PAOLA—Program Analysis of Object-Oriented Languages

Wolfram Amme, Markus Schordan
Laszlo Böszörmenyi, Wilhelm Rossak

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

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.

In object-oriented languages, objects are accessed via references. An object reference is in principle the same as a pointer to a storage cell. As in imperative languages a correct program analysis of object-oriented languages must be based on the results of a safe alias analysis. One of the main issues is overriding of methods which depends on type information and references to objects established at run-time.

Program analysis often uses a well-known technique of static analysis—monotone data flow analysis. By doing so, a program has to be transformed into a control flow graph in a first step. Thereafter, for each program statement the desired information can be derived by traversing the control graph iteratively.

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

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. Each type corresponds to one edge of the call of a virtual function. Later, by typed alias analysis we are able to restrict the number of edges that carry information. By this we identify overridden methods that are not invoked at run-time. If it is not possible to reduce the number of possibly invoked methods to one the information is combined at the join node of the *exit* nodes.

It is imperative to our analysis that types are computed in the same way as reaching definitions. Additionally the subtype relation and set of types computed for a particular node is used at call nodes to reduce the set of methods that may be invoked at run-time. Since our analysis is a conservative analysis we may not identify all methods that are not invoked but we are able to reduce

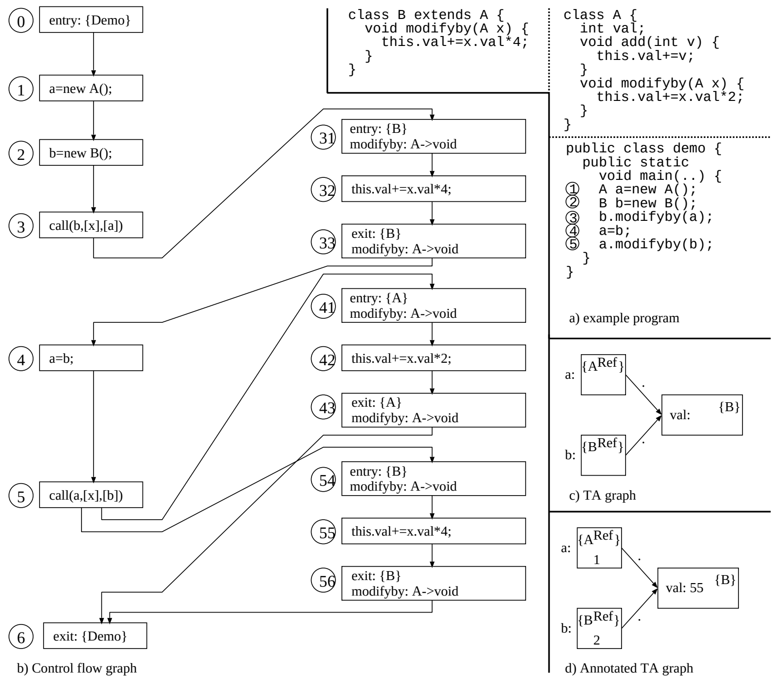

Figure 1: Source, control flow graph, TA graph, and annotated TA graph.

the number of invoked methods significantly because our method is well suited for computing reaching definitions.

Let us illustrate the basic mechanisms by example. Fig. 1 (b) shows parts of an interprocedural control flow graph, which we derived from the simple Java program listed in Fig. 1 (a). The program consists of three classes. One class *Demo* holds the main method whereas class A is the base class of class B. Further, the method *modifyby* that is defined in A is overridden in class B. We store information about the receiver object, the formal parameters and the actual paramters in the call node. The type information is stored in the entry node of the respective method.

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 names of variables of the corresponding object. Eventually, a reference of one object to another object is expressed by a labeled edge in the *TA graph.* Figure 1 (c) contains the TA graph, which describes the storage after analyzing the given program, that is before execution of statement 5. As we can see, besides compiler constructed storage cells there could be an object instance of type B in storage which is reachable by the set of variables $\{a, b\}$.

The availability of object types, which we assign to the nodes of an TA graph, leads to a more precise control flow analysis. With this technique we can choose the target of a calling method during data flow analysis, i.e. with the TA graphs in Fig. 1 (c) we can determine that the calling method in statement 5 must be the modifyby routine of class B and not that of class A.

To solve our intrinsic task—performing a program analysis of object-oriented languages—we additionally can assign information, which we need for optimization of programs, like value, last definition, last use of an object resp. available expressions, etc., to the nodes of a TA graph. Fig. 1 (d) contains an annotated TA graph for statement 6 in which we hold the program statements that define the contents of a storage object last, i.e., the reaching definitions. Integrating such kind of annotations into the data flow analysis leads to an elegant form of program analysis. With the additional type information at program point 5 we are able to determine which method is invoked and we compute that it must be program point 55 where a value is assigned to the variable `val` of the dynamically allocated object that is referenced by variable `a`.

A very significant 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 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. In this way, we hope to 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.