Pointer Analysis: Haven't We Solved This Problem Yet?

Michael Hind
IBM Watson Research Center
30 Saw Mill River Road
Hawthorne, New York 10532
hind@watson.ibm.com

ABSTRACT

During the past twenty-one years, over seventy-five papers and nine Ph.D. theses have been published on pointer analysis. Given the tomes of work on this topic one may wonder, "Haven't we solved this problem yet?" With input from many researchers in the field, this paper describes issues related to pointer analysis and remaining open problems.

1. INTRODUCTION

Analyzing programs written in languages with pointers requires knowledge of pointer behavior. Without such knowledge, conservative assumptions regarding pointer accesses must be made, which can adversely affect the precision and efficiency of any analysis that requires this information, such as an optimizing compiler or a program understanding tool.

A pointer analysis attempts to statically determine the possible runtime values of a pointer. As such an analysis is, in general, undecidable [51, 49, 70, 42], a large collection of approximation algorithms have been published that provide a trade-off between the efficiency of the analysis and the precision of the computed solution. The *worst-case* time complexities of these analyses range from almost linear [94] to doubly exponential [88]. To complicate matters, worst-case behavior is often not indicative of typical performance.

Given the long history of pointer analysis research, it is appropriate to take stock of the current field and to outline those problems that remain open. This paper, with valued input from many pointer analysis researchers, attempts to serve this role, as well as categorizing existing work.

2. BACKGROUND

A *pointer alias analysis* attempts to determine when two pointer expressions refer to the same storage location. A *points-to analysis* [27, 22, 2], or similarly, an analysis based on a "compact representation" [13, 5, 38], attempts to determine what storage locations a pointer can point to. This information can then be used to determine the aliases in the program.[1] Alias information is central to determining what memory locations are modified or referenced.

There are several dimensions that affect the cost/precision trade-offs of interprocedural pointer analyses. How a pointer analysis addresses each of these dimensions helps to categorized the analysis. An empirical comparison with a difference in more than one dimension can limit the usefulness of the comparison. Some of the dimensions are

Flow-sensitivity: Is control-flow information of a procedure used during the analysis? By not considering control flow information, and therefore computing a conservative summary, flow-insensitive analyses compute one solution for either the whole program (such as [2, 94, 108, 91]) or for each method (such as [5, 38, 55]), whereas a flow-sensitive analysis computes a solution for each program point. Flow-insensitive analyses thus can be more efficient, but less precise than a flow-sensitive analysis. Flow-insensitive analyses are either *equality-based* [94, 108], which treat assignments as bidirectional and typically use a union-find data structure, or *subset-based* [2, 5, 38], which treat an assignment as a unidirectional flow of values.

Context-sensitivity: Is calling context considered when analyzing a function or can values flow from one call through the function and return to another caller?

Heap modeling: Are objects named by allocation site, or is a more sophisticated shape analysis performed?

Aggregate modeling: Are elements of aggregates distinguished or collapsed into one object?

Whole program: Does an analysis require the whole program or can a sound solution be obtained by analyzing only components of a program?

Alias representation: Is an explicit alias representation [51, 64] or a points-to/compact representation used?

3. GENERAL ISSUES

Before discussing open problems, we first address some more general issues that plague the field of pointer analysis.

3.1 Terminology

The pointer analysis community has sometimes done a disservice to its audience by using different terminology to refer to the same concepts. For example, context-sensitive/insensitive analysis are also known as *poly/mono-variant* analyses. *Unification-based* flow-insensitive analyses are also known as *Steensgaard-style* analyses and similarly, *inclusion-based* flow-insensitive analyses are also known as *Andersen-*

[1] The point at which such information is inferred can affect the precision and efficiency of the analysis [61, 6, 38, 86].

PASTE'01, June 18-19, 2001, Snowbird, Utah, USA.

style analyses. When these analyses are formulated as constraint-based analyses, they are referred to as *term* or *equality*-based and *inclusion* constraints, respectively. This dichotomy of terminology can be attributed to work in the related field of type inference [93, 37, 17, 65], which itself is also known as *control-flow analysis* and *class analysis*.[2]

Pointer analysis, (pointer) *alias* analysis, and *points-to* analysis are often used interchangeably. We prefer to use *pointer analysis* as a general term for an analysis that analyzes pointers and *alias* (*points-to*) analysis for analyses that produce *alias* (*points-to*) relations.

3.2 Metrics

How does one measure the precision of a pointer analysis? A popular metric, which we define as the *direct metric*, is to record the average number of objects aliased to pointer expressions appearing in the program [52]. Although this *could be* a direct indication of the precision of an analysis, there are several flaws with this metric [20].

- Because an analysis models an unbounded number of dynamic objects (due to recursive locals and dynamic allocation), the number of objects aliased to a pointer expression can be greatly skewed depending on the model chosen [80]. For example, an analysis that models the heap as one object will report a (low) average of one object (the whole heap!) for all heap-directed pointers, incorrectly suggesting a precise analysis.

- Pointer information is usually the input to other client analyses, and as such the precision of a pointer analysis can depend on how it affects the client. For example, it may take only one extra alias to create a dependence that prevents code motion in a time-critical loop. In contrast, adding additional aliases to a pointer dereference that already is fully dependence constrained will have no effect for a dependence-based client.

- Even with a consistent modeling scheme, a particular value is not meaningful in isolation because the program may contain pointers that can point to different objects at runtime, such as the SPEC95 program `go`, which contains a function with a pointer parameter that can be aliased to 100 distinct global arrays [39].

There are three alternative categories of metrics, two static and one dynamic. The first metric compares the static precision to worst-case assumptions, be it using the direct method [51] or the results of dependence-based queries [31]. This metric will be particularly useful in evaluating analyses for strongly-typed languages such as Java, where worst-case assumptions are not as bad as in C. The second alternative static metric is to implement a pointer analysis client and report its precision [90, 63, 3, 30, 20, 86, 95, 55, 57, 40, 60, 31, 77]. The advantage of this metric is that it more clearly ascertains the effectiveness of a pointer analysis for a particular client analysis. The limitation is that it only measures one client, and again, the importance of the static measure of the client may be questioned.

[2]Type inference attempts to ascertain the runtime types of pointer values, whereas points-to analysis uses a finer level of granularity and tries to determine the (named) objects held in a pointer. The type inference literature, although certainly worth exploring, is beyond the scope of this paper.

The third approach dynamically measures how pointer information affects a runtime property, such as program performance [14, 105, 20, 30, 12, 31], dynamic points-to relations [20, 62], or a dynamic characterization of the client analysis [20, 30, 12, 31]. Such metrics are limited to a single execution of the program, and thus, represent a lower bound on their static counterparts. Because all metrics have their strengths and weaknesses a combination should be used [20].

3.3 Reproducible Results

In most sciences a result is not accepted until it can be independently verified. Unfortunately, this practice is not well accepted in pointer analysis. Furthermore, when one does undertake such a verification, it is difficult to publish such results, unless it disputes common wisdom, such as [80].

Susan Horwitz echoes this sentiment: "Improvements proposed by researchers seem promising, but seldom are claims independently verified, and often promising leads are abandoned. It seems that duplicating others' results is considered very important in the physical sciences, but gets short shrift in computer science. Should we/can we change that attitude?"

Even if publishability issues are left aside, it can be difficult to reproduce a result because of different intermediate representations, benchmark suites, or benchmark versions. For example, some intermediate representations decompose all assignments into canonical forms to limit the number of pointer dereferences in a statement with the goal of simplifying the exposition and implementation of the analysis. However, this can increase the number of program variables and pointer dereference expressions, making comparisons to results that do not perform this simplification meaningless.

However, some progress has been made on this topic because many researchers share the benchmarks used in their studies and in some cases make their implementations publicly available [4, 69, 15].

4. OPEN QUESTIONS

This section discusses open questions in pointer analysis.

4.1 Scalability

Equality-based flow-insensitive analyses [94] can analyze million-line programs quickly. Although recent work [15, 55, 59] has improved the precision of equality-based analyses, it is still not clear if the precision is sufficient. Meanwhile, significant work [23, 97, 76, 24, 72, 34] has increased the efficiency of the more precise subset-based flow-insensitive analyses. Studies [12, 40] suggest that subset-based analyses may provide sufficient precision for some clients. Thus, the convergence of these two efforts may result in a reasonably precise analysis that can be effective on very large programs. However, even the precision of subset-based analyses may not be sufficient for all clients. Clearly, more studies need to be conducted in this area.

Tom Reps asserts, "Various clients of pointer-analysis algorithms can produce poor results due to inherent imprecisions in the pointer-analysis phase. However, there are some interesting precision/efficiency trade-offs: for instance, it can be the case that a more precise pointer analysis runs more quickly than a less precise one. Moreover, even when a more precise pointer analysis is less efficient per se, in the context of the processing carried out by the client(s) of the analysis, the overall cost may decrease [90]. Thus, is it pos-

sible to understand what should go into a pointer analysis according to the needs of the client of the analysis? Present studies of 'scalable' analyses have not had much to say about more precise approaches: is there a 'scalable' analysis that provides context-sensitivity? flow sensitivity? some degree of these properties?"

Amer Diwan adds, "It is easy to make a pointer analysis that is very fast and scales to large programs. But are the results worth anything? While more people have done work in the area [14, 30, 20, 40, 41], we still need a better understanding of what pointer analysis one should use."

4.2 Improving Precision

How can we improve the precision of an analysis without sacrificing scalability? Bill Landi and Manuvir Das provide some interesting ideas for loosening the soundness/safety constraint on pointer analysis. Bill Landi writes, "One analysis that sparks interest is the detection of bugs in an R&D setting with, for example, a use-before-define (UBD) analysis on 20+ million LOC applications. The only way I have been able to address this situation is to reconsider the notion of safety. Clearly, if an analysis is to be used in program optimization, the roots of such analyses, safety is an essential concept. However, this does not mean that it is essential for all applications of analyses. My experience with unsafe analyses have led to some surprising observations. First, when I removed from my context- and flow-sensitive alias analysis algorithm [51, 48] a step that was essential to ensure safety, but did not seem to me to be necessary in almost all cases, a much faster analysis was possible. In one case runtime went from several days to several minutes. Further, in all the tests I did using this new calculation for computing MOD (about 20 programs), I only found one case of a missing modification. Second, when UBD results are reported to a user as potential bugs, both false-positives and false-negatives can be tolerated if they are within reason. If my analysis can find 90%, or even 25%, of the users bugs this is a vast improvement over using nothing. If there are too many false-positives, users will reject the analysis as they must spend too much time pursuing false leads. However, I was told the users actually liked the false-positives in my analysis because they claimed when my analysis got confused it was a good indication that the code was poorly written and likely to have other problems. This came as a complete surprise. While additional study is needed to claim these observations to be valid in a broader sense, they lead me to conclude that the notation of safety should be reconsidered for many applications of static analysis."

Manuvir Das adds, "Profile-directed optimizations do not need sound pointer information. Recent work [62] shows how dynamic points-to data can be used to perform optimization with checks to ensure safety, with the usual tradeoffs. More interestingly, error detection doesn't have to be sound at all! Consider a tool that examines some code paths and reports errors along those paths. Such tools do not find pointer analysis useful because it makes analysis too conservative. Here, we can use profile pointer data to let the tools know about only the pointer values that actually arise during test executions. It leads to incomplete coverage, but that's ok since the tool is not meant to find every error."

Another way to improve precision without sacrificing scalability is to limit the program scope where high-precision is required [79]. Susan Horwitz writes, "A potentially important area of research would address determining where accuracy is vital (e.g., certain regions of code, certain pointer variables) and finding ways to improve results just for those critical regions. Another idea is to find special cases where certain techniques work well (even if they don't work in general). An example is recent work [85] on techniques for symbolic bounds analysis for divide-and-conquer programs."

Bjarne Steensgaard observes, "There are also opportunities for finding as-yet undiscovered opportunities for improvements by simply looking at existing programming patterns. For example, Manuvir Das looked at several large C programs and found that passing the address of a variable as a function parameter was a large contributor to the loss of precision for flow-insensitive, context-insensitive algorithms. He found a way to add 'one level flow' to an otherwise unification-based algorithm to achieve results comparable with flow-sensitive, context-insensitive algorithms."

4.3 Designing an Analysis for a Client's Needs

One approach to determining appropriate precision and scalability is to consider the client problem's needs.

Barbara Ryder expands on this topic: "Pointer analyses should be designed to be appropriate in cost and precision for specific groups of client problems. What is practical and acceptably precise for one usage, may be impractical and too approximate for another. There is no one class of problems (e.g., all C programs over 1 million lines of code) that should decide the efficacy of a pointer analysis. We do not need a different pointer analysis per client problem, but rather we should look for classes of client problems with similar needs, and design analyses for these classes, checking performance in the context of these usages. We can all write an unbounded number of papers that compare different pointer analysis approximations in the abstract. However, this does not accomplish the key goal, which is to design and engineer pointer analyses that are useful for solving real software problems for realistic programs."

Manuel Fähndrich discusses client problems: "I think there are two distinct uses of pointer analysis, 1) optimizations, and 2) error detection/program understanding. These two uses have vastly different requirements on pointer analysis. For optimizations, there seems to be some upper bound on how much precision is useful because taking advantage of more precision usually translates into specializing more code, which needs to be bounded. In my opinion, the spectrum of analyses mostly covers the needs for optimizations. For error detection and program understanding, the picture is different. For these applications, there seems to be a lower bound on precision, below which, pointer information is pretty useless. Gearing pointer analysis towards error detection requires more work on precision and scaling issues."

Manuvir Das discusses correctness in the context of debugging tools: "All of the flow-insensitive pointer algorithms can be used to produce flow-insensitive versions of reaching definitions, which, assuming the analysis is context-sensitive, can be used in correctness and debugging tools. In fact, I believe that correctness (the elimination of certain kinds of errors, not the extreme step of verification) is the 'killer app' for pointer analysis. Although the hardware companies have made compiler optimizers look bad, there is nothing they can do about ensuring code correctness, which will matter more and more."

4.4 Flow-Sensitivity

Empirical studies [38, 40, 41] suggest that for context-insensitive analyses, a flow-sensitive analysis does not offer much precision improvement over a subset-based flow-insensitive analysis. Manuvir Das provides some intuition: "Flow-sensitivity is all about strong update. A spurious flow of pointer values produced by a flow-insensitive context-insensitive analysis occurs because such an analysis lacks strong update through two kinds of assignments: explicit assignments and parameter passing. As long as the analysis is context-sensitive, it effectively treats parameter passing as strong updates and only loses on the explicit assignments. At least for C programs, most explicit assignments updates of pointers deal with traversing data structures, for which pointer analysis is no good anyway. Although I don't believe a flow-sensitive pointer analysis is of any use above a flow-insensitive context-sensitive pointer analysis, a path-sensitive pointer analysis can be very useful in error detection tools. Because computing path-sensitive pointer information would be extremely expensive, we need to investigate ways to obtain this information."

4.5 Context-Sensitivity

Because a context-sensitive analysis can be exponential in the worst case, the efficiency of such analyses has been addressed [22, 105, 104]. However, another question is whether context-sensitivity improves precision. So far the results have been mixed: context-sensitivity did not improve precision for a common flow-sensitive analysis [80] and a similar result has been shown for subset-based flow-insensitive analysis [26] and an extension of the equality-based flow-insensitive analysis [16]. However, context-sensitivity has been shown to be beneficial for simpler equality-based flow-insensitive analysis [26]. This latter result seems to validate the recent activity in partially context-sensitive equality-based analysis [55, 24, 72, 16]. Other work [56, 58] attempts to recover additional context in client analyses.

The papers above used the direct precision metric, which has the limitations described in Section 3.2. Although these studies are extremely valuable, their generality should be confirmed in different environments, selecting different pointer analysis dimensions, using different programs.

Manuvir Das adds his opinion about whether context-sensitivity is useful: "The answer is definitely a yes, even though recent studies [26, 16] suggest otherwise. First, a more general use of pointer analysis is to develop tools that track the flow of values or information in programs. Here, context-sensitivity is crucial to avoid spurious flow of values. Second, if the client optimization or error detection is willing to be context-sensitive or to specialize copies of procedures, context-sensitive pointer analysis can produce much more precise information. Unfortunately, there is no hard evidence of this, but we're working on it."

Erik Ruf writes, "One of the big issues/sources of confusion in most points-to analysis lies in the idea of computing a points-to solution and then using it for some purpose. Usually, a fixed context-sensitivity strategy (and a corresponding fixed naming strategy) is used, which may or may not correspond to what's appropriate for the client. For example, the traditional approach of performing a context-sensitive flow with respect to callees affecting callers, but not with respect to callers affecting callees (as occurs when no cloning is performed) can yield bad code (or, in software engineering applications, a worst-case view for the user). On the other hand, eagerly building clones inside a stand-alone pointer analysis is undesirable because of the potential exponential work, much of which may not be useful. Even highly parameterized standalone analyses pay costs for representing contexts not experienced by the client, yet cannot take full advantage of client-level semantic information to improve points-to information within specialized contexts."

Erik Ruf continues, "I advocate an approach where pointer analysis is integrated with the client analysis. Allowing the client's generation and consumption of information to drive the pointer analysis has the potential to improve both precision and performance. A simple example of this approach is the incorporation of client analysis information into procedure-level pointer/alias summaries [81]. Client-driven pointer analyses may also be able to selectively apply more aggressive approaches to obtaining precision, such as those of [68]."

4.6 Heap Modeling

Shape analysis algorithms [53, 43, 9, 19, 36, 35, 18, 74, 28, 29, 87, 88, 89, 103, 21, 54] have demonstrated high precision, over schemes that name objects based on allocation site (as in [47, 44, 83, 43, 9]), but their scalability to even medium programs is uncertain. Mooly Sagiv summarizes the current state of the field: "Although I believe we are making good progress, in all honesty, we should say that we are not there yet."

Tom Reps observes, "We have certainly not come to the end of the line in research on shape analysis. There are all sorts of interesting issues here, ranging from a better understanding of how to identify the important ingredients of an analysis (the 'instrumentation predicates') to efficiency questions. The approach is also yielding insights into problems that are not what you would *ordinarily* think of as pointer problems or shape-analysis problems; however, 'shape' serves as a kind of metaphor for many kinds of properties of system and/or memory configurations that can arise as a computation evolves."

Manuvir Das adds, "One suggestion is to develop shape analyses that work well on the common case, but may lose precision in the general case. Another suggestion is to combine a cheap global pointer analysis with a local shape analysis. This approach relies on the fact that heap traversal code is local, and pointer analysis can be used to ensure that non-local code is not modifying the heap structure being traversed."

A similar approach was expressed by Laurie Hendren's group in analyzing C [22]. This approach first distinguishes stack and heap-directed pointers and then performs a simple shape analysis [28, 29] on just the heap-directed pointers. An alternative line of research attempts to compensate for the loss of precision experienced by the allocation site naming scheme in the presence of user-defined memory allocation routines [13, 1, 34].

4.7 Modeling Aggregates

A key implementation detail is whether aggregate components are distinguished or summarized into one object. C/C++'s weak typing makes this difficult to address correctly. Thus, most published work does not distinguish aggregates. However, this difficulty does not exist in a strongly-type language like Java, and therefore, components should

be distinguished in such languages [77]. Most recent work has chosen to distinguish components [105, 71, 84, 106, 60, 33, 34, 31, 77]. Unfortunately, few researchers [106, 86, 60, 77] have studied the impact of this decision.

Rakesh Ghiya states, "We need to focus more energy on improving the basis information for pointer analysis (especially malloc-site identification in the presence of user-defined memory management, and handling of fields), as opposed to solely focusing on incremental improvements in the propagation techniques."

4.8 Demand-Driven/Incremental Analyses

Because the efficiency of pointer analysis is often a concern, it would seem that a demand-driven or incremental approach would be useful. To date, only limited progress has been made on incremental analysis [107, 98] and all demand-driven analyses are flow-insensitive [73, 75, 15, 33, 24, 72, 16]. It remains an open question as to whether the precise flow-sensitive analyses, such as those that use context-sensitivity or perform shape analysis, can be performed in a demand-driven manner.

4.9 Java and Object-Oriented Languages

There has been little work done in the area of pointer analysis for object-oriented programs. Some have modified pointer analyses to handle dynamic dispatch in C++ [67, 8, 66], while others have developed new modular algorithms [11, 10]. Others have focused on Java [102, 25, 96, 60, 77]. In Java all references are heap-directed, and thus, distinguishing the heap is even more important than it is in C. This may raise the importance of simpler shape analysis techniques [28, 29].

Jong-Deok Choi raises another issue: "Must-alias analysis is very important for Java because must alias information can help avoid re-accessing the same memory location. Also, we have found that must alias information can reduce the overhead for deterministic replay on SMP systems. However, not much work has been done for must-alias analysis."

Bjarne Steensgaard looks to the future: "I believe pointer analyses will continue to adapt to changes in both their input (programming languages and programming styles) and their output (tools and other analyses). It is fairly obvious that many of the pointer analysis algorithms that worked reasonably well on programs written in the C style of languages perform poorly on programs written in the Java style of languages. New algorithms will be created to deal with the specific issues and take advantage of the opportunities created by changes in programming paradigms, programming languages, and programming styles."

Although C will remain an important domain for pointer analysis in the context of program understanding tools, mostly because of legacy code, it is important to look at new languages, such as Java. In addition to offering new features over C, the efficacy of the techniques developed for C need to be revalidated in these languages.

4.10 Incomplete Programs

Most pointer analyses require the whole program. However, Michael Burke notes, "Component programming and the use of library code are becoming more prevalent. These trends will continue, making whole program analysis less useful. We need pointer analyses of components and libraries that are parameterized with respect to how they are configured in a full application. Although there has been some work in this area [79, 78], I haven't seen a full solution to this problem."

Amer Diwan adds, "There has been relatively little work understanding how to perform pointer analysis in a really 'real' environment. Components of a real environment include very large programs, programs that use threads, incomplete programs — libraries etc., incremental compilation and fast turn-around times, dynamic linking, and ugly programs. Although there has been some work in these area (e.g., [15, 34], etc. on very large programs, [84] on threads, and [79, 78] on incomplete programs) there hasn't been anywhere near enough. In particular, a pointer analysis that analyzes threads or incomplete programs is still a novelty and we do not understand how the precision of analyses is affected. Maybe an analysis that is right on the 'sweet spot' in an ideal world situation (whole program, no threads, ...) may be the worst one to use in a real world situation. Regarding ugly programs, many real world programs (particularly for unsafe languages) have features that are practically impossible for a pointer analysis to get. For example, consider the C program `li` that includes a garbage collector. If a pointer analysis is unfortunate enough to analyze the garbage collector it will most likely determine that everything aliases everything else. I think that we will need to learn some lessons from the parallelization community: for the most part (in my biased opinion) most of the parallelization folks do not believe anymore that fully automatic parallelization is doable in general, and are resorting to getting feedback from the user or requiring certain kinds of programming styles. Maybe pointer analysis needs to do similar things."

Manuel Fähndrich expands on the idea of programmer involvement: "Modular analysis addresses both the problem of analyzing incomplete programs and the problem of scaling an analysis to large programs. Interface declarations that describe sharing and non-sharing relationships between data structures (shape descriptions) could be a way towards getting more precise pointer information for error detection and optimizations at the cost of programmer annotations. I'm excited by recent progress in the area of type systems [99, 100] and formal logics [46] that provide languages in which to express such interface declarations."

4.11 Engineering Insights

A significant characteristic of a pointer analysis is both its time and memory efficiency. Most conference papers do not allow sufficient room to describe an algorithm, an empirical comparison of its precision and efficiency, related work, and implementation details. Unfortunately, this last section rarely gets written. However, careful engineering of a pointer analysis, particularly a flow-sensitive analysis, can dramatically improve its performance and scalability [82, 104, 109, 110, 39]. If we expect that production systems will use a new algorithm, we must encourage the implementors to describe the engineering of their analysis.

5. CONCLUSIONS

In summary, it is clear that many open pointer analysis problems remain. Because client problems have different scalability and precision requirements, future work should identify the client problems they are addressing, such as optimizations or program understanding tools. Furthermore,

the metrics used to evaluate such analysis should be appropriate for the client problems.

One possible direction for future research is to require the programmer to aid the analysis with assertions or additional type information as discussed in Sections 4.10. An example of this is the pointer type qualifier `restrict` introduced in the recent ANSI standard for C [7]. When such a keyword is used for a pointer, the programmer asserts that the pointer will not point to a global or local that is directly accessed during the scope of the pointer. The net effect is that such pointers no longer require stack-based pointer analysis. Similar suggestions have been proposed for C pointers [45, 35] and for Java collection classes [60].

6. ACKNOWLEDGMENTS

We are grateful to Michael Burke, Jong-Deok Choi, Manuvir Das, Amer Diwan, Manuel Fähndrich, Rakesh Ghiya, Susan Horwitz, Bill Landi, Tom Reps, Erik Ruf, Barbara Ryder, Mooly Sagiv, and Bjarne Steensgaard for sharing their ideas on the current state of pointer analysis. We also thank Donglin Liang for his input and Matthew Arnold, Manuvir Das, Manuel Fähndrich, Stephen Fink, David Grove, G. Ramalingam, Barbara Ryder, and Laureen Treacy for improving the quality of this paper.

7. REFERENCES

[1] R. Z. Altucher and W. Landi. An extended form of must alias analysis for dynamic allocation. In *22nd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 74–84, Jan. 1995.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. Available at ftp.diku.dk/pub/diku/semantics/papers/D-203.dvi.Z.

[3] D. C. Atkinson and W. G. Griswold. Effective whole-program analysis in the presence of pointers. In *ACM SIGSOFT 1998 Symposium on the Foundations of Software Engineering*, pages 131–142, Nov. 1998.

[4] BANE — The Berkeley ANanlysis Engine. http://www.cs.berkeley/Research/Aiken/bane.html.

[5] M. Burke, P. Carini, J.-D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Lecture Notes in Computer Science, 892*, pages 234–250. Springer-Verlag, 1994. Proceedings from the *7th Workshop on Languages and Compilers for Parallel Computing*.

[6] M. Burke, P. Carini, J.-D. Choi, and M. Hind. Interprocedural pointer alias analysis. Research Report RC 21055, IBM T. J. Watson Research Center, Dec. 1997.

[7] C9X Final Commmittee Draft, Apr. 1999. http://reality.sgi.com/homer_craypark/c9x/restrict-c99.html.

[8] P. Carini, M. Hind, and H. Srinivasan. Flow-sensitive interprocedural type analysis for C++. Research Report RC 20267, IBM T. J. Watson Research Center, Nov. 1995.

[9] D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 296–310, 20–22 June. *SIGPLAN Notices* 25(6), June 1990.

[10] R. Chatterjee. *Modular Data-flow Analysis of Statically Typed Object-Oriented Programming Languages*. PhD thesis, Rutgers University, Oct. 1999. DCS-TR-406.

[11] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In *26th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, Jan. 1999.

[12] B.-C. Cheng and W. mei Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pages 57–69, 18–21 June. *SIGPLAN Notices* 35(5), June 2000.

[13] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Jan. 1993.

[14] K. D. Cooper and J. Lu. Register promotion in C programs. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 308–319, 15–18 June. *SIGPLAN Notices* 32(5), May 1997.

[15] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pages 35–46, 18–21 June. *SIGPLAN Notices* 35(5), June 2000.

[16] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Seventh International Static Analysis Symposium*, July 2001.

[17] G. DeFouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. In *Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 222–236, Jan. 1998.

[18] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 230–241, 20–24 June. *SIGPLAN Notices* 29(6), June 1994.

[19] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 157–168, Jan. 1990.

[20] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 106–117, 17–19 June. *SIGPLAN Notices* 33(5), May 1998.

[21] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *Seventh International Static Analysis Symposium*, June 2000.

[22] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 242–256, 20–24 June. *SIGPLAN Notices* 29(6), June 1994.

[23] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 85–96, 17–19 June. *SIGPLAN Notices* 33(5), May 1998.

[24] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pages 253–263, 18–21 June. *SIGPLAN Notices* 35(5), June 2000.

[25] S. Fink, K. Knobe, and V. Sarkar. Unified analysis of array and object references in strongly typed languages. In *Seventh International Static Analysis Symposium*, June 2000.

[26] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Seventh International Static Analysis Symposium*, June 2000.

[27] R. Ghiya. Interprocedural aliasing in the presence of function pointers. ACAPS Technical Memo 62, McGill University, Dec., 1992.

[28] R. Ghiya and L. J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6):547–578, 1996.

[29] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15, Jan. 1996.

[30] R. Ghiya and L. J. Hendren. Putting pointer analysis to work. In *25th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 121–133, Jan. 1998.

[31] R. Ghiya, D. Laverty, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of the ACM SIGPLAN'01*

Conference on Programming Language Design and Implementation, 20–22 June. To appear.

[32] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 97–105, 17–19 June. *SIGPLAN Notices* 33(5), May 1998.

[33] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, 20–22 June. To appear.

[34] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, 20–22 June. To appear.

[35] L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis of imperative programs. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 249–260, 17–19 June. *SIGPLAN Notices* 27(7), July 1992.

[36] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, Jan. 1990.

[37] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, Apr. 1993.

[38] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, July 1999.

[39] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In G. Levi, editor, *Lecture Notes in Computer Science, 1503*, pages 57–81. Springer-Verlag, 1998. Proceedings from the *5th International Static Analysis Symposium*.

[40] M. Hind and A. Pioli. Which pointer analysis should I use? In *International Symposium on Software Testing and Analysis*, Aug. 2000.

[41] M. Hind and A. Pioli. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming*, 39(1):31–55, Jan. 2001.

[42] S. Horwitz. Precise flow-insensitive may-alias analysis is NP-Hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, Jan. 1997.

[43] S. Horwitz, P. Pfeiffer, and T. W. Reps. Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 28–40, 21–23 June. *SIGPLAN Notices* 24(7), July 1989.

[44] P. Hudak. A semantic model of reference counting and its abstraction. In *Conference Record of the 1986 ACM Symposium of LISP and Functional Programming*, pages 351–363, Aug. 1986.

[45] J. Hummel, L. J. Hendren, and A. Nicolau. Abstract description of pointer data structures: An approach for improving the analysis and optimization of imperative programs. *ACM Letters on Programming Languages and Systems*, 1(3):243–260, Sept. 1992.

[46] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *28th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, Jan. 2001.

[47] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, 1981.

[48] W. Landi. *Interprocedural Aliasing in the Presence of Pointers*. PhD thesis, Rutgers University, 1992. LCSR-TR-174.

[49] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, Dec. 1992.

[50] W. Landi and B. Ryder. Pointer-induced aliasing: A problem classification. In *18th Annual ACM Symposium on the Principles of Programming Languages*, pages 93–108, Jan. 1991.

[51] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 235–248, 17–19 June. *SIGPLAN Notices* 27(7), July 1992.

[52] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 56–67, 23–25 June. *SIGPLAN Notices* 28(6), June 1993.

[53] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 21–34, 22–24 June. *SIGPLAN Notices* 23(7), July 1988.

[54] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing and Analysis*, Aug. 2000.

[55] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In O. Nierstrasz and M. Lemoine, editors, *Lecture Notes in Computer Science, 1687*, pages 199–215. Springer-Verlag, Sept. 1999. Proceedings of the *7th European Software Engineering Confrence* and *ACM SIGSOFT Foundations of Software Engineering*.

[56] D. Liang and M. J. Harrold. Equivalence analysis: A general technique to improve the efficiency of data-flow analyses in the presence of pointers. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Sept. 1999.

[57] D. Liang and M. J. Harrold. Reuse-driven interprocedural slicing in the presence of pointers and recursion. In *1999 International Conference on Software Maintenance*, Sept. 1999.

[58] D. Liang and M. J. Harrold. Towards efficient and accurate program analysis using light-weight context recovery. In *Twenty-second International Conference on Software Engineering*, June 2000.

[59] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Seventh International Static Analysis Symposium*, July 2001.

[60] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analysis for Java. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, June 2001.

[61] T. Marlowe, W. Landi, B. Ryder, J.-D. Choi, M. Burke, and P. Carini. Pointer-induced aliasing: A clarification. *SIGPLAN Notices*, 28(9):67–70, Sept. 1993.

[62] M. Mock, M. Das, C. Chambers, and S. Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, June 2001.

[63] J. D. Morgenthaler. *Static Analysis for a Software Transformation Tool*. PhD thesis, University of California at San Diego, Aug. 1997.

[64] R. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie Mellon University, 2001. CMU-CS-01-124.

[65] J. Palsberg. Equality-based flow analysis versus recursive types. *ACM Transactions on Programming Languages and Systems*, 20(6):1251–1264, Sept. 1998.

[66] H. D. Pande. *Compile Time Analysis of C and C++ Systems*. PhD thesis, Rutgers University, May 1996.

[67] H. D. Pande and B. G. Ryder. Static type determination for C++. In *Proceedings of the Sixth Usenix C++ Conference*, 1994.

[68] J. Plevyak and A. A. Chien. Precise concrete type inference for object oriented languages. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–340, Oct. 1994.

[69] PROLANGS Analysis Framework. http://www.prolangs.rutgers.edu.

[70] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, Sept. 1994.

[71] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *26th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, Jan. 1999.

[72] J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *28th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, Jan. 2001.

[73] T. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 163–196. Kluwer Academic Publishers, Mar. 1994.

[74] T. Reps. Shape analysis as a generalized path problem. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, June 1995.

[75] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11–12):701–726, Nov. 1998.

[76] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pages 47–56, 18–21 June. *SIGPLAN Notices* 35(5), June 2000.

[77] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java based on annotated constraints. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2001.

[78] A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *International Conference on Compiler Construction*, Apr. 2001.

[79] A. Rountev, B. G. Ryder, and W. Landi. Data-flow analysis of program fragments. In O. Nierstrasz and M. Lemoine, editors, *Lecture Notes in Computer Science, 1687*, pages 235–252. Springer-Verlag, Sept. 1999. Proceedings of the *7th European Software Engineering Confrence* and *ACM SIGSOFT Foundations of Software Engineering*.

[80] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 13–22, 18–21 June. *SIGPLAN Notices* 30(6), June 1995.

[81] E. Ruf. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pages 208–218, 18–21 June. *SIGPLAN Notices* 35(5), June 2000.

[82] E. Ruf. Partitioning dataflow analyses using types. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 15–26, Jan. 1997.

[83] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *15th Annual ACM Symposium on the Principles of Programming Languages*, pages 285–293, Jan. 1988.

[84] R. Rugina and M. C. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, pages 77–90, 1–4 May. *SIGPLAN Notices* 34(5), May 1999.

[85] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pages 182–195, 18–21 June. *SIGPLAN Notices* 35(5), June 2000.

[86] B. G. Ryder, W. A. Landi, P. A. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems*, 23(2), Mar. 2001.

[87] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 16–31, Jan. 1996.

[88] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, Jan. 1998.

[89] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *26th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, Jan. 1999.

[90] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In P. V. Hentenryck, editor, *Lecture Notes in Computer Science, 1302*, pages 16–34. Springer-Verlag, 1997. Proceedings from the *4th International Static Analysis Symposium*.

[91] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive point-to analysis. In *24th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 1–14, Jan. 1997.

[92] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, 1981.

[93] O. Shivers. *Control-Flow Analysis of High-Order Languages*. PhD thesis, Carnegie-Mellon University, 1991. CMU-CS-91-145.

[94] B. Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, Jan. 1997.

[95] P. A. Stocks, B. G. Ryder, W. A. Landi, and S. Zhang. Comparing flow and context sensitivity on the modifications-side-effects problem. In *International Symposium on Software Testing and Analysis*, pages 21–31, Mar. 1998.

[96] M. Streckenback and G. Snelting. Points-to for Java: A general framework and an empirical comparison. Technical report, University of Passau, Nov. 2000.

[97] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *27th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, Jan. 2000.

[98] F. Vivien and M. C. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, 20–22 June. To appear.

[99] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, July 2000.

[100] D. Walker and G. Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, Sept. 2000.

[101] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, Jan. 1980.

[102] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 1999.

[103] R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *International Conference on Compiler Construction*, Mar. 2000.

[104] R. P. Wilson. *Efficient Context-Sensitive Pointer Analysis for C Programs*. PhD thesis, Stanford University, Dec. 1997.

[105] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 1–12, 18–21 June. *SIGPLAN Notices* 30(6), June 1995.

[106] S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, pages 91–103, 1–4 May. *SIGPLAN Notices* 34(5), May 1999.

[107] J.-S. Yur, B. G. Ryder, and W. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *Twenty-first International Conference on Software Engineering*, May 1999.

[108] S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *4th Symposium on the Foundations of Software Engineering*, pages 81–92, Oct. 1996.

[109] S. Zhang, B. G. Ryder, and W. Landi. Experiments with combined analysis for pointer aliasing. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 11–18, June 1998.

[110] X.-X. S. Zhang. *Practical Pointer Aliasing Analyses for C*. PhD thesis, Rutgers University, Aug. 1998.