Available online at www.sciencedirect.com
SCIENCE @ DIRECT®

ELSEVIER Electronic Notes in Theoretical Computer Science 131 (2005) 75–84

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Abstract Interpretation and Object-oriented Programming: Quo Vadis?

Francesco Logozzo[1]

École Polytechnique
F-91128 Palaiseau, France

Agostino Cortesi[2,3]

Dipartimento di Informatica
Università Ca' Foscari
I-30170 Venice, Italy

Abstract

The aim of this position paper is to draw a quick overview of the main contributions in abstract interpretation of object-oriented programs, and to draw possible lines of research in this field.

Keywords: Abstract interpretation, Object-oriented programming, static analysis

1 Introduction

Abstract Interpretation is a theory for static analysis of software systems that formalizes the notion of approximation and abstraction in a mathematical setting, and which is independent of particular languages and applications. Nevertheless, when looking at the literature produced in the last two decades

---

[1] E-mail: `Francesco.Logozzo@polytechnique.edu`

[2] E-mail: `cortesi@dsi.unive.it`

[3] Partially supported by MIUR FIRB grant n.RBAU018RCZ and by MIUR PRIN'04 grant n.2004013015

doi:10.1016/j.entcs.2005.01.024

(see the electronic version with extended bibliography of [14]), the amazingly rich suite of problems and solutions that fit in the abstract interpretation setting is often dependent both on the specific programming language and on the given property to be analyzed (that might also be language dependent). Since abstract interpretation has a very semantic-based character, it is not surprising that language paradigms with strong semantic foundations, e.g. functional and logic programming, have been in the past a very fertile test bed for the development of sophisticated abstract domains and specialized fixpoint algorithms. On the other hand, when looking at the contributions in the area of object-oriented programming, the picture is somehow still fragmented, and this may overshadow the great potentialities of abstract interpretation on the mainstream programming platforms where the OO paradigm is getting a leader position.

This paper is aimed at providing a general survey of existing literature on abstract interpretation for object-oriented languages, and draw a few hints on how the research in this field may get further advances.

2 What has been done...

First, let us try to revisit a few interesting contributions (no claim of being complete!) in the static analysis of object-oriented languages. They are mainly focused on optimization issues.

2.1 Class Analysis

A class analysis computes, for each program point `PP` and for each variable $\mathtt{x}_{\mathtt{PP}}$ a set of classes $\mathcal{C}_{\mathtt{x}_{\mathtt{PP}}}$ such that if in an execution of the program `x`, at program point `PP`, has a runtime type `C`, then $\mathtt{C} \in \mathcal{C}_{\mathtt{x}_{\mathtt{PP}}}$. Class analysis is useful (i) for optimization of object-oriented programs, so to statically resolve virtual calls, and (ii) for the static construction of the control-flow graph of a program, so to provide the first step for a further analysis. In fact, if class analysis infers that a given program point, corresponding to a method invocation, $\mathcal{C}_{\mathtt{x}_{\mathtt{PP}}}$ is a singleton, then there is no need for a look-up procedure in the class hierarchy to determine the method to be invoked at runtime.

Several class analyses have been proposed in the literature, which consider different values for the ratio precision/cost. For instance, the seminal work by Palsberg and Schwartzbach, [37], presents a very precise, but also expensive, analysis for untyped object-oriented languages. These results have been improved by the same authors, [36], as well as by others, [4,18,15,45], which consider fast, but also imprecise, class analyses for the removal of virtual function calls in C++.

Another approach for reducing the cost of the analysis is by modularizing it: Besson and Jensen introduced a modular class analysis based on DATALOG, [5], and Probst described an analysis to incrementally construct the control-flow graph of Java programs, [39].

Spoto and Jensen, [44,24], provided a uniform, abstract interpretation-based, view of such analyses. The authors define a concrete trace semantics, and they proved how existing analyses are an abstraction of such a semantics. Given an execution trace $\sigma_0\sigma_1\sigma_2\ldots\sigma_n$ and a function *typeOf* which returns the runtime type of an object, the analysis of Palsberg and Schwartzbach is obtained by considering an abstraction function such that

$$\alpha_{PS}(\sigma_0\sigma_1\sigma_2\ldots\sigma_n) = \lambda\mathtt{PP}.\ \lambda\mathtt{x}.\ \{\mathit{typeOf}(\sigma_i(\mathtt{x})) \mid i \in [0\ldots n], \sigma_i(pp) = \mathtt{PP}\}.$$

The fast type analysis of Bacon and Sweeney is obtained by considering a further abstraction which collects together the types of a given variable through all the program points:

$$\alpha_{BS}(\sigma_0\sigma_1\sigma_2\ldots\sigma_n) = \lambda\mathtt{x}.\ \bigcup_{pp\in\mathtt{Program}} \alpha_{PS}(\sigma_0\sigma_1\sigma_2\ldots\sigma_n)(pp)(\mathtt{x}).$$

As a corollary of their formalization, Spoto and Jensen formally relate the relative precision of the analyses, by considering the relative precision of the corresponding abstract domains.

2.2 Pointer Analysis

A pointer analysis computes, for each program point $\mathtt{PP}$, and for each variable $\mathtt{x}$ a set $\mathcal{A}_{\mathtt{xPP}}$ of heap objects, such that if in an execution of the program, at program point $\mathtt{PP}$, $\mathtt{x}$ points to an heap object h, then $h \in \mathcal{A}_{\mathtt{xPP}}$.

A precise and scalable pointer analysis is a basic requirement for an effective static analysis of object-oriented programs. In fact, in real-world object-oriented languages, objects are heap-allocated and they are unequivocally identified by their heap address. As a consequence, a precise determination of the addresses a variable may point to allows one to have a precise information on the objects a program is made of. Furthermore, pointer analysis implies class analysis. In fact, given a result $\mathcal{A}_{\mathtt{xPP}}$ of a pointer analysis, the set of classes $\mathtt{x}$ can evaluate to at program point $\mathtt{PP}$ is given by an abstraction function α_L, which collects the types of the heap objects whom address is in $\mathcal{A}_{\mathtt{xPP}}$:

$$\alpha_L(\mathcal{A}_{\mathtt{xPP}}) = \{\mathit{typeOf}(\mathrm{heap}(h)) \mid h \in \mathcal{A}_{\mathtt{xPP}}\} = \mathcal{C}_{\mathtt{xPP}}.$$

The first attempts for an effective pointer analysis of object-oriented language focused on modifications/adaptations of existing pointer analyses for C. For instance, Rountev, Milanova, and Ryder proposed a pointer analysis for Java, [41,34], that adapts the Anderesen's pointer analysis for C [3]; and Ramalingam *et al*. presented an analysis for inferring the local heap structure of Java containers, [40], which uses TVLA [25]. More recently, Pollet, Le Charlier, and Cortesi introduced two abstract domains to express type, structural, and sharing information about dynamically created objects, [38]; and Chang and Leino presented an algebra of abstract domains that is essentially the reduced product of a precise alias analysis for heap-allocated objects and a generic abstract domain (parameter of the algebra), [9].

2.3 Escape Analysis

Escape analysis determines whether the lifetime of an object oversteps its static scope. If $\overline{\mathsf{PP}}$ is the exit point of a method, then an escape analysis computes the set $\mathcal{E}_{\overline{\mathsf{PP}}}$ of the heap-allocated objects at $\overline{\mathsf{PP}}$. Escape analysis is useful for program optimization, and in particular for (i) stack-allocating objects and (ii) removing synchronization. Escape analysis is strictly related to pointer analysis. In fact, if $\mathcal{A}$ is the information computed by a pointer analysis, for all the program points and variables, then

$$\alpha_B(\mathcal{A}) = \bigcup_{\mathtt{x} \in \mathit{Vars}} \mathcal{A}_{\overline{\mathsf{PP}}_{\mathtt{x}}} = \mathcal{E}_{\overline{\mathsf{PP}}}.$$

Gay and Steensgaard apply a very fast, but imprecise, escape analysis to the stack allocation in Java, [21]; Bogda and Höltz addressed the problem of synchronization elimination in concurrent Java programs, through the use of a more precise analysis [8]; Blanchet developed an escape analysis for the full Java whom soundness proof relies on a pointer analysis [7,6]. The analysis of Blanchet is precise and efficient enough to be applied to boost stack allocation and synchronize removal tasks. Several others escape analyses have been developed, with different values of the precision/cost ration. We recall the Whaley-Rinard and Viven-Rinard analyses, based on points-to escape graphs, [47,46]; the Choi *et. al* analysis, based on connection graphs, [10]; the Ruf analysis, which exploits static fields, [42].

2.4 Inference of Class Invariants

Class invariants represent the basis of good software engineering of object-oriented programs, [33]. A class invariant is a property of a class valid before and after the execution of any method of the class. It can be characterized as an abstraction of the trace semantics, where just the states corresponding to the entry points and exit points of method invocations of instances of a class are retained, [30,29].

Abstract Interpretation	**Class Hierarchies**
Abstract Domains	Classes
Approximation Order ($\sqsubseteq$)	Subclassing Relation ($\leq$)
Most Abstract Domain($\top_{\mathcal{D}}$)	`Object`
Reduced Product	Multiple Inheritance
Domain Refinement	Class Extension

Fig. 1. The parallel between abstract domains and class hierarchies.

Automatic-inferred class invariants are useful for modular software verification of classes, for optimization, for code documentation and for compiler designing. Ghemawat, Randall, and Scales [22] and subsequently Aggarwal and Randall [1] presented a static analysis for the removal of checks on array bounds, that essentially computes a class invariant in the form of $\texttt{a} == \texttt{null} \lor 0 \leq \texttt{b} \leq \texttt{a.length}$. Detlefs was interested in inferring correct explicit deallocation of elements of long-lived data structures [16]. Flanagan and Leino developed Houdini, a tool based on ESC/Java, for the inference of invariants, [20]. Ernst designed Daikon, a tool for the inference of pseudo-class invariants, [19]. Logozzo introduced a generic framework for the inference of class invariants, which takes into account inheritance, polymorphism, mutually recursive classes, [32,28,26,27].

2.5 *Other Analyses*

Among the other analyses that have been designed for object oriented languages, we recall the one of Christensen, Møller, and Schwartzbach, that approximates the result of string expressions, [11]; the one of Distefano, Katoen and Rensik, who present a temporal logic for object-oriented programs and the corresponding model-checking algorithm, [17]; the one of Owen and Watson, who present an analysis to remove unnecessary box/unboxing operations, [35]; the one of Zee and Rinard, which allows one to remove write barriers, [48]; the one of Alur, Cerny, Madhusudan, and Nam, which synthesises interface specifications for Java classes, [2]; and the one of Salcianu and Rinard, which checks if a Java method is pure or not, [43].

3 ... and what could be done

In the previous survey we have seen how abstract interpretation is an effective technology for the analysis, the verification and the optimization of object-oriented languages. We think that it can be used *also* for the formalization and the description of object-oriented systems.

In fact, there are similarities between abstract interpretation theory and class hierarchies. A basic result in abstract interpretation theory is that, if the concrete domain is a complete lattice, then the set of all its abstractions is a complete lattice too, [13].

Let $\mathcal{D}$ be a lattice of abstractions, and let $\mathcal{H}$ a class hierarchy. The order on $\mathcal{D}$ is the relative precision of the abstract domains, i.e., $\mathtt{D}_1 \sqsubseteq \mathtt{D}_2$ iff $\mathtt{D}_1$ is an abstraction of $\mathtt{D}_2$. Intuitively, this means that $\mathtt{D}_1$ captures at least all the information of $\mathtt{D}_2$, i.e., $\mathtt{D}_2$ is a refinement of $\mathtt{D}_1$, [23]. On the other hand, the order on $\mathcal{H}$ is the subclass relation, i.e., $\mathtt{C}_1 \leq \mathtt{C}_2$ iff $\mathtt{C}_1$ is a subclass of $\mathtt{C}_2$. Intuitively, it means that the class $\mathtt{C}_1$ is a specialization, or a refinement, of the class $\mathtt{C}_2$. Stated differently, $\mathtt{C}_2$ is a class more abstract than $\mathtt{C}_1$. As a consequence, (i) $\top_{\mathcal{D}}$, the greatest element of $\mathcal{D}$, is the the most abstract domain; and (ii) `Object`, the common superclass to all the classes in $\mathcal{H}$, is the most abstract class of the hierarchy.

Exploiting such a parallel between the two concepts, lattices of abstract domains and class hierarchies, we can say that the $\mathcal{H}$-counterpart for the meet operation on $\mathcal{D}$ is multiple inheritance. In fact, the meet operation of two abstract domains $\mathtt{D}_1$ and $\mathtt{D}_2$ is the reduced product, i.e., the most abstract domain $\mathtt{D}_3$, which contains all the information of $\mathtt{D}_1$ and $\mathtt{D}_2$, [13]. On the other hand, if $\mathtt{C}_3$ is a subclass of both $\mathtt{C}_1$ and $\mathtt{C}_2$, then it contains all the fields of, and it may behave as, its superclasses. Furthermore, abstract domain refinement and the extension of classes are related concepts, too. In fact, the refinement of a given abstract domain is a domain that captures all the properties captured by the refined domain *plus* some others (specific to the refinement). On the other hand, the extension of a given base class is a class that inherits all the behaviors of the ancestor, *plus* some others (consider, e.g., the classical `2DPoints` and `3DPoints` classes, [12]).

We think that the analogies between the two concepts, that are summarized in Figure 1, deserve to be further studied, thus yielding to a cross-porting of results. For instance, which are the counterpart (i) in class hierarchies for the reduce cardinal power; and (ii) in abstract domains for interfaces and for polymorphism? We have begun a study in which we apply abstraction interpretation techniques to the definition and the manipulation of class hierarchies, [31], and the first results are very encouraging.

Future research in Abstract Interpretation for OO languages might consider the issue of validating the whole Object-oriented software engineering process: analysis (OOA), design (OOD) and implementation (OOI). Namely, it might be interesting to investigate how abstract interpretation can provide an alternative formal approach to requirement specifications (which are just abstractions of the behavior of the desired system), as well as a guideline for the software design (by exploiting class invariants), its implementation (by exploiting object invariants), and system integration (by a suitable abstract representation of non functional requirements).

4 Conclusions

In this position we drew a quick overview of the main contributes in abstract interpretation of object-oriented languages, and we sketched some analogies between two core concepts of the two fields, respectively abstract domains and class hierarchies.

References

[1] A. Aggarwal and K. H. Randall. Related field analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)*, volume 36(5) of *SIGPLAN Notices*, pages 214–220. ACM Press, June 2001.

[2] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 31th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'05)*. ACM Press, 2005.

[3] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Unversity of Copenhagen, 1994.

[4] D.F. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA'96)*. ACM Press, 1996.

[5] F. Besson and T. Jensen. Modular class analysis with datalog. In *10th International Symposium on Static Analysis (SAS'03)*, volume 2694 of *LNCS*. Springer Verlag, 2003.

[6] B. Blanchet. Escape analysis for object oriented languages. Application to Java. In *14th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 20–34, Denver, Colorado, November 1999.

[7] B. Blanchet. Escape analysis for Java^{TM}: Theory and practice. *ACM Transactions on Programming Languages*, 25(6):713–775, nov 2003.

[8] J. Bogda and U. Höltze. Removing unnecessary synchronization in Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99)*. ACM Press, 1999.

[9] B.-Y. E. Chang and K.R.M. Leino. Abstract interpretation with alien expressions and heap structures. In *Proceedings of the 6th International conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*. Springer-Verlag, 2005.

[10] J. D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and Midkiff S. P. Escape analysis for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99)*. ACM Press, 1999.

[11] A. S. Christensen, A. Moller, and M. I. Schwartzbac. Precise analysis of string expressions. In *Proceedings of the International Symposium on Static Analysis (SAS'03)*, number 2694 in LNCS. Springer-Verlag, 2003.

[12] W. R. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, November 1994.

[13] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '79)*, pages 269–282. ACM Press, 1979.

[14] P. Cousot and R. Cousot. Static analysis of embedded software: Problems and perspectives, invited paper. In T.A. Henzinger and C.M. Kirsch, editors, *Proc.* First Int. Workshop on Embedded Software, EMSOFT 2001, volume 2211 of *Lecture Notes in Computer Science*, pages 97–113. Springer, 2001.

[15] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysi. In *Proceedings of the 9th European Conference on Object-oriented Programming (ECOOP'95)*, number 952 in LNCS. Springer-Verlag, 1995.

[16] D. Detlefs. Automatic inference of reference-count invariants. In *Proceedings of the 2nd ACM Workshop on Semantics, Programa Analysis, and Computing Environments for Memory Management*. ACM Press, 2004.

[17] D. Distefano, J.P. Katoen, and A. Rensik. On a temporal logic for object-based systems. In *4th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000)*, volume 177 of *IFIP Conference Proceedings*, pages 285–304, Stanford, CA, U.S.A., September 2000. Kluwer Academic Publishers.

[18] K. Driesen and U. Hölzle. The direct cost of virtual function calls in C++. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'96)*. ACM Press, 1996.

[19] M. Ernst. *Dynamically Discovering Likely Program Invariants.* PhD thesis, University of Washington Department of Computer Science and Engineering, 2002.

[20] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe (FME 2001)*, volume 2021 of *Lectures Notes in Computer Science*, pages 500–517. Springer-Verlag, March 2001.

[21] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Proceedings of the 9th International Conference on Compiler Construction (CC'00)*, volume 1781 of *LNCS*. Springer-Verlag, 2000.

[22] S. Ghemawat, K. H. Randall, and D. J. Scales. Field analysis: getting useful and low-cost interprocedural information. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, volume 35(5) of *ACM SIGPLAN Notices*, pages 334–344. ACM Press, June 2000.

[23] R. Giacobazzi and F. Ranzato. Refining and compressing abstract domains. In *Proceedings of the international conference on Automata, Languages and Programming (ICALP'97)*, volume 1256 of *LNCS*. Springer-Verlag, 1997.

[24] T. Jensen and F. Spoto. Class analysis of object-oriented programs through abstract interpretation. In *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *Lecture Notes in Computer Science*, pages 261–275. Springer-Verlag, April 2001.

[25] T. Lev-Ami and M. Sagiv. Tvla: A system for implementing static analyses. In *7th International Symposium on Static Analysis (SAS'00)*, volume 1824 of *LNCS*. Springer-Verlag, 2000.

[26] F. Logozzo. Class-level modular analysis for object oriented languages. In *Proceedings of the 10th Static Analysis Symposium 2003 (SAS '03)*, volume 2694 of *Lectures Notes in Computer Science*, pages 37–54. Springer-Verlag, June 2003.

[27] F. Logozzo. An approach to behavioral subtyping based on static analysis. In *Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004)*, Electronic Notes in Theoretical Computer Science. Elsevier Science, April 2004.

[28] F. Logozzo. Automatic inference of class invariants. In *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, volume 2937 of *Lectures Notes in Computer Science*, pages 211–222. Springer-Verlag, January 2004.

[29] F. Logozzo. *Modular Static Analysis of Object-oriented Languages.* PhD thesis, Ecole Polytechnique, June 2004.

[30] F. Logozzo. Class invariants as abstract interpretation of trace semantics. *Computer Languages, Systems and Structures*, 2005.

[31] F. Logozzo and A. Cortesi. Semantic class hierarchies by abstract interpretation. In *submitted for publication*, 2005.

[32] Francesco Logozzo. Separate compositional analysis of class-based object-oriented languages. In *Proceedings of the 10th International Conference on Algebraic Methodology And Software Technology (AMAST'2004)*, volume 3116 of *Lectures Notes in Computer Science*, pages 332–346. Springer-Verlag, July 2004.

[33] B. Meyer. *Object-Oriented Software Construction (2nd Edition).* Professional Technical Reference. Prentice Hall, 1997.

[34] A. Milanova, A. Rountev, and Ryder B. G. Parameterized object sensitivity for points-to and side-effect analyses for java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002).* ACM Press, 2002.

[35] T. Owen and D. Watson. Reducing the cost of object boxing. In *Proceedings of the 13th International Conference on Compiler Construction (CC'04)*, number 2985 in LNCS. Springer-Verlag, 2004.

[36] N. Oxhj, J. Palsberg, and M.I. Schwartzbach. Making type-inference practical. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP'92)*, number 651 in LNCS. Springer-Verlag, 1992.

[37] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *Proceeding of the ACM SIGPLAN Conference on Object-Oriented Programming (OOPSLA'91).* ACM Press, 1991.

[38] I. Pollet, B. Le Charlier, and A. Cortesi. Distinctness and sharing domains for static analysis of Java programs. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP '01)*, volume 2072 of *Lectures Notes in Computer Science*, pages 77–98. Springer-Verlag, 2001.

[39] C. Probst. Modular control flow analysis for libraries. In *Proceedings of the Static Analysys Symposium (SAS '02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 165–179. Springer-Verlag, 2002.

[40] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*, volume 37, 5 of *ACM SIGPLAN Notices*, pages 83–94, New York, June 17–19 2002. ACM Press.

[41] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *16th ACM Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01)*, pages 43–55. ACM Press, November 2001.

[42] E. Ruf. Effective synchronization removal for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00).* ACM Press, 2000.

[43] A. Salcianu and M. Rinard. Purity analysis for java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Intepretation (VMCAI'05)*, volume 3385 of *LNCS*. Springer-Verlag, 2005.

[44] F. Spoto and T. Jensen. Class Analyses as Abstract Interpretations of Trace Semantics. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(5):578–630, September 2003.

[45] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000)*. ACM Press, 2000.

[46] F. Vivien and M. C. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. ACM Press, 2001.

[47] J. Whaley and M. C. Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99)*. ACM Press, 1999.

[48] K. Zee and M. Rinard. Write barrier removal by static analysis. In *17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '02)*, volume 37(11) of *SIGPLAN Notices*, pages 191–210. ACM Press, November 2002.