PAOLA
Program Analysis of Object-oriented Languages

Wolfram Amme, Markus Schordan,
Laszlo Boeszoermenyi, Wilhelm Rossak

(amme|rossak)@informatik.uni-jena.de
(markuss|laszlo)@ifi.uni-klu.ac.at

Abstract

This paper introduces *annotated TA graphs*—a new structure to performing program analysis in object-oriented languages. In contrast to known methods, with *annotated TA graphs* we can derive information of a program statement by a single corresponding graph. Our technique is based on performing a program analysis by solving a monotone data flow system.
Resolution of virtual functions is one of the major issues in the analysis of object-oriented languages. With our technique we can determine at compile time which method will be selected at run time. In the paper we describe the construction of a data flow framework that use *annotated TA graphs* as data flow information sets. Further, we demonstrate how to use *annotated TA graphs* to compute reaching definitions and constant value propagation.

1 Introduction

PAOLA is a collaboration between University of Klagenfurt (Austria) and University of Jena (Germany). The main focus of the PAOLA project is the development of new techniques for program analysis of object-oriented languages. A further aspect of the project is the evaluation of the developed methods by means of real applications. For this we plan to integrate the techniques into an optimizing compiler for the programming language Java.

Object-oriented programming has become a widely-used, important programming paradigm that is supported in many different languages. The C++ programming language is currently the most widely available and used language, and Java is a new type-safe language which has become popular recently. The analysis we present can be applied to a subset of Java and C++.

C++ gives the control of memory and handling of pointers into the hands of the programmer, whereas Java uses garbage collection and forces the programmer to use only dynamic data structures. This leads to a very high ammount of dereference operations during the execution of Java programs. The problem of aliasing for both, programmers and program analysis, is becoming increasingly important.

In C++ methods can be declared to be virtual whereas in Java a method defaults to be virtual and can be declared to be non-virtual using the keyword final. Both C++ and Java calls to virtual methods are dynamically dispatched. Since the method invoked, is decided at run-time, we need an analysis at compile time which is able to determine the method that is invoked. Such an analysis can provide the necessary information for aggressive optimizations at compile time as well as for the tools to support the programmer in the development process of large applications.

The static analysis we present is a type based alias analysis that uses partial in-lining, and gives the most precise results when applied as global analysis to a program.

We concentrate on one of the major issues of analysis in object-oriented languages, the overriding of methods; and present an algorithm that is based on a typed control flow graph, which gives very precise information represented as typed alias graphs. We do not deal with overloaded methods because those can be determined at compile time without alias analysis, and shadowed variables can be renamed, so that we can consider them distinct.

In section 3 we present how the control flow graph we use is created and how we deal with overridden methods. The language we analyse is a subset of object oriented languages like C++, Java, and Modula-3 and is described in section 5. The code examples we use to illustrate the details of our analysis are written in Java. The details of our typed alias analysis are presented in section 6 and related work in 8.

2 Program Analysis of Object-Oriented Languages

One of the major issues in the analysis of object-oriented languages is dynamic dispatch. As methods can be overridden and selected at run-time depending on type information, the development of an object-oriented system is easier because we are able to give a subclass a different behavior than the super class. However additional problems arise when we want to determine at compile time the method that is called at run-time. This information is crucial to the precision of any data flow analysis.

When an object is dynamically created by *new* in a Java or C++ program and the reference to this object is assigned to a variable, the type of the created object must be a subtype (or the supertype itself) of the variable the reference is assigned to. In the case of overridden methods the determination of which method is called at run-time depends on the type of the object originally created. This information is obtained by use of data flow analysis, described in section 6.

Additionally our programs are restricted to single level pointers. One pointer cannot be aliased to another, as this requires multiple levels of indirection [1]. As a result, when a pointer changes its value, it does not affect the value of any other pointers [2]. Within this context, the receiver type decides which virtual method is invoked at a call site, and the invoked method can affect aliasing. Using both alias information and type information, we are able to determine exactly which method is invoked if no conditionals introduce multiple possibilities for a subsequent program point.

To illustrate this objective we show an example of a simple program and an overview of the problems our algorithm deals with. We present the results of our analysis of the example program in section 7.

```
public class demo {
  public static void main(String argv[]) {
1   A a=new A();
2   B b=new B();
3   a.add(1);
4   b.modifyby(a);
5   a=b;
6   a.modifyby(b);
7   b.add(1);
  }
}
```

```
class B extends A {
  void modifyby(A x) {
    this.val+=x.val*4;
  }
}
```

```
class A {
  int val;
  void add(int v) {
    this.val+=v;
  }
  void modifyby(A x) {
    this.val+=x.val*2;
  }
}
```

Figure 1: A simple program.

The Java program in Fig. 1 consists of three classes. One class demo holds the main method[1] whereas class A is the base class of class B. The

[1] For reasons of simplicity of the example we do not deal with array passed as parameter to main.

method add is an inherited method, method modifyby is overridden in class B.

At the program points 1 and 2, new objects are created. We do not include return values in our analysis, and assume the constructors to be empty default constructors and the protected attribute value *val*, which is also inherited, as initialized with its default value of zero. At program points 3 to 7, five different problems in the program analysis arise. At point 3 we have to cope with a method call of the base class, at point 4 with the method call of a subclass which is overridden, at point 5 with an assignment, and at point 6 the overridden method in class B is invoked. This is a crucial point in the analysis because we need alias analysis to determine the object that was originally created, and its type to determine the method that is invoked. We will show how to determine exactly which method is invoked in this particular example. At point 7 the inherited method add is invoked. The object modified by add can be accessed via paths from variable a and variable b which will be shown in our alias analysis as well.

3 Interprocedural Control Flow Graph

To be able to determine the methods invoked at run-time we first construct a control flow graph, and give some additional type information to entry nodes of methods. We also use the class hierarchy information to restrict the control flow graph.

A control flow graph for a method consists of nodes, which represent single-entry, single-exit regions of executable code, and edges, which represent execution branches between code regions. We represent a program with an interprocedural control flow graph (ICFG). An ICFG is a triple (N,E,e), where N is the set of nodes, with the entry nodes holding additional type information of the classe(s) the method belongs to, and E is the set of edges. Node e is the entry node of main. N contains a node for each statement in the program, an entry and exit node for each method, and a call node for each method call.

For a non-virtual function call, we model the control flow in the called method by an interprocedural edge from a *call* node to the corresponding *entry* node. Virtual methods make it impossible to determine the correspondence between a *call* node and an *entry* node before analysis, since the method invoked depends on the type of the receiver at the call site. Therefore we establish multiple edges from the call node to the entry nodes of all methods, that may be called at run-time. Later, by typed alias analysis we are able to restrict the number of edges. All methods are partially in-lined and we get one graph for the entire program with multiple edges to be able to cope with virtual methods.

For each call in the program, a *call* node is created, as well as nodes for each statement of the method, and additionally entry and exit nodes are created, which are designated with the call. At point 4 we need to construct only the nodes of the overriding method of class B because we know already, when the ICFG is generated, that variable b is of type B. As *modifyby* is an overridden method, it is the method represented in the ICFG and thus is the only method possible to be invoked. This reasoning doesn't help at point 6. Here we have to create two edges to both methods *modifyby*, which could be possibly invoked by objects of class A and B, respectively. The entry nodes hold additional type information that is used by the flow analysis. At program point 6 the type information will then be used later by the typed alias analysis, to reduce the number of possible methods called.

4 Monotone data flow systems

Our method is based on expressing alias analysis of object-oriented languages as a *monotone data flow system*. Therefore, we have to model our data flow problem as a *data flow framework* $(L, \vee, F)$, where L is called the *data flow information set*, $\vee$ is the *union operator*, and F is the set of *semantic functions*.

The data flow information set is a conceptual universe of objects upon which the analysis is working. In our case, an element of the data flow information set covers, simultaneously, both alias information and type information for a single program point at a specific time during the analysis. We translate each program into a set of recurrence equations whose formal variables denote elements from the data flow information set. Each operator in an equation is either a semantic function or the union operator. A semantic function corresponds directly to a statement in the program, and models the effect of executing this statement. The union operator implements joining paths in the control flow graph.

The least fix-point, with respect to an implicitly given partial order, of the set of recurrence equations describes all alias resp. type information on the program. By an inductive process that starts with all variables initialized to the one element of the data flow information set, we successively approach the fix-point of the system. If the semantic functions are monotone and $(L, \vee)$ forms a bounded semi-lattice with a one element and a zero element,

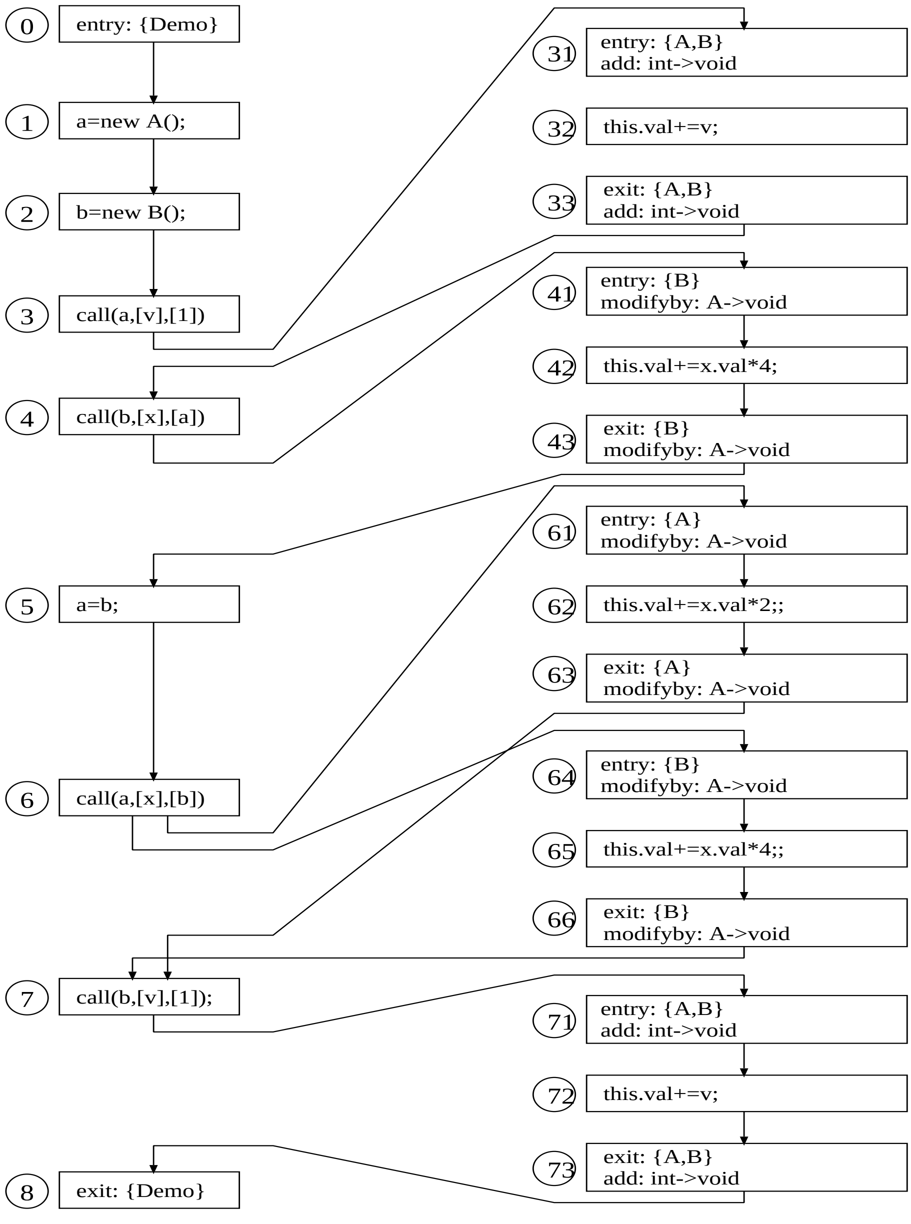

Figure 2: Interprocedural control flow graph

we can use a general iterative algorithm, that always terminates and yields the least fix-point of the data flow system [3].

As we perform a static analysis, we are not able to infer the direction of branches taken during program execution. Therefore, we must consider all paths in the control flow graph of a program, which might be a proper superset of the execution paths actually appearing at run time, thus rendering the analysis inaccurate. This is a general problem of static analysis, and not due to our specific model.

5 Programming model

The data flow analysis presented in this paper works for properly restricted object oriented languages such as Modula-3, C++ or Java. Particularly, we are interested in program analysis of programs written in Java. Therefore, we turn one's attention to object oriented features made available by the programming language Java.

Target programs are not allowed to handle data structures with more than one level of indirection, i.e., reference variables may not refer to objects that contain other reference variables. Reference variables can obtain their values via assignment or by application of the *new operator*. Variables can be declared as instance variables and class variables, respectively. For each target program, a control flow graph or an interprocedural control flow graph must be constructed. Finally, we assume also that our programs have been fully type-checked.

We accept arbitrary applications of object-oriented techniques such as variables shadowing, methods overloading resp. methods overriding. As mentioned in the Java language specifications [4], we assume that variables shadowing and methods overloading are selected at compile-time. Overridden methods, on the other hand, are always chosen dynamically at run-time.

Methods are allowed to access their arguments through call-by-value. We assume that all methods of a program have the result type *void*, which means that a method call produces no result values. Methods with result type other than *void* are modeled by the introduction of an additional reference parameter. Thus, objects which are referenced by that parameter variable hold the result value of a method.

Prior to the essential analysis we must transform the target program into a kind of normal form. Therefore, each access to an instance variable must be performed by the *this keyword*. Additionally, we have to rename program variables in such a manner that the declaration of variables will be unique. Note, transforming a program into its normal form is not essential, but working with a program in normal form simplifies our analysis.

For exactness of our data flow analysis we restrict the programming model in such a manner, that recursive procedures are not supported. But our data flow analysis can be extended to cover recursive procedures (see for example [5]).

6 Alias Analysis

We use typed alias graphs (*TA graphs*) as data flow information sets. Each *TA graph* denotes possible structures of the store and some aspects of its state at a certain point in program execution. In *TA graphs*, alias information is expressed by the set of paths with which we can access an object at one program point. Nodes of a *TA graph* represent the objects present in the store. We mark the nodes of a *TA graph* with the class name of the corresponding object. Additionally, to express the structure of an object we annotate each node with name of variables of the corresponding object. As we will explain below, by marking the nodes with type information, the control flow information given by the control flow graph can be made more precise during data flow analysis. Eventually, a reference of one object to another object is expressed by a labeled edge in the *TA graph*.

Fig. 5 shows the results of an alias analysis with *TA graphs* for the program given in Fig. 2. For each program statement, we calculated a *TA graph* that describes the store immediately before the execution of the statement. We point out that the analysis for the program of Fig. 2 becomes stationary after the second iteration.

The *TA graph* derived for statement 6 contains three objects, which are reachable by the sets of paths $\{a\}$, $\{b\}$, and $\{a., b.\}$, respectively. To express the type of the objects, we mark its corresponding nodes with reference types of class A, B, and with name of class B, respectively. At the moment of analysis the objects which are reachable via $\{a\}$ resp. $\{b\}$ contain the same reference to an object of class B, which is reachable by the set of paths $\{a., b.\}$.

If a node of the control flow graph has more than one predecessor we must integrate all information stemming from these predecessors. We use the *union operator* $\vee$ on *TA graphs* to express the fusion of the contained data flow information. The nodes of a *TA graph* can be uniquely distinguished by the sets of access paths reaching the nodes and the class type of the corresponding object. It is therefore obvious to join the information of two

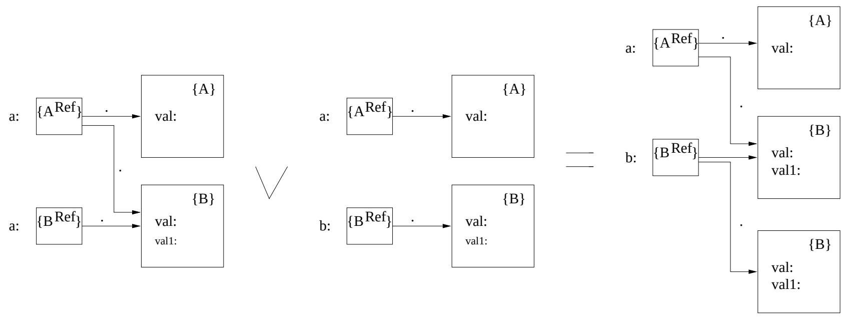

Figure 3: Joining two TA graphs.

TA graphs as follows:

Nodes that are reachable in both graphs by the same set of paths, and whose corresponding objects have the same class type, are collapsed to a single node. Fig. 3 shows an application of our union operator.

Let L be the set of all *TA graphs*. Then (L,$\vee$) forms a semi-lattice that is bounded since the number of the variables is always finite and we allow only one level of indirection. The empty *TA graph* is the *one element* and the *TA graph* resulting from iteratively applications of the union operator to all elements of L is the *zero element* of the semi-lattice.

Fig. 4 defines the semantic functions used by our method. In this definition, TA_{in} stands for a *TA graph* before the application of the semantic function, and TA_{out} for the corresponding *TA graph* after the application of the semantic function. These semantic functions are composed of auxiliary functions that perform transformations on *TA graphs*. Splitting the semantic functions is not essential, but simplifies the presentation.

Our method separates an object into static and non-static members, i.e., class and instance variables. Therefore, at the beginning of our data flow analysis, we insert into the init graph of the *main* method, for each class A a node, which represents the class variables of A. These nodes are reachable via the corresponding class name.

GenerateObject(A, p, class) starts a new edge labelled with *'.'* on every node of A that can be reached via p and attaches a blank node—which represents the non static members of class *class*—to it. If A contains a node that stands for static members of *class*, we have to insert an edge from every node that is reachable via p to the node that is reachable via *class.*.

DeleteEdges(A, p) deletes all edges emanating from any node of A that can be reached via p and performs a garbage collection thereafter, i.e., deletes all nodes not reachable via any path starting at a variable.

Assignments to reference variables

- *p = q;*
  TA_{out} = *InsertEdges(DeleteEdges(*TA_{in}*,p),p,q)*
- *p = null;*
  TA_{out} = *DeleteEdges(*TA_{in}*,p)*
- *p = new class();*
  TA_{out} = *GenerateObject(DeleteEdges(*TA_{in}*,p),p,class)*

Entry points

- *Entry Set*
  TA_{out} = *Declare(*TA_{in}*,Set).*

Exit points

- *Exit Set*
  TA_{out} = *Reject(*TA_{in}*,Set).*

Method calls

- *call(p, fplist, aplist);*
  TA_{out} = *Call(*TA_{in}*,p,fplist, aplist).*

Others

$TA_{out} = TA_{in}$.

Figure 4: Semantic functions.

InsertEdges(A, p, q) marks all nodes that are reachable via path *q*. Then the function inserts into A an edge labelled with *'.'* from any node of A that can be reached via p to copies of any marked node of A. Finally, *InsertEdge(A, p, q)* deletes all marked nodes.

We use *Call(A, p, fplist, aplist)* to define the semantic function of a node that represents a method call. The function *Call(A, p, fplist, aplist)* requires two lists—*fplist* and *aplist* as parameters. List *fplist* holds formal parameters and list *aplist* actual parameters of the method call. An application of *Call(A, p, fplist, aplist)* works as follows:
First, it marks all nodes that are reachable via *p* with the special keyword *this*. By doing so, we mark objects that could perform the method call. Marking the nodes of a *TA graph* with *this* indicates, where a method finds its instance variables. After introduction of *this* keywords, for each formal parameter *fp* we allocate a blank node and insert an edge from *fp* to it. As the formal parameter type is a reference type, *Call(A, p, fplist, aplist)* inserts for each edge that starts at a node reachable via the corresponding

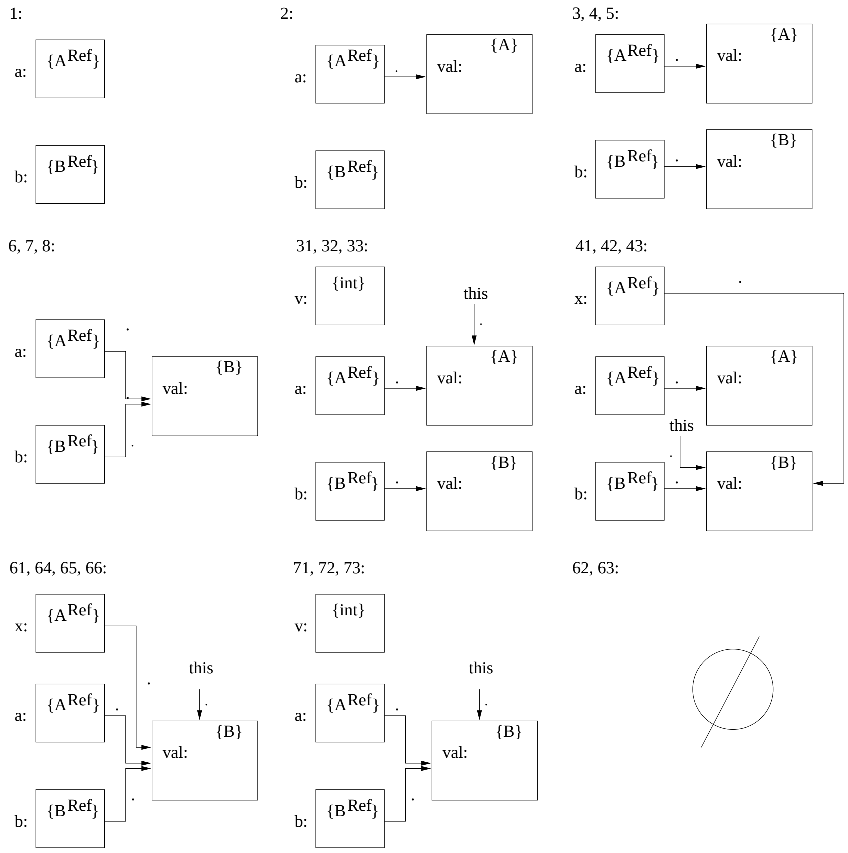

Figure 5: Example of an alias analysis with TA graphs.

actual parameter *ap* another edge, starting at the node pointed by *fp* and terminating at the same target node.

$Declare(A, Set)$ attaches a blank entry node to A for each local variable $n \in Set$. We use this function to introduce local procedure variables into a *TA graph*. But, the main task of $Declare(A, Set)$ is to determine whether the method call can occur when A is the ongoing *TA graph* of the *entry node*. Therefore, $Declare(A, Set)$ verifies if one of the *this* keywords refers to an object that can call the method. If there is no adequate node in A, function $Declare(A, Set)$ returns the one element as a result, i.e., the data flow analysis will propagate no information to the method body—which means such a method call cannot occur. In contrast, if there is a *this* in A, that points to a node representing an object of an adequate class, and $Declare(A, Set)$ proceeds as follows:

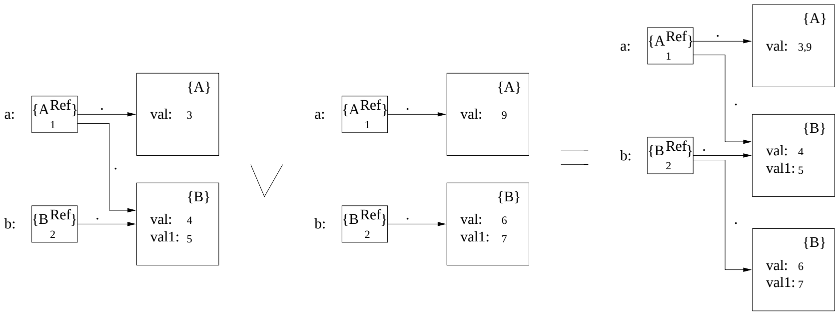

Figure 6: Joining two annotated TA graphs.

It marks all nodes which are reachable by the *this keyword* and which represent an object of a class that doesn't define the method. Thereafter, the function removes all marked nodes from A and returns the arising *TA graph*.

Verifying the feasibility of a method call enables the information which is given by the control flow graph to be more precise, i.e., we integrate parts of control flow analysis into data flow analysis. Fig. 5 shows that with our method, we can determine that the method call in statement 6 executes the implementation of *modifyby()* given in class B, but not that of class A.

$Reject(A, Set)$ consider that the scope of a local variable in a method will be left. $Reject(A, Set)$ deletes all nodes in A that can be reached via a local variable $n \in Set$. Thereafter, it performs a garbage collection, i.e., deletes all nodes not reachable via any path starting at a variable.

7 Program analysis

Program analysis is an indispensable task within optimizing and parallelizing compilers, and is also widely applicable towards various other software development techniques; for example, source-level debugging, automated testing, semantic browsing, integration of independently altered versions of programs. To solve our intrinsic task—performing a program analysis of object-oriented languages—we additionally assign information, which is the result of classical program analysis, such as value, last definition, last use of an object resp. available expressions, etc., to the nodes of a *TA graph*. A *TA graph* which contains additional information about program analysis is called an annotated *TA graph*.

We demonstrate our method by concentrating on the calculation of reaching definitions of a program. Reaching definitions are defined as the

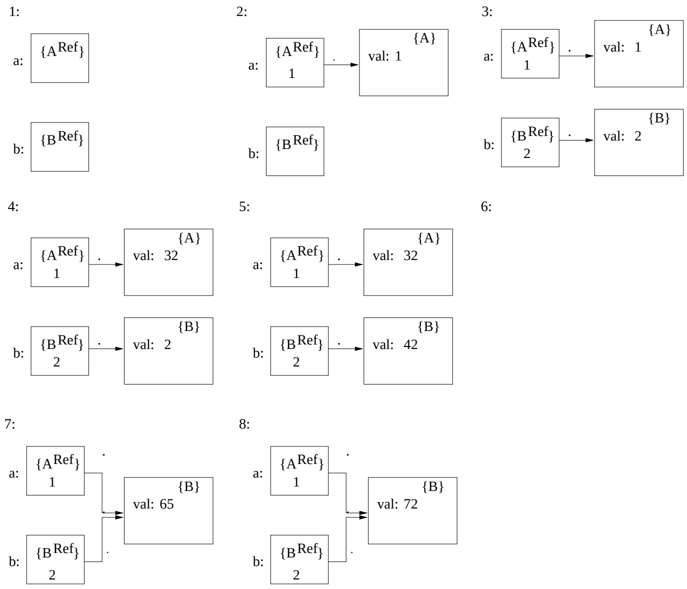

Figure 7: Reaching Definitions with annotated TA graphs.

problem of determining, for a specific program point and a storage object, all program points where the value of this storage object has been (or could have been) written last. An annotated *TA graph*, which is used for the calculation of reaching definitions, contains in principal the same information as a *TA graph*. However, besides the alias and type information, we can also learn, which program statements define local variables resp. instance variables of objects, last. Fig. 7 shows the results of an analysis with annotated *TA graphs* for the program given in Fig. 2. For each program statement of the main method we have determined an annotated *TA graph* that describes the reaching definitions before the execution of the statement.

When calculating reaching definitions with annotated *TA graphs*, we must extend the data flow framework, which is given in the previous section. The union operator must be extended as follows. If nodes that are reachable in both graphs by the same set of paths are collapsed to a single node, additionally we have to attach to these nodes the union of the labels of the collapsed nodes. Fig. 6 shows an example of joining two annotated *TA graphs*.

Semantic functions of the constructed data flow framework must consider the updating of local variables and instance variables, respectively. Fig. 8 contains the semantic functions for the determination of reaching defini-

Assignments to non reference variables

- *s: p = ...*
  TA_{out} = *Update(TA_{in},p,s);*

Assignments to reference variables

- *s: p = q;*
  TA_{out} = *Update(InsertEdges(DeleteEdges(TA_{in},p),p,q),p,s)*
- *s: p = null;*
  TA_{out} = *Update(DeleteEdges(TA_{in},p),p,s)*
- *s: p = new class();*
  TA_{out} = *Update(GenerateObject(DeleteEdges(TA_{in},p),p,class,s),p,s)*

Method calls

- *s: call(p, fplist, aplist);*
  TA_{out} = *Call(TA_{in},p,fplist, aplist,s).*

Figure 8: Semantic functions for the determination of reaching definitions.

tions with annotated *TA graphs*. We denote only the semantic functions, which functionality must be extended. In particular, we have to introduce a function called *Update()*, used to model the updating of a variable. An application *Update(A, p, s)* registers the corresponding variable name with the statement number s in every node that can be reached via the path p in A. An application of *Call(A,p,fplist, aplist,s)* must model the definition of formal parameters. Therefore, *Call(A,p,fplist, aplist,s)* registers the statement number s in every node that is inserted for an element $n \in$ fplist. Finally, we have to attend definitions that occur when a new object is generated. Modeling this kind of initial definitions, GenerateObject(A,p,class,s) updates all instance variables of the generated object with statement number s.

Annotated *TA graphs* can be used for solving further problems of program analysis. It would only be necessary to construct a data flow framework that models the problem. Fig. 9 for example, shows the results of a data flow analysis with *TA graphs* for the program in Fig. 2, where we use the concept of annotated *TA graphs* for an extended value propagation [6].

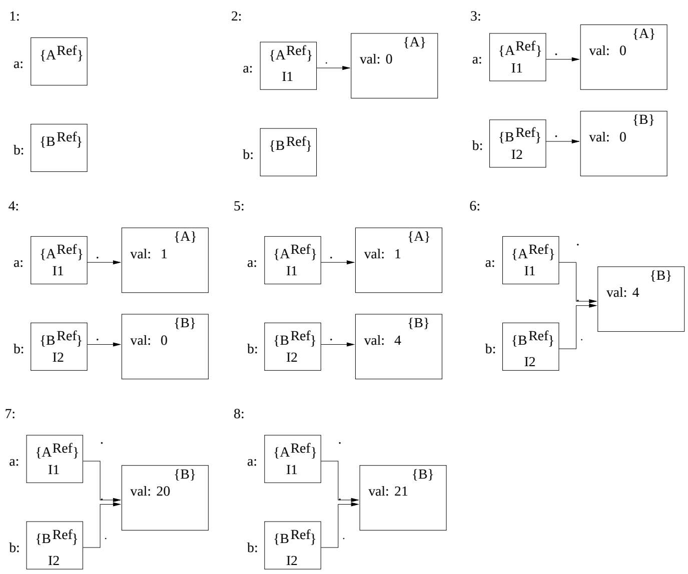

Figure 9: Value Propagation with annotated TA graphs.

8 Related work

Many algorithms have been developed to solve aliasing problems introduced through reference formals. A precise, polynomial-time algorithm for calculating alias information in FORTRAN programs is described for example in [7].

Conditional alias analysis has been suggested for interprocedural calculation of *may alias* information in programs with pointers in [8]. This technique analyzes the code of a procedure under certain input assumptions and then combines the results of this conditional analysis with the situations that actually may occur during program execution. Based on this approach, algorithms for computing precise program-point-specific aliases in C programs with a single level of indirection [1] and approximate aliases for multiple levels of indirection [9] have been designed.

Furthermore, in [1] a polynomial-time algorithm for computing *must alias* information for data structures with a single level of indirection is presented. Based on conditional aliasing, in [10] the first polynomial-time approximate algorithm for interprocedural def-use associations for programs with single level pointers is developed. Similar to the technique described above, the reaching definitions of a program statement are used to derive the def-use associations.

An interesting technique for the calculation of *may alias* information in programs with pointers is developed in [11]. This work proposes a memoryless model to solve the *may alias* problem, employing algebraic methods. Starting from a new semantic model for data aliasing, an algorithm is stated that allows extraction of precise *may aliases* in a compact way for certain list manipulating programs. Due to the lack of a strong updating operation, this algorithm may yield spurious aliases. As mentioned above, the method offers no algorithm for the calculation of *must alias* information. Therefore this algorithm may be used for conflict analysis, only.

[12] developed an algorithm to determine the possible shapes that heap-allocated structures in a program can take on. Their method is quite accurate and can also be applied to cyclic data structures. Manipulation of the representation is based on sets of access paths, distinguishing this work from previously published approaches. They use an elegant form of materialization. Therefore, when information appears that is not denoted by the graph, a new node from the condensation node is materialized.

Amme and Zehendner describe in [13] a store-less method to determine data dependences in programs with pointers. They use so-called A/D graphs as data flow information set. Amme and Zehender perform a single-pass data dependence analysis by solving a monotone data flow system for a class of restricted imperative languages. Based on an intraprocedural analysis with A/D graphs they develop a method to derive a safe approximation of the data dependences by employing k-bounded A/D graphs.

Rinard and Diniz use type equality to disambiguate memory references. But the type system they use does not have inheritance [14].

Diwan, McKinley, and Moss evaluate three alias analysis based on programming language types. The most precise of these three is a flow-insensitive analysis that uses type compatibility and additional high-level information such as field names [15]. They use redundant load elimination to demonstrate the effectiveness of the algorithms in terms of opportunities for optimization. But they are not able to determine which method is invoked at run-time from a set of overridden methods (see program point 6 in our example program) because their algorithm is flow independent.

Pande and Ryder [2] present a polynomial time combined algorithm to perform program-point-specific, interprocedural type determination and aliasing algorithm for C++. The algorithm is a work-list based, fixed point iteration method that reports points-to-type and may-hold alias information.

9 Conclusions

In this paper we have presented an algorithm of a compile time alias analysis for object-oriented languages like Java/C++. Our technique is based on a monotone data flow system. The alias information is represented by typed alias graphs. By using the type information in our TA graphs we are able to determine which virtual method will be used at run time. As a first step an interprocedural control flow graph (ICFG) is created. We assign each node in the ICFG a semantic function, which defines how the alias information is propagated during the data flow analysis. Finally we presented how annotated TA graphs can be used to compute reaching definitions and value propagation.

The use of annotated TA graphs for program analysis promises a significant improvement over known methods. Currently, we have started to construct an experimental system to obtain preliminary data on the usefulness of our method. As a further step we are attempting to extend the present research to programs with arbitrary data structures. Our final aim is to be able to analyze any program written in Java.

A very pregnant point of research work in the PAOLA project will be the evaluation of our methods by means of real applications. By doing so, we hope to answer the question, how far an optimization resp. a parallelization of object oriented programs, which is based on our program analysis, can improve the runtime behavior of a program. In contrast to other works we are not only interested in performing a static evaluation of our method — which means we are not only interested in measures as number of appearing alias pairs, number of data dependences, etc. We perform a dynamic evaluation of our techniques as well. We hope, by doing so, we will get an answer to the question, how precise an alias analysis resp. a program analysis should be, so that a significant improvement in runtime behavior of a program can be reached. In other words we want to answer the question, what is the truly meaning of a precise, often time-consuming static analysis in practice.

References

[1] William Landi and Barbara Ryder. Pointer-induced aliasing: A problem taxonomy. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 18th Annual ACM Symposium on Principles of Program-*

ming Languages (POPL '91), pages 93–103, Orlando, FL, USA, January 1991. ACM Press.

[2] Hemant Pande and Barbara Ryder. Static type determination for C++. In USENIX Association, editor, *Proceedings of the 1994 USENIX C++ Conference: April 11–14, 1994, Cambridge, MA*, pages 85–97, Berkeley, CA, USA, April 1994. USENIX.

[3] B. Chapman H. Zima. *Supercompilers for Parallel and vector Computers*. ACM Frontier Series. Addison-Wesley, 1991.

[4] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, USA, second edition, 1998.

[5] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In David S. Wise, editor, *Proceedings of the SIGPLAN '88 Conference on Programming Lanugage Design and Implementation (SIGPLAN '88)*, pages 21–34, Atlanta, GE, USA, June 1988. ACM Press.

[6] W. Amme, P. Braun, F. Thomasset, and E. Zehendner. Data dependence analysis of assembly code. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 340–347, Paris, October 1998.

[7] K. D. Cooper and K. Kennedy. Fast interprocedural alias analysis. In ACM, editor, *POPL '89. Proceedings of the sixteenth annual ACM symposium on Principles of programming languages, January 11–13, 1989, Austin, TX*, pages 49–59, New York, NY, USA, 1989. ACM Press.

[8] William Landi. Interprocedural aliasing in the presence of pointers. Technical Report lcsr-tr-174, Rutgers University Laboratory for Computer Science Research, 1991.

[9] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. *ACM SIGPLAN Notices*, 27(7):235–248, July 1992.

[10] Hemant D. Pande, William Landi, and Barbara G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[11] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN'94 Conf. on Programming Language Design and Implementation*, pages 230–241, Orlando (Florida, USA), June 1994. ACM. SIGPLAN Notices, 29(6).

[12] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conference Record of POPL '96: The* 23$^{\text{rd}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–31, St. Petersburg Beach, Florida, 21–24 January 1996.

[13] Wolfram Amme and Eberhard Zehendner. Data dependence analysis in programs with pointers. *Parallel Computing*, 24(3-4):505–525, May 1998.

[14] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis : A new analysis framework for parallelizing compilers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implemantation*, pages 54–67, New York, May 21–24 1996. ACM Press.

[15] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 106–117, Montreal, Canada, 17–19 June 1998.