Recovering Structural Information for Better Static Analysis

George Balatsouras[*]

National and Kapodistrian University of Athens
Department of Informatics and Telecommunications
gbalats@di.uoa.gr

Abstract. To reach a truly broad level of program understanding, static analysis techniques need to create an abstraction of memory that covers all possible executions. Such abstract models may quickly degenerate after losing essential structural information about the memory objects they describe, due to the use of specific programming idioms and language features, or because of practical analysis limitations. In many cases, some of the lost memory structure may be retrieved, though it requires complex inference that takes advantage of indirect uses of types. Such recovered structural information may, then, greatly benefit static analysis. This dissertation shows how we can recover structural information, first (i) in the context of C/C++, and next, in the context of higher-level languages without direct memory access, like Java, where we identify two primary causes of losing memory structure: (ii) the use of reflection, and (iii) analysis of partial programs. We show that, in all cases, the recovered structural information greatly benefits static analysis on the program.

Keywords: Pointer Analysis; Object-Oriented Programming; Type Hierarchy; Reflection

1 Introduction

The most promising and powerful of existing static analysis techniques rely on the creation of some *abstract memory model* of the program. What objects will the memory contain, at some state of execution? What will their structure be like? A faithful abstract representation of the actual memory is, however, a demanding task; its precision often decisive for the value of whatever the static analysis is aiming to eventually compute (be it the identification of complex bug patterns or the opportunities for effective optimizations).

Thesis.

There is *implicit structural information* in the program, about the memory it will allocate, that can improve the quality of the abstract memory

[*] Dissertation Advisor: Yannis Smaragdakis, Professor

model constructed by static analysis. This structural information is not readily available, but may be recovered via inference, primarily by tracking the use of types in the program.

We provide a number of techniques that recover such lost memory structure, in two different settings: (1) in C/C++ programs, as a typical case of low-level code with direct memory access, where the program's memory structure is often lost due to specific programming idioms and the inherent low-level nature of the language, and (2) in Java programs, where, despite the high-level nature of the language, structural information may be lost (a) for *partial programs* (i.e., libraries or any programs that lack some of their parts), which, in the form of Java Archives (JARs), constitute the main distributable code entity of this managed language, or (b) due to Java's *reflection* mechanism, which allows runtime inspection of classes, interfaces, fields and methods, and can be used to instantiate new objects, invoke methods, get/set field values, and so on, without exact static type information (e.g., the name of the method to be invoked can be created dynamically using plain string operations).

2 Structure-Sensitive Points-To Analysis for C and C++

Points-to analysis computes an abstract model of the memory that is used to answer the following query: *What can a pointer variable point-to, i.e., what can its value be when dereferenced during program execution?* This query serves as the cornerstone of many other static analyses aiming to enhance program understanding or assist in bug discovery (e.g., deadlock detection), by computing higher-level relations that derive from the computed points-to sets. In the literature, one can find a multitude of points-to analyses with varying degrees of precision and speed.

One of the most popular families of pointer analysis algorithms, *inclusion-based* analyses (or Andersen-style analyses [3]), originally targeted the C language, but has been extended over time and successfully applied to higher-level object-oriented languages, such as Java [6,7,21,25,29]. Surprisingly, precision-enhancing features that are common practice in the analysis of Java programs, such as field sensitivity or online call-graph construction are absent in many analyses of C/C++ [12,15,30,14,8,13].

In the case of field sensitivity, the reason behind its frequent omission when analyzing C is that it is much harder to implement correctly than in Java. As noted by Pearce et al. [24], the crucial difference is that, in C/C++, it is possible to have the address of a field taken, stored to some pointer, and then dereferenced later, at an arbitrarily distant program point. In contrast, Java does not permit taking the address of a field; one can only load or store to some field directly. Hence, `load/store` instructions in Java bytecode (or any equivalent IR) need an extra field specifier, whereas in C/C++ intermediate representations (e.g., LLVM bitcode) `load/store` requires only a single address operand. The precise field affected is not explicit, but only possibly computed by the analysis itself.

The effect of such difference in the underlying IRs, as far as pointer analysis is concerned, is far from trivial. In C, the computed points-to sets have an expanded domain, since now the analysis must be able to express that a variable `p` *at some offset* `i` may point-to another variable `q` *at some offset* `j`, with these offsets corresponding to either field components or array elements.

The best-documented approach on how to incorporate field sensitivity in a C/C++ points-to analysis is that of Pearce et al. [23,24]. The authors extend the constraint-graph of the analysis by adding (positive) weights to edges; the weights correspond to the respective field indices. For instance, the instruction "`q = &(p->f`$_{\mathtt{i}}$`)`" would be encoded as a constraint $q \supseteq p + i$. However, this approach does not take types into account. In fact, types are not even statically available at all allocation sites, since most standard C allocation routines are type-agnostic and return byte arrays that are cast to the correct type at a later point (e.g., `malloc()`, `realloc()`, `calloc()`). Thus, field i is represented with no regard to the type of its base object, even when this base object abstracts a number of concrete objects of different types. The lack of type information for abstract objects is a great source of imprecision, since it results in a prohibitive number of spurious points-to inferences.

We argue that type information is an essential part in increasing analysis precision, even when it is not readily available. The abstract object types should be rigorously recorded in all cases, especially when indexing fields, and used to filter the points-to sets. In this spirit, we present a *structure-sensitive* analysis for C/C++ that employs a number of techniques in this direction, aiming to retrieve high-level structure information for abstract objects in order to increase analysis precision:

1. First, the analysis records the type of an abstract object when this type is available at the allocation site. This is the case with stack allocations, global variables, and calls to C++'s `new()` heap allocation routine.

2. In cases where the type is not available (as in a call to `malloc()`), the analysis deviates from the allocation-site abstraction and creates multiple abstract objects per allocation site: one for every type that the object could have. Thus, each abstract object of type `T` now represents the set of all concrete objects of type `T` allocated at this site. To determine the possible types for a given allocation site, the analysis creates a special type-less object and records the cast instructions it flows to (i.e., the types it is cast to), using the existing points-to analysis. This is similar to the use-based *back-propagation* technique used in past work [17,19,27], in a completely different context—handling Java reflection.

3. The field components of abstract objects are represented as abstract objects themselves, as long as their type can be determined. That is, an abstract object `SO` of struct type `S` will trigger the creation of abstract object `SO.f`$_{\mathtt{i}}$, for each field `f`$_{\mathtt{i}}$ in `S`. (The aforementioned special objects trigger no such field component creation, since they are typeless.) Thus, the recursive cre-

ation of subobjects is bounded by the type system, which does not allow the declaration of types of infinite size.

4. Finally, the analysis treats array elements similarly to field components (i.e., by representing them as distinct abstract objects, if we can determine their type), as long as their respective indices statically appear in the source code. That is, an abstract object `AO` of array type `[T×N]` will trigger the creation of abstract object `AO[c]`, if the constant `c` is used to index into type `[T×N]`. The object `AO[*]` is also created, to account for indexing at unknown (variable) indices.

The last point offers some form of array-sensitivity as well and is crucial for analyzing C++ code, lowered to an intermediate representation such as LLVM bitcode, in which all the object-oriented features have been translated away. To be able to resolve virtual calls, an analysis must precisely reason about the exact v-table index that a variable may point to, and the method that such an index may itself point-to. That is, a precise analysis should not merge the points-to sets of distinct indices of v-tables.

We offer an implementation of our approach over the full LLVM bitcode intermediate language, in the form of a new static analysis tool, `cclyzer`[1]. We show that our approach yields much higher precision than past analyses, allowing accurate distinctions between subobjects, v-table entries, array components, and more. Especially for C++ programs, this precision is invaluable for a realistic analysis. Compared to the state-of-the-art past approach, our techniques exhibit substantially better precision along multiple metrics and realistic benchmarks (e.g., 40+% more variables with a single points-to target).

3 More Sound Static Handling of Java Reflection

Moving to higher-level languages, like Java, we note that essential structural information is often lost in Java programs too, yet for different reasons. A source of analysis imprecision, especially in determining the types of abstract objects constructed by the analysis, lies in the use of Java's reflection mechanism: the ability to inspect and dynamically retrieve classes, methods, attributes, etc. at runtime.

By using the Reflection API, Java programs can encompass dynamic behavior. However, statically reasoning about the behavior of software that uses reflection can be especially cumbersome. Unfortunately, reflection is ubiquitous in large Java programs. When a Java program accesses a class by supplying its name as a run-time string, via the `Class.forName` library call, the static analysis has very few available courses of action: It needs to either conservatively over-approximate (e.g., assume that *any* class can be accessed, possibly limiting the set later, after the returned object is used), or to perform a string analysis that will allow it to infer the contents of the `forName` string argument. Both

[1] `cclyzer` is publicly available at `https://github.com/plast-lab/cclyzer`

options can be detrimental to the scalability of the analysis: the conservative over-approximation may never become constrained enough by further instructions to be feasible in practice; precise string analysis is impractical for programs of realistic size. It is telling that *no practical Java program analysis framework in existence handles reflection soundly* [18], although other language features are modeled soundly.[2]

Full soundness is not practically achievable, but it can still be approximated for the well-behaved reflection patterns encountered in regular, non-adversarial programs. Therefore, it makes sense to treat soundness as a continuous quantity: something to improve on, even though we cannot perfectly reach. To avoid confusion, we use the term *empirical soundness* for the quantification of how much of the dynamic behavior the static analysis covers. Computable metrics of empirical soundness can help quantify how close an analysis is to the fully sound result. Based on such metrics, one can make comparisons (e.g., "more sound") to describe soundness improvements.

The second challenge of handling reflection in a static analysis is *scalability*. The online documentation of the IBM WALA library [10] concisely summarizes the current state of the practice, for *points-to analysis* in the Java setting.

> *Reflection usage and the size of modern libraries/frameworks make it very difficult to scale flow-insensitive points-to analysis to modern Java programs. For example, with default settings,* WALA*'s pointer analyses cannot handle any program linked against the Java 6 standard libraries, due to extensive reflection in the libraries.*

The same caveats routinely appear in the research literature. Multiple published points-to analysis papers analyze well-known benchmarks with reflection disabled [28,16,1,2].

A representative quote [28] illustrates:

> *Hsqldb and jython could not be analyzed with reflection analysis enabled [...] —hsqldb cannot even be analyzed context-insensitively and jython cannot even be analyzed with the 1obj analysis. This is due to vast imprecision introduced when reflection methods are not filtered in any way by constant strings (for classes, fields, or methods) and the analysis infers a large number of reflection objects to flow to several variables. [...] For these two applications, our analysis has reflection reasoning disabled. Since hsqldb in the DaCapo benchmark code has its main functionality called via reflection, we had to configure its entry point manually.*

We describe an approach for handling reflection with improved empirical soundness (as measured against prior approaches and dynamic information), again, in the context of a points-to analysis. Our approach is based on the combination of string-flow and points-to analysis from past literature augmented with (a) substring analysis and modeling of partial string flow through string builder classes;

[2] In our context, *sound* = over-approximate, i.e., guaranteeing that all possible behaviors of reflection operations are modeled.

(b) new techniques for analyzing reflective entities based on information available at their use-sites. In experimental comparisons with prior approaches, we demonstrate a combination of both improved soundness (recovering the majority of missing call-graph edges) and increased performance. Our approach requires no manual configuration and achieves significantly higher empirical soundness without sacrificing scalability, for realistic benchmarks and libraries (DaCapo Bach and Java 7).

In experimental comparisons with the recent ELF system [17] (itself improving over the reflection analysis of the DOOP framework [7]), our algorithm discovers most of the call-graph edges missing (relative to a dynamic analysis) from ELF's reflection analysis. This improvement in empirical soundness is accompanied by *increased* performance relative to ELF, demonstrating that near-sound handling of reflection is often practically possible. Concretely, our work for reflection:

· introduces key techniques in static reflection handling that contribute greatly to empirical soundness. The techniques generalize past work from an intra-procedural to an inter-procedural setting and combine it with a string analysis;
· shows how scalability can be addressed with appropriate tuning of the above generalized techniques;
· thoroughly quantifies the empirical soundness of a static points-to analysis, compared to past approaches and to a dynamic analysis;
· is implemented and evaluated on top of an existing open framework (DOOP [7]).

4 Class Hierarchy Complementation for Java

Whole-program static analysis is essential for clients that require high-precision and a deeper understanding of program behavior. Modern applications of program analysis, such as large scale refactoring tools [9], race and deadlock detectors [22], and security vulnerability detectors [20,11], are virtually inconceivable without whole-program analysis.

For whole-program analysis to become truly practical, however, it needs to overcome several real-world challenges. One of the somewhat surprising real-world observations is that whole-program analysis requires the availability of much more than the "whole program". The analysis needs an overapproximation of what constitutes the program. Furthermore, this overapproximation is not merely what the analysis computes to be the "whole program" after it has completed executing. Instead, the overapproximation needs to be as conservative as required by any intermediate step of the analysis, which has not yet been able to tell, for instance, that some method is never called.

Consider the example of trying to analyze a program P that uses a third-party library L. Program P will likely only need small parts of L. However, other, entirely separate, parts of L may make use of a second library, L'. It is typically not possible to analyze P with a whole program analysis framework without also supplying the code not just for L but also for L', which is an unreasonable burden. In modern languages and runtime systems, L' is usually not necessary in order to either compile P or run it under any input. The problem is exacerbated in the current era of large-scale library reuse. In fact, it is often the case that the user is not even aware of the existence of L' until trying to analyze P.

Our research consists precisely of addressing such need in full generality. *Given a set of Java class and interface definitions, in bytecode form, we compute a "program complement", i.e., skeletal versions of any referenced missing classes and interfaces so that the combined result constitutes verifiable Java bytecode.*

To see why the problem has interesting depth and complexity, consider a simple fragment of Java bytecode and the constraints it induces. Our convention here is that single-letter class names at the lower end of the alphabet (`A`, `B`, ...) correspond to known types, while class names at the high end of the alphabet (`X`, `Y`, `Z`) denote phantom types. We present bytecode in a slightly condensed form, to make clear what method names or type names are referenced in every instruction.

```
public void foo(X, Y)
0: aload_2     // load on stack 2nd argument (of type Y)
1: aload_1     // load on stack 1st argument (of type X)
2: invokevirtual X.bar:(LA;)LZ; // method 'Z bar(A)' in X
3: invokevirtual B.baz:()V;     // method 'void baz()' in B
 ...
```

Although the above fragment is merely four bytecode instructions long, it induces several interesting constraints for our phantom types `X`, `Y`, and `Z`:

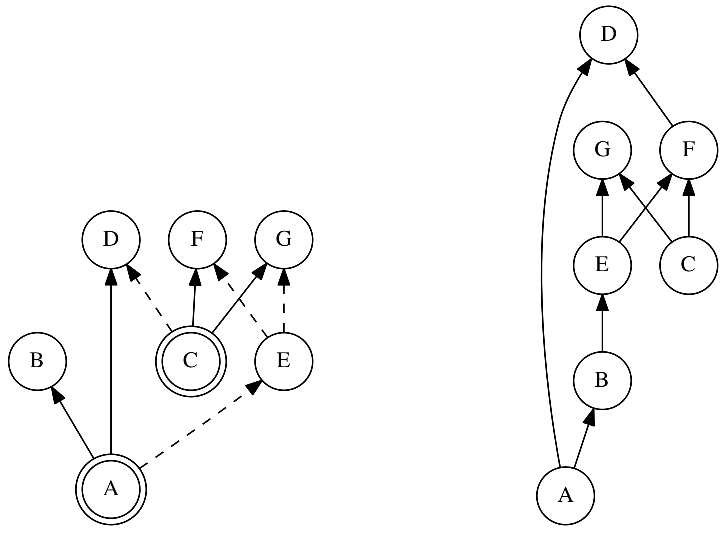
– `X` has to support a method `bar` accepting an argument of type `A` and returning a value of type `Z`.
– `Y` has to be a subtype of `A`, since an actual argument of declared type `Y` is passed to `bar`, which has a formal parameter of type `A`. This constraint also means that if `A` is known to be a class (and not an interface) then `Y` is also a class.
– `Z` has to be a subtype of `B`, since a method of `B` is invoked on an object of declared type `Z` (returned on top of the stack by the earlier invocation).

Our goal is to satisfy all such constraints and generate definitions of phantom types `X`, `Y`, and `Z` that are compatible with the bytecode that is available to the tool (i.e., exists in known classes). Compatibility with existing bytecode is defined as satisfying the requirements of the Java verifier, which concern type well-formedness.

Note that such definitions will contain essential parts of missing structural information for the phantom types: method and field members, as well as supertypes. Any subsequent static analysis that will operate on the types produced by complementation will create abstract objects that are much closer, in structure, to reality.

Of these constraints, the hardest to satisfy are those involving subtyping. Constraints on members (e.g., `X` has to contain a "`Z bar(A)`") are easy to satisfy by just adding type-correct dummy members to the generated classes. This means that the core of the general program complementation problem is solving the *class hierarchy complementation problem*: given a partial type hierarchy and a set of subtyping constraints, compute a complete type hierarchy that satisfies the subtyping constraints *without* changing the direct parents of known types.

Solving the hierarchy complementation problem, constitutes the main novelty of our approach. The problem appears to be fundamental, and even of a certain interest in purely graph-theoretic terms. For a representative special case, consider an object-oriented language with multiple inheritance (or, equivalently, an interface-only hierarchy in Java or C#). A partial hierarchy, augmented with constraints, can be represented as a graph, as shown in Figure 1a. The known part of the hierarchy is shown as double circles and solid edges. Unknown (i.e., missing) classes are shown as single circles. Dashed edges represent subtyping constraints, i.e., indirect subtyping relations that have to hold in the resulting hierarchy. In graph-theoretic terms, a dashed edge means that there is a path in the solution between the two endpoints. For instance, the dashed edge from C to D in Figure 1a means that the unknown part of the class hierarchy has a path from C to D. This path cannot be a direct edge from C to D, however: C is a known class, so the set of its supertypes is fixed.



(a) Constraint Graph (b) Solution

Fig. 1: Example of constraints in a multiple inheritance setting. Double-circles signify known classes, single circles signify unknown classes. Solid edges ("known edges") signify direct subtyping, dashed edges signify transitive subtyping.

In order to solve the above problem instance, we need to compute a directed acyclic graph (DAG) over the same nodes,[3] so that it preserves all known nodes and edges, and adds edges *only to unknown nodes* so that all dashed-edge constraints are satisfied. That is, the solution will not contain dashed edges (indirect subtyping relationships), but every dashed edge in the input will have a matching directed path in the solution graph. Figure 1b shows one such possible solution. As can be seen, solving the constraints (or determining that they are unsatisfiable) is not trivial. In this example, any solution has to include an edge from B to E, for reasons that are not immediately apparent. Accordingly, if we change the input of Figure 1a to include an edge from E to B, then the constraints are not satisfiable—any attempted solution introduces a cycle. The essence of the algorithmic difficulty of the problem (compared to, say, a simple topological sort) is that we cannot add extra direct parents to known classes A and C—any subtyping constraints over these types have to be satisfied via existing parent types. This corresponds directly to our high-level program requirement: we want to compute definitions for the missing types only, without changing existing code. For a language with single inheritance, the problem is similar, with one difference: the solution needs to be a tree instead of a DAG. (Of course, the input in Figure 1a already violates the tree property since it contains known nodes with multiple known parents.)

We provide algorithms to solve the hierarchy complementation problem in the single inheritance and multiple inheritance settings. We also show that the problem in a language such as Java, with single inheritance but multiple subtyping and distinguished class vs. interface types, can be decomposed into separate single- and multiple-subtyping instances. We implement our algorithms in a tool, JPhantom,[4] which complements partial Java bytecode programs so that the result is guaranteed to satisfy the Java verifier requirements. In a sense, JPhantom aims to recover structural information for phantom classes, via inference, by tracking their use in existing code. JPhantom is highly scalable and runs in mere seconds even for large input applications and complex constraints (with a maximum of 14s for a 19MB binary).

5 Conclusions

To summarize, we advocate that there are many opportunities in recovering implicit structural information about memory that can improve static analysis of programs, but require complex inference that takes advantage of indirect uses of types. We have examined three different scenarios to test and evaluate our thesis, regarding generic C/C++ programs, and Java programs that either use reflection or are missing parts of their code. In all cases, we where able to improve static analysis, by recovering memory structure that was not previously evident.

[3] Inventing extra nodes does not contribute to a solution in this problem.

[4] JPhantom is available online at `https://github.com/gbalats/jphantom`

6 Publications

The contents of this doctoral dissertation are based on the following published papers:

– *Structure-Sensitive Points-To Analysis for C and C++* [5]
– *More Sound Static Handling of Java Reflection* [27]
– *Class Hierarchy Complementation: Soundly Completing a Partial Type Graph* [4]
– *Pointer Analysis* [26]

References

1. Ali, K., Lhoták, O.: Application-only call graph construction. In: Proc. of the 26th European Conf. on Object-Oriented Programming. pp. 688–712. ECOOP '12, Springer (2012)
2. Ali, K., Lhoták, O.: Averroes: Whole-program analysis without the whole program. In: Proc. of the 27th European Conf. on Object-Oriented Programming. pp. 378–400. ECOOP '13, Springer (2013)
3. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. Ph.D. thesis, DIKU, University of Copenhagen (May 1994)
4. Balatsouras, G., Smaragdakis, Y.: Class hierarchy complementation: Soundly completing a partial type graph. In: Proc. of the 28th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications. pp. 515–532. OOPSLA '13, ACM, New York, NY, USA (2013)
5. Balatsouras, G., Smaragdakis, Y.: Structure-sensitive points-to analysis for C and C++. In: Proc. of the 23rd International Symp. on Static Analysis. SAS '16, Springer (2016)
6. Berndl, M., Lhoták, O., Qian, F., Hendren, L.J., Umanee, N.: Points-to analysis using BDDs. In: Proc. of the 2003 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 103–114. PLDI '03, ACM, New York, NY, USA (2003)
7. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications. OOPSLA '09, ACM, New York, NY, USA (2009)
8. Das, M.: Unification-based pointer analysis with directional assignments. In: Proc. of the 2000 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 35–46. PLDI '00, ACM, New York, NY, USA (2000)
9. Dig, D.: A refactoring approach to parallelism. IEEE Software 28(1), 17–22 (2011)
10. Fink, S.J., et al.: WALA UserGuide: PointerAnalysis. `http://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis`
11. Guarnieri, S., Livshits, B.: GateKeeper: mostly static enforcement of security and reliability policies for Javascript code. In: Proc. of the 18th USENIX Security Symposium. pp. 151–168. SSYM' 09, USENIX Association, Berkeley, CA, USA (2009), `http://dl.acm.org/citation.cfm?id=1855768.1855778`
12. Hardekopf, B., Lin, C.: The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In: Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 290–299. PLDI '07, ACM, New York, NY, USA (2007)

13. Hardekopf, B., Lin, C.: Exploiting pointer and location equivalence to optimize pointer analysis. In: Proc. of the 14th International Symp. on Static Analysis. pp. 265–280. SAS '07, Springer (2007)
14. Heintze, N., Tardieu, O.: Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In: Proc. of the 2001 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 254–263. PLDI '01, ACM, New York, NY, USA (2001)
15. Hind, M., Burke, M.G., Carini, P.R., Choi, J.: Interprocedural pointer alias analysis. ACM Trans. on Programming Languages and Systems 21(4), 848–894 (1999)
16. Kastrinis, G., Smaragdakis, Y.: Hybrid context-sensitivity for points-to analysis. In: Proc. of the 2013 ACM SIGPLAN Conf. on Programming Language Design and Implementation. PLDI '13, ACM, New York, NY, USA (2013)
17. Li, Y., Tan, T., Sui, Y., Xue, J.: Self-inferencing reflection resolution for Java. In: Proc. of the 28th European Conf. on Object-Oriented Programming. pp. 27–53. ECOOP '14, Springer (2014)
18. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.Y.E., Guyer, S.Z., Khedker, U.P., Møller, A., Vardoulakis, D.: In defense of soundiness: A manifesto. Communications of the ACM 58(2), 44–46 (Jan 2015)
19. Livshits, B., Whaley, J., Lam, M.S.: Reflection analysis for Java. In: Proc. of the 3rd Asian Symp. on Programming Languages and Systems. pp. 139–160. APLAS '05, Springer (2005)
20. Madsen, M., Livshits, B., Fanning, M.: Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In: Proc. of the ACM SIGSOFT International Symp. on the Foundations of Software Engineering. pp. 499–509. FSE '13, ACM (2013)
21. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to and side-effect analyses for Java. In: Proc. of the 2002 International Symp. on Software Testing and Analysis. pp. 1–11. ISSTA '02, ACM, New York, NY, USA (2002)
22. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 308–319. PLDI '06, ACM, New York, NY, USA (2006)
23. Pearce, D.J., Kelly, P.H.J., Hankin, C.: Efficient field-sensitive pointer analysis for C. In: Proc. of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. pp. 37–42. PASTE '04, ACM, New York, NY, USA (2004)
24. Pearce, D.J., Kelly, P.H.J., Hankin, C.: Efficient field-sensitive pointer analysis of C. ACM Trans. on Programming Languages and Systems 30(1) (2007)
25. Rountev, A., Milanova, A., Ryder, B.G.: Points-to analysis for Java using annotated constraints. In: Proc. of the 16th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications. pp. 43–55. OOPSLA '01, ACM, New York, NY, USA (2001)
26. Smaragdakis, Y., Balatsouras, G.: Pointer analysis. Foundations and Trends® in Programming Languages 2(1), 1–69 (2015), `http://dx.doi.org/10.1561/2500000014`
27. Smaragdakis, Y., Balatsouras, G., Kastrinis, G., Bravenboer, M.: More sound static handling of Java reflection. In: Proc. of the 13th Asian Symp. on Programming Languages and Systems. pp. 485–503. APLAS '15, Springer (2015)
28. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: Understanding object-sensitivity. In: Proc. of the 38th ACM SIGPLAN-SIGACT Symp.

on Principles of Programming Languages. pp. 17–30. POPL '11, ACM, New York, NY, USA (2011)

29. Whaley, J., Rinard, M.C.: Compositional pointer and escape analysis for Java programs. In: Proc. of the 14th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications. pp. 187–206. OOPSLA '99, ACM, New York, NY, USA (1999)
30. Zheng, X., Rugina, R.: Demand-driven alias analysis for C. In: Proc. of the 35th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. pp. 197–208. POPL '08, ACM, New York, NY, USA (2008)