

Alias Analysis On Type Inference For Class Hierarchy In Java

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

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

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

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

Abstract

The integration of alias analysis with type information increases the precision of alias detection, especially for inheritance among classes. This paper presents a compile-time flow-sensitive context-insensitive alias analysis algorithm with type information for Java. First, we propose an aliased element representation for an object to compute aliases efficiently. Second, the algorithm computes aliases for shadowed variables by regarding constructors as functions. Third, it performs type inference for each reference variable. The inferred type information increases the precision of subsequent alias analysis by building a complete calling graph, not only for overridden methods but also for both shadowed variables and constructors. Fourth, it presents algorithms to compute aliases for each statement. As a result, the precision and efficiency of the algorithm is improved.

1. Introduction

An alias is defined as two or more reference/pointer variables that point at the same memory location. Aliasing complicates data-flow analysis for compiler optimization and parallel computing.

As a static analysis, alias analysis has been developed to be used in reordering instructions and enhancing performance. Also, it helps exploiting instruction-level parallelism statically. It is also possible to avoid race conditions, context switch and communication overhead by allocating the aliased data locally for distributed computing. Aliases are detected by *intra-procedural* and *inter-procedural* analysis: *intra-procedural* analysis computes aliases of each statement in a procedure using a control flow graph (CFG); *inter-procedural* analysis computes aliases among procedures using a calling graph (CG) [3, 4, 5, 11, 13, 16, 17].

For the object-oriented languages C++ and Java, a key static analysis is to infer possible types of objects in the presence of dynamically-typed virtual functions. For example, the static determination of possible types of an object 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 [2, 4, 6, 10, 15, 17].

The integration of alias analysis and type inference in our proposed algorithm allows determination of the static type of an object in the presence of shadowed variables and overridden methods during alias analysis in Java. It improves the precision of inter-procedural analysis as shown in Figure 12 without negatively

```

class Shape {
    TmpObj r = new TmpObj(10);
    double area(){
        1) return (r.r * r.r);
    }
}

class Circle extends Shape {
    TmpObj r =
        new TmpObj("test", 2);
    int x, y; // location
    double area(){
        1) return (3.14 * r.r);
    }
}

class Foo {
    public static void main(String args[]){
        Circle c = new Circle();
        TmpObj t;

        1) ((Shape)c).area();
        2) t = ((Shape)c).r; // implies function
        3) t.fn();
    }
}

class TmpObj {
    int r;
    String str;
    TmpObj(int i){
        1) r = i;
    }

    TmpObj(String s, int i){
        1) r = i * i;
        2) str = s;
    }

    void fn(){
        ...
    }
}

```

Figure 1. An Example Java Code

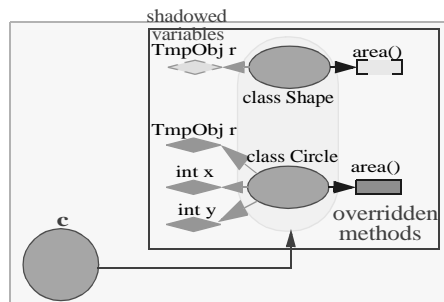
affecting the efficiency of the analysis. The integration was applied for our previous work [17] that presented *referred-set* alias representation for better precision and performance of the analysis.

The rest of this paper is organized as follows. Section 2 presents problem statement of alias analysis with type information in object-oriented language and describes related works. Section 3 proposes our alias analysis algorithm on type inference. Section 4 shows a framework of our alias analysis system and an example which illustrates the precision of our algorithm. Section 5 presents conclusions.

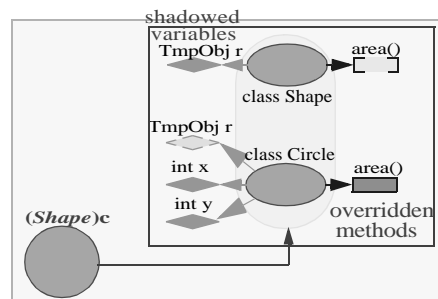
2. Problem Statement

2.1. Alias Analysis for Object-Oriented Language

In order to show several properties of an object-oriented language, Figure 1 presents Java classes with overriding attributes and overloading constructors. Class **Circle** is a subclass of class **Shape** and **Circle** includes an overridden method **area()** and a variable **r** with new value of the type **TmpObj**, one of overloading constructors. Those overriding attributes need to be considered in building **CG** in order to compute safe alias set because the computed results will be different depending on which methods and variables are analyzed. But previous work [4, 5, 8, 16] has considered only virtual function in C++ for their alias analysis algorithms so that it would build incomplete **CG**.



Circle c = new Circle();
(a) the initial creation of an object c



(b) the assigning of objects with casting types

Figure 2. Shadowed variables and Overridden methods of Figure 1

This section focuses on three properties of Java for alias analysis because these will affect the efficiency and the precision of the alias computation. First, a class includes object reference variables and objects instantiated. Second, types of a reference variable are dynamic because of its polymorphism. Thus, type information is needed since (1) an object has its own class type, (2) an object accesses its attributes which are declared as other class types, and (3) a class has inheritance (hierarchy). Using hierarchy, a subclass inherits all attributes of a superclass even though such attributes are not defined within the subclass. Finally, there exist shadowed variables and overridden methods in Java; shadowed variable is a variable defined in one class with same named variable of its superclass; overridden method is a method defined in one class with the same named method and same argument types of its superclass.

Figure 2 shows shadowed variables and overridden methods of the example code in Figure 1. An object **c** is initially declared as a class type **Circle**, as in Figure 2 (a), but it is type-casted by its superclass **Shape**, as in Figure 2 (b). Class **Shape** and its subclass **Circle** both have a field name **r** and a method named **area**. After the type casting in Figure 2 (b), the object reference **(Shape)c** refers to the field **r** of a class type **Shape** and to the method **area** of a class type **Circle**; the first is termed *shadowed variable* and the latter termed *overridden method*, that complicate calling graph construction.

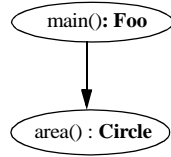


Figure 3. CG of Figure 1 without constructors for overridden methods only (Incomplete)

Alias Analysis algorithm requires a trade off between precision and efficiency because the detecting of more precise alias set will increase computing time; precise aliases can be defined as aliases which must occur at the statements along the execution path of a code; safe aliases can be defined as aliases which are the super set of the precise aliases along the path but computed approximately [3, 4, 5, 11, 13, 16, 17].

2.2. Related Work

Previous work [4, 5, 16] has integrated alias analysis with type information in order to construct calling graph more precisely for virtual functions in C++. They determined the type of an object which invokes virtual function by analyzing the object creation statement and assignment statement. Then, they integrate their alias analysis algorithm with the CG built by the type information of objects. However, as shown in Figure 3, their CG becomes incomplete because they have only considered the type of an object which invoked overridden methods but not shadowed variables. We suggest our type inference operation to compute more precise aliases for both shadowed variables and overridden methods in Figure 6.

Also, their aliased elements are the binding of an object and its naming that will produce many redundant aliased elements in Java and cause huge time complexity because objects are theoretically infinite so that it would increase the alias set and because its dynamic type look-up for a reference variable is indirect for each object in their type tables.

Pande [16] presented the first algorithm which solved type determinations and pointer aliases simultaneously in C++ programs. **Carini** [4] enhanced the flow-sensitive pointer alias analysis algorithm by computing both type information and the pointer alias computation within the same data flow contexts. **Chatterjee** [5] presented a flow-sensitive conditional points-to analysis in C++.

3. Alias Analysis on Type Inference

3.1. Terminology

This section defines terminologies used in this paper.

LHS and RHS reference variables

Basic elements of aliased element. **LHS** and **RHS** are reference variables to an object in an assignment statement; **LHS = RHS**. Right Hand Side (**RHS**) is a reference or an object created. Left Hand Side (**LHS**) is a reference variable to Right Hand Side (**RHS**) reference/object.

Type information of reference variables

LHS and **RHS** reference variables need type information. Type information consists of **class type**, **field type**, and **method type**. **Class type** is the class type by which a reference variable is initially declared. **Field type** stands for the class type of a shadowed variable. **Method type** represents the type of an overridden method.

Aliased Element

Basic element of an Alias Set. It consists of two object reference variables: (**LHS**, **RHS**) which refers to the same object.

Alias Set

A set which consists of several aliased elements.

Statement

Holds an alias set. It mainly consists of two statements: assignment statement and call statement. For each statement, we can compute two alias sets **A_{out}** and **A_{in}**.

A_{in}: alias set before the statement

A_{out}: alias set after the statement

Transfer Function

Computes alias set. **A_{in}** becomes input of transfer function **trans_s** then it computes the **A_{out}** of a statement **s**. For this statement, some aliased elements are generated and others killed.

A_{out} = **trans_s(A_{in})**

Kill(s)

Aliased elements killed at a statement **s** because **LHS** refers to another **RHS**.

Gen(s)

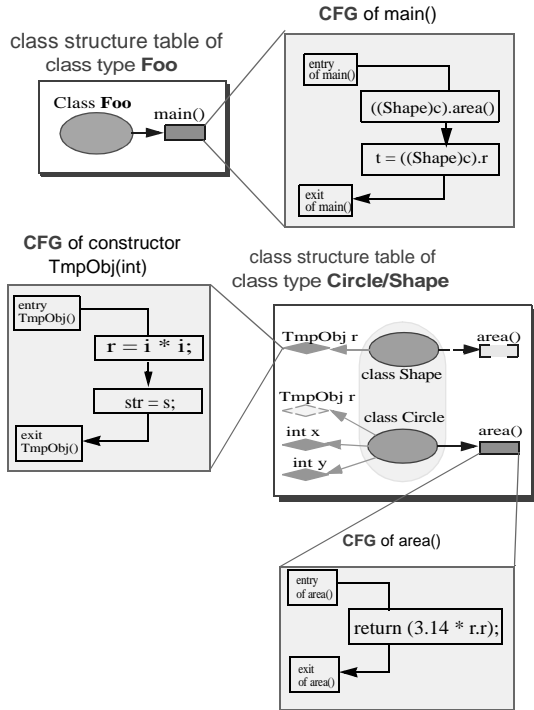
Aliased elements generated at a statement **s**.

GenNew(s)

As a statement **s** generates aliased element (**LHS**, **RHS**), it also generates new aliased elements by combining with other aliased elements using a **transitive closure** rule. If a statement generates aliased element (**P**, **Q**) and **A_{in}** contains elements (**Q**, **x**), it causes another aliased element (**P**, **x**).

3.2. Class Structure Table

In order to apply our alias analysis algorithm for Java, we construct CFGs, a CG, and type tables (**TT**). Each procedure can be translated to **CFG** = (**P**, **entry**, **exit**, **N**, **E**) during parsing. **CFG** is the basic structure on which aliases are computed intra-procedurally on assignment



Type Table for the objects of main()

Object name	initial type	type of shadowed field	type of overridden method
c	Circle	Circle	Circle
s	Shape	NA	NA
(Shape) c	NA	Shape	Circle
t	TmpObj	NA	NA

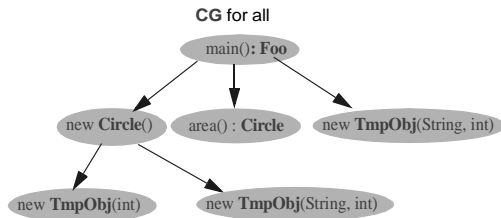


Figure 4. Example of Class Structure Table

and call statements; **P** represents a procedure itself; **Entry** is the first node of the CFG for procedure **P**; **Entry** node collects the alias set passed from the calling procedure for global variables and parameters while **R** is a called procedure; **Exit** is the last node which passes an alias set to calling procedure **Q** after computing alias set of called procedure **R** while **R** is a called procedure and **Q** is a calling procedure of the procedure **R**. **N** is a node which stands for a statement within procedure **P**. It contains the alias set computation for the statement. **E** is an edge which connects nodes and holds aliased set

```

class AliasSet {
    Name nameLHS;
    Name nameRHS;
}

class FieldTable {
    String name;
    ClassStructure fieldType;
    int scope;
}

class CallingGraphNode {
    Vector parent;
    Vector child;
    //for invoking a method of other class
    String className;
    String methodName;
    ClassStructure classType;
    Vector actualParameter;
}

class Node {
    Node parent;
    Node child;
    Vector aliasSet;
}

class ClassStructure {
    String className;
    Vector super, sub;
    Vector field;
    //collection of CFG
    Vector method;
}

class Statement {
    Name nameLHS;
    //if exists
    Name nameRHS;
}

class ControlFlowGraphNode {
    String methodName;
    String returnType;
    Vector formalParameter;
    Vector variable;
    // node for entry, exit,
    // and each statement
    Vector Node;
    // for call statement
    CallingGraphNode CG;
}

class Name {
    String className;
    String castType;
    ClassStructure classType;
    ClassStructure fieldType;
    Vector methodType;
    // field or method invoked
    Name invokedName;
}

```

Figure 5. Class Structure Table

data. Our CFG is classified as *flow-sensitive* computing problem which computes safe alias set; *flow-sensitive* problem is to examine the body of a procedure with regard to individual control flow paths of the body.

Calling Graph (CG) shows interactions among CFGs from calling procedure to called procedures. In an object-oriented language, **Type Table**, which presents dynamic type of each object, is needed for inter-procedural analysis. **Type Table** collects a reference variable and its possible class types in order to construct safe CG so that subsequent analysis should be safe, specially for overridden methods and shadowed variables. Its possible types can be collected with our algorithm in Figure 6. **Type Table** consists of four elements: object reference variable, its original class type, shadowed variable type, and overridden method type as in Figure 4.

Finally, Class Structure Table (CST) is proposed in order to represent all of CFG, CG, and Type Table. We can copy attributes (data and operations on it) of each class to CST; In Java, data is a named field and operation on the data is a named method.

```

/*
 * infers type of Shadowed Variables or Overridden Methods for inherited object references.
 * LHS: assigned statement, RHS: assignment statement
 * application example:
 * a is a class type A; b is a class type B which is a subclass of A
 * case 1: a = b; a = (A)b; a = (B)b; a = new B(); // assignment
 * case 2: invoking instances with (A)b; // type casting for invoking
 */
TypeInference(name LHS, name RHS) {
  if(RHS.invokedName != NULL){ // case 2
    // Shadowed Variables;
    // all of fields in (A)b refer to fields of type of A;
    if(RHS has cast type expression) RHS.fieldType is assigned
    by the cast type;
    LHS.fieldType = RHS.fieldType; // when LHS exists

    // Overridden Methods; all of methods in (A)b refer to methods
    of type of b; when LHS exists
    LHS.methodType = RHS.methodType = RHS.classType;
  } else if (LHS exists){
    if((RHS.invokedName is NULL)
    AND (LHS.classType is a superclass of RHS.classType))
    OR (LHS is declared with 'new its superclass type') {
      // Shadowed Variables;
      // all of fields in a refer to fields of type of a;
      LHS.fieldType = LHS.classType;

      // Overridden Methods; all of methods in a refer to methods
      of type of b;
      LHS.methodType = RHS.methodType;
    }
  } // end of else-if statement
}

```

Figure 6. Type Inference for inherited references

Figure 4 presents a CST for the example code of Figure 1. In addition to storing all the information of an example code in CST, we can save space not by allocating memory space for all objects: we represent the (potentially infinite) space of all objects instantiable at run-time from a class, by using just one CST associated with the class itself.

Figure 5 represents the data structure of CST. Class *ClassStructure* represents CST. Class *CallingGraphNode* is to build a CG with parents calling this procedure, children called by this procedure, and actual parameters for each called procedure. Class *ControlFlowGraphNode* contains entry, exit, and links of A_{in} and A_{out} nodes to build a CFG. Each node has an alias set for a statement. Class *AliasSet* consists of aliased element LHS and RHS which refers the same object. This representation is more economical than the object naming pairs representation of the previous work [3, 4, 5, 11, 13, 16]. Class *Name* is to build a type table and contains all possible type information.

3.3. Alias Analysis Algorithm

Alias analysis algorithm handles two statements: assignment and call statement. As type information is needed for each object in order to detect safe paths of calling graph specially for shadowed variables and overridden methods, **TypeInference** operation is presented in Figure 6. By inferring the relationships within a statement,

it will decide two types of an object, field type of shadowed variable and method types of overridden method in addition to original class type of the object. Now we have *lemma 1* and it is proven as follows.

Lemma 1. *If each type of reference variables a and b is safe, the type information computed by algorithm **TypeInference**(a, b) for the cast type of a by b is safe.*

Proof: Suppose that a reference a has a class type A and a reference b has a class type B which is a subclass of A . The type of each reference is initially decided at compile time. When a is assigned by b or memory-allocated by class type B , type information of a for its variables and methods is decided respectively at run-time by the class type B according to the syntax of Java. Also, a cast object $(A)b$ decides the type of its shadowed variables as type A according to the syntax of Java. Thus, type information computed by algorithm **TypeInference**(a, b) is safe. ■

While building CG, type information of a reference in TT determines all possible callees for safety. Also, the type information makes the constructing of CG precise by considering not only the type of overridden methods but also the type of shadowed variables and constructors. When there are call statements of possible callee(s) P within a method Q (a caller), the CG should be constructed by connecting an edge(s) from node Q to node(s) P . Aliases are computed by passing an alias set of call statement Q to the entry alias set of P and by returning the exit alias set of P to call statement of Q . We have defined the construction (in a straight forward manner) of the CG in a function *ConstructCallingGraph*($m, currCG, c$) with parameters; c : class type which contains calling procedure P , $currCG$: current calling graph, and m : callee of caller P . Callee m has its declared class type and run-time class type T inferred by **TypeInference** algorithm so that the algorithm will add a safe edge(s) from caller P to the callee m . For the callee m with type T inferred by our **TypeInference** algorithm, an edge(s) from caller P to the callee m should be safely connected because *lemma 1* shows that the class type T should be safe.

As in Figure 7, during execution of our algorithm, if it meets an assignment statement, new alias set might be generated and then some members of the alias set should be killed. The generated alias set should have its own **type table** which is filled with types of references according to type inference operation of Figure 6.

Lemma 2. *For the assignment statement $a = b$ of reference variables a and b , **assignmentStatement**(a, b, s, c) computes a safe alias set (s : a node which is the statement, c : class which contains the statement).*

```

/*
 * if a statement within a method is assignment statement
 * LHS and RHS: assignment statements,
 * statement: current statement,
 * typeltself: class type which contains current method
 * and this statement
 */
assignmentStatement(Name LHS, Name RHS, Node statement,
                    CallingGraph CG, ClassStructure typeltself) {
    // decide type of LHS based on RHS
    TypeInference(LHS, RHS);

    // union aliasSets of parent statements;
    Ain = Union(statement.parent);
    // Cother is other class type - not equals to typeltself
    if(LHS or RHS refers to the class type Cother){
        refers to the class structure of type Cother;
    }
    Aout = Ain + { ( LHS, RHS ) } - Kill; // transitive closure rule (TC)
    statement.child.aliasSet = Aout;
}

```

Figure 7. Alias Analysis for assignment statement

```

/*
 * bind the formal and actual parameters
 * type should be decided for them
 * RHS: method at calling site
 */
Vector bind(Name RHS) {
    formalParameters = RHS.methodType.formalParameters;
    actualParameters = RHS.callingGraph.actualParameters;

    // decide the type of parameters if inherited class types
    if(formalParameters exists and type of formalParameters
       and actualParameters is different) {
        typeInference(actualParameters, formalParameters);
    }
    for each element i of formalParameters {
        aliasSet.addElement( {{formalParameters.elementAt(i),
                               actualParameters.elementAt(i)}});
    }
    return aliasSet;
}

```

Figure 8. Binding of formal and actual parameters

Proof: As shown in Figure 7, A_{in} is a union of all parent sets of s . **Kill** removes aliased element $\{(a, x) \mid x: \text{any reference}\}$ in A_{in} . Therefore, A_{out} that is computed from A_{in} , **Kill**, and (a, b) is safe. ■

Figure 8 shows how to bind formal parameters and actual parameters for a called procedure and a calling procedure. Figure 9 presents the operation of computing aliases on a call statement. If the call statement has return value, a type decision operation should be applied since it might imply inheritance. If the called procedure is an attribute of other classes that is named C_{other} as a qualified expression of $r.m$ when reference r has a type C_{other} and m is one of its procedures, **CST** of the other class C_{other} will be referred to. Also, **CG** is constructed on each call statement during the loop of the algorithm in order to detect aliases subsequently. The data flow of an alias set (A_{out}) in a *call* statement is computed with an alias information of the statement propagated to a called method (A_{exit} of **RHS**). Also, the alias set from

```

/*
 * if a statement within a procedure is call statement
 * LHS: assigned statement, RHS: called method statement,
 * statement: current statement,
 * typeltself: class type which contains current procedure
 * and this statement
 */
callStatement(Name LHS, Name RHS, Node statement,
              CallingGraph CG, ClassStructure typeltself) {
    // decide type of LHS based on RHS
    if ((LHS exists and types of LHS and RHS are different)
        OR (LHS does not exist - null- and RHS is cast))
        TypeInference(LHS, RHS);
    // union aliasSets of parent statements;
    Ain = Union(statement.parent);
    // Cother is other class type - not equal to typeltself
    if(LHS or RHS refers to the class type Cother) {
        refers to the alias information of the method's Aexit of Cother;
    }
    if(Calling Graph of the called procedure is not constructed) {
        CG = constructCallingGraph(RHS, CG, typeltself);
    }
    if(LHS exists) // Aout = statement.child.aliasSet;
        Aout = Ain + { (LHS, return of RHS) } + Aexit of RHS
              + bind(parameters of RHS) - Kill; // TC
    else Aout = Ain + Aexit of RHS + bind(parameters of RHS) - Kill; // TC
    statement.child.aliasSet = Aout;
}

```

Figure 9. Alias Analysis for call statement

the called method modifies the alias set of the *call* statement when the return alias set includes non-local variables and actual parameters that represent **bind**(parameters of **RHS**) operation.

Figure 10 shows our alias analysis algorithm on type inference by adapting **Carini** [4]'s to our **CST** and type inference procedure in Java. It loops until alias set and **CG** converge. There are **for** loops on each procedure within and among the classes. It invokes either the operation **assignment statement** or **call statement** depending on which statement exists at the node of the CFG of a procedure.

The computation of an entry alias set Q_{entry} for called procedure Q can be modeled by a data-flow equation as follows. Let $A_{in}(c)$ be the input alias set of the call statement c . The effect of a node n can be computed by the following equation:

$$Q_{entry} = A_{in}(c) + gen(c) - kill(c) + exit\ of\ constructor$$

In this equation, **gen**(c) represents an alias set generated at the statement c . **kill**(c) denotes the alias set killed at the statement. The exit of constructor for each **CG** needs to be considered in case **CG** has overloading constructor for precision.

Now, *lemma 3* and *theorem 1* are defined as follow for Figure 10 and proved for our alias analysis algorithm in Java as follows as in **Carini** [4]'s.

Lemma 3. *The iteration loop of the algorithm, alias analysis on type inference, terminates because alias*

```

/* initial alias set of each procedure in each class is stored in its
 * class structure during parsing
 * initial type information is decided for all objects in a program
 * during parsing
 * procedure, function, method and constructor are named procedure
 * initialize an alias set of each node for CFGs
 */
//initialize calling graph with main() and its class
CallingGraph CG = new CallingGraph(mainClass, main);
loop {
  //initial procedure of CG is main()
  for each procedure P on alternating between topological
  or reverse topological traversal order of CG {
    // analyzing each statement of CFG within a procedure P
    for each statement node in CFG of P
      // detect alias set and its types for each statement
      if(currentStatement is assignment statement){
        assignmentStatement(LHS of currentStatement,
                             RHS of currentStatement,
                             currentStatement, C);
        //including constructor and its reference.
      } else if(currentStatement is call statement){
        callStatement(LHS of currentStatement,
                     RHS of currentStatement,
                     currentStatement, CG, C);
      } else {Aout = union(Ain);
      // union all of the parent statements for conditional
      // or loop statements
      if(point of union) UnionAliasSet(currentStatement.Parent);
      Aexit of P = Aout of last statement;
    } //end of for-loop; each statement node in CFG of a P
    for each called procedure Q of call site c in CFG of P {
      Qentry = Ain(c) + gen(c) - kill(c) + exit of constructor;
    }
  } //end of for-loop; each procedure P
} until alias set converges for all of procedures in CG

```

Figure 10. algorithm: Alias Analysis on Type Inference

set and CG for all of procedures must converge if we restrict the number of loops.

Proof: First, since the number of procedures in a programming code is limited, the construction of the CG will converge if we consider only calling statements within a loop block and the number of loops for it. Second, when a definition of an alias set is added to each statement, it will stay forever. Therefore, since number of definitions of alias set is finite for each statement of each procedure, alias set computation must eventually halt. ■

Theorem 1. *The algorithm, alias analysis on type inference, detects safe alias set by inferring safe runtime type information of a reference variable at any statement for overriding attributes (shadowed variables and overridden methods) in a Java code.*

Proof: Suppose that the algorithm is executed on an example Java code as its input in a trace. The trace is a sequence of statements that could be consecutively executed on the CFG of a procedure within CG because it is flow sensitive analysis. The current trace position is named (P, s); P represents the procedure and s represents the statement. We have an induction hypothesis which assumes that the algorithm detects safe alias set at (P, s).

First, when next statement s_n is an assignment statement, alias set will be updated and type information of LHS object reference variable will be inferred safely as shown in *Lemma 1* and *Lemma 2*.

Second, when next statement s_n is a call statement for callee P_n , entry of callee P_n and alias set of s_n will be updated with exit alias set of P_n safely by using the equation of alias analysis [1, 3]. If callee P_n is not in CG, it will be added to CG according to the type information of an object which invokes the callee as shown in *Lemma 1* and *Lemma 2*.

Therefore, since next trace (P, s_n) generates safe alias set and CG, all of the alias set up to end of the trace will be safe with *Lemma 3*. If we consider arbitrary position (P, s) as the beginning trace of the example Java code, all of the alias set should be safe during execution of the algorithm. ■

In order to compute the complexity of our algorithm for an example programming code, we have defined A_{max} as the maximum number of alias sets among statements in each procedure, E_{cg} as the maximum number of edges for the calling graph of the code, and P as the number of procedures in the CG. The worst case of time complexity for the outer loop is $O[A_{max} \times (E_{cg} + P)]$. For each procedure p , we have defined $E_{cfg}(p)$ as the maximum number of edges for the CFG, and $C(p)$ as number of call sites in a procedure p . The worst case of time complexity for the loop of each procedure p in a calling graph is $O[E_{cfg}(p) + C(p)]$. Therefore, worst case of time complexity of our algorithm is $O[A_{max} \times (E_{cg} + P) \times (E_{cfg}(p) + C(p))]$.

It is much less than existing algorithms [4, 5, 11] because our type inference operation has a time complexity of $O[\# \text{ of reference variables}]$ that is much smaller than the existing algorithms' $O[\# \text{ of virtual function call sites} \times \# \text{ of aliases at each call site}]$ for an object. When an object invokes a virtual function, it refers to type information indirectly since the initial type of an object does not contain the type information of the virtual function call in the existing algorithms. Also, existing alias relations generates the number of aliased elements in an exit node as $((O \times A_o + O) \times N_{pes})$; O is the number of objects in a program; A_o is the maximum number of aliased element for an object. Those are bigger than aliased elements of $O(nC_2)$ because n is practically less than 4; n is the number of aliased references for an object.

Even though our algorithm takes more time to handle constructors as functions in order to compute more safe alias set than others [4, 5, 8, 16], our type inference algorithm and TT will take less time and will compute a

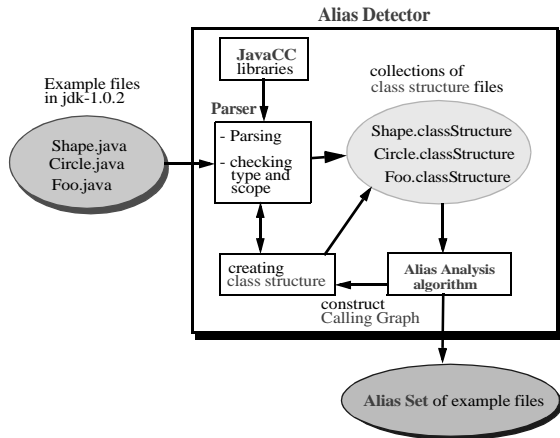


Figure 11. Alias detection for Java codes

safer alias set for constructors, shadowed variables and overridden methods than others [4, 5, 8, 16].

4. Methodology and an Example

Figure 11 shows our alias detection system in Java. It consists of two main parts: parser and alias analysis.

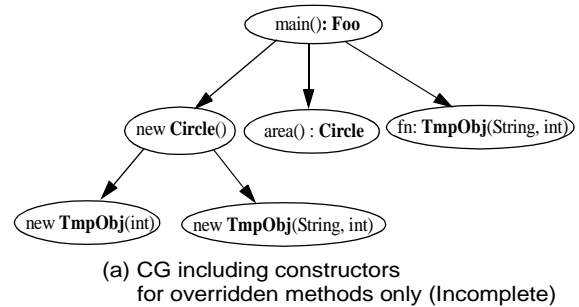
Java Compiler Compiler (**JavaCC**) shown in Figure 11 is a parser generator developed by Sun Micro Systems. The parser reads the example input classes and stores information of the attributes of the classes by creating each **CST**. Also, type and scope checking is done during parsing. As a result, our alias detector detects alias set of given Java codes. Our prototype system of Figure 11 is being implemented.

Figure 12 (b) is the final **CG** of Figure 1 on our algorithm. It is completely built by considering the shadowed variables, the overridden methods and constructors. The **CG** of Figure 3, built on existing algorithms [4, 5, 8, 16], fails to find the alias set for shadowed variables because it only focuses on overridden methods and it cannot construct the nodes of callees invoked by constructors. Even though the calling graph of Figure 12 (a) constructs the nodes of callees invoked by constructors, the **CG** is incomplete because it does not have type information for shadowed variables.

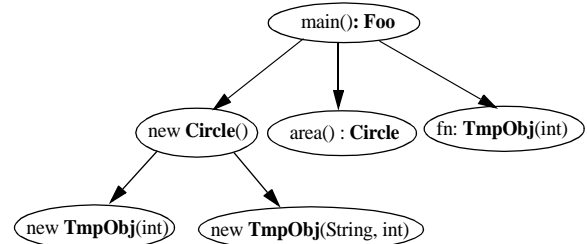
5. Conclusions

In this paper, more precise alias analysis algorithm has been presented by adding all possible type information and by considering constructor as procedure.

This algorithm detects more precise alias sets for both shadowed variables and overridden methods. Additionally, its efficiency is not negatively affected even though the precision is improved by adding type information. Our work is the first implementation of alias analysis on type inference for Java as far as we know.



(a) CG including constructors for overridden methods only (Incomplete)



(b) CG including constructors for shadowed variables and overridden methods (Complete)

Figure 12. Comparison of Calling Graph for Figure 1

The type information presented in this paper is also applicable to C++.

6. Reference

- [1] A. W. Appel, Modern Compiler Implementation in Java, Cambridge University Press, 1997.
- [2] D. F. Bacon and P. F. Sweeney, "Fast Static Analysis of C++ Virtual Function Calls", OOPSLA'96 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications, ACM SIGPLAN Notices volume 31 number 10, San Jose, California, October 1996.
- [3] J. Choi, M. Burke, and P. Carini, "Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects", The Twenties Annual ACM SIGACT SIGPLAN Symposium on POPL, Jan. 1993.
- [4] P. R. Carini, M. Hind, and H. Srinivasan, "Flow-Sensitive Type Analysis for C++", IBM Research Report, RC20267, Nov. 1995.
- [5] R. Chatterjee and B. Ryder, "Scalable, flow-sensitive type for statically typed object-oriented languages", Technical Report, DCR-TR-326, Rutgers University, Aug. 1997.
- [6] 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.
- [7] G. DeFouw, D. Grove, and C. Chambers, "Fast Interprocedural Analysis", Dept. of Computer Sci. and Eng., Univ. of Washington, Technical Report 97-07-02, July 1997.
- [8] A. Diwan, K. S. McKinley and J. E. B. Moss, "Type-Based Alias Analysis", SIGPLAN'98 conf. on Programming Lan-

guage Design and Implementation, 1998.

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

[10] U. Holzle and O. Agesen, "Dynamic vs. Static Optimization Techniques for Object-Oriented Languages", Theory and Practice of Object Systems, 1(3), 1996.

[11] M. Hind, M. Burke, P. R. Carini, and J. Choi, "Interprocedural Pointer Alias Analysis", ACM TOPLAS, July 1999.

[12] Sun Micro Systems, Java Compiler Compiler, <http://www.metamata.com/JavaCC/>, Version 1.1. 2000.

[13] W. Landi, B. G. Ryder, and S. Zhang, "Interprocedural Modification Side Effect Analysis With Pointer Aliasing", Technical Report, LCSR-TR-195, Rutgers University, 1992.

[14] Steven S. Muchnick, Advanced Compiler Design Implementation, Morgan Kaufmann Publishers, June 1997.

[15] 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.

[16] H. D. Pande and B. G. Ryder, In LNCS 1145, "Data-flow Based Virtual Function Resolution", Proc. of the Third International Symposium on Static Analysis, 1996.

[17] Je Woo, Jong Woo and J-L Gaudiot, "Flow-Sensitive Alias Analysis with Referred-Set Representation for Java", The Fourth International Conference/Exhibition on High Performance Computing in Asia Pacific Region, May 2000.