Towards cheap, accurate polymorphism detection

Nevena Milojković

Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch/

Abstract. Polymorphism, along with inheritance, is one of the most important features in object-oriented languages, but it is also one of the biggest obstacles to source code comprehension. Depending on the run-time type of the receiver of a message, any one of a number of possible methods may be invoked. Several algorithms for creating accurate call-graphs using static analysis already exist, however, they consume significant time and memory resources. We propose an approach that will combine static and dynamic analysis and yield the best possible precision with a minimal trade-off between used resources and accuracy.

1 Introduction

Developers often make assumptions about method invocations based on clues from source code. They need to understand their source code better. Knowing the call-graph structure at compile time would be of great benefit to the developers, and it would enhance the tools relying on static information.

While writing code, developers are often interested in the run-time behaviour of the system under development, especially in the run-time types of variables [6]. Whereas current IDEs mainly focus on the source code, and not on the dynamic behaviour, developers could benefit from both of them. Gathering the information from a running application is straightforward, presuming that the application can be instrumented and executed. However, in some cases it is not possible, or it could cause some additional costs.

Even though static and dynamic techniques exist to extract run-time information, both have shortcomings in performance, and in recall and precision. Our idea is to explore which static algorithms are affordable and give good, usable results, possibly combined with the information dynamically collected.

2 Description of the approach

In order to construct the most precise call-graphs possible in object-oriented systems, many algorithms have been developed over the years [3,1,7]. Shivers' k-CFA analysis [5] is a widely known family of control-flow analyses, accepted also in the object-oriented world, even though it was created primarily for functional languages. It has been established as being one of the most reliable analyses,

although one of the most expensive. 1- and 2-CFA are considered to be "heavy" analyses in OO languages, but feasible [4,2]. Other analyses, such as CHA and RTA analyses scale well, but, in most cases, they are not accurate enough. In general, the precision of one such algorithm is correlated with the number of nodes in the call-graph, *i.e.* the number of reachable methods, which represents a conservative approximation of a program behavior during run-time.

We intend to explore the trade-off between static and dynamic techniques to build accurate call graphs. Whereas static techniques generally produce false positives (theoretically possible but actually infeasible execution paths), dynamic techniques may produce false negatives (feasible paths that simply aren't covered). We wish to explore how static and dynamic techniques can be combined to yield higher precision at a reasonable cost. To obtain dynamic information we will execute tests where the method and the source code coverage are large enough to give us consistent results. In cases where it is possible to predict all potential inputs, the preferred way would be to run the application, under the assumption that it will finish in reasonable time.

We propose a tool that will return more precise information whenever it is possible to do so. For example, static analysis in some cases gives too ambiguous results, *i.e.* a really large set of methods possibly used at a certain call site. In such cases, our program-understanding tool could offer dynamically collected information which provides a more focused set, often more relevant for developers. Another approach would be to use an inexpensive static analysis in order to determine the call sites where polymorphism is possible, and then analyze only these call sites dynamically, thus reducing cost.

References

1. David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. *SIGPLAN Not.*, 31(10):324–341, October 1996.
2. Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not.*, 44(10):243–262, October 2009.
3. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 77–101, Aarhus, Denmark, August 1995. Springer-Verlag.
4. David Van Horn Matthew Might, Yannis Smaragdakis. Resolving and exploiting the k-CFA paradox. In *PLDI*, pages 305–315, 2010.
5. Olin Grigsby Shivers. *Control-flow Analysis of Higher-order Languages of Taming Lambda*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991. UMI Order No. GAX91-26964.
6. Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 23–34, New York, NY, USA, 2006. ACM.
7. Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 281–293, New York, NY, USA, 2000. ACM.