

Pointer Analysis

Edited by

Ondřej Lhoták¹, Yannis Smaragdakis², and Manu Sridharan³

¹ University of Waterloo, CA, olhotak@uwaterloo.ca

² University of Athens, GR, yannis@smaragd.org

³ IBM TJ Watson Research Center – Yorktown Heights, US,
msridhar@us.ibm.com

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 13162 “Pointer Analysis”. The seminar had 27 attendees, including both pointer analysis experts and researchers developing clients in need of better pointer analysis. The seminar came at a key point in time, with pointer analysis techniques acquiring sophistication but still being just beyond the edge of wide practical deployment. The seminar participants presented recent research results, and identified key open problems and future directions for the field. This report presents abstracts of the participants’ talks and summaries of the breakout sessions from the seminar.

Seminar 14.–19. April, 2013 – www.dagstuhl.de/13162

1998 ACM Subject Classification F.3.2 Logics and Meanings of Programs: Semantics of Programming Languages: Program Analysis, D.3.4. Programming Languages: Processors: Compilers

Keywords and phrases pointer analysis, points-to analysis, alias analysis, static analysis, programming languages

Digital Object Identifier 10.4230/DagRep.3.4.91

Edited in cooperation with Gogul Balakrishnan

1 Executive Summary

Ondřej Lhoták

Yannis Smaragdakis

Manu Sridharan

License © Creative Commons BY 3.0 Unported license
© Ondřej Lhoták, Yannis Smaragdakis, Manu Sridharan

The Dagstuhl seminar on *Pointer Analysis* brought together experts in pointer analysis and researchers building demanding clients of pointer analysis, with the goal of disseminating recent results and identifying important future directions. The seminar was a great success, with high-quality talks, plenty of interesting discussions, and illuminating breakout sessions.

Research Context

Pointer analysis is one of the most fundamental static program analyses, on which virtually all others are built. It consists of computing an abstraction of which heap objects a program variable or expression can refer to. Due to its importance, a large body of work exists on pointer analysis, and many researchers continue to study and develop new variants. Pointer analyses can vary along many axes, such as desired precision, handling of particular language



Except where otherwise noted, content of this report is licensed under a Creative Commons BY 3.0 Unported license

Pointer Analysis, *Dagstuhl Reports*, Vol. 3, Issue 4, pp. 91–113

Editors: Ondřej Lhoták, Yannis Smaragdakis, and Manu Sridharan



DAGSTUHL
REPORTS

Dagstuhl Reports
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

features, and implementation data structures and optimizations. Given the subtle implications of these design choices, and the importance of low-level details often excluded from conference-length papers, it can be difficult even for pointer analysis experts to understand the relationship between different analysis variants. For a non-expert aiming to use pointer analysis in a higher-level client (for verification, optimization, refactoring, etc.), choosing the right analysis variant can be truly daunting.

Pointer analysis is a mature area with a wealth of research results, at a temptingly close distance from wide practical applicability, but not there yet. The breakout application of precise analysis algorithms has seemed to be around the corner for the past decade. Although research ideas are implemented and even deployed in limited settings, several caveats always remain. These include assumptions about client analyses (i.e., the pointer analysis algorithm is valid only under assumptions of how the information will be used), assumptions about the analyzed program (e.g., that some language features are absent or that their presence does not affect the analysis outcome), assumptions about modularity (e.g., that the code to be analyzed constitutes the whole program), etc. The right engineering packaging of pointer analysis algorithms as well as a convenient characterization of their domain of applicability are still elusive.

In this light, the seminar aimed to emphasize the relationship of pointer analysis algorithms with client analyses, as well as practical deployment issues. The seminar brought together researchers working on pointer analysis for various programming languages with researchers working on key analysis clients. Our main goals were (1) to deepen understanding of the relationships between existing pointer analysis techniques, and (2) to gain a better understanding of what pointer analysis improvements are required by clients, thereby setting an exciting agenda for the area going forward.

Seminar Format

Our seminar employed a somewhat unusual format for participant talks, intended to encourage a deeper discussion of each participant’s work. Each participant was allotted a 40-minute slot to present their work, consisting of 20 minutes of presentation and 20 minutes of discussion. The presentation and discussion times in each slot were enforced using a chess clock: when a question arose during a talk, the clock was “flipped” to discussion time, and after the discussion, it was flipped back to speaker time. (The times were not very strictly enforced; in some cases, the audience would “donate” time to the speaker to complete his/her presentation.) This format had two key benefits:

- It enabled discussion to freely occur during the talk, removing the worry that the speaker would have no time left to complete his/her presentation.
- It encouraged the audience to ask more questions, in order to “use up” the allotted audience time.

Overall, the format was very successful in encouraging good discussion, and most participants enjoyed it.

In addition to talks, we held four 90-minute breakout sessions. The session topics were proposed by participants before and during the seminar and voted on by participants. The sessions were scheduled two at a time, and participants could choose which session to attend. The discussions held in these sessions were quite illuminating, and are summarized in Section 4 of this report. Finally, the last half-day of the seminar was spent on additional discussion of the breakout session topics, and on an initial effort to collectively improve the Wikipedia article on pointer analysis.¹

¹ See http://en.wikipedia.org/wiki/Pointer_analysis.

Seminar Results

Recent advancements in pointer analysis have come from several different directions:

- Formulations (CFL, Datalog)—highly-complex analyses have been specified in terms of concise specifications, by utilizing declarative notations.
- Greater precision—interesting analyses that maintain finer-grained abstractions while maintaining scalability have been invented.
- Optimizations—data structures such as BDDs have been used to make complex analyses feasible.
- Demand-driven, refinement—the analysis problem has been specialized effectively when pointer information only needs to be computed for select program sites.
- Partial programs—analyses have been formulated to work without fully analyzing all libraries, or even all application code.

Such advances were discussed in detail during many participant talks in the seminar, and in the breakout sessions.

Recent work in pointer analysis has been driven by new clients for the analysis and by new programming languages. Along with ongoing use of pointer analysis in traditional optimizing compilers, recent years have seen many other clients emerge that require effective pointer analysis, e.g., in the areas of program verification and bug finding, refactoring, and security. These clients were well-represented by seminar attendees, who gave many interesting talks on novel uses of pointer analysis (particularly in the security domain). The rich exchanges between researchers building novel clients and those with pointer analysis expertise were one of the most valuable aspects of the seminar. Additionally, one breakout session covered the difficulties in designing an effective general pointer-analysis API that is suitable for a wide variety of clients.

Mainstream programming has been transitioning to increasingly heap-intensive languages: from C-like languages to object-oriented languages like Java and C#, and more recently to scripting languages like JavaScript and Ruby. As languages become more heap-intensive, the need for effective pointer analysis is greater, motivating continuing work in this area. The seminar talks covered a wide and diverse set of languages, each with its own considerations. A few talks covered pointer analysis for higher-level languages such as JavaScript and MATLAB. Such languages are becoming increasingly popular, and they are very heap-intensive compared to C-like languages, motivating the need for better pointer analysis. A couple of talks presented techniques for control-flow analysis of functional languages like Scheme. While the pointer analysis and control-flow analysis communities often use similar techniques, the relationships between the techniques is often obscured by differing terminology and presentation styles. The presentations on control-flow analysis and the corresponding discussions were helpful in bridging this gap.

The seminar included a good deal of discussion on practical issues with pointer analysis, including evaluation methodologies and issues arising in real-world deployments. A key theme that arose from these discussions was the need for pointer analysis to be at least partially unsound to be useful in practice, and how this need for unsoundness has not been explained properly in the literature. Analyses that made soundness compromises for practicality were deemed “soundy,” a tongue-in-cheek term that caught on quickly among participants. Recently, some seminar participants presented a well-received PLDI Fun and Interesting Topics (FIT) talk on the notion of “soundiness,” and several participants have agreed to collectively co-author a publishable document on the topic.

Conclusions

Overall, the *Pointer Analysis* Dagstuhl seminar was a great success. The seminar brought together 27 researchers from both academia and industry (including Google, IBM, Microsoft, NEC), with a good mix of junior and senior researchers. There were many interesting talks, with deep discussion facilitated by the chess clock time maintenance. The seminar facilitated interaction between pointer analysis experts and researchers building novel clients (a key goal for the seminar from the beginning), and also between researchers working on analyses for a variety of languages. Breakout sessions enabled further discussion of certain particularly interesting topics. In particular, there were invaluable discussions of many practical issues that often get short shrift in conference papers. These discussions sparked the notion of “soundness,” which may have broader impact via a future publication.

2 Table of Contents

Executive Summary

<i>Ondřej Lhoták, Yannis Smaragdakis, Manu Sridharan</i>	91
--	----

Overview of Talks

Does it have to be so hard? <i>José Nelson Amaral</i>	97
Scalable and Precise Program Analysis at NEC <i>Gogul Balakrishnan</i>	97
Challenges in vulnerability detection for the Java runtime library <i>Eric Bodden</i>	98
Precise Heap Reachability by Refutation Analysis <i>Bor-Yuh Evan Chang</i>	98
Precise and Fully-Automatic Verification of Container-Manipulating Programs <i>Isil Dillig and Thomas Dillig</i>	98
The End of Pointer Analysis? <i>Julian Dolby</i>	99
The Business of Pointer Analysis <i>Samuel Z. Guyer</i>	100
Pointer analysis for dynamic information flow control <i>Christian Hammer</i>	100
Pointer Analysis Meets MATLAB <i>Laurie J. Hendren</i>	100
The Approximations vs. Abstractions Dilemma in Pointer Analysis <i>Uday Khedker</i>	101
Incomplete Program Analysis <i>Ondřej Lhoták</i>	101
Challenges in Pointer Analysis of JavaScript <i>Benjamin Livshits</i>	101
Comparing Different Points-To Analyses <i>Welf Löwe</i>	102
Towards a Quantitative Understanding of Heap Structure and Application to Analysis Design <i>Mark Marron</i>	102
Control-flow analysis of higher-order programs <i>Matt Might</i>	103
Inference and Checking of Context-sensitive Pluggable Types <i>Ana Milanova</i>	103
Pointer Analysis for Refactoring JavaScript Programs <i>Anders Møller</i>	103
New Search Techniques for Query-Driven Dataflow Analysis <i>Mayur Naik</i>	104

Sparse Analysis Framework <i>Hakjoo Oh</i>	104
Empirical Evaluation of Points-To Analyses <i>Erhard Plödereder</i>	104
Set-Based Pre-Processing for Points-To Analysis <i>Yannis Smaragdakis</i>	106
Pointer Analysis for Probabilistic Noninterference <i>Gregor Snelting</i>	106
Pointer Analysis and Reflection <i>Manu Sridharan</i>	107
Modular combination of shape abstraction with numeric abstraction <i>Xavier Rival</i>	107
Scaling flow analysis using big-step semantics <i>Dimitris Vardoulakis</i>	108
Breakout Sessions	
Better APIs for Clients <i>José Nelson Amaral</i>	108
Pointer analyses for open programs (libraries/frameworks) <i>Eric Bodden</i>	110
Shape Analysis and Pointer Analysis: Working Together <i>Bor-Yuh Evan Chang</i>	110
Practical Aspects of Pointer Analysis <i>Manu Sridharan</i>	112
Participants	113

3 Overview of Talks

3.1 Does it have to be so hard?

José Nelson Amaral (University of Alberta, CA)

License © Creative Commons BY 3.0 Unported license
© José Nelson Amaral

The brightest and most capable students are the ones that undertake research in the area of pointer and reference analysis. Yet, often they take a long time to graduate, and sometimes they produce fewer research results than students that undertake research in different areas. Moreover, often the outcome of their research is incomplete and unsatisfying. On the other hand, many of the papers published in the area — even the best ones that appear in the top venues — leave readers and reviewers with a sense of a good work that is incomplete. In this discussion we look at several shortcomings in currently published papers in pointer and reference analysis. Then we will talk about some undertakings by the community that could change this situation, making research in analysis more rewarding and productive for students and practitioners, and accelerating the speed of innovation in this area.

3.2 Scalable and Precise Program Analysis at NEC

Gogul Balakrishnan (NEC Laboratories America, Inc. – Princeton, US)

License © Creative Commons BY 3.0 Unported license
© Gogul Balakrishnan

Joint work of Balakrishnan, Gogul; Ganai, Malay; Gupta, Aarti; Ivancic, Franjo; Kahlon Vineet, Li, Weihong; Maeda, Naoto; Papakonstantinou, Nadia; Sankaranarayanan, Sriram; Sinha, Nishant; Wang, Chao
Main reference G. Balakrishnan, M. K. Ganai, A. Gupta, F. Ivancic, V. Kahlon, W. Li, N. Maeda, N. Papakonstantinou, S. Sankaranarayanan, N. Sinha, C. Wang, “Scalable and precise program analysis at NEC,” in Proc. of 10th Int’l Conf. on Formal Methods in Computer-Aided Design (FMCAD’10), pp. 273–274, IEEE, 2010.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5770960


In this talk, I will briefly present the program analysis tools that we have developed at NEC Labs, and describe how pointer analysis is used in these tools. Specifically, I will talk about Varvel and ARC++. Varvel is tool for finding bugs in C and C++ programs, and is based on static analysis and model checking. ARC++ is a tool to find bugs in C++ programs based on user-specified error patterns.

References

- 1 G. Balakrishnan, M. K. Ganai, A. Gupta, F. Ivancic, V. Kahlon, W. Li, N. Maeda, N. Papakonstantinou, S. Sankaranarayanan, N. Sinha, and C. Wang. Scalable and precise program analysis at nec. In *FMCAD*, pages 273–274, 2010.
- 2 F. Ivancic, G. Balakrishnan, A. Gupta, S. Sankaranarayanan, N. Maeda, H. Tokuoaka, T. Imoto, and Y. Miyazaki. Dc2: A framework for scalable, scope-bounded software verification. In *ASE*, pages 133–142, 2011.
- 3 P. Prabhu, N. Maeda, G. Balakrishnan, F. Ivancic, and A. Gupta. Interprocedural exception analysis for C++. In *ECOOP*, pages 583–608, 2011.
- 4 J. Yang, G. Balakrishnan, N. Maeda, F. Ivancic, A. Gupta, N. Sinha, S. Sankaranarayanan, and N. Sharma. Object model construction for inheritance in C++ and its applications to program analysis. In *CC*, pages 144–164, 2012.

3.3 Challenges in vulnerability detection for the Java runtime library

Eric Bodden (TU Darmstadt, DE)


License  Creative Commons BY 3.0 Unported license
© Eric Bodden

Joint work of Bodden, Eric; Hermann, Ben; Lerch, Johannes

In this talk I will discuss a recent client analysis that we use to detect vulnerabilities in the Java runtime library. I will discuss challenges this analysis poses in terms of managing calling contexts and different analysis directions. In particular, it currently appears challenging to synchronize with each other a forward and backward analysis in such a way that they both only consider common calling contexts.

3.4 Precise Heap Reachability by Refutation Analysis

Bor-Yuh Evan Chang (University of Colorado – Boulder, US)

License  Creative Commons BY 3.0 Unported license
© Bor-Yuh Evan Chang

Joint work of Blackshear, Sam; Chang, Bor-Yuh Evan; Sridharan, Manu

Main reference S. Blackshear, B.-Y.E. Chang, M. Sridharan, “Thresher: Precise Refutations for Heap Reachability,” in Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI’13), pp. 275–286, ACM, 2013.

URL <http://doi.acm.org/10.1145/2462156.2462186>

Precise heap reachability information that can be provided by a points-to analysis is needed for many static analysis clients. However, the typical scenario is that the points-to analysis is never quite precise enough leading to too many false alarms in the client. Our thesis is not that we need more precise up-front points-to analyses, but rather we can design after-the-fact triage analyses that are effective at refuting facts to yield targeted precision improvements. The challenge that we explore is to maximally utilize the combination of the up-front and the after-the-fact analyses.

We have investigated refutation analysis in the context of detecting statically a class of Android memory leaks. For this client, we have found the necessity for an analysis capable of path-sensitive reasoning interprocedurally and with strong updates—a level of precision difficult to achieve globally in an up-front manner. In contrast, our approach applies a refutation analysis that mixes a backwards symbolic execution with results from the up-front points-to analysis to prune infeasible paths quickly.

3.5 Precise and Fully-Automatic Verification of Container-Manipulating Programs

Isil Dillig and Thomas Dillig (College of William and Mary)

License  Creative Commons BY 3.0 Unported license
© Isil Dillig and Thomas Dillig

One of the key challenges in automated software verification is obtaining a conservative yet sufficiently precise understanding of the contents of data structures in the heap. A particularly important and widely-used class of heap data structures is containers, which

support operations such as inserting, retrieving, removing, and iterating over elements. Examples of containers include arrays, lists, vectors, sets, maps, stacks, queues, etc.

In this talk, we will describe a sound, precise, scalable, and fully-automatic static analysis technique for reasoning about the contents of container data structures. This technique is capable of tracking position-value and key-value correlations, supports reasoning about arbitrary nestings of these data structures, and integrates container reasoning directly into a heap analysis, allowing, for the first time, the verification of complex programs that manipulate heap objects through container data structures. More specifically, we will describe a symbolic heap abstraction that augments a graph representation of the heap with logical formulas and that reduces some of the difficulty of heap reasoning to standard logic operations, such as existential quantifier elimination and satisfiability. I will present experimental results demonstrating that our technique is very useful for verifying memory safety in complex heap- and container-manipulating C and C++ programs that use arrays and other container data structures from the STL and QT libraries.

3.6 The End of Pointer Analysis?

Julian Dolby (IBM TJ Watson Research Center, Hawthorne, USA)

License © Creative Commons BY 3.0 Unported license
© Julian Dolby

Pointer analysis means computing an approximation of the possible objects to which any program variable may refer; it has traditionally been done by conservatively approximating all possible data flow in the program, resulting in a conservative approximation of the objects held by any variable. This has always been a bit fake—no tools soundly approximate all possible reflective and JNI behavior in Java, for instance—but even the comforting illusion of soundness has become unsustainable in the world of framework- and browser-based Web applications. The frameworks are built on ubiquitous complex reflective behavior, and the browser appears as a large, complex, poorly-specified native API; the frameworks and the applications themselves are written in JavaScript, the lingua franca of the Web, the dynamic nature of which gives pointer analysis no help. Whether this world can be analyzed soundly is perhaps technically still an open problem, but the prognosis seems grim at best.

We have been exploring deliberately unsound analyses which make no attempt to approximate all possible data flow in a program; certain constructs are ignored not because they are unimportant, but simply because they are too hard. The tradeoff is now between how much can we ignore and still provide useful information versus how little can we ignore and still be tractable in practice. The good news so far is that there appear to be good tradeoffs, at least for a range of applications supporting IDE services. I will discuss recent and ongoing work in providing key information for IDE services: callgraphs and smart completions.

3.7 The Business of Pointer Analysis

Samuel Z. Guyer (Tufts University)

License © Creative Commons BY 3.0 Unported license
© Samuel Z. Guyer

Over the past few years I have had the opportunity to work with a company that relies on pointer analysis as part of its core business – finding security vulnerabilities in software. I have worked closely with them to select and implement algorithms from the literature, and I have been able to see how these algorithms work (or don't work) at an industrial scale. Some of the most interesting issues I have encountered, however, are not related to the question of "Does it work on real software?" In this talk I will describe some of the challenges of deploying sophisticated analyses for commercial purposes. They are important and research-worthy problems that have not, to my knowledge, received much attention in the academic community.

3.8 Pointer analysis for dynamic information flow control

Christian Hammer (Universität des Saarlandes)

License © Creative Commons BY 3.0 Unported license
© Christian Hammer

Dynamic information flow control is a powerful technique to ensure that confidential data cannot leak illicitly, and that untrusted data must not be used for trusted computations. However, since the standard security policy noninterference is a 2-trace-property, it cannot be enforced soundly and precisely by looking at one execution trace alone. One must either use conservative approximations or resort to static analysis about variables that might be modified in an alternative branch. This talk will present these challenges and how pointer analysis for a dynamic language like JavaScript, while challenging, is imperative to improve the precision of dynamic information flow.

3.9 Pointer Analysis Meets MATLAB

Laurie J. Hendren (McGill University, CA)

License © Creative Commons BY 3.0 Unported license
© Laurie J. Hendren

Joint work of Hendren, Laurie J.; Lameed, Nurudeen, Doherty, Jesse; Dubrau, Anton; Radpour, Soroush
URL <http://www.sable.mcgill.ca/mclab>

MATLAB is a dynamic array-based language commonly used by students, scientists and engineers. Although MATLAB has call-by-value semantics and no explicit pointers, there are many flow analysis problems that are similar to pointer analysis. In this talk I discussed why it is important for our research community to work on programming languages like MATLAB. I then outlined the key flow analysis problems that need to be solved and gave a summary of what my research group has accomplished and what we plan to work on in the future (<http://www.sable.mcgill.ca/mclab>).

3.10 The Approximations vs. Abstractions Dilemma in Pointer Analysis

Uday Khedker (Indian Institute of Technology, Mumbai, India)


License  Creative Commons BY 3.0 Unported license
© Uday Khedker

Given the vital importance of pointer analysis and the inherent difficulty of performing precise pointer analysis for practical programs, a large fraction of pointer analysis community has come to believe that compromising on precision is necessary for scalability and efficiency. This is evidenced by the fact that a large number of reported investigations in pointer analysis involve a significant amount of engineering approximations.

We find it hard to accept this assumption as the final inevitability. We believe that a lot could be gained by exploring a science of pointer analysis that tries to build clean abstractions. In our opinion, this is a road less travelled in pointer analysis. Without undermining the engineering efforts, we propose that a search for approximations should begin only after building clean abstractions and not before it. The talk describes our efforts in this direction.

3.11 Incomplete Program Analysis

Ondřej Lhoták (University of Waterloo, CA)

License  Creative Commons BY 3.0 Unported license
© Ondřej Lhoták
Joint work of Lhotak, Ondrej; Ali, Karim; Naeem, Nomair

Points-to analyses requiring the whole program are inefficient, hard to get right, non-modular, and fragile. This severely hinders development of client analyses and practical adoption. I discuss two possible solutions: the access path abstraction, which does not require the analysis to know about all allocation sites in the program, and the separate compilation assumption, which enables sound yet precise analysis of an application without analysis of its separately compiled libraries.

3.12 Challenges in Pointer Analysis of JavaScript

Benjamin Livshits (Microsoft Research – Redmond)


License  Creative Commons BY 3.0 Unported license
© Benjamin Livshits

This talk two specific challenges that arise in the process of doing – or attempting to do – static analysis for JavaScript programs. The first is that JavaScript programs on the web are not static – far from it – they’re in fact streaming. As such, the notion of whole program analysis needs to be reevaluated and perhaps discarded in favor of incrementality. Incremental pointer analysis for JavaScript is addressed in the Gulfstream project.

The second challenge is that of analyzing programs surrounded by complex *environments*, such as the DOM API, or node.js. Understanding the surrounding frameworks is both challenging and necessary. This is the topic of this talk and an upcoming FSE 2013 paper.

3.13 Comparing Different Points-To Analyses

Welf Löwe (*Linnaeus University – Växjö*)


License  Creative Commons BY 3.0 Unported license
© Welf Löwe

Comparing the accuracy of different points-to analysis approaches is important for us as the research community: it allows us to focus on successful approaches and to drop the less successful ones. However, comparing accuracy only works if the analyses are either strictly conservative or strictly optimistic. Unfortunately, only few such analyses exist in practice; most of them are conservative only on a subset of the languages they are designed for and, hence, neither conservative nor optimistic in general. Practical issues add to the problem of comparability: analyses are defined for different languages and versions and run-time systems thereof, and there are no commonly accepted standard benchmarking suites nor accuracy metrics defined. This makes it often impossible to take two research publications and reliably tell which one describes the more accurate points-to analysis.

In this talk, we discuss theoretical and practical issues with comparing points-to analyses and we suggest a methodology on how to benchmark them. We then and argue for a Gold Standard, i.e., a set of benchmark programs with known exact analysis results. Such a Gold Standard would allow assessing the exact accuracy of points-to analysis. Since such a Gold Standard cannot be computed automatically, it needs to be created semi-automatically by the research community. We suggest a methodology on how this could be achieved.

3.14 Towards a Quantitative Understanding of Heap Structure and Application to Analysis Design

Mark Marron (*Microsoft Research, Redmond, USA*)

License  Creative Commons BY 3.0 Unported license
© Mark Marron

This talk looks at two related questions (1) what kinds of heap structures and sharing relations appear in object-oriented programs and (2) how can this information be used to guide the design of a heap analysis. I will show results which indicate that in practice the heap is a relatively simple structure where the vast majority of sharing (aliasing) and shapes that are present can be described by a small number of simple concepts that are closely related to standard programming idioms. I will also outline a hybrid shape/points-to analysis, which is both precise and computationally lightweight, that was designed based on these quantitative results. These initial results demonstrate the potential for leveraging empirical data during the design, or evaluation, of a heap analysis and demonstrate the potential for further work on the quantitative characterization of the heaps that appear in real-world programs.

3.15 Control-flow analysis of higher-order programs

Matt Might (University of Utah, US)

License © Creative Commons BY 3.0 Unported license
© Matt Might
URL <http://matt.might.net/>

Control-flow analysis of higher-order programs and pointer analysis share much in common with each other. This talk serves as a tutorial on the basic method in higher-order control-flow analysis—the modern formulation of Shivers k-CFA. It discusses one common enhancement—abstract garbage collection. The talk concludes with cultural differences between the control-flow analysis and pointer analysis communities, noting the years- or decades-long lag between problem discovery, decidability, tractability, feasibility and evaluation in the CFA community.

3.16 Inference and Checking of Context-sensitive Pluggable Types

Ana Milanova (Rensselaer Polytechnic, US)

License © Creative Commons BY 3.0 Unported license
© Ana Milanova
Joint work of Milanova, Ana; Huang, Wei

We develop a framework for inference and checking of pluggable types, also known as type qualifiers. The framework allows us to formulate context-sensitive pluggable type systems (e.g., Ownership, Immutability, Taint, others) and infer and check types on large Java codes. The key novelty is 1) support for context sensitivity, and 2) a scalable inference engine, which allows type inference with zero or small number of user annotations. We formulate two analyses, traditionally powered by pointer analysis: 1) purity inference and 2) taint analysis, as type inference problems in our framework, and discuss our results.

3.17 Pointer Analysis for Refactoring JavaScript Programs


Anders Møller (Aarhus University, DK)

License © Creative Commons BY 3.0 Unported license
© Anders Møller
Joint work of Møller, Anders; Feldthaus, Asger; Millstein, Todd; Schäfer, Max; Tip, Frank

Modern IDEs support automated refactoring for many programming languages, but for dynamic languages, such as JavaScript, the tools are still primitive. This talk presents two approaches toward tool supported renaming refactoring for JavaScript: 1) using (almost sound) pointer-analysis for fully automatic refactorings, and 2) using a pragmatic variant of Steensgaard-style analysis for semi-automatic refactorings.

3.18 New Search Techniques for Query-Driven Dataflow Analysis

Mayur Naik (Georgia Institute of Technology)

License  Creative Commons BY 3.0 Unported license
© Mayur Naik

A central problem in static analysis concerns how to balance its precision and cost. A query-driven analysis seeks to address this problem by searching for an abstraction that discards program details that are unnecessary for proving an individual query. I will describe our results and experience over the past four years addressing this problem in the context of query-driven dataflow analyses that are parametric in the abstraction. The abstraction is chosen from a large family that allow abstracting different parts of a program with varying precision. A large number of fine-grained abstractions enables an analysis to specialize to a query but poses a hard search problem in practice. Our main result is a set of new search techniques (black-box and white-box approaches, deterministic and randomized approaches, purely static and hybrid dynamic-static approaches) for new problems (minimal abstractions, necessary conditions, impossibility results) that show promise for realistic pointer-related analyses on medium-to-large Java programs from the Dacapo suite.

3.19 Sparse Analysis Framework

Hakjoo Oh (Seoul National University, KR)

License  Creative Commons BY 3.0 Unported license
© Hakjoo Oh

Joint work of Oh, Hakjoo; Heo, Kihong; Lee, Wonchan; Lee, Woosuk; Yi, Kwangkeun
Main reference H. Oh, K. Heo, W. Lee, W. Lee, K. Yi, “Design and Implementation of Sparse Global Analyses for C-like Languages,” in Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI’12), pp. 229–238, ACM, 2012.
URL <http://dx.doi.org/10.1145/2254064.2254092>

In this talk, I present a general method for achieving global static analyzers that are precise, sound, yet also scalable. Our method, on top of the abstract interpretation framework, is a general sparse analysis technique that supports relational as well as non-relational semantics properties for various programming languages. Analysis designers first use the abstract interpretation framework to have a global and correct static analyzer whose scalability is unattended. Upon this underlying sound static analyzer, analysis designers add our generalized sparse analysis techniques to improve its scalability while preserving the precision of the underlying analysis. Our method prescribes what to prove to guarantee that the resulting sparse version should preserve the precision of the underlying analyzer.

3.20 Empirical Evaluation of Points-To Analyses

Erhard Plödereder (Universität Stuttgart, DE)

License  Creative Commons BY 3.0 Unported license
© Erhard Plödereder

Joint work of Frohn, Simon; Staiger-Stoehr, Stefan; Plödereder, Erhard

Over the years, we have run several experiments to evaluate the performance and precision of various points-to analyses in the context of global program analyses. The results have

significantly influenced the approach taken towards points-to analyses in the Bauhaus program analysis framework. In a first part, this talk reports on the results of a comparative study (work by Simon Frohn) of the three well-known algorithms by Andersen, Das, and Steensgaard. The evaluation showed that the Das algorithm ran in near-linear time and was hardly more expensive than Steensgaard's but produced considerably more precise results. The Anderson algorithm lived up to its worst-case cubic performance, while not improving precision by much over the Das algorithm. Field sensitivity added considerable precision but also added a (small) factor to the execution time. An important insight was that the size of the points-to sets clearly correlated with the size of the test programs, refuting the hypothesis that locality would put constant bounds on the size of these sets. Quantitative results are shown in comparative charts both on execution time and performance metrics of the three algorithms including some variations. The second part of the talk describes the principles of our best-yet algorithm developed by Stefan Staiger-Stoehr in his Ph.D. thesis partly in response to the findings of Simon Frohn. As a flow-sensitive analysis it combines control flow analysis, call graph construction, SSA-construction and points-to analysis. It starts out presuming no data-flow effects from dereferenced pointers and then iterates towards a conservative fix-point using the gradually constructed points-to sets. Strong indirect updates are recognized and exploited. Proven to be of cubic complexity in MOP-accuracy and of bi-quadratic complexity in MOVP-accuracy, the algorithm is efficient enough to allow the analysis of programs of a quarter of a million lines of code in less than one hour, and often in less than 10 minutes. Again, empirical data on execution time and precision are presented, comparing variants that are flow-insensitive, flow-sensitive, and flow-sensitive with strong indirect updates. The data shows considerable reduction of the average size of points-to sets by flow-sensitive analysis but only marginal reductions from strong indirect updates. Execution time measurements were not entirely conclusive between linear and quadratic behavior for tests with up to 250.000 lines of code. Surprisingly, the strong indirect updates make a significant difference on the number of SSA-nodes generated (the algorithm uses Phi-nodes to represent weak updates). Compared to flow-insensitive analysis, up to 25% of SSA-nodes are avoided by flow-sensitive analysis, and up to 40% of strong indirect updates are recognized. A factor of 5 in execution time between the least and the most discriminating analyses makes the added effort well worthwhile, as the SSA-form is subsequently processed by various user-oriented program analyses.

References

- 1 Simon Frohn. *Konzeption und Implementierung einer Zeigeranalyse für C und C++*. Diploma thesis, University of Stuttgart, Stuttgart, Germany, 2006
- 2 Stefan Staiger-Stoehr. *Kombinierte statische Ermittlung von Zeigerzielen, Kontroll- und Datenfluss*. doctoral dissertation, University of Stuttgart, Stuttgart, Germany, 2009
- 3 Stefan Staiger-Stoehr. *Practical Integrated Analysis of Pointers, Dataflow and Control Flow*. ACM Transactions on Programming Languages and Systems, Vol. 35, No. 1, Article 5, April 2013

3.21 Set-Based Pre-Processing for Points-To Analysis

Yannis Smaragdakis (University of Athens, GR)

License © Creative Commons BY 3.0 Unported license
© Yannis Smaragdakis

Joint work of Smaragdakis, Yannis; Balatsouras, George; Kastrinis, George

Main reference Y. Smaragdakis, G. Balatsouras, G. Kastrinis, “Set-Based Pre-Processing for Points-To Analysis,” in Proc. of the 28th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’13), to appear.

We present set-based pre-analysis: a virtually universal optimization technique for flow-insensitive points-to analysis. Points-to analysis computes a static abstraction of how object values flow through a program’s variables. Set-based pre-analysis relies on the observation that much of this reasoning can take place at the set level rather than the value level. Computing constraints at the set level results in significant optimization opportunities: we can rewrite the input program into a simplified form with the same essential points-to properties. This rewrite results in removing both local variables and instructions, thus simplifying the subsequent value-based points-to computation. Effectively, set-based pre-analysis puts the program in a normal form optimized for points-to analysis.

Compared to other techniques for off-line optimization of points-to analyses in the literature, the new elements of our approach are the ability to eliminate statements, and not just variables, as well as its modularity: set-based pre-analysis can be performed on the input just once, e.g., allowing the pre-optimization of libraries that are subsequently reused many times and for different analyses. In experiments with Java programs, set-based pre-analysis eliminates 30% of the program’s local variables and 30% or more of computed context-sensitive points-to facts, over a wide set of benchmarks and analyses, resulting in an over 20% average speedup.

3.22 Pointer Analysis for Probabilistic Noninterference

Gregor Snelting (KIT – Karlsruhe Institute of Technology, DE)

License © Creative Commons BY 3.0 Unported license
© Gregor Snelting

Joint work of Dennis Giffhorn; Gregor Snelting

Main reference D. Giffhorn, G. Snelting, “A New Algorithm for Low-Deterministic Security,” Karlsruhe Reports in Informatics 06/2012, revised 2013.

URL <http://pp.info.uni-karlsruhe.de/publication.php?id=giffhorn13lsod>

Information Flow Control (IFC) analyses program source or machine code to discover possible violations of confidentiality (i.e. secret information is leaked to public ports) or integrity (i.e. critical computations are manipulated from outside). IFC algorithms must handle realistic programs in e.g. full Java; they must be provably sound (discover all potential leaks) and precise (produce no false alarms). For full Java, this not only requires flow- context- object- and field-sensitive analysis of explicit and implicit information flow, but also a precise pointer analysis, in particular for nested objects and exception handling. For concurrent programs, all potential leaks exploiting scheduling or interleaving effects must be discovered.

Probabilistic noninterference (PN) is the established technical criterion for IFC of concurrent programs operating on shared memory. IFC and PN algorithms can be based on non-standard type systems, or on program dependence graphs (PDGs). In any case, PN requires precise May-Happen-in-Parallel (MHP) information, which in turn requires precise pointer and alias analysis.

The talk presents basic examples for IFC and precision issues, and sketches PDG-based PN, as recently introduced by Giffhorn & Snelting. It then discusses challenges for pointer analysis in PN and MHP. It is shown that dynamic pushdown networks, as introduced by Müller-Olm et al., allow for lock-sensitive IFC and PN analysis, but require precise must-alias information.

References

- 1 C. Hammer, G. Snelting: *Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs*. International Journal of Information Security, Vol. 8 No. 6, 2009, pp. 399–422.
- 2 D. Giffhorn, G. Snelting: *A New Algorithm for Low-Deterministic Security*. Karlsruhe Reports in Informatics 06/2012, revised 2013, submitted for publication.

3.23 Pointer Analysis and Reflection

Manu Sridharan (IBM TJ Watson Research Center – Yorktown Heights, US)

License © Creative Commons BY 3.0 Unported license
© Manu Sridharan

Joint work of Sridharan, Manu; Dolby, Julian; Fink, Stephen J.; Chandra, Satish; Schaefer, Max; Tip, Frank

Over the last several years, our group at IBM Research has put considerable effort into building industrial-strength pointer analyses for Java and JavaScript programs. For both languages, one of the biggest challenges we faced was handling reflective code, particularly in libraries and frameworks. In my talk, I presented some of the problems we have faced, approaches we have tried, the strengths and weaknesses of these approaches, and some ideas on how to make progress going forward.

References

- 1 Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Dynamic determinacy analysis. In *PLDI*, 2013.
- 2 Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4F: taint analysis of framework-based web applications. In *OOPSLA*, 2011.
- 3 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *ECOOP*, 2012.
- 4 Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *PLDI*, 2009.

3.24 Modular combination of shape abstraction with numeric abstraction

Xavier Rival (ENS, Paris)

License © Creative Commons BY 3.0 Unported license
© Xavier Rival

In this talk, we will discuss techniques to combine in a single static analysis pointer abstract domains with value abstract domains. Such combinations are required whenever pointer information is required in order to discover value (numeric, boolean...) properties and vice versa. We will show an abstract domain combination technique, which allows to build static analyzers in a modular manner, and let domains that abstract different kinds of elements exchange information.

3.25 Scaling flow analysis using big-step semantics

Dimitris Vardoulakis (Google Inc. – Mountain View, US)

License © Creative Commons BY 3.0 Unported license
© Dimitris Vardoulakis

Joint work of Vardoulakis, Dimitris; Shivers, Olin

Main reference CFA2: a Context-Free Approach to Control-Flow Analysis, ESOP 2010.

URL <http://www.ccs.neu.edu/home/dimvar/papers/cfa2-lmcs11.pdf>

Traditional flow analyses for higher-order languages, such as k-CFA, approximate programs as control-flow graphs. This approximation does not allow precise analysis of function calls and returns; during the analysis, a function may be called from one program point and return to a different one.

I will argue that call/return mismatch is undesirable in a flow analysis, especially in a higher-order setting, where function call and return is the central mechanism for control-flow transfer. Pushdown flow analyses, such as CFA2, offer greater precision than traditional analyses because they can match an unbounded number of calls and returns.

The increased precision of CFA2 is not obtained at the expense of scalability. I will discuss how to implement CFA2 as an abstract interpretation of a big-step operational semantics. The use of big-step semantics minimizes caching of abstract states (and thus memory usage) because it does not require one to remember the analysis results for each sub-expression in the program. It also makes the analysis faster because it requires fewer comparisons between abstract states than summarization-based analyses. I have implemented this analysis for JavaScript and used it to analyze all Firefox add-ons, with promising results.

References

- 1 Dimitrios Vardoulakis. *CFA2: Pushdown Flow Analysis for Higher-Order Languages*. PhD dissertation, Northeastern University, August 2012.

4 Breakout Sessions

4.1 Better APIs for Clients

José Nelson Amaral (University of Alberta, CA)

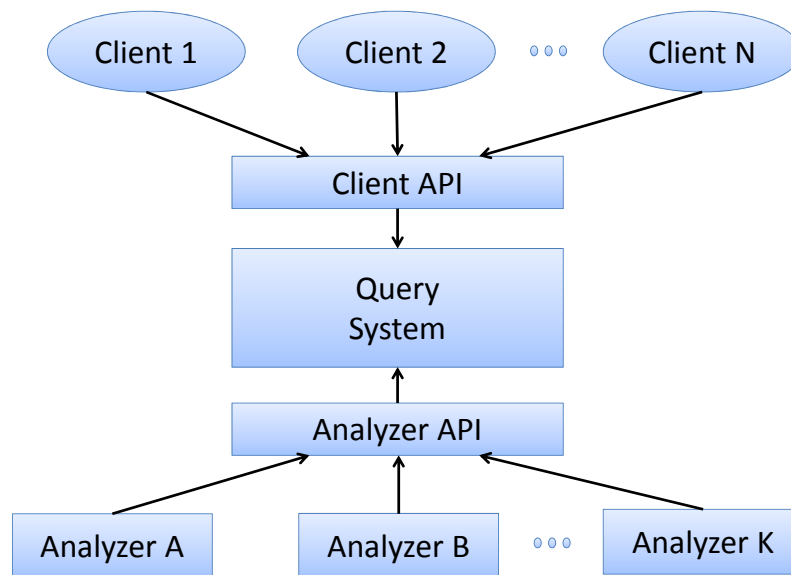
License © Creative Commons BY 3.0 Unported license
© José Nelson Amaral

This breakout session was motivated by the observation that it is difficult for a single person/group to design and implement a significant pointer/alias/reference analysis and then also implement meaningful clients to use the analysis and test its effectiveness on creating new opportunities for optimization based on the analysis results.

The initial idea is that if an interface could be defined between the analyses and the clients, then a new analysis could be tested on several existing clients and a new client could try to use several existing analyses.

However, after discussing the initial idea, we realised that we would need not a single interface, but rather two interfaces and a query system as shown in Fig. 1:

- **Client API:** interfaces with the clients and answers the questions that a client may ask
- **Analyzer API:** provides the facts that are generated by the analysis to the query system
- **Query System:** translate analysis facts into answers to client queries.



■ **Figure 1** Interfaces and the Query System for Pointer Analysis Clients.

The ensuing discussion made it obvious that the problem is significantly more complicated than the motivation and initial idea implied: both the analyzer and the client representations are usually tied to some intermediate representation of the code.

Client API: Examples of Possible Functions

```

must_alias(x, y, pi, cj)
must_not_alias(x, y, pi, cj)
points_to_set(x, pi, cj)

```

where:

x, y : variables in the program (how to represent abstraction of heap locations?)

pi : a point in the program (can we represent a point in a program without tying it to a specific intermediate representation?)

cj : a context (how should contexts be represented?)

This simple notation does not address either path sensitivity or field sensitivity.

Analyzer API

The analyzer API seems to need to be more complex than the client API and we did not look into it in detail.


Fairness

While such a system could potentially be built to measure the effect of an analysis outcome on clients, it would probably be unwise to use it to measure how fast an analysis works with

a client. For instance, if the client specifies a point in the program but the analyzer is flow insensitive, there is an overhead incurred by the framework that would not exist in a direct connection of the client with the analyzer without going through the APIs.

4.2 Pointer analyses for open programs (libraries/frameworks)

Eric Bodden (TU Darmstadt, DE)

License  Creative Commons BY 3.0 Unported license
© Eric Bodden

We talked about two possible problem cases, one being the situation where one wants to summarize library information for speeding up client analyses, effectively preventing the library from being analyzed over and over again. The question is how to pre-analyze a library as much as possible, even though nothing is known about possible clients. One idea was to use Class Hierarchy Analysis (CHA) but it was noted that even this would be unsound, as types may be missing. Regular allocation-site-based analyses yield similar problems, also causing call edges to be missed. It was hence suggested to simply proceed as follows:

- perform a points-to analysis on a best-effort basis, persisting the points-to information that can be computed without any client code
- then, as the client code is analyzed, simply load the persisted analysis facts and continue the fixed point iteration from there

We also noted that summaries for points-to analyses might be hard to store and interpret, as they would probably need to involve specifications with complex access paths. Summaries for client analyses may make more sense but to date there is no efficient system for computing such summaries. Another problem with summaries in general is that callbacks create holes which the summary must accommodate for.

The second application scenario was analyzing a library for internal code vulnerabilities that could be exploited by malicious client code. Interestingly, in this scenario any sound analysis must make worst-case assumptions; so if there is a callback that could execute client code then the analysis better assume that this code could call any library code it has a handle to because that's exactly what an attacker could do. A correct but imprecise assumption would hence be that the callback could call any method at all in the library. Of course, this would not lead one anywhere. The art is hence to find out what object handles the client could get access to, and based on this information what functions the client could possibly call. An interesting observation was that in turn the library itself can only call methods it knows about, i.e., methods that are defined in some interface that is part of the library itself (unless the library uses reflection, that is).

4.3 Shape Analysis and Pointer Analysis: Working Together

Bor-Yuh Evan Chang (University of Colorado – Boulder, US)

License  Creative Commons BY 3.0 Unported license
© Bor-Yuh Evan Chang

This session focused on identifying the characteristics that would typically define an analysis as either a shape analysis or a pointer analysis. While both are static program analyses that

infer relationships about memory addresses, they are mostly considered distinct sub-areas today. The intent of this session was first to better understand the technical and cultural differences between these two kinds of memory analyses and then by doing so to explore how to leverage the two perspectives together.

We first summarize the discussion about defining these two kinds of memory analyses. In the following, we try to speak in general terms, as certainly there are specific analysis algorithms that blend characteristics.

An initial, strawman definition of the difference was raised that is oversimplifying but still informative. Pointer analysis infers relationships about “stack pointers.” Or more precisely, it infers relationships about static, named addresses (e.g., global and local variables). Shape analysis infers relationships about “heap pointers.” Or more precisely, it infers relationships about dynamic, unnamed addresses (e.g., malloced addresses).

Of course, both kinds of analyses can derive relationships about both stack and heap pointers, and the most significant challenges for both arise from dynamically-allocated addresses.

Pointer analysis abstracts the heap of dynamically-allocated memory cells with an upfront, static partitioning. For a standard context-insensitive allocation site-based abstraction, the partitioning is clear: every dynamically-allocated address is bucketed into its allocation site and every partition is named by its allocation site. Adding context-sensitivity still fits this definition even though it seems a bit less clear: the partitions are more refined and not all partitions are necessarily named during the course of the analysis, but the partition to which a concrete memory cell belongs does not change during the course of the analysis. Overall, we called the pointer analysis approach “staticifying the dynamic.”

Shape analyses typically vary the heap abstraction during the course of the analysis (e.g., the partitioning of the heap may vary across program points). Though not necessarily a prerequisite, materialization and summarization operations that change the heap partitioning arise from this perspective. Materialization decides how a heap partition should be split, while summarization decides how heap partitions should be merged.


An important point raised was that the differences described above are orthogonal to the distinction between store-based versus store-less abstractions, even though some combinations are less represented in the literature than others.

We identified some cultural differences. A sub-area, often tied to shape analysis, tends to focus on what’s possible with no or little loss of precision. For example, it is not uncommon for such analyses to simply halt when a strong update is not possible (i.e., the concrete cell being updated cannot be identified). Another sub-area, often tied to pointer analysis, tends to focus on handling all programs and all language features with a best-effort approach to precision: weak updates must be supported even if we wish to avoid them. An important observation is that these perspectives need not be tied to whether an analysis is deriving shape or pointer information.

In summary, it is oversimplifying to say that “shape analysis is more precise pointer analysis” or “pointer analysis is more scalable shape analysis.” These clichés have a kernel of truth, but it seems likely that further advances can be made by borrowing perspectives and ideas from each sub-area.

4.4 Practical Aspects of Pointer Analysis

Manu Sridharan (IBM TJ Watson Research Center - Yorktown Heights, US)

License  Creative Commons BY 3.0 Unported license
© Manu Sridharan

This breakout session focused on discussion of aspects of pointer analysis typically not discussed in conference papers, such as negative results and low-level details. A number of interesting topics were covered, as summarized below:

- For Java points-to analysis, differences in reflection handling can make a dramatic impact on scalability, which can be non-obvious just from reading a paper.
- Although BDDs are essential in making some straightforward analysis specifications scalable, the participants could find no compelling case for using BDDs in *highly tuned* points-to analysis, at the present time. (This is a delicate balance that may shift in the future, however.) BDDs have high constant-factor overheads that can have a significant performance impact, and other techniques, such as careful writing of Datalog rules or a shared bit-vector repository, can be as effective in eliminating redundancy from the results.
- Cycle elimination seems to yield little to no benefit for Java points-to analysis, as type filters lead to many fewer cycles than for large C programs. However, other opportunities for compression of the constraint graph remain, as discussed, e.g., by Hardekopf and Lin [1].
- The topic of debugging point-to analysis implementations and tracking down the root cause of imprecision was discussed. Suggestions included:
 - standard practices for good software engineering (unit tests and assertions)
 - comparing analysis variants with well-known precision relationships, e.g., ensuring that a context-sensitive analysis is strictly more precise than the context-insensitive version.
 - comparing results from different analysis frameworks; while this may involve significant work due to differing handling of language constructs, the work can be worthwhile. Additionally, it was suggested that a good fuzz testing harness for pointer analyses would be highly useful. For tracking down imprecision, delta debugging was suggested, though this becomes tricky in the presence of multiple files.
- For points-to analysis of JavaScript and other dynamic languages, differing string handling was identified as another obscure source of very large differences in analysis precision and performance.
- Finally, participants listed known production systems making use of sophisticated points-to analysis. Examples included link-time optimization in some commercial compilers, the JavaScript JIT compiler in Mozilla Firefox,² Veracode’s analysis engine,³ JavaScript taint analysis in IBM Security AppScan Source Edition,⁴ and in an architecture analysis system.

References

- 1 Ben Hardekopf and Calvin Lin. Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis. In *SAS*, 2007.

² <https://wiki.mozilla.org/IonMonkey>

³ <http://www.veracode.com>

⁴ <http://www-03.ibm.com/software/products/us/en/appscan-source/>

Participants

- Jose Nelson Amaral
University of Alberta, CA
- Gogul Balakrishnan
NEC Lab. America, Inc. –
Princeton, US
- Eric Bodden
TU Darmstadt, DE
- Bor-Yuh Evan Chang
University of Colorado –
Boulder, US
- Isil Dillig
College of William and Mary, US
- Thomas Dillig
College of William and Mary, US
- Julian Dolby
IBM TJ Watson Res. Center –
Hawthorne, US
- Samuel Z. Guyer
Tufts University, US
- Christian Hammer
Universität des Saarlandes, DE
- Laurie J. Hendren
McGill University, CA
- Uday Khedker
Indian Institute of Technology –
Mumbai, IN
- Ondrej Lhotak
University of Waterloo, CA
- Benjamin Livshits
Microsoft Res. – Redmond, US
- Welf Löwe
Linnaeus University – Växjö, SE
- Mark Marron
Microsoft Res. – Redmond, US
- Matt Might
University of Utah, US
- Ana Milanova
Rensselaer Polytechnic, US
- Anders Moeller
Aarhus University, DK
- Mayur Naik
Georgia Inst. of Technology, US
- Hakjoo Oh
Seoul National University, KR
- Erhard Plödereder
Universität Stuttgart, DE
- Xavier Rival
ENS – Paris, FR
- Yannis Smaragdakis
University of Athens, GR
- Gregor Snelting
KIT – Karlsruhe Institute of
Technology, DE
- Manu Sridharan
IBM TJ Watson Research Center
– Yorktown Heights, US
- Bjarne Steensgaard
Microsoft Res. – Redmond, US
- Dimitris Vardoulakis
Google Inc. –
Mountain View, US

