \rightarrow A_{rr} = < o_{new} > \end{array}}{\begin{array}{l} A_{KILL}(n) = \phi, \\ A_{GEN}(n) = \{A_{rr} \mid RO = a \vee RO = \text{new } T\} \end{array}}
$$

In this rule, *RO* stands for an object returned on a return statement. When the name *a* of a return object is a qualified expression, entire objects pointed to by its primary expression should be considered as in an assignment statement node. But it is not considered here for simplicity.

Next is a rule for an exit node type.

$$
\frac{\begin{array}{l} PRED(n) = \{p \mid p \text{ is a predecessor node of } n\}, \\ A_{IN}(n) = \bigcup_{p \in PRED(n)} A_{OUT}(p), \\ LOCAL(M) = \{v \mid v \text{ is a local variable of } M\} \end{array}}{\begin{array}{l} A_{KILL}(n) = \{A_v \in A_{IN}(n) \mid v \in LOCAL(M)\}, \\ A_{GEN}(n) = \phi \end{array}}
$$

There might be direct edges from return statement nodes to an exit node so that it can have several predecessor nodes. *LOCAL* stands for a set of local variables defined in a called method *M,* including formal parameter variables.

4.2. Rules for interprocedural analysis

Interprocedural propagation rules should be considered for a call statement node and an entry node. The data flow of an alias set in a call statement denotes that an alias information of the statement is propagated to a called method and it affects an alias information of the called method. After alias sets of the called method are computed, the result alias set is passed back to the call statement of the calling method. The set from the called method modifies the alias set of the call statement when a return value is assigned.

We divide a call node into a *precall* node and a *postcall* node virtually to simplify the analysis of a call statement. A precall node receives an alias set from a predecessor node of a current call node and computes its own alias set $A_{OUT}(n)$. This alias set is propagated to an entry node of a called method. During the propagation, the referred-sets for references inaccessible from the called method are killed. Since this kill set is an input of a postcall node and cannot be accessed by the called method, it does not need to be propagated. $A_{OUT}(n)$ of a precall node is not transferred to a postcall node directly because the called method might modify the set. In previous approaches [1, 2, 5], when computing a result alias set of a call node, alias relations killed in a called method can be used, and an imprecise alias set can be propagated to subsequent nodes. This propagation might make subsequent alias relations overestimated, and generate nonexistent call relations. Thus, those approaches compute redundant alias relations and make an alias analysis inefficient.

A postcall node collects a kill set of a precall node and alias sets of exit nodes in all possible called methods. Referred-sets of references accessible in a calling method are selected, and the remains are killed. If a return value of a called method is assigned to a variable, a kill and a gen set are computed by the same rule as an assignment statement node. We can compute a result alias set of a post call node, and it becomes an alias set of a call node.

The following rule defines a kill and a gen set of a precall node.

$$
\frac{\begin{array}{l} A_{IN}(n) = A_{OUT}(n_{pred}), \\ RHS = E_c.M_c \\ \quad \rightarrow ON(E_c) = \{o \mid o \text{ is an object name referred by } E_c\} \\ \qquad \wedge NLOCAL(M) = \phi, \\ RHS = M_c \\ \quad \rightarrow ON(E_c) = \phi \\ \qquad \wedge NLOCAL(M) = \{v \mid v \text{ is a nonlocal variable in } M\}, \\ \forall i \quad a_i = \text{the } i\text{th actual parameter}, \\ \forall i \quad f_i = \text{the } i\text{th formal parameter}, \\ \forall i \quad A_{a_i} \in A_{IN}(n) \\ \quad \rightarrow A_{f_i} = A_{a_i} \wedge PASS(M) = PASS(M) \cup A_{a_i}, \\ \forall d \, \forall o \in ON(E_c) \quad A_{o.d} \in A_{IN}(n) \\ \quad \rightarrow A_d = \bigcup_{q.d \in A_{IN}(n)} A_{q.d} \wedge PASS(M) = PASS(M) \cup A_d, \\ \forall v \in NLOCAL(M) \quad PASS(M) = PASS(M) \cup A_v, \\ \forall d \, \forall o \in PASS(M) \quad A_{o.d} \in A_{IN}(n) \\ \quad \rightarrow PASS(M) = PASS(M) \cup A_{o.d}, \end{array}}{\begin{array}{l} A_{KILL}(n) = \{A_{k.d} \mid k \notin PASS(M)\} \\ \qquad \cup \{A_v \mid v \notin NLOCAL(M)\}, \\ A_{GEN}(n) = \{A_{f_i} \mid A_{a_i} \in A_{IN}(n)\} \\ \qquad \cup \{A_d \mid \exists o \;\; o \in ON(E_c) \wedge A_{o.d} \in A_{IN}(n)\} \end{array}}
$$

NLOCAL(M) represents a set of non-local variables in a calling method *M*. $ON(E_c)$ is a set of object names that can be accessed by a primary expression E_c. *PASS(M)* is a set of all object names that are passed to a callee. The set is defined recursively from object names in a new referred-set that is propagated to the callee. In the above rule, an actual parameter with a form of qualified expression is not considered for simplicity. The similar

computation as an assignment statement should be performed to find a set of object names accessed by a corresponding formal parameter.

The following rule is for an entry node.

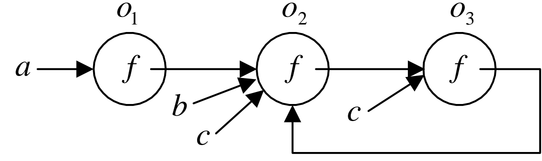
$$\frac{\begin{array}{l} PRECALL(M_c) \\ \quad = \{ p \mid p \text{ is a precall node of the statement calling } M_c \}, \\ A_{IN}(n) = \bigcup_{p \in PRECALL(M_c)} A_{OUT}(p) \end{array}}{\begin{array}{l} A_{KILL}(n) = \phi, \\ A_{GEN}(n) = \phi \end{array}}$$

$PRECALL(M_c)$ is a set of precall nodes for call statement nodes that invoke a called method M_c containing this node. This set can be computed by considering ingoing edges of the called method in a *CG*. An entry node merges alias sets from all precall nodes and then propagates the merged set to its subsequent node.

The rule of the postcall node is defined as follows:

$$\frac{\begin{array}{l} EXIT(M) = \{ e \mid e \text{ is an exit node of a callee from } M \}, \\ A_{IN}(n) = A_{KILL}(n_{precall}) \cup \bigcup_{e \in EXIT(M)} A_{OUT}(e), \\ RHS = E_c.M_c \\ \quad \rightarrow FIELD(E_c) \\ \qquad = \{ f \mid f \text{ is a field name in an object referred by } E_c \}, \\ RHS = M_c \rightarrow FIELD(E_c) = \phi, \\ RHS = \text{new } M_c \rightarrow FIELD(E_c) = \phi \wedge A_{rr} = < o_{new} >, \\ ON(E_c) = ON(E_c) \text{ in the precall node}, \\ PASS(M) = PASS(M) \text{ in the precall node}, \\ \forall f \in FIELD(E_c) \;\; \forall o \in ON(E_c) \quad A_f \in A_{IN}(n) \\ \quad \rightarrow A_{o.f} = A_f \wedge PASSB(M) = PASSB(M) \cup A_f, \\ \forall d \; \forall o \, (o \in PASS(M) \vee o \in PASSB(M)) \wedge A_{o.d} \in A_{IN}(n), \\ \quad \rightarrow PASSB(M) = PASSB(M) \cup A_{o.d}, \\ \forall v \quad v \notin FIELD(E_c) \wedge A_v \in A_{IN}(n) \\ \quad \rightarrow PASSB(M) = PASSB(M) \cup A_v, \end{array}}{\begin{array}{l} A_{KILL}(n) = \{ A_a \mid a = LHS \} \cup \{ A_{rr} \} \\ \qquad \cup \{ A_f \mid f \in FIELD(E_c) \} \\ \qquad \cup \{ A_{k.d} \mid k \notin PASS(M) \wedge k \notin PASSB(M) \} \\ \qquad \cup \{ A_{o.d} \mid o \in ON(E_c) \}, \\ A_{GEN}(n) = \{ A_a / A_{rr} \mid LHS = a \\ \qquad \wedge (A_{rr} \in A_{IN}(n) \vee A_{rr} = < o_{new} >) \} \\ \qquad \cup \{ A_{o.f} \mid A_f \in A_{IN}(n) \\ \qquad \wedge f \in FIELD(E_c) \wedge o \in ON(E_c) \} \end{array}}$$

$EXIT(M)$ is a set of exit nodes of all possible called methods as explained before. We can compute $EXIT(M)$ from a *CG* by collecting all outgoing edges from a current call node in a caller *M*. $FIELD(E_c)$ represents a set of field variables defined in classes of objects accessed by E_c. $PASSB(M)$ is a set of objects accessible in a caller after passed back from a called method. Thus, referred-sets related to objects that do not belong to the $PASSB(M)$ should be killed.



(a)

```
...
a.update(c);
...

void update(T i) {
        f = i.f;
}
```

(b)

Figure 4. Example of an interprocedural alias analysis

Figure 4 is an example to compute alias set with the above rules. An alias set in a state of Figure 4(a) is:

$$\begin{array}{l} A_{OUT}(n) = \{ < o_1 >_a, < o_2 >_b, < o_2, o_3 >_c, \\ \qquad < o_2 >_{o_1.f}, < o_3 >_{o_2.f}, < o_2 >_{o_3.f} \} \end{array}$$

When a method invocation of Figure 4 (b) is occurred in this state, a result alias set of a precall node can be computed in the following sequence of rule applications:

$$\begin{array}{l} A_{IN}(n) = \{ < o_1 >_a, < o_2 >_b, < o_2, o_3 >_c, \\ \qquad < o_2 >_{o_1.f}, < o_3 >_{o_2.f}, < o_2 >_{o_3.f} \} \\ ON(E_c) = \{ o_1 \}, \; NLOCAL(M) = \phi \\ A_i = A_c = < o_2, o_3 >, \; A_f = < o_2 > \\ PASS(M) = \{ o_2, o_3 \} \\ A_{KILL}(n) = \{ A_{o_2.f}, A_a, A_b, A_c \} \\ A_{GEN}(n) = \{ A_i, A_f \} \\ A_{OUT}(n) = \{ A_{o_2.f}, A_{o_3.f}, A_i, A_f \} \end{array}$$

```
Algorithm AliasAnalysis
construct an initial CG with main method;
repeat {
    for each method T.M ∈ N_CG ,
            alternating between topological and reverse topological order {
        for each node n ∈ N_CFG_T.M in structural order {
            if n is a call statement node {
                if RHS = E_c.M_c {
                    compute the set of inferred types from the referred-set for E_c ;
                    compute the set TYPES of 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} ;
                }
                for each type t ∈ TYPES {
                    if t.M_c is not in CG
                        create a CG node for t.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 A_OUT(n_precall) for a precall node n_precall ;
                compute A_OUT(n_postcall) for a postcall node n_postcall ;
            } else
                compute A_OUT(n) using data-flow equation and propagation rule;
            }
        }
    }
} until CG and alias set for every CFG node converge
```

Figure 5. Flow-sensitive alias analysis algorithm

$A_{OUT}(n)$ of a precall node propagates to an entry node of a called method, *update()*. After performing an intraprocedural alias analysis of the method *update()*, we can compute a result alias set of an exit node as follows:

$$A_{OUT}(n) = \{<o_2, o_3>_f, <o_3>_{o_2.f}, <o_2>_{o_3.f}\}$$

A result set of a postcall node or a call node is computed by applying the rule for a postcall node as follows:

$$A_{IN}(n) = \{<o_1>_a, <o_2>_b, <o_2, o_3>_c, <o_2>_{o_1.f}, <o_2, o_3>_f, <o_3>_{o_2.f}, <o_2>_{o_3.f}\}$$

$$FIELD(E_c) = \{f\},\ ON(E_c) = \{o_1\},\ PASS(M) = \{o_2, o_3\}$$

$$A_{o_1.f} = A_f = <o_2, o_3>$$

$$PASSB(M) = \{o_2, o_3\}$$

$$A_{KILL}(n) = \{A_{o_2.f}, A_f\}$$

$$A_{GEN}(n) = \{A_{o_2.f}\}$$

$$A_{OUT}(n) = \{<o_1>_a, <o_2>_b, <o_2, o_3>_c, <o_2, o_3>_{o_1.f}, <o_3>_{o_2.f}, <o_2>_{o_3.f}\}$$

5. Alias analysis algorithm

Figure 5 shows our alias analysis algorithm. The frame of this algorithm is similar to the interprocedural type analysis algorithm [5]. It is an iterative method used in an interprocedural data flow analysis with a *CG* [9]. The iterative algorithm continues to execute its computation by visiting all nodes of a *CG* until a fixed point of the program status is arrived.

In order to possibly save the execution time to arrive at a fixed point, our algorithm traverses each node of a *CG* in a topological order and a reverse topological order

[4, 5, 6]. The fixed point in our algorithm means that the topology of a *CG* and an alias set of each *CFG* node are not changed anymore. The algorithm terminates its computation at that moment.

An inner loop in the algorithm performs an intraprocedural alias analysis for each method. In the inner loop, each node of a *CFG* for the method is traversed with computing its alias set, and the computed alias set is propagated to the next node. The traversing order is important to update alias sets of whole nodes in one pass. The algorithm visits each node in a structural order to achieve this purpose. The visit is performed in an order from an entry node to an exit node. When a flow construct node for an *if* statement is traversed, its all branches should be traversed before its merging node is visited. We improve the efficiency of an intraprocedural analysis with the structural order in comparison with a fixed point method of a previous work [5]

When a method of a call statement node is an overridden method, its resolution should be considered for the safety of alias sets. It is impossible to precisely resolve a dynamic overridden method in a static analysis. However, we can collect all possible objects referred to by a primary expression to qualify a called method, and then retrieve a type of each object from the type table. Thus, we can safely predict all possible methods invoked. If the method is found in a class of a type inferred by this type inference, it becomes a resolved method. If not, a super class of the type is searched to find the method. Algorithm updates a *CG* with these resolved methods.

While proceeding the algorithm, aliased objects for a reference are increased, and then resolved methods might be increased. Therefore, the *CG* is built incrementally during the execution of the algorithm. But it will converge finally because the number of aliased objects is not incremented infinitely.

6. Complexity of algorithm

Our algorithm consists of three nested loops. We first calculate the number of iterations of the most outer loop. As defined in section 2.2, O_t and A_o are the number of objects and the maximum number of aliased references for an object, respectively. $O_t \times A_o$ means the maximum number of refer-to relations between references and objects existing in each node of a *CFG*. We can estimate the worst time complexity as $O(O_t \times A_o \times E_{cg})$, where E_{cg} is the number of edges in a *CG*, since each relation is transferred from an entry node to an exit node one by one in an iteration.

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

The most dominant part in the inner loop is to process a call statement node so that its worst time complexity depends on the complexity of a call statement. The time complexity for computing a set of inferred types is $O(p \times O_r \times R + O_r)$, when R is the maximum number of references accessible from an arbitrary point of a program; O_r is the maximum number of may-aliased objects; p is a small constant determined on the depth of a primary part in a qualified expression.

The time complexity for resolving possible methods is $O(T_i \times H)$, when T_i is the maximum number of sub classes for a class and H is the maximum number of levels in hierarchy among classes. The time complexity for the resolution of overridden methods and the updating of a *CG* is $O(O_r \times R + T_i \times (H + N_{cg} + C_c))$, 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 node is $O(O_p \times R)$, respectively, when O_p is the maximum number of objects propagated to a called method.

Therefore, the worst case time complexity of the whole algorithm becomes $O(O_t \times A_o \times E_{cg} \times N_{cg} \times N_{cfg} \times (R \times (O_r \times O_p) + T_i \times (H + N_{cg} + C_c)))$.

The worst space complexity for storing alias sets becomes $O(N_{cg} \times N_{cfg} \times O_t \times A_o)$, and the space complexity for a type table is $O(O_t)$. The worst space complexity of outgoing edges for a call statement is $O(T_i \times H)$, and thus the worst space complexity of a *CG* is $O(N_{cg} \times C_s \times T_i \times H)$, when C_s is the maximum number of call statements in a method. But the space complexity of a *CG* is practically $O(N_{cg} + E_{cg})$ that is usually much smaller than the worst space complexity.

7. Conclusion

Java is a popular object-oriented language for an Internet world and it might be applicable for high-performance computing such as parallel processing. Aliases cause race conditions, context switching overheads, and communication overheads in parallelized programs especially. Because references only are used to point to objects and assignments among references generate hidden aliases in Java, we need to detect aliases carefully.

In this paper, we proposed a flow-sensitive alias analysis algorithm by adapting existing alias analyses [5, 6] for C or C++ to Java. The algorithm becomes more precise and efficient than previous works, by introducing

a referred-set representation and data propagation rules based on the representation.

The referred-set representation is more efficient for Java, while the conventional alias representations have been built for pointer-using languages such as C or C++. It makes a type inference and an alias propagation efficient especially.

We classified *CFG* node types in order to describe a propagation rule for each type in detail. The rule is used to compute a kill and a gen set at each node. The result alias set of each node is computed from these sets by applying an alias data flow equation. The interprocedural rules enhance the preciseness of our algorithm by excluding redundant alias relations in conventional approaches. We also improved the efficiency of our algorithm by traversing a *CFG* in a structural order.

References

[1] Hemant D. Pande and Barbara G. Ryder, *Static Type Determination and Aliasing for C++*, Technical Report LCSR-TR-236, Rutgers University, December 1994.

[2] Hemant D. Pande and Barbara G. Ryder, *Static Type Determination and Aliasing for C++*, Technical Report LCSR-TR-250-A, Rutgers University, October 1995.

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

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

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

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

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

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

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

[10] D. Diwan, K. S. McKinley, and J. E. B. Moss, "Type-Based Alias Analysis", *SIGPLAN '98 Conference on Programming Language Design and Implementation*, 106-117, 1998.

[11] R. Rugina and M. Rinard, "Pointer Analysis for Multithreaded Programs", *Proc. the ACM SIGPLAN 1999 Conference on Programming Languages and Implementation*, May 1999.

[12] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley, 1996.