Refinement in Object-Sensitivity Points-To Analysis via Slicing

GIRISH MASKERI RAMA*, Infosys Limited, India
RAGHAVAN KOMONDOOR, Indian Institute of Science, India
HIMANSHU SHARMA, Indian Institute of Science, India

Object sensitivity analysis is a well-known form of context-sensitive points-to analysis. This analysis is parameterized by a bound on the names of symbolic objects associated with each allocation site. In this paper, we propose a novel approach based on object sensitivity analysis that takes as input a set of client queries, and tries to answer them using an initial round of inexpensive object sensitivity analysis that uses a low object-name length bound at all allocation sites. For the queries that are answered unsatisfactorily, the approach then pin points "bad" points-to facts, which are the ones that are responsible for the imprecision. It then employs a form of program slicing to identify allocation sites that are potentially causing these bad points-to facts to be generated. The approach then runs object sensitivity analysis once again, this time using longer names for just these allocation sites, with the objective of resolving the imprecision in this round. We describe our approach formally, prove its completeness, and describe a Datalog-based implementation of it on top of the Petablox framework. Our evaluation of our approach on a set of large Java benchmarks, using two separate clients, reveals that our approach is more precise than the baseline object sensitivity approach, by around 29% for one of the clients and by around 19% for the other client. Our approach is also more precise on most large benchmarks than a recently proposed approach that uses SAT solvers to identify allocation sites to refine.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; *Software safety*; *Object oriented languages*; Software verification;

Additional Key Words and Phrases: Precise and scalable points-to analysis, client-driven refinement, Java

ACM Reference Format:
Girish Maskeri Rama, Raghavan Komondoor, and Himanshu Sharma. 2018. Refinement in Object-Sensitivity Points-To Analysis via Slicing. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 142 (November 2018), 27 pages.
https://doi.org/10.1145/3276512

1 INTRODUCTION

Points-to analysis is a fundamental problem in static program analysis, and involves the use of static abstractions to determine the memory locations that each variable or each field of an object can point to. Points-to analysis has varied applications, such as in compilation and optimization of programs, verification of programs, and program understanding and program transformation tools. A large number of different approaches have been proposed over the last two decades or so for

*Part of this work was done when the author was pursuing part-time PhD at Indian Institute of Science, Bangalore.

Authors' addresses: Girish Maskeri Rama, Infosys Limited, India, girish_rama@infosys.com; Raghavan Komondoor, Indian Institute of Science, Bangalore, India, raghavan@iisc.ac.in; Himanshu Sharma, Indian Institute of Science, Bangalore, India, himanshu.ecko@gmail.com.

2475-1421/2018/11-ART142
https://doi.org/10.1145/3276512

points-to analysis, for different languages, and at different points in the precision vs. scalability tradeoff [Kanvar and Khedker 2016; Smaragdakis and Balatsouras 2015; Sridharan et al. 2013].

An important characteristic of any points-to analysis approach is whether it is *context insensitive*, wherein information entering into a method from all call sites are merged and propagated back to all return-sites, or whether it is *context sensitive*, wherein different call-sites to a method are distinguished at least to a partial extent. Context insensitivity leads to significant loss of precision compared to context sensitivity [Lhoták and Hendren 2006]. Hence, almost all modern points-to analysis approaches use some form of context sensitivity. There are several broad families of context-sensitive approaches; for instance, ones that (i) use *procedure summaries* [Burke et al. 1994; Chatterjee et al. 1999; Emami et al. 1994; Madhavan et al. 2012], or (ii) analyze each method under different contexts, with each context representing (a portion of) a call stack (also known as k-CFA) [Guyer and Lin 2003; Whaley and Lam 2004], or (iii) analyze each method under different contexts via *object sensitivity* [Milanova et al. 2005], wherein the contexts for an invocation site are encoded using the static symbolic IDs of possible *receiver* objects at the invocation site.

Object-sensitivity analysis is designed specifically for object-oriented languages such as Java. This technique has been found to be both more scalable and more precise than other alternatives [Lhoták and Hendren 2006; Smaragdakis et al. 2011]. Several recently proposed points-to analysis approaches are based on object sensitivity [Smaragdakis et al. 2011; Tan et al. 2016; Zhang et al. 2014]. It has also been implemented in popular static analysis tools such as Wala [Wala [n. d.]], Doop [Doop [n. d.]], and Petablox [Petablox [n. d.]].

The precision (and scalability) of object-sensitivity analysis is parameterized by the lengths of the names of symbolic objects that are used in the analysis. Using longer names increases the number of (static) symbolic objects, and hence, the number of different contexts in which each method is analyzed. This increases precision. However, the increased number of method contexts increases the time requirement of the analysis, and can adversely impact the scalability of the analysis. In fact, k-CFA context sensitivity also has a similar property; longer context names can be used to increase the number of contexts, but at the expense of scalability.

A basic idea that has been employed in the literature to deal with this conundrum is client-driven *refinement* [Guyer and Lin 2003; Sridharan and Bodík 2006; Zhang et al. 2014]. The intuition here is that given a specific client *query* regarding the points-to set of a given variable, a subset of method invocations that are pertinent to resolving the query are somehow identified. These invocations alone are analyzed under greater number of contexts, while other method invocations are analyzed using their original contexts. In other words, extra effort is spent only on necessary method invocations, thereby obtaining better precision for the given query without exploding the expense of the analysis.

Our main contribution in this paper is a new refinement technique for object-sensitivity analysis, which uses a novel form of *program slicing* to identify allocation sites that are pertinent to the given query and that are to be refined. In the remainder of this section we introduce a running example, give an informal overview of our approach, and contrast our approach with other existing refinement techniques.

1.1 Motivating Example

Figure 1 depicts a toy Java program that we will be using as a running example throughout this paper. The program has a class 'M', with three methods, namely, 'main', 'foo', and 'bar', and a class 'Contain', with two methods, namely, 'get' and 'put'. A, B, and C are three other classes, with B and C extending A; the definitions of these classes are omitted from the figure in the interest of brevity. The class 'Contain' simulates a container; each object of this class points to a single object of type A via its 'f' field.

```
1 class M {
2  static void main() {
3   M v1 = new M(); // s1
4   M v2 = new M(); // s2
5   M v3 = new M(); // s3
6   M v4 = new M(); // s4
7   A t1;
8   if (..)
9    t1 = v1.foo();
10  else if (..) t1 = v2.foo();
11  else {
12   A v5 = new B(); // s5
13   A v6 = new C(); // s6
14   // B and C both extend A
15   Contain c1 = v3.bar(v5);
16   Contain c2 = v4.bar(v6);
17   t1 = c1.get();
18  }
19  (B) t1;
20 } // end main
21 A foo() {
22  Contain cf = new Contain(); // s7
23  A vf = new B(); // s8
24  cf.put(vf);
25  A rf = cf.get();
26  return rf;
27 }
28
29 Contain bar(A vb) {
30  Contain cb = new Contain(); // s9
31  cb.put(vb);
32  return cb;
33 }
34 } // end class M
35
36 class Contain {
37 A f;
38 A get() {
39  A rg = this.f;
40  return rg;
41 }
42 put(A vp) {
43  this.f = vp;
44 }
45 }
```

Fig. 1. Example program

Consider a query about what objects the variable t1 in function 'main' can point to. This query would be useful in verifying whether the downcast in Line 19 is definitely safe. In reality, t1 can only point to objects allocated at sites s_8 and s_5. This is the ideal answer to this query, and this answer would enable the downcast to be declared "safe".

1.2 Milanova's Object Sensitivity Analysis

We now consider Milanova et al.'s object-sensitivity analysis, which is our baseline analysis. A sampling of the steps in this analysis when applied on our running example are illustrated in the first two columns of Figure 2. We have used the basic mode of the analysis, wherein all symbolic objects allocated at all sites use names of length 1 (this is also known as "$k = 1$ object-sensitivity analysis"). Column (1) shows the points-to facts inferred by the analysis, in the order in which they are inferred. Symbolic (static) objects are named o_i, o_j, etc., with the convention that o_i represents objects allocated at site s_i, o_j represents objects allocated at site s_j, and so on. We use the variable name 'this_m' to denote the implicit variable 'this' in method m. Each points-to fact is labeled for convenience with a number, such as $\langle 1 \rangle$, $\langle 2 \rangle$, etc. Each row in Column (2) in Figure 2 indicates the previously inferred facts and the lines of code that cause the facts in the same row in Column (1) to be inferred. Note that the object-sensitivity analysis is *flow-insensitive*; this means that all the facts are associated with all the points in the program. In the rest of this paper, wherever there would be no confusion, we simply say "object" to mean a (static) symbolic object.

Note that there are two basic kinds of facts that are inferred by the analysis. Facts other than $\langle 8 \rangle$ and $\langle 10 \rangle$ in the figure are of the first kind, and indicate the points-to sets of variables. For any

	(1) Points-to fact(s) with $k_i = 1, \forall i$	(2) Caused by	(3) Points-to facts(s) with k_9=2
1	$\langle 1 \rangle$ v4 $\rightarrow o_4$, $\langle 2 \rangle$ v6 $\rightarrow o_6$	Lines 6, 13	Same as Column (1)
2	$\langle 3 \rangle$ (o_4, this_bar) $\rightarrow o_4$	Lines 16, 29, $\langle 1 \rangle$	Same as Column (1)
3	$\langle 4 \rangle$ (o_4, vb) $\rightarrow o_6$	Lines 16, 29, $\langle 1 \rangle$, $\langle 2 \rangle$	Same as Column (1)
4	$\langle 5 \rangle$ (o_4, cb) $\rightarrow o_9$	Line 30, $\langle 3 \rangle$	(o_4, cb) $\rightarrow o_{49}$
5	$\langle 6 \rangle$ (o_9, this_put) $\rightarrow o_9$	Lines 31, 42, $\langle 5 \rangle$	(o_{49}, this_put) $\rightarrow o_{49}$
6	$\langle 7 \rangle$ (o_9, vp) $\rightarrow o_6$	Lines 31, 42, $\langle 4 \rangle$, $\langle 5 \rangle$	(o_{49}, vp) $\rightarrow o_6$
7	$\langle 8 \rangle$ o_9.f $\rightarrow o_6$	Line 43, $\langle 6 \rangle$, $\langle 7 \rangle$	o_{49}.f $\rightarrow o_6$
8	$\langle 9 \rangle$ c2 $\rightarrow o_9$	Line 32, 16, $\langle 5 \rangle$, $\langle 1 \rangle$	c2 $\rightarrow o_{49}$
9	$\langle 10 \rangle$ o_9.f $\rightarrow o_5$, $\langle 11 \rangle$ c1 $\rightarrow o_9$	Analogous to $\langle 8 \rangle$, $\langle 9 \rangle$, but due to Line 15, 'bar', 'put'	o_{39}.f $\rightarrow o_5$, c1 $\rightarrow o_{39}$
10	$\langle 12 \rangle$ (o_9, this_get) $\rightarrow o_9$	Lines 17, 38, $\langle 11 \rangle$	(o_{39}, this_get) $\rightarrow o_{39}$
11	$\langle 13 \rangle$ (o_9, rg) $\rightarrow o_6$	Line 39, $\langle 8 \rangle$, $\langle 12 \rangle$	(o_{39}, rg) $\rightarrow o_5$
12	$\langle 14 \rangle$ t1 $\rightarrow o_6$	Lines 40, 17, $\langle 13 \rangle$, $\langle 11 \rangle$	t1 $\rightarrow o_5$

Fig. 2. Points-to facts, and causality between facts, corresponding to example in Figure 1

non-static method, its variables have separate points-to sets for each "receiver" object context under which the method is invoked. For instance, facts $\langle 3 \rangle$ and $\langle 4 \rangle$ are for context o_4. They arise because at the invocation site in Line 16 in the program (see Figure 1) the "base" variable v4 points to the "receiver" object o_4. Similarly, facts $\langle 6 \rangle$ and $\langle 7 \rangle$ are for context o_9, because the base variable 'cb' in the invocation in Line 31 points to receiver object o_9. 'cb' in turn points to o_9 (see fact $\langle 5 \rangle$) due to the allocation site in Line 30.

Facts $\langle 8 \rangle$ and $\langle 10 \rangle$ are of the second kind. For instance, fact $\langle 8 \rangle$ indicates that the 'f' field of object o_9 points to object o_6. This was inferred due to the "put-field" in Line 43, in conjunction with the points-to sets of 'this_put' and 'vp' (as indicated by facts $\langle 6 \rangle$ and $\langle 7 \rangle$).

Note the imprecision in fact $\langle 14 \rangle$. Variable t1 actually cannot point to (concrete) objects allocated at site s_6. This imprecision causes the analysis to declare the downcast as potentially unsafe. The imprecision is fundamentally because in the analysis variables c1 and c2 both point to the same (symbolic) object, namely, o_9 (see facts $\langle 9 \rangle$ and $\langle 11 \rangle$). Therefore, the invocation sequence via Line 16 and then Line 31 causes o_9.f to point to what v6 points to, namely, o_6 (see fact $\langle 8 \rangle$). Subsequently, the invocation at Line 17 causes t1 to be assigned to what is pointed to by o_9.f, which includes o_6 (as well as o_5). Whereas, in reality, the invocations at Lines 16 and 17 cannot interact with each other, because c1 and c2 necessarily point to different 'Contain' objects (although both these objects are allocated at site s_9).

1.3 Our Proposal: Refinement Via Program Slicing

Our proposal is a query-driven refinement approach for the object sensitivity analysis. The preliminary step is to run the object-sensitivity approach in a suitably efficient mode, such as $k = 1$ (i.e., all symbolic objects having names of length 1). Our approach is applied only if the initial analysis mentioned above returns an unsatisfactory outcome for the given query. For instance, for a downcast-safety query, an unsatisfactory outcome would be one where the downcast is declared potentially unsafe. In this case our approach is applied. The starting point of our approach is the points-to fact that caused the unsatisfactory outcome. We call such a fact a "bad" fact, and the object that occurs in this fact a "bad" object. The core of our approach is to perform a form of backward traversal among the set of points-to facts inferred by the preliminary analysis, along a

causality relation, to identify the facts that directly or transitively caused the analysis to infer the seed "bad" fact. We call these facts the "causing" facts of the bad fact. Then, the allocation sites that are involved in the inference of these causing facts are marked for refinement. A second round of the object sensitivity analysis is now applied, this time using longer object names for objects allocated at the marked allocation sites (and names of length 1 at the remaining allocation sites). This round of the analysis is likely to be more precise than the preliminary analysis, and could potentially yield a satisfactory answer to the query. In case it does not, the steps mentioned above can be iterated as many time as desired, using the facts derived by the object-sensitivity analysis in any iteration as the basis for the next iteration.

For instance, consider the facts generated by the preliminary analysis, as shown in Column (1) of Figure 2. Fact ⟨14⟩ (in the last row) is the "bad" fact, as it causes the downcast in Line 19 of the program (see Figure 1) to be declared as potentially unsafe. Our backward traversal basically starts from the bad fact, and finds the facts that cause, directly or transitively, the bad fact to be inferred. These causality relationships are captured in Column (2) of Figure 2. For instance, fact ⟨14⟩ is directly caused by facts ⟨13⟩ and ⟨11⟩ (see the last row, Column (2)). Fact ⟨13⟩ is caused by facts ⟨8⟩ and ⟨12⟩, and so on. Basically, the backward traversal eventually reaches all the facts that are mentioned in Column (2) in Figure 2. Of these facts, facts ⟨1⟩, ⟨2⟩, and ⟨5⟩ are directly caused by allocation sites, namely, sites s_4, s_6, and s_9, in the example program. Therefore, these allocation statements are marked for refinement.

In the next round of the object-sensitivity analysis, we use longer names (say, of length 2) at site s_9 (sites s_4 and s_6 are in 'main', and cannot use names of length more than 1). The facts that would be inferred by the second round of object-sensitivity analysis are shown in Column (3) in Figure 2. Note, the object o_9 has been refined into two distinct objects, namely, o_{49} and o_{39}. o_{49} represents objects created at site s_9 when the containing method (i.e., 'bar') is called with o_4 as the receiver object (i.e., from Line 16 in the program). Whereas, o_{39} represents objects created at site s_9 when 'bar' is called with o_3 as the receiver object (i.e., from Line 15 in the program). With this refinement, c1 and c2 point to distinct objects, namely, o_{39} and o_{49}. This eliminates the inter-dependence between Lines 16 and 17 in the program, and hence helps infer that t1 points to o_5 alone, and not to o_6.

Our approach is efficient, because it tries to minimize unnecessary marking of allocation sites for refinement. The way this is achieved is by marking only those sites that are reached in the traversal from "bad" facts, i.e., facts that cause a query to be answered imprecisely. In our example, if the traversal were to be done from *all* facts that the variable t1 pointed to in the first round of analysis, then the fact 't1 $\rightarrow o_8$' (which is not a bad fact) would have caused allocation sites s_7 and s_8 to be also marked for refinement. Such unnecessary refinement would increase the running time of the second round of the analysis without any corresponding increase in the precision with which the query is answered.

1.4 Precision Of Our Approach

The precision of any refinement approach is the extent to which allocation sites are not unnecessarily marked for refinement. For instance, in the running example, our backwards traversal approach marks site s_9 for refinement, but *does not* mark site s_7 (in method 'foo') for refinement. Refining s_7 would split it into two objects o_{17} and o_{27}, and would end up analyzing methods 'get' and 'put' under separate contexts o_{17} and o_{27}, instead of just o_7. This would increase the expense of the analysis. However, this would have no impact on the precision with which the downcast safety query is answered, because the refinement (or not) of s_7 does not affect the points-to sets of the variables that are actually relevant to the query, namely, c1 and c2.

Our approach uses two principle ideas to avoid marking allocation sites unnecessarily for refinement. These are:

- Perform the backwards traversal of the causality relationship among the points-to facts *context sensitively*.
- Perform a focused form of backwards traversal that reaches only the facts that had caused the points-to analysis to infer the "bad" fact. Facts that cause "harmless" facts to be inferred are not reached in the traversal. A harmless fact could be one, for instance, that involves the query variable (t1 in the example), but is not responsible for the query not getting answered satisfactorily originally. An example of such a "harmless" fact in our example would be 't1 $\rightarrow o_5$'.

We discuss both these aspects in more detail in Section 3 of this paper. However, it is notable that if our backwards traversal approach did not possess *either one* of the two features above, then it would lose precision, and would mark certain allocation sites unnecessarily for refinement, such as site s_7 in the running example.

The primary objective of our backwards traversal is to identify points-to facts that directly or indirectly cause a given seed (or "bad") points-to fact to be inferred by the points-to analysis. However, because each points-to fact is generated by the points-to analysis due to the presence of one or more specific statements in the program, the statements corresponding to the facts that are reached in the backward traversal effectively form a slice of the program when the seed (i.e., "bad") fact is treated as the slicing criterion. We discuss the connection of our approach with backward slicing in a more detailed manner in Section 3.

1.5 Novelty Of Our Approach

Several other approaches exist that selectively refine the precision of points-to analysis in accordance with the needs of a given client analysis [Guyer and Lin 2003; Sridharan and Bodík 2006; Zhang et al. 2014]. Our approach is empirically more scalable than some of these approaches. It is potentially simpler to implement than almost all these approaches, because these approaches are based on an intertwining of the actual points-to analysis with the client analysis. Finally, some of these approaches do not target refinement of object sensitivity in particular, which is our focus. We discuss related work in more detail in Section 6.

1.6 Contributions

In summary, the main contributions of this paper are as follows:

- A novel, program-slicing based approach to refine object-sensitivity analysis to answer user queries more precisely.
- Our context-sensitive program-slicing approach, viewed standalone, could be of interest to the community. It is a generalization of the "def-use" analysis in Milanova et al.'s paper. Our slicing technique differs from most standard techniques for context-sensitive slicing of Java programs, which are based on system dependence graphs [Horwitz et al. 1990; Liang and Harrold 1998], and which do not scale readily to larger programs [Sridharan et al. 2007].
- An implementation of our refinement approach, based on the Petablox [Petablox [n. d.]] program analysis framework.
- An evaluation of our implementation on nine benchmarks from the Dacapo 2006 benchmark suite [Blackburn et al. 2006], using two separate client analyses. Our evaluation reveals that for a given time budget (10 hours per benchmark), our approach gives 29% more precise results than then baseline object-sensitivity analysis for one of the clients, and and 19% more precise results for the other client.

$$\frac{\text{“v = w”} \in \mathcal{S},\ ((c, \text{w}) \rightarrow o) \in \mathcal{F}}{((c, \text{v}) \rightarrow o) \in \mathcal{F}}\ [\text{M-Asgn}] \qquad \frac{\begin{array}{c}\text{“v = w.f”} \in \mathcal{S},\ ((c, \text{w}) \rightarrow o_1) \in \mathcal{F} \\ (o_1.f \rightarrow o_2) \in \mathcal{F}\end{array}}{((c, \text{v}) \rightarrow o_2) \in \mathcal{F}}\ [\text{M-Get}]$$

$$\frac{\begin{array}{c}\text{“v.f = w”} \in \mathcal{S},\ ((c, \text{v}) \rightarrow o_1) \in \mathcal{F} \\ ((c, \text{w}) \rightarrow o_2) \in \mathcal{F}\end{array}}{(o_1.f \rightarrow o_2) \in \mathcal{F}}\ [\text{M-Put}] \qquad \frac{\text{“}s_q\text{: v = new” in meth } m^j,\ ((c, \text{this\_}m^j) \rightarrow c) \in \mathcal{F}}{((c, \text{v}) \rightarrow mkName(c, q, k_q)) \in \mathcal{F}}\ [\text{M-New}]$$

$$\frac{\text{“v = w.m(z)”} \in \mathcal{S},\ \{(c, \text{z}) \rightarrow o_1, (c, \text{w}) \rightarrow c_1\} \subseteq \mathcal{F},\ m^j(\text{p}) = dispatch(c_1, m),}{((c_1, \text{p}) \rightarrow o_1) \in \mathcal{F},\ ((c_1, \text{this\_}m^j) \rightarrow c_1) \in \mathcal{F}}\ [\text{M-Call}]$$

$$\frac{\text{“v = w.m(z)”} \in \mathcal{S},\ ((c, \text{w}) \rightarrow c_1) \in \mathcal{F},\ m^j(\text{p}) = dispatch(c_1, m),\ ((c_1, \text{ret\_}m^j) \rightarrow o_2) \in \mathcal{F}}{((c, \text{v}) \rightarrow o_2) \in \mathcal{F}}\ [\text{M-Ret}]$$

Fig. 3. Object sensitivity approach

Our approach also gives higher precision on all nine benchmarks when compared with Zhang et al.'s query-driven refinement approach [Zhang et al. 2014].

- A proof of "completeness" of our approach. That is, every allocation site that, upon refinement, can possibly improve the precision of a query, is definitely marked for refinement by our slicing approach. In other words, the approach will not fail to answer a query satisfactorily if at all it can be answered satisfactorily by refining some set of allocation sites.
  Note that in general our approach does *not* guarantee that only allocation sites that can possibly impact the precision of a query will be refined. In other words, our approach identifies an over-approximation of the allocation sites that need to be refined.

The rest of this paper is structured as follows. Section 2 provides key background that is necessary for us to present our approach. We describe our approach in Section 3. This section also discusses certain key properties of our approach, and also contains a proof of completeness. Sections 4 and 5 describe our implementation, and our empirical evaluation, respectively. Section 6 discusses related work. Finally, Section 7 concludes the paper.

2 BACKGROUND

In this section we give a brief introduction to the object-sensitivity points-to analysis [Milanova et al. 2005], on which our approach is based. We give the points-to analysis rules for this analysis in Figure 3. Our notation and formulation of these rules differs from that in the original paper mentioned above, in order to make it easier for us to subsequently describe our backward traversal rules (in Section 3). However, despite the modified formulation, the analysis essentially remains the same as in the original paper, and infers the exact same facts conceptually. In the rules, we use the symbol $\mathcal{S}$ to denote the set of all statements in the given program P, and the symbol $\mathcal{F}$ to denote the set of points-to facts that is being currently inferred. Recall from the introduction that we had given to this analysis in Section 1.2, that in this analysis symbolic objects serve as contexts. Thus, if we have a fact of the form $(c, \text{v}) \rightarrow o$, it means the following: (a) c and o are symbolic objects, (b) when the method m^j within which v is declared is invoked with c as the receiver object, then v may point to o. There is also another kind of fact used in the analysis, namely, of the form $o_1.f \rightarrow o_2$.

Rules M-Asgn, M-Get, and M-Put in Figure 3 are self-evident. We now focus on the rule M-New. Say the allocation statement "v = new" with label s_q is currently being processed (e.g., see labels s_1, s_2, etc., in the program in Figure 1). Say this statement is in a method m^j, and let c be a context under which this method is invoked. That is, c is one of the receiver objects for this method (see the

fact '(c, this_m^j) $\rightarrow c$' in the antecedent of the rule). This rule allots a symbolic object to represent objects allocated at this site under context c, by appending the site ID q after c. This is handled by the utility function *mkName*, which is as defined below.

$$\begin{aligned} mkName(o_{ij\ldots p}, q, k_q) &= o_{ij\ldots pq}, \text{ if } |ij\ldots p| < k_q \\ &= o_{j\ldots pq}, \text{ otherwise} \end{aligned}$$

Note, k_q is a user parameter, which indicates the maximum allowed name-length at site s_q (in the baseline '$k = 1$ analysis', this bound is 1 for all allocation sites). Therefore, the function *mkName* drops the leftmost side ID from the concatenated name if the concatenated name has length more than k_q.

The logic above is illustrated in Row 4 in Figure 2, which shows the points-to fact for variable cb inferred due to the allocation site s_9. The invocation context of 'bar' is o_4 in (see the fact '(o_4, this_bar) $\rightarrow o_4$' in Row 2, Column (1), which is due to Line 16 in the program). In Row 4, Column (3), k_9 is 2, resulting in the concatenated name o_{49}. Whereas, in Row 4, Column (1), k_9 is 1, which causes o_{49} to become just o_9.

We now focus on the rule M-Call. Say the invocation statement "v = w.m(z)" occurs in some method under invocation context c. The object c_1 that is pointed to by the base variable "w" under context c is first recovered. Thus, c_1 is the receiver object of this invocation. $dispatch(c_1, m)$ returns the method m^j in the program that the invocation mentioned above dispatches to when w points to c_1. Say 'p' is the formal parameter of m^j (for simplicity of presentation we assume that each method has a single formal parameter). The rule now infers that under the receiver context c_1, the variable this_m^j points to c_1 itself. The rule also infers that under the same context c_1, the formal parameter 'p' points to the same object that the actual parameter 'z' points to, namely, o_1.

As an illustration of the rule M-Call, consider Rows 4-6 in Column (3) in Figure 2. Also, in conjunction, consider the call to 'put' in Line 31 in the program (see Figure 1). The base variable at the call-site, namely, cb, points to o_{49} (see Row 4, Column (3) in Figure 2). From the last digit '9' in the name of this object, the analysis infers that this object was allocated at site s_9. Therefore, its type is inferred to be 'Contain'. Therefore, $dispatch(o_{49}, \text{put})$ returns the 'put' method in class 'Contain'. The actual parameter in the call site, namely, vb, points to o_6 (see Fact $\langle 4 \rangle$ in Row 3, Column (1)). Therefore, the rule infers the two facts shown in Rows 5-6, Column (3).

The Rule M-Ret is analogous to the Rule M-Call, and deals with the scenario of a function returning an object to a caller. ret_m^j is a placeholder, and denotes the variable in method m^j that holds the return value from the method. Notice the context sensitivity in this rule: if the return variable ret_m^j in method m^j points to an object o_2 under a context c_1, then the lhs v at an invocation statement is made to point to this object o_2 only if the receiver object at that invocation statement is also c_1.

Note, all "static" methods, including 'main' are always analyzed under an empty (or 'ϵ') context. Also, if 'g' is a global variable, its points-to facts are always of the form '(ϵ, g) $\rightarrow o$'. In the interest of brevity we have omitted these details from Figure 3.

3 OUR APPROACH

Our approach for refinement in object sensitivity is summarized in Figure 4. In Line 1, for all allocation sites i, the name-length bound k_i is initialized to 1. Line 2 invokes object sensitivity analysis, which returns a set of points-to facts $\mathcal{F}$.

The approach is given a set of queries Q as input. The queries need to be such that:

- They can be answered directly using points-to facts.
- There is a notion of a satisfactory answer for every query.

Procedure: Refine object sensitivity

Require: Program P. Set of queries Q.

1: Initialize $k_i = 1$, for alloc sites i.
2: Analyze P using object-sensitivity. Let $\mathcal{F}$ be the resulting points-to facts.
3: Answer the queries in Q using the facts $\mathcal{F}$.
4: **while** time budget not exhausted and some queries in Q not yet answered satisfactorily **do**
5: Identify the bad facts corresponding to the queries that are not yet answered satisfactorily. Let $\mathcal{F}_b$ be the set of all such bad facts.
6: Apply rules in Figure 5 on $\mathcal{F}$, to obtain the causality relationship '$\rightsquigarrow$' on the facts in $\mathcal{F}$.
7: **for all** facts of the form f_1 = '$((c, \text{v}) \rightarrow o)$' in $\mathcal{F}$ such that there exists some allocation statement " s_q: v = new", and o is an object allocated at s_q, and there exists some fact f_b in $\mathcal{F}_b$ such that $f_1 \rightsquigarrow^* f_b$ **do**
8: Increase the value of k_q by some non-negative number (following some policy).
9: **end for**
10: Analyze P using object-sensitivity. Let $\mathcal{F}$ be the resulting points-to facts.
11: Answer the queries in Q using the facts $\mathcal{F}$.
12: **end while**

Fig. 4. Our refinement approach

- For any query for which the answer is unsatisfactory, the set of "bad" facts that are the cause of the unsatisfactory answer can be identified.

How to perform the last step above is not a feature of our approach per se, but is a client-specific feature (i.e., depends on what the queries are asking for). For instance, if the safety of a downcast '(B) t1' is being checked by a query, and if it is declared as potentially unsafe due to the points-to set of t1, then every fact of the form '$(c, \text{t1}) \rightarrow o_i$', where o_i's type is not B or a sub-type of B, is a bad fact for this query.

As another example, if a call-site 'v.m(z)' is being checked whether it dispatches to a unique method in the program (this is called the "monosite" analysis), then, all facts in which the variable v occurs are bad *provided* there are at least two different facts '$(c_i, \text{v}) \rightarrow o_i$' and '$(c_j, \text{v}) \rightarrow o_j$' such that o_i and o_j are of types such that these types have different implementations of the method 'm'. Note that even though all facts in which v occurs need to be considered as bad facts, the approach could still be beneficial, because in general even in this scenario only a subset of all allocation sites in the program would impact the bad facts by causality and would get assigned a higher value of k_q.

As a third example, consider the static data-race detection client. Here, one needs to check using points-to analysis whether a statement of the form "t = v.f" executing in one thread and a statement of the form "w.f = s" executing in another thread could refer to a common object. For this analysis, only points-to facts that involve *common* symbolic objects that *both* v and w point to need to serve as bad facts.

Line 3 in Figure 4 answers all the queries using the points-to facts $\mathcal{F}$. Line 5 in our approach collects together all bad facts pertaining to all unsatisfactory queries. Line 6 applies the rules in Figure 5 to infer a causality relation '$\rightsquigarrow$' among the facts in $\mathcal{F}$. There is a very close correspondence between these rules and the object-sensitivity analysis rules in Figure 3. Intuitively, whenever a set S of antecedent facts are used by a rule in Figure 3 to infer a consequent fact f, the corresponding rule in Figure 5 includes (f_1, f) in the relationship '$\rightsquigarrow$' for every fact f_1 in S. Hence, the rules in Figure 5 are self-explanatory.

$$\frac{\text{“v = w”} \in \mathcal{S},\ \{(c,\mathrm{w}) \rightarrow o,\ (c,\mathrm{v}) \rightarrow o\} \subseteq \mathcal{F}}{((c,\mathrm{w}) \rightarrow o) \rightsquigarrow ((c,\mathrm{v}) \rightarrow o)}\ [\text{C-Asgn}] \qquad \frac{\begin{array}{c}\text{“v = w.f”} \in \mathcal{S},\ ((c,\mathrm{w}) \rightarrow o_1) \in \mathcal{F} \\ \{(c,\mathrm{v}) \rightarrow o_2,\ o_1.f \rightarrow o_2\} \subseteq \mathcal{F}\end{array}}{\begin{array}{c}((c,\mathrm{w}) \rightarrow o_1) \rightsquigarrow ((c,\mathrm{v}) \rightarrow o_2) \\ (o_1.f \rightarrow o_2) \rightsquigarrow ((c,\mathrm{v}) \rightarrow o_2)\end{array}}\ [\text{C-Get}]$$

$$\frac{\begin{array}{c}\text{“v.f = w”} \in \mathcal{S},\ ((c,\mathrm{v}) \rightarrow o_1) \in \mathcal{F} \\ \{(c,\mathrm{w}) \rightarrow o_2,\ o_1.f \rightarrow o_2\} \subseteq \mathcal{F}\end{array}}{\begin{array}{c}((c,\mathrm{v}) \rightarrow o_1) \rightsquigarrow (o_1.f \rightarrow o_2) \\ ((c,\mathrm{w}) \rightarrow o_2) \rightsquigarrow (o_1.f \rightarrow o_2)\end{array}}\ [\text{C-Put}] \qquad \frac{\text{“}s_q\text{: v = new” in meth } m^j,\ ((c,\ \mathrm{this}_{m^j}) \rightarrow c) \in \mathcal{F}}{((c,\ \mathrm{this}_{m^j}) \rightarrow c) \rightsquigarrow ((c,\mathrm{v}) \rightarrow mkName(c,q,k_q))}\ [\text{C-New}]$$

$$\frac{\begin{array}{c}\text{“v = w.m(z)”} \in \mathcal{S},\ \{(c,\mathrm{z}) \rightarrow o_1, (c,\mathrm{w}) \rightarrow c_1\ \} \subseteq \mathcal{F},\ m^j(\mathrm{p}) = dispatch(c_1,m), \\ \{(c_1,\mathrm{p}) \rightarrow o_1, (c_1,\mathrm{this}_{m^j}) \rightarrow c_1\ \} \subseteq \mathcal{F}\end{array}}{((c,\mathrm{w}) \rightarrow c_1) \rightsquigarrow \{(c_1,\mathrm{p}) \rightarrow o_1,\ (c_1,\mathrm{this}_{m^j}) \rightarrow c_1\}, ((c,\mathrm{z}) \rightarrow o_1) \rightsquigarrow ((c_1,\mathrm{p}) \rightarrow o_1)}\ [\text{C-Call}]$$

$$\frac{\text{“v = w.m(z)”} \in \mathcal{S},\ \{(c,\mathrm{w}) \rightarrow c_1, (c,\mathrm{v}) \rightarrow o_2\} \subseteq \mathcal{F},\ m^j(\mathrm{p}) = dispatch(c_1,m),\ ((c_1,\mathrm{ret\_}m^j) \rightarrow o_2) \in \mathcal{F}}{((c_1,\mathrm{ret\_}m^j) \rightarrow o_2) \rightsquigarrow ((c,\mathrm{v}) \rightarrow o_2), ((c,\mathrm{w}) \rightarrow c_1) \rightsquigarrow ((c,\mathrm{v}) \rightarrow o_2)}\ [\text{C-Ret}]$$

Fig. 5. Rules for the causality relationship

Reverting back to the summary of the approach in Figure 4, Lines 7 and 8 identify allocation sites that result in facts from which bad facts are reachable along the causality relationship. This reachability computation is the same as the backward traversal that we had introduced in Section 1.3. Basically, the allocation sites identified here are the ones that need to be refined. The k_q value for each such site s_q is incremented by a certain value (the policy to determine this increment can be provided to our approach). Lines 10 and 11 perform object-sensitivity analysis again, using the updated k_q values, and answer the queries again. The loop in lines 4-12 is repeated as long as the user is willing to wait and as long as queries exist that are yet to be answered satisfactorily. Note, there is no guarantee that all queries will be answered satisfactorily eventually.

3.1 Illustration Of Our Approach

We now illustrate our overall approach. Let us consider the downcast in Line 19 in the running example (see Figure 1) as the query. From the initial “k=1” object sensitivity analysis (in Line 2 of Figure 4), the facts that are produced regarding the downcasted variable t1 are t1 $\rightarrow o_6$, t1 $\rightarrow o_5$, and t1 $\rightarrow o_8$. Of these, the latter two facts are “harmless”, since sites s_5 and s_8 allocate objects of type B. t1 $\rightarrow o_6$ is the only bad fact, and is hence the only element of $\mathcal{F}_b$ (see Line 5 in Figure 4).

Line 6 of the approach is now applied. The causality relationship that is inferred in this line is basically represented in the first two columns in Figure 2. The format of this figure is that in each row, for the fact appearing in Column (1), its immediate causality predecessors are shown in Column (2) of the same row. For instance, Row 8 in Figure 2 represents the following pairs in the causality relationship ‘$\rightsquigarrow$’: $\langle 1 \rangle \rightsquigarrow \langle 9 \rangle$ and $\langle 5 \rangle \rightsquigarrow \langle 9 \rangle$. These pairs are inferred by Rule C-Ret in Figure 5, triggered by the ‘return’ statement in Line 32 in the program.

From the format mentioned above about Figure 2, it follows that every fact that appears in Column (2) in Figure 2 is a direct or transitive predecessor, via the causality relationship, of the bad fact t1 $\rightarrow o_6$ (this bad fact appears in Row 12, Column (1), in Figure 2).

Moving on to the loop in lines 7-9 in Figure 4, the allocation site s_9 satisfies the condition in Line 7; this is due to the fact $\langle 5 \rangle$ in Figure 2, which appears in multiple rows in Column (2) of the figure, and which is a transitive predecessor of the bad fact t1 $\rightarrow o_6$. Sites s_4 and s_6 also satisfy

the condition in Line 7; this happens due to facts ⟨1⟩ and ⟨2⟩, respectively, in Row 1, Column 1 of Figure 2. Other allocation sites in the program, and in particular, sites s_7 and s_8, are not causality predecessors of the bad fact. Therefore, k_4, k_6, and k_9 get incremented; say all these get incremented to 2 (from 1). The next round of the object sensitivity analysis, which happens in Line 10 in Figure 4, results in more refined points-to facts. Some of these facts were shown in Figure 2, Column (3). This refinement causes the variables c1 and c2 to now point to two distinct objects, namely, o_{39} and o_{49} (see Column (3) in Figure 2). Earlier, they used to point to the same object, namely, o_9 (see Column (1) in the same figure). As earlier discussed in Section 1.3, this "non-aliasing" of c1 and c2 suffices to eliminate the bad fact t1 $\rightarrow o_6$ from this (second) round of object sensitivity analysis. At this point, the downcast query has been answered satisfactorily (since the bad fact has disappeared). Therefore, on this example, the approach terminates after just the first iteration of the loop in lines 4-12 in Figure 4.

3.2 Key Properties Of Our Approach, and Relation To Program Slicing

In this part we discuss certain key aspects of the causality relationship '$\rightsquigarrow$' that we infer on the points-to facts. We argue that traversal of the points-to facts using this relationship amounts to a form of program slicing. We also discuss the key properties of this slicing approach, and novelty of this approach over other forms of program slicing in the literature.

3.2.1 Relation To Program Slicing. The *backward slicing* problem [Weiser 1981], which has been well-studied in the literature, is to identify a subset of statements in the program that are relevant to producing values at a specified memory location of interest and/or at a specified program location of interest. The specification mentioned above is typically called a "slicing criterion". There also exists the analogous problem of forward slicing.

Our fact-traversal approach also effectively acts like a slicing approach. This is because any pair $f1 \rightsquigarrow f2$ in the causality relationship is inferred by some rule in Figure 5, and any application of any rule is with reference to a specific statement in the program (as mentioned in the antecedent of the rule). Therefore, any traversal of the causality relationship also effectively traverses the program statements that are associated with the causality pairs.

In our setting, the slicing criterion is a points-to fact, while the backward traversal from this fact identifies statements that are responsible for the generation of this fact. In other words, our traversal basically computes a data-dependence slice [Ferrante et al. 1987] of the given program, using a given points-to fact as a criterion, ignoring all reads and writes of non-pointer memory locations. Viewed another way, in a hypothetical program that makes use of only pointer values, our approach would compute a complete data-dependence slice.

Our program slicing approach is a generalization of the "def-use analysis" presented by Milanova et al. [Milanova et al. 2005]. Their approach determines which definitions in "put" statements reach which uses "get" in statements. Our approach generalizes to all flows of data, including ones that involve local variables, parameters, return values, etc. The precision of our approach is identical to that of their approach when only writes to and reads from object fields are considered.

3.2.2 Relation to conditioned program slicing. In our approach, the slicing criterion need not be just a variable, but is a points-to fact in general. That is, the criterion can be of the form '$(c, \mathrm{v}) \rightarrow o$' or of the form '$o_1.f \rightarrow o_2$'. In the first case, the criterion indicates that we are only interested in statements that cause the variable v to point to object o under a context c. Therefore, statements that cause the variable v to point to other objects, or cause v to point to o but only in contexts other than c, will not be included in the slice. Similarly, the second form of criterion mentioned above identifies statements that cause field 'f' of object o_1 to point to o_2. For this reason, at a high-level

our approach can be considered as a form of *conditioned slicing* [Canfora et al. 1998; Field et al. 1995; Harman et al. 2001].

To the best of our knowledge, our slicing approach is the first practical slicing approach to employ the notion of pointing to a specific symbolic object as a slicing criterion. The previous conditioned approaches mentioned above were typically concerned about conditions on primitive values, and hence used expensive techniques like symbolic execution to compute the slices.

In our application, which is refinement of allocation sites for object sensitivity, slicing back from specific facts can prevent allocation sites that cannot possibly help with the answering a query satisfactorily from being refined. In our running example, if one were to simply slice back from the variable t1 at Line 19 in the program (which is the downcast site), then *all* allocation sites in the program will be reached (and hence, refined). However, if the bad fact 't1 $\rightarrow o_6$' alone is used as the slicing criterion, then allocation site s_7 is not reached in the backward traversal. We had already discussed in Section 1.4 how refinement of site s_7 would increase the cost of the analysis without having any effect on the solution to the given downcast-safety query.

3.2.3 Precision and context-sensitivity of our slicing approach. In our slicing approach, a fact f_2 would be reachable from a fact f_1 during a forward traversal along the causality relationship iff f_1 is a direct or transitive ancestor of f_2 according to the antecedent-consequent relationship that arises during the production of the points-to facts by the object sensitivity analysis (whose rules are given in Figure 3). In other words, the context sensitivity of our slicing approach is based directly on object sensitivity.

Context sensitivity has long been recognized to be important for precision [Horwitz et al. 1990]. The same is true in our application, where we identify allocation sites to refine. For instance, on our running example, if our approach were not context sensitive, then even if we traversed back only from the bad fact 't1 $\rightarrow o_6$' along the causality relationship, we would still reach site s_7 and refine it. In the interest of brevity we omit a detailed discussion of this phenomenon.

3.2.4 Contrast with SDG based slicing. Our slicing approach can be thought of as employing object sensitivity to attain context sensitivity, as discussed in Section 3.2.3. In contrast, existing tools and research approaches that we are aware of that provide support for program slicing [Chen and Xu 2001; Hammer and Snelting 2004; Liang and Harrold 1998; Sridharan et al. 2007; Wala [n. d.]] are based on the traditional context-sensitive slicing approach that uses System Dependence Graphs (SDGs) [Horwitz et al. 1990]. SDGs are an inter-procedural extension of program dependence graphs (PDGs). SDG slicing is in principle *fully* context sensitive. Consider the partial example program shown in Figure 6. The call to 'fun1' basically results in the value in v4 getting copied into v5.m. Similarly, the call to 'fun2' results in the value in v6 getting copied into v7.m. SDG slicing has the potential to determine these flows precisely[1].

SDG slicing is based on building PDGs procedures for individual procedures. The PDG for procedure fun1 is shown to the right in Figure 6 (the PDG for fun2 would be identical in structure). The ovals represent variables or objects, the dashed arrows represent points-to edges, while the two curved arrows indicate value flows that occur within the procedure. Note that the PDG of fun1 needs its own copies of the heap objects o_1, o_2, and o_3 (we have labeled these copies o_{11}, o_{12}, and o_{13}). Similarly, fun2's PDG (not shown in the figure), would need its own copies of these same three objects. In short, the PDG of each procedure contains copies of all symbolic objects that are potentially referred to in the procedure.

It has been observed in the literature that due to the reason mentioned above, SDG-based slicing with data flows through heap objects tracked context-sensitively does not terminate successfully on

[1]Provided the writes to the object fields in the functions fun1 and fun2 are somehow treated as strong updates.

```
1 void fun1(b, c) {
2   b.f.g.h = this;
3   c.m = b.f.g.h;
4 }
5 // fun2 is identical to foo1
6
7 main() {
8   // v1 → o1, o1.f → o2, o2.g → o3
9   // v4 → o4, v5 → o5, v6 → o6, v7 → o7
10  v4.fun1(v1, v5);
11  v6.fun2(v1, v7);
12 }
13
```

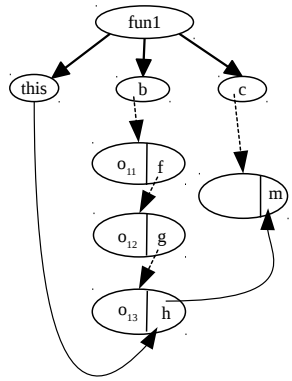

Fig. 6. SDG slicing

large programs [Sridharan et al. 2007]. Our own initial experiments with the Wala tool confirmed the same. It were these observations that motivated us to come up with the alternative "object sensitive" approach for context sensitive slicing that we actually propose in this work. On paper, our approach is less precise than the SDG approach. In the example in Figure 6, our approach would determine that v5.m and v7.m could both get their values from both v4 and v6. However, in practice, SDG-based slicing does not terminate at all on large benchmarks; whereas our approach terminates on large programs, and provides good precision for many code idioms, such as the ones that are used in our running example.

3.3 Proof Of Completeness

Our approach is *complete*, in the sense that every allocation site that can possibly help answer any of the given queries satisfactorily will necessarily be reached and marked for refinement during the backward traversal. We provide an argument for this below.

3.3.1 Basic Definitions. An "allocation-site to length" map K is a function that maps each allocation site q to a length bound k_q. That is, $K(q) = k_q$, for any allocation site q. K_1 denotes the "initial" map, which maps all allocation sites to 1. A *step* in the object-sensitivity analysis is an application of a rule in Figure 3, which uses some facts and produces some facts. A *derivation* that is "based on" a given map K is a sequence of steps such (a) any antecedent fact used by any step is produced in an earlier step, and (b) any step that processes any allocation site q uses $k_q = K(q)$ as the name-length bound.

3.3.2 When Is an Allocation-Site s_c To Be Considered Relevant To Answering a Query Satisfactorily? It is important to formalize this notion in order to eventually prove completeness. We claim that an allocation-site s_c is relevant to answering a query satisfactorily if there exists a "bad" fact f'_b corresponding to this query, and there exist maps K_2 and K_3 such that:

- $K_2(q) \geq K_1(q)$, for all allocation sites q.
- $K_3(s_c) > K_2(s_c)$, and $K_3(q) = K_2(q)$ for all allocation sites q other than s_c.
- $f'_b \in \mathcal{F}_1$, $f'_b \in \mathcal{F}_2$, and $f'_b \notin \mathcal{F}_3$, where each $\mathcal{F}_i$ is the set of all points-to facts derived by the object-sensitivity analysis by using the map K_i.

Basically, the points above imply that (i) s_c, possibly in conjunction with some other allocation sites, when refined, can rule out the bad fact f'_b, (ii) those other facts when refined alone without s_c being refined alongside do not rule out f'_b.

R_1	R_2
Step 1	Step 1
...	...
Step j: processes site s_c, produces f_c	Step j: processes site s_c, produces f_c'
...	...
Last step: produces bad fact f_b	Last step: produces bad fact f_b'

Fig. 7. Corresponding derivations

3.3.3 Proof. If an allocation s_c exists that satisfies the property above, then (1) there must exist a derivation R_2 based on map K_2 that finally infers f_b'. Wolog, we assume that this derivation has no "useless" steps. That is, every step (except the last step) produces output facts that are used by some subsequent step.

(2) Secondly, this derivation must process site s_c in some step j, and must produce a fact of the form $f_c' = ((c', \mathrm{v}) \rightarrow o')$ in that step, where v is the variable at the lhs of the allocation site s_c and o' is an object allocated at this site. The reason being, if this was not the case, then R_2 would also be a valid derivation based on K_3, which contradicts the assumption that $f_b' \notin \mathcal{F}_3$.

(3) There exists a derivation R_1 based on K_1 such that: (a) R_1 has the same number of steps as R_2. (b) R_1 processes the same sequence of statements as R_2. (c) in each step i, each fact f_1 that is used in step i of R_1 is a "suffix" of the corresponding fact f_1' that is used in step i of R_2, and each fact f_2 that is produced in step i of R_1 is a "suffix" of the corresponding fact f_2' that is produced in step i of R_2.

A fact f_1 is said to be a "suffix" of another fact f_1' if both facts refer to the same variable name (or field name), and the name of each object mentioned in f_1 is a suffix of the name of the corresponding object mentioned in f_1'.

In this setting, we say that R_1 *corresponds to* R_2.

Due to space constraints, we are unable to argue why such a derivation R_1 must exist. The argument is conceptually straightforward, but needs a detailed case-by-case analysis of each inference rule of the object-sensitivity analysis. This result is available as a separate proof on the home page of the second author of this paper.

As an illustration of the correspondence of a derivation to a more refined derivation, the first 10 rows in Column (3) in Figure 2 can be taken as (a prefix of) the sequence of steps in derivation R_2, while the corresponding rows in Column (1) are the corresponding steps in the derivation R_1. As further illustration, fact '$(o_4, \mathrm{cb}) \rightarrow o_9$' corresponds to fact '$(o_4, \mathrm{cb}) \rightarrow o_{49}$' in the corresponding steps in Row 4 of the table, with fact '$(o_4, \mathrm{cb}) \rightarrow o_9$' being a suffix of fact '$(o_4, \mathrm{cb}) \rightarrow o_{49}$'.

Recall that run R_2 infers the bad fact f_b' in its final step. Let f_b be the corresponding last fact inferred by R_1. Let f_c be the fact corresponding to f_c', produced in step j in R_1. This entire situation is represented schematically in Figure 7. In this figure, every step under the R_1 column corresponds to the same step under R_2.

We make the following assumption about any client query: if any fact f_b' is a bad fact for the query, then any fact that is a suffix of f_b' is also a bad fact for the query. This assumption is naturally satisfied by commonly known clients. For instance, if '$(o_{ij}, \mathrm{t1}) \rightarrow o_{kl}$' is a bad fact for a hypothetical downcast safety query, then '$(o_j, \mathrm{t1}) \rightarrow o_l$' would be a bad fact, too. This is because (a) o_l is allocated at the same site as o_{kl}, and hence has the same type, and (b) a downcast site is potentially unsafe if in *any* context the downcasted variable points to an object of the wrong type.

Therefore, f_b is also a bad fact. Since s_c is processed in step j in R_2, by correspondence s_c is processed in step j by R_1 also. Since R_1 corresponds to R_2, R_1 cannot have any useless steps, either. That is, the fact f_c produced in step j in R_1 is a direct or transitive ancestor of the final fact f_b

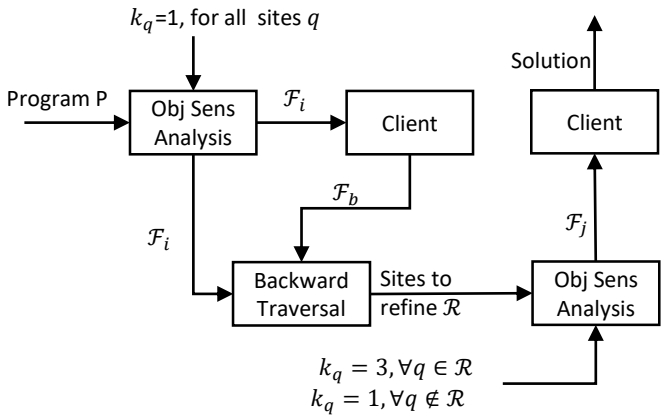

Fig. 8. Tool architecture

according to the antecedent-consequent relationship that arises due to the rules of the object-sensitivity analysis (Figure 3) during the production of the points-to facts by the steps of R_1. Now, by the argument at the beginning of Section 3 (before Section 3.1), f_c must be a direct or transitive predecessor of f_b as per the causality relationship '$\rightsquigarrow$' on the facts generated by R_1. Therefore, f_c would be reached in a backward traversal from f_b along the causality relationship on the facts produced by R_1. Since f_c corresponds to f_c', and since f_c' is of the form $((c', \mathrm{v}) \rightarrow o')$, where o' is an object allocated at s_c, f_c must be of the form $((c, \mathrm{v}) \rightarrow o)$, where o is a suffix of o'. Therefore, o must also be allocated at site s_c. Therefore, f_c (with s_c) satisfies the condition required to be satisfied by f_1 (with s_q) mentioned in Line 7 of our approach (see Figure 4). Therefore, site s_c would be marked for refinement. This completes our argument.

4 IMPLEMENTATION

We have implemented our approach using the Petablox [Petablox [n. d.]] program analysis framework. Petablox in turn uses Soot [Soot [n. d.]] as a front-end, and uses "bddbddb" [bddbddb [n. d.]] as the backend Datalog engine.

4.1 Overall Tool Architecture

Our tool architecture is as shown in Figure 8. The overall workflow is achieved using Java code (as an extension of the 'JavaAnalysis' class defined in Petablox). The individual steps are either implemented in Datalog or in Java. The basic object sensitivity step (invoked two times in our tool) is already available in Petablox (in the file 'CtxtsAnalysis.java'). We modified this code to take as an additional input a relation that specifies the 'k_q' value to be used at each allocation site q. The initial round of object sensitivity analysis (which is the first occurrence of the rectangle labeled "Obj Sens Analysis") uses $k_q = 1$ at all allocation sites q. The output set of facts from this, namely, $\mathcal{F}_i$, is fed into the client analysis. We have implemented two different client analyses for our evaluation (discussed in further detail later in this section). A client analysis uses the set of points-to facts given to it, and emits a "satisfactory" or "unsatisfactory" result for each query. It also emits the set of all "bad" facts $\mathcal{F}_b$ corresponding to all queries that were not answered satisfactorily. We have implemented our client analyses using Datalog.

The set of all facts $\mathcal{F}_i$ as well as the set of bad facts $\mathcal{F}_b$ are fed as input to our backwards traversal algorithm (see the rectangle labeled "Backward Traversal" in Figure 8). We have implemented this

traversal in Datalog. Note that our tool does not actually create and persist the causality relationship '$\rightsquigarrow$'. Instead, our Datalog rules access the points-to facts $\mathcal{F}_i$, and starting from the facts that are also in $\mathcal{F}_b$, find the direct and transitive predecessors of these facts along the causality relation via a fix-point computation. In other words, effectively, in each step of this computation, a set of facts that are already in the growing fix-point set are taken, are considered as an antecedent of one of the rules in Figure 5, and the source fact(s) in the causality relationship that is present in the consequent of this rule are added to the growing fix-point set.

For instance, our Datalog rule to process an assignment statement of the form 'v = w' is as follows:

```
TRCD(c0,w,c1) :-  MobjVarAsgnInst(_,v,w), TRCD(c0,v,c1), CVC(c0,w,c1).
```

The relation TRCD is a part of the growing fix-point set that we compute. A tuple (c0,w,c1) being added to the relation 'TRCD' means that points-to fact '(c0, w) $\rightarrow$ c1' is reached in the backward traversal. 'MobjVarAsgnInst(_,v,w)' is a reference to a built-in Petablox relation, and evaluates to true if 'v = w' is an assignment statement in some method in the program. 'CVC(c0,w,c1)' is also a reference to a built-in relation, and evaluates to true if fact '(c0, w) $\rightarrow$ c1' is in $\mathcal{F}_i$. The rule above basically implements Rule C-Asgn in Figure 5.

From all the facts reached in this traversal, the ones caused by allocation statements are identified, and the corresponding allocation sites are placed in the output set $\mathcal{R}$ (which is the set of allocation sites to refine). The object sensitivity analysis is applied again now (see the second occurrence of the rectangle labeled "Obj Sens Analysis"). In this second round of analysis, we set k_q to 3 for all allocation sites in the set $\mathcal{R}$. Note, we have chosen the value 3 as a compromise, as higher values tended to increase the time requirement quite a lot. The resultant points-to facts from this round (i.e., $\mathcal{F}_j$) are fed to the same client analysis again. This time, the solutions to the queries from the client are taken as the final solutions.

By default, our implementation performs only iteration of the outer loop in our approach (see Figure 4). In other words, it performs only one round of refinement. We do have an optional component in our implementation which performs a second round of refinement, using $k_q = 5$ at allocation sites that are marked for refinement in this round of refinement (and using $k_q = 1$ at all other allocation sites).

Petablox allows certain parts of the application (or libraries) to be excluded from the scope of the analysis if high efficiency is required. Excluding parts of the code from analysis can introduce unsoundness. Hence, in our experiments we do not exclude any parts of the application or libraries from the scope of the analysis.

4.2 Modified Object Naming Scheme

We made another modification within the core object sensitivity analysis, which is orthogonal to our primary contribution of refinement. This is in cognizance of the fact that programs often go through deep layers of calls through library code. Since the names of objects are suffixes of their true contexts, this process ends up removing allocation site IDs in the "application layer" from the object's names. This ends up reducing precision in the application layer, while preserving precision in the library layer. Whereas, developers might prefer more precision in the application layer, where the code that they wrote themselves resides.

Therefore, we adopt a modified object naming scheme as a heuristic. Whenever multiple allocation sites from the library layer appear consecutively in an object's name, we retain only the last (i.e., deepest) site among these library sites. The skipped library site names then do not count towards the name-length of the object. For instance, say o_{abclmn} is a full context name. Say a, b, c are allocation sites in the application layer, and l, m, n are allocation sites in the library layer. Say the name-length

bound k_n at site n is 3. Instead of using the pure suffix o_{lmn}, we use the modified name o_{bcn}. Note how this modification increases the portion of the name that is devoted to application sites and reduces the portion of the name that is devoted to library-layer sites.

This heuristic is easily seen to be sound. Note, we always retain the deepest allocation site's name in any object's name, because this site indicates the allocation site where this object was allocated, which is a required information for most client analyses.

Our implementation segregates application allocation sites from library allocation sites using a set of patterns. The allocation sites that occur in packages whose names match java.*, sun.*, javax.*, com.sun.*, com.ibm.* or org.apache.harmony.* are considered library allocation sites, while all other allocation sites are considered application allocation sites.

It is notable that the same intuition has been earlier exploited in the context of k-CFA by Medicherla et al. [Medicherla and Komondoor 2015]. Also, this intuition is exploited in a more general way in the work of Tan et al. [Tan et al. 2016], which also introduces "holes" in object names to reduce unnecessary context information.

4.3 Client Analyses

The first client analysis that we implemented is checking safety of downcasts. This is a commonly used client in various research papers [Lhoták and Hendren 2006; Sridharan and Bodík 2006; Zhang et al. 2014].

The second client we implemented is one that identifies "immutable" allocation sites. These are allocation sites that produce objects such that after execution returns from the constructor, the "root" object created at the site as well as other objects reachable from the root object (which would have been linked to the root object within the constructor) are not mutated further anywhere in the program. Identifying immutable allocation sites has various applications [Haack et al. 2007; Marinov and O'Callahan 2003; Porat et al. 2000], such as assisting program understanding, and enabling code optimizations and refactorings.

To check the mutability of any allocation site s_q conservatively, we check if there is any "put" statement anywhere in the program (but not inside the constructor corresponding to the allocation site) of the form "v.f = w", and any points-to fact of the form $(c, \mathrm{v}) \rightarrow o$, where o is either an object allocated at site s_q, or is reachable from any (symbolic) object allocated at s_q via a sequence of one or more field dereferences as per the points-to facts. If such "put" statements exist, we conservatively declare the site s_q as being possibly mutable, and include all facts such as the one mentioned above into the set of bad facts $\mathcal{F}_b$.

5 EMPIRICAL RESULTS

For our evaluations, we considered 9 real-world large programs from the DaCapo 2006 benchmark suite [Blackburn et al. 2006]. We analyzed these programs using our approach, using both the downcast safety as well as the immutable site client analyses. For the downcast safety client, following previous researchers [Sridharan and Bodík 2006], we checked only downcast sites in the application layer. For the immutability client also we checked only the allocation sites in the application layer. In both cases, we distinguished the application layer from the library layer using the library package patterns that we had mentioned in Section 4.2.

To serve as a baseline, we also applied the plain object sensitivity analysis for each client, which uses a uniform value of k at all allocation sites. We tried different values for k, such as 1, 2, and 3.

All experiments were performed on desktop computers with Intel i7 processors, with 40 GB allotted to the VM for each analysis. We used a uniform time budget of 10 hours in all our experiments, and stopped any run of any analysis on any benchmark if it did not complete within that time.

Before we report our actual empirical results, we mention an empirical validation of the completeness of our slicing approach (see Section 3.3), which we performed. For this, we compared the precision of our approach with k_q set to 2 (instead of 3) at all sites that are marked for refinement by the slicing, with the precision of the baseline approach (which uses $k_q = 2$ at *all* allocation sites). For this validation, we disabled our modified object naming scheme (Section 4.2), because this scheme is not incorporated in the baseline approach and because we want an apples-to-apples comparison. We did the comparison mentioned above on 6 benchmarks, and observed that the number of downcast warnings from both approaches were identical in all six cases. This serves as an empirical validation that our slicing marks for refinement all allocation sites that could possibly impact the precision of the answer to the given query.

We structure the rest of this discussion in the form of four research questions (RQs).

5.1 RQ 1: What Is the Scalability and Precision Of Our Approach For Downcast Safety?

The results for the downcast safety analysis are summarized in Table 1. Column 1 shows the benchmark program on which the analysis was performed. Column 2 shows the total number of application-layer downcasts sites (these are the downcasts that were checked by the analyses). Column 3 shows the total number of allocation sites in the program. Column 4 shows the number of downcasts in the program that were proved as safe using a simple, inexpensive context-insensitive (CI) points-to analysis. Column 5 shows the time taken by the CI analysis. All timings we report are in 'H:MM' format, and are end-to-end timings, inclusive of all rounds and steps. Note, in columns 4, 7, and 9, 'R' means number of downcasts (R)esolved as safe.

Column 6 shows the highest value of k for which the baseline object sensitivity analysis terminated in the allocated time budget. Columns 7 and 8 show the number of downcasts that were resolved as safe and time taken, by this baseline approach, when the value in Column 6 is used uniformly as the value of k at all allocation sites.

Similarly, Columns 9 and 11 show the number of downcasts resolved as safe and time taken by our approach, respectively. Column 10 shows the number of allocation sites refined by our approach (i.e., given a value $k_q = 3$ in the second round of object sensitivity analysis). Note, for any benchmark for which our approach hit the 10-hour timeout, we record a 'DNT' (i.e., Did Not Terminate) in Column 11. In this scenario, we record in Column 9 the results produced by the first round of our approach (i.e., the round that uses $k_q = 1$ uniformly at all allocation sites). This round terminates quite quickly on all benchmarks.

Finally the precision *gain* of our approach over the base object sensitivity approach is shown in Column 12. Downcast sites that resolved as safe by CI are considered "easy", and are excluded from consideration while calculating gain. In other words, the gain in each row is calculated using the following formula, where the numbers mentioned are column numbers. Note that previous researchers [Zhang et al. 2014] have also excluded queries resolved by CI from their gain calculations.

$$gain = ((9) - (4))/((7) - (4))$$

We observed that with the baseline object sensitivity approach, as the value of k increases, the time taken increased substantially. Indeed, none of the benchmarks could be analyzed within the time budget of 10 hours with k_q set to 3 at all allocation sites. Since our approach uses $k_q = 3$ at selected allocation sites, it scales much better. In particular, on two benchmarks, namely, 'pmd' and 'sunflow', the baseline approach did not terminate within the time budget with even $k_q = 2$ at all sites, whereas our approach terminates. And on 5 other benchmarks, both approaches terminate, but our approach is 2 to 7 times faster.

Table 1. Precision and scalability for downcast safety client analysis

(1) program	(2) total casts	(3) # alloc sites	(4) CI R	(5) CI time	(6) base k	(7) base R	(8) base time	(9) our R	(10) our sites	(11) our time	(12) gain
antlr	187	8933	39	0:02	2	123	6:11	132	1977	1:31	1.11
bloat	1323	10053	222	0:02	2	452	9:48	291	3534	DNT	0.30
chart	665	13861	96	0:04	1	139	2:02	139	3599	DNT	1.00
pmd	925	11774	408	0:04	1	432	0:44	555	2702	9:24	6.12
xalan	49	12052	16	0:04	2	36	8:01	38	1695	1:04	1.10
avrora	337	11252	42	0:03	2	140	6:03	149	3777	2:47	1.09
sunflow	138	13642	43	0:04	1	63	1:05	101	2505	5:31	2.90
luindex	215	9025	99	0:02	2	141	4:34	155	1980	2:00	1.33
lusearch	237	8700	67	0:02	2	150	4:31	154	1963	1:32	1.05
										Geometric mean:	1.29

Our approach is significantly more precise than the baseline approach. On 7 of the benchmarks, namely, all the ones other than 'bloat' and 'chart', our approach was able to resolve more queries as safe than the object sensitivity analysis was able to using the highest feasible value of k within the budget of 10 hours. In the extreme case of 'pmd', the precision of our approach is more than 6 times that of the baseline approach. The geometric mean of the gain of our approach across all the benchmark programs is 1.29.

We now make a few extra observations about our approach and implementation. We measured the contribution to the total running time of our approach by the backward slicing component (i.e., the causality rules traversal). We found that this was close to two hours for the two benchmarks 'bloat' and 'chart' (where our approach exhausted its time budget), and was in the range of 25-56 minutes for the remaining 7 benchmarks.

We also measured the "gain" of our approach over a simple baseline that uses $k_q = 1$ at all allocation sites. This was just to see the effect of the higher value of $k_q = 3$ at some sites. The mean gain of our approach with this calculation works out to 2.91 (contrast this with the gain of 1.29 mentioned above).

Finally, we also experimented with two rounds of refinement (see Section 4.1) on the benchmarks on which the first round finished relatively quickly. These are the benchmarks antlr, xalan, luindex, and lusearch. On all these benchmarks, unfortunately, the second round of refinement marked no extra downcast sites as safe that the previous round did not mark as safe. Note, however, that this is not a definitive confirmation that higher values of k (above 3) are not helpful at all. With unlimited time budgets, and with values of k even higher than 5, it is possible that the approach could potentially provide higher precision on some of the benchmarks.

5.2 RQ 2: What Is the Scalability and Precision Of Our Approach For Immutability?

We conducted a similar evaluation for the immutable site client as we did for downcast safety. The results are summarized in Table 2. The columns have similar meanings as the similarly named columns in Table 1. Column 2 alone has a distinct meaning, which is the number of allocation sites that are in the application layer that are checked by the various analyses. Another thing to note is that the numbers under the 'R' (Resolved) columns in this table are the number of allocation sites (from among the sites referred to in Column 2) that are declared as immutable by the respective analyses.

Table 2. Precision and scalability for immutable site client analysis

(1) program	(2) # app alloc sites	(3) CI R	(4) CI time	(5) base k	(6) base R	(7) base time	(8) our R	(9) our sites	(10) our time	(11) gain
antlr	1462	340	0:13	1	824	0:35	937	4269	6:25	1.23
bloat	2552	478	0:20	1	1087	0:53	1087	5348	DNT	1.00
chart	3671	1999	0:32	1	1817	1:18	1817	6691	DNT	1.00
pmd	1809	432	0:26	1	691	1:48	691	5369	DNT	1.00
xalan	2356	2094	0:16	2	2188	7:13	2188	4036	2:32	1.00
avrora	3712	1335	0:18	1	1630	0:37	2375	6089	6:25	3.52
sunflow	1151	717	0:22	1	646	2:45	646	6546	DNT	1.00
luindex	1417	334	0:16	1	633	0:55	648	4725	6:24	1.05
lusearch	1125	290	0:14	2	549	8:39	553	4513	4:18	1.01
									Geometric mean:	1.19

Both analyses take more running time for this client than they do for the downcast client. This is primarily due to the increased number of queries to check (contrast the numbers in Column 2 in the two tables). The baseline approach scales to $k = 2$ on only two benchmarks, namely, 'xalan' and 'lusearch'. On these two benchmarks, the baseline analysis takes at least 2 times more than running time than our approach. On three benchmarks, namely, 'antlr', 'avrora', and 'luindex', the baseline approach hits the timeout with $k = 2$ at all allocation sites, whereas our approach completes both rounds. On four other benchmarks, the baseline approach does not scale to $k = 2$ and our approach hits the timeout. Therefore, to conclude, our approach was faster than the baseline approach on 5 benchmarks, while neither approach scaled to the four remaining benchmarks using values of k greater than 1.

Regarding precision, our approach gave better precision than the baseline approach on 4 benchmarks, and the same precision on 5 benchmarks. The geometric mean of the gain in precision of our approach over the baseline is 1.19. In the extreme case of 'avrora', the precision of our approach is 3.52 times the precision of the baseline approach. It is notable that the comparatively poorer performance of our approach on this client vis-a-vis its performance on the downcast safety client is due to the increased number of queries, which cause an increased number of allocation sites to be refined, which in turn causes the timeout to be hit for more benchmarks.

As with RQ 1, we now make a few extra observations. The contribution to the total running time of our approach by the backward slicing component (i.e., the causality rules traversal) was in the range of 20-45 minutes across all nine benchmarks. The "gain" of our approach over the simple baseline that uses $k_q = 1$ at all allocation sites was 1.22 (contrast this with the gain of 1.19 mentioned above). Finally, we tried two rounds of refinement (see Section 4.1) on the two benchmarks on which the first round finished relatively quickly, namely, xalan and lusearch. On both these benchmarks, unfortunately, the second round of refinement marked no extra allocation sites as immutable that the previous round did not mark as safe.

5.3 RQ 3: What Is the Contribution Of Our Modified-Naming Scheme, Described in Section 4.2, To the Effectiveness Of Our Approach?

The main contribution of this paper is a backward slicing technique to determine important allocation sites where extra precision is required to rule out a bad fact. The modified naming scheme that we use (which we abbreviate as "MNS"), described in Section 4.2, is another important

component in our implementation. In this RQ we evaluate the importance and interplay of these two features.

Our first attempt to answer this question was to incorporate MNS into the baseline object sensitivity approach also, and run it with $k = 2$ and with MNS at all allocation sites. The idea was that if the thus-enhanced baseline approach performed as well as our approach, then selective refinement using slicing would be shown to be not very useful. However, we discovered that this enhanced baseline approach did not terminate on even the smallest of the Dacapo benchmarks, even after 24 hours of running. We surmise that the reason for this is that MNS implicitly has a similar effect as using a higher value of k, while actually using a lower value of k. In other words, using MNS with $k = 2$ at all allocation sites may have a similar effect as using $k = 3$ at all allocation sites. An exponentially greater number of contexts is hence generated, resulting in non-scalability.

We then tried another experiment, which was to run a variant of our approach, which incorporates MNS, but using $k = 2$ (instead of $k = 3$) at allocation sites that are reached in the slice. $k = 1$ was used at all other sites. We tried this variant on 6 benchmarks for the downcast client, and on 4 benchmarks for the immutability client. This variant of our approach obviously scaled well (as it uses a lower of k at refined allocation sites than the main variant of our approach). However, the surprising finding was that on 7 of the 10 analysis runs mentioned above, this variant generated the exact same number of warnings as our main variant, which uses $k = 3$ at refined allocation sites (along with MNS). And on 5 of the 10 runs, the baseline analysis with $k = 2$ at all allocation sites terminated and generated more warnings than this variant.

The experiments above lead us to the following empirical conclusions:

- $k = 2$ at refined sites with MNS gave higher precision than $k = 2$ at all sites without MNS. In fact, its precision was close to that obtainable by using $k = 3$ at all refined sites. This implies that MNS has a similar effect as using a value of k that is effectively higher than the value actually used. This was a surprising but interesting conclusion for us.
- Our slicing based approach to determine allocation sites that are pertinent to the generation of the bad fact is *equally useful* in the context of applying MNS as it is in the context of explicitly using higher values of k. The slicing basically enables MNS overall to scale by allowing us to use $k = 1$ (and hence, not use MNS) at allocation sites that are not relevant to the bad facts. This is clearly evidenced by the fact that MNS does not scale when applied in conjunction with $k = 2$ at all allocation sites, but scales when applied in conjunction with $k = 2$ at allocation sites that are in the slice.

5.4 RQ 4: How Does Our Approach Compare With a Recently Proposed Approach [Zhang et al. 2014]?

This recently proposed approach is closely related to our approach. Their approach is also iterative, in the sense that in each iteration it adds allocation sites to the set of sites to refine. However, in each iteration they use a precise (but expensive) MaxSAT based approach to identify a few sites that are *definitely* relevant to the imprecision in solving the queries (in our approach this relevance is established only conservatively). They then increase the k_q values for these sites q to a relatively high value, namely, 10. Another difference from our approach is that their initial analysis is the CI analysis, whereas our initial analysis is $k = 1$ object sensitivity analysis.

We used as-is their implementation of their approach, which is included within Petablox [Petablox [n. d.]]. In fact, in addition to the structural similarity between our approach and theirs, the other reason why we chose to compare our approach with theirs was the opportunity to perform an apples-to-apples comparison using the same front-end, same underlying implementation of object sensitivity analysis, etc. We ran their approach with their default configuration settings, which

Table 3. Precision and scalability comparison with Zhang et al.'s analysis

(1) program	(2) total casts	(3) their R	(4) their time	(5) their # iterations	(6) our R
antlr	187	131	1:06	16	132
pmd	925	413	DNT	3	555
xalan	49	25	DNT	5	38
avrora	337	51	DNT	4	149
sunflow	138	51	2:57*	3	101
luindex	215	107	1:46*	4	155
lusearch	237	76	DNT	4	154
toba-s	62	13	2:27	8	13
javasrc	110	41	2:47	8	41

were shared with us by the Petablox developers via private communication. However, we did make one change to their default configuration. By default they exclude certain libraries from the scope of the analysis. We changed this configuration in their analysis in order to not exclude anything from the scope of analysis; this is to ensure a uniform comparison with the results from our approach. We had discussed in Section 4.1 the reason why it is undesirable to exclude libraries from the scope of the analysis.

For this experiment, we used only the downcast safety client. This is because their code already includes support for this client. Applying their analysis on a new client like immutability analysis would have required more work from us. This is because in their approach the queries are resolved *in conjunction* with the points-to analysis in each iteration; this necessitates addition of non-trivial code to their approach to deal with each separate client.

The results from their approach are summarized in Table 3. Column 2 in Table 3 shows the total number of downcasts checked. The numbers in this column match the numbers in Column 2 in Table 1. Note, the benchmarks 'bloat' and 'chart' are excluded from Table 3, because of some difficulties we faced in running their tool on these benchmarks.

Column 3 in Table 3 shows the number of downcasts resolved to "safe" by their approach, while Column 4 shows the time taken by their approach. Benchmarks on which their analysis did not terminate within the time budget are indicated using a 'DNT' in Column 4. Benchmarks on which their analysis terminated abnormally before hitting the timeout are indicated by an asterix ("*"). Column 5 shows the number of iterations completed successfully by their analysis before normal termination or abnormal termination or before the timeout was reached. A zero here means that only the initial CI round was completed. Note, the numbers in Column 3 indicate the number of queries resolved to "safe" in the last successful iteration before normal termination or abnormal termination or timeout. Column 6 shows the number of downcasts resolved as safe by our approach (these numbers are simply taken from the 'our R' column in Table 1). In addition to the seven Dacapo benchmarks, for this comparison we also included two smaller benchmarks which they refer to in their paper [Zhang et al. 2014]. These are toba-s and javasrc.

Note that on two of the seven Dacapo benchmarks, their approach terminates abnormally within the time budget. This was likely due to certain memory usage issues in the MaxSAT solver. Their implementation by default uses the "mifumax" MaxSAT solver, which appears to work well on small-to-medium sized instances. But on larger instances, since the problem is fundamentally computational challenging, mifumax does not scale.

With regard to precision, it can be noted that our approach gives equal or more precision than their approach on all the benchmarks. On the larger benchmarks, the main reason for this appears to be the non-scalability of MaxSAT solving to larger instances. On the smaller benchmarks, we believe that our approach is able to match their approach on precision because values of k as high as 10 are not required to obtain as much precision as is possible by increasing object-name lengths alone.

5.5 Limitations and Discussion

We are very encouraged by the precision and scalability of our approach. Within the given time budget of 10 hours, our approach exhibited better precision than the baseline object sensitivity analysis on all benchmarks other than 'bloat' for the downcast client, and on four benchmarks for the immutability client. The baseline approach outperformed our approach in only one instance – on 'bloat', for the downcast client. Our approach also equals or out-performs the recent MaxSAT-based refinement approach [Zhang et al. 2014] on all benchmarks.

This said, there exist certain limitations in our implementation and experimentation methodology. A clear limitation is due to the implementation of the Milanova approach that is available within the Petablox tool, which we use. This efficiency of this implementation is somewhat low. We surmise that its efficiency could be improved, which would benefit both our analysis as well as the baseline analysis. Another limitation is with regard to using multiple rounds of refinement. We have tried this to some extent, with no observed increase in precision, but could have tried it more systematically for all benchmarks.

The default implementation of the approach of Zhang et al., which we used in our comparative evaluation, uses precise MaxSAT solving. An open question is how their approach would perform using approximate solvers. We have not explored so far whether and how to make their implementation of their approach use approximate solvers.

6 RELATED WORK

There is a very large body of work that has been reported in the literature about points-to analysis in general. In this section we focus our discussion primarily on client-driven refinement approaches that are closely related to our approach. Towards the end of the section we also touch upon other approaches that refine abstractions, although not necessarily in the context of a given client.

The approach of Guyer et al. [Guyer and Lin 2003] is one of the early approaches that proposed client driven refinement of points-to analysis. A client analysis and an inexpensive variant of points-to analysis are initially performed side-by-side. Both analyses are assumed to be specified as data-flow analyses. During this joint analysis, pointers that happen to pollute the client analysis because they have potentially imprecise points-to sets are identified. This is done using a form of slicing. Polluting pointers identified in this way are then refined, either by tracking value flows into them flow sensitively, or by refining the depth of context sensitivity of procedures in which these pointers are set. The analysis is then repeated again, to see if better results are obtained. Some other differences between their approach and ours are that (a) the form of slicing they use appears to be context insensitive, which can be imprecise, (b) they target refinement of call-site sensitivity, and not of object sensitivity, and (c) they target C programs.

Sridharan et al. [Sridharan and Bodík 2006] propose a CFL (Context Free Language) reachability based approach to perform a points-to analysis jointly with the analysis required for a query. A CFL-annotated graph is used, wherein paths encode value flows. Initially extra edges are present in the graph, which speeden up the analysis but reduce precision. Precision is reduced by not insisting on balanced calls and returns, and not insisting when there is a path from a statement of the form "v.f = w" to a statement of the form "t = s.f" that v and s point to a common object. This is hence a

coarse abstraction. For queries that cannot be resolved satisfactorily using coarse abstractions, the imprecision-inducing edges are removed in a stagewise, iterative manner, to see if the precision improves. Our approach differs from this approach in being based quite directly on program slicing, and in targeting refinement of object sensitivity rather than call-site sensitivity.

The approach of Liang et al. [Liang and Naik 2011] is quite closely related to our approach. Their approach works on a Datalog analysis that jointly performs points-to analysis and answers the queries. In an initial round of analysis, a cheaper abstraction is used. The approach then traverses the Datalog derivation in the backward direction to identify all instances of abstraction in the analysis that are potentially contributing to the unsatisfactory resolution of the queries. These abstractions are then refined in subsequent rounds. The authors of this paper note that their traversal is analogous to slicing. However, their approach needs the points-to analysis to be implemented in Datalog. In fact, they add instrumentation rules to the Datalog analysis, and use the output from these instrumentation rules to traverse the primary derivation in the backward direction. In contrast, our approach does not assume that the points-to analysis is written in Datalog, and does not rely on instrumenting the analysis. Furthermore, in their empirical evaluation, Liang et al. use mostly smaller programs, and use only two of the programs from the Dacapo benchmark suite, namely, lusearch and avrora. They also do not report running times for their analysis.

Zhang et al. [Zhang et al. 2014] extend upon the work mentioned above. Instead of refining all abstractions that appear to contribute to imprecision, they refine abstraction instances iteratively. In each iteration they identify, using a MaxSAT solver, a small number of abstraction instances that necessarily induce imprecision across all possible derivations, and refine only these abstraction instances. In this way, they are able to refine more deeply certain abstraction instances that are definitive causes of imprecision. We presented in Section 5 a comparative empirical evaluation of their approach and our approach on a range of benchmarks.

All the approaches mentioned above have some common characteristics that our approach does not share. One, they intertwine the points-to analysis with the query solving. Secondly, they need certain forms of instrumentation to be applied into the points-to analysis itself. These features make for good conceptual elegance, and can increase precision at least for smaller programs. However, our approach would be simpler to implement, because we do not make any assumptions on how the points-to analysis is implemented, and do not need to instrument it to observe its steps in a fine-grained manner. Our backward traversal is a pure post-pass on the facts that result from points-to analysis.

For a given client analysis, the approach of Oh et al. [Oh et al. 2014] relies on a given, less precise but more efficient pre-analysis that is a sound abstraction of the main analysis. Their approach performs the pre-analysis with full context sensitivity, and uses the results of this analysis to identify calling contexts (and their suffixes) that may need to be distinguished to achieve good precision in the main analysis. The main analysis is then performed using appropriate refinement such that the identified contexts are distinguished, and other contexts are not necessarily distinguished.

We now briefly mention two recent approaches that perform selective refinement of abstractions, but not necessarily in the context of a client analysis. In "introspective analysis" [Smaragdakis et al. 2014], the idea is to perform an initial context insensitive analysis, and then use certain metrics to identify regions of the program where high context sensitivity would cause explosion in analysis time. A subsequent round of context sensitive analysis is then performed, with high context sensitivity employed at the remaining regions. The "still k-limiting" approach of Tan et al. [Tan et al. 2016] also uses a fast context-insensitive prepass. In this pass they identify portions of context strings that can be elided, without reducing the distinguishing power of the context strings. The resulting context strings (that could have "holes" in them) are used in a subsequent context sensitive analysis.

A paper by Reps [Reps 1995] is interesting to mention, even though it does not relate to points-to analysis or to refinement. This paper provides a Datalog analysis for inter-procedural slicing. However, heap objects are not addressed.

7 CONCLUSIONS AND FUTURE WORK

In this paper we presented a program slicing based approach for refinement in object sensitivity analysis. The approach is based on performing an initial low context-sensitivity analysis, and then identifying points-to facts that are potential contributors to imprecision in answering the given queries. The allocation sites that potentially cause these "bad" facts are then identified. These sites are refined to greater context depths, and the analysis re-applied. Our approach has similarities with certain previous approaches. Its main novelty, however, is in using a form of slicing on the facts themselves, to find the impacting allocation sites. We feel this approach is simpler than previous approaches, and potentially more scalable. We also feel that our slicing sub-routine could be of interest standalone, because previous slicing approaches are predominantly SDG-based, which does not appear to be a scalable technique. We have also proved the completeness of our refinement approach theoretically.

An evaluation of our approach empirically on a set of large benchmarks revealed that our approach scales readily to large benchmarks, while providing more precision than a baseline approach as well as a recently proposed refinement approach.

Several improvements to the implementation and the experimentation methodology used in this paper are possible; we had mentioned some of these in Section 5.5. Conceptual advances to the approach are also possible. Our approach still marks a lot of sites for refinement, which prevents very long object names to be used at refined sites. A staged approach for refinement could be tried, that somehow ranks the sites that have been identified in descending order of their likelihood that they will positively impact the query. Now, sites can be refined in rank order, either one by one, or in groups, using higher values of k_q. More "refined" notions of refinement itself could also be explored; for instance, instead of just increasing the name-length at an allocation site that is to be refined, perhaps some important contexts that need to be distinguished at that site could also be identified. It would be interesting future work to explore all these directions. An ambitious target for any future points-to analysis approach would be to be very precise, while being so efficient in terms of both running time and memory requirement that it could be used interactively within a development environment while a develop is writing new code or making changes to existing code.

REFERENCES

bddbddb [n. d.]. bddbddb: BDD-Based Deductive Database. http://bddbddb.sourceforge.net/. ([n. d.]).

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 *Proc. 21st Annual ACM SIGPLAN Conf. on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. 169–190.

Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. 1994. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 234–250.

G. Canfora, A. Cimitile, and A. De Lucia. 1998. Conditioned program slicing. *Information and Software Technology* 40, 11 (1998), 595–607.

Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. 1999. Relevant Context Inference. In *Proc. ACM Symposium on Principles of Programming Languages (POPL '99)*. ACM, 133–146.

Zhenqiang Chen and Baowen Xu. 2001. Slicing Object-oriented Java Programs. *SIGPLAN Not.* 36, 4 (April 2001), 33–40.

Doop [n. d.]. Doop program analysis framework. https://bitbucket.org/yanniss/doop. ([n. d.]).

Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. 1994. Context-sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, 242–256.

Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

John Field, G. Ramalingam, and Frank Tip. 1995. Parametric Program Slicing. In *Proc. Int. Symp. on Principles of Prog. Langs. (POPL)*. 379–392.

Samuel Z Guyer and Calvin Lin. 2003. Client-driven pointer analysis. In *International Static Analysis Symposium (SAS)*. Springer, 214–236.

Christian Haack, Erik Poll, Jan Schäfer, and Aleksy Schubert. 2007. Immutable objects for a Java-like language. In *European Symposium on Programming*. Springer, 347–362.

Christian Hammer and Gregor Snelting. 2004. An Improved Slicer for Java. In *Proc. 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '04)*. ACM, 17–22.

M. Harman, R. Hierons, C. Fox, S. Danicic, and J. Howroyd. 2001. Pre/post conditioned slicing. In *Proc. Int. Conf. on Software Maintenance (ICSM)*. 138–147.

Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 1 (1990), 26–60.

Vini Kanvar and Uday P Khedker. 2016. Heap abstractions for static analysis. *ACM Computing Surveys (CSUR)* 49, 2 (2016), 29.

Ondřej Lhoták and Laurie Hendren. 2006. Context-sensitive points-to analysis: is it worth it?. In *International Conference on Compiler Construction*. Springer, 47–64.

Donglin Liang and Mary Jean Harrold. 1998. Slicing objects using system dependence graphs. In *Proc. International Conference on Software Maintenance*. IEEE, 358–367.

Percy Liang and Mayur Naik. 2011. Scaling Abstraction Refinement via Pruning. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, 590–601.

Ravichandhran Madhavan, Ganesan Ramalingam, and Kapil Vaswani. 2012. Modular heap analysis for higher-order programs. In *International Static Analysis Symposium*. Springer, 370–387.

D. Marinov and R. O'Callahan. 2003. Object equality profiling. In *Proc. Conf. on Object-Oriented programing, Systems, languages, and applications*.

R. K. Medicherla and R. Komondoor. 2015. Precision vs. scalability: Context sensitive analysis with prefix approximation. In *2015 IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 281–290.

Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 14, 1 (2005), 1–41.

Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective Context-sensitivity Guided by Impact Pre-analysis. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 475–484.

Petablox [n. d.]. Petablox program analysis platform. https://github.com/petablox-project/petablox/wiki. ([n. d.]).

Sara Porat, Marina Biberstein, Larry Koved, and Bilha Mendelson. 2000. Automatic Detection of Immutable Fields in Java. In *Proc. of the 2000 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '00)*. IBM Press, 10–.

Thomas W Reps. 1995. Demand interprocedural program analysis using logic databases. In *Applications of Logic Databases*. Springer, 163–196.

Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Foundations and TrendsÂő in Programming Languages* 2, 1 (2015), 1–69.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-sensitivity. In *Proc. ACM Symposium on Principles of Programming Languages (POPL '11)*. ACM, 17–30.

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective Analysis: Context-sensitivity, Across the Board. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 485–495.

Soot [n. d.]. Soot: A framework for analyzing and transforming Java and Android applications. http://sable.github.io/soot. ([n. d.]).

Manu Sridharan and Rastislav Bodík. 2006. Refinement-based Context-sensitive Points-to Analysis for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, 387–400.

Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J Fink, and Eran Yahav. 2013. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Springer, 196–232.

Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. 2007. Thin slicing. In *PLDI '07: Proc. Conference on Programming Language Design and Implementation*. 112–122.

Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *International Static Analysis Symposium*. Springer, 489–510.

Wala [n. d.]. *T.J. Watson Libraries for Analysis (WALA).* http://wala.sf.net

Mark Weiser. 1981. Program slicing. In *Proc. International Conference On Software Engineering*. IEEE Press, 439–449.

John Whaley and Monica S Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proc. Conference on Programming Language Design and Implementation*. ACM, 131–144.

Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On Abstraction Refinement for Program Analyses in Datalog. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 239–248.