Evaluation of a Type-Inference Framework for Java Applications

Antonio Magnaghi, Shuichi Sakai and Hidehiko Tanaka

Information Science Department, The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan
{magnaghi,sakai,tanaka}@mtl.t.u-tokyo.ac.jp

1. Introduction

The popularity of Java has increased remarkably 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 [MH98].
In this paper we introduce a framework to enhance static analysis of Java applications by exploitation of type-inference [A95, CCZ97, Mi78]. And we experimentally evaluate how such a framework can be utilized to improve interprocedural analysis for source code level optimizations.
Our approach is related to the work of [CCZ97]. In [CCZ97] the authors propose a type-inference algorithm for Eiffel and mention the possibility of applying it to the Java language. But they do not identify limitations to their definition of living classes that arise in the case of Java programs. Instead, the framework we develop consists of new operative definitions that address Java-specific semantic features. In particular, Application Taxonomy (AT) and application Living Classes Set (LCS) are key-concepts that lead us to propose a practical type-inference algorithm for Java applications based on Living Classes Analysis (LCA). In addition, we take advantage of the byte-code based execution model that allows byte-code files to be decompiled [Ja]. This enables us to investigate a new perspective for performing more aggressive analysis which we extend also, by decompilation, to those programs available only in byte-code format (for instance third-party class libraries). Hence we achieve a more complete and precise program representation. In the typical case of compiled languages source level optimizers have to assume the worst case hypothesis when performing interprocedural analysis in the correspondence of library routine invocations because of source code unavailability.

In addition, this paper evaluates the impact of our type-inference framework (LCA) on a benchmark suite of five programs. LCA is employed to optimize call-graph construction. Most consolidated call-graph construction techniques [GDD97] utilize interprocedural data-flow analysis, but we instead investigate the possibility and convenience of LCA type-inference. Compared to data-flow analysis, LCA has the advantage of simplicity, yet it has an effective impact on call-graph optimizations. Obtained results show that LCA produces a worthwhile reduction (up to 20%) in the number of methods (constructors) in application call-graphs. We also evaluate the efficiency of LCA for eliminating late binding in correspondence of call-graph call-sites. Our results seem encouraging: LCA reveals that 92.7% of late binding occurrences can be replaced by static bindings. If we compare the value we obtained (92.7%) with the data reported in [CCZ97] (80%), LCA appears to be a more efficient type-inference method.

Exposition is organized as follows. First, (section 2) the concept of Application Taxonomy (AT) is introduced to conveniently represent Java applications. Second, we provide the definition of AT living classes and the algorithm to compute them for a given Java application (section 3). In section 4, we outline the type-inference algorithm based on Living Classes Analysis (LCA). Through LCA we show that it is possible to effectively refine type information about program expressions; section 5 offers a quantitative evaluation of our techniques for the specific case of call-graph construction. AT and LCA are used to optimize call-graphs of five real Java applications. The utilization of LCA produces an

appreciable improvement in the quality of constructed call-graphs compared to the case where LCA is not utilized. Our conclusions are then presented in section 6.

2. Program Representation: the Application Taxonomy

The program to analyze is represented through the *Application Taxonomy* (AT). AT statically groups all classes (interfaces) that are involved by any possible execution of the application. Based on input data to the application, the execution path can utilize or generate instances of different classes. AT collects, in a control-flow insensitive manner, all possible reference types needed during application execution. A partial order relationship is naturally defined on AT elements based on inheritance (extends and implements keywords). For instance, if class B extends class A, then B is a son of A.
Throughout the remainder of the paper, the term "reference type" will identify, based on the context, either a class or an interface.

Let the AT *generator* be the class of the application program from which execution begins, namely the class containing the main method. Upon AT instantiation, the generator is added to the taxonomy.
Inserting a reference type *t* in AT triggers the following actions:

- *t* is parsed (after decompilation, if necessary) and from its members (fields, constructors and methods) information is collected about utilized reference types. Some of the principal reference type attributes of *t* are summarized by the following sets:
  1.) *extended_r_t*: reference types of which *t* is a direct descendant;
  2.) *implemented_r_t*: reference types implemented by *t*;
  3.) *thrown_r_t*: reference types thrown as exceptions by member constructors or methods of *t*;
  4.) *returned_r_t*: reference types returned by member methods of *t*;
  5.) *field_r_t*: declaration reference types of member fields of *t*;
  6.) *declaration_r_t*: declaration reference types of variables declared inside method or constructor bodies, or reference types used in cast expressions;
  7.) *formal_r_t*: declaration reference types of formal parameters;
  8.) *static_r_t*: reference types of which at least one static member is referenced in *t*;
  9.) *living_r_t*: classes instantiated (new keyword) in *t*.
- reference types extracted from *t* (points 1. through 9.) are inserted in AT though the same process, possibly leading to the identification of further AT elements.

AT construction ends when no additional reference types are collected.
AT nodes are distinguished in *class-nodes* and *interface-nodes*, which respectively correspond to classes or interfaces.
In section 3, discussion will focus on *static_r_t*, *living_r_t*, and also on *returned_native_r_t*. This latter AT node attribute is a subset of *returned_r_t*, and groups reference types returned by native member methods of an AT node.

Some restrictions apply to the AT model for program representation. AT is designed for those cases where classes are not generated dynamically during program execution (for instance, by extending the abstract class java.lang.ClassLoader). Such a circumstance potentially would not allow the above AT construction algorithm to statically identify all necessary reference types.

Attention must be paid also to native methods whose behavior can not be described precisely. A native method might actually return an object that is an instance of a subclass of the return reference type declared in its signature, preventing static detection of the actually involved reference types. We require that such a circumstance does not occur. Therefore, a native method is assumed to return an instance of the signature return reference type, which is necessarily a class because interfaces can not be instantiated.

3. Application Living Classes

We aim at inferring the classes an expression can be an instance of at run-time based on the analysis of objects instantiated by the application. Thus, this section preliminary defines the application *Living Classes Set* (LCS).

Let CS be the set comprising all AT class-nodes. If C is an element of CS, $C.a$ denotes attribute a of taxonomy class-node C. In the previous section, among taxonomy node attributes, we defined: $C.field\_r\_t$, $C.static\_r\_t$, $C.living\_r\_t$ and $C.returned\_native\_r\_t$.

Let nestedRefComp(.) be a function from CS to the power set of CS, and such that it associates to a class C in CS the union of 1.) the set of declaration class types of fields in C and 2.) the set of field declaration class types encountered by recursively inspecting each class reference type in $C.field\_r\_t$. For example, if class A contains a reference member field whose declaration type is class B, and if class B contains only primitive member fields, then nestedRefComp(A)={B}.
The precise definition of nestedRefComp(.) can be specified as follows:

$$\text{nestedRefComp}(C) = \text{auxNestedRefComp}(C, \varnothing)$$

where auxNestedRefComp(.,.) is an auxiliary function dealing with definitions of circular data structures:

$$\text{auxNestedRefComp}(C,S) = (C.field\_r\_t \cap \text{CS}) \cup \left(\bigcup_{\substack{C \in C.field\_r\_t \cap \text{CS} \\ C \notin S}} \text{auxNestedRefComp}(C', S \cup C.field\_r\_t) \right)$$

The first argument of auxNestedRefComp(.,.) is the AT class-node to inspect, whereas the second argument is a set collecting reference types already met, which, therefore, must be discarded.

LCS is defined as follows. A class C belongs to LCS iff at least one of these conditions is verified:

1.) C is the AT generator
2.) $\exists C' \in \text{LCS}: C \in C'.living\_r\_t$
3.) $\exists C' \in \text{LCS}: C \in C'.static\_r\_t$
4.) $\exists C' \in \text{LCS}, \exists C'' \in C'.returned\_native\_r\_t: C \in \text{nestedRefComp}(C'')$

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. Condition (1.) states that the AT generator is always considered a living class. The generator contains the main method and the JVM class loader generates an instance of it when the application is started. Because of conditions (2.) and (3.), a class C is in LCS if an exemplar of it is instantiated by class C' which is already in LCS, or if C' code references one of C static members (fields or methods).

Condition (4.) deals with native methods according to previous assumptions. If C' is a living class for the application (it is in LCS), and if it contains a native method returning reference type C'', we conservatively assume that such a method can be invoked, returning an instance of C''. All nested reference components of C'' potentially are generated by the native method upon creation of the returned object C''. Therefore all nested reference types of returned types by native methods belong to LCS (condition 4.).

4. Type-Inference via Living Classes Analysis

AT and LCS are used to perform type-inference and, thus, to refine type information associated with program expressions. At run-time, because of polymorphism, an expression can be an instance of any subclass of the class (interface) that represents the expression static type. Therefore, program call-sites can actually dispatch messages to several different methods, limiting interprocedural analysis effectiveness. *Living Classes Analysis* (LCA) employs information about those AT classes that may be instantiated by the application in order to improve static analysis.

Let type(.) be a function that returns the unique static type associated with a program expression. type(.) is defined properly 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 be described equivalently 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 the expression *expr* has static reference type *t*=type(*expr*), then at run-time *expr* is an instance of classes in subTreeClasses(σ (*t*)). This represents a conservative assumption. In order to refine type information, the function typeInference(.) is introduced. For every application reference type *t*:

$$\text{typeInference}(t) = \text{subTreeClasses}(\sigma(t)) \cap \text{LCS}$$

As LCS is a super-set of the classes that are actually instantiated through any program execution path, the above definition enables the removal of all those classes that belong to the taxonomy but that the application does not generate. Therefore, the value typeInference(type(*expr*)) is evaluated to perform type-inference on any expression *expr*. This represents an estimation of the classes *expr* can be an instance of when the application is run, and type information refinement is obtained as typeInference(type(*expr*))$\subseteq$subTreeClasses(σ(type(*expr*))).

In [M99], through a working example, we discuss the developed concepts of AT and type-inference by LCA. Additionally, [M99] utilizes LCA for call-graph construction optimization. This is the subject of next section. The example we propose in [M99] depicts the typical situation met in the implementation of the Visitor Pattern [GHJ94] and shows how LCA effectively improves program analysis in such a case.

5. Experimental Evaluation of LCA

Type-inference algorithms for the refinement of type information find several fruitful areas of application in the design and development of optimizing compilers for OO programming languages. In the remainder of the exposition, we will focus on the application of LCA to call-graph construction because of the importance of call-graphs for interprocedural optimizing compilers.

Substantial improvements in application performance can be achieved by allowing optimizing compilers to make less conservative assumptions across method invocation boundaries. 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 the Java dynamic dispatching mechanism, the set of possible callees at each call-site is difficult to evaluate precisely, and necessitates the computation of 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.

The choice of benchmarks is a delicate task 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. Table 1 summarizes some features of these programs.

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

Table 1

The number of classes in table 1 refers to classes (interfaces) contained in the application distribution package. The number of code lines refers to the application source code if it is available, otherwise it is obtained through byte-code decompilation. All benchmarks are pure Java applications. For convenience, benchmarks will be identified by the sequential numbers they have in table 1.

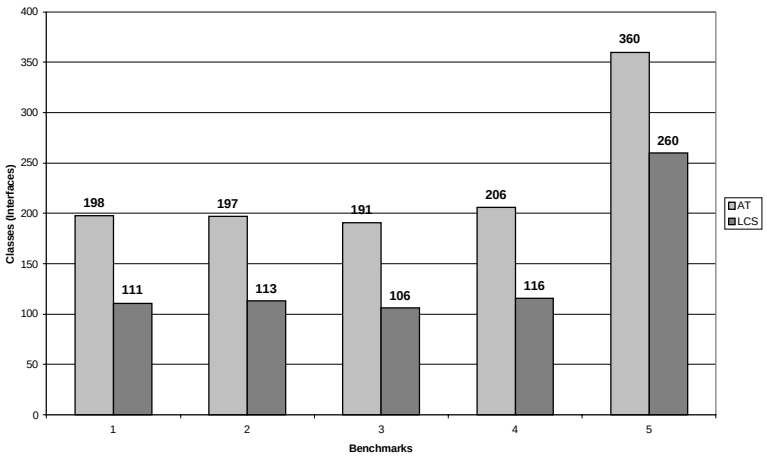

Figure 1

The experiment we conducted can be described as follows. Firstly AT is constructed and LCS is evaluated. Figure 1 shows for each benchmark (horizontal axis) the cardinality (vertical axis) of AT (light gray bars) and LCS (dark gray bars). A preliminary observation is that in 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 table 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).

Additionally, it is interesting to 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 significant manner from benchmark 1 to benchmark 4. But the fifth benchmark shows a different behavior: 72.2% of taxonomy classes are living classes. Such a high percentage of living classes affects LCA precision as the 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 was 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. Figure 2 and 3 show the achieved results.

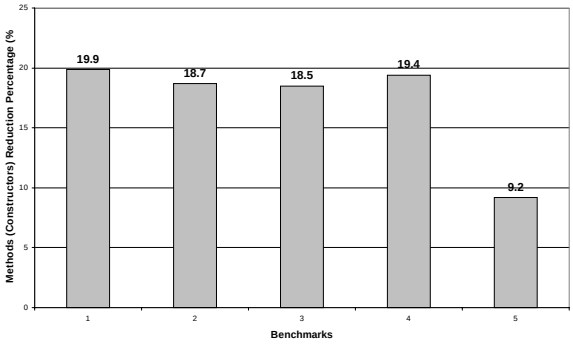

Figure 2

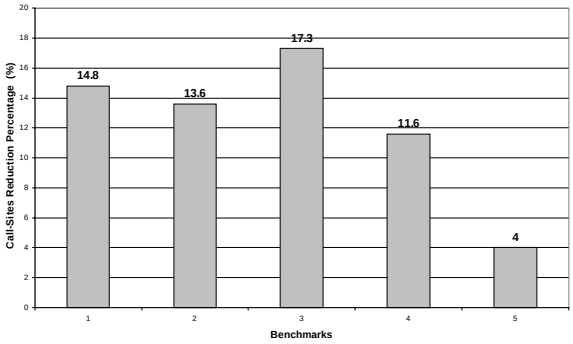

Figure 3

Figure 2 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 (horizontal axis) the reduction percentage (vertical axis) of the number of call-graph methods (constructors). LCA performs efficiently on benchmarks 1,2,3 and 4: it is possible to obtain by LCA a 20% reduction of methods (constructors) in the call-graph compared to the case when LCA is not employed. Instead, in the case of the fifth benchmark 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.

Figure 3 evaluates LCA impact on call-graph enhancement by considering the reduction in the number of overall call-sites in the case where LCA is employed in comparison to where it is not utilized (the call-site reduction percentage is reported along the vertical axis). Figure 3 shows that LCA achieves the highest reduction percentage on the third benchmark (17.3% reduction), and that the lowest reduction value (4%) is attained in correspondence of the fifth benchmark.

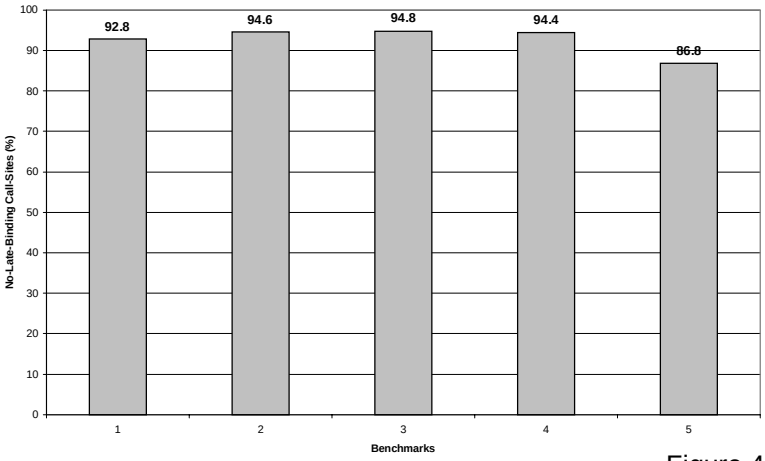

Figure 4

In figure 4, we provide additional experimental data about LCA collected from benchmarks 1,2,3,4 and 5. The application call-graphs are exposed to LCA optimization. The diagram indicates the percentage (vertical axis) of call-sites for which run-time method dispatch requires only one method.

Our LCA-based approach enables compilers to locate a remarkable amount of late binding occurrences that can be replaced by static bindings (on the average 92.7%).

6. Conclusions

The contributions of this paper can be summarized as follows. 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: (1.) in the number of call-graph constructors/methods (20%), and (2.) in the number of late binding occurrences (over 92.7%). 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 aim to improve 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 be 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 [MST98, MST99], 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) [DMM98, MST98].

References

[A95] O. Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pp. 2-26, 1995

[ASU86] A. Aho, R. Sethi, J. Ullman. *Compilers: Principles, Techniques and Tools,* Addison-Wesley, 1986

[A97] A. Appel. *Modern Compiler Implementation in Java*, Cambridge University Press, 1997

[BNN97] C. Brownhill, A. Nicolau, S. Novack, C. Polychronopoulos. Achieving Multi-level Parallelization. In *Proceedings of ISHPC'97International Symposium,* Lecture Notes in Computer Science, Springer-Verlag, pp. 183-194, 1997

[CCZ97] 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

[DMM98] A. Diwan, K. McKinley, E. Moss. Type-Based Alias Analysis. In *Proceedings of the ACM SIGPLAN98 Conference on Programming Language Design and Implementation*, pp. 106-117, 1998

[GHJ94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns,* Addison-Wesley, 1994

[GJ90] J. Graver, R. Johnson. A Type System for Smaltalk. In Proceedings of POPL, pp. 139-150, 1990

[GDD97] D. Grove, G. DeFouw, J. Dean, C. Chambers. Call Graph Construction in Object-Oriented Languages. In *Proceedings of the ACM OOPSLA97 Conference*, pp. 108-124, 1997

[H98] W. Hwu. *ISCA-98 Tutorial, Java: VM Architecture, Software Architecture, Implementations and Applications*, 25th International Symposium on Computer Architecture, 1998

[Ja] "*Jasmine Java Decompiler*", http://members.tripod.com/~SourceTec/jasmine.htm

[M98] A. Magnaghi. *Java Native-Thread Support on Sun Enterprise Servers*, Internal Report, The University of Tokyo, June 1998

[M99] A. Magnaghi et al. *The Visitor Pattern: an Applicability Example of LCA*, Internal Report, The University of Tokyo, February 1999

[MST98] A. Magnaghi, S. Sakai, H. Tanaka. An Inter-procedural Approach for Optimizations of Java Programs. In *Proceedings of Information Processing Society of Japan*, 1998

[MST99] A. Magnaghi, S. Sakai, H. Tanaka. Inter-procedural 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

[MH98] A. McManus, J. Hunt. The Need for Speed. In *SIGS Java Report*, Vol. 3, No. 5, pp.39-44, May 1998

[Mi78] R. Milner. A Theory of Type Polymorphism in Programming. In *Journal of Computer and Systems Sciences*, pp. 348-375, 1978

[RD97] M. Rinard, P. Diniz. Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers. In *ACM Transactions on Programming Languages and Systems*, Vol. 19, No. 6, pp. 942-991, November 1997

[V98] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill, 1998