How (Not) to Write Java Pointer Analyses after 2020

Manas Thakur
IIT Mandi
India
manasthakur@iitmandi.ac.in

Abstract

Despite being a very old discipline, pointer analysis still attracts several research papers every year in premier programming language venues. While a major goal of contemporary pointer analysis research is to improve its efficiency without sacrificing precision, we also see works that introduce novel ways of solving the problem itself. What does this mean? Research in this area is not going to die soon.

I too have been writing pointer analyses of various kinds, specially for object-oriented languages such as Java. While some standard ways of writing such analyses are clear, I have realized that there are an umpteen number of nooks and pitfalls that make the task difficult and error prone. In particular, there are several misconceptions and undocumented practices, being aware of which would save significant research time. On the other hand, there are lessons from my own research that might go a long way in writing correct, precise and efficient pointer analyses, faster. This paper summarizes some such learnings, with a hope to help readers beat the state-of-the-art in (Java) pointer analysis, as they move into their research careers beyond 2020.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Compilers**; Object oriented languages.

Keywords: Pointer analysis, Java, Correctness, Efficiency

ACM Reference Format:
Manas Thakur. 2020. How (Not) to Write Java Pointer Analyses after 2020. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '20), November 18–20, 2020, Virtual, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3426428.3426923

Onward! '20, November 18–20, 2020, Virtual, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8178-9/20/11…$15.00
https://doi.org/10.1145/3426428.3426923

1 Introduction

At the dawn of the twenty-first century, Michael Hind asked if we weren't done with solving the problems associated with pointer analysis yet [12]. He concluded that pointer analysis, though a problem very intensively worked upon in the previous century, was still an active research area with several challenges lined up to be solved, specially for object-oriented languages such as Java. Twenty years from then, pointer analysis still finds a decent place in the proceedings of premier programming language conferences and journals, with the kinds of problems pointed out by Hind still being addressed, and novel ways of solving the same actively being invented. The central reason behind the continued interest is two-fold: (i) applications of precise pointer-analysis results have continued to expand, even more so in parallel programs; whereas (ii) precise pointer analysis continues to be unscalable for large applications, even in the serial world. Before we dive in to understand the motivation behind this paper, a few words on these points, from the perspective of new researchers, deserve attention.

First, why should one care about pointer analyses? For a program involving memory allocation on the heap, *points-to analysis* tells which objects on the heap may be pointed to by which variables or field references at run-time. The resultant points-to relationships are important enablers for various other analyses and optimizations (such as alias analysis, escape analysis, virtual-call resolution, and so on). In other words, if you want to perform almost any analysis or optimization on programs with dynamic memory allocation, there is a high chance the efficacy of the same would depend heavily on the precision of the underlying pointer analysis. The problem becomes more interesting for languages like Java, where idiomatic program design focuses centrally on defining classes and instantiating objects on the heap.

Second, what is the problem with performing precise pointer analyses? The answer to this boils down to the *amount* of points-to relationships that need to be created and maintained during the analysis in order to more precisely model the actual points-to relationships at run-time. Obviously, at run-time, each variable (and field reference) will point to exactly one object on the heap. However, as static analysis can only approximate the run-time, the precision varies across several dimensions such as field-sensitivity, flow-sensitivity, context-sensitivity, and path-sensitivity. Out of these, for Java, field-sensitivity is considered essential for usefulness of pointer-analysis results; path-sensitivity is exponential and usually not given much attention; context-sensitivity is very useful but has several variations most of which do not scale well; and flow-sensitivity interacts differently with different context-sensitivity variations. Researchers in this area have been paying special attention to scaling context-sensitivity, and as we will see in this paper, some exciting results, specially for particular kinds of pointer analyses, have started peeping in just very recently.

I started studying and implementing Java pointer analyses in 2014, as part of my PhD thesis. The intent was to improve the precision of the analyses performed by just-in-time (JIT) compilers (where resource availability is stringent), without increasing the time spent during JIT compilation. The result was the development of the PYE framework [41] that mitigated inefficiency by offloading complex analyses to the static compiler, and maintained precision by generating partial results in form of *conditional values*. Meanwhile, I encountered interesting challenges and insights related to context-sensitive pointer analyses, which led to the development of some novel abstractions [40, 42] for context-sensitivity.

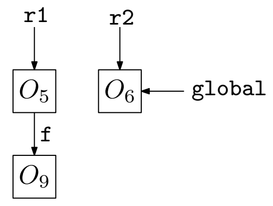
Though the exercises I did over the last few years were quite fruitful, the path was filled with difficulties of various kinds. First, there were misconceptions, either because of the corresponding subtleties not being explicit in the literature, or because of differences between learning about pointer analyses using toy languages versus implementing them for a full programming language. Second, it was often the case that the idea was in place, the first shot over its implementation was done, but the analysis failed miserably to scale when performed over large real-world benchmarks. Third, there were issues resulting either from certain tricky language features or from the state-of-the-art tools available for writing pointer analysis, which posed threats to the correctness of the generated results.

The premise of this paper is that the life of future pointer-analysis researchers would be much easier if the kinds of issues highlighted above, and their *acceptable* solutions, were known beforehand. To provide a redressal, this paper highlights pertinent issues under the three categories discussed above, with possible available solutions, and the challenges still remaining to be solved. The examples in the exposition are taken from Java, but the concepts are applicable to any language involving object allocation on a heap.

The rest of the paper is organized as follows. Section 2 gives an overview of some preliminary concepts and terms used through the later sections. Section 3 describes several misconceptions that plague the mind when a student moves from learning about pointer analysis from a classroom setting to implementing them for real-world programming languages. Section 4 extracts certain key ideas from recent advancements aimed at imparting efficiency, that is, reducing the resources consumed, while maintaining or even improving the precision. Section 5 highlights key challenges in ensuring and arguing correctness in presence of certain language features, distinguishing between acceptable ways to get the ideas published and taking the ideas to production. Finally, Section 6 discusses relevant related work and Section 7 concludes the paper by illuminating some of the ways to pursue pointer analysis research in future.

```
 1 class B {B f;}
 2 class C {
 3  static B global;
 4  void foo() {
 5    B r1 = new B();
 6    B r2 = new B();
 7    synchronized(r2) {...}
 8    global = r2;
 9    r1.f = new B();
10  } }
```

(a)



(b)

Figure 1. (a) A Java code snippet. (b) Points-to graph after analyzing the method foo.

2 Background

This section presents some preliminary concepts, terms and notations that are used throughout the paper.

2.1 Points-to Analysis

Points-to analysis is a static program-analysis technique that establishes which pointers, or reference variables, may point to which objects or storage locations, at runtime. The results obtained by points-to analysis are key to several other heap analyses and related optimizations; for example, alias analysis, shape analysis, escape analysis, null-check elimination, call-graph construction, method inlining, and so on.

We say that the points-to set of a variable *var* is S, if the elements of the set S represent the objects that may be pointed-to by the variable during program execution. One of the ways of statically representing all the run-time objects allocated at line l is using an abstract object denoted as O_l. Using this notation, for the code shown in Figure 1a, the *may-points-to* sets of the reference variables r1 and r2 are $\{O_5\}$ and $\{O_6\}$, respectively.

2.2 Points-to Graphs

Points-to graphs and their variations are widely used [37, 39, 40, 44] for representing the points-to relations in Java programs. A points-to graph $G(N, E)$ comprises of (i) a set N of nodes that represent variables and abstract objects in the program; and (ii) a set E of edges that represent points-to relationships among the nodes in the program. An edge can optionally have a label representing the field in the corresponding points-to relationship. For example, Figure 1b shows the points-to graph for the code shown in Figure 1a.

The graph denotes that the variables `r1`, `r2` and `global` point to the sets $\{O_5\}$, $\{O_6\}$ and $\{O_6\}$, respectively, and that the field `f` of the object O_5 points to the set $\{O_9\}$. The reader may like to go through one of the standard texts describing points-to and similar graphs [7, 44] to understand the way graphs such as these are constructed for given Java programs.

2.3 Modular Analysis

Modular analysis, as proposed by Cousot and Cousot [8], is a well-explored technique to scale interprocedural analyses. Under this approach, different modules of a program are analyzed separately, and the modular results are composed to obtain the results for the whole program [6, 7, 37, 44]. A recent survey by Madhavan et al. [23] evaluates several modular analysis techniques in a well-formalized framework. In the context of Java pointer analysis, modular analysis typically implies analyzing each method in the program separately (possibly resulting into a per method points-to graph), and then merging the results at method call-sites to form interprocedural analysis results.

2.4 JIT Compilation

The programs written in some of the modern programming languages (such as Java and C#) are translated to machine code in two stages: the program is first statically compiled to a platform-independent intermediate code (such as Bytecode for Java), which might then get translated to native code on a possibly different machine. The second step, which often happens when the program is already being interpreted, is called just-in-time (JIT) compilation, and is a common way to optimize programs in virtual machines [10, 28, 30].

A particular challenge with JIT compilation is that as the time spent in program analysis gets added to the execution time of the program being compiled, the trade-off between an analysis's precision and its efficiency is much more pronounced than in static compilers. A recent technique to balance this trade-off was proposed as part of the PYE framework [41], where complex analysis of Java applications is performed in the static Java compiler using Soot [43], and the (partial) conditional results generated for the application and the libraries (analyzed offline) are merged during JIT compilation in the OpenJDK HotSpot JVM.

3 Clearing Misconceptions

Pointer analysis for modern object-oriented languages such as Java and C++ involves handling several different statements and possible corner cases. Most new analysis writers start with implementing prototypes, often for subsets of these languages, in popular tools such as LLVM [16], Soot [43] and DOOP [5], before actually implementing their research ideas for real-world benchmark programs written using the complete (superset) language. Consequently, new analysis writers are often challenged by surprises and unexpected scenarios that sometimes are detected only in the later stages of their implementations. This section discusses three such misconceptions usually resulting from working with prototype languages and programs, in the context of performing pointer (or heap) analyses for Java-like languages.

3.1 There Ain't Just Four Statements

Courses on program analysis and compiler design usually follow a very pedagogic approach for introducing students to writing program analyses. They define a subset programming language and provide students with skeleton programs wherein students are asked to fill in the gaps as part of programming assignments. These assignments are often a *case analysis* over the different kinds of statements that may occur in programs written in the subset language. In my experience, as a reminiscent of standard Andersen's pointer analysis [1], students are often taught that generating constraints for a pointer analysis involves handling four kinds of statements on variables of pointer types: (i) reference assignment (`a = &b` or `a = new A()`); (ii) copy (`a = b`); (iii) load (`a = *b` or `a = b.f`); and (iv) store (`*a = b` or `a.f = b`).

Once students encounter programs written in real languages, it is imperative that they soon realize that the number of statements they need to handle is nowhere near four, and it varies with the kind and scope of the analysis they intend to write. For example, for writing an interprocedural thread-escape analysis for Java [3], the minimum number of statements that need to be handled are at least nine: reference assignment, copy, load, store, and (v) null assignment (`a = null`); (vi) exception throw and handler (`throw a` and `catch(a)`; (vii) method call (`a.foo(b)`); (viii) method entry (`foo(a,b)`), including the handling of the zeroth parameter (`this`) for virtual methods; and (ix) return statement (`return a`). Additionally, if the analysis intends to take special care of flow-sensitivity for synchronization elision [41] (an application of thread-escape analysis), then a tenth statement needs to be considered: (x) synchronization (`synchronized(a)`). A timely awareness of the fact that real pointer analyses are much more complex to write than their programming assignments would not only help students better estimate the amount of time required (and hence to plan their studies), but also enhance the confidence and subsequent arguments about correctness of their algorithms.

3.2 Graphs Need Not Have Nodes and Edges

As mentioned in Section 2.2, points-to graphs are a common way to implement different kinds of pointer and heap analyses. As also mentioned in Section 2.2 and in various research articles [7, 44] on the subject, a points-to graph is essentially a graph $G(N, E)$ comprising of a set N of nodes and another set E of edges. Further, for points-to graphs constructed for programs written in Java-like languages, there can be two kinds of nodes – those representing variables

```java
interface Node {...}
interface Edge {
  Node src; Node dst; ...
}

class VariableNode implements Node {...}
class ObjectNode implements Node {...}

class StackEdge implements Edge {...}
class HeapEdge implements Edge {
  Field f; ...
}

class PointsToGraph {
  Set<Node> nodes; Set<Edge> edges; ...
}
```

Figure 2. A possible design of a points-to graph.

(say N_V) and those representing (abstract) objects (say N_O); and two kinds of edges – those representing points-to relationships originating from the stack (variables to objects; say E_S) and those representing points-to relationships from the heap (objects to objects via fields; say E_H). Consequent to such descriptions in literature, Figure 2 shows the most straightforward way to implement points-to graphs, say in Java. Simple it may look, but the design shown in Figure 2 is highly inefficient. There are two primary culprits for this inefficiency, as discussed next.

The first reason behind the inefficiency of the design in Figure 2 is the need to maintain consistency between the abstract objects stored in the set `nodes` and as source `src` and destination `dst` of the points-to relationships stored in `edges`. In order to elucidate this point, consider the following question: "When should a node be a part of a points-to graph?" Say the statement being analyzed is s: $a = b$, as a result of which a variable edge should be added from a to all the objects pointed to by b in the points-to graph before s. Say the analysis being performed is flow-sensitive. At this point, if there were edges resulting from a before s, they may be removed from the graph after s. If a removed edge was the only edge to an object o, should o be a part of the points-to graph after s or not? On the other hand, if points-to graphs did not maintain an explicit set of nodes, removing redundant objects would not be required as an explicit operation at all. However, note that this choice should depend on the analysis being performed. For example, an analysis written to garbage collect unreachable objects on the heap may want to remove o after s, but an analysis that prints some information about the objects allocated in a method (say *escape status* [3, 41]) might want to keep o in the set of nodes till the call to the print routine.

The second reason behind the inefficiency of the design in Figure 2 becomes apparent when one needs to iterate over the objects reachable from a given node in the points-to graph. For several kinds of statements (such as $a = b$, $a = b.f$ and $a.f = b$), the points-to graph needs to be updated by iterating over the set of objects pointed to by the variable or the field reference on the right side of a statement (such as b or $b.f$), and adding edges from the variable or the field reference on the left side of a statement (such as a or $a.f$). For such operations, it is almost always necessary to collect the points-to set by iterating over the outgoing edges from the node corresponding to a given source (variable or object). This requires two look-up iterations: first for the node corresponding to the source, and then for the points-to set of the resultant node. It would save a significant amount of analysis time if the points-to graph was simply a map from nodes to a set of another nodes, denoting their points-to sets. However, this is still easier said than done: there would be a need to store an additional label (representing the field) for edges on the heap (that is, in E_H), which would require further engineering to denote the mapping from keys (nodes) of the map to the objects in the points-to set.

3.3 Not All of Them Are Flow-sensitive

Another issue faced by new analysis writers while taking ahead their literature review arises from misunderstanding the need of flow-sensitivity. The standard way to model flow-sensitivity [25] is by using a worklist-based iterative dataflow analysis (IDFA), wherein the flow sets (in our case, points-to graphs) are maintained and updated as IN and OUT sets before and after each program point (in our case, statement). Typically, novices try to fit each analysis they write into the IDFA approach. The confusion surmounts when papers (citations skipped to avoid incompleteness) do not specify whether the analysis proposed therein is flow-sensitive or insensitive and assume the reader to understand this trivially. This distinction is very important from an implementation perspective, as there is a huge difference in the scalability of flow-sensitive and flow-insensitive analyses [11].

Apart from the perspective of scalability, several problems do not require flow-sensitivity by definition – popular examples being object- and type-sensitive pointer analyses [24, 35], and some of the pre-analyses in various recent works that use the idea of *staging* (discussed in detail in Section 4.3). For example, Thakur and Nandivada [40] estimate the required amount of value contexts [14, 29] (points-to graphs reaching the entry points of methods) by computing the depth of the subgraphs reachable from each parameter of a method, in a pre-analysis; this information is independent of the flow and does not require performing an expensive iterative dataflow analysis. The key takeaway is that analysis writers should proactively judge if their ideas need flow-sensitive information, and if not, then save enormous amounts of time in scaling and proving the correctness of flow-sensitivity in their analyses.

4 Imparting Efficiency

This section is dedicated to the real challenge that drives pointer analysis research: scalability of precise pointer analyses over large real-world programs. As pointer analysis enables a large number of optimizations and even other analyses, enhancements in its precision along any analysis dimension (such flow-sensitivity, field-sensitivity and context-sensitivity) are important. Out of these possible precision enhancements, it has been shown that context-sensitivity plays a significant role in improving the precision of the results generated for object-oriented programs such as those in Java [18]. Consequently, over the last few years, much of the research in pointer analysis has been focused upon improving the scalability of context-sensitive analyses, while looking for novel ways to balance the trade-offs with their precision. This section extracts five key principles from some recent advancements in scaling precise context-sensitive analyses, with the suggestion that more work in some of these directions should form the road ahead.

4.1 Context Abstraction Matters a Lot

A context-insensitive analysis generates a single summary information for a method, which is (soundly) applicable across all contexts in which the method is called, thus merging the dataflow values among different call-return pairs. On the other hand, a context-sensitive analysis specializes the analysis information for a method for each context from which it is called, thus quite often enhancing the precision. However, the precision and the scalability vary with "what constitutes the context", termed as the *context abstraction* of a context-sensitive analysis. This section gives an overview of the common choices available, along with the scenarios in which each of them tends to generate better results.

Popular context abstractions can be divided into two broad categories: those based on object-sensitivity [24] and those based on call-strings [32]. Object-sensitivity distinguishes contexts based on the receiver object on which a method is called, and is particularly suited in cases where most methods modify or access the state of the receiver. On the other hand, object-sensitivity fails to distinguish the contexts of class (or *static*) methods (as they have the same receiver every time they are called – the object corresponding to the class to which they belong), and hence are not suited for the same. Similarly, object-sensitive analyses are not suited for methods that modify or access the state of parameters other than the receiver. However, in object-oriented languages, as it is more common to write methods that operate on the receiver, cases such as these are typically less frequent.

There are analyses for which rather than the actual receiver objects, only the precise type of receiver objects yields good enough precision. Examples include pointer analyses for resolving virtual calls (thus enabling method inlining), for constructing precise call-graphs, and so on. For such analyses, instead of object-sensitivity, another context abstraction called type-sensitivity [35] is a good candidate for generating context-sensitive results. Type-sensitive analyses scale better than their object-sensitive counterparts, and hence could be the pick for type-based analyses and optimizations.

The classical call-strings approach [33] distinguishes contexts based on the string formed by the calling sequence of a method. As call strings are an approximation over the actual run-time call stack, they can be easily mapped to program execution. However, the length of call-strings grows exponentially in presence of recursive calls, and their bounded (*k*-limited) versions usually do not scale well for the achievable precision. Khedker and Karkare [14] reduce the number of contexts in the call-strings approach by merging call-strings in which the same dataflow value reaches the entry of a method (terming the resultant abstraction *value contexts*). Though quite promising in terms of maintaining the full precision of call-strings, value contexts do not scale well if the dataflow values tend to be very large. This is typically true for pointer analyses, as points-to graphs (the value contexts) usually consist of thousands of nodes and edges, making merging contexts by checking their similarity a very expensive operation. Section 4.2 discusses a related alternative context abstraction that works particularly well for certain kinds of pointer analyses.

The takeaway from this section is that there are various choices available to choose as a context abstraction, and each of them suits particular cases. Hence, the recommendation in 2020 is to avoid getting into the lure of using the same context abstraction for the complete program (or for all the programs in the benchmark suite under consideration). It is better to profile or estimate the needs of the different kinds of methods (such as static and non-static) in a given program, and use an appropriate approach based on the computed information. Section 4.3 discusses a promising way to compute information that may help in deciding which context abstraction to apply for a given method in a program. For a more comprehensive discussion on the relative precisions of various context abstractions from Java program-analysis literature, the reader is referred to a recent work by Thakur and Nandivada [42].

4.2 Generalization Is Not Always the Best Thing

Given the wide applicability of pointer analysis, the research in this space has been centered around coming up with ways to scale the computation of general *points-to* information, irrespective of the application for which the results may be employed. Observing that in spite of continuous advancements triggered by this generalization, we still do not have precise, specially context-sensitive, pointer analyses that could be used in time- and memory-critical systems (such as JIT compilers), I argue that while moving forward, we should work towards designing analysis- or application-specific abstractions. In particular, for scaling context-sensitivity, Thakur and Nandivada [40, 42] propose several variants of novel analysis-specific context abstractions, and also use them as part of scaling precise analyses for JIT compilers [41].

A particular example of analysis-specific contexts, termed *level-summarized relevant value* (or *LSRV*) contexts, was recently [40] used to obtain highly scalable variants of the call-strings approach for performing thread-escape and control-flow analyses on Java programs. LSRV contexts have two underlying ideas concerning analyses involving points-to graphs: (i) approximate how much of a points-to graph reaching a given method is actually accessed or modified by the method; and (ii) instead of using actual nodes of the points-to graph, project the relevant nodes to form another graph that uses elements from the lattice of the optimization intended to be performed using the computed pointer-analysis results. For example, after performing the first step to filter out the irrelevant portions of a points-to graph reaching a method, if the application depends only on the escape status of objects, then the LSRV context would be a graph containing only the escape statuses of the corresponding objects from the points-to graph. The advantage of this approach is that the graphs after projecting the relevant information tend to be significantly smaller than original points-to graphs and also lead to the merging of a higher number of value contexts (if the lattice used for projection is smaller than the lattice of points-to graphs, which is quite often the case).

The catch associated with using analysis- or application-specific pointer analyses is obvious: the results may no longer be equally useful for performing different dependent analyses and optimizations. However, given the scalability advantages, it would be prudent to channelize future research towards increasing the reuse of information computed by using multiple analysis-specific abstractions.

4.3 Staging: The New Buzzword

This section discusses a very effective strategy for imparting efficiency without sacrificing, and often enhancing, the precision of pointer analysis: staging. The idea of staging is to break the analysis into multiple phases, such that the complete analysis information is available after the last phase. An effective approach to staging an analysis is to first perform a pre-analysis that gathers information about the various parts of a program, and to subsequently use that information to scale the main analysis. In order to avoid inheriting the problems of the expensive main analyses, such pre-analyses often need to be insensitive across various analysis dimensions. Interestingly, fast pre-analyses have been found to be capable of gaining strong enough insights about the program that the main analysis can often be scaled quite astonishingly utilizing the same. Multiple ways of staging an analysis have been adopted over the last few years in the pointer analysis community, some of which are discussed next.

The first class of pre-analyses includes those that are aimed at filtering the information that is not required for maintaining precision. For example, while performing k-limited object-sensitive and call-strings based analyses, a pre-analysis can be used to estimate the depth of the context (chain of receivers or call-sites) required to maintain precision for a given method [27, 38]. Similarly, it is also possible to estimate which all constituents of a value- or an object-sensitive context can be dropped by estimating relevance of the context for a given method [39, 40]. The insight behind these approaches is that the reason a particular sensitivity leads to enhanced precision can be mapped to certain patterns in a given program, and those patterns can be *approximated* while being insensitive in that particular dimension, that is, without incurring the overheads of the problem being tried to be solved. An explicit example of this insight was used by Li et al. [19] to identify methods that are part of the object-flow paths where information would get merged if those methods were analyzed context-insensitively. Thus, the methods identified to maintain precision in the pre-analysis are analyzed context-sensitively, whereas the remaining methods can be analyzed context-insensitively without incurring much loss of precision (the loss depends on the precision of the pre-analysis).

A second class of pre-analyses includes those that target scalability, with or without losing precision. For example, pre-analyses have been used to identify methods analyzing which precisely may thwart the scalability of a given analysis [20], and then to analyze those methods conservatively. On the other hand, sometimes simply postponing the analysis of certain methods to a separate pass enhances the scalability, an example being the *post analysis* of LSRV contexts [40]. The insight there interestingly relates to the need for *garbage collection* in languages with an automatically managed heap (such as Lisp, Java and C#). In particular, as an analysis stage keeps running for long without releasing resources (such as accumulated points-to graphs), memory consumption keeps increasing and the time required for standard operations (such as getting the points-to graph of a method from the set of graphs maintained for all the methods in all the contexts) keeps increasing. Thus, when it can be identified that postponing the analysis of a method would not lead to loss of precision, it helps to simply skip it in the current pass and analyze it afresh on a clean(er) slate.

A promising future direction in writing staged analyses would be to use them in forming interesting amalgamations of different abstractions. For example, the author of this paper would be interested in leveraging the idea of staging to reuse information that is computed using different analysis-specific abstractions for different parts of a program (that is, using the concepts presented in Sections 4.1, 4.2 and 4.3 together!). Readers are encouraged to contact the author in case of further (or more complex) ideas on the subject.

```
1  class D {
2    D g;
3  }
4  class C {
5    static D f;
6    void foo() {
7      D x = bar();
8      D y = f.g;
9      /* Q1: Will f.g throw a NullPointerException? */
10     /* Q2: Are x and y aliases? */
11   }
12 }
```

Figure 3. Example illustrating some complexities while performing pointer analysis in presence of static fields.

4.4 To Static or Not to Static

Dataflow values can reach, and be modified by, a method in two ways: either via its parameters or via global variables (static fields in Java), the latter deserving special attention. To illustrate the kinds of problems static fields may introduce, consider the code snippet shown in Figure 3. The class `C` includes a static field `f`, which is by default `null`. Does it remain `null` and hence result into a `NullPointerException` at line 8? If there is no exception thrown and the execution proceeds, can `x` and `y` potentially alias after line 8? None of these questions can be answered precisely just by looking at `foo` or even `bar`, and require the availability of the whole potential points-to graph reachable from `f` (which can be modified in any method that can access `f`). To make situations worse, effects of `f` possibly being accessed simultaneously by multiple threads need to be factored in (as objects reachable from static fields may be shared by threads). Handling the portion of the heap reachable from static fields thus involves performing some kind of escape analysis, which in turn depends on the results of a precise pointer analysis, thus introducing the second notoriously famous cycle in pointer analysis (the first being the interplay between pointer analysis and call-graph construction).

A trivial and practically employed way to handle static fields is to handle them naively, whereby anything is assumed to be reachable from all the static fields of a program – though this sounds very conservative, it might often work well in practice [40] (depending on the amount of useful information maintained using static fields). Another extreme is to analyze the program in a top-down manner, while taking care of the possible execution orders, ensuring that the points-to graph reachable from static fields is always an over-approximation of the actual possible heap. An example of this approach appears in the implementation of value contexts in VASCO [29], which is also a good place to look at an attempt that tries to update the heap reachable from static fields in an efficient manner.

```
1  class ArrayList<E> implements List<E> {
2    Object[] arr;
3    public ArrayList() {
4      arr = new Object[X]; // X: some default initial size
5    }
6    public E get(int i) { return arr[i]; }
7    public void add(E e, int i) {
8      // Bounds check and expand routines skipped
9      arr[i] = e;
10   }
11 }
```

Figure 4. Simplified structure of `java.util.ArrayList`.

4.5 You Enter the Library, You Stay in The Library

Let us conclude the discussion on efficiency with an often problematic source of inefficiency in analyzing even small Java programs: the JDK library. It is almost impossible to write a Java program that does not call methods from the JDK library (definitely impossible if the call to the constructor of `java.lang.Object` is considered). Further, because of a heavy use of inheritance in the JDK libraries, various works report having had to treat them specially [36, 40], lest their analyses would not terminate in a reasonable amount of time. In order to understand the reasons behind the inefficiency while performing such *whole-program* pointer analyses over Java programs, consider the typical structure of JDK collection classes illustrated by `java.util.ArrayList` as shown in Figure 4. The `ArrayList` is inherently implemented using an array, which is initialized with some default size and expanded when required. Not tuning a pointer analysis to handle such classes properly leads to two kinds of problems, as discussed next.

First, as an `ArrayList` is only one of the possible implementations of a `List`, if the call-graph for a given program is constructed using an algorithm that does not consider the dynamic type of objects (such as class-hierarchy analysis [9], which is the default in many tools such as Soot [43]), then subsequent calls to all occurrences of methods such as `add` and `get` would lead to the analysis of all such methods across all the implementations of `java.util.List`, in the JDK as well as in the application. Thus, it would be prudent to use a pointer-analysis based call-graph construction algorithm for analyzing programs containing heavy inheritance in general, and involving JDK libraries in particular.

The second source of inefficiency with the `ArrayList` implementation shown in Figure 4 concerns *heap cloning* [26], which is a technique to specialize (abstract) objects with the context in which they are allocated. For example, consider the array object O_4 allocated in `ArrayList`'s constructor at line 4. If the pointer analysis being performed was context-insensitive, or even if it was context-sensitive but did not employ heap cloning, then all the `ArrayList` objects created throughout the program would have their array field `arr` pointing to the same object O_4. Consequently, say there are two `ArrayLists` `l1` and `l2` containing elements of different types (say E_1 and E_2), the objects added to `l1` and `l2` via the `add` method would get merged, and performing a `get` operation[1] on either of l_1 or l_2 would return both the objects together. Thus, `l1.arr` and `l2.arr` would effectively be aliases (imprecise). Further, say both classes E_1 and E_2 share the inheritance hierarchy and override a method `m`, then the pointer analysis, on encountering a statement `x.m();` where `x` is obtained by calling `get` on any of the two lists, would conclude that the method `m` defined in both E_1 and E_2 may be called, thus again leading to imprecision.

To conclude, the key takeaway from this subsection is that in order to avoid getting your analyses stuck inside JDK libraries, consider analyzing them either very precisely (adopting context-sensitivity with heap cloning), or accept having to treat them specially (say assuming or by defining a set of specifications). In particular, JDK libraries are a typical example of the case where improving the precision of pointer analysis affects its scalability positively.

5 Arguing Correctness

Till now we have seen various misconceptions that slow down the process of writing pointer analyses, in particular for new analysis writers, and various challenges in terms of efficiency faced by even expert analysis writers, along with several insights on how to mitigate the same. Once an analysis scales and delivers reasonable numbers, one is faced with the following pertinent question[2]: "Is my analysis correct?" More than writing a formal proof of correctness for each analysis, it is important that the analysis writers first convince themselves that their analysis would indeed enable the intended optimizations in a correct manner. This section addresses some of the challenges thrown up by contemporary programming languages, and techniques adopted to design efficient pointer analyses. The list is in no sense exhaustive, but specifically important in context of Java pointer analysis as discussed in this paper.

5.1 Did You Handle Them All?

In academic settings and in a research paper, it is typically not expected that implementations *handle* all the nooks and corners of the language under consideration. However, before taking the ideas to production-level environments, it is important to answer if the ideas would *work* correctly in their presence – if yes, then how much would be the impact; if not, then modulo which features. Thus, an important question to be asked by analysis writers, at least to convince themselves, is if the unhandled possibilities of their target language pose a serious threat to their techniques, first with respect to correctness and then with respect to the impact of the numbers obtained in their evaluation.

The first item on the checklist of this section is the various kinds of statements relevant for pointer analysis, as mentioned in Section 3.1. Unless one is working on a subset language (or intermediate representation) proven to be equivalent to the full language under consideration, it is unwise to assume that all the statements that may affect the information computed by the analysis being performed have been handled correctly. An example of the same was discussed in Section 3.1, where one may find a given set of statements to be sufficient for pointer analysis in general, but may have to expand the set for specific analyses or optimizations (such as handling the `synchronized` construct while performing thread-escape analysis). While using tools such as LLVM and Soot, one trivial but effective way to check whether you have handled all kinds of statements is to simply go over the class hierarchy in the respective tool's documentation.

Apart from handling the statements that may affect points-to relationships, another question to ask while analyzing Java programs concerns the handling of calls to non-Java (native) methods. There are two equally attractive ways to address native methods in Java: (i) Assume the worst, that is, the points-to sets of all the objects reachable from the arguments passed to native methods, after the call, are unknown, which is not too bad given that most native methods in Java are related to mathematical computations and hence not very critical for pointer analysis. (ii) Analyze native methods manually (yes, you read it right) and hard-code their handling into the implementation, which also is not too bad if the number of referred native methods is not very high.

For completeness, there is another way of handling uncommon language features: claim that your target programs do not have those features and relax (obviously, this has to be supported with an analysis of the benchmarks under consideration). However, there are features, which though may not be very interesting in terms of spending energy in handling them in static pointer analyses, yet are so widely used in standard Java programs that they cannot be hand waived in contemporary pointer analysis implementations. One such painful feature is the focus of the next section.

5.2 Did You Reflect Enough?

Perhaps the biggest show-stopper while writing static analyses for Java programs is *reflection*. Using reflection, it is possible to pass the name of a class as a string, create an instance of that class, and access/modify the fields or invoke the methods of the obtained object. Figure 5 shows an example of using reflection to call a particular `play` method based on the name of the game passed by the user. The problem with this code is that which `play` method is called cannot be determined statically, thus leading to an imprecision in the

[1]This example assumes that different array indices are treated as the same field, say '`[]`'.

[2]If the writers did not ask such questions to themselves, the reviewers of a decent peer reviewed conference/journal would anyway make sure they are addressed.

```
1  interface Game { void play(); }
2  class Cricket implements Game {
3    void play() { ... }
4  }
5  class Soccer implements Game {
6    void play() { ... }
7  }
8  class M {
9    void m(String c) throws Exception {
10     Class<?> cls = Class.forName(c);
11     Method mtd = cls.getDeclaredMethod("play", null);
12     Object obj = cls.newInstance();
13     mtd.invoke(obj); // obj is the receiver
14   } }
```

Figure 5. Example of a reflective call in Java.

call graph. Furthermore, the name of the method itself may have been taken as a parameter, in which case any method[3] of any class could have been called at line 13.

Reflection turns out to be an even bigger killer for static analyses on real-world Java benchmarks such as DaCapo v9.x [2], where the entry method of a benchmark itself is invoked by passing the name of the benchmark to the DaCapo wrapper `jar`, taken as a command-line argument during runtime. As a result, a sound static analysis would not be able to determine any particular benchmark for analysis, and would analyze all the benchmarks, leading to highly imprecise results (and a very bad scalability). In fact, due to this very reason, several recent works still prefer to use older DaCapo benchmarks (2006 version), which provided a patched `jar` that did not use reflection – sound but not really up-to-date.

A popular way to analyze DaCapo (and in general, any benchmark that uses reflection) with Soot [43] is using the tool TamiFlex [4]. Running a benchmark with TamiFlex creates a log file listing the actual method called from each reflective call-site. These logs are then used by Soot to complete the call-graph for interprocedural static analysis[4]. The problem with using TamiFlex is apparent: the log is correct only for a particular run of the benchmark; hence if the benchmark was driven by events that could lead to different outcomes, the analysis results would be unsound. Recently, there have been studies and improvements [21, 31] suggesting ways to distinguish the sound parts of an analysis from the potentially unsound ones due to reflection, but taming this dynamic feature to write complete and sound pointer (and even other) analyses continues to haunt the static analysis community, with no fool-proof solution in sight.

[3]An improvement could be done by observing that the parameter list used to get the method at line 11 is `null`, indicating that the method does not take any parameters.

[4]In fact, if you analyze a benchmark that uses reflection with Soot but without TamiFlex, the call-graph will have missing edges, thus leading to an unsound as well as incomplete analysis.

```
1  class Y {...}
2  class X {
3    Y g;
4    X() { g = new Y(); }
5  }
6  class N {
7    void n() {
8      X s = new X();
9      new L().l(s);
10     synchronized(s.g) {...}
11   } }
```

```
11 class L {
12   void l(Object t) {
13     // doesn't access t.g
14   } }
```

Figure 6. A Java code snippet to demonstrate static+JIT analysis. The library class `L` is unavailable while analyzing the application classes `X`, `Y` and `N`, and vice-versa.

5.3 Static+JIT is Not the Same as Modular

Pointer analysis in JIT compilers is very different from that in static compilers. As static analyses are performed offline, it is perfectly fine for them to take several hours and to consume several gigabytes of memory, before generating results that can be used to perform optimizations. However, in JIT compilers, as the time spent in analysis affects the execution time of the program, and given the already present scalability challenges with pointer analyses, most such analyses in typical JIT compilers [10, 28, 30] are performed very conservatively (for example, intraprocedurally), thus neglecting the use of almost all the advancements from the static analysis community. The recently proposed PYE framework [41] proposes to solve this problem by performing the analysis of the different modules of the program statically, while denoting inter-module dependence using conditional values for the elements in the domain of an analysis, and only resolving the conditions during JIT compilation. Albeit a very promising approach for imparting precise analysis results to JIT optimizations, a correctness issue originates from the careful observation that splitting the analysis of the different modules of a program (such as the application and the JDK) across static and JIT compilation is not the same as standard modular analysis followed by composition [7, 44].

Consider the Java program shown in Figure 6. Suppose the goal is to determine if the synchronization operation at line 10 can be elided. Clearly, this requires knowing that the method `l` of the library class `L` does not make the object pointed-to by `t.g` thread-shared. However, typically the code for the library class `L` is not accessible while analyzing the application classes `X`, `Y` and `N`. PYE resolves this issue by statically generating a conditional value representing that the synchronization operation at line 10 can be elided based on the information about the field `f` of the first argument passed to `l`, and resolves this condition in the JIT compiler. Though this looks similar to modular analysis, the problem is that there is a gap between the time such conditional values are generated statically and when they are resolved in the JIT compiler. Consequently, the problems that can be caused by this delay need to be accounted for, before eliding synchronization operations in the JVM. Observe that such a challenge simply does not exist in standard modular analysis: the analysis results are declared to be complete, and hence usable for performing optimizations, only after composing the results of the different modules.

There are two ways to handle the possible security issue discussed above. First, verify that the results indeed correspond to the code that is being linked in the JVM, while handling cases such as dynamic classloading [15] in a possibly imprecise but sound manner. Second, use the results of such static+JIT analyses under guarded conditions. Both these promising ideas are works in progress.

5.4 What to Do When Things Go Wrong?

First of all, how do analysis writers detect whether the results computed by their analyses are wrong? This is a difficult question, and even the not-so-perfect answers depend usually on the analysis being performed. For example, an analysis that elides synchronization could instrument the program at each pre-transformation synchronization point, and check at run-time that a thread other than the current one indeed does not access the corresponding memory location (similar to the approach proposed by Lee et al. [17]). Another option that sometimes helps is to *print* enough information that could be verified after performing transformations; for example, the DaCapo harness [2] pre-computes a checksum on the output and the error streams, and explicitly notifies the user if the checksum computed at runtime is different.

As this paper puts special emphasis on new researchers, the first step to handling wrong analysis results has to be philosophical. Reckon that most software that is out there is buggy, and that there is always a way to move ahead. The next step is to find the source of the bug. As a PhD student, I recall spending days looking at the dumps of my points-to graphs, often containing thousands of nodes and edges, trying to understand where did the analysis go wrong. Most of the times the observable effect was a non-terminating analysis, and the cause turned out to be some piece of implementation that broke monotonicity. This experience helped recently when one of my students faced a similar issue: we step-by-step looked into the handling of cases that might make the information *swing* up-and-down over the underlying lattice and thus break monotonicity (result: we could resolve the issue in two hours).

The process of debugging becomes more involved when working with multi-stage analyses or optimizations; for example, static and JIT as in PYE. Once you find a bug, how do you pin-point whether its source is in the static analysis, in the JIT implementation, or possibly in the way you have integrated the same. The general solution here is to rely on good software design principles: break down the program into modules that compose well. However, in particular for pointer analyses written in Java, ideas such as limiting the number of places where you mutate objects, such as maintaining `final` fields and immutable data structures for (parts of) points-to graphs, helps quite a bit in limiting problems in the first place and in debugging them later, if required.

6 Related Work

The previous sections have illuminated several aspects related to designing and implementing pointer analyses for Java programs: common misconceptions, challenges and ways to impart efficiency, and road-blocks in ensuring and arguing correctness – all based on the author's experience of writing various kinds of pointer analyses over the last few years. This section highlights some of the recent related works that either survey or present ways to address one or more of the issues discussed in the rest of this paper.

Smaragdakis and Balatsouras [34] present a comprehensive discussion of pointer analysis, with a declarative flavour. This approach has quite frequently been used over recent years [19, 20, 36] while implementing various techniques to scale object-sensitive analyses with heap cloning for performing pointer analysis over Java programs. Though the discussion in the current paper has often given examples from analyzing Java programs in Soot [43], many of the points are equally applicable while expressing the analysis in a declarative manner in tools such as DOOP [5].

The importance of the abstraction used in designing an analysis, including that of the context abstraction used in a context-sensitive analysis, heavily affects not only the precision of the analysis but also its scalability to large programs. Kanvar and Khedker [13] present a detailed study of the various choices available while writing such analyses in general, and Thakur and Nandivada [42] evaluate existing and novel choices of context abstractions for Java programs. Many of the insights discussed in Section 4.1 have been drawn from the experiences of implementing the latter.

The importance of soundness while taking the results of static pointer analyses to production, and the absence of soundness in most recent works, in presence of dynamic features such as reflection in Java and other languages, was highlighted recently by Livshits et al. [22]. The authors of that article, while acknowledging that techniques to handle all such cases soundly are not foreseeable in the near future, coin the term *soundiness* to describe such analyses, and suggest that contemporary researchers should enlist the cases modulo which their techniques work correctly. The additional takeaway from the current paper is that in future, more novel modular ways of establishing correctness would be required, specially as we move ahead with designing novel abstractions to scale precise analyses and inventing novel ways of organizing analyses themselves.

7 Conclusion

This paper attempted to cover a vast sphere – precise, scalable and correct pointer analysis of Java programs – based on the experience of someone working towards keeping precise analyses efficient and observing the recent research trends in the community. The aim was not to just showcase the recent advancements in last few years, but to enrich the literature by documenting often missed lessons drawn from the struggles behind implementing and getting good research work published. A particular target audience was the community of new and budding analysis writers, who would take up the evergreen task of improving the state-of-the-art in pointer analysis, specially for object-oriented programs.

The paper presented lessons under three categories: (i) misconceptions arising due to undocumented principles and practices to be kept in mind when advancing from the stage of a "learner" in classrooms to a "researcher" in improving pointer analyses; (ii) the ideas behind the various techniques employed by contemporary researchers in scaling precise, specially context-sensitive, Java pointer analyses; and (iii) the hurdles people face in first themselves getting confident, and then convincing the reviewers and the community, about the validity, the vitality and the reliability of their ideas in presence of quirky language features.

Pointer analysis was an active research area at the dawn of the twenty-first century, and has continued getting traction twenty years down the line. Now as we move onward – beyond 2020 – the key meat of this paper is that in the coming years, we should expect to see more and more work that specializes pointer and program analysis, in general, for the particular problems at hand. The ideas of staging an analysis to gain insights about the program, of splitting the analysis across various phases in a program's life cycle, and of mixing multiple novel abstractions while analyzing the different parts of the same program, would go a long way in improving the overall programming-language ecosystem.

Acknowledgements

The insights presented in this paper are based on years of collaborations and discussions with associates at the author's alma maters and workplaces, professors at various other institutions, researchers at various conferences held throughout the world, and finally, encouraging comments and suggestions from anonymous reviewers. Though it would be impossible to name all these contributors, writing this paper increased the craving for one (more) coffee with them, sitting across the same table, in a Coronavirus-free world.

References

[1] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Ph.D. Dissertation. Cornell University.

[2] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06).* ACM, New York, NY, USA, 169–190.

[3] Bruno Blanchet. 2003. Escape Analysis for JavaTM: Theory and Practice. *ACM Trans. Program. Lang. Syst.* 25, 6 (Nov. 2003), 713–775. DOI: http://dx.doi.org/10.1145/945885.945886

[4] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. https://github.com/secure-software-engineering/tamiflex. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11).* ACM, New York, NY, USA, 241–250. DOI: http://dx.doi.org/10.1145/1985793.1985827

[5] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09).* Association for Computing Machinery, New York, NY, USA, 243–262. DOI: http://dx.doi.org/10.1145/1640089.1640108

[6] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6, Article 26 (Dec. 2011), 66 pages. DOI: http://dx.doi.org/10.1145/2049697.2049700

[7] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape Analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '99).* ACM, New York, NY, USA, 1–19. DOI: http://dx.doi.org/10.1145/320384.320386

[8] Patrick Cousot and Radhia Cousot. 2002. Modular Static Program Analysis. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02).* Springer-Verlag, London, UK, UK, 159–178. http://dl.acm.org/citation.cfm?id=647478.727794

[9] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995,* Mario Tokoro and Remo Pareschi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 77–101.

[10] Graal. 2020. OpenJDK Graal. http://openjdk.java.net/projects/graal/. (2020).

[11] Ben Hardekopf and Calvin Lin. 2007. The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07).* ACM, New York, NY, USA, 290–299. DOI: http://dx.doi.org/10.1145/1250734.1250767

[12] Michael Hind. 2001. Pointer Analysis: Haven't We Solved This Problem Yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01).* Association for Computing Machinery, New York, NY, USA, 54–61. DOI: http://dx.doi.org/10.1145/379605.379665

[13] Vini Kanvar and Uday P. Khedker. 2016. Heap Abstractions for Static Analysis. *ACM Comput. Surv.* 49, 2, Article 29 (June 2016), 47 pages. DOI: http://dx.doi.org/10.1145/2931098

[14] Uday P. Khedker and Bageshri Karkare. 2008. Efficiency, Precision, Simplicity, and Generality in Interprocedural Data Flow Analysis: Resurrecting the Classical Call Strings Method. In *Proceedings of the International Conference on Compiler Construction (CC'08).* Springer-Verlag, Berlin, Heidelberg, 213–228. http://dl.acm.org/citation.cfm?id=1788374.1788394

[15] Thomas Kotzmann and Hanspeter Mössenböck. 2005. Escape Analysis in the Context of Dynamic Compilation and Deoptimization. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE '05).* ACM, New York, NY, USA, 111–120. DOI: http://dx.doi.org/10.1145/1064979.1064996

[16] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86.

[17] Kyungwoo Lee, Xing Fang, and Samuel P. Midkiff. 2007. Practical Escape Analyses: How Good Are They?. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE '07)*. Association for Computing Machinery, New York, NY, USA, 180–190. DOI: http://dx.doi.org/10.1145/1254810.1254836

[18] Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *Proceedings of the 15th International Conference on Compiler Construction (CC'06)*. Springer-Verlag, Berlin, Heidelberg, 47–64. DOI: http://dx.doi.org/10.1007/11688839_5

[19] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided Context Sensitivity for Pointer Analysis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 141 (Oct. 2018), 29 pages. DOI: http://dx.doi.org/10.1145/3276511

[20] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first Pointer Analysis with Self-tuning Context-sensitivity. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 129–140. DOI: http://dx.doi.org/10.1145/3236024.3236041

[21] Yue Li, Tian Tan, and Jingling Xue. 2019. Understanding and Analyzing Java Reflection. *ACM Trans. Softw. Eng. Methodol.* 28, 2, Article 7 (Feb. 2019), 50 pages. DOI: http://dx.doi.org/10.1145/3295739

[22] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. DOI: http://dx.doi.org/10.1145/2644805

[23] Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani. 2015. A Framework For Efficient Modular Heap Analysis. *Found. Trends Program. Lang.* 1, 4 (Jan. 2015), 269–381. DOI: http://dx.doi.org/10.1561/2500000020

[24] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (Jan. 2005), 1–41. DOI: http://dx.doi.org/10.1145/1044834.1044835

[25] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation.* Morgan Kaufmann.

[26] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. 2004. Importance of Heap Specialization in Pointer Analysis. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '04)*. ACM, New York, NY, USA, 43–48. DOI: http://dx.doi.org/10.1145/996821.996836

[27] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective Context-sensitivity Guided by Impact Pre-analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 475–484. DOI: http://dx.doi.org/10.1145/2594291.2594318

[28] Eclipse OpenJ9. 2020. The Eclipse OpenJ9 Virtual Machine. https://www.eclipse.org/openj9/. (2020).

[29] Rohan Padhye and Uday P. Khedker. 2013. Interprocedural Data Flow Analysis in Soot Using Value Contexts. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis (SOAP '13)*. ACM, New York, NY, USA, 31–36. DOI: http://dx.doi.org/10.1145/2487568.2487569

[30] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot^{TM} Server Compiler. In *Proceedings of the 2001 Symposium on Java^{TM} Virtual Machine Research and Technology Symposium - Volume 1 (JVM'01)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=1267847.1267848

[31] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. 2018. Systematic Evaluation of the Unsoundness of Call Graph Construction Algorithms for Java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops (ISSTA '18)*. Association for Computing Machinery, New York, NY, USA, 107–112. DOI: http://dx.doi.org/10.1145/3236454.3236503

[32] M Sharir and A Pnueli. 1978. *Two Approaches to Interprocedural Data Flow Analysis.* New York Univ. Comput. Sci. Dept., New York, NY. https://cds.cern.ch/record/120118

[33] Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages.* Technical Report.

[34] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (April 2015), 1–69. DOI: http://dx.doi.org/10.1561/2500000014

[35] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-sensitivity. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 17–30. DOI: http://dx.doi.org/10.1145/1926385.1926390

[36] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective Analysis: Context-sensitivity, Across the Board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 485–495. DOI: http://dx.doi.org/10.1145/2594291.2594320

[37] Alexandru Sălcianu and Martin Rinard. 2005. Purity and Side Effect Analysis for Java Programs. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*. Springer-Verlag, Berlin, Heidelberg, 199–215. DOI: http://dx.doi.org/10.1007/978-3-540-30579-8_14

[38] Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting. In *Static Analysis*, Xavier Rival (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 489–510.

[39] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 278–291. DOI: http://dx.doi.org/10.1145/3062341.3062360

[40] Manas Thakur and V. Krishna Nandivada. 2019. Compare Less, Defer More: Scaling Value-Contexts Based Whole-Program Heap Analyses. In *Proceedings of the 28th International Conference on Compiler Construction (CC 2019)*. Association for Computing Machinery, New York, NY, USA, 135–146. DOI: http://dx.doi.org/10.1145/3302516.3307359

[41] Manas Thakur and V. Krishna Nandivada. 2019. PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 16 (July 2019), 37 pages. DOI: http://dx.doi.org/10.1145/3337794

[42] Manas Thakur and V. Krishna Nandivada. 2020. Mix Your Contexts Well: Opportunities Unleashed by Recent Advances in Scaling Context-Sensitivity. In *Proceedings of the 29th International Conference on Compiler Construction (CC 2020)*. Association for Computing Machinery, New York, NY, USA, 27–38. DOI: http://dx.doi.org/10.1145/3377555.3377902

[43] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13–. http://dl.acm.org/citation.cfm?id=781995.782008

[44] John Whaley and Martin Rinard. 1999. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*. ACM, New York, NY, USA, 187–206.