An Efficient Tunable Selective Points-to Analysis for Large Codebases

Behnaz Hassanshahi Raghavendra K.R. Padmanabhan Krishnan
Bernhard Scholz * Yi Lu

Oracle Labs, Brisbane
{behnaz.hassanshahi, raghavendra.kr, paddy.krishnan, yi.x.lu}@oracle.com

Abstract

Points-to analysis is a fundamental static program analysis technique for tools including compilers and bug-checkers. Although object-based context sensitivity is known to improve precision of points-to analysis, scaling it for large Java codebases remains an challenge.

In this work, we develop a tunable, client-independent, object-sensitive points-to analysis framework where heap cloning is applied selectively. This approach is aimed at large codebases where standard analysis is typically expensive. Our design includes a pre-analysis that determines program points that contribute to the cost of an object-sensitive points-to analysis. A subsequent analysis then determines the context depth for each allocation site. While our framework can run standalone, it is also possible to tune it – the user of the framework can use the knowledge of the codebase being analysed to influence the selection of expensive program points as well as the process to differentiate the required context-depth. Overall, the approach determines where the cloning is beneficial and where the cloning is unlikely to be beneficial.

We have implemented our approach using Soufflé (a Datalog compiler) and an extension of the DOOP framework. Our experiments on large programs, including OpenJDK, show that our technique is efficient and precise. For the OpenJDK, our analysis reduces 27% of runtime and 18% of memory usage for a negligible loss of precision, while for Jython from the DaCapo benchmark suite, the same analysis reduces 91% of runtime for no loss of precision.

* Current Affiliation/Contact:School of Information Technologies, University of Sydney, bernhard.scholz@sydney.edu.au

SOAP'17

ISBN
...$15.00

DOI: nnnn.nnnn

1. Introduction

Points-to analysis is a fundamental analysis that is necessary to statically analyze programs especially from languages that support reference types and virtual dispatch like Java[1], C# and C++. The objectives for analyzing programs is varied, spanning from call-graph construction for compiler optimization to bug-finding and security for verification.

Technically, points-to analysis is a static program analysis technique that builds a heap abstraction of the input program without executing it. Adding context sensitivity to points-to analyses has been the standard way to improve the precision. Context-sensitivity is introduced by versioning variables and object-creation sites. Each version is attached to a context. There are many kinds of context sensitivities, including call-site-based, object-based and type-based. Object-based context sensitivity (object sensitivity) is shown to be the most effective context sensitivity in terms of both precision and computational costs for object-oriented programs when compared to call-site-based and type-based context-sensitive points-to analysis [4, 8].

However, applying a sufficiently precise object-sensitive points-to analysis uniformly to large programs is expensive with respect to runtime and memory consumption [9]. Our key observation in this work is that if the chosen heap cloning context is not sufficient for a particular object, the resulting points-to relation has many spurious facts for other related objects. This choice has a cascading effect on other objects, resulting in valuable resources being spent on computing irrelevant and unnecessary facts. Alternatively, increasing the level of object sensitivity for all allocation sites in an attempt to improve precision is not always feasible because it typically incurs unacceptably high computational costs without corresponding benefits. Smaragdakis et al. [9] have proposed an *introspective* points-to analysis to tackle the performance challenge. However, our experiments show that using their metrics to conduct 2O+1H points-to on OpenJDK[1] results in high loss of precision.

In this paper, we address the following research problem. Given a large program P, for which a context-sensitive points-to analysis does not scale, identify the allocation sites

in P that contribute to the generation of spurious points-to facts, and determine the object-sensitivity depth required for each such allocation site in P so as to make the points-to analysis scalable and precise. While our framework can run standalone, it is also possible to be tuned by the user to determine which allocation sites in P should be considered and which context depths are necessary for them.

The key ideas are the following:

1. A technique based on context-insensitive points-to analysis (which is inexpensive but very imprecise) to extract the kernel, i.e., a portion of the input program for which fixed object-sensitive points-to analysis is computationally expensive compared to the rest of the input program. The analysis accepts parameters from the user which is used to determine the size of the kernel. If the kernel is too big then the cost of the analysis cannot be reduced while if the kernel is too small the precision will suffer.
2. Metrics based on fixed object-sensitive points-to analysis on the kernel that determine the allocation sites where the contexts are insufficient or where contexts do not add much value.
3. A heuristic to identify the context-depths that will overcome the efficiency bottleneck with no or subtle loss of precision of fixed object-sensitive points-to analysis. That is, the metrics in the previous step are used to identify allocation sites where contexts need to be increased as well as to identify allocation sites where contexts can be reduced without decreasing precision significantly. This is also influenced by parameters which guide the calculation of the desired depth. If too many objects are chosen for deep contexts, the analysis is unlikely to scale because it is almost equivalent to having a deep context-sensitive analysis for the entire code-base while if too few objects are chosen, there will be significant loss of precision.

The rest of this paper is organised as follows. Section 2 establishes definitions and preliminaries required to understand the technicalities of our approach. Section 3 describes our technique in detail. Section 4 describes our implementation and experimental results. Section 5 surveys the related work in the literature and contrasts them with our approach. Section 6 concludes the paper by summarising our contributions and pointing to future directions.

2. Preliminaries

In this section, we summarise the key concepts and definitions that are used in the sequel of the paper. We port points-to analysis of the DOOP [3] framework and optimize it to Soufflé Datalog engine [6]. DOOP framework uses the domain of different sets, like the set of program variables, object allocation sites, method invocation sites, field names etc. The input program is represented as relations on these domains. From these relations, the points-to analysis is computed. For detailed definitions, please refer to [7].

Input Relations. We first describe the relations that capture the structure of the input program. We assume that the domain of classes (or types), methods and variables are defined.

$Alloc(o, m)$ indicates that there is an allocation site labelled o in method m.

$Store(b, f, v)$ indicates that there is an assignment of the form $b.f \leftarrow v$ where b and v are variables and f is a field. We overload this relation for storing elements in an array where f represents the index. However, as we do not distinguish the different elements in the array, the field f will represent a wild-card.

$Load(b, f, v)$ indicates that there is an assignment of the form $v \leftarrow b.f$ where b and v are variables and f is a field. As in the case of $Store$, the field f will represent a wild-card for arrays.

Computed Relations. Our framework computes the following relations and uses them in different stages of the analysis.

$PointsTo(hc, h, c, v)$ is the result of object-sensitive points-to analysis. It states that the variable v under the object context c points to an object h qualified with the heap context hc.

$PointsTo(h, v)$ is the result of context-insensitive points-to analysis. It states that the variable v points to an object h. We overload the same relation name both for the results of context-insensitive and object-sensitive points-to analysis.

$PointedToByVar(o)$ is the set $\{v \mid PointsTo(o, v)\}$.

$MethodPointsTo(m)$ is the set $\{o \mid PointsTo(o, v)$ for a variable v declared in $m\}$.

$ReachableCtx(m, c)$ states that method m is reachable in context c from the program's main entry/entries.

Object Sensitivity. The above definitions do not specify the exact contexts used in the analysis. To define a general n_1O+n_2H object-sensitive points-to analysis [9], the terms n_1O and n_2H have the following meanings:

n_1O**:** n_1**-Method cloning.** An instance method m that is a potential target of an invocation is distinguished with respect to an ordered sequence of objects of the form $(o_1, o_2, \ldots, o_{n_1})$, where o_{n_1} is the base object of the invocation of m and o_i is the object creating o_{i+1}. We often refer to this context as the object context.

n_2H**:** n_2**-Heap cloning.** A heap object o is distinguished with respect to an ordered sequence of objects of the form $(o_1, o_2, \ldots, o_{n_2})$, where o_{n_2} is the object creating o

[1] Java and JDK are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

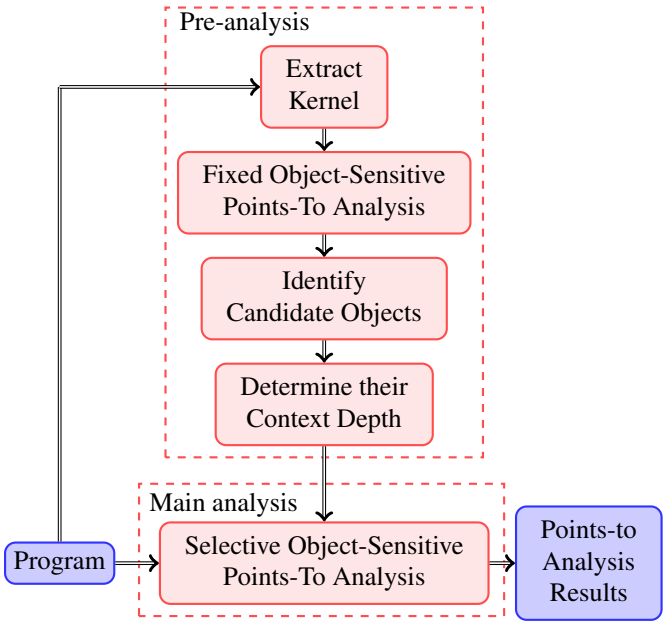

Figure 1: Workflow in Selective Points-To Analysis

and o_j is the object creating o_{j+1}. We often refer to this context as the heap context.

3. Selective Object-sensitive Points-to

In this section, we describe our general selective object-sensitive points-to analysis framework. Fig. 1 describes the key steps and the workflow in our framework. The approach can be separated into two stages: pre-analysis and main analysis.

The pre-analysis stage identifies the heap allocation sites that are potential bottlenecks when applying a chosen fixed object-sensitive points-to analysis uniformly on the input program. It also determines the requisite depth of object sensitivity needed for these allocation sites. The main analysis uses the information gathered in pre-analysis (i.e., the selective sensitivity for each allocation site) and performs the points-to analysis.

We describe the four steps in the pre-analysis in detail.

3.1 Extracting the Kernel

The first step is to extract the portion of the input program that contains the allocation sites that possibly generate spurious points-to information from a fixed object-sensitive points-to analysis. We refer to this portion of the input program as the kernel.

We extract the kernel using the results of a computationally inexpensive, context-insensitive, Anderson-style points-to analysis on the input program. Given thresholds K_1 and K_2, a class c of the input program is *selected* if:

- there is an object o with type c where the cardinality of $PointedToByVar(o)$ is larger than K_1, or
- there is an object o allocated in a method of class c where the cardinality of $PointedToByVar(o)$ is larger than K_1, or
- there is a method m in class c with the cardinality of $MethodPointsTo(m)$ is larger than K_2.

Then the kernel is constructed from the input program by removing all objects (allocation sites) except the selected classes. The intuition here is that for the remaining portion of the program, a context-insensitive or a fixed object-sensitive analysis is sufficiently precise.

Note that if there is even a single object o such that the cardinality of $PointedTByVar(o)$ is larger than K_1, the entire class identified by o and all its related objects will be retained. Also note that the kernel may not be a valid program but has the program elements that can make context-sensitive points-to analysis less precise.

Our framework uses the context-insensitive points-to analysis results to estimate the values for K_1 and K_2. The estimated values can be tried and chosen iteratively by setting a timeout limit. We start with the *mean* value between the minimum and maximum $PointedToByVar$ value for K_1. Similarly we select K_2 based on $MethodsPointsTo$. Intuitively, these values identify the heavy heap allocation sites and methods that cause a large percentage of the final points-to results. With these thresholds, if the computation of the fixed object sensitive points-to analysis of the kernel exceeds a timeout, then larger thresholds are selected. Typically we iterate by taking the mean value between the current K_1 (K_2) and the maximum $PointedToByVar$ ($MethodPointsTo$) value for K_1 (K_2).

Alternatively, the users can tune the analysis by choosing the thresholds interactively based on the estimated thresholds and their knowledge of the codebase. In this way, our framework is flexible in choosing a subset of the program that contains the problematic heap allocations.

3.2 Fixed Object-Sensitive Analysis

The second step (as shown in Fig. 1) is to perform a fixed object-sensitive points-to analysis on the kernel. For this step, we use the standard $mO+nH$ points-to analysis of the DOOP [3, 8] framework. Note that the fixed object-sensitive points-to analysis on the original program P is expensive but applying it on the kernel is not, as only a subset of the objects in the original program is retained. The main purpose here is to observe the value of contexts.

3.3 Identifying Candidate Objects

The third step (as shown in Fig. 1) is to identify the candidate allocation sites from the results of the fixed object-sensitive points-to analysis on the kernel where the contexts are not effective in removing spurious points-to tuples. Objects in the kernel that potentially have the *compounded smashing effect*, as we call it, are selected. Typically, certain program elements are not handled precisely. For example, the elements

of the arrays are *smashed*, i.e., they are not distinguished in a simple points-to analysis. That means, no matter what, we already have some amount of spuriousness. Adding context sensitivity to analyse such program elements may have a cascading effect, leading to many spurious points-to facts being generated. This we describe as compounded smashing effect. The reason could be either insufficient context sensitivity or context sensitivity is not the way to precisely handle program elements with compounded smashing effect.

3.4 Determinining Context Depths

The fourth and final step in the pre-analysis determines the requisite object sensitivity depth for the candidate allocation sites selected in the previous step. To help describe this process, we define some terms and metrics.

InFlow For a given object o, and field f, $InFlow_f(o)$ gives a measure of the heap contexts related to the heap objects that are stored in the field f of object o, as given by the fixed object-sensitive points-to analysis on the kernel. Recall that for arrays, the field f is ignored. To simplify the presentation we ignore the field f in all cases – but technically the $Load$ and $Store$ pairs are matched up via the named field f. Thus $InFlow(o)$ is defined as:

$$\left\{ (h, hc, oc) \;\middle|\; \begin{array}{l} Store(b, *, v) \text{ for some variables } b, v \\ PointsTo(oc, o, c, b) \text{ and} \\ PointsTo(hc, h, c, v) \text{ for some context } c \end{array} \right\}$$

OutFlow For a given variable v, object o, field f and a context oc, $OutFlow_f(v, o, oc)$ gives a measure of the heap contexts related to the heap objects that are loaded from the field f of the object o qualified with v's context oc. But as in the case of $InFlow$, we ignore the field f to simplify the explanation. More precisely, the $Outflow(v, o, oc)$ is defined as the set:

$$\left\{ (c, h, hc) \;\middle|\; \begin{array}{l} Load(b, *, v) \text{ for some variable } b \\ PointsTo(oc, o, c, b) \text{ and} \\ PointsTo(hc, h, c, v) \end{array} \right\}$$

The intuition behind $OutFlow$ is analogous to $InFlow$ and captures the cascading or multiplier effect of the $Load$ operation. However, note that this set is defined per (loaded) variable and per heap cloning context.

ContextValue For an object o and a context oc, $ContextValue(o, oc)$ estimates the value of the heap context oc generated for o. In other words, it determines the effectiveness of the context generated for an object. More precisely, we define $ContextValue(o, oc)$ as:

$$\min_{v} \frac{|InFlow(o)| \cdot |CtxInOutFlow(v, o, oc)|}{|OutFlow(v, o, oc)|}$$

where $CtxInOutFlow(v, o, oc)$ is

$$\{c \mid (c, h, hc) \in OutFlow(v, o, oc) \text{ for some } h, hc\}.$$

Intuitively, for an object o with sufficiently large $InFlow$, if $ContextValue(o,\ oc) = 1$, it means that the context oc has not distinguished any points-to facts. If $ContextValue(o, oc)$ is greater than a threshold, say K_3, then context oc is valuable as it has distinguished some points-to facts and so we can keep that. When $ContextValue(o, oc) < K_3$ and $Inflow(o) > K_4$, our heuristic is to increase the context depth for o to the maximum desired if a deeper context adds value (e.g., Collections such as HashMap). Otherwise, we switch off the context for o. Note that increasing the context depth for o entails correspondingly increasing the context depths for the objects in the context oc. Similar to K_1 and K_2, we treat K_3 and K_4 as input parameters to our analysis. Intuitively, very high values for K_3 and K_4 will not refine the depth (so the results are likely to be imprecise) while choosing very low values will refine too many allocation sites (so the analysis is unlikely to scale).

Now the main analysis is straightforward. It applies a selective object-sensitive points-to analysis on the whole input program. This analysis applies a fixed object sensitivity to objects not in the kernel. For every object in the kernel, the context depth as identified by the pre-analysis is applied.

4. Implementation and Experiments

We implemented our framework using DOOP[3] in Datalog declarative language. We use Soufflé as the Datalog engine [6]. We used SQLite relational database system to compute the metrics of Section 3. We used the workstation – Xeon E5-2699 2.30GHz machine with 396 GB RAM – utlizing 8 cores for all our experiments.

For our experiments, we considered OpenJDK7-b147 and the programs from DaCapo 2006-10-MR2 [2]. As the JDK is a library, the points-to analysis is extended with the construction of the Most General Application (MGA) [1] modeling the unknown application. Programs in DaCapo benchmark is analyzed with the JDK. The sizes of OpenJDK and the largest program in DaCapo – Jython (with JDK), is given in Table 1.

For the purposes of our experiments, we set the default maximum context-sensitivity to be 3O+3H. Note that it is not feasible to compute 3O+3H context-sensitive points-to analysis for OpenJDK.

4.1 Framework Setup

In order to extract the kernel in the pre-analysis phase, we need to provide K_1 and K_2 as thresholds. We have found the mean sizes of PointedToByVar and MethodsPointsTo very effective for K_1 and K_2 respectively and applied them to all of the programs in the Dacapo benchmark. For larger and

Benchmarks	Size			Runtime(seconds)			Memory (GB)		
	#Variables	#Call-sites	#Heap-allocations	2O1H	Ours	Imp.	2O1H	Ours	Imp.
OpenJDK	1440875	591262	4815	16200	11880	27%	186	153	18%
Jython	142641	59379	2425	1280	109	91%	2.8	2.8	0%

Table 1: Size of Programs and Performance Improvements

Benchmarks	#VarPointsTo			#Alias			#ReachableMethods		
	2O1H	Ours	Loss	2O1H	Ours	Loss	2O1H	Ours	Loss
OpenJDK	10100000	10400000	3%	845518	854600	1%	39909	39880	-0.1%
Jython	68519	68519	0%	28294	28294	0%	2876	2876	0%

Table 2: Experimental results – Precision

more complex codebases like OpenJDK the 2O+1H points-to analysis on the kernel extracted by the initial estimates of K_1 and K_2 did not terminate within a timeout limit (30 minutes). Following the heuristic explained in Section 3.1, we revised K_1 and K_2 to the mean of the current values and the maximum $|PointedToByVar|$ and $|MethodPointsTo|$ respetively. The 2O1H points-to analysis on the extracted kernel timed out. On the next mean, with $K_1 = 20000$ and $K_2 = 50000$, the 2O1H points-to analysis terminated in 25 minutes. Hence we used these thresholds for OpenJDK and recommend thresholds from third mean onwards for large programs like OpenJDK.

We use objects that have Array type as candidates due to their compounded smashing effect. Appropriate values for K_3 and K_4 that correspond to $ContextValue$ and $InFlow$ can be chosen based on the following observations. A low value (e.g., 10) for K_3 would result in missing some of the objects whose contexts have low values. On the other hand, a high value (e.g., 400) for K_3 would unnecessarily increase/decrease context depths for objects whose context were performing well. Similarly, a low value (e.g., 10) for K_4 would consider the objects that are not contributing that much to points-to propagation. On the other hand, a high value (e.g., 400) may miss objects that are responsible for generating spurious contexts. For all the benchmarks, we used $K_3 = 200$ and $K_4 = 200$ in our experiments and also recommend them to be used in general (note that small changes to these values does not matter).

4.2 Loss of Precision

We compare the precision of our technique and 2O+1H using three clients—size of variable to object points-to relation, size of alias relation and the number of reachable methods. Our technique (with the above thresholds) found that no objects for programs in DaCapo benchmarks, except for Jython, needs context refinement. This implies that for small programs like programs other than Jython in DaCapo benchmark, regular 2O1H can be used directly. We now focus on Jython and OpenJDK. Table 1 shows the performance numbers for the benchmarks and the improvements obtained. Table 2 contrasts the numbers from three precision clients. From both of the tables, we observe that there is no or very small loss of precision. Hence it is clear that for a possible slight loss of precision we gain high runtime efficiency and potential memory reduction for large programs like OpenJDK.

Note that we do not include the steps of the pre-analysis in runtime. This is because we are unable to reuse the results of points-to analysis in subsequent analyses effectively. Having determined the context depths for objects needing refinement by running the preanalysis once, we run the main analysis independently. Hence the cost of preanalysis is ammortized over many runs of the main analysis.

5. Related Work

Research works on context-sensitive analysis may be classified into client-independent and client-dependent/demand-driven analyses.

5.1 Client-independent Analyses

Object sensitivity was first introduced by Milanova et al. [4]. They empirically show that object sensitivity is better than call-site sensitivity for object-oriented programs. They also introduce a parametric object sensitivity for targeted context sensitivity. Here, the framework user must identify the parts of the program where more or less object sensitivity is required. Our main distinguishing contribution in this paper is to identify these parts of the program and their required context depths.

Smaragdakis et al. [9] proposed an introspective context-sensitive points-to analysis in DOOP framework. First, a context-insensitive, Anderson-style points-to analysis is done. Based on metrics such as *PointedToByVar, MethodPointsTo*, they then determine allocation sites and method invocation sites where contexts are necessary. Finally, they perform a fixed object-sensitive analysis switching off con-

text sensitivities at allocation sites and method invocation sites that do not satisfy the selected heuristic. Applying these heuristics on the 2O+1H points-to analysis over OpenJDK resulted a significant loss of precision. It removed only 35% of the context-insensitive points-to facts, whereas ours removed 96.7%. Hence, we investigate techniques that go beyond the binary selection between context-insensitive and a fixed object sensitivity. We apply a spectrum of context sensitivities to different parts of the program and achieve scalability without significantly losing precision.

Wei et al. [10] propose adaptive context-sensitive points-to analysis for JavaScript programs. They apply machine learning algorithms on the function characteristics collected from context insensitive points-to analysis results to determine the kind of context sensitivities – insensitive, 1 call-site, 1 object, ith-parameter object – to be applied. Based on the associated context sensitivities for each function, they finally do a selective context-sensitive points-to analysis on the whole program. However, the depth of contexts does not go beyond 1. For our focus on object-oriented Java programs, we considered only object-sensitive analyses, as has been clearly established as the way to gain precision in Milanova et al. [4] work. We then investigated varied depths of object sensitivity for different allocation sites, which helped us to scale to large programs.

5.2 Client-dependent Analyses

In principle, client-dependent analysis suffers from its non-generality because the technique and the results may not work for a different client. Our objective of points-to analysis is to extract the basic structure of the program, so that many clients, such as call-graph construction, taint analysis, escape analysis, make use of it. This objective is the fundamental distinction of the client-dependent context-sensitive points-to analysis works.

Oh et al. [5] use an *impact* pre-analysis to determine at a program point whether call-site context (with predefined depth) is necessary for a method. Here, they deal with C programs, hence the focus is only on method contexts and not heap cloning. First, they fix a call-site context depth. Then they do an *impact* pre-analysis, i.e., they do the context-sensitive (with the defined depth) analysis of the whole program but with a simpler or the simplest abstract domain, such as $(\top, \bot)$. From the results of this analysis, they determine whether to apply context sensitivity (of the defined depth) for a method. This technique is not applicable for our objective of constructing points-to sets for object-oriented programs as constructing simple abstract domains to evaluate context sensitivity of points-to analysis is not known.

Zhang et al. [11] proposed a counter-example guided approach to iteratively clone a method, thus adding call-site sensitivity, for a given set of client queries. Cloning methods based on call-sites is proven to be a unviable way to increase precision for object-oriented programs [4]. Similarly, employing context-insensitive points-to analysis and MAXSAT solver multiple times iteratively is not a feasible method when our objective is to scale large programs of the order of JDK.

6. Conclusion

In this paper we have presented a framework to scale object-sensitive points-to analysis for large object-oriented programs. It involves identifying and experimenting on the kernel of the program. Then based on our metrics, a selective object-sensitive points-to analysis is applied on the input program. Our experiments on large programs, such as the JDK and Jython from DaCapo [2] benchmarks, show the huge effectiveness of our technique. Further experimentation on other large programs is required.

References

[1] N. Allen, P. Krishnan, and B. Scholz. Combining type-analysis with points-to analysis for analyzing Java library source-code. In *Proceedings of the SOAP Workshop*, pages 13–18. ACM, 2015.

[2] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[3] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, pages 243–262, 2009.

[4] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM TOSEM*, pages 1–41, January 2005.

[5] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi. Selective context-sensitivity guided by impact pre-analysis. In *PLDI*, pages 475–484, 2014.

[6] B. Scholz, H. Jordan, P. Subotić, and T. Westmann. On fast large-scale program analysis in Datalog. In *Compiler Construction (CC)*, pages 196–206, 2016.

[7] Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 1(2):1–69, 2015.

[8] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *POPL*, 2011.

[9] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *PLDI*, 2014.

[10] S. Wei and B. G. Ryder. Adaptive context-sensitive analysis for javascript. In *ECOOP*, pages 712–734, 2015.

[11] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in datalog. In *PLDI*, 2014.