Dimensions of Precision in Reference Analysis of Object-oriented Programming Languages ⋆

Dr. Barbara G. Ryder

Division of Computer and Information Sciences
Rutgers University
New Brunswick, New Jersey 08903 USA

Abstract. There has been approximately a ten year history of reference analyses for object-oriented programming languages. Approaches vary as to how different analyses account for program execution flow, how they capture calling context, and how they model objects, reference variables and the possible calling structure of the program. A taxonomy of analysis dimensions that affect precision (and cost) will be presented and illustrated by examples of existing reference analysis techniques.

1 Introduction

Almost 25 years after the introduction of Smalltalk-80, object-orientation is a mature, accepted technology. Therefore, it is appropriate now to take a historical look at analyses for object-oriented programming languages, examining how they have evolved, particularly with respect to ensuring sufficient precision, while preserving practical cost. Object-oriented languages allow the building of software from parts, encouraging code reuse and encapsulation through the mechanisms of inheritance and polymorphism. Commonly, object-oriented languages also allow dynamic binding of method calls, dynamic loading of new classes, and querying of program semantics at runtime using reflection.

To understand the control flow in an object-oriented program requires knowledge of the types of objects which can act as receivers for dynamic method dispatches. Thus, to know the possible calling structure in a program, the set of possible object types must be known; but to determine the set of possible types of objects, some representation of possible interprocedural calling structure must be used. Essentially the program representation (i.e., the calling structure) is dependent on the analysis solution and *vice versa*. This interdependent relationship makes analysis of object-oriented languages quite different from that of procedural languages [18]. In addition, dynamic class loading may require a runtime recalculation of some analysis results [42].

Therefore, there is a fundamental need for *reference analysis* in any analysis of object-oriented languages, in order to obtain a program representation. The term *reference analysis* is used to define an analysis that seeks to determine information about the set of objects to which a reference variable or field may point during execution. This study will discuss the dimensions of reference analysis which lead to variations in the

⋆ This research was sponsored, in part, by NSF grant CCR: CCR-9900988.

precision obtained in its solution. Examining these dimensions will illustrate similarities and differences between analyses, and identify sources of precision and tradeoffs in cost. Examples of these dimensions will be discussed in the context of different analyses. Open issues not yet fully addressed will also be discussed.

Optimizing compilers and program development tools, such as test harnesses, refactoring tools, semantic browsers for program understanding, and change impact analysis tools, use reference analysis and its client analyses (e.g., side effect analysis, escape analysis, def-use analysis). There are real tradeoffs between the usability of the analysis results in terms of precision and the cost of obtaining them, the time and memory required. These tradeoffs are especially significant for interactive tools. It is important, therefore, to validate analyses by measures corresponding to their eventual use in client applications, even if a full application is not built. Use of benchmark suites which allow evaluation of different techniques using the same input data-sets is crucial; more efforts for building such suites should be encouraged by the research community.

This study is not an attempt at an encyclopedic categorization of all analyses of object-oriented languages; rather the goal is to enumerate characteristics which differentiate the precision (and affect the cost) of different analyses and to give examples of different design choices in existing analyses. There are other papers which cover many of the existing reference analyses and compare and contrast them [18, 28]. This paper, by design, will be incomplete in the set of analyses mentioned.

Overview. Section 2 presents the dimensions of precision to be discussed and explain them intuitively. Section 3 discusses each dimension more fully, cites reference analysis examples of choices with respect to that dimension, and then discusses the relative influence of that dimension on reference analysis precision (and cost). Section 4 presents some open issues with regard to analysis of object-oriented programs. Finally, Section 5 summarizes these discussions.

2 Preliminaries

Recall that *reference analysis* determines information about the set of objects to which a reference variable or reference field may point during execution. Historically, various kinds of reference analyses have been developed. *Class analysis* usually involves calculation of the set of classes (i.e., types) associated with the objects to which a reference variable can refer during program execution; this information has been used commonly for call graph construction. Intuitively, class analysis can be thought of as a reference analysis in which one abstract object represents all the instantiations of a class. *Points-to analysis* of object-oriented languages is a term used often for analyses that distinguish different instantiations of a class (i.e., different objects). Points-to analyses [23, 33] are often designed as extensions to earlier pointer analyses for C [43, 3]. *Refers-to analysis* [45] is a term sometimes used to distinguish a points-to analysis for object-oriented languages from a points-to analysis for general-purpose pointers in C. The term *reference analysis* will be used as denoting all of these analyses for the remainder of this paper.

Most of the analyses used here as examples are reference analyses which are fundamental to understanding the semantics of object-oriented programs. Recall from Sec-

tion 1, that the interprocedural control flow of an object-oriented program cannot be known without the results of these analyses. Thus, other analyses – including side effect, escape,[1] def-uses, and redundant synchronization analyses – require a reference analysis in order to obtain a representation of interprocedural flow for a program. Thus, reference analyses are crucial to any analysis of object-oriented code.

The characteristics or dimensions that directly affect reference analysis precision are presented below. The design of a specific analysis can be described by choices in each of these dimensions. After the brief description here, in Section 3 each dimension and the possible choices it offers will be illustrated in the context of existing analyses.

- **Flow sensitivity**. Informally, if an analysis is *flow-sensitive*, then it takes into account the order of execution of statements in a program; otherwise, the analysis is called *flow-insensitive*. Flow-sensitive analyses perform *strong updates* (or kills); for example, this occurs when a definition of a variable supersedes a previous definition. The classical dataflow analyses [2, 25, 19] are flow-sensitive, as are classical abstract interpretations [12].
- **Context sensitivity**. Informally, if an analysis distinguishes between different calling contexts of method, then it is *context-sensitive*; otherwise, the analysis is called *context-insensitive*. Classically, there are two approaches for embedding context sensitivity in an analysis, a call string approach and a functional approach [38]. *Call strings* refer to using the top sequence on the call stack to distinguish the interprocedural context of dataflow information; the idea is that dataflow information tagged with consistent call strings corresponds to the same calling context (which is being distinguished). The functional approach involves embedding information about program state at the call site, and using that to distinguish calls from one another.
- **Program representation (i.e., calling structure).** Because of the interdependence between possible program calling structure and reference analysis solution in object-oriented languages, there are two approaches to constructing an interprocedural representation for an object-oriented program. A simple analysis can obtain an approximation of the calling structure to be used by the subsequent reference analysis. Sometimes this representation is then updated using the analysis solution, when certain edges have been shown to be infeasible. Alternatively, the possible call structure can be calculated lazily, on-the-fly, interleaved with reference analysis steps. The latter approach only includes those methods in the call graph which are *reachable* from program start according to the analysis solution.
- **Object representation.** This dimension concerns the elements in the analysis solution. Sometimes one abstract object is used to represent all instantiations of a class. Sometimes a representative of each creation site (e.g., *new*) is used to represent all objects created at that site. These two naming schemes are those most often used, although alternatives exist.
- **Field sensitivity.** An object or an abstract object may have its fields represented distinctly in the solution; this is called a *field-sensitive* analysis. If the fields in an

[1] Sometimes reference analysis is performed interleaved with the client analysis, for example [36, 11, 5, 49, 6].

object are indistinguishable with respect to what they reference, then the analysis is termed *field-insensitive*.

- **Reference representation.** This dimension concerns whether each reference representative corresponds to a unique reference variable or to groups of references, and whether the representative is associated with the entire program or with sections of the program (e.g., a method). This dimension relates to reference variables as object representation relates to objects.
- **Directionality.** Generally, flow-insensitive analyses treat an assignment `x = y` as *directional*, meaning information flows from `y` to `x`, or alternatively as *symmetric* meaning subsequent to the assignment, the same information is associated with `x` and `y`. These approaches can be formulated in terms of constraints which are *unification* (i.e., equality) constraints for symmetric analyses or *inclusion* (i.e., subset) constraints for directional analyses.

By varying analysis algorithm design in each of these dimensions, it is possible to affect the precision of the resulting solution. The key for any application is to select an effective set of choices that provide sufficient precision at practical cost.

3 Dimensions of Analysis Precision

There is much in common in the design of pointer analysis for C programs and some reference analyses for Java and C^{++}. Both flow-sensitive and context-sensitive techniques were used in pointer analysis [21]. In general, the analysis community decided that flow sensitivity was not scalable to large programs. Context sensitivity for C pointer analysis also was explored independent of flow sensitivity [17, 22, 35], but the verdict on its effectiveness is less clear. Keeping calling contexts distinguished is of varying importance in a C program, depending on programming style, whereas in object-oriented codes it seems crucial for obtaining high precision for problems needing dependence information, for example. In general, program representation in pointer analysis was on the statement level, represented by an abstract syntax tree or flow graph. Solution methodologies included constraint-based techniques and dataflow approaches that allowed both context-sensitive and context-insensitive formulations.

Some reference analyses calculated finite sets of types (i.e., classes) for reference variables, that characterized the objects to which they may refer. The prototypical problem for which these analyses were used is *call graph construction* (i.e., dynamic dispatch resolution). More recently, reference analyses have been used for discovering redundant synchronizations, escaping objects and side-effect analysis [11, 5, 6, 49, 41, 36, 33, 26, 23, 32]. These client analyses require more precision than call graph construction and thus, provide interesting different applications for analysis comparison.

Recall that the dimensions of analysis precision include: *flow sensitivity, context sensitivity, program representation, object representation, field sensitivity, reference representation* and *directionality*. In the following discussions, each dimension is considered and examples of reference analyses using specific choices for each dimension are cited. The goal here is to better understand how these analyses differ, not to select a *best* reference analysis.

3.1 Flow sensitivity

An early example of a flow- and context-sensitive reference analysis was presented by Chatterjee et. al [8]. This algorithm was designed as a backwards and forwards dataflow propagation on the strongly connected component decomposition of the approximate calling structure of the program. Although the successful experiments performed on programs written in a subset of C^{++} showed excellent precision of the reference solution obtained, there were scalability problems with the approach.

Whaley and Lam [48] and Diwan et. al [15] designed techniques that perform a flow-sensitive analysis within each method, allowing kills in cases where an assignment is unambiguous. For example, an assignment `p = q` does allow the algorithm to re-initialize the set of objects to which `p` may point here only to those objects to which `q` may point; this is an example of a *kill* assignment. By contrast, the assignment `p.f = q` is not a *kill* assignment because the object whose `f` field is being mutated is not necessarily unique. This use of flow sensitivity has the potential of greater precision, but this potential has not yet been demonstrated for a specific analysis application.

Given that object-oriented codes generally have small methods, the expected payoff of flow sensitivity on analysis precision would seem minimal. Concerns about scalability have resulted in many analyses abandoning the use of flow sensitivity, in favor of some form of context sensitivity.

3.2 Context sensitivity

Classically, there are two approaches to embedding context sensitivity in an analysis, using call strings and functions [38]. *Call strings* refer to using the top sequence on the runtime call stack to distinguish the interprocedural context of dataflow information; the idea is only to combine dataflow information tagged with consistent call strings (that is, dataflow information that may exist co-temporally during execution). Work in control flow analysis by Shivers [39], originally aimed at functional programming languages, is related conceptually to the Sharir and Pnueli call string approach. These control flow analyses are distinguished by the amount of calling context remembered; the analyses are called *k-CFA*, where k indicates the length of the call string maintained. The functional approach uses information about the state of computation at a call site to distinguish different call sites. Some reference analyses that solely use inheritance hierarchy information are context-insensitive [15, 14, 4]; some later, more precise analyses [2] are also context-insensitive to ensure scalability (according to their authors) [15, 47, 45, 33, 23, 48].

Other reference analyses use both the call-string and functional notions of classical context sensitivity [38]. Palsberg and Schwartzbach presented a 1-CFA reference analysis [29]. Plevyak and Chien [30] described an incremental approach to context sensitivity, which allows them to refine an original analysis when more context is needed to distinguish parts of a solution due to different call sites; their approach seems to combine the call string and functional approaches in order to handle both polymorphic functions and polymorphic containers. Agesen [1] sought to improve upon the Palsberg and Schwartzbach algorithm by specifically adding a functional notion of context

[2] which incorporate interprocedural flow through parameters

sensitivity. In his Cartesian product algorithm, he defined different contexts using tuples of parameter types that could access a method; these tuples were computed lazily and memoized for possible sharing between call sites. Grove and Chambers [18] also explored the two notions of context sensitivity in different algorithms, using both call strings and tuples of parameter type sets (analogous to Agesen's Cartesian product algorithm). Milanova et. al defined *object sensitivity* [26], a functional approach that effectively allows differentiation of method calls by distinct receiver object.

This active experimentation with context sensitivity demonstrates its perceived importance in the analysis community as enabling a more precise analysis. The prevalence of method calls in an object-oriented program leads to the expectation that more precise analyses for object-oriented languages can be obtained by picking the 'right' practical embedding of context sensitivity.

3.3 Program representation (i.e., calling structure)

Early reference analyses [15, 14, 4] were used to provide a static call graph that initialized computation for a subsequent, more precise reference analysis [29, 8, 23, 45]. Other analyses constructed the call graph lazily, as new call edges became known due to discovery of a new object being referred to [27, 31, 48, 33, 26, 23]. Grove and Chambers discuss the relative merits of both approaches and conclude that the lazy construction approach is preferred [18].

Clearly, the trend is to use the lazy construction approach so that the analysis includes a reachability calculation for further accuracy. This can be especially significant when programs are built using libraries; often only a few methods from a library are actually accessed and excluding unused methods can significantly affect analysis cost as well as precision.

3.4 Object representation

Representation choices in analyses often are directly related to issues of precision. There are two common choices for reference analysis. First, an analysis can use one abstract object per class to represent all possible instantiations of that class. Second, objects can be identified by their creation site; in this case, all objects created by a specific *new* statement are represented by the same abstract object. Usually the reason for selecting the first representation over the second is efficiency, since it clearly leads to less precise solutions.

The early reference analyses [15, 14, 4, 29] all used one abstract object per class. Some later reference analyses made this same choice [45, 47] citing reasons of scalability. Other analyses used creation sites to identify equivalence classes of objects each corresponding to one representative object in the reference analysis solution [33, 18, 23, 48]. There are other, more precise object naming schemes which establish finer-grained equivalence classes for objects [18, 26, 27, 31, 24].

While the use of one abstract object per class may suffice for call graph construction, for richer semantic analyses (e.g., side effect, def-use and escape analyses) the use of a representative for each object creation site is preferable.

3.5 Field sensitivity

Another representation issue is whether or not to preserve information associated with distinct reference fields in an object. One study [33] indicated that not distinguishing object fields may result in imprecision and increased analysis cost. The majority of analyses which use representative objects also distinguish fields because of this precision improvement.

It is interesting that Liang et. al [23] reported that there appeared to be little difference in precision when fields were used either with an abstract object per class or a representative object per creation site with inclusion constraints; more experimentation is needed to understand more fully the separate effects of each of the dimensions involved in these experiments.

3.6 Reference representation

This dimension concerns to whether or not each reference is represented by a unique representative throughout the entire program. For most reference analyses, this is the case. Sometimes, all the references of the same type are represented by one abstract reference of that type [45]. Alternatively there can be one abstract reference per method [47]. These two alternatives reduce the number of references in the solution, so that the analysis is more efficient.

Tip and Palsberg [47] explored many dimensions of reference representation. Several analyses were defined whose precision lay between RTA [4] and 0-CFA [39, 18]. They experimented with abstract objects without fields and an unique reference representation (i.e., CTA analysis), abstract objects with fields and an unique reference representation (i.e., MTA analysis), abstract objects and one abstract reference per method (i.e., FTA analysis), and abstract objects with fields with one abstract reference per method (i.e., XTA analysis). The XTA analysis resulted in the best performance and precision tradeoff for call graph construction, their target application.

The VTA analysis [45] of the SABLE research project at McGill University specifically contrasted the use of unique reference representatives versus the use of one abstract reference representative per class. The latter was found to be too imprecise to be of use.

3.7 Directionality

Reference analysis usually is formulated as constraints that describe the sets of objects to which a reference can point and how these sets are mutated by the semantics of various assignments to (and through) reference variables and fields. There is a significant precision difference between symmetric and directional reference analyses, which are formulated as unification constraints or inclusion constraints, respectively. The unification constraints are similar to those used in Steensgaard's pointer analysis for C [43]; the inclusion constraints are similar to those used by Andersen's pointer analysis for C [3] .

Precision differences between these constraint formulations for C pointer analysis were explained by Shapiro and Horwitz [37]. Considering the pointer assignment statement `p = q`, the unification analysis will union the points-to set of `p` with the points-to

set of q, effectively saying both pointer variables can point to the same set of objects after this assignment; this union is accomplished recursively, so that if *p is also a pointer then its set is unioned to that of *q. An inclusion analysis will conclude after this same assignment statement that the points-to set of p includes the points-to set of q, maintaining the direction of the assignment.[3] Similar arguments can show why inclusion constraints can be expected to yield more a precise reference analysis solution than unification constraints as was shown by Liang et. al [23]. Ruf developed a context-sensitive analysis based on unification constraints as part of a redundant synchronization removal algorithm [36].

Solution procedures for both types of constraints are polynomial time (in the size of the constraint set), but unification constraints can be solved in almost linear worst case cost [43], whereas inclusion constraints have cubic worst case cost. Although these worst case costs are not necessarily experienced in practice, this difference has been considered significant until recently when newer techniques have shown that inclusion constraints in reference analysis can be solved effectively in practice [16, 44, 20, 33, 48, 26]. Thus, it seems that the increased precision of inclusion constraints are worth the possible additional cost, but this may depend on the accuracy needs of the specific analysis application.

4 Open Issues

There still are open issues in the analysis of object-oriented languages for which solutions must be found. Some of them are listed below.

- **Reflection.** Programs with reflection constructs can create objects, generate method calls, and access fields of objects at runtime whose declared types cannot be known at compile-time. This creates problems for analyses, because the program is effectively incomplete at compile-time. Most analyses transform a program to account for the effects of reflection before analyzing the program empirically.
- **Native methods.** Calls to native methods (i.e., methods not written in the object-oriented language, often written in C) may have dataflow consequences that must be taken into account by a safe analysis.
- **Exceptions.** In Java programs checked exceptions appear explicitly and unchecked exceptions appear implicitly; both can affect flow of control. Since obtaining a good approximation to possible program control flow is a requirement for a precise analysis, some approaches have been tried [40, 7, 9, 10], but this is still an open problem.
- **Dynamic class loading.** Dynamic class loading may invalidate the dynamic dispatch function previously calculated by a reference analysis [42]. This suggests the possibility of designing an incremental reference analysis; however, it will be difficult to determine the previously-derived information that has been invalidated.
- **Incomplete programs.** Often object-oriented programs are either libraries or library clients, and thus partial programs. Analysis of such codes has been addressed [47,

[3] A combination of these constraints was used for C pointer analysis by Das and showed good precision in empirical experiments for practical cost [13].

46, 34], but more work is needed. Having a good model for partial program analysis for object-oriented languages may allow analyses to be developed for component-based programs; it is likely however, that some reliance on component-provider-based information may be necessary.

– **Benchmarks.** It is very important to use benchmark suites in testing analyses, because reproducibility is required for strong empirical validation. Some researchers have used the SPEC compiler benchmarks,[4] or have shared collected benchmark programs.[5]

5 Conclusions

Having presented an overview of the dimensions of precision in reference analysis of object-oriented languages, the current challenge in analysis research is to match the *right* analyses to specific client applications, with appropriate cost and precision. This task is aided by a clear understanding of the role of each dimension in the effectiveness of the resulting analysis solution.

The nature of object-oriented languages is that programs are constructed from many small methods and that method calls (with possible recursion) are the primary control flow structure used. Thus, it is critical to include some type of context sensitivity in an analysis, to obtain sufficient precision for tasks beyond simple dynamic dispatch. Arguably, the functional approach offers a more practical mechanism than the call-string approach embodied in k-CFA analyses and it seems to be more cost effective. It is also clear that a solution procedure using inclusion constraints can be practical and delivers increased precision over cheaper unification constraint resolution.

These opinions are held after experimentation by the community with many dimensions of analysis precision. However, no one analysis *fits* every application and many of the analyses discussed will be applicable to specific problems because their precision is sufficient *to do the job*. A remaining open question is *Can the analysis community deliver useful analyses for a problem at practical cost?* The answer is yet to be determined.

Acknowledgements I am very grateful to my graduate student, Ana Milanova, for her insightful discussions about this topic and her enormous help with editing this paper. I also wish to thank Dr. Atanas Rountev for his editing feedback on this paper and many valuable discussions.

[4] http://www.specbench.org/osg/jvm98

[5] Examples include the programs in [47] and those found at http://prolangs.rutgers.edu/ http://www.sable.mcgill.ca/ashes/

References

1. O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *European Conference on Object-Oriented Programming*, pages 2–26, 1995.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
3. L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen, 1994. Also available as DIKU report 94/19.
4. D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, 1996.
5. Bruno Blanchet. Escape analysis for object-oriented languages: application to java. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 20–34. ACM Press, 1999.
6. Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in java. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 35–46. ACM Press, 1999.
7. R. Chatterjee. *Modular Data-flow Analysis of Statically Typed Object-oriented Programming Languages*. PhD thesis, Department of Computer Science, Rutgers University, October 1999.
8. R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *Symposium on Principles of Programming Languages*, pages 133–146, 1999.
9. Ramkrishna Chatterjee and Barbara G Ryder. Data-flow-based testing of object-oriented libraries. Department of Computer Science Technical Report DCS-TR-382, Rutgers University, March 1999.
10. Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of java programs. In *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 21–31. ACM Press, 1999.
11. Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19. ACM Press, 1999.
12. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points. In *Conference Record of the Fourth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, January 1977.
13. Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 35–46, June 2000.
14. J. Dean, D. Grove, and C. Chambers. Optimizations of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101, 1995.
15. A. Diwan, J.Eliot B. Moss, and K. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 292–305, 1996.
16. Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation*, pages 85–96. ACM Press, 1998.

17. J. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for c. In *in Proceedings of International Symposium on Static Analysis*, April 2000.
18. David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6), 2001.
19. M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, 1977.
20. Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: a million lines of c code in a second. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, pages 254–263, 2001.
21. Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN - SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, June 2001.
22. D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of the 7th Annual ACM SIGSOFT Symposium on the Foundations of Software Engineering*, LNCS 1687, pages 199–215, September 1999.
23. D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 73–79, June 2001.
24. Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Evaluating the precision of static reference analysis using profiling. In *Proceedings of the international symposium on Software testing and analysis*, pages 22–32. ACM Press, 2002.
25. T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. *Acta Informatica*, 28:121–163, 1990.
26. A. Milanova, A. Rountev, and B. Ryder. Parameterized object-sensitivity for points-to and side-effect analyses for Java. In *International Symposium on Software Testing and Analysis*, pages 1–11, 2002.
27. N. Oxhoj, J. Palsberg, and M. Schwartzbach. Making type inference practical. In *European Conference on Object-Oriented Programming*, pages 329–349, 1992.
28. J. Palsberg. Type-based analysis and applications (invited talk). In *Proceedings of 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 20–27. ACM Press, July 2001.
29. J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–161, 1991.
30. J. Plevyak and A. Chien. Precise concrete type inference for object oriented languages. In *Proceeding of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '94)*, pages 324–340, October 1994.
31. J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–340, 1994.
32. C. Razafimahefa. A study of side-effect analyses for Java. Master's thesis, McGill University, December 1999.
33. A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 43–55, October 2001.
34. A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java software. In *International Conference on Software Engineering*, 2003.
35. E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, June 1995.
36. E. Ruf. Effective synchronization removal for Java. In *Conference on Programming Language Design and Implementation*, pages 208–218, 2000.

37. M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the Twenty-fourth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 1–14, January 1997.
38. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.
39. Olin Shivers. *Control-flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie-Mellon University School of Computer Science, 1991.
40. S. Sinha and M.J. Harrold. Analysis of programs that contain exception-handling constructs. In *Proceedings of International Conference on Software Maintenance*, pages 348–357, November 1998.
41. Amie L. Souter and Lori L. Pollock. Omen: A strategy for testing object-oriented software. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 49–59. ACM Press, 2000.
42. Vugranam C. Sreedhar, Michael Burke, and Jong-Deok Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 196–207, 2000.
43. Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the Twenty-third Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 32–41, 1996.
44. Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Conference Record of the Twenty-seventh Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, 2000.
45. V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 264–280, 2000.
46. Peter F. Sweeney and Frank Tip. Extracting library-based object-oriented applications. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 98–107. ACM Press, 2000.
47. F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 281–293, 2000.
48. J. Whaley and M. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Static Analysis Symposium*, 2002.
49. John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 187–206. ACM Press, 1999.