4L−2

Call-Graph Optimization of Java Applications

Antonio Magnaghi, Shuichi Sakai and Hidehiko Tanaka
The University of Tokyo

1 Introduction

Java popularity has remarkably increased over the last few years and the utilization of Java programming language has flourished both in academic and industrial projects. The demand for high-performance Java applications is, however, stressing the necessity of appropriate compilation techniques and aggressive optimization procedures. In this paper we present a framework to enhance static analysis of Java applications by exploitation of type-inference [1, 4]. And we experimentally evaluate how such a framework can be utilized to improve interprocedural analysis for source code level optimizations. For brevity reasons, the proposed concepts are not strictly formalized and exposition simply outlines the implemented algorithms.

2 Type-Inference via Living Classes Analysis

The program to analyze is represented by the Application Taxonomy (AT). AT collects, in a control-flow insensitive manner, all reference types (classes or interfaces) needed for compiling the application. AT elements are distinguished in class-nodes and interface-nodes. AT nodes are organized in a tree data structure that collects all syntactic and semantic information necessary for successive analysis procedures. The developed representation model (AT) formulates also proper conditions in order to deal with programming constructs critical to static analysis, such as dynamic class generation and native methods.

We aim at inferring the classes an expression can be instance of at run-time based on the analysis of objects instantiated by the application. Thus, we define the application Living Classes Set (LCS). LCS collects all AT class-nodes that correspond to classes conservatively generated by any application execution path. A class C is excluded from LCS if and only if there can be no execution path in the application that causes C to be instantiated.

Let $type(.)$ be a function that returns the unique static type associated to a program expression. $type(.)$ is properly defined as Java is strictly statically typed. Let $subTreeClasses(.)$ be a function that maps every AT node n to the set of class-nodes of the AT sub-tree rooted in n. There exists a one-to-one correspondence σ between reference types in the application and AT nodes. Hence $subTreeClasses(.)$ can equivalently be described as a function that maps every reference type t in the application to the set s of all application class types that subclass t. Therefore, if expression $expr$ has static reference type $t = type(expr)$, then at run-time $expr$ is instance of classes in $subTreeClasses(\sigma(t))$. This represents a conservative assumption.

In order to refine type-information, the function $typeInference(.)$ is introduced. For every application reference type t: $typeInference(t) = subTreeClasses(\sigma(t)) \cap LCS$

3 Experimental Evaluation of LCA

Given a program call-graph representing the possible callees at each call site, interprocedural analysis summarizes the effects of callees at each method entry. Because of Java dynamic dispatching mechanism, the set of possible callees at each call-site is difficult to evaluate precisely, necessitating to compute the possible classes of message receivers or the possible values returned by invoked methods. Generally, more consolidated call-graph construction techniques rely on interprocedural data-flow analysis. In this section we investigate an alternative approach: type-inference through LCA is applied to enhance call-graph construction.

Application	Classes	Code Lines
1. httpserver	1	62
2. proxy	3	309
3. RngPack	8	1419
4. dent	22	4286
5. Jasmine	177	15585

Figure 1: Benchmarks

The choice of benchmarks is a delicate aspect because of the lack of standard programs that are widely accepted by the Java community. This is due, on one hand, to the broad variety of programming contexts that Java APIs address, and, on the other hand, to the relative newness of the language. We chose a set of five Java benchmarks based on diversification of application area and complexity as well: 1.) httpserver: is a simple HTTP-server application; 2.) proxy: is a generic cascading proxy server supporting single socket network applications; 3.) RngPack: implements a random number generator; 4.) dent: is a formatter of Java source code; 5.) Jasmine: is a Java byte-code decompiler. Figure 1 summarizes some features of these programs. The number of classes in figure 1 refers to classes (interfaces) contained in the application distribution package. The number of code lines refers to the application source code whether it is available, otherwise it is obtained through byte-code decompilation. All benchmarks are pure Java applications.

The conducted experimentation can be described as follows. Firstly AT is constructed and LCS is evaluated. Figure 2 shows for each benchmark the cardinality of AT and LCS. A preliminary observation is that for benchmarks 1 to 4 the number of taxonomy classes does not vary much (about 200 elements). The number of classes (interfaces) that constitute the applications is relatively small (see figure 1), and therefore the majority of AT nodes are represented by classes (interfaces) belonging to jdk APIs. Benchmark 5 represents an exception: 49.2% of AT nodes are classes (interfaces) that belong to the application (177 AT nodes out of 360).

	$\|AT\|$	$\|LCS\|$
1. httpserver	198	111
2. proxy	197	113
3. RngPack	191	106
4. dent	206	116
5. Jasmine	360	260

Figure 2: AT and LCS Cardinality

It is interesting to additionally observe that for applications 1, 2, 3, and 4 the percentage of living classes in the AT is considerably stable (respectively: 56%, 57.3%, 55.5% and 56.3%) even if the size of the applications in terms of lines of code varies in a remarkable way from benchmark 1 to benchmark 4. The fifth benchmark shows a different behavior instead: 72.2% of taxonomy classes are living classes. Such a high percentage of living classes affects LCA precision as following experimental steps clearly prove. Successively, experimentation consists in evaluating the effectiveness of LCA for call-graph optimization. Hence, two approaches are used when producing the application call-graphs. In the first place, for each benchmark, the call-graph is produced without the exploitation of type-inference by LCA. Therefore, only AT is taken into consideration when analyzing call-sites. Then, the call-graph is computed again, but LCA is performed in order to gather more precise information. Figures 3 visualizes the achieved results.

	Method Reduction (%)
1. httpserver	19.9
2. proxy	18.7
3. RngPack	18.5
4. dent	19.4
5. Jasmine	9.2

Figure 3: Method Reduction

Figure 3 compares the number of methods (constructors) included in the constructed call-graphs when LCA is not employed against the case when LCA is carried out. It shows for each benchmark the reduction percentage of the number of call-graph methods (constructors). LCA performs efficiently on benchmarks 1,2,3,4: it is possible to obtain by LCA a 20% reduction of methods (constructors) in the call-graph compared to the case where LCA is not employed. In the case of the fifth benchmark, instead, LCA produces a 9.2% reduction. As previously observed such a benchmark shows a higher percentage of living classes (72.2%) compared to the other selected applications.

4 Conclusions

We proposed both a conceptual framework and an implementation to carry out type-inference on Java programs. Using this framework, we empirically accessed a set of benchmarks, which vary in complexity and area of application. We applied LCA type-inference to these benchmarks in order to optimize their call-graphs. Obtained results showed that LCA enabled substantial improvements in call-graph construction. For those benchmarks where the percentage of application living classes was approximately 50%, LCA type-inference led us to a significant reduction in the number of call-graph constructors/methods (20%). However, in one case (benchmark 5) the percentage of living classes was remarkably higher (more than 72%) than in the other benchmarks, and call-graph optimization achieved by LCA was limited. We are improving LCA performance also in such situations by adopting a control-flow sensitive algorithm for evaluation of application living classes. We expect that LCA type-inference can result beneficial not only to call-graph construction, but also to other essential tasks performed by optimizing compilers. Specifically, in our research project about automatic parallelization of Java programs [2, 3], LCA is adopted not only to enhance program call-graphs, but also to develop an interprocedural analysis framework where aliasing conflicts are investigated via Type-Based Aliasing Analysis (TBAA) [3].

References

[1] S. Collin, D. Colnet, O. Zendra. Type Inference for Late Binding: the SmallEiffel Compiler. In Proceedings of the Joint Modular Languages Conference (JMLC'97), Lecture Notes in Computer Science, Springer-Verlag, pp. 67-81, 1997

[2] A. Magnaghi, S. Sakai, H. Tanaka. An Interprocedural Approach for Optimizations of Java Programs. In Proceedings of Information Processing Society of Japan, 1998

[3] A. Magnaghi, S. Sakai, H. Tanaka. Interprocedural Analysis for Parallelization of Java Programs. In Proceedings of the 4th International Conference on Parallel Computation (ACPC99), Lecture Notes in Computer Science, Springer-Verlag, pp. 594-595, 1999

[4] A. Magnaghi, S. Sakai, H. Tanaka. Evaluation of a Type-Inference Framework for Java Applications, Proceedings of the Workshop on Java for High-Performance Computing, ACM International Conference on Supercomputing (ICS99), pp. 67-74, June, 1999, Rhodes, Greece.