Virtual Method Resolution with Typed Alias Graphs

Markus Schordan, Wolfram Amme

markuss@ifi.uni-klu.ac.at amme@informatik.uni-jena.de

Abstract

We present an algorithm that computes alias configurations, propagated types and reaching definitions in one single pass. Object-oriented features like virtual method resolution, inheritance and a method call of a super class within a method of a subclass are fully integrated in our analysis. We use a monotone data flow system and introduce typed alias graphs as data flow information sets.

1 Introduction

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++. In this paper we focus on how to deal with virtual methods, the this (self) reference, and the call of a method of a super class. 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 tools to support the programmer in the development process of large applications.

The static analysis we present is a type based alias analysis that gives the most precise results when applied as global analysis to a program. We perform three essential tasks in our analysis. The possible types of objects at each program point, the reaching definitions for each field of an object and virtual method resolution.

We present an algorithm that is based on a control flow graph and 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 2 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. 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 3.

2 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. The consideration of super-calls makes some more reasoning on the type hierarchy necessary and enables us to determine exactly the virtual methods invoked for the given example with our algorithm.

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 and a tupel consisting of the formal and actual parameters of the call. We attach the type of the receiver object for which the CFG of a method is created to the entry node. During flow analysis this type information is compared with the type *this* refers to. If it is the same type then this method can be invoked at run-time. In the other case the TA graph is set to the one element and this path is considered not to be executed at run-time. E is the set of edges. Node e is the entry node of the program. N contains a node for each statement in the program, an entry and exit node for each method, and a call and return node for each method call.

We create multiple edges and nodes for virtual methods to represent all methods that may be selected by dynamic dispatch at run-time. The number of edges that carry information may be reduced because the TA graph may be set to the one element at several entry nodes. By this we

```
class A {
   public int val;
   void define() {
      this.val=1;
   }
   void modifyby(A x) {
      this.define();
      this.val+=x.val;
   }
}
```

```
class B extends A {

   void define() {
      this.val=2;
   }
   final void modifyby(A y){
      super.modifyby(y);
      this.val*=y.val;
   }
}
```

```
class C extends B {

   final void define() {
      this.val=3;
   }
}
```

Figure 1: The classes of the example program.

identify overridden methods that are not invoked at run-time. If exactly one edge emanating from the entry nodes carries information then such a method is identified to be a candidate for in-lining. If it is not possible to reduce the number of possibly invoked methods to one the information is combined at the *return* nodes.

Since our analysis is a conservative analysis we may not identify all methods that are not invoked but we are able to reduce the number of invoked methods significantly.

2.1 Special treatment of super

The super keyword can be used in any subclass to call methods of the corresponding super class. A call to a method using super is treated different to an ordinary method call in our creation of the ICFG. As parameter of the method call we pass on a *this* reference. To the entry node of the created method that represents the super-call, we attach the same type as is attached to the entry node of the calling method. This is essential and is illustrated by the super-call in our example at point 7.2.2. - To deal correctly with the use of *this* in different methods it is necessary to use an indexed *this*. But, not to unnecessarily increase the number of alias pairs, we do not create a new index with calls on super because any call on *super* or *this* means that the this reference remains the same during the method call. This is the case in our example at points 7.1.2, 7.2.2, and 7.2.2.2. Because we represent each method call by one path in the ICFG it is safe to check at entry nodes whether *this* refers to an object of a type for which the ICFG of the method was created. That way we consider the fact that the virtual method selected at run-time corresponds to the type of the receiver object.

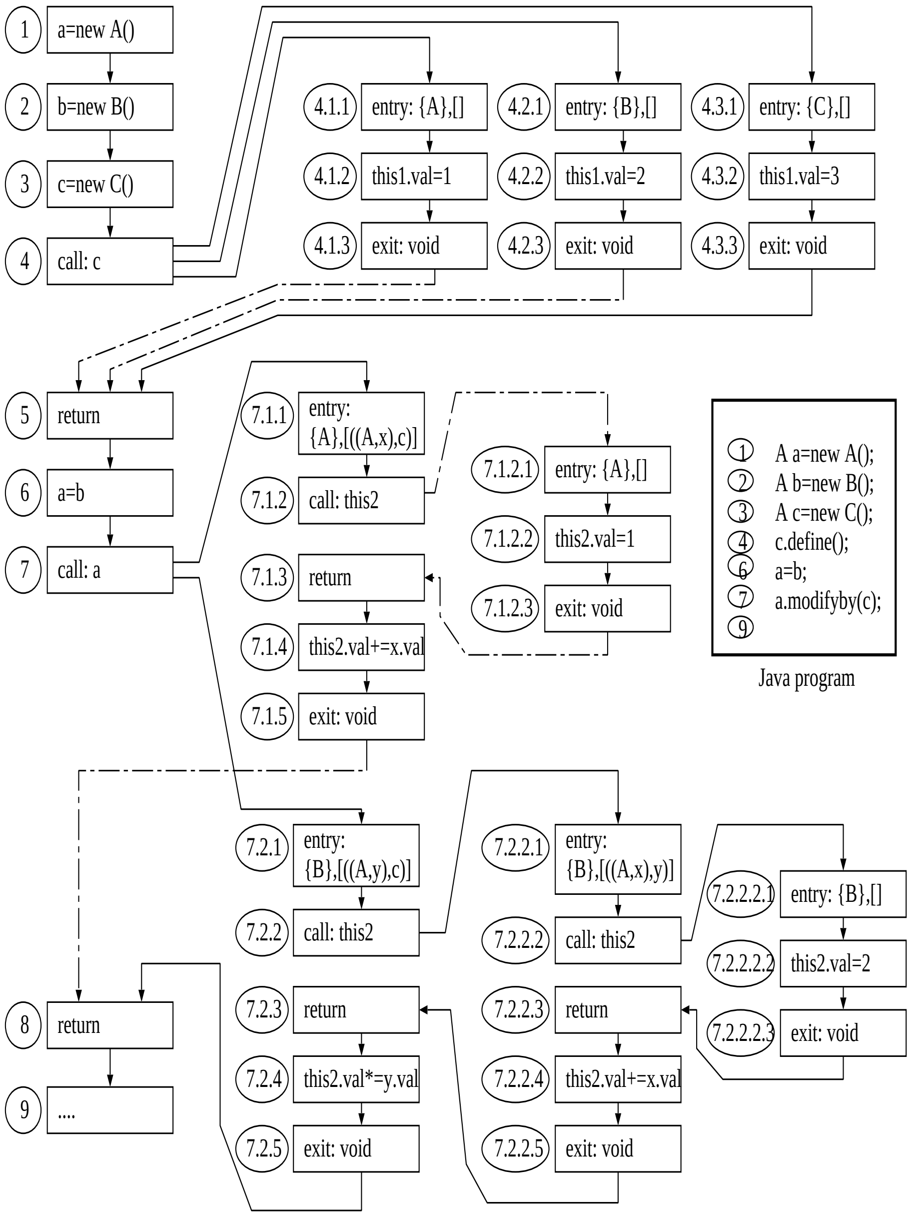

Figure 2: Interprocedural control flow graph

2.2 Example

Let us illustrate the basic mechanisms for dealing with overriding of methods by an example. In Fig. 1 the source of the three classes A,B, and C is given, in Fig.2 the main program with its corresponding ICFG. At statement 4 we have to construct nodes for all three *define* methods of classes A,B, and C because it depends on the run-time type of the variable c which method is called at run-time. Analogous at statement 7, where nodes for the two methods *modifyby* of classes A and B are created and two edges connecting the call node with each entry node. We do not have to consider class C because *modifyby* is declared final in class B.

Additional reasoning is necessary on method calls with *this* and super to obtain the control flow graph of Fig. 2. Either the method *modifyby* of class A or B is called. For both cases we create nodes representing the respective method and attach the type to the entry node. Let us focus on the case of modifyby when the receiver object is of type B. This corresponds to our representation of method modifyby with type B attached to the entry node. From the entry node on it is safe to assume the type, *this* refers to, as type B. This is important to handle the super-call of modifyby of class B correctly. The super-call is represented in the ICFG by a this call and we create a path representing the method modifyby of class A but attach type B to the entry node. This ensures that our check at the entry node during flow analysis succeeds. We only need to create one path for the call of *define* because we only represent the case that the receiver object is of type B. Again we attach type B to the entry node.

When we apply our flow sensitive alias analysis we are able to reduce the number of multiple edges. This works fine for the given example at points 4.1.1, 4.2.1, and 7.1.1. - but in practice it may not always be possible to reduce the number of information carrying edges emanating from an entry node, i.e. the number of possibly called methods, to one. In the same pass we perform reaching definitions in our alias analysis as we describe in section 3.2.

3 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 the names 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. 4 shows the results of an alias analysis with *TA graphs* for the program given in Fig. 2 and Fig. 1. For each program statement, we calculated a *TA graph* that describes the store immediately before the execution of the statement.

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;*
  TA_{out} = *Call*(TA_{in},*p*,*s*).

Figure 3: Semantic functions for the determination of reaching definitions.

Each node in the CFG a semantic function is assigned which defines how our monotone data flow system propagates and alters the TA graphs during analysis. Fig. 3 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.

Following we describe the semantic functions and how the TA graphs are altered:

GenerateObject(A, p, class, s) 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.

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.

InsertEdges(A, p, q) inserts edges into A. To all nodes that are reachable via q a node starting from p is added to the graph.

Call(A, p, s) defines the semantic function of a node that represents a method call. The function *call* only handles the treatment of the *this* reference. If necessary it creates a this-node and establishes edges from *this* to all other nodes that are refered by p. This is similar to analyse the assignment this=p.

3.1 Entry and exit nodes

At each entry node of the CFG the type of the class the method belongs to and the list of formal and actual parameters of the original program are stored. Since we create a separate path for each method possibly invoked by a virtual method call only the type of the corresponding receiver object is attached to the entry node.

Therefore, at entry nodes is checked whether the this-node refers to an object that can call the method. If there is no adequate node in TA graph, function *entry* passes on the one element (the empty set), 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 then we proceed with the modified TA Graph.

At exit nodes we need to delete the local context. All nodes in the TA graph A that can be reached via a local variable are deleted including all edges emanating from them. The this reference is treated different if it was reused in the call of the method. In that case edges emanating from *this* remain unaltered (Fig 2: points 7.2.2.5, 7.2.2.2.3).

3.2 Reaching definitions

We demonstrate our method by concentrating on the calculation of reaching definitions of a program. Reaching definitions are defined as the 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 analyse, which program statements define local variables resp. instance variables of objects, last.

Semantic functions of the constructed data flow framework must consider the updating of local variables and instance variables, respectively. Fig. 3 contains the semantic functions for the determination of reaching definitions with annotated *TA graphs*. A function called *Update()*, is 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.

Fig. 4 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. At the top of each TA graph we note down the numbers of *all* program points where that TA graph is computed as result of our analysis. The def-values of variables that are not defined by assignment in the program are left free. Those are assumed to be initialized with default values depending on the programming language analyzed.

For the given example program we are able to determine all program points where the member val of class B and C is defined. Those lines in the ICFG that are not solid represent edges where only the empty graph is propagated. This is the case for paths that are determined not to be executed at the entry points of methods.

There is one assignment for the member val of class C at point 4.3.2. Therefore we note 4.3.2 as the definition point of val. This is done at point 4.3.2 by the semantic function for assignment to non-reference variables. For the member val of class B three assignments are made. At the points 7.2.2.2.2, 7.2.2.4 and 7.2.4 val is defined with a new value. We note down the program position where val is defined. For any optimization based on alias graphs this is an ideal information to extract dependence information of variables and program positions in a program. For each program point we are able to determine the reaching definitions in the presence of virtual methods and inheritance.

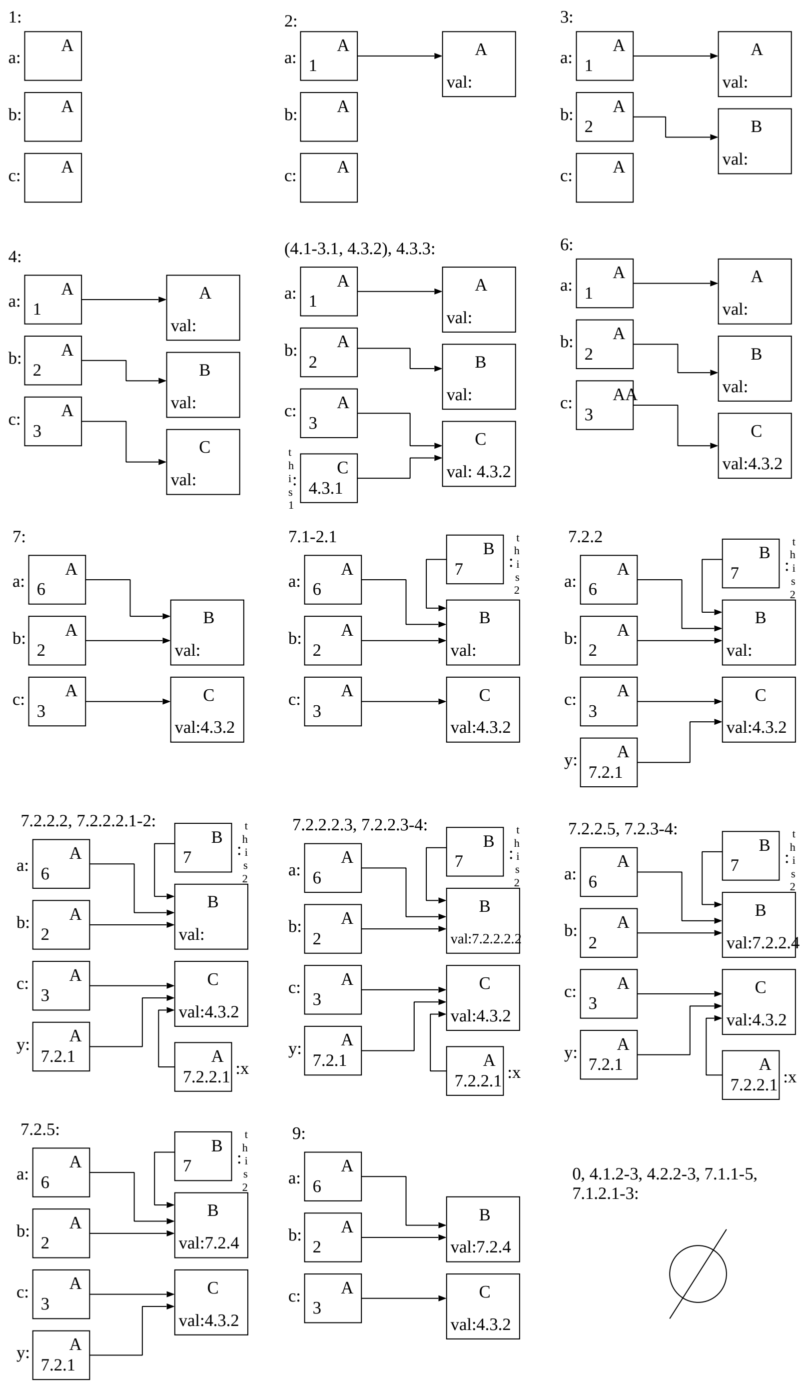

Figure 4: Example of an alias analysis with TA graphs.

4 Related work

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 [1]. 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 7 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. In contrast to their approach we represent all information by one single alias graph during analysis.

In [3] the effectiveness of four methods for detecting whether a method invocation is monomorphic (always calls the same method) is investigated. They apply type hierarchy analysis, type propagation, aggregate analysis and interprocedural type propagation (only to scalars). In contrast to our approach they apply type propagation and aggregate analysis in two different passes. We integrate type propagation, virtual function resolution and reaching definitions (which is more precise than aggregate analysis) in one single pass and represent the data in one uniform data structure, so called typed alias graphs. It is interesting to note that our approach shows that it is not necessary to separate type propagation from the computation of reaching definitions.

Rinard and Diniz use type equality to disambiguate memory references. But the type system they use does not have inheritance [4].

[5] 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 summary node is materialized.

Amme and Zehendner describe in [6] a store-less method to determine data dependences in programs with pointers. They use so-called A/D graphs as data flow information set and 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.

5 Conclusions

In this paper we have presented an algorithm for 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. We presented how annotated TA graphs can be used to compute reaching definitions in object-oriented programs in the presence of virtual methods.

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/C++.

References

[1] 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.

[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] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96)*, pages 292–305, 1996.

[4] 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.

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

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